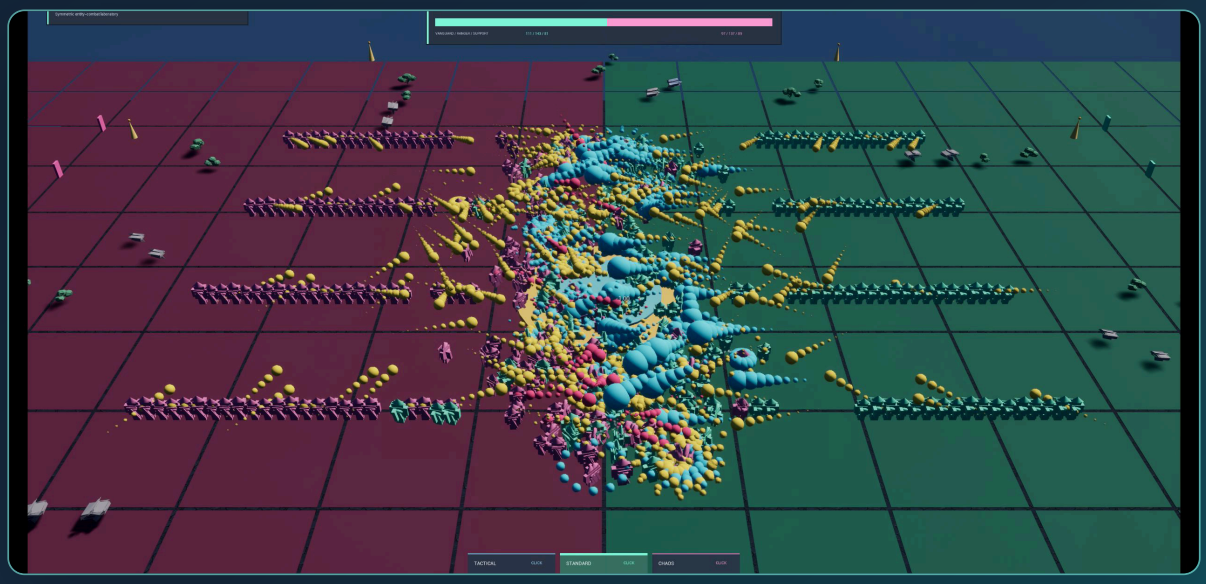
ASHENFORGE

Mass Forge

Turn Unreal Mass entities into gameplay-ready combatants.

Attributes, tags, effects, abilities, damage, targeting, and projectiles - from Blueprint or C++.



Complete manual - Version 0.2.0
September 24, 2026

Contents

Getting Started

1. Overview
2. Blueprint Quick Start
3. Blueprint Recipe Index
4. Entity Config Wizard
5. Gameplay Starter Kit
6. Starter Attributes
7. Known Issues and FAQ

Blueprint and C++ Workflows

1. Blueprint Authoring
2. Blueprint Async Actions
3. Actor and Entity Parity
4. C++ Gameplay Quick Start
5. C++ High-Volume Integration
6. StateTree and Behavior Integration
7. Deferred Commands
8. Transactions and Events

Gameplay Systems

1. Abilities
2. Ability Lifecycles
3. Ability Conditions
4. Ability Execution Plans
5. Target Selection
6. Targeting Policies
7. Damage
8. Combat Integration
9. Instant Effects
10. Persistent Effects
11. Effect Magnitude Calculations
12. Gameplay Tags
13. Projectiles
14. Regeneration
15. Derived Attributes

Editor and Debugging

1. Gameplay Asset Authoring
2. Schema Editor
3. Project Health
4. Entity Inspector
5. Gameplay Event History
6. Error and Result Codes

Examples and Presentation

1. Playable Showcases
2. How the Examples Are Authored
3. Character Animation and VAT Evidence

Architecture, Scale, and Persistence

1. Architecture
2. Entity Lifecycle Safety
3. Entity-State Persistence
4. Persistence and Migration
5. Time, Travel, and Save Contract
6. Deterministic Ordering
7. Memory and Capacity
8. Processor Performance Contract
9. High-Volume Effects
10. Persistent Scheduler
11. Regeneration LOD Policy
12. Performance Benchmarking
13. Reference Results

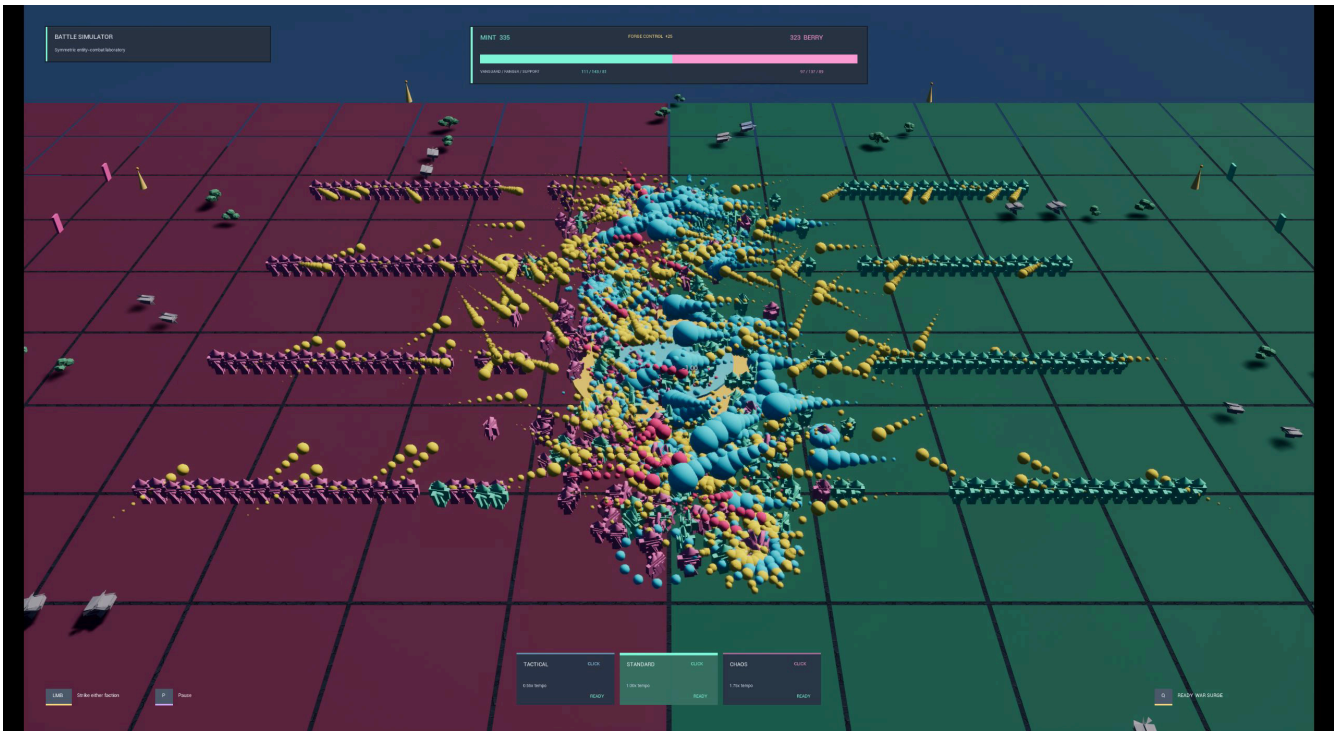
Release and Support

1. Engine Version Support
2. Versioning and Upgrades
3. Third-Party Content
4. Support Policy
5. Support Request Template
6. CI and Verification

GETTING STARTED

Add combat without replacing Unreal Mass

Unreal Mass handles data-oriented populations. Mass Forge gives those entities a shared combat vocabulary: Health, resources, effects, abilities, targeting, projectiles, death, saved state, and diagnostics. The entities and processors remain native Mass; the plugin adds the gameplay rules around them.



The included Battle Simulator is one inspectable consumer of the public API: the armies use Mass Forge attributes, targeting, damage, healing, effects, projectiles, and outcome state while the example owns its formations, pacing, presentation, and rules.

Spawn one entity from a Blueprint encounter, arrange thousands through a C++ population system, or process a bounded working set from a much larger allocation. Each path uses the same attributes, damage definitions, effects, abilities, targeting policies, and projectile rules.

Clear ownership, no hidden game framework

Mass Forge equips and operates	Your project remains free to define
Attributes, derived values, regeneration, tags, effects, abilities, damage, targeting, projectiles, death requests, state snapshots, and inspection	Spawn logic, pooling, wave composition, AI decisions, navigation, movement, factions, representation, animation, UI, progression, balance, victory, and defeat

The included examples are inspectable consumers of the plugin, not mandatory architecture. Copy their patterns, replace them, or connect Mass Forge to an existing simulation.

Choose your first path

- **Blueprint developers:** generate an editable Combat Ready package, spawn an entity, change Health, apply damage and healing, activate an ability, and launch an impact-driven projectile in the Blueprint Quick Start.
- **C++ developers:** call the same public gameplay APIs from ordinary classes, then adopt compiled bindings and processor-safe paths where scale requires them in the C++ Gameplay Quick Start.
- Designers should use Gameplay Asset Authoring, Project Health, and the Entity Inspector.
- Evaluators can play the five standalone examples, inspect the memory and capacity contracts, and reproduce the supported-engine evidence.

> **INFO - Initial-release scope**
>
> Mass Forge's supported initial release is single-player. Multiplayer research remains preserved for post-v1 development but is not part of the advertised workflow, public example catalogue, or support promise.

GETTING STARTED

Blueprint-only quick start

This tutorial takes a new project from installation to a working Mass Forge attribute, deterministic damage, healing, and an ability without C++. It also shows the production-safe queued form and the adapter used when a player remains an ordinary Actor.

The examples use `Vital.Health`, `Vital.MaxHealth`, and `Support.Mana` because the Entity Config wizard creates those starter entries. They are editable examples rather than required Mass Forge names.

> **TIP - Blueprint-first path**
>
> Everything in this learning slice is implemented with public Blueprint nodes and editable Data Assets. The plugin supplies safe Mass operations; your project still owns combat rules, input, UI, AI, representation, and game-specific progression.

Before rebuilding this tutorial, enable **Show Plugin Content**, open `/MassForge/MassForge/L_MF_BlueprintQuickStart`, and inspect its placed `BP_MF_BlueprintQuickStart__OPEN_ME` actor. The Blueprint at `/MassForge/MassForge/BlueprintExamples/Blueprints/BP_MF_BlueprintQuickStart` provides a numbered reference for the exact public nodes, distinct source/target handles, visible snapshot-driven projectile, 27 editable Data Assets, and four material instances used below (32 assets total). Copy it into project Content before making changes; treat the plugin-owned graph as a versioned reference.

Press Play in the shipped learning map to exercise the graph repeatedly:

- **Left Mouse Button** launches a visible Mass-owned projectile; its damage commits only when impact resolves.
- **1** activates the granted self-heal ability.
- **H** applies the reusable healing Effect to the target.
- **R** destroys the generation-checked target and spawns a fresh target, demonstrating handle replacement.

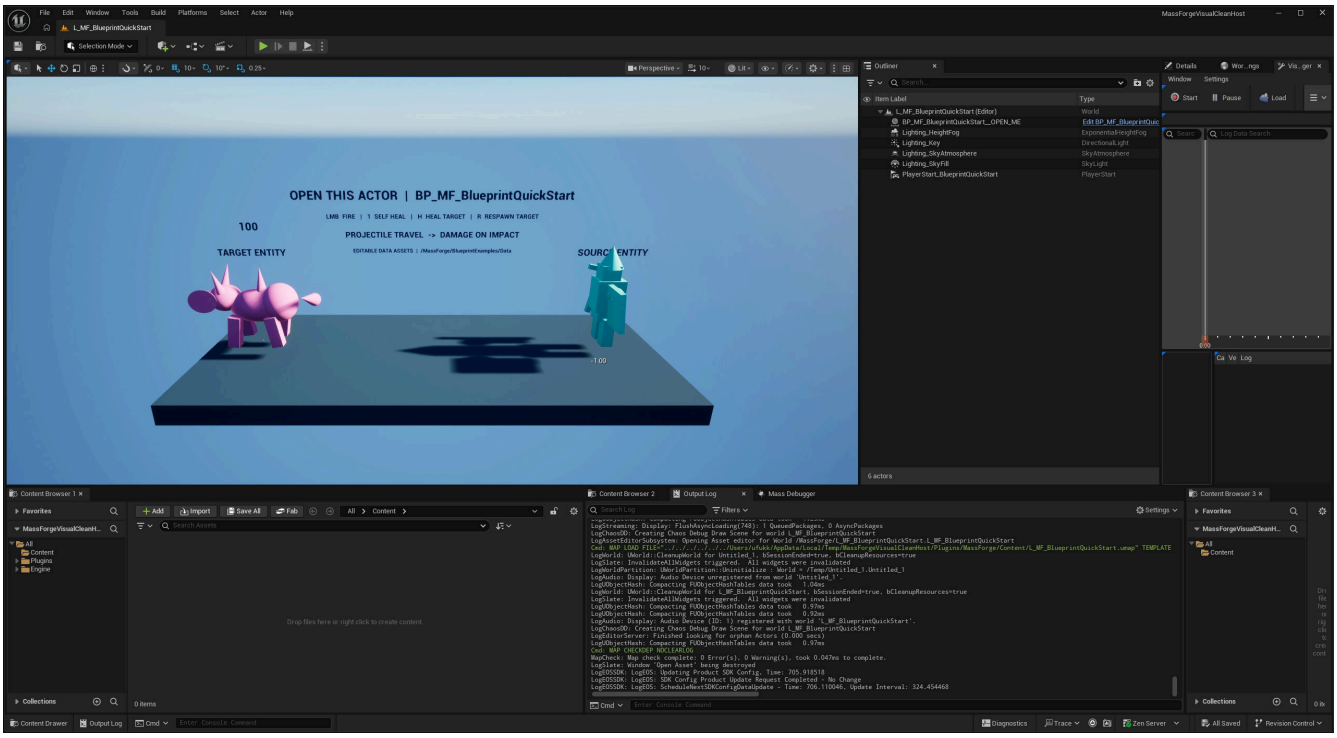
The saved map includes a Player Start and a warning-free Quick Start GameMode. The camera is owned by the teaching Blueprint, the target begins at `100` Health, and no projectile or damage runs automatically. The target disappears only after its real entity Health reaches zero; **R** creates a visible replacement at `100` Health.

The actor enables input in its own Blueprint Event Graph. Its source and target spawn failure pins are connected to visible/logged diagnostics, so a broken setup does not silently continue with an unset handle.

The target's floating Health label is read from the real actorless Mass entity on Tick. It is not a duplicated UI variable. This keeps the visible projectile, impact-driven damage, healing, and respawn path auditable in one small Blueprint-owned scene.

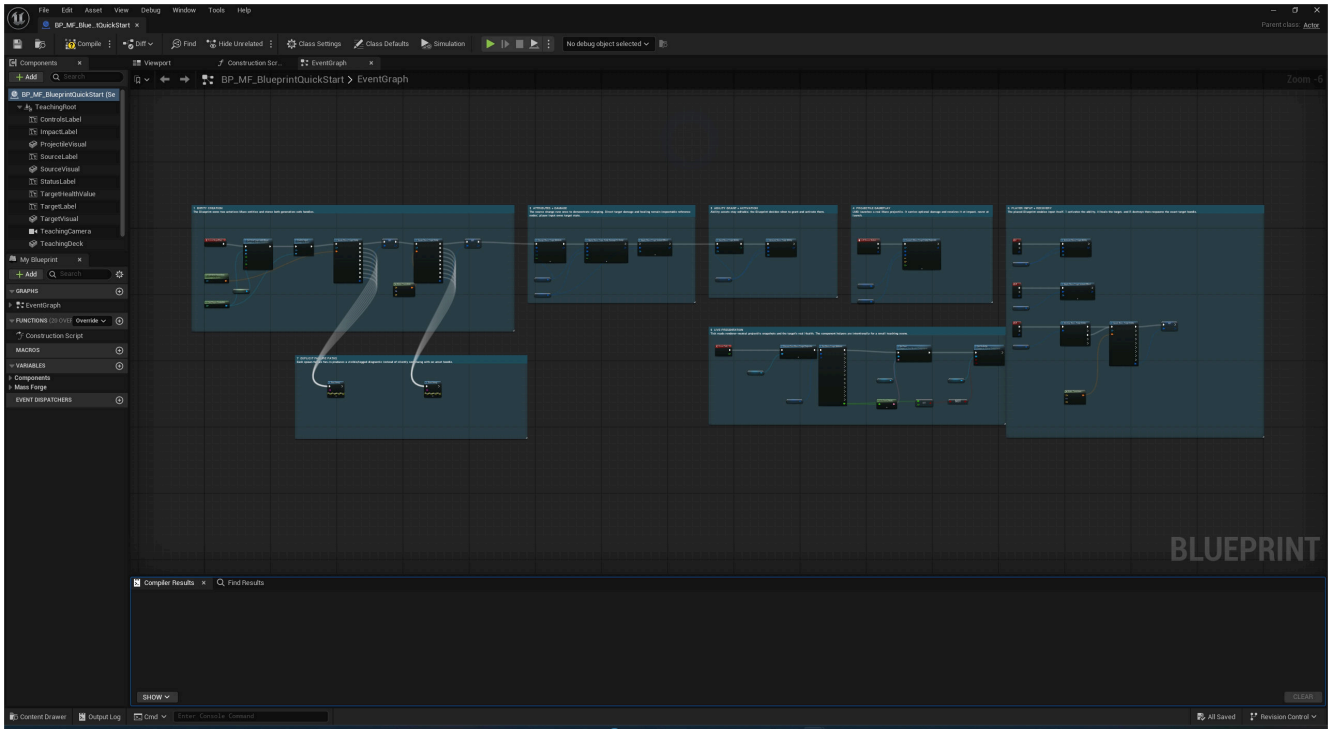
*This is the result of one real projectile impact: the target began at 100 Health, the visible projectile traveled, and the floating value now reads the entity's authoritative 80 Health. Press **H** to heal or **R** to replace the exact target handle.*

The teaching deck below is serialized in the plugin map before Play: the Outliner contains the Blueprint gameplay actor, Player Start, and authored lighting actors. Runtime behavior updates this existing scene; it does not secretly construct a different showcase world in `BeginPlay`.

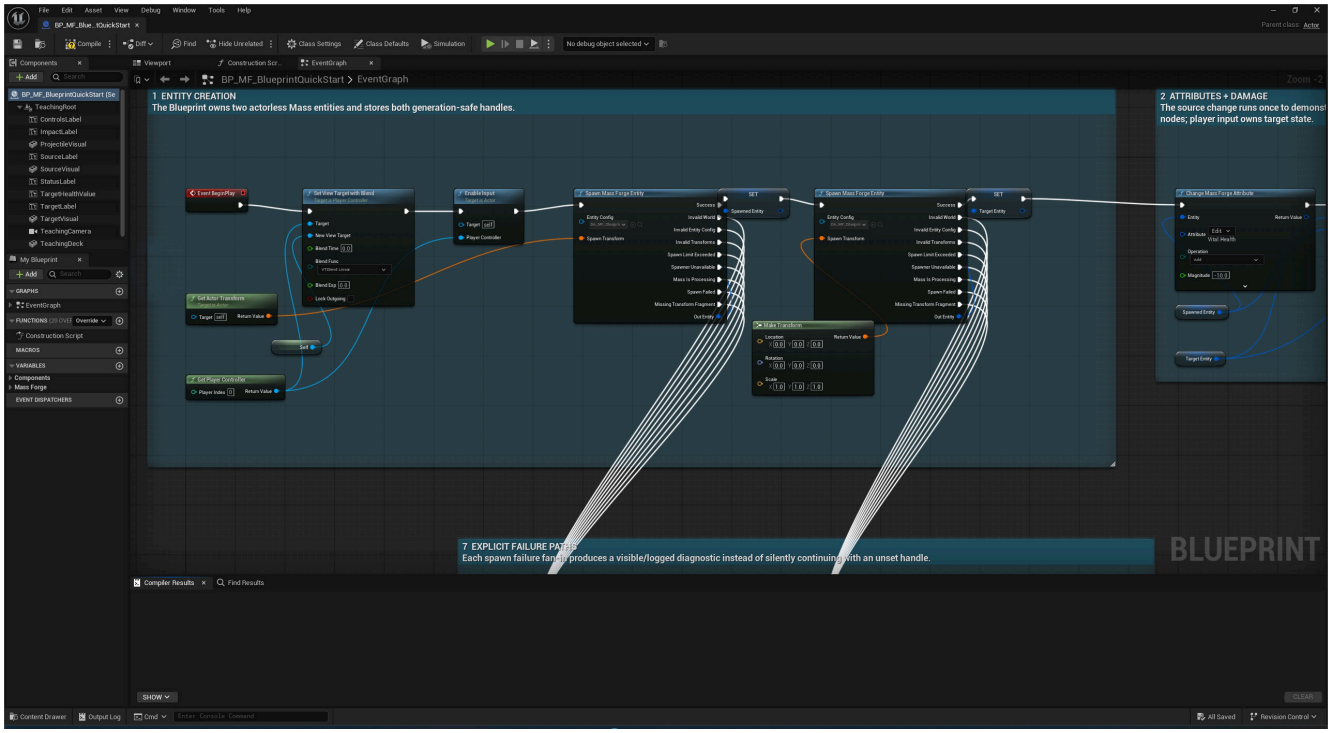


Read the shipped Event Graph

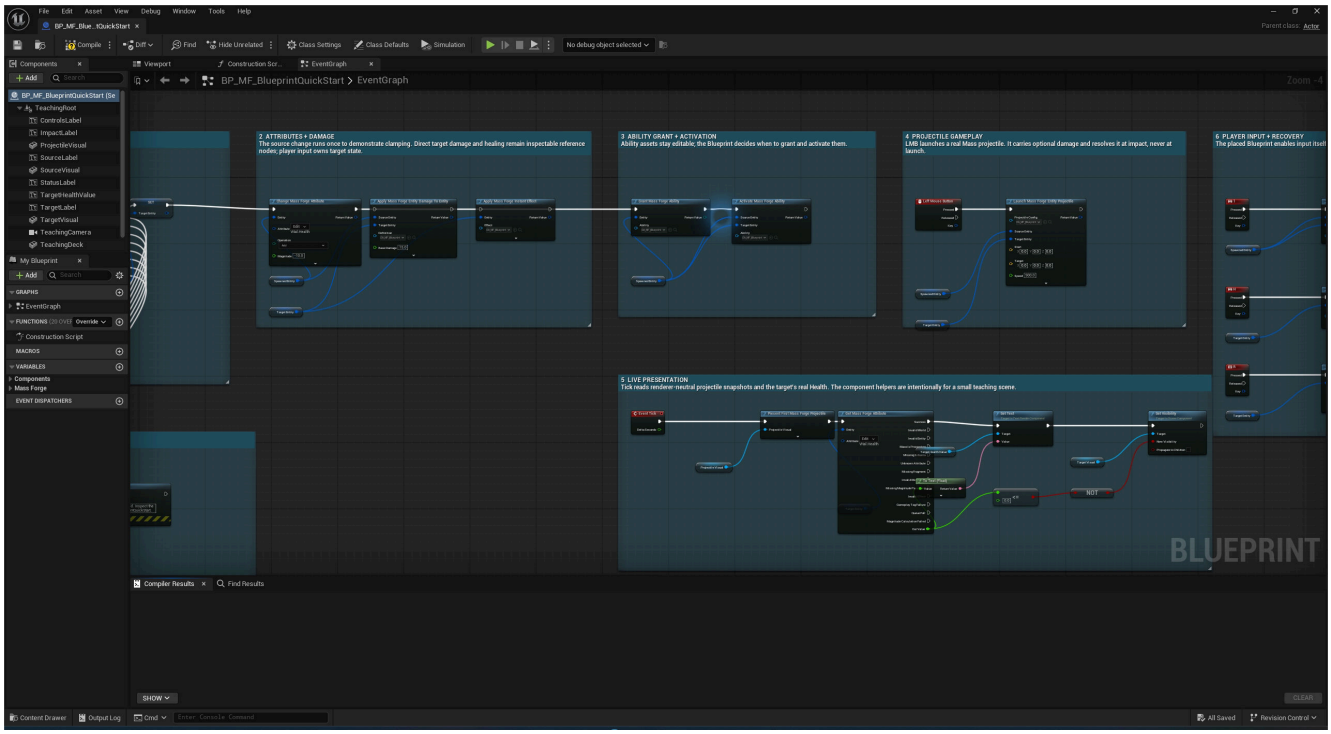
The complete graph is intentionally divided into seven numbered stations. The overview shows the whole ownership flow; the closer views keep real node names, asset pins, handle wires, and execution order readable.



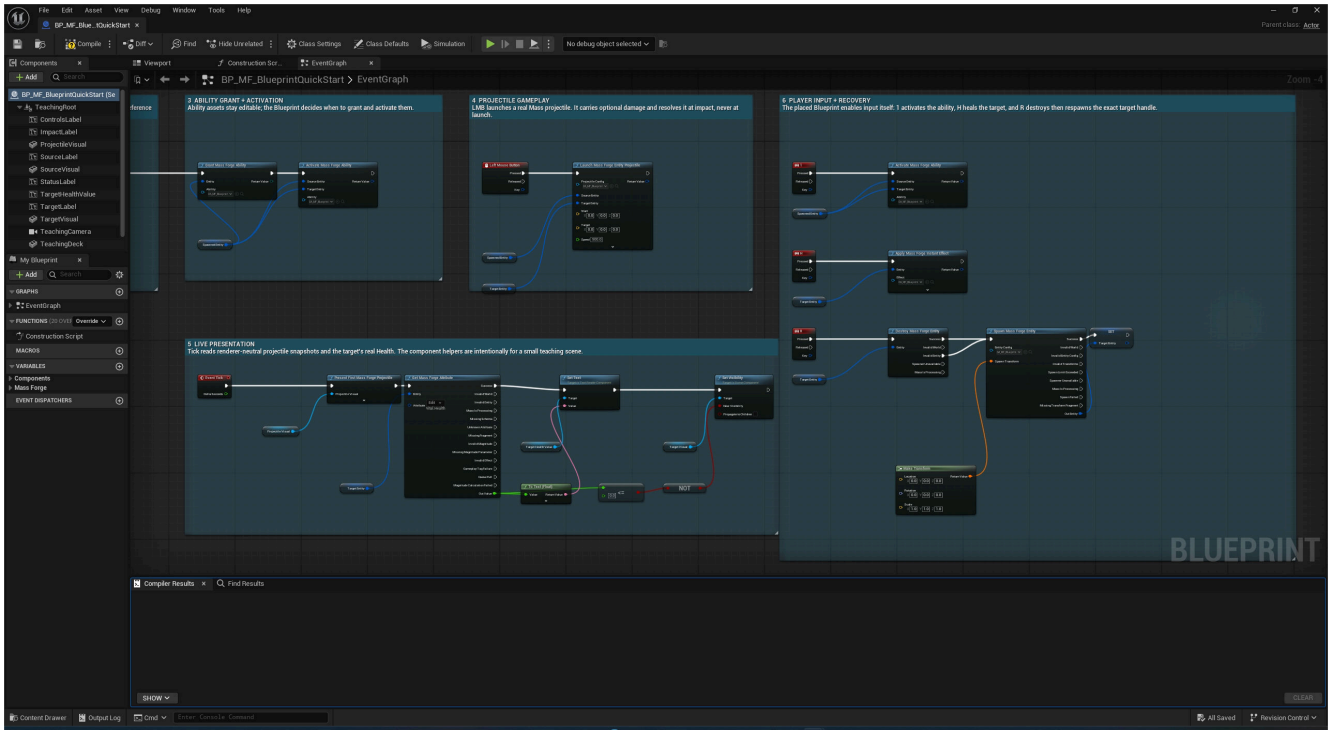
Station 1 owns two actorless entities and stores their complete generation-safe handles. Both spawn nodes expose every failure result instead of converting setup failure into an unset handle.



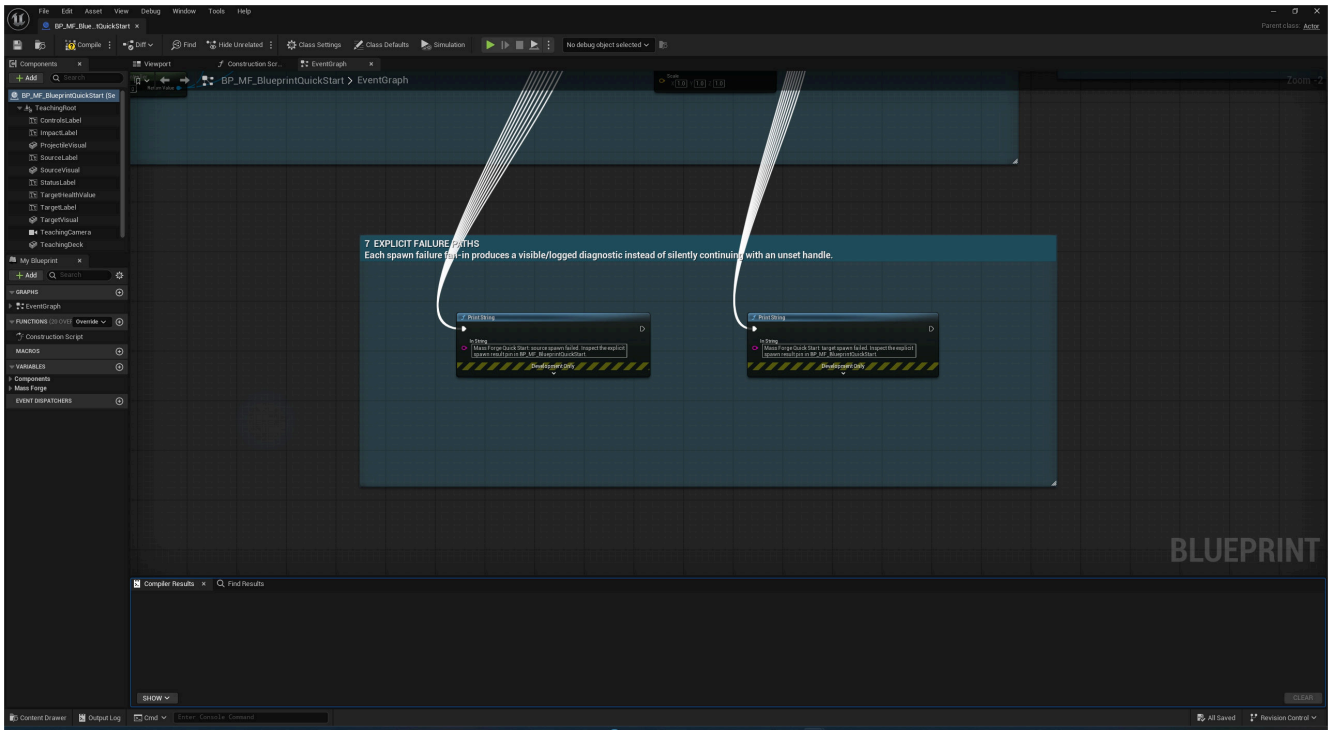
Stations 2-4 demonstrate a direct attribute change, entity-to-entity damage, reusable healing, ability grant/activation, and a Mass-owned projectile whose optional damage resolves at impact.



Stations 5-6 keep presentation separate from authority: Tick reads renderer-neutral snapshots and real Health, while input launches the ability/effect/projectile actions and replaces the exact destroyed target handle.



Station 7 makes spawn failure visible in the viewport and log. It is deliberately separate from the success chain so a tutorial user can inspect and extend the recovery policy without touching gameplay authority.



Five-minute first attribute

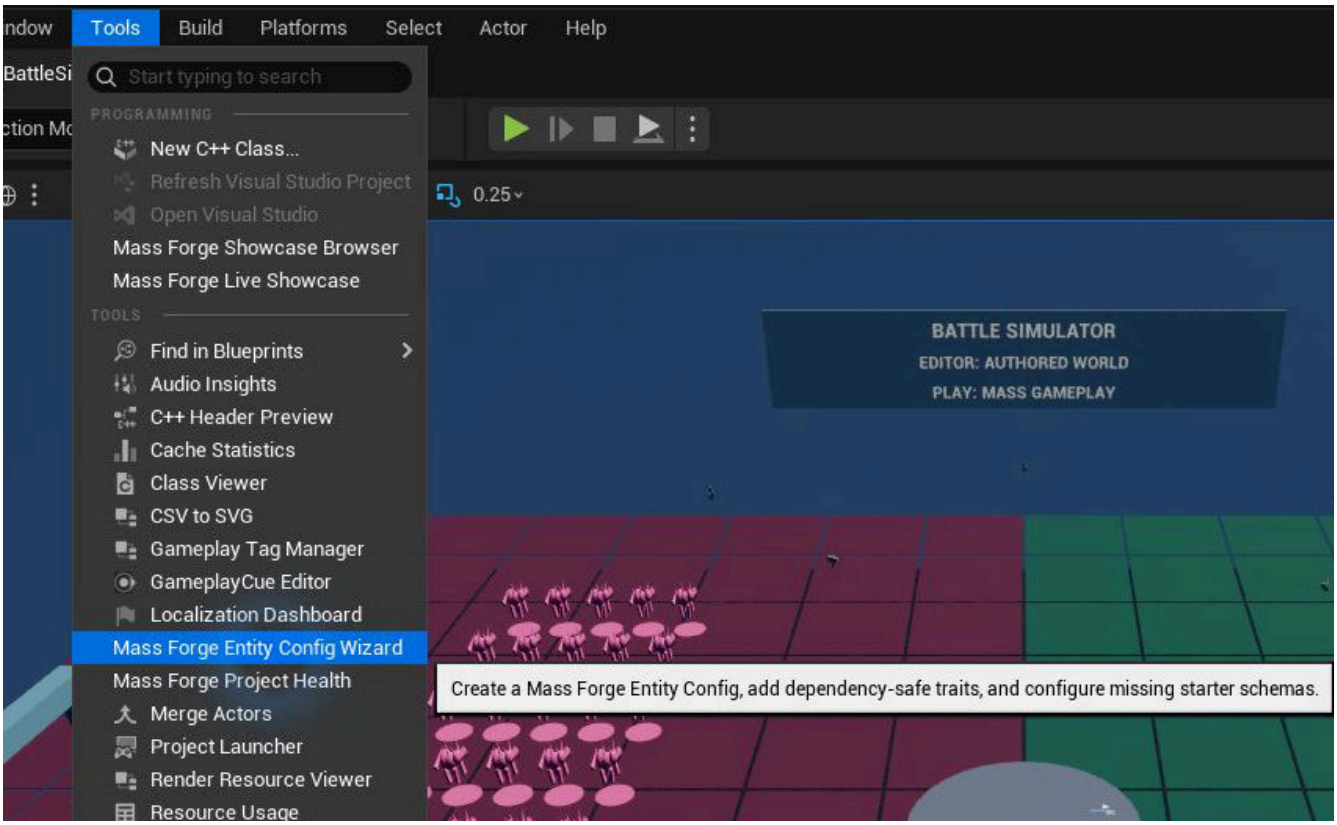
1. Install and enable

1. Close Unreal Editor.
2. Copy the distributed `MassForge` folder into the project's `Plugins` folder.
3. Open the project in Unreal Engine 5.6, 5.7, or 5.8 on Win64.
4. Enable **Mass Forge** under **Edit > Plugins**, then restart when prompted.

If the plugin does not load, confirm that `MassForge.uplugin` is directly inside `Plugins/MassForge`, not inside a second nested `MassForge` folder.

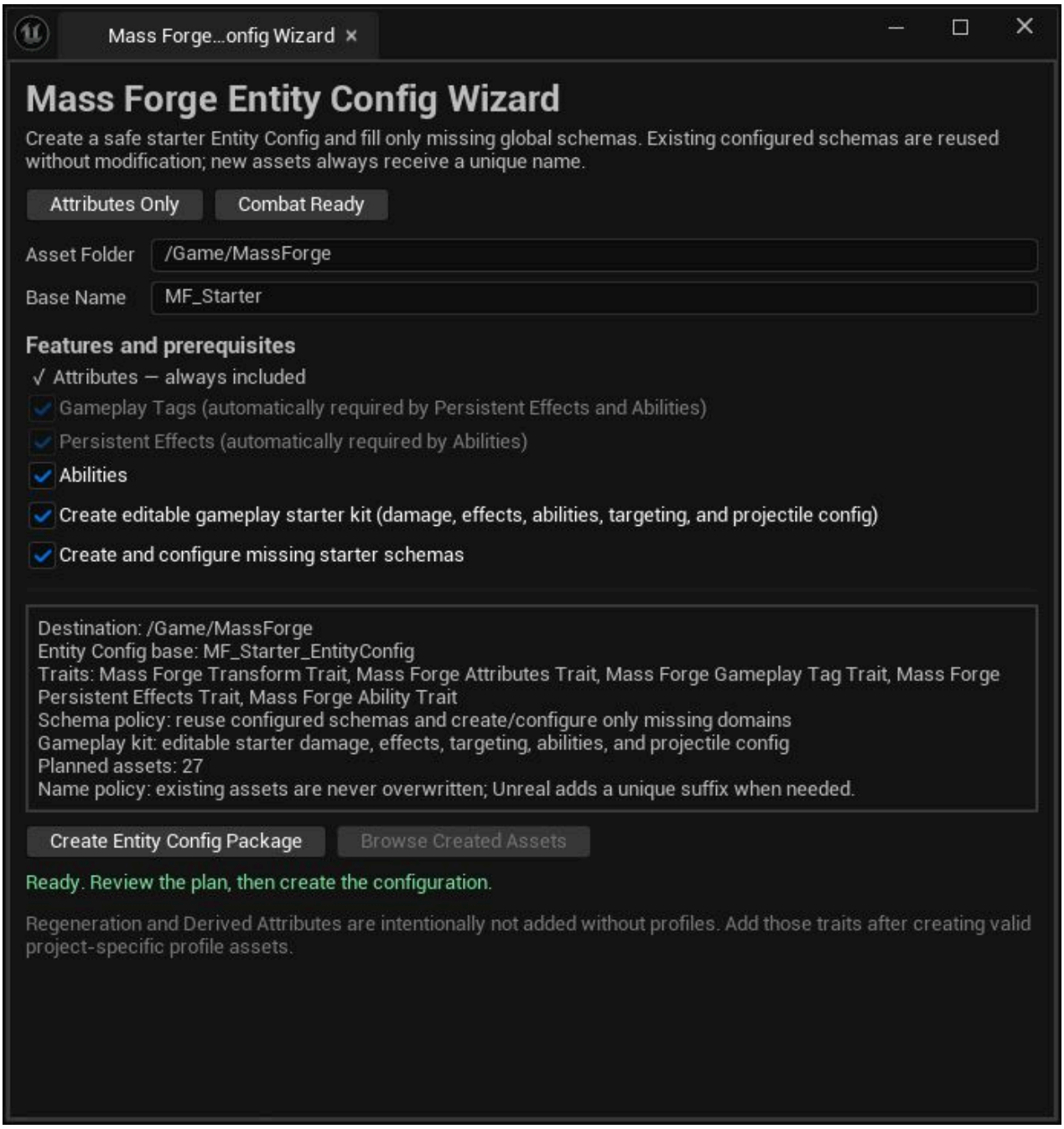
2. Generate the starter package

1. Open **Tools > Mass Forge Entity Config Wizard**.

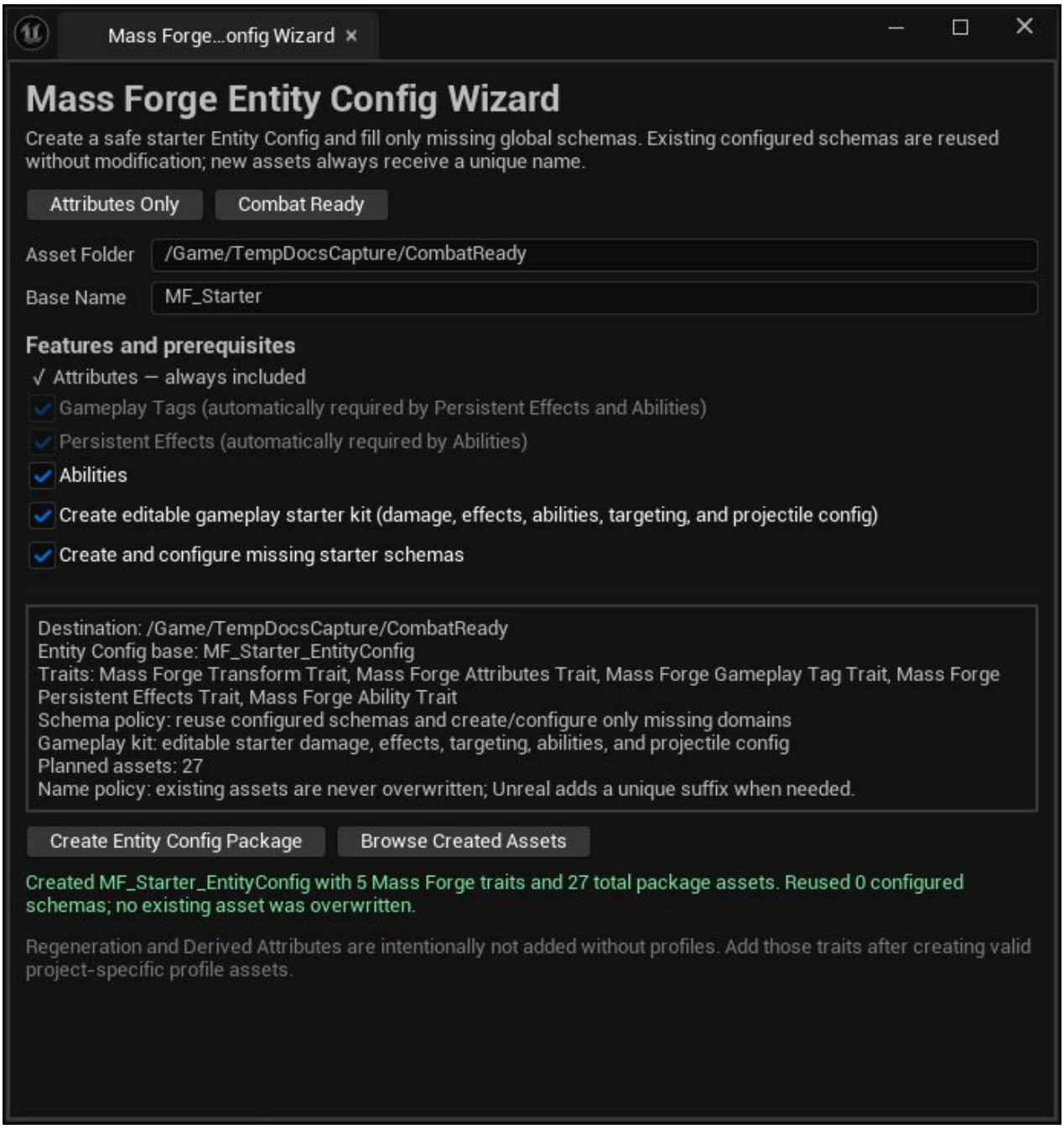


*Open the standard Unreal **Tools** menu and choose the highlighted Mass Forge entry. This is an editor authoring tool; it does not run during the game.*

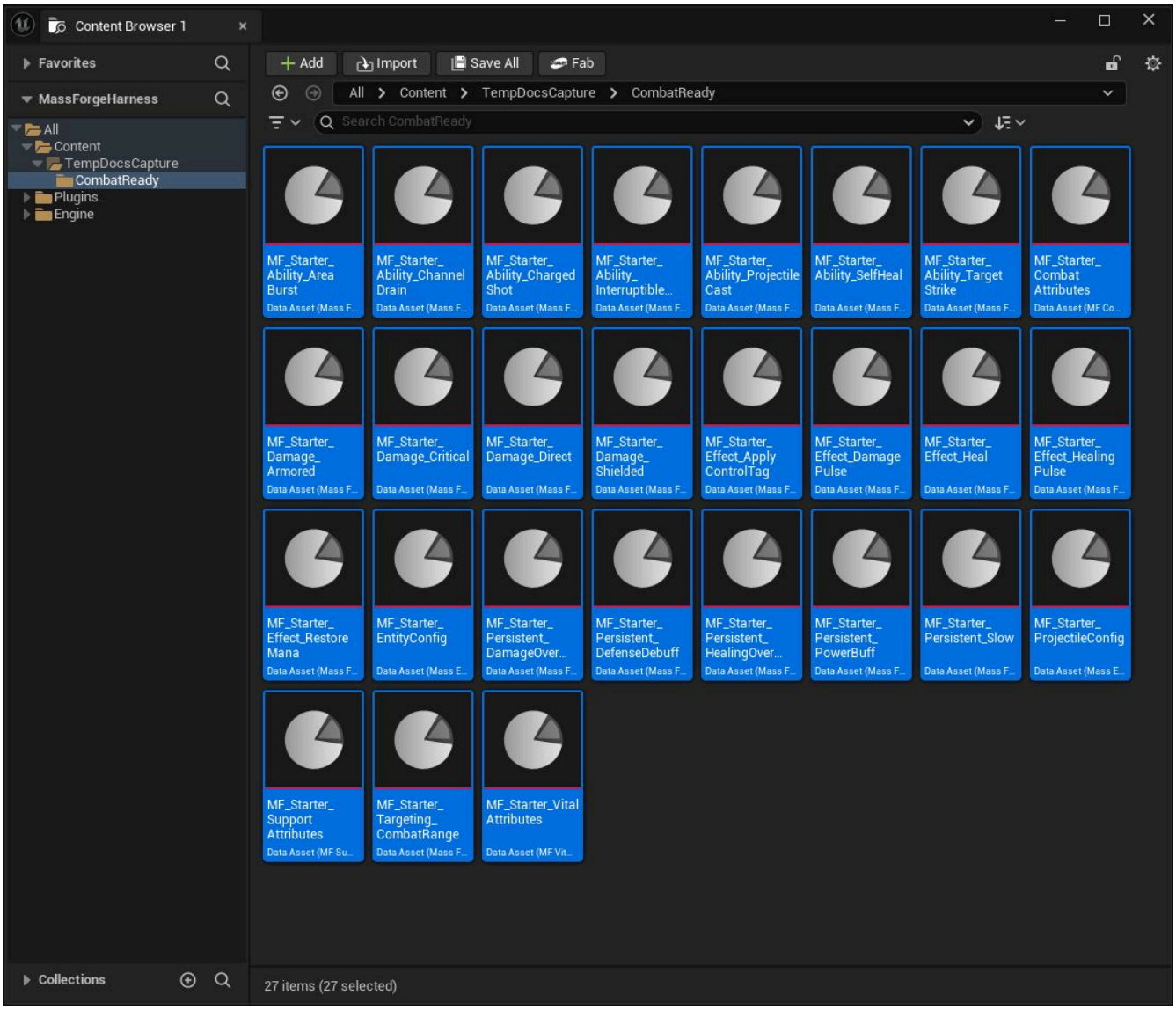
2. Select **Combat Ready**.
3. Keep **Create and configure missing starter schemas** enabled.
4. Choose a project-owned folder such as `/Game/Combat/MassForge` and a base name such as `Soldier`.
5. Select **Create Entity Config Package**.



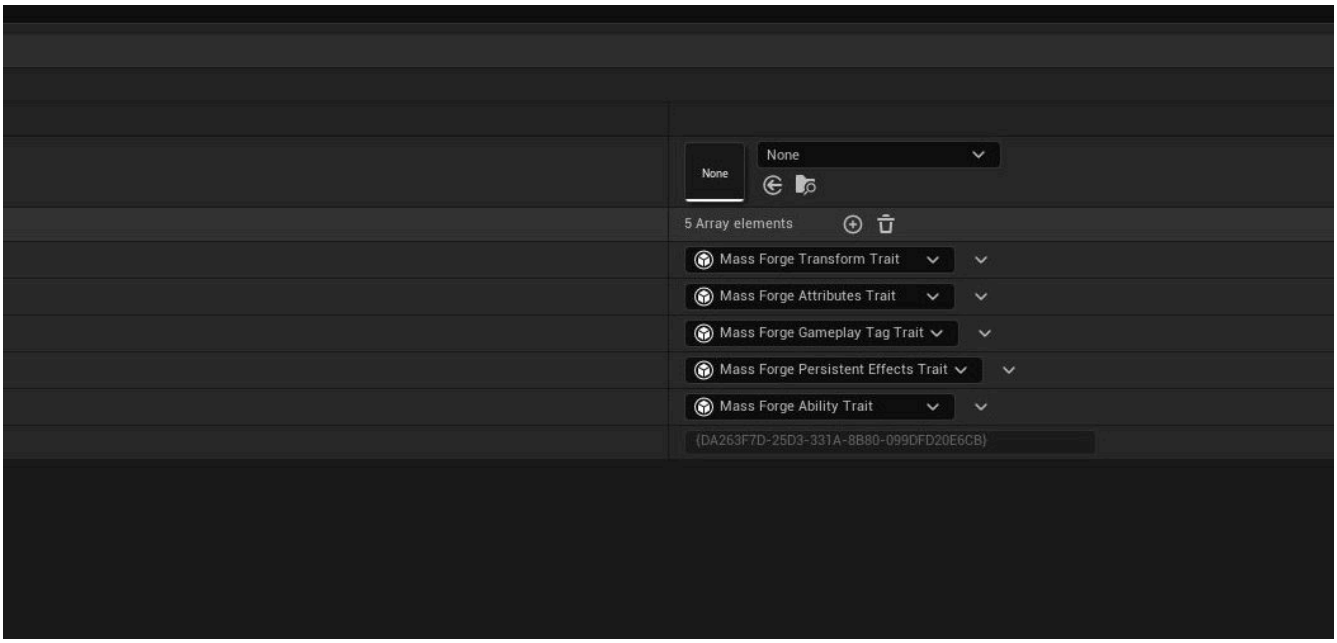
*The live plan names every trait, schema, and gameplay asset that will be created. Change the folder and base name before pressing **Create Entity Config Package**; existing assets are never overwritten.*



After generation, read the green result before continuing. This run created one Entity Config with five Mass Forge traits plus 27 editable package assets and did not overwrite existing content.

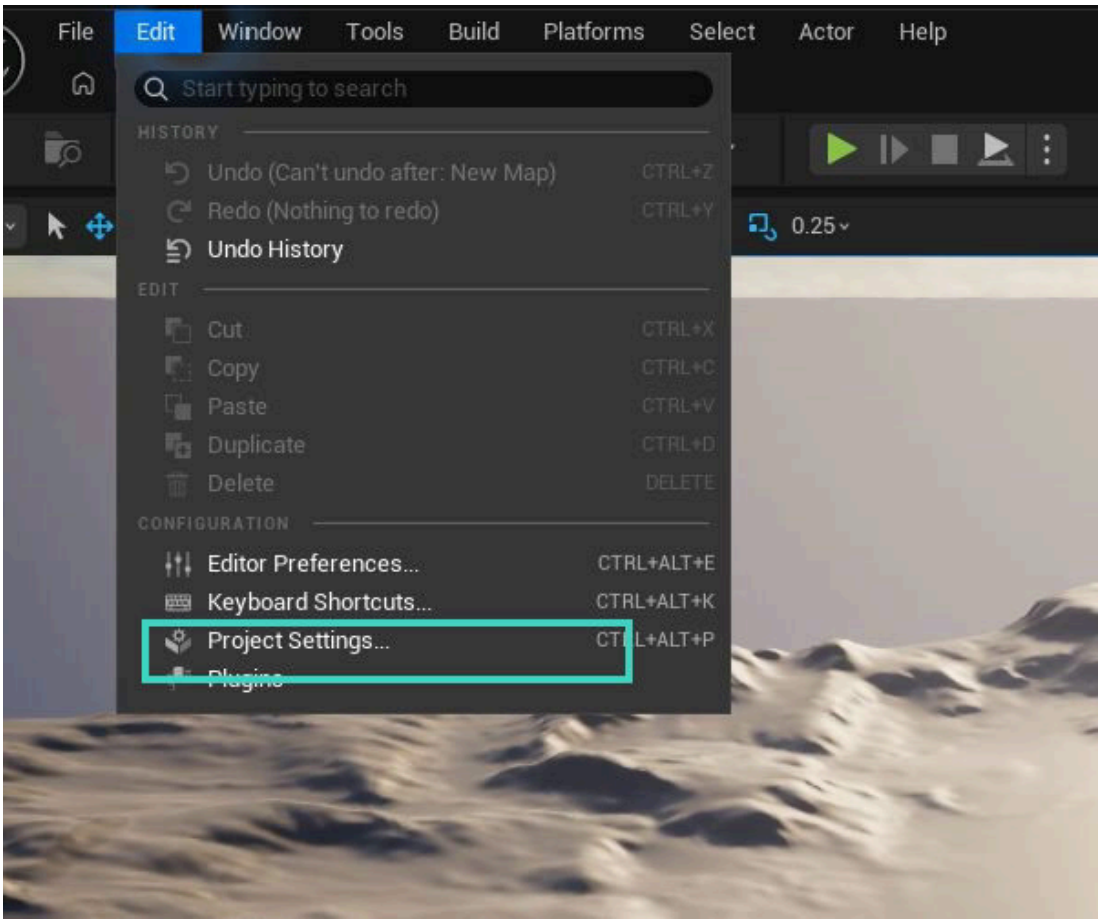


The wizard selects the complete result in the Content Browser. These are ordinary project-owned Data Assets: inspect, duplicate, rename, or replace them to fit the game.

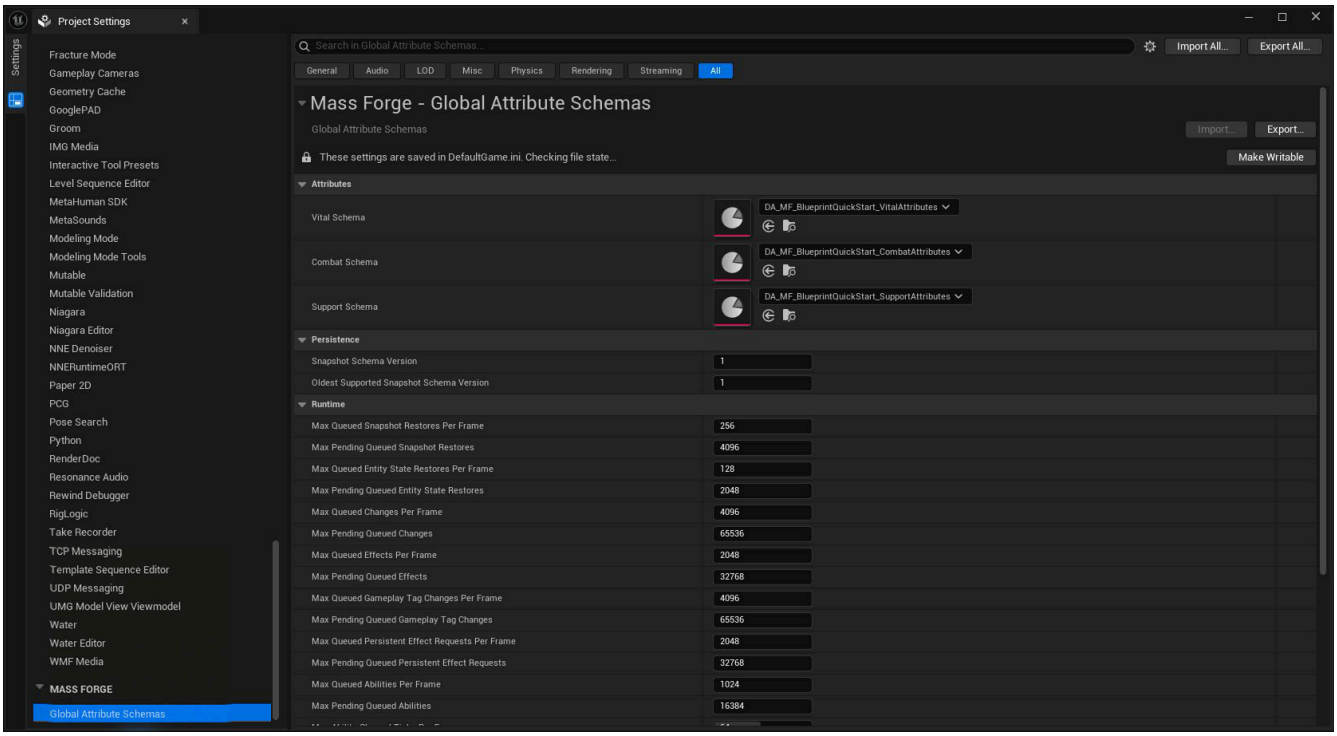


Open the generated Entity Config and confirm these five traits before spawning. They provide transform storage plus the attribute, tag, persistent-effect, and ability fragments used by this tutorial.

The wizard creates an ordinary Mass Entity Config plus any missing Vital, Combat, and Support schemas. It never overwrites an existing asset and reuses schemas already configured in **Project Settings > Mass Forge > Global Attribute Schemas**.

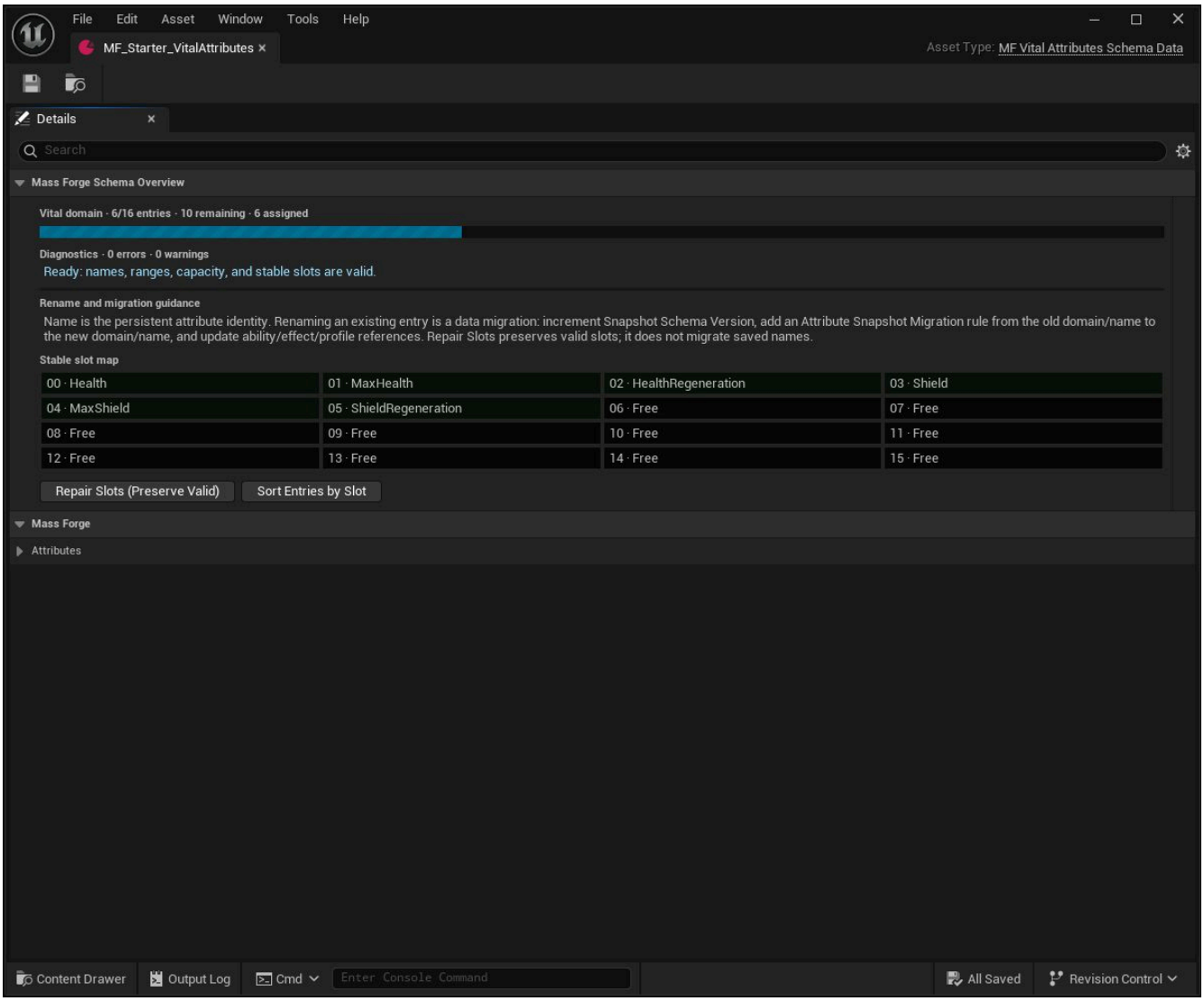


*Open **Edit > Project Settings** to verify the project-wide schema assignments after generation.*

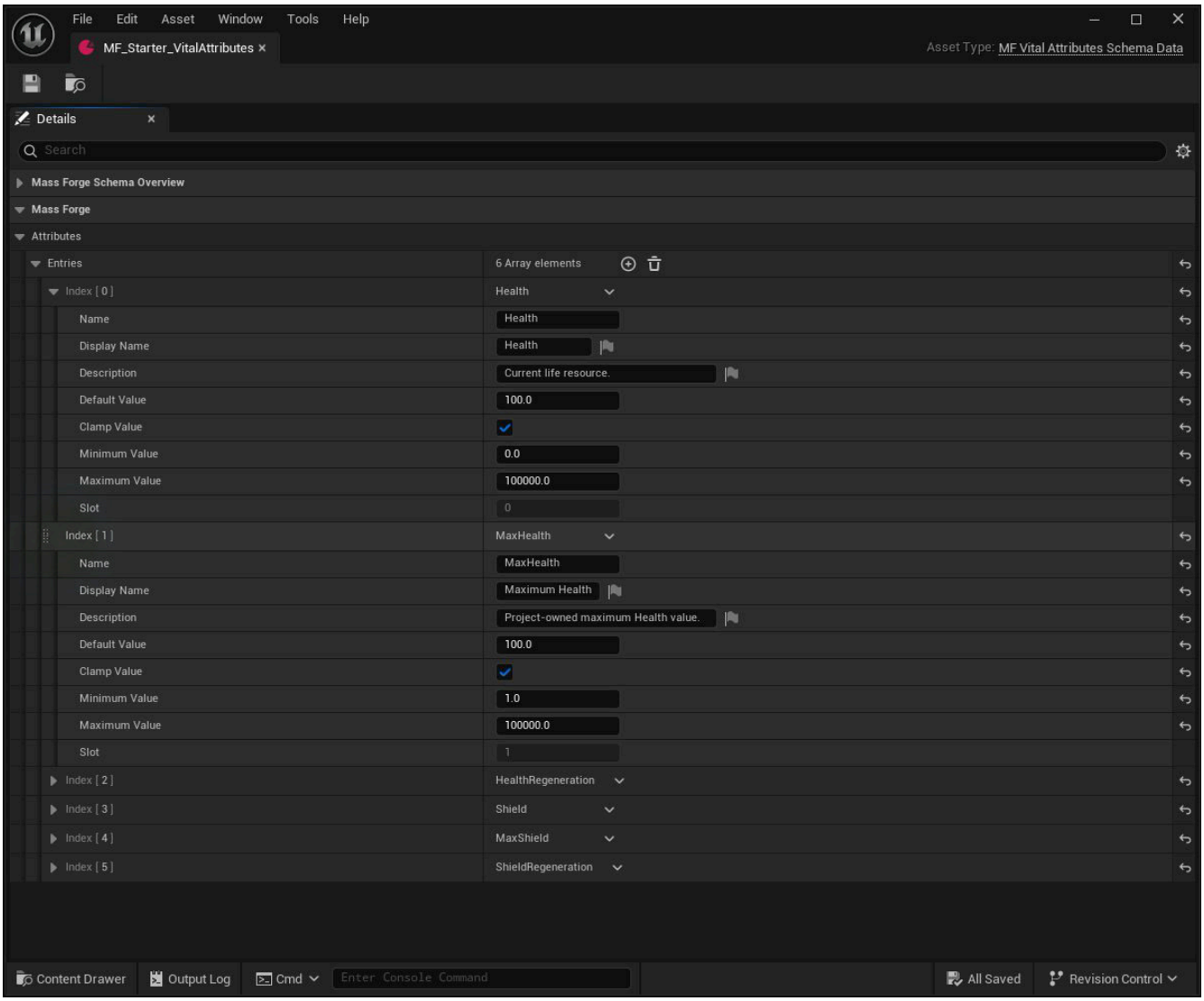


All three schema domains must point to the intended project assets. The runtime limits below them are advanced capacity controls; leave the defaults unchanged for this first workflow.

Open the generated Vital and Support schemas. Their overviews should show unique, in-range slots and no blocking diagnostics. For this tutorial, confirm that `Vital.Health`, `Vital.MaxHealth`, and `Support.Mana` exist. Change their defaults and bounds now if the project's units differ from the starter values.



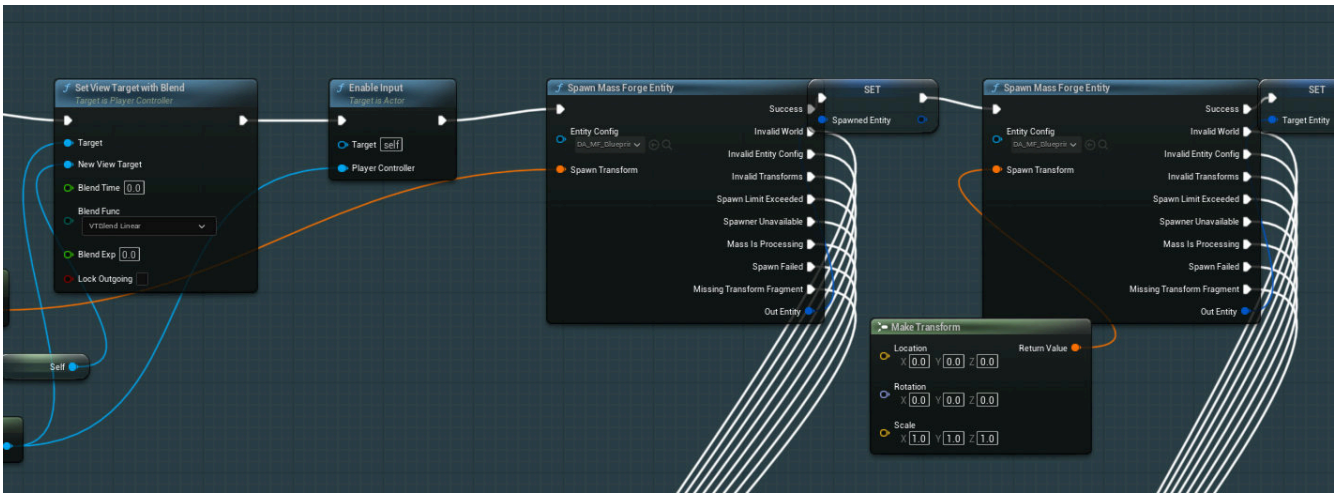
The overview is the fast safety check: zero errors, zero warnings, unique slots, and remaining capacity. Repair identity or slot problems here before building dependent assets.



Expand the entries to author real project values. `Health` is the current resource in slot 0; `MaxHealth` is a separate project-owned capacity in slot 1, not a hidden automatic binding.

3. Spawn and resolve an entity

Call **Spawn Mass Forge Entity** with the generated Entity Config and a world transform. The generated config includes **Mass Forge Transform Trait**, so the node returns a generation-safe handle on its **Success** execution pin. Use **Spawn Mass Forge Entities** with an array of transforms for a bounded Blueprint batch. Blueprint batches are capped at 10,000 returned handles; use a project C++ population path for larger allocations instead of moving enormous handle arrays through a Blueprint graph.



*Store the complete returned handle only from **Success**. The example keeps source and target handles distinct and routes every failure pin to its diagnostic station outside this close-up.*

The matching **Destroy Mass Forge Entity** node validates the complete generation and rejects Mass-processing overlap. **Valid MF Entity?** is a convenience predicate for UI and ordinary Blueprint flow; every gameplay operation still performs its own full validation.

In a Blueprint that receives the represented Actor:

`Actor → Get Mass Forge Entity From Actor → Success execution pin`

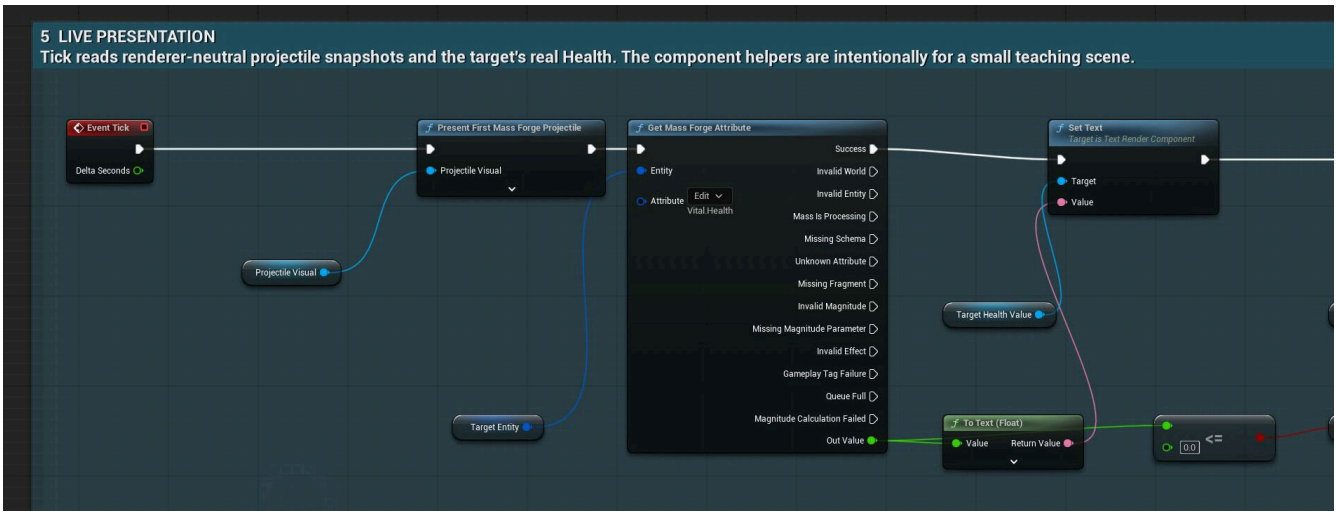
Use the returned **Mass Forge Entity Handle** only from the **Success** pin and store it only as long as needed. Other execution pins distinguish an invalid world or Actor, an Actor from another world, an unavailable representation subsystem, an ordinary unrepresented Actor, and a stale representation. The handle contains an index and serial number; every operation revalidates both, so it cannot silently target a recycled entity.

Actorless entities use exactly the same handle-based nodes. Obtain their handles from the radius/segment target-selection nodes, a project target provider, the Entity Inspector, or a Mass processor integration. Representation is not required by the attribute system.

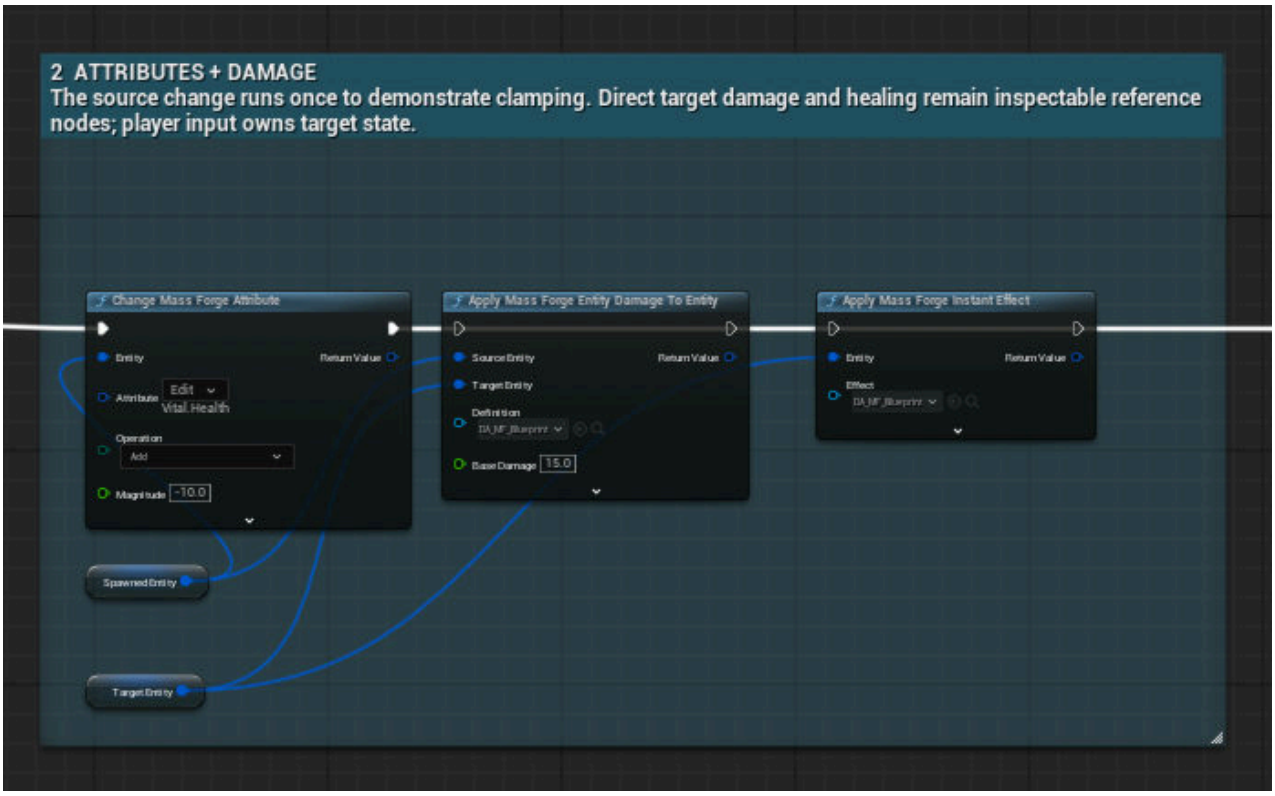
4. Read and change Health

From the successful resolve branch:

1. Call **Get Mass Forge Attribute**.
2. Select `Vital.Health` from the schema-backed Attribute picker.
3. Use the **Success** execution pin and display the returned value.
4. Call **Change Mass Forge Attribute** with `Add` and `-10`.
5. Branch on `Result`; on success, use `Old Value`, `New Value`, and `Was Clamped` from the returned change structure.



The shipped teaching graph reads the entity's real Health and writes it to the visible label. It does not maintain a second UI-owned Health variable; failed reads take an explicit non-success execution path.



This compact teaching chain places the three common operations side by side. In production, branch on each returned result and use the queued or async form whenever execution may overlap Mass processing.

The direct change is appropriate only while the Mass entity manager is idle. A successful change obeys the schema bounds, so Health cannot pass a configured minimum or maximum.

> **WARNING - Direct versus queued**
>
> Use a direct node only at a lifecycle point where Mass is known to be idle. From callbacks, timers, asynchronous work, or uncertain execution contexts, use the matching queued/async node and wait for its terminal result.

Production-safe queued attribute graph

Use the queued node from timers, callbacks, asynchronous gameplay, or any Blueprint path that may overlap Mass processing:

`Event → Queue Mass Forge Attribute Change → Branch on Receipt.Result → Store Receipt.RequestId`

Then:

1. Call **Get Mass Forge Attribute Subsystem**.
2. Bind once to **On Attribute Request Completed**.
3. Match the callback's request ID against the stored ID.
4. Inspect the completed change result before updating UI or advancing gameplay.

Queue acceptance means the request entered a bounded FIFO queue; it is not proof that execution later succeeded. The entity and attribute are revalidated at execution. `Queue Full` is explicit back-pressure, while stale entities and invalid attributes are reported by the completion result.

For ordinary Blueprint gameplay, **Change Mass Forge Attribute (Async)** performs this submission, request-ID matching, and delegate cleanup in one node while preserving the same detailed completion and rejection paths. Use the explicit queue plus subsystem delegate when several systems intentionally share a central listener.

Production-safe queued snapshot restore

After loading and migrating a project-owned save snapshot, use the direct **Restore Mass Forge Attribute Snapshot** node only at a guaranteed Mass-idle lifecycle point. Otherwise:

`Load/Migrate → Queue Mass Forge Attribute Snapshot Restore → Branch on Receipt.Validation.Result → Store Receipt.RequestId`

Bind once to **On Attribute Snapshot Restore Request Completed** on **Get Mass Forge Attribute Subsystem**, match the positive request ID, then branch on `Completion.Restore.Result`. Rejected submissions have ID `0` and no later event. An accepted request owns a copy of the snapshot, revalidates the exact entity generation and schema at execution, commits all values or none, publishes its attribute changes, and finally emits one completion.

Snapshot restores run before queued raw attribute changes in the first coordinator phase. This makes the saved snapshot the base state; a same-tick queued gameplay change is applied afterward. See PERSISTENCE_GUIDE.md for migrations, version diagnostics, and save-scope limits.

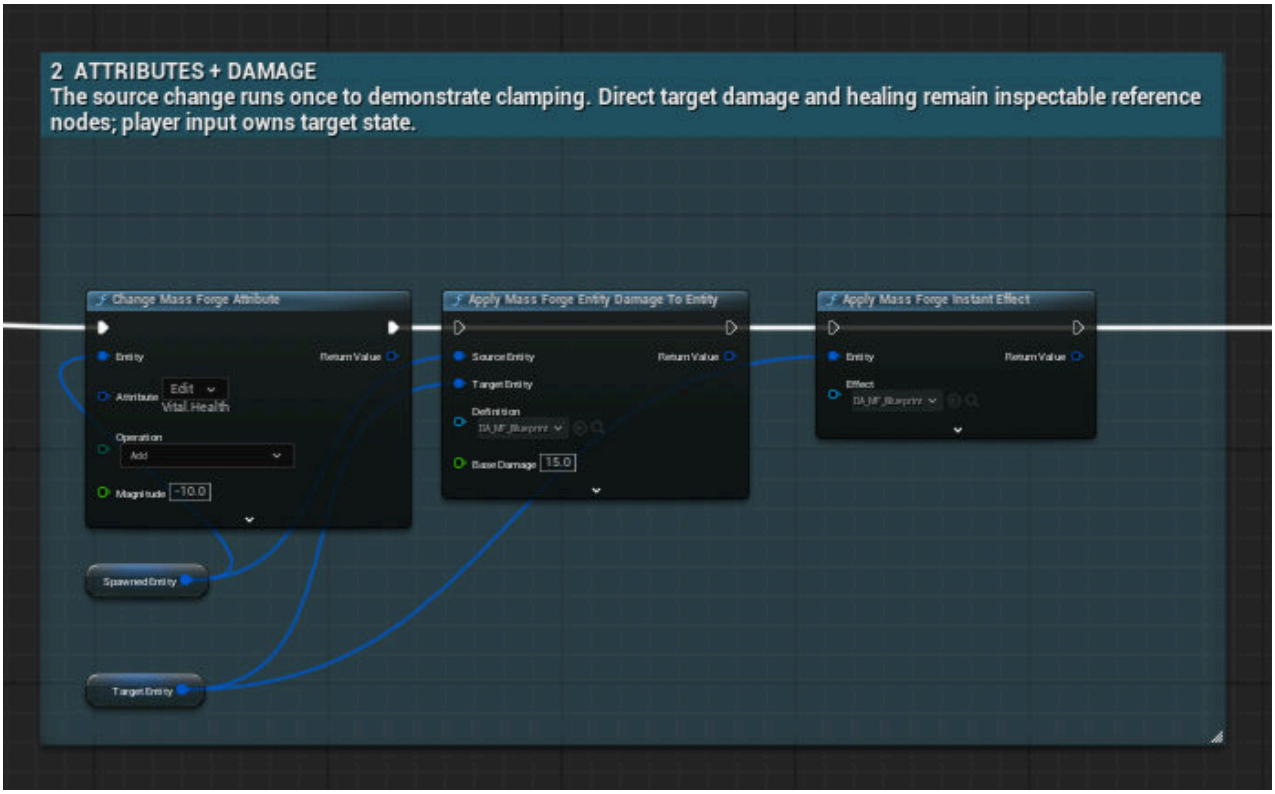
For a stable checkpoint that must also preserve exact counted tags, ability grants, cooldowns, charges, regeneration timing, active Persistent Effects, and active cast/channel phase, use **Capture Mass Forge Entity State** and **Restore Mass Forge Entity State**. In timing-sensitive graphs, use **Queue Mass Forge Entity State Restore**, store its positive request ID, then match **On Entity State Restore Completed** on **Get Mass Forge Entity State Subsystem**. Format 4 gives restored effects and lifecycles fresh world-local handles. When an active lifecycle targets another entity, store a project-owned stable relationship key with **Capture Mass Forge Entity State with Lifecycle Target Reference**, recreate both entities, then use **Restore Mass Forge Entity State with Lifecycle Target** or **Queue Mass Forge Entity State Restore with Lifecycle Target** with the same key and the new target handle. Never save the old Mass handle. Formats 1-3 remain readable under their documented boundaries. See ENTITY_STATE_PERSISTENCE_GUIDE.md.

Deterministic player-to-Mass damage

Author the Damage Definition

1. Create **Miscellaneous > Data Asset > Mass Forge Damage Definition**.
2. Set a stable `Damage Id`, for example `Damage.Player.Primary`.
3. Select `Vital.Health` as **Health Attribute**.
4. Optionally enable Shield, source Power, mitigation, criticals, tag gates, a death tag, and one of the three post-death policies.
5. Save the asset and resolve every validation error shown in its authoring overview.

For an ordinary player Actor damaging a Mass target, call **Apply Mass Forge Actor Damage To Entity**. Supply the player Actor, target handle, definition, and non-negative base damage. The convenience node places the player in the effect context and resolves a source entity automatically if that Actor represents one.



Damage receives both source and target handles; healing and reusable resource changes enter through an authored Instant Effect. The nearby direct attribute node is intentionally a lower-level stat operation, not a replacement damage formula.

Always inspect the returned **Mass Forge Damage Application Result**. `Success` includes the resolved formula stages, Health/Shield before and after, overkill, critical state, atomic transaction, kill state, and death-handling scheduling result. A failed result makes no partial Health, Shield, or death-tag commit.

Queued damage graph

When the call may overlap Mass processing, build a **Mass Forge Damage Request** and use **Queue Mass Forge Damage**:

- set `Target Entity` to the generation-checked target;

- leave `Source Entity` unset for an ordinary player unless the formula samples source attributes;
- set `Context.Source Actor` to the player;
- set Base Damage and the optional penetration, critical roll, and shield-bypass values;
- store the accepted request ID and match it in **On Damage Request Completed** from **Get Mass Forge Damage Subsystem**.

If the definition samples source attributes, an ordinary Actor is insufficient because it has no Mass Forge attribute fragments. Represent the player as a Mass entity or author a definition that does not require source attributes.

Apply Mass Forge Damage (Async) is the low-ceremony production form of the same queue contract. It returns only the matching request's complete damage application and keeps submission rejection separate from execution completion.

Healing and resource restoration

Healing is an Instant Effect, not negative damage.

1. Create a **Mass Forge Instant Effect** with an ID such as `Effect.Heal.Small`.
2. Add one attribute modifier: `Vital.Health`, operation `Add`, magnitude `25`.
3. Save and validate the effect.
4. Call **Apply Mass Forge Instant Effect** with the target handle.

The Vital schema clamps the result to the authored Health maximum. A Mana restore uses the same recipe with `Support.Mana`. Several modifiers can be placed in one Instant Effect; the complete attribute/tag transaction succeeds atomically or changes nothing.

Use **Queue Mass Forge Instant Effect** on overlap-prone paths. Match its request ID in **On Instant Effect Request Completed** and inspect the full result. Do not replace healing with `Change Mass Forge Attribute` when the operation needs source context, tags, several atomic modifiers, or reusable designer-authored data.

For a local Blueprint flow, **Apply Mass Forge Instant Effect (Async)** performs that correlation automatically.

One Blueprint-authored self-heal ability

Create `Ability.SelfHeal` as a **Mass Forge Ability**:

1. Keep **Execution Type** set to `Instant`.
2. Add `Support.Mana` with amount `10` under **Costs**.
3. Add `Effect.Heal.Small` under **Self Effects**.
4. Set **Cooldown Seconds** to the desired value, for example `5`.
5. Leave **Requires Target** disabled.
6. Add the ability to **Starting Abilities** on the generated Mass Forge Ability Trait, or call **Grant Mass Forge Ability** at runtime.

The minimal activation graph is:

```
Entity Handle → Can Activate Mass Forge Ability → inspect Result → Activate Mass Forge Ability
```

*Granting controls runtime ownership; activation performs the ability's validated costs, cooldowns, tags, effects, and lifecycle. Add **Can Activate Mass Forge Ability** before activation when UI or AI must explain a rejection without changing state.*

Pass an unset target handle for this self-only ability. The same graph works for actorless and represented Mass entities. No presentation listener is required.

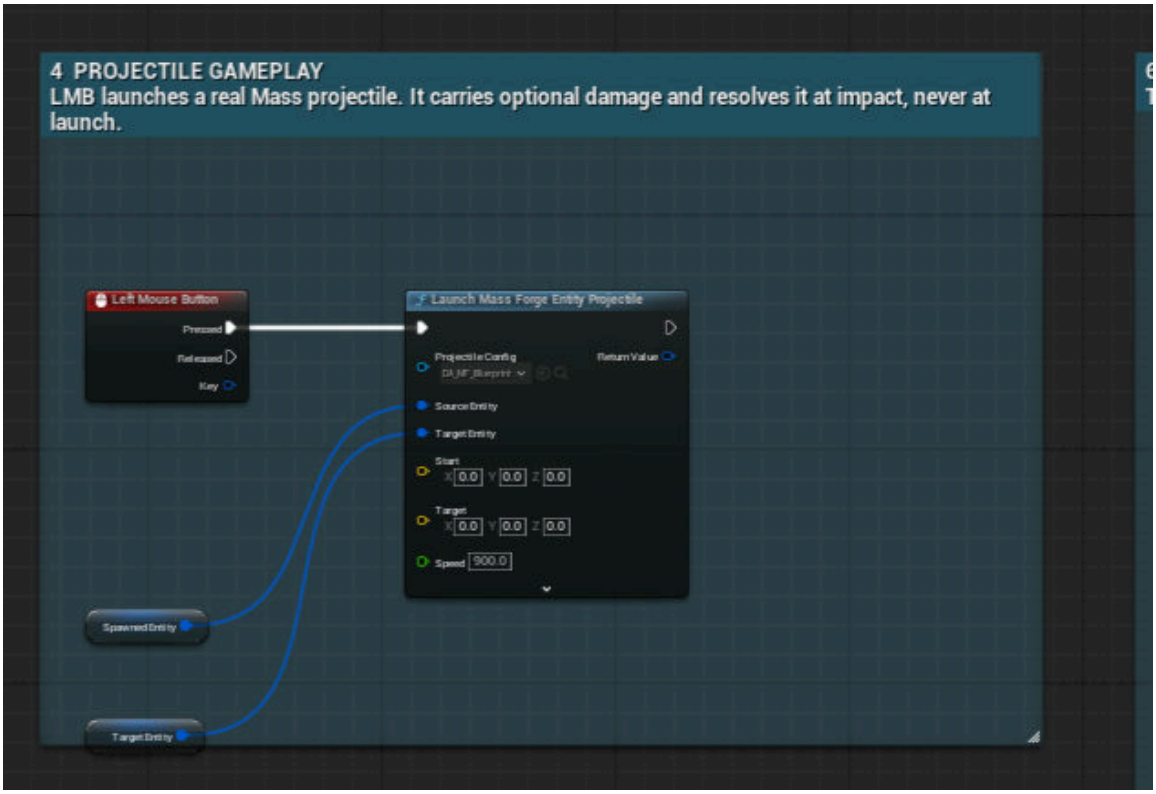
For production callbacks, use **Queue Mass Forge Ability Activation**, store its accepted request ID, and match **On Ability Operation Request Completed** from **Get Mass Forge Ability Subsystem**. Costs, cooldown, tags, self effects, target effects, and persistent outcomes commit as one validated transaction; a later failure rolls the provisional state back. The older **On Ability Request Completed** event remains an activation-only compatibility event.

Runtime ownership and lifecycle control use the same production pattern. **Queue Mass Forge Ability Grant**, **Queue Mass Forge Ability Revocation**, **Queue Active Mass Forge Ability Cancellation**, and **Queue Active Mass Forge Ability Interruption** all return the same receipt and complete through **On Ability Operation Request Completed**. Grant/Revoke/Activate/Cancel/Interrupt share one bounded FIFO, so caller order is preserved. Queued cancellation and interruption capture the current lifecycle ID; if that lifecycle ends or is replaced before execution, the completion explicitly reports `No Active Ability` or `Lifecycle Changed` and leaves the replacement untouched. Represented Actors can use **Queue Active Mass Forge Ability Cancellation from Actor** and **Queue Active Mass Forge Ability Interruption from Actor**.

For ordinary local graphs, use **Grant Mass Forge Ability (Async)**, **Revoke Mass Forge Ability (Async)**, and **Activate Mass Forge Ability (Async)**. They expose the same structured operation completion without a manually stored request ID.

Visible Mass-owned projectile

Use **Launch Mass Forge Entity Projectile** when both source and target are Mass entities and the projectile config already owns its optional Damage Definition. Supply the exact source and target handles, start/target locations, and speed. Damage is not applied on launch: the Mass projectile travels first, resolves at impact, and only then enqueues its authored damage.



This is the shipped Quick Start graph. Presentation reads Mass Forge projectile snapshots; authority remains in the projectile entity and its impact result, so replacing the visual renderer does not change combat.

Use **Launch Mass Forge Projectile (Async)** when the local Blueprint must await the exact terminal impact/expiry result. Use **Get Mass Forge Projectile Snapshots** to feed a project-owned Niagara system, HISM renderer, pooled Actor layer, or debugging visual.

Mass-to-player and player-to-Mass adapters

An ordinary player Actor deliberately keeps its project-owned health, GAS, inventory, and replication model. To receive Mass-authored effects without becoming a Mass entity:

1. Add **Mass Forge Effect Receiver Component** to the player Blueprint.
2. Bind **On Effect Received**.
3. Translate the supplied Instant Effect and fully resolved context into the player's own gameplay system.
4. Call **Apply Mass Forge Entity-to-Actor Effect** from the Mass source handle to the player.

For the reverse direction, call **Apply Mass Forge Actor-to-Entity Effect** or the richer **Apply Mass Forge Actor Damage To Entity** node. Use the Damage Definition when the Mass target needs the shared damage formula; use an Instant Effect for a direct authored transaction such as healing, resource restoration, or a fixed attribute/tag change.

The generic dispatch order is represented Mass entity first, Actor interface second, then exactly one receiver component. The receipt reports the selected route, resolved source/target context, receiver object, and exact rejection reason. It never guesses between multiple receiver components and never casts to a sample player class.

Multiplayer scope

Multiplayer is intentionally outside the supported initial-release workflow and no multiplayer example is included in the public launcher. Keep the Blueprint Quick Start local and authoritative. Networking source and an advanced reference remain in the plugin for evaluation, but their integration surface is experimental for this release and is not part of the advertised onboarding or support promise.

Common Blueprint node index

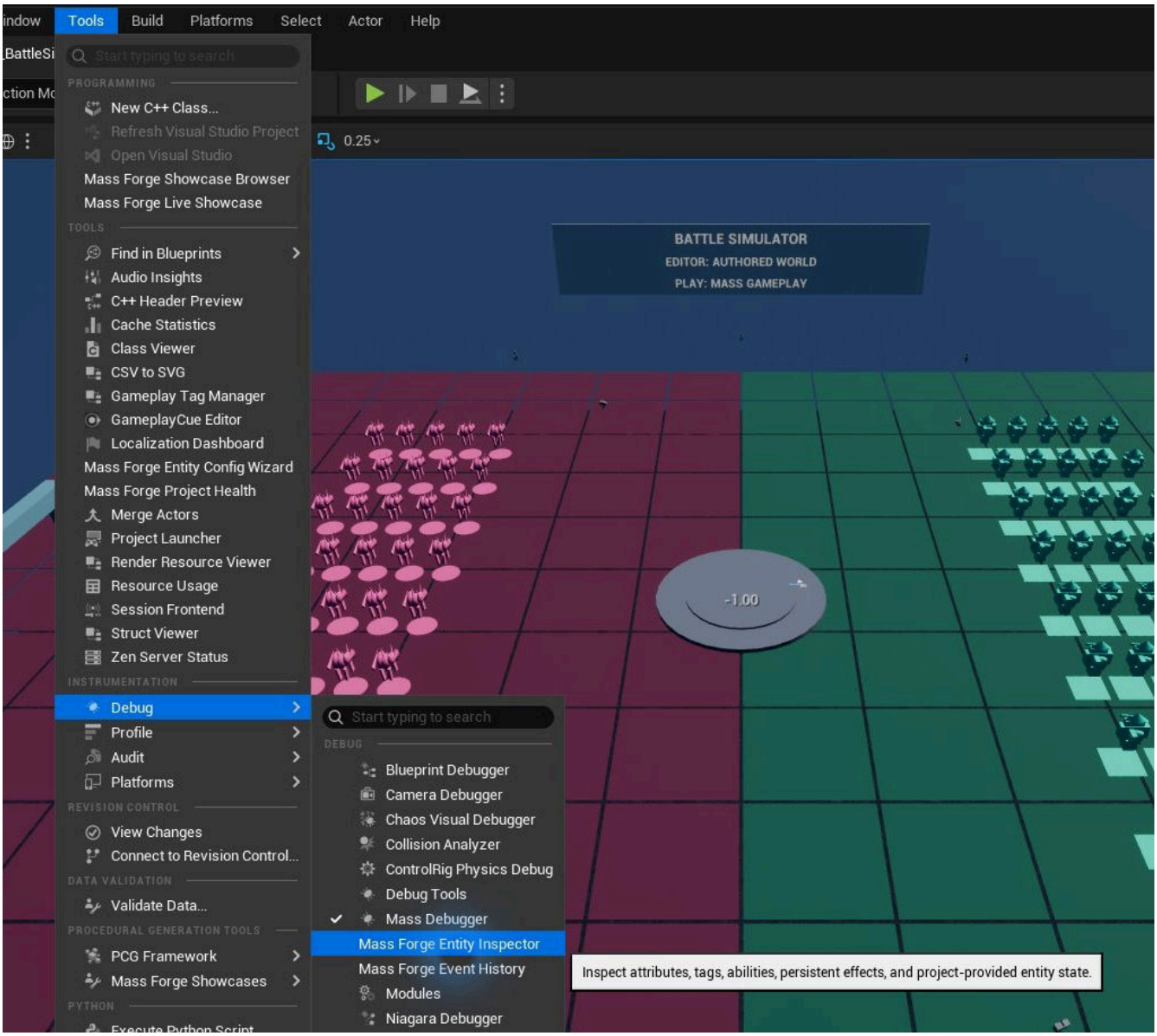
Goal	Recommended node
Spawn one placed actorless entity	**Spawn Mass Forge Entity**
Spawn a bounded transform batch	**Spawn Mass Forge Entities**
Validate or destroy a handle	**Valid MF Entity?** / **Destroy Mass Forge Entity**
Resolve a represented entity	**Get Mass Forge Entity From Actor**
Read one stat	**Get Mass Forge Attribute**
Set, add, or multiply one stat	**Change Mass Forge Attribute** or **Queue Mass Forge Attribute Change**
Await one safe stat mutation	**Change Mass Forge Attribute (Async)**
Restore saved attribute state	**Restore Mass Forge Attribute Snapshot** or **Queue Mass Forge Attribute Snapshot Restore**
Restore a durable combat checkpoint	**Restore Mass Forge Entity State** or **Queue Mass Forge Entity State Restore**
Capture a targeted active cast/channel	**Capture Mass Forge Entity State with Lifecycle Target Reference**
Restore a targeted active cast/channel	**Restore Mass Forge Entity State with Lifecycle Target** or **Queue Mass Forge Entity State Restore with Lifecycle Target**
Apply one reusable atomic effect	**Apply Mass Forge Instant Effect** or **Queue Mass Forge Instant Effect**
Await one reusable atomic effect	**Apply Mass Forge Instant Effect (Async)**
Await one complete damage transaction	**Apply Mass Forge Damage (Async)**
Grant an ability	**Grant Mass Forge Ability** or **Queue Mass Forge Ability Grant**
Revoke an ability	**Revoke Mass Forge Ability** or **Queue Mass Forge Ability Revocation**
Check an ability without side effects	**Can Activate Mass Forge Ability**
Activate an ability	**Activate Mass Forge Ability** or **Queue Mass Forge Ability Activation**
Await ability grant/revoke/activation	Corresponding **(Async)** ability node
Cancel an active cast/channel	**Cancel Active Mass Forge Ability** or **Queue Active Mass Forge Ability Cancellation**

Goal	Recommended node
Interrupt an active cast/channel	**Interrupt Active Mass Forge Ability** or **Queue Active Mass Forge Ability Interruption**
Remove an active timed/infinite effect	**Remove Mass Forge Persistent Effect** using its returned handle
Launch and await real projectile impact	**Launch Mass Forge Projectile (Async)**
Launch a common entity projectile without constructing nested structs	**Launch Mass Forge Entity Projectile**
Render Mass-owned projectiles	**Get Mass Forge Projectile Snapshots** into Niagara, HISM, or an Actor pool

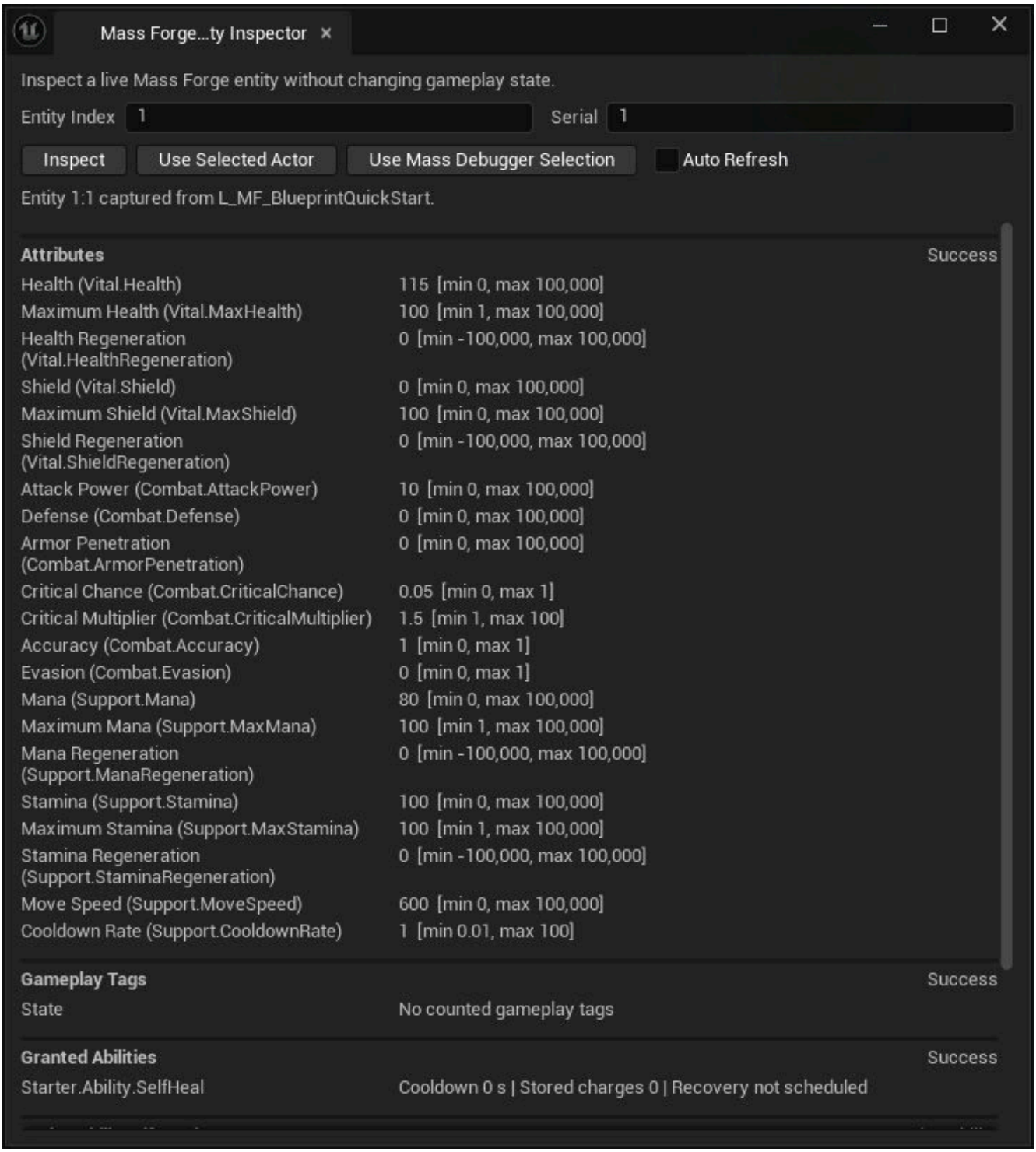
The subsystem getter nodes expose typed completion and lifecycle events when a queued operation or ongoing ability must be observed. Prefer the convenience library nodes for ordinary calls; use the subsystem directly for event binding and advanced orchestration.

Verification before building gameplay on top

1. Open **Tools > Mass Forge Project Health** and resolve every error.
2. Open **Tools > Debug > Mass Forge Entity Inspector** and confirm the live entity's Health, Mana, granted ability, cooldown, tags, and active effects.
3. Test the direct graph from a known Mass-idle path.
4. Test the queued graph and prove the request ID reaches exactly one matching completion.
5. Exercise invalid, stale, queue-full, insufficient-resource, cooldown, and dead-target branches rather than handling only success.
6. Run Unreal content validation after changing schemas or gameplay assets.

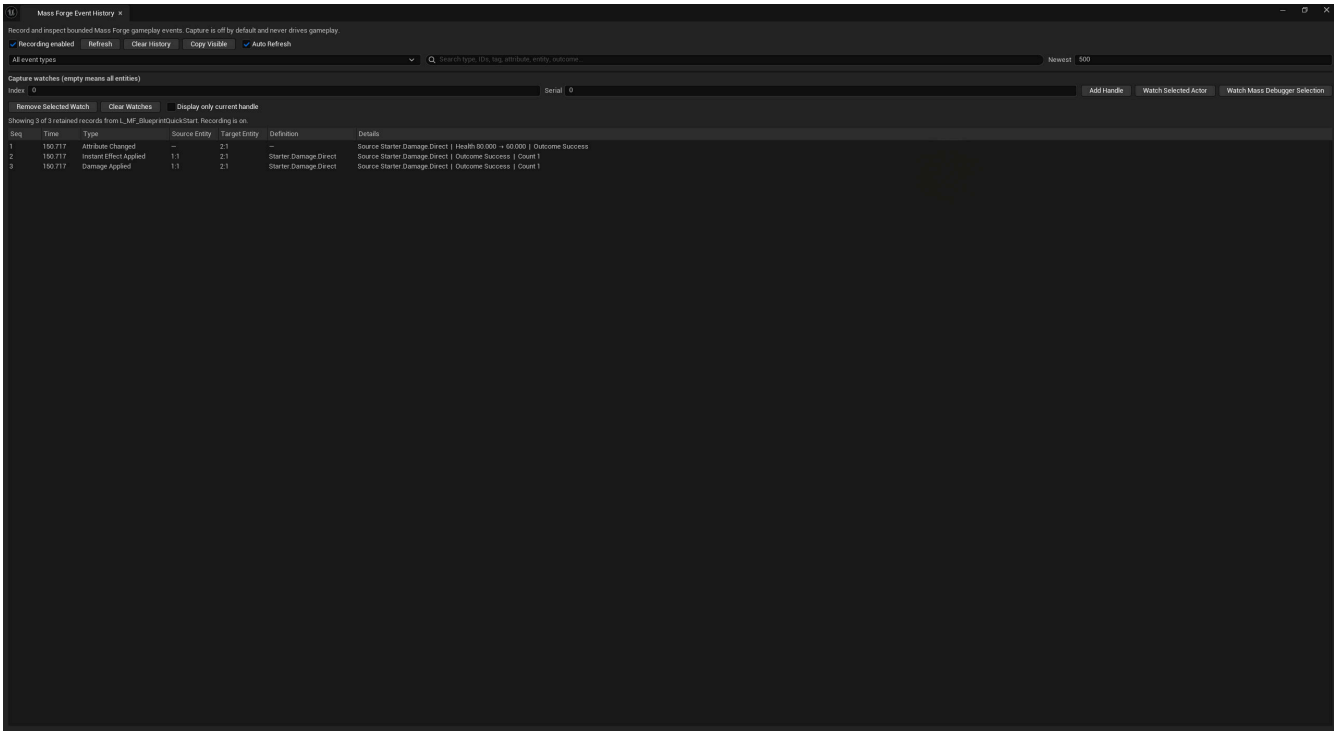


*In Unreal Engine 5.8, both diagnostic tabs live under **Tools > Debug**. Open the Entity Inspector after Play begins so it can attach to the active world.*



*Select an entity through the Mass Debugger or enter its complete index and serial, then press **Inspect**. The panel reads Mass Forge state without changing it.*

Project Health audits project-owned content and schema configuration before runtime testing. A green summary is the baseline; it does not replace exercising failure branches in the project's actual gameplay map.



Event History is an observation tool. Here one impact produced the expected attribute, effect, and damage records, making the input-to-impact-to-state-change chain inspectable without driving gameplay from the editor window.

Mass Forge's automation suite constructs real Blueprint pins for the public libraries and exercises real Mass worlds for attribute queues, atomic effects, damage, abilities, Actor receivers, stale handles, back-pressure, and callback re-entry. Project-specific graphs still need a playtest because their spawning, representation, player state, networking, and presentation remain project-owned.

Continue with Blueprint Async Actions, the Blueprint authoring guide, damage system guide, combat integration guide, abilities guide, and deferred command contract for the complete behavior behind these starter graphs.

GETTING STARTED

Blueprint recipe index

Use this page when you know the gameplay result you want but not which Mass Forge asset or node owns it. Every recipe below is available to Blueprint projects. The native high-volume loop is optional until a project needs to iterate very large populations every frame.

Start by opening `/MassForge/MassForge/L_MF_BlueprintQuickStart`. Its placed `BP_MF_BlueprintQuickStart__OPEN_ME` actor contains executable examples for entity creation, attributes, damage, healing, ability grant/activation, Mass projectiles, failure handling, and target respawn. The five public gameplay maps add Blueprint-owned startup graphs and editable definition references around their native high-volume simulation.

Choose the execution form

Situation	Use	Meaning
A known Mass-idle setup or ordinary game-thread input path	Direct node	Completes now and returns a detailed result.
A processor callback, timer, delegate, or path that may overlap Mass processing	Queue node	Returns a receipt now; match its request ID in the subsystem completion event.
A local Blueprint flow that wants one correlated completion	Async node	Wraps queue submission, request matching, and delegate cleanup in one latent-style action.

Queue acceptance is not gameplay success. Always inspect the later completion result. Never retry `Queue Full` in an unbounded Tick loop.

Create a new attribute

1. Open or create a **Mass Forge Vital/Combat/Support Attributes Schema**.
2. Add an entry with a stable name, default, minimum, maximum, and unique slot.
3. Save and resolve the schema editor diagnostics.
4. Assign the schema in **Project Settings > Mass Forge > Global Attribute Schemas**.
5. Add **Mass Forge Attributes Trait** to the Entity Config that needs the domain.
6. Select the new attribute from any schema-backed Attribute picker.

Use the Vital domain for Health-like state, Combat for formula inputs, and Support for movement/resources. These are storage domains, not a mandatory game taxonomy.

See Starter Attributes and Schema Editor.

Create an entity

Fast path:

1. Open **Tools > Mass Forge Entity Config Wizard**.

2. Choose a starter profile and a project-owned `/Game/...` destination.
3. Create the package; the wizard creates or reuses schemas and an ordinary Mass Entity Config.
4. Call **Spawn Mass Forge Entity** and keep the returned handle only from the **Success** execution pin.

For several placements, use **Spawn Mass Forge Entities** with a bounded transform array. Blueprint returns at most 10,000 handles per batch; use a native population/spawner path for larger allocations.

Read, decrease, or restore Health

Goal	Blueprint node	Important output
Read Health	**Get Mass Forge Attribute**	Use `Success`, then `Out Value`.
Add/subtract a raw amount	**Change Mass Forge Attribute**	Inspect old/new values and clamping.
Safely change during uncertain timing	**Queue Mass Forge Attribute Change**	Store the receipt request ID and match completion.
Reusable heal/resource transaction	**Apply Mass Forge Instant Effect**	Inspect the atomic effect result.
Restore saved values	**Restore Mass Forge Attribute Snapshot**	Use the queued form outside a guaranteed idle point.

Use a Damage Definition for attacks. A raw `Health - 10` is appropriate for a tutorial or explicit administrative rule, not for combat that needs mitigation, Shield, criticals, tags, death policy, or event context.

Damage in every supported direction

Source	Target	Recommended node
Mass entity	Mass entity	**Apply Mass Forge Entity Damage To Entity**
Actor/player	Mass entity	**Apply Mass Forge Actor Damage To Entity**
Mass entity	represented Mass Actor	Resolve the Actor's handle, then entity-to-entity damage
Mass entity	ordinary Actor/player	**Apply Mass Forge Entity-to-Actor Effect** plus the Effect Receiver component
Actor/player	represented Mass Actor	**Apply Mass Forge Actor Damage To Entity** after Actor-to-entity resolution

A **Mass Forge Damage Definition** owns the Health/Shield attributes, source Power, mitigation, critical rules, tag gates, death tag, and post-death policy. The returned damage result exposes every formula stage and whether death handling was scheduled.

See Damage and Actor and Entity Parity.

Create a reusable heal, buff, debuff, DoT, or HoT

- Use a **Mass Forge Instant Effect** for one atomic group of attribute/tag changes.
- Use a **Mass Forge Persistent Effect** for duration, period, stacking, reversible modifiers, DoT, or HoT.
- Store the returned persistent-effect handle when the game must remove or refresh the exact application.
- Use Effect Magnitude Calculations when a magnitude depends on source/target attributes or context.

The starter content includes heal, Mana restore, damage pulse, control-tag, slow, power buff, defense debuff, DoT, and HoT assets under `/MassForge/MassForge/BlueprintExamples/Data`.

Create a new ability

1. Create a **Mass Forge Ability** Data Asset.
2. Set a stable ID and choose Instant, Cast, or Channel execution.
3. Configure costs, cooldown/group cooldown, charges, conditions, targeting policy, and effects.
4. Grant it through the Entity Config's Ability Trait or **Grant Mass Forge Ability**.
5. Call **Can Activate Mass Forge Ability** for UI prediction when useful.
6. Call **Activate Mass Forge Ability** and handle its structured result.
7. Use cancel/interrupt nodes for active cast or channel lifecycles.

The shipped assets demonstrate self-heal, targeted strike, area burst, projectile cast, charged shot, channel drain, and interruptible cast.

See Abilities, Ability Lifecycles, and Ability Conditions.

Create a skillshot or projectile ability

1. Create a projectile-ready Mass Entity Config with **Mass Forge Projectile Trait**.
2. Create the Damage Definition or Effect that should resolve at impact.
3. Author the ability's costs, cooldown, cast/channel timing, and target requirements.
4. On the ability's release/presentation event, call **Launch Mass Forge Entity Projectile** with source/target handles, start/target positions, speed, arc, lifetime, and impact payload.
5. Render **Get Mass Forge Projectile Snapshots** with Niagara, HISM, or a bounded Actor pool.
6. Apply hit feedback only from projectile resolution. Do not deal damage at launch and then draw a cosmetic projectile afterward.

Use **Launch Mass Forge Projectile (Async)** when one Blueprint flow should await the matching impact/expiry. The Quick Start's left-click path is the smallest inspectable reference; the Horde and Battle samples show the high-volume presentation bridge.

See Mass Projectiles.

Create an attack

An attack is orchestration, not a separate opaque subsystem:

1. Select a target with a radius, segment, or policy query.
2. Check tags, range, resources, cooldown, and project rules.
3. Activate an ability or launch a projectile.
4. Resolve Damage/Effects at the release frame or projectile impact.
5. listen to gameplay events for animation, audio, UI, or analytics.
6. Reacquire when the handle becomes stale or the target dies.

Blueprint is suitable for player attacks and bounded groups. Put per-frame decisions for thousands of entities in a Mass processor or native fixed-step coordinator, while keeping the definitions and high-level orchestration in Blueprint/Data Assets.

Add factions, states, and targeting gates

Use counted gameplay tags for faction, alive/dead, control state, immunities, and temporary status. Add starting tags in the Entity Config, then use add/remove/query nodes at runtime. Damage Definitions, Effects, Abilities, and Targeting Policies can require or block tags without imposing a Mass Forge-specific faction taxonomy.

For same-faction friendly-fire prevention, add **Mass Forge Gameplay Tag Damage Policy Component** to one manager Actor or Game State and populate **Faction Tags** with your project-owned exact tags. For alliances, safe zones, or ownership rules, subclass **Mass Forge Damage Policy Component** and override **Evaluate Mass Forge Damage Policy**. Return **Abstain** when your rule does not apply so other bounded world policies can participate.

See Gameplay Tags, Combat and cross-ownership effects, Target Selection, and Targeting Policies.

Death, destruction, and respawn

A Damage Definition can leave the entity alive with a death tag, defer destruction, or destroy it according to its post-death policy. Treat the old handle as stale after destruction.

For respawn:

1. finish or cancel gameplay tied to the old handle;
2. destroy the exact generation-checked entity if it still exists;
3. spawn a new entity from the project-owned Entity Config;
4. replace every stored handle with the newly returned handle;
5. reacquire presentation and target relationships.

The Quick Start's **R** path demonstrates explicit destroy-then-spawn handle replacement.

Debug and prove the setup

1. Run **Mass Forge Project Health** before PIE and packaging.
2. Inspect a live entity in **Mass Forge Entity Inspector**.
3. Use **Gameplay Event History** to follow attribute, damage, effect, ability, and projectile outcomes.
4. In the sample maps, use **Mass Forge Live Showcase** to read allocated, active, simulated, represented, animated, and resolved counts separately.
5. Exercise invalid/stale handles, missing schema entries, insufficient cost, cooldown, queue full, and dead-target branches.

What remains native by design

Blueprint can author and call every ordinary gameplay operation above. Native C++ remains the intended path for:

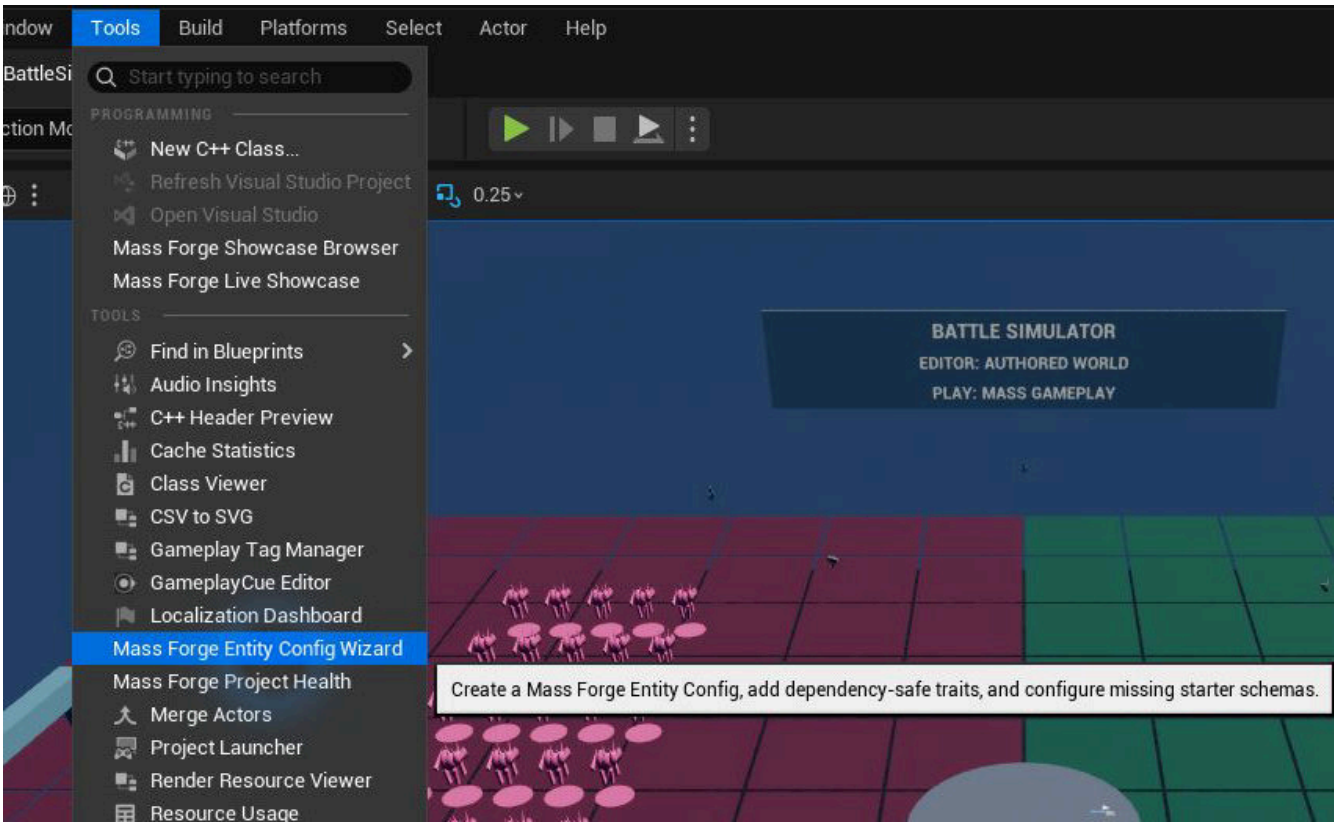
- per-frame iteration over very large entity populations;
- custom Mass processors, traits, fragments, and representation backends;
- bespoke SIMD/batched algorithms and project-specific memory layouts;
- allocations larger than the bounded Blueprint batch-return contract;
- the fixed-step high-volume loops inside the public showcase games.

That boundary is a performance and safety choice, not missing Blueprint gameplay parity. Continue with Blueprint Quick Start, Blueprint Authoring, or C++ Gameplay Quick Start.

GETTING STARTED

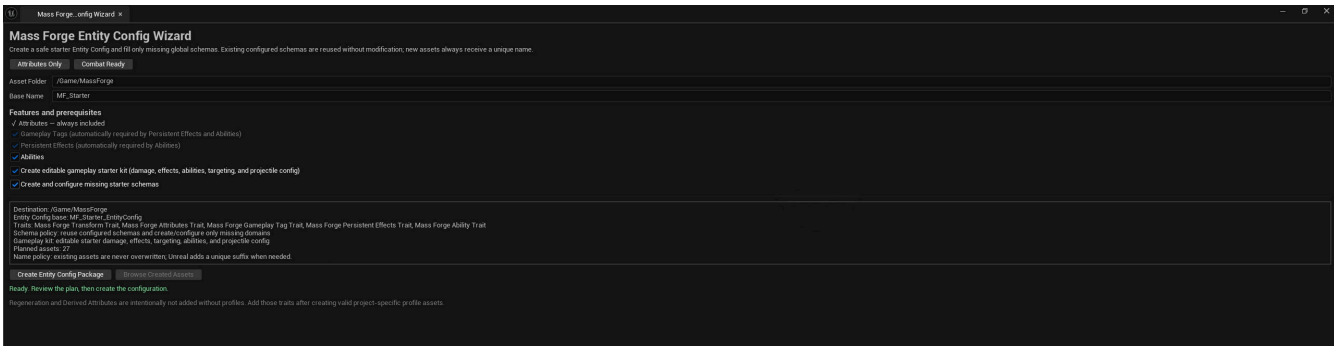
Entity Config wizard

The **Mass Forge Entity Config Wizard** creates a safe starting point for a project's Mass entities. Open it from **Tools > Mass Forge Entity Config Wizard**. It creates an ordinary Mass Entity Config, adds the selected Mass Forge traits, and can create and register only the global attribute schemas the project is missing.

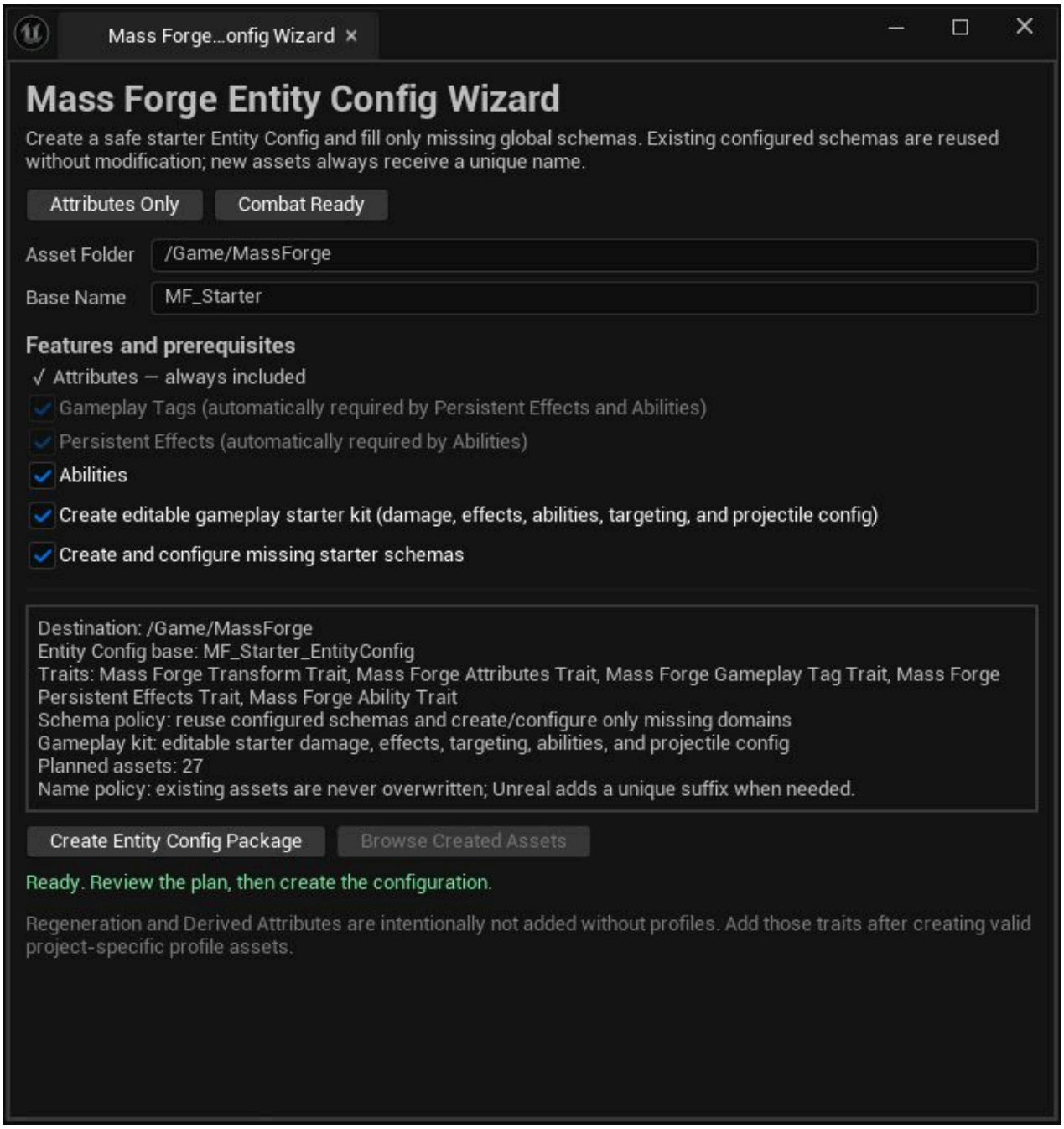


*The wizard is available from Unreal's standard **Tools** menu after the plugin has loaded.*

The wizard is an authoring shortcut, not a locked gameplay framework. Every generated schema and Entity Config remains normal editable Unreal content.



The live plan at the bottom is the exact package the wizard will create. Review it before selecting **Create Entity Config Package**; the wizard does not hide generated traits or assets behind a private framework.



*For the broad Blueprint starter path, choose **Combat Ready**, keep the starter kit and missing-schema options enabled, then review the destination and planned assets before creation.*

Choose a preset

- **Attributes Only** adds **Mass Forge Transform Trait** and **Mass Forge Attributes Trait**. Use it for projects that need Blueprint spawning, targeting, placement, named stats, and resource values.
- **Combat Ready** adds **Mass Forge Transform Trait**, **Mass Forge Attributes Trait**, **Mass Forge Gameplay Tag Trait**, **Mass Forge Persistent Effects Trait**, and **Mass Forge Ability Trait**. This is the broadest dependency-safe starter configuration.
- The individual feature checkboxes support configurations between those presets.

The **Combat Ready** preset also enables an editable gameplay starter kit: four damage formulas, four instant effects, four persistent effects, a reusable range policy, four representative abilities, and a Transform + Projectile Entity Config. Disable the starter-kit checkbox when only the base Entity Config and schemas are wanted.

Attributes are always present. Selecting Persistent Effects also selects Gameplay Tags. Selecting Abilities also selects Persistent Effects and Gameplay Tags. The dependency checkboxes show this closure and cannot be disabled while a selected feature requires them.

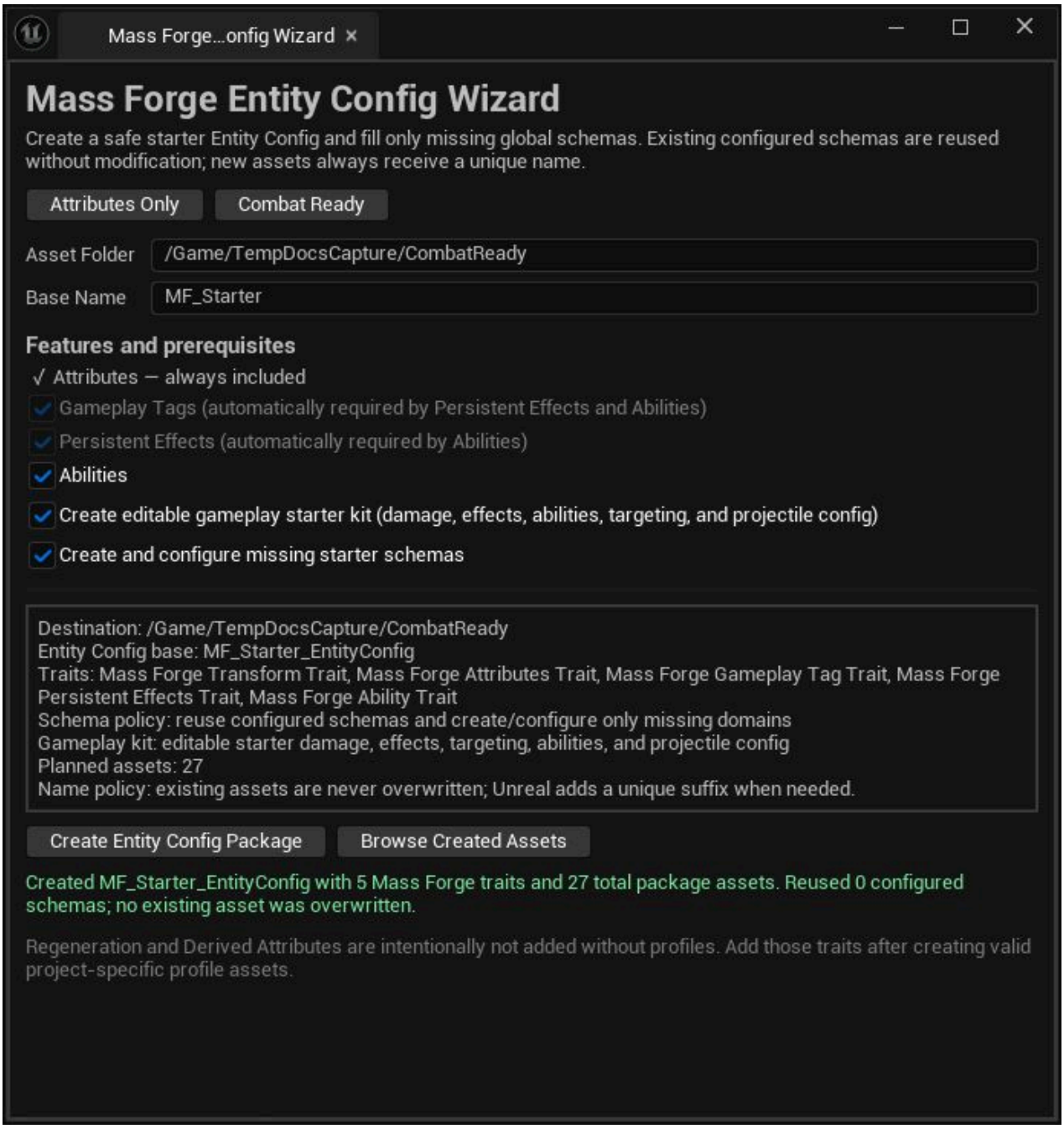
The Transform Trait adds Unreal's standard `FTransformFragment`; it does not introduce a competing movement system. The Blueprint **Spawn Mass Forge Entity/Entities** nodes require this fragment so a successful spawn always receives the requested world transform. Projects that create configs manually can add the same trait or supply the standard transform fragment through another project-owned trait.

Regeneration and Derived Attributes are deliberately excluded from both presets because those traits require valid project-specific profile assets. Create and validate the appropriate profile first, then add its trait to the generated Entity Config.

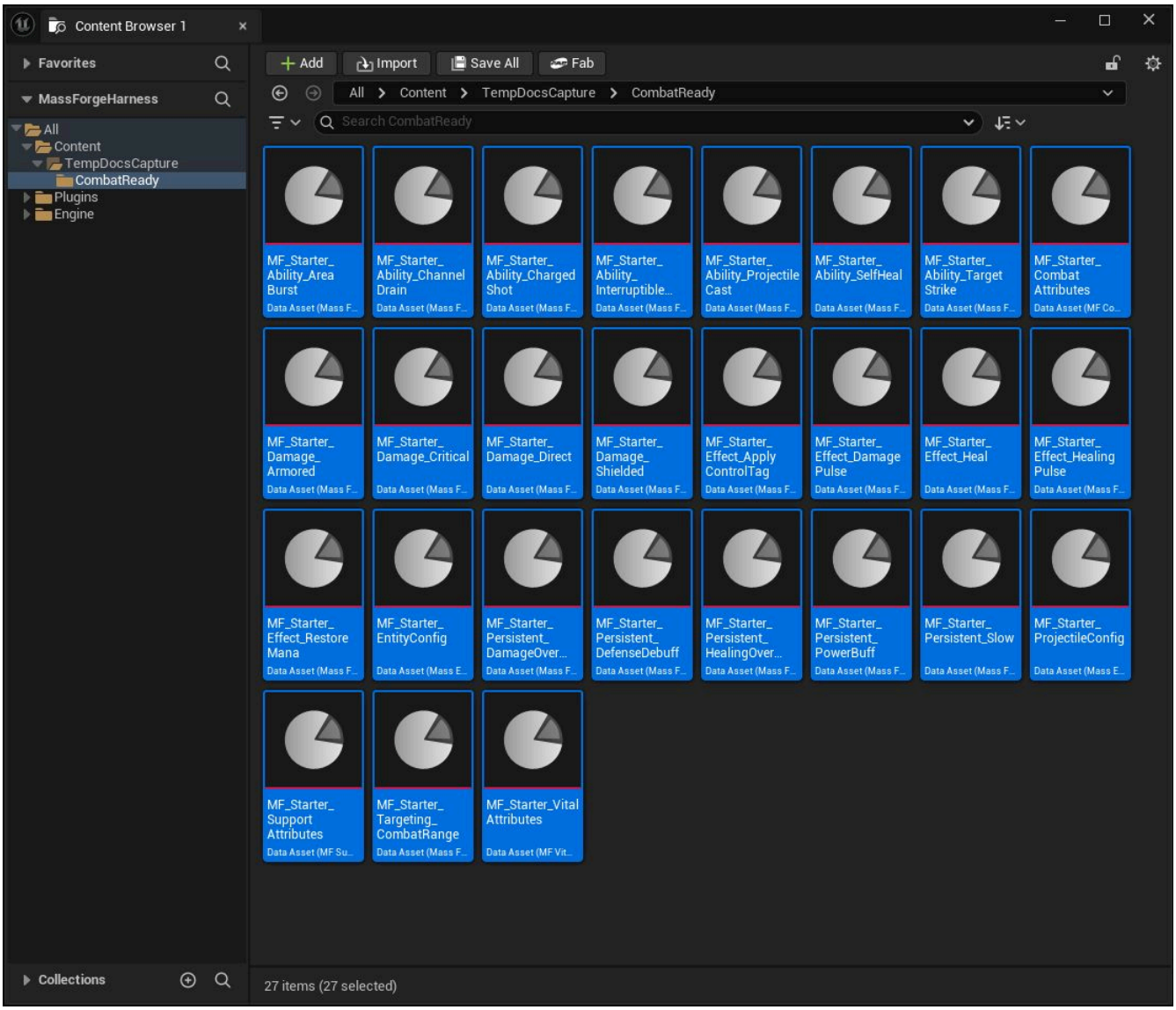
Create the package

1. Choose an **Asset Folder** beneath `/Game`. The default is `/Game/MassForge`.
2. Enter a **Base Name**. Invalid object-name characters are sanitized, and an empty result is rejected.
3. Choose a preset or enable the desired features.
4. Leave **Create editable gameplay starter kit** enabled for a complete Blueprint-ready combat baseline, or disable it for only schemas and an Entity Config.
5. Leave **Create and configure missing starter schemas** enabled for a first-time setup. Disable it when all three global schemas are already configured and the wizard should fail instead of creating missing content.
6. Review the live plan, then select **Create Entity Config Package**.

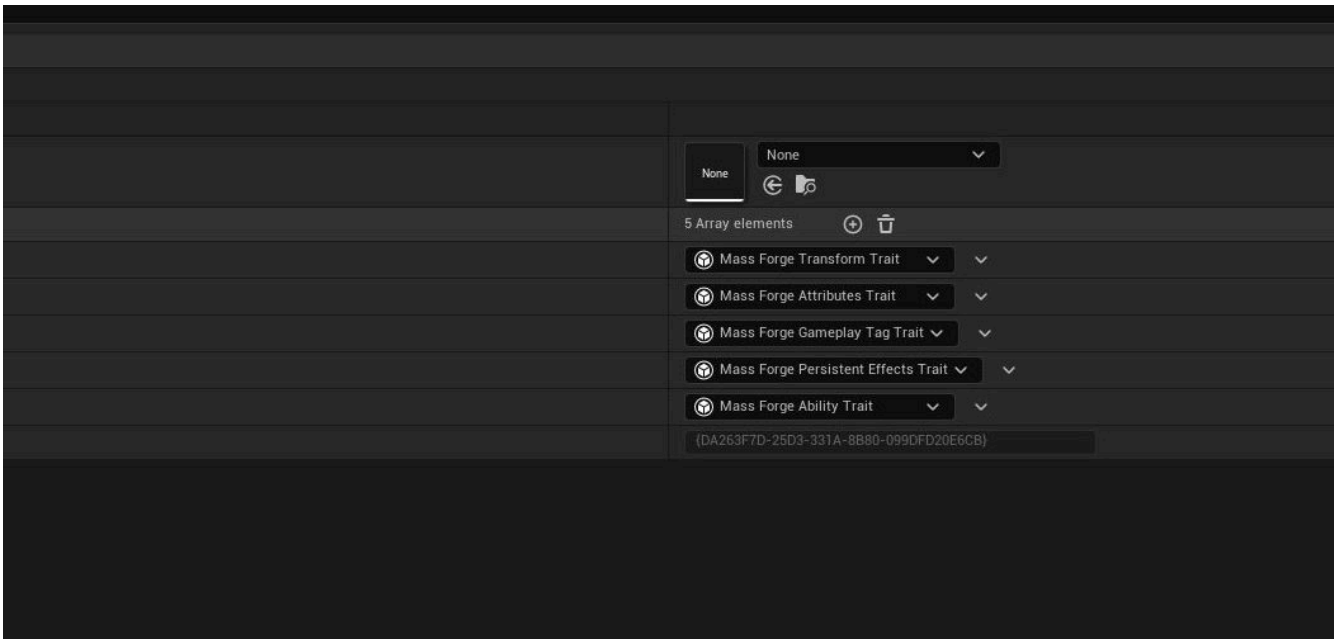
The wizard never overwrites an asset. When a requested name already exists, Unreal assigns a unique suffix. A successful run selects every newly created asset in the Content Browser and opens the new Entity Config.



Stop and read the result banner: it states what was created, how many assets were produced, whether schemas were reused, and that no existing asset was overwritten.



The generated package is visible project Content, not hidden runtime state. The selected assets include schemas, definitions, effects, abilities, targeting, and both entity configs.



Open the Entity Config and verify the trait closure. The standard Transform trait supports spawning and targeting; the four gameplay traits opt the entity into the Mass Forge systems selected by the preset.

Schema behavior

The project has one configured schema per Vital, Combat, and Support domain. The wizard resolves each domain independently:

- A configured schema is reused without changing its entries.
- A missing schema is created only when **Create and configure missing starter schemas** is enabled.
- Newly created schemas are saved and written to **Project Settings > Mass Forge > Global Attribute Schemas**.
- If creation of missing schemas is disabled, the wizard checks all three domains before creating any asset.

The starter entries are deliberately general-purpose templates:

- **Vital:** Health, MaxHealth, HealthRegeneration, Shield, MaxShield, ShieldRegeneration.
- **Combat:** AttackPower, Defense, ArmorPenetration, CriticalChance, CriticalMultiplier, Accuracy, Evasion.
- **Support:** Mana, MaxMana, ManaRegeneration, Stamina, MaxStamina, StaminaRegeneration, MoveSpeed, CooldownRate.

Every starter entry has a stable in-range slot, display metadata, a finite default, and an appropriate initial clamp policy. See the starter attributes guide for units, bounds, and a rename/migration example. These names are optional examples, not mandatory conventions or a hidden combat formula. Review them before authoring dependent abilities, effects, damage definitions, UI, or saved data.

Customization and persistence

Open a generated schema to use the **Mass Forge Schema Overview**. Add, remove, or replace starter attributes before the project ships. Once snapshots have been released, changing an attribute `Name` or moving it to another domain changes its persistent identity and requires a schema-version increment plus a migration asset. Slot repair preserves identity and does not require a migration.

See SCHEMA_EDITOR_GUIDE.md for validation and slot repair, and PERSISTENCE_GUIDE.md before changing released identities.

Failure and recovery

Invalid Content paths, empty base names, and disallowed missing schemas fail before asset creation. Later editor failures report whether asset creation, saving, or project-settings persistence failed. Successfully created assets are kept so work is not silently discarded; use **Browse Created Assets** to inspect them, correct source-control or save conditions, and either finish configuration manually or run the wizard again. A repeat run remains safe because names are uniqued and traits are added idempotently.

First verification

After generation:

1. Open each new schema and confirm its overview has no blocking diagnostics.

2. Review or replace the starter names, bounds, defaults, and descriptions.
3. Open the Entity Config and configure starting abilities or initial Gameplay Tags where needed.
4. Add validated Regeneration or Derived Attribute profiles only if the project uses those systems.
5. Spawn a test entity and use the **Mass Forge Entity Inspector** to confirm the expected fragments and initial values.
6. Open **Mass Forge Project Health**, rescan, and resolve every schema, reference, and migration error.
7. Run Unreal content validation before packaging a release.

GETTING STARTED

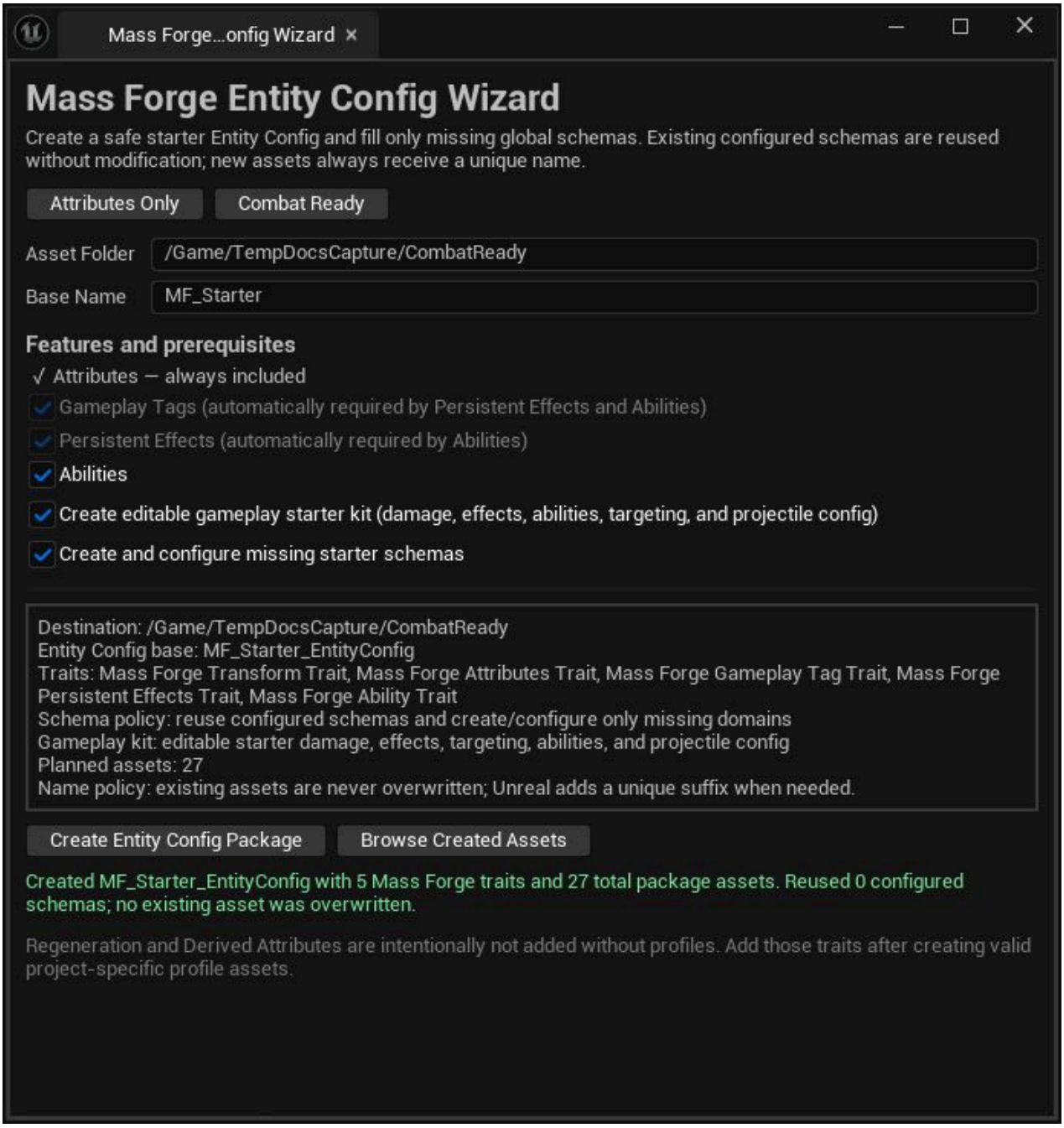
Gameplay starter kit

The Entity Config Wizard can create a complete editable starter package in project Content. The package is a teaching baseline, not a mandatory taxonomy: duplicate, rename, move, or replace its assets after assigning project-stable IDs.

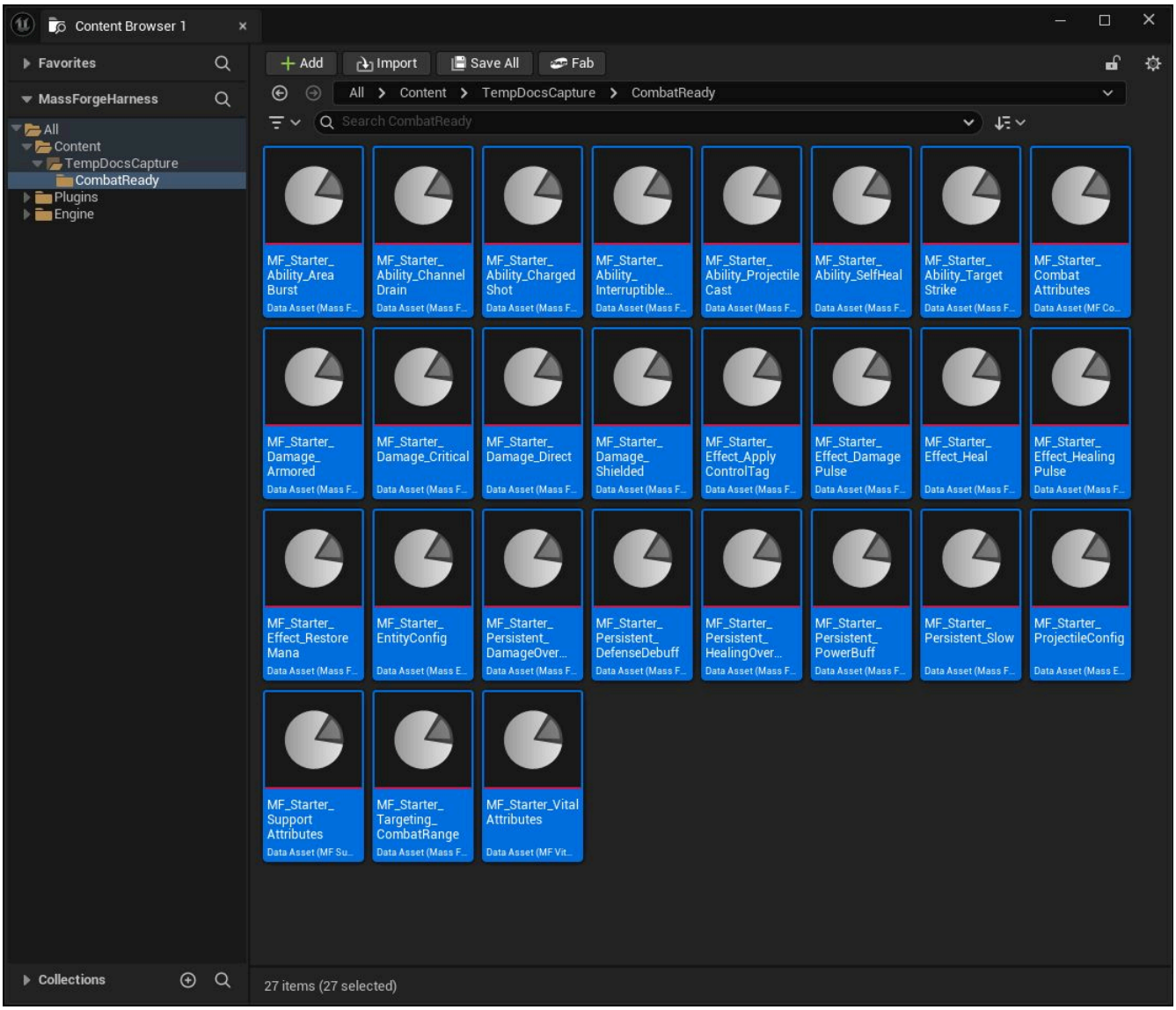
Create the package

1. Open **Tools > Mass Forge Entity Config Wizard**.
2. Choose **Combat Ready**.
3. Select a project folder such as `/Game/Combat/MassForge` and a unique base name.
4. Keep **Create editable gameplay starter kit** enabled.
5. Select **Create Entity Config Package**.
6. Open **Mass Forge Project Health**, scan, and resolve every error before using the assets in a release map.

The operation never overwrites existing assets. Unreal creates unique package names while the stable gameplay IDs remain visible for deliberate review.



A successful default Combat Ready run reports five traits and 27 package assets. If the counts or schema reuse differ, review the destination and configured project schemas before continuing.



The starter kit is not a sealed sample framework. Every selected item is an ordinary Unreal asset owned by the project and can be opened from this result view.

Generated assets

Group	Assets	What they teach
Schemas	Vital, Combat, Support	Stable attribute IDs, defaults, bounds, and domain ownership
Entity configs	Combat Ready Entity Config, Projectile Config	Required trait closure, standard transforms, and actorless projectile storage
Damage	Direct, Armored, Shielded, Critical	Final damage, armor ratio, shield routing, source power, and deterministic critical input
Instant effects	Damage Pulse, Healing Pulse, Heal, Restore Mana, Apply Control Tag	Atomic attribute/tag transactions and a replaceable control-tag example
Persistent effects	Damage Over Time, Healing Over Time, Power Buff, Slow, Defense Debuff	Periodic execution, refresh stacking, buffs, debuffs, and reversible Add/Multiply modifiers

Group	Assets	What they teach
Targeting	Combat Range	Actorless-safe 2D range admission using Mass transforms
Abilities	Self Heal, Target Strike, Projectile Skillshot Contract, Channelled Drain, Area Burst Payload, Charged Shot, Interruptible Cast	Self/target execution, area orchestration, costs, cooldowns, cast/channel timing, independent charges, targeting, interruption, and projectile handoff

The complete Combat Ready starter operation creates 27 editable assets. The generated examples use `MassForge.Sample.Status.Stunned` only as a visible teaching tag; replace it with the project's own control taxonomy before shipping.

Inspect the shipped Blueprint graph first

Enable **Show Plugin Content** in the Content Browser, then open:

`/MassForge/MassForge/BlueprintExamples/Blueprints/BP_MF_BlueprintQuickStart`

The matching saved learning map is intentionally easy to find at the plugin root:

`/MassForge/MassForge/L_MF_BlueprintQuickStart`

Its compiled Event Graph is deliberately linear and numbered. It spawns distinct source and target entities, stores both generation-safe handles, changes Health, applies entity-to-entity damage, applies healing, grants and activates an ability, and launches a real Mass-owned projectile with impact damage. A separate numbered Tick section drives only the small scene's visible projectile component from renderer-neutral snapshots. Every gameplay node references the editable assets under `/MassForge/MassForge/BlueprintExamples/Data`.

The map contains saved lighting, a camera, and the placed Blueprint teaching actor. The Blueprint's source/target meshes, teaching deck, and in-world labels are Construction Script components, so the lesson is visible and inspectable before PIE. Runtime code does not construct the world behind the user's back.

This graph is the complete inspectable Blueprint learning slice. It owns input, source/target spawning, visible failure diagnostics, Health display, damage, healing, ability grant/activation, impact-driven projectile presentation, first-death handling, and target replacement. No sample C++ class owns that tutorial's gameplay sequence. Copy the Blueprint and Data Assets into project Content before modifying them; plugin updates may replace the originals. The public Entity Config Wizard creates the same 27-asset teaching package in a project-owned `/Game` folder without overwriting existing content.

A blank project uses the three plugin starter schemas as replaceable defaults, so this map can run immediately. The first time the Entity Config Wizard creates a project-owned package, it treats those plugin teaching defaults as templates-not as the project's permanent schemas-and creates/registers editable `/Game` copies. Explicit project settings always take precedence.

For maintainers, the plugin-owned copy is reproducible rather than hand-maintained:

```
UnrealEditor-Cmd <Project>.uproject -run=MFGenerateBlueprintExamples -RebuildGeneratedAssets
```

`-RebuildGeneratedAssets` removes only the exact plugin-owned `/MassForge/MassForge/BlueprintExamples` generated root. It must not be used for project-owned copies.

Projectile skillshot contract

The generated Projectile Skillshot ability intentionally does not apply damage at cast completion. Its cast lifecycle defines admission, Mana cost, range, cooldown, and release timing. The Blueprint recipe launches a Mass Forge projectile when that lifecycle commits; the projectile carries the Damage Definition and Damage Request, and damage is committed only when the projectile impact resolves.

This separation prevents the common sample bug where Health changes immediately while a decorative projectile is still travelling.

Blueprint ownership

The generated Data Assets contain rules and stable IDs. The project Blueprint owns presentation and orchestration:

- keep the returned entity handle after **Spawn Mass Forge Entity**;
- grant and activate the selected Ability asset;
- observe cast/channel lifecycle events;
- launch the projectile with the generated Projectile Config;
- render projectile snapshots with Niagara, HISM, an Actor pool, or no renderer;
- update UI from structured results and subsystem events;
- react to kill/deactivation events according to the project's pooling policy.

Mass Forge owns bounded actorless state, validation, transactions, scheduling, and deterministic results. It does not silently invent project input, targeting visuals, faction rules, animation, or UI.

Safe modification order

1. Duplicate the generated package before experimenting with released stable IDs.
2. Adjust schema defaults and bounds before creating dependent save data.
3. Point effects and damage definitions at the intended attributes.
4. Configure ability costs, cooldowns, target policies, and payloads.
5. Connect projectile launch and presentation in Blueprint.
6. Run Project Health and native content validation.
7. Play the Blueprint Quick Start map and inspect the entity in the Entity Inspector.

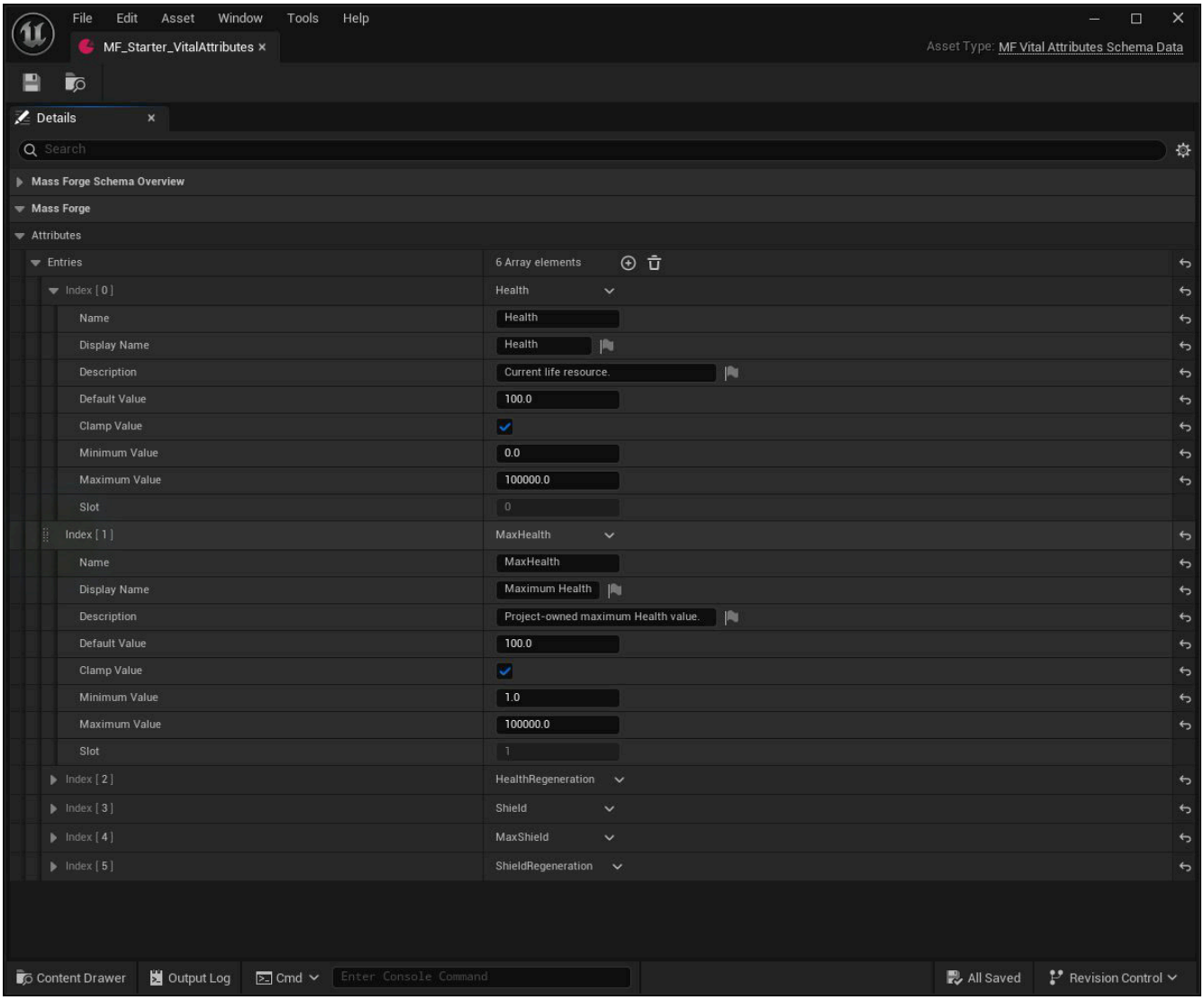
See Blueprint quick start, gameplay asset authoring, Mass projectiles, and entity lifecycle safety.

GETTING STARTED

Starter attribute pack

The Entity Config wizard can generate three project-owned schema assets containing a small, coherent authoring baseline. The pack demonstrates resource values, maxima, signed rates, normalized chances, raw ratings, and multipliers without making any name or formula mandatory.

Existing configured schemas are always reused without modification. The tables below apply only when the wizard creates a missing schema.



The starter pack is editable project Content. Use the shown defaults and bounds as a baseline, then change them to the units and balance of the game before saves ship.

Vital domain

Attribute	Meaning	Default	Initial bounds
Health	Current life resource	100	0-100,000
MaxHealth	Project-owned Health capacity	100	1-100,000
HealthRegeneration	Signed Health change per second	0	-100,000-100,000
Shield	Current absorbable protection resource	0	0-100,000
MaxShield	Project-owned Shield capacity	100	0-100,000
ShieldRegeneration	Signed Shield change per second	0	-100,000-100,000

Health and Shield are current resources. Their `Max` partners are ordinary attributes; Mass Forge does not silently bind a current value to its maximum. Use a Derived Attribute profile, effect, or project rule when a project wants that relationship.

The regeneration attributes are signed rates. A positive value can drive regeneration and a negative value can drive degeneration. A Regeneration Profile decides which target samples which rate and at what interval; merely owning a rate attribute does not activate processing.

Combat domain

Attribute	Meaning	Default	Initial bounds
AttackPower	Raw source power rating	10	0-100,000
Defense	Raw mitigation rating	0	0-100,000
ArmorPenetration	Raw penetration rating	0	0-100,000
CriticalChance	Normalized probability	0.05	0-1
CriticalMultiplier	Damage multiplier	1.5	1-100
Accuracy	Normalized project hit-chance input	1	0-1
Evasion	Normalized project avoidance input	0	0-1

Raw ratings do not imply a formula. A Damage Definition may use AttackPower and Defense directly, while the caller may derive request penetration from ArmorPenetration. Accuracy and Evasion are supplied as common normalized inputs for project targeting or policy code; the stock damage calculation does not roll hit chance.

CriticalChance is a fraction, not a percentage point value: `0.25` means 25%. CriticalMultiplier is a multiplier: `1.5` means 150% of pre-critical damage. Damage requests provide their deterministic critical roll explicitly.

Support domain

Attribute	Meaning	Default	Initial bounds
Mana	Current ability resource	100	0-100,000
MaxMana	Project-owned Mana capacity	100	1-100,000
ManaRegeneration	Signed Mana change per second	0	-100,000-100,000
Stamina	Current action resource	100	0-100,000
MaxStamina	Project-owned Stamina capacity	100	1-100,000
StaminaRegeneration	Signed Stamina change per second	0	-100,000-100,000
MoveSpeed	Project-owned movement-speed value	600	0-100,000
CooldownRate	Project-owned cooldown-speed multiplier	1	0.01-100

Mana and Stamina are current resources suitable for ability costs and Instant Effects. MoveSpeed intentionally has no assumed Unreal movement-unit conversion. CooldownRate uses `1` as its neutral multiplier, but Mass Forge does not automatically scale cooldowns from it; the project decides where and how that input participates.

Safe customization

Before the first shipped save, edit, replace, or remove any starter entry freely. Keep every name unique within its domain, preserve unique in-range slots, use finite defaults/bounds, and run the schema overview plus Project Health after changes.

After saves have shipped, an attribute's domain and name are persistent identity. Slot repair does not change identity, but a rename or domain move requires an explicit schema migration.

Rename example

Suppose released schema version 1 used `Vital.MaxHealth`, and version 2 should call it `Vital.HealthCapacity`:

1. Rename the schema entry to `HealthCapacity` while keeping a valid unique slot.
2. Create a **Mass Forge Attribute Snapshot Migration** with stable ID `Migration.Attributes.1To2`.
3. Set Source Schema Version to `1` and Target Schema Version to `2`.
4. Add one **Rename or Move** rule from `Vital.MaxHealth` to `Vital.HealthCapacity`.
5. Set **Snapshot Schema Version** to `2` and keep **Oldest Supported Snapshot Schema Version** at `1` while version-1 saves remain supported.
6. Run Project Health and confirm a complete migration path.
7. During load, call **Migrate Mass Forge Attribute Snapshot** before **Restore Mass Forge Attribute Snapshot**.

Migration changes identity without scaling or reinterpreting the saved float. If units change, perform the project-specific numeric conversion in the surrounding SaveGame upgrade before restore.

Suggested first uses

- Configure a Damage Definition with `Vital.Health`, optional `Vital.Shield`, `Combat.AttackPower`, and `Combat.Defense`.
- Configure a Regeneration Profile targeting `Vital.Health` and sampling `Vital.HealthRegeneration` with scale `1`.
- Author a self-heal ability that spends `Support.Mana` and applies a positive `Vital.Health` Instant Effect.
- Use Accuracy, Evasion, ArmorPenetration, MoveSpeed, and CooldownRate only in project rules that state their units and formula explicitly.

See the Blueprint-only quick start, schema editor guide, regeneration guide, damage guide, and persistence guide for the related workflows.

GETTING STARTED

Known issues and current limitations

This page describes the boundaries of the current `0.2.0` beta. Confirmed defects should be added with a tracking reference, workaround, affected versions, and removal version; none are currently recorded as unresolved release defects.

Current limitations

- **Platform support:** only Win64 is currently advertised and verified. Other targets have not completed the release matrix.
- **Engine support:** the verified source-compatible lines are Unreal Engine 5.6, 5.7, and 5.8. See ENGINE_VERSION_SUPPORT.md for exact patch baselines.
- **Networking:** multiplayer is outside the supported initial-release scope and no multiplayer example ships in the public launcher. Networking source remains available for evaluation, but its integration surface, performance, bandwidth behavior, prediction, hostile-input hardening, and upgrade compatibility are not advertised release promises.
- **Samples:** the five-map customer launcher, skeletal/VAT horde twins, original character pack, and automated/package gates pass on the advertised UE 5.6-5.8 matrix. Broader gameplay breadth, long-session soak, accessibility, and Marketplace promotional media remain outside the verified technical distribution gate.
- **Performance publication:** reference-hardware 10k/100k/1M simulation throughput plus a packaged 420-enemy VAT-horde frame/GPU/draw/primitive snapshot are published. The numbers are scoped separately. Long-session representative worst-frame capture, broader renderer settings/hardware, and worker-thread attribution remain open; no one-million-visible-enemies claim is made.
- **Editor language:** user-facing editor text is localization-ready through text macros where implemented, but the full localization and accessibility review remains open.
- **CI infrastructure:** the repository contains a valid three-engine GitLab pipeline, but the private project currently has no matching Windows Unreal Engine runners. Local verification remains authoritative until runners are attached.
- **Direct calls during Mass processing:** direct mutation/effect/damage/ability calls intentionally reject unsafe access. Use the matching queued operation from callbacks or code that may overlap Mass processing.

FAQ

Does Mass Forge replace Unreal Gameplay Ability System?

No. Mass Forge is a Mass-native attributes, effects, damage, tags, and abilities framework for actorless or represented entities. It does not require an Ability System Component or one UObject per entity. A project may still use GAS for players and Mass Forge for crowds, connecting them through the Actor/Entity effect and damage adapters.

Are Health, Mana, Armor, or other starter names mandatory?

No. The wizard's Vital, Combat, and Support entries are editable examples. Runtime behavior uses project-authored stable Attribute IDs, definitions, policies, and formulas. Rename or move a deployed attribute with an explicit snapshot migration after data has shipped.

Does every Mass entity need an Actor?

No. Attributes, effects, abilities, targeting, networking identity, persistence, and inspection work with generation-checked Mass handles. Actor adapters are optional integration paths for represented entities and ordinary player/gameplay Actors.

When should a Blueprint use the direct node or the queued node?

Use a direct node only when the entity manager is known to be idle. Use the matching queued node from callbacks, asynchronous systems, or code that may overlap Mass processing. A successful queued submission returns a positive request ID and exactly one completion unless its world tears down.

Does Mass Forge promise one million visible animated enemies?

No. The published 10k/100k/1M results measure actorless Mass Forge simulation paths and report their exact scope. Representation, rendering, navigation, collision, and VAT costs are separate. Any visible-crowd claim requires a packaged-game profile on representative hardware.

What survives save/load or travel?

Durable entity-state format 4 can preserve attributes, counted tags, grants, cooldowns, charge recovery, regeneration remainder, active Persistent Effects, and one cast/channel under documented remapping rules. Session references and world-local handles never enter SaveGame data.

How can a project display an entity's "mind"?

Register a bounded world-level Entity Inspection Provider and emit project-owned sections such as intent, target, task, or faction. The runtime snapshot and editor Entity Inspector display them without storing a UObject on every entity or making debug data simulation authority.

Troubleshooting flow

1. Run **Mass Forge Project Health** and resolve every error before testing runtime behavior.
2. Confirm the Actor resolves successfully or the Mass handle still has the exact index/serial generation and required traits.
3. Read the returned typed result and its nested result fields; use ERROR_AND_RESULT_CODES.md for the specific cause and remedy.
4. If the result is **Mass Is Processing**, switch the call to its bounded queued counterpart and correlate the positive request ID with the terminal completion event.
5. For save/load, retain the received format/schema versions, run the complete migration chain on a temporary copy, recreate population relationships, and restore only after all validation succeeds.
6. If the issue persists, reproduce it in the smallest project possible, run **Copy Support Report**, review it for private data, and attach the fields requested by SUPPORT_REQUEST_TEMPLATE.md.

Reporting a new defect

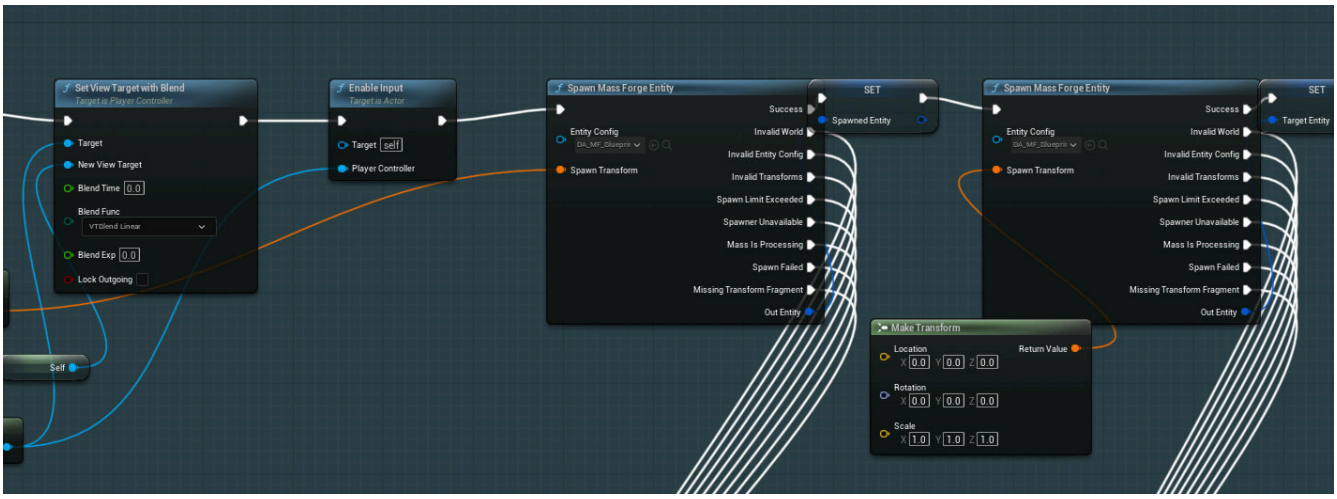
Run Project Health, copy the support report, and use SUPPORT_REQUEST_TEMPLATE.md. Do not paste passwords, access tokens, private source, complete saved games, or unrelated logs. A report that reproduces against an unmodified Mass Forge release is the fastest path to a regression test and fix.

BLUEPRINT AND C++ WORKFLOWS

Blueprint authoring guide

Mass Forge exposes the same runtime capabilities through context-sensitive Blueprint libraries and directly through its world subsystems. The convenience nodes are the recommended starting point: they obtain the correct subsystem from the hidden world-context pin and return the same detailed result structures as the lower-level API.

New users should begin with the Blueprint-only quick start, which connects installation, the Entity Config wizard, first attribute access, damage, healing, one ability, and player↔Mass adapters into one end-to-end walkthrough. When the desired gameplay result is known but the correct asset or node is not, use the task-oriented Blueprint recipe index.



Mass Forge Blueprint graphs use explicit success/failure execution pins and complete entity handles. Keep source and target variables distinct, and route failure pins instead of allowing an unset handle to continue.

Schema-backed Attribute ID picker

Every editable `Mass Forge Attribute ID` now uses one shared editor customization in Data Assets, Details panels, and unconnected Blueprint pins.

- The picker reads the currently configured Vital, Combat, and Support schemas.
- Options are searchable, grouped by domain, and sorted by stable runtime slot.
- Each option shows its `Domain.Name` identity; its tooltip includes the runtime slot and designer description.
- Only a unique name with one unique, in-range slot is selectable. Duplicate identities, slot collisions, missing names, and invalid slots are excluded instead of authoring a value that cannot resolve safely.
- A normal-color value is configured and unambiguous. Amber means the identity is empty. Red means its schema is missing or the identity is absent, ambiguous, or lacks one valid runtime slot. Hover the value for the exact reason.
- Expand the struct when raw Domain and Name fields are needed for migration work or intentionally incomplete authoring. Runtime validation remains authoritative.

The list is rebuilt whenever the menu opens, so saved schema changes do not require an editor restart. If no valid options appear, configure schemas under **Project Settings > Mass Forge > Global Attribute Schemas**, or use **Tools > Mass Forge Entity Config Wizard**.

Connected pins continue to accept any `Mass Forge Attribute ID` produced elsewhere in the graph. The picker improves literal defaults; it does not restrict dynamic IDs or change the runtime data layout.

Finding nodes

Search for **Mass Forge** to see the complete branded palette. Nodes are organized beneath these stable categories:

- `Mass Forge | Entity`
- `Mass Forge | Attributes`
- `Mass Forge | Effects`, `Effects | Actors`, and `Effects | Dispatch`
- `Mass Forge | Persistent Effects` and `Persistent Effects | Actors`
- `Mass Forge | Tags` and `Tags | Actors`
- `Mass Forge | Damage`, `Damage | Actors`, and `Damage Policy`
- `Mass Forge | Abilities`, `Abilities | Actors`, `Abilities | Lifecycle`, `Abilities | AI`, and `Abilities | Target Selection`
- `Mass Forge | Persistence` and `Persistence | Actors`
- `Mass Forge | Inspection`

Common synonyms such as `stat`, `skill`, `spell`, `buff`, `debuff`, `damage`, `heal`, `save`, `load`, `player`, `StateTree`, and `cooldown` are authored as search keywords. Context-sensitive search from an Actor, entity handle, effect asset, ability asset, snapshot, or subsystem narrows the available actions naturally.

Compact titles are used only for small predicates and conversions where pin meaning remains clear, such as **MF Entity**, **Has Ability?**, **Has Tag?**, **Tag Count**, and **Cooldown Left**. Transaction nodes retain their descriptive full titles.

Common workflow matrix

Goal	Direct node	Production-safe deferred or lifecycle path
Spawn or destroy an actorless entity	`Spawn Mass Forge Entity` / `Destroy Mass Forge Entity` while Mass is idle	Schedule the call at a project-owned safe boundary; use C++ population spawning for batches above the Blueprint limit.
Resolve a represented entity	`Get Mass Forge Entity from Actor` and branch on its explicit result	Use the handle only from `Success`; every later call revalidates its index and serial number.
Read one attribute	`Get Mass Forge Attribute`	Read while Mass is idle; processing overlap returns `Mass Is Processing`.
Set, add, or multiply a stat	`Change Mass Forge Attribute`	`Queue Mass Forge Attribute Change` plus the subsystem completion event.
Restore a migrated attribute snapshot	`Restore Mass Forge Attribute Snapshot` at a guaranteed Mass-idle point	`Queue Mass Forge Attribute Snapshot Restore` plus `On Attribute Snapshot Restore Request Completed`.
Save or restore a durable combat checkpoint	`Capture Mass Forge Entity State` / `Restore Mass Forge Entity State` at a guaranteed Mass-idle point	`Queue Mass Forge Entity State Restore` plus `On Entity State Restore Completed`; format 4 preserves regeneration timing, active Persistent Effects, and one active cast/channel with fresh handles. Use the target-reference/binding variants for an external lifecycle target.
Apply immediate damage, healing, tags, or resources	`Apply Mass Forge Instant Effect`	`Queue Mass Forge Instant Effect` plus the subsystem completion event.

Goal	Direct node	Production-safe deferred or lifecycle path
Route an effect across Actor/Mass ownership	The explicit Entity-to-Entity, Actor-to-Entity, Entity-to-Actor, or Actor-to-Actor node	Resolve the desired ownership route before entering Mass processing.
Apply the reusable damage pipeline	`Apply Mass Forge Damage` or an Entity/Actor convenience node	`Queue Mass Forge Damage` plus `On Damage Request Completed`.
Grant/revoke/start/stop an ability	Direct Grant, Revoke, Activate, Cancel, or Interrupt node	Matching queued operation plus `On Ability Operation Request Completed`; AI activation can use the pollable ticket nodes.
Observe a cast or channel	`Observe Mass Forge Ability Lifecycle`	The observer is presentation-only and cleans itself up when the lifecycle ends.
Inspect an entity	`Capture Mass Forge Entity Inspection`	Capture on demand; custom provider output is bounded.

Direct nodes fail before touching data when Mass is processing. Queued nodes return a request ID and apply bounded back-pressure. See DEFERRED_COMMANDS_GUIDE.md for the fixed cross-type execution phases and callback deferral rules.

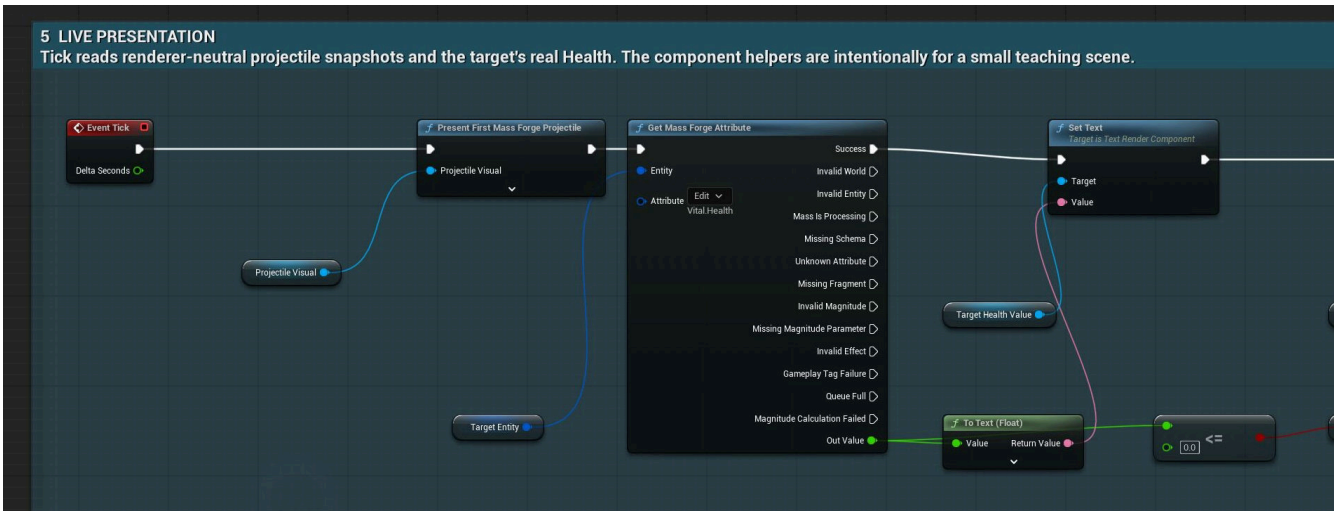
Actor and entity paths

Entity-handle nodes are the canonical actorless API. Every common entity operation now has a represented-Actor convenience counterpart across attributes, focused attribute snapshots, durable entity-state snapshots, Instant Effects, counted tags, Persistent Effects, ability ownership/state/lifecycle, damage targets, and inspection. Actor adapters resolve once and delegate to the canonical entity implementation, so results, transactions, queue limits, ordering, and generation safety stay identical. See the complete Actor and entity-handle parity guide for the node matrix and failure mapping.

An ordinary player Actor does not need to become a Mass entity to send effects or damage to Mass. Strict represented-Actor nodes reject ordinary Actors; cross-ownership dispatch and Actor-source damage nodes intentionally accept them where documented. Conversely, an ordinary Actor target can implement **Mass Forge Effect Receiver** or use **Mass Forge Effect Receiver Component** to accept a Mass-authored Instant Effect through project-owned player health logic. See COMBAT_INTEGRATION_GUIDE.md.

Pins and result handling

Optional diagnostic inputs-such as effect context, source labels, custom inspection sections, penetration, critical roll, and shield bypass-are collapsed under advanced pins on common nodes. Their defaults preserve the documented baseline behavior.



*Enumeration-result nodes expand outcomes into execution pins, so a graph can keep success readable without losing precise failure handling. The Quick Start routes the returned value into a display only from **Success**.*

Important operations return a result enum, receipt, or result structure. Do not discard it in production graphs:

1. Branch on the top-level result.
2. Keep queued request IDs until the matching completion event arrives.
3. Use the failed attribute, tag, policy, requirement, effect index, or targeting-policy fields for UI and diagnostics.
4. Treat queue acceptance as admission only; the request is revalidated at execution.
5. Never infer success from an unchanged value because clamping, zero magnitude, or an idempotent operation can succeed without a visible delta.

Enum-result enumeration nodes expose outcome execution pins. Larger atomic operations preserve their full result structure so Blueprints can inspect calculation and rollback detail without a second query.

Authoring checklist

- Global schemas are configured and pass the Schema Overview diagnostics.
- Literal Attribute ID pins are selected from the schema-backed picker.
- Red or amber Attribute IDs are resolved before asset validation and packaging.
- Direct calls are used only from Mass-idle lifecycle points.
- Queued request IDs are paired with their typed completion event.
- Actor adapters and entity-handle nodes are not mixed accidentally across worlds.
- Optional advanced pins are expanded only when their policy is intentional.
- Every important result is handled or logged with its detailed context.
- The project runs Unreal content validation after schema or gameplay-asset changes.

BLUEPRINT AND C++ WORKFLOWS

Blueprint async actions

Mass Forge's async Blueprint nodes combine bounded queue submission, request-ID correlation, and delegate cleanup into one latent-style node. Use them for ordinary gameplay events that may overlap Mass processing. The detailed direct and queue APIs remain available for advanced orchestration and high-volume C++ code.

Choose the right call style

Situation	Recommended API
A setup or editor-owned path that is guaranteed to run while Mass is idle	Direct node
An input event, timer, animation event, callback, or other ordinary Blueprint gameplay path	Async node
Several systems share one central completion listener	Queue node plus subsystem delegate
Thousands of homogeneous operations are produced in a Mass processor	C++ processor/deferred-command integration

Async nodes do not turn a rejected request into a success. They expose two explicit paths:

- **Completed** means the accepted request reached its terminal execution result. Inspect the returned result structure because execution can still reject a stale entity, invalid definition, or changed lifecycle.
- **Rejected** means no positive request ID was allocated. Typical reasons are an invalid world, invalid input, stale entity, or a full bounded queue. No later completion will arrive.

Calling **Cancel** stops this Blueprint object from listening. It does not retract gameplay work that was already accepted by a subsystem.

Attribute change

Use **Change Mass Forge Attribute (Async)** for a safe `Set`, `Add`, or `Multiply` operation. Select the schema-backed Attribute ID, supply the complete entity handle, and branch from:

- **Completed**: inspect `Result`, `Old Value`, `New Value`, and `Was Clamped`.
- **Rejected**: handle the submission result immediately.

For damage or healing that needs reusable authored rules, several atomic modifiers, source context, tags, or events, use a Damage Definition or Instant Effect instead of a raw attribute change.

Instant effect

Use **Apply Mass Forge Instant Effect (Async)** for reusable healing, resource restoration, and atomic attribute/tag transactions. The node retains the definition until submission and waits only for its own request ID. **Completed** returns the full Instant Effect application, including every modifier result.

Damage

Use **Apply Mass Forge Damage (Async)** with a Damage Definition and a complete Damage Request. Damage still resolves through the authored shield, mitigation, critical, tag, and death policy. **Completed** returns the same complete damage application used by C++; it is not a simplified boolean.

Ability ownership and activation

Use the operation-specific nodes:

- **Grant Mass Forge Ability (Async)**
- **Revoke Mass Forge Ability (Async)**
- **Activate Mass Forge Ability (Async)**

All three return a common `Mass Forge Ability Request Completion`. Inspect `Operation`, `Result`, `Ability Id`, entity handles, and the detailed activation payload where applicable. A successful activation request can start an authored cast or channel; observe its presentation phases with **Observe Mass Forge Ability Lifecycle**.

Persistent effects and tags

Use **Apply Mass Forge Persistent Effect (Async)** and **Remove Mass Forge Persistent Effect (Async)** for finite or infinite buffs, debuffs, damage over time, healing over time, and slows. The apply completion contains the exact stable handle required for a later removal.

Use **Add Mass Forge Gameplay Tag (Async)** and **Remove Mass Forge Gameplay Tag (Async)** for counted tag changes outside a larger atomic gameplay transaction. The completion reports previous and new counts and never permits underflow.

Projectile travel and impact

Use **Launch Mass Forge Projectile (Async)** when gameplay must wait for actual travel and resolution:

1. Build a `Mass Forge Projectile Launch Request` with the projectile Entity Config, start, target, speed, radius, lifetime, source/target entities, presentation channel, cue, and optional Damage Definition/request.
2. Use **Launched** to create or register presentation for the returned projectile ID and entity handle.
3. Read projectile snapshots each presentation update, or bridge them into project-owned Niagara, HISM, or Actor rendering.
4. Use **Resolved** for impact/expiry presentation. Damage, when configured, has already been resolved by the projectile impact phase and is included in the result.
5. Use **Rejected** when no projectile was created.

Do not apply the same damage at ability commit and again at projectile impact. A projectile skillshot's ability commit launches the projectile; the projectile owns impact-time damage.

Lifetime and safety contract

- Every action registers with the current Game Instance and releases all subsystem bindings when it completes, is cancelled, or is destroyed.
- Completions are matched by positive request or projectile ID, so unrelated world traffic cannot advance the graph.
- Inputs are copied or held strongly until queue submission. Subsystems retain their documented soft/weak definitions afterward.
- Entity handles remain generation-checked at execution time.
- Queue capacity and per-frame work limits are unchanged; async nodes are convenience wrappers, not an unbounded scheduling path.

For centralized request routing, back-pressure recovery, or C++ integration, continue with Deferred Commands. For visible skillshots, continue with Mass Projectiles.

BLUEPRINT AND C++ WORKFLOWS

Actor and entity-handle parity

Mass Forge stores gameplay state on Mass entities. `Mass Forge Entity Handle` is therefore the canonical, actorless API and remains valid for entities that have no Actor representation. Actor convenience nodes are strict adapters for represented Mass entities: they resolve the Actor's current index-and-serial handle once, then call the same canonical operation.

This keeps Blueprint graphs approachable without creating a second attribute, tag, effect, ability, or damage implementation.

Contract

1. An Actor convenience node first resolves the Actor in the world supplied by the hidden World Context pin.
2. On successful resolution, it delegates to the corresponding entity-handle node without changing validation, ordering, atomicity, clamping, stacking, costs, cooldowns, or queue limits.
3. A direct Actor call returns the same operation result and payload as the entity call on the resolved handle.
4. A queued Actor call resolves at submission time and queues the complete entity generation. A later representation change cannot redirect the request to another entity.
5. Failed Actor resolution performs no mutation. A rejected queued call has request ID `0` and promises no completion event.
6. Actor adapters do not make representation authoritative. Removing or replacing an Actor representation does not remove or transfer Mass Forge gameplay state.

Call **Get Mass Forge Entity from Actor** first when a graph needs the detailed resolution distinction between Invalid Actor, Different World, Representation Unavailable, Not Represented, and Stale Representation. Convenience nodes map those failures into their operation family as documented below.

Attributes, snapshots, and Instant Effects

Entity-handle node	Represented-Actor node
`Get Mass Forge Attribute`	`Get Mass Forge Attribute from Actor`
`Change Mass Forge Attribute`	`Change Mass Forge Attribute for Actor`
`Queue Mass Forge Attribute Change`	`Queue Mass Forge Attribute Change for Actor`
`Capture Mass Forge Attribute Snapshot`	`Capture Mass Forge Attribute Snapshot from Actor`
`Restore Mass Forge Attribute Snapshot`	`Restore Mass Forge Attribute Snapshot to Actor`
`Queue Mass Forge Attribute Snapshot Restore`	`Queue Mass Forge Attribute Snapshot Restore to Actor`
`Apply Mass Forge Instant Effect`	`Apply Mass Forge Instant Effect to Represented Actor`
`Apply Mass Forge Instant Effect with Context`	`Apply Mass Forge Instant Effect with Context to Represented Actor`
`Queue Mass Forge Instant Effect`	`Queue Mass Forge Instant Effect to Represented Actor`

Complete entity-state persistence

Entity-handle node	Represented-Actor node
`Capture Mass Forge Entity State`	`Capture Mass Forge Entity State from Actor`
`Capture Mass Forge Entity State with Lifecycle Target Reference`	`Capture Mass Forge Entity State from Actor with Lifecycle Target Reference`
`Restore Mass Forge Entity State`	`Restore Mass Forge Entity State to Actor`
`Restore Mass Forge Entity State with Lifecycle Target`	`Restore Mass Forge Entity State to Actor with Lifecycle Target`
`Queue Mass Forge Entity State Restore`	`Queue Mass Forge Entity State Restore to Actor`
`Queue Mass Forge Entity State Restore with Lifecycle Target`	`Queue Mass Forge Entity State Restore to Actor with Lifecycle Target`

These adapters preserve the same atomic attributes/tags/ability-grant/cooldown/charge/effect/lifecycle snapshot semantics, copied queued payload and target binding, generation revalidation, and terminal completion contract as the canonical entity operations. The represented Actor identifies the entity whose state is restored; an external lifecycle target remains an explicit project-resolved entity binding and may be actorless.

The represented-Actor Instant Effect nodes are deliberately strict. **Dispatch Mass Forge Instant Effect to Actor** is a different ownership-boundary adapter: it prefers a represented Mass entity but may instead call one ordinary Actor receiver. Use strict nodes when the target must own Mass Forge state; use Dispatch when a non-Mass player or gameplay Actor is a supported target.

Counted Gameplay Tags

Entity-handle node	Represented-Actor node
`Add Mass Forge Gameplay Tag`	`Add Mass Forge Gameplay Tag to Actor`
`Remove Mass Forge Gameplay Tag`	`Remove Mass Forge Gameplay Tag from Actor`
`Queue Add Mass Forge Gameplay Tag`	`Queue Add Mass Forge Gameplay Tag to Actor`
`Queue Remove Mass Forge Gameplay Tag`	`Queue Remove Mass Forge Gameplay Tag from Actor`
`Has Mass Forge Gameplay Tag`	`Has Mass Forge Gameplay Tag on Actor`
`Get Mass Forge Gameplay Tag Count`	`Get Mass Forge Gameplay Tag Count from Actor`
`Query Mass Forge Gameplay Tag`	`Query Mass Forge Gameplay Tag on Actor`
`Get Owned Mass Forge Gameplay Tags`	`Get Owned Mass Forge Gameplay Tags from Actor`

Persistent Effects

Entity-handle node	Represented-Actor node
`Apply Mass Forge Persistent Effect`	`Apply Mass Forge Persistent Effect to Actor`

Entity-handle node	Represented-Actor node
`Queue Mass Forge Persistent Effect`	`Queue Mass Forge Persistent Effect to Actor`
`Get Active Mass Forge Persistent Effects`	`Get Active Mass Forge Persistent Effects from Actor`

Removal and single-instance queries use `Mass Forge Persistent Effect Handle`, which already contains the owning entity generation. They do not require an Actor variant.

Abilities

Entity-handle node	Represented-Actor node
`Grant Mass Forge Ability`	`Grant Mass Forge Ability to Actor`
`Revoke Mass Forge Ability`	`Revoke Mass Forge Ability from Actor`
`Has Mass Forge Ability`	`Has Mass Forge Ability for Actor`
`Get Mass Forge Ability Charge State`	`Get Mass Forge Ability Charge State from Actor`
`Get Mass Forge Cooldown Group Remaining`	`Get Mass Forge Cooldown Group Remaining from Actor`
`Get Granted Mass Forge Abilities`	`Get Granted Mass Forge Abilities from Actor`
`Get Active Mass Forge Cooldown Groups`	`Get Active Mass Forge Cooldown Groups from Actor`
`Queue Mass Forge Ability Grant`	`Queue Mass Forge Ability Grant to Actor`
`Queue Mass Forge Ability Revocation`	`Queue Mass Forge Ability Revocation from Actor`
`Can Activate Mass Forge Ability`	`Can Activate Mass Forge Ability from Actors`
`Activate Mass Forge Ability`	`Activate Mass Forge Ability from Actors`
`Queue Mass Forge Ability Activation`	`Queue Mass Forge Ability Activation from Actors`
`Request Mass Forge Ability Activation for AI`	`Request Mass Forge Ability Activation for AI from Actors`
`Get Active Mass Forge Ability Lifecycle`	`Get Active Mass Forge Ability Lifecycle from Actor`
`Cancel Active Mass Forge Ability`	`Cancel Active Mass Forge Ability from Actor`
`Interrupt Active Mass Forge Ability`	`Interrupt Active Mass Forge Ability from Actor`
`Queue Active Mass Forge Ability Cancellation`	`Queue Active Mass Forge Ability Cancellation from Actor`
`Queue Active Mass Forge Ability Interruption`	`Queue Active Mass Forge Ability Interruption from Actor`

AI request polling, cancellation, and forgetting operate on a request ID rather than an entity, so their semantics are already representation-independent.

Damage

Damage Definitions always resolve against a Mass target. The convenience surface supports both endpoint forms and the same four combinations for direct and queued work:

Source	Target	Direct node	Queued node
Entity	Entity	`Apply Mass Forge Entity Damage To Entity`	`Queue Mass Forge Entity Damage To Entity`
Actor	Entity	`Apply Mass Forge Actor Damage To Entity`	`Queue Mass Forge Actor Damage To Entity`
Entity	represented Actor	`Apply Mass Forge Entity Damage To Actor`	`Queue Mass Forge Entity Damage To Actor`
Actor	represented Actor	`Apply Mass Forge Actor Damage To Actor`	`Queue Mass Forge Actor Damage To Actor`

An Actor source may be ordinary or represented. When represented, its entity handle is included so source-attribute formulas can use it. An ordinary Actor remains valid for base-damage-only requests and stays available in the Effect Context. An Actor target must represent a live Mass entity; use the generic Effect Receiver adapter when damaging a non-Mass player whose health belongs to another system.

`Apply Mass Forge Damage` and `Queue Mass Forge Damage` remain the explicit request-structure nodes for advanced callers. The convenience nodes build that same request and do not introduce another damage pipeline.

Inspection and target selection

`Capture Mass Forge Entity Inspection` and `Capture Mass Forge Actor Inspection` read the same snapshot provider. Ability target selection returns canonical handles and optional represented Actors together; collision-hit conversion resolves to the same generation-checked handle used by actorless selection.

Failure mapping

Actor resolution outcome	Attribute/effect family	Snapshot family	Tag family	Persistent family	Ability family	Damage target
Invalid World	`Invalid World`	`Invalid World`	`Invalid World`	`Invalid World`	`Invalid World`	`Invalid World`
Invalid Actor, Different World, Representation Unavailable, Not Represented, or Stale Representation	`Invalid Entity`	`Invalid Entity`	`Invalid Entity`	`Invalid Entity`	source: `Invalid Entity`; target: `Invalid Target`	`Invalid Target`

Pure convenience predicates return `false`, numeric convenience queries return `0`, and failed array queries clear their output. Use the explicit resolver before the operation when the graph must display the more detailed representation cause.

Blueprint usage rule

- Use entity-handle nodes in Mass processors, actorless AI, saved runtime references, high-volume systems, and reusable gameplay libraries.
- Use Actor convenience nodes in Actor-owned UI, overlap/hit callbacks, Character or Controller graphs, and low-frequency presentation code.
- Use a queued variant whenever the call can occur during Mass processing. Store the positive request ID and correlate it with the matching completion event.
- Never retain an Actor-derived handle across world replacement. Resolve it again in the new world.

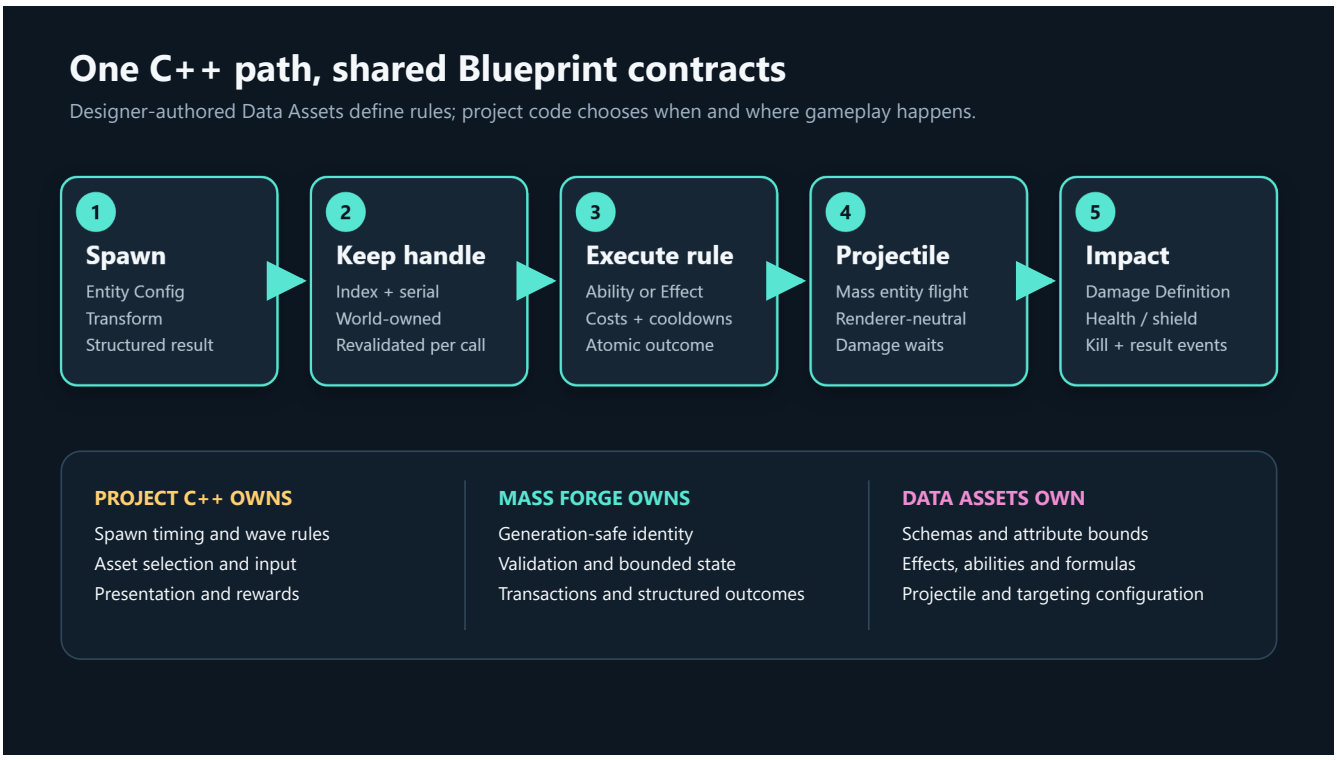
The runtime world automation test exercises both paths against the same represented entities, including direct and queued attributes, snapshots, Instant Effects, tags, Persistent Effects, ability ownership/state, every damage endpoint combination, and rejection of unrepresented Actors.

BLUEPRINT AND C++ WORKFLOWS

C++ gameplay quick start

Mass Forge exposes the same supported gameplay path to C++ and Blueprint. The public Blueprint function libraries are ordinary public C++ APIs, so a project module can call them directly and receive the same generation-safe handles and structured outcomes shown in the Blueprint tutorials.

This page uses designer-authored Data Assets for configuration and C++ for orchestration. It does not construct schemas, abilities, or damage formulas in code.



Read this left to right before copying code. Project C++ owns encounter rules and presentation, editable Data Assets own configuration, and Mass Forge owns generation-safe identity, validation, bounded state, and structured transactions.

Module setup

Add the plugin modules used by the project to the project's `Build.cs`:

```
PublicDependencyModuleNames.AddRange(new[]
{
    "MassForge",
    "MassEntity",
    "MassSpawner"
});
```

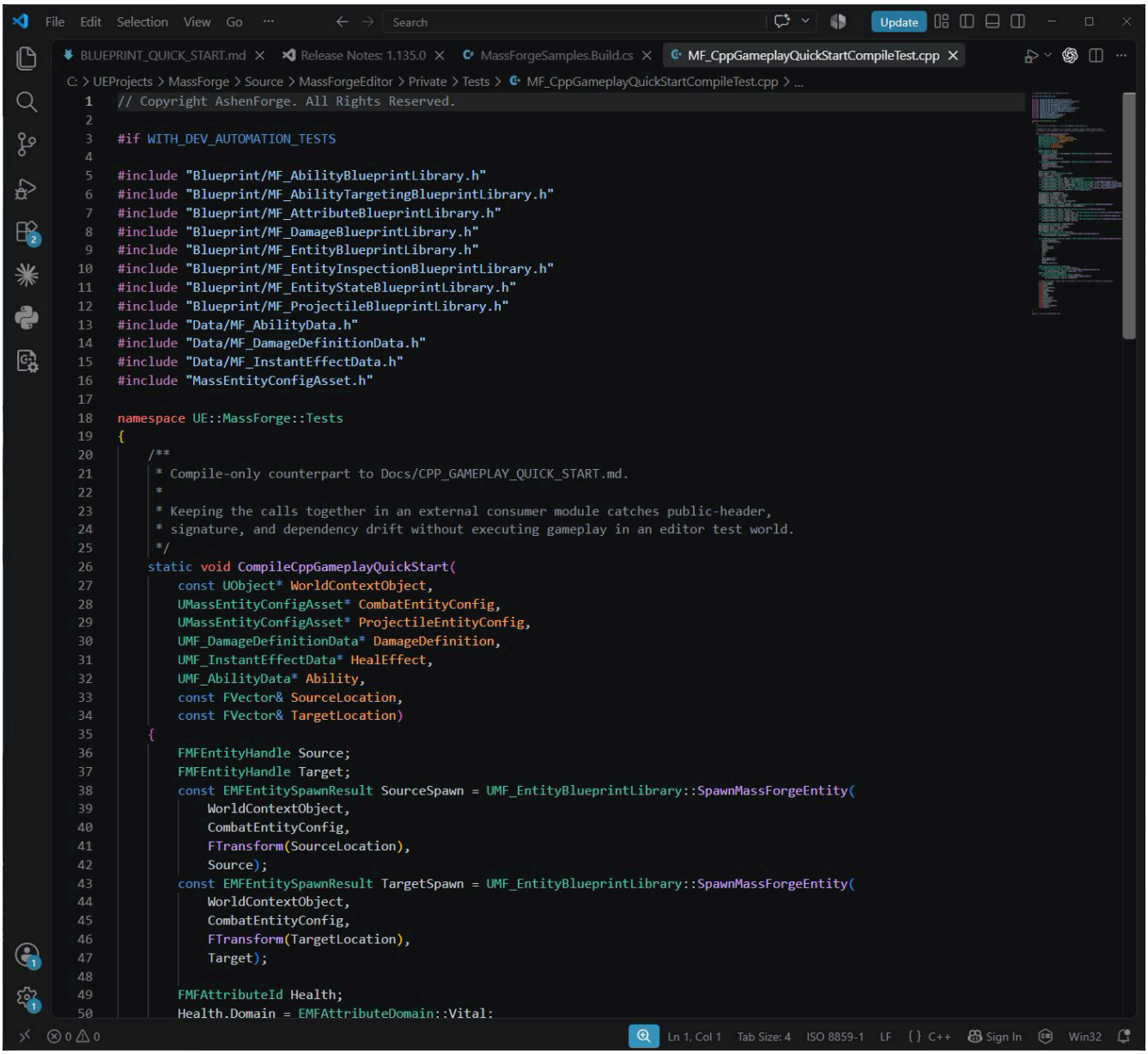
```
using UnrealBuildTool;

1 reference
public class MassForgeSamples : ModuleRules
{
    0 references
    public MassForgeSamples(ReadOnlyTargetRules Target) : base(Target)
    {
        PCHUsage = PCHUsageMode.UseExplicitOrSharedPCHs;

        PublicDependencyModuleNames.AddRange(new[]
        {
            "Core",
            "CoreUObject",
            "DeveloperSettings",
            "Engine",
            "EnhancedInput",
            "GameplayTags",
            "InputCore",
            "MassCommon",
            "MassEntity",
            "MassForge",
            "MassSpawner",
            "Niagara",
            "Slate",
            "SlateCore",
            "UMG"
        });

        if (Target.Version.MajorVersion > 5 ||
            (Target.Version.MajorVersion == 5 && Target.Version.MinorVersion >= 8))
        {
            PublicDependencyModuleNames.Add("MassCore");
        }

    }
}
```

Add the modules to the consumer module that owns the gameplay code. `MassForge` exposes the supported gameplay API, while Unreal's `MassEntity` and `MassSpawner` provide the native entity configuration and spawn types used by the example.

Include only public headers. A gameplay class that follows the complete example below needs:

```
#include "Blueprint/MF_AbilityBlueprintLibrary.h"
#include "Blueprint/MF_AbilityTargetingBlueprintLibrary.h"
#include "Blueprint/MF_AttributeBlueprintLibrary.h"
#include "Blueprint/MF_DamageBlueprintLibrary.h"
#include "Blueprint/MF_EntityBlueprintLibrary.h"
#include "Blueprint/MF_EntityInspectionBlueprintLibrary.h"
#include "Blueprint/MF_EntityStateBlueprintLibrary.h"
#include "Blueprint/MF_ProjectileBlueprintLibrary.h"
#include "Data/MF_AbilityData.h"
#include "Data/MF_DamageDefinitionData.h"
#include "Data/MF_InstantEffectData.h"
#include "MassEntityConfigAsset.h"
```

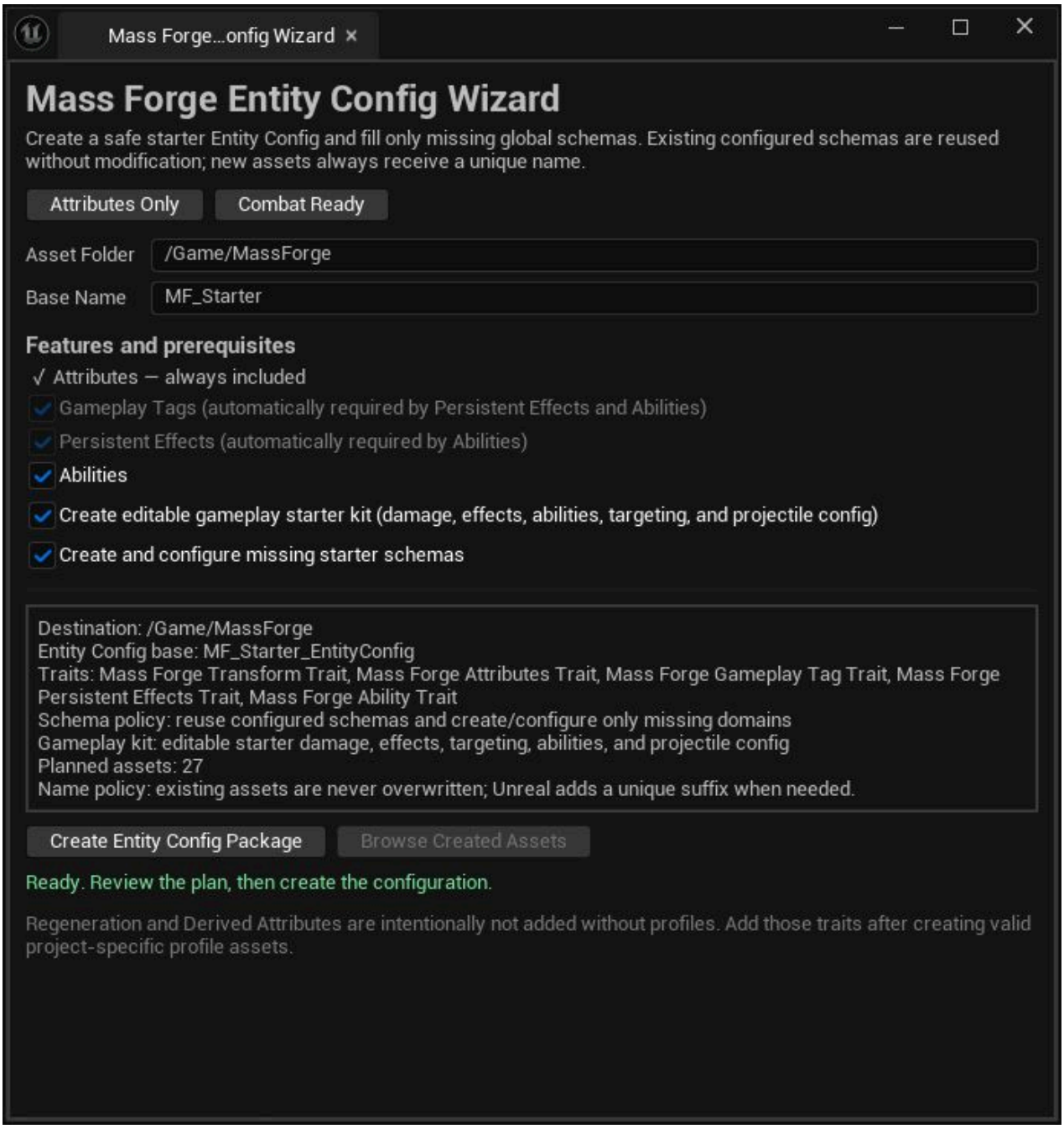
Never include a file from a `Private` directory. `FMFEntityHandle` is the supported index-and-serial identity; do not store a raw entity index or rebuild a handle from partial data.



The include paths begin at the plugin's public boundary. The checked source intentionally includes and instantiates every public type used below, so public-header, module-dependency, and signature drift fails the release compile test.

Author the data first

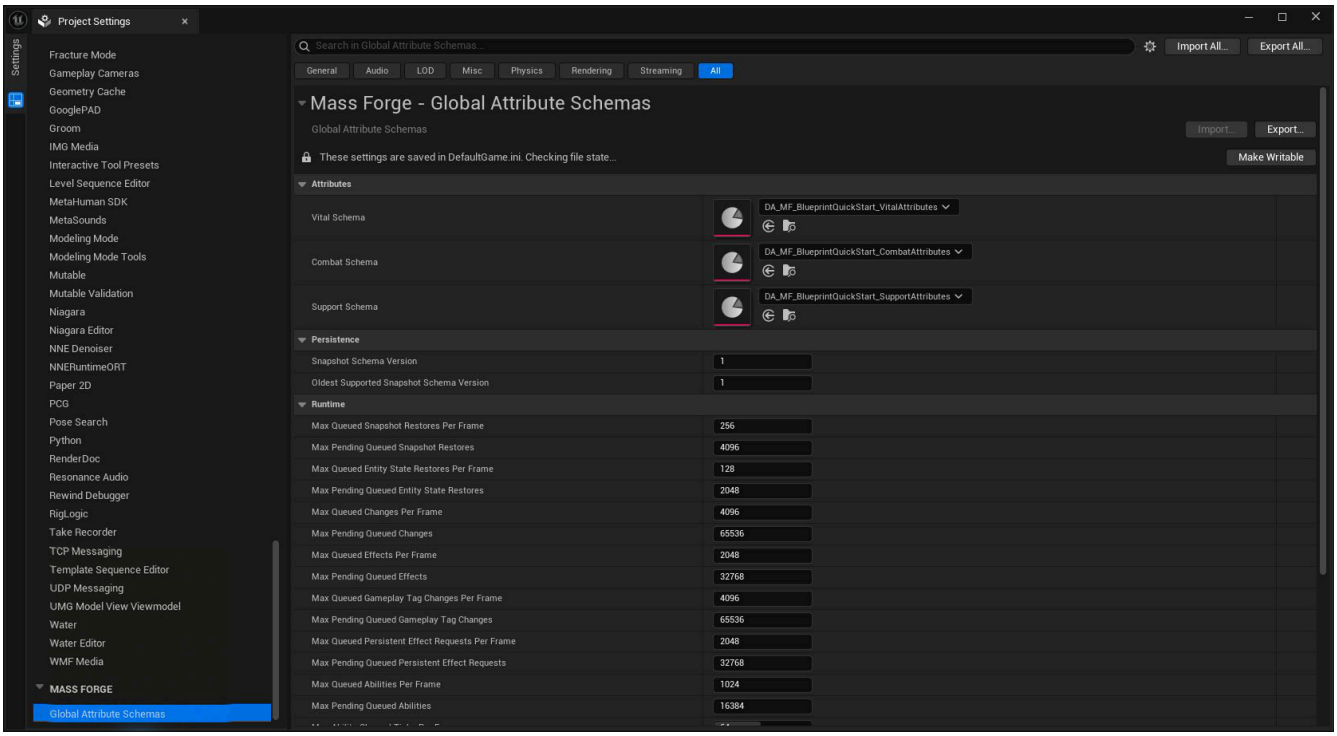
Use the Entity Config wizard or duplicate the assets in `/MassForge/MassForge/BlueprintExamples/Data` into project Content. Keep these as editable project-owned inputs:



C++ orchestration consumes the same editable Entity Config, schemas, definitions, effects, abilities, and projectile config as Blueprint. ***Combat Ready*** *is a starting package, not generated gameplay code.*

- a Combat Ready Mass Entity Config with the Transform, Attributes, Gameplay Tags, Effects, and Ability traits required by the intended gameplay;
- Vital, Combat, and Support schemas;

- a Damage Definition;
- Instant/Persistent Effects;
- Ability Data Assets;
- a projectile Entity Config with the Mass Forge Transform and Projectile traits.



Verify these project-wide assignments before resolving attribute IDs in C++. The names used below must exist uniquely in the configured schemas.

The C++ examples below assume those assets are assigned through `UPROPERTY` references. Do not synchronously load gameplay assets inside a Mass processor loop.

Spawn generation-safe entities

```
FMFEntityHandle Source;
const EMFEntitySpawnResult SpawnResult =
    UMF_EntityBlueprintLibrary::SpawnMassForgeEntity(
        this,
        CombatEntityConfig,
        GetActorTransform(),
        Source);

if (SpawnResult != EMFEntitySpawnResult::Success)
{
    // Surface the exact enum in diagnostics or UI.
    return;
}
```

```cpp
#if WITH_DEV_AUTOMATION_TESTS
#include "Data/MF_AbilityData.h"
#include "Data/MF_DamageDefinitionData.h"
#include "Data/MF_InstantEffectData.h"
#include "MassEntityConfigAsset.h"

namespace UE::MassForge::Tests
{
    /**
     * Compile-only counterpart to Docs/CPP_GAMEPLAY_QUICK_START.md.
     *
     * Keeping the calls together in an external consumer module catches public-header,
     * signature, and dependency drift without executing gameplay in an editor test world.
     */
    static void CompileCppGameplayQuickStart(
        const UObject* WorldContextObject,
        UMassEntityConfigAsset* CombatEntityConfig,
        UMassEntityConfigAsset* ProjectileEntityConfig,
        UMF_DamageDefinitionData* DamageDefinition,
        UMF_InstantEffectData* HealEffect,
        UMF_AbilityData* Ability,
        const FVector& SourceLocation,
        const FVector& TargetLocation)
    {
        FMFEntityHandle Source;
        FMFEntityHandle Target;
        const EMFEntitySpawnResult SourceSpawn = UMF_EntityBlueprintLibrary::SpawnMassForgeEntity(
            WorldContextObject,
            CombatEntityConfig,
            FTransform(SourceLocation),
            Source);
        const EMFEntitySpawnResult TargetSpawn = UMF_EntityBlueprintLibrary::SpawnMassForgeEntity(
            WorldContextObject,
            CombatEntityConfig,
            FTransform(TargetLocation),
            Target);

        FMFAttributeId Health;
        Health.Domain = EMFAttributeDomain::Vital;
        Health.Name = TEXT("Health");
        float CurrentHealth = 0.0f;
        const EMFAttributeAccessResult Read = UMF_AttributeBlueprintLibrary::GetMassForgeAttribute(
            WorldContextObject, Source, Health, CurrentHealth);
        const FMFAttributeChangeResult Change = UMF_AttributeBlueprintLibrary::ChangeMassForgeAttribute(
            WorldContextObject, Source, Health, EMFAttributeOperation::Add, -10.0f, TEXT("Example.DirectChange"));
        const FMFAttributeRequestReceipt QueuedChange = UMF_AttributeBlueprintLibrary::QueueMassForgeAttributeChange(
            WorldContextObject, Source, Health, EMFAttributeOperation::Add, -10.0f, TEXT("Example.QueuedChange"));
        const FMFInstantEffectResult Heal = UMF_AttributeBlueprintLibrary::ApplyMassForgeInstantEffect(
            WorldContextObject, Source, HealEffect, TEXT("Example.Heal"));
```

Spawning returns two things with different jobs: an explicit outcome for the current request and a complete generation-safe handle for future work. Preserve both; never infer success from a raw index.

`this` is any object with the intended gameplay world. Use `SpawnMassForgeEntities` for a bounded list of transforms. That convenience path is capped at 10,000 returned handles; large population creation belongs in a project-owned C++ Mass spawning path.

Before a later operation, `IsMassForgeEntityValid(this, Source)` is a convenient predicate. Every Mass Forge operation still revalidates the full handle itself. `DestroyMassForgeEntity` is immediate and therefore valid only while Mass is idle.

Read and change an attribute

```
FMFAttributeId Health;
Health.Domain = EMFAttributeDomain::Vital;
Health.Name = TEXT("Health");

float CurrentHealth = 0.0f;
const EMFAttributeAccessResult ReadResult =
    UMF_AttributeBlueprintLibrary::GetMassForgeAttribute(
        this, Source, Health, CurrentHealth);

if (ReadResult == EMFAttributeAccessResult::Success)
{
    const FMFAttributeChangeResult Change =
        UMF_AttributeBlueprintLibrary::ChangeMassForgeAttribute(
            this,
            Source,
            Health,
            EMFAttributeOperation::Add,
            -10.0f,
            TEXT("Example.DirectChange"));

    if (!Change.WasSuccessful())
    {
        // Inspect Change.Result and Change.Attribute.
    }
}
```

The direct path is for a known Mass-idle point. For general game-thread orchestration, queue the request and retain the receipt ID:

```
const FMFAttributeRequestReceipt Receipt =
    UMF_AttributeBlueprintLibrary::QueueMassForgeAttributeChange(
        this,
        Source,
        Health,
        EMFAttributeOperation::Add,
        -10.0f,
        TEXT("Example.QueuedChange"));

if (!Receipt.WasAccepted())
{
    // No completion will arrive. Handle queue-full or validation failure now.
}
```

Bind once to `UMF_AttributeSubsystem::OnAttributeRequestCompletedNative` and correlate its completion by `RequestId`. Acceptance is not execution success: the target and schema are validated again in the deterministic deferred phase.

```
#if WITH_DEV_AUTOMATION_TESTS
        const FVector& TargetLocation)
    {
        FMFEntityHandle Source;
        FMFEntityHandle Target;
        const EMFEntitySpawnResult SourceSpawn = UMF_EntityBlueprintLibrary::SpawnMassForgeEntity(
            WorldContextObject,
            CombatEntityConfig,
            FTransform(SourceLocation),
            Source);
        const EMFEntitySpawnResult TargetSpawn = UMF_EntityBlueprintLibrary::SpawnMassForgeEntity(
            WorldContextObject,
            CombatEntityConfig,
            FTransform(TargetLocation),
            Target);

        FMFAttributeId Health;
        Health.Domain = EMFAttributeDomain::Vital;
        Health.Name = TEXT("Health");
        float CurrentHealth = 0.0f;
        const EMFAttributeAccessResult Read = UMF_AttributeBlueprintLibrary::GetMassForgeAttribute(
            WorldContextObject, Source, Health, CurrentHealth);
        const FMFAttributeChangeResult Change = UMF_AttributeBlueprintLibrary::ChangeMassForgeAttribute(
            WorldContextObject, Source, Health, EMFAttributeOperation::Add, -10.0f, TEXT("Example.DirectChange"));
        const FMFAttributeRequestReceipt QueuedChange = UMF_AttributeBlueprintLibrary::QueueMassForgeAttributeChange(
            WorldContextObject, Source, Health, EMFAttributeOperation::Add, -10.0f, TEXT("Example.QueuedChange"));
        const FMFInstantEffectResult Heal = UMF_AttributeBlueprintLibrary::ApplyMassForgeInstantEffect(
            WorldContextObject, Source, HealEffect, TEXT("Example.Heal"));

        FMFDamageRequest DamageRequest;
        DamageRequest.SourceEntity = Source;
        DamageRequest.TargetEntity = Target;
        DamageRequest.BaseDamage = 25.0f;
        DamageRequest.CriticalRoll = 1.0f;
        DamageRequest.Context.Origin = SourceLocation;
        DamageRequest.Context.bHasOrigin = true;
        const FMFDamageApplicationResult Damage = UMF_DamageBlueprintLibrary::ApplyMassForgeDamage(
            WorldContextObject, DamageDefinition, DamageRequest);

        const EMFAbilityResult Grant = UMF_AbilityBlueprintLibrary::GrantMassForgeAbility(
            WorldContextObject, Source, Ability);
        const FMFAbilityActivationResult CanActivate = UMF_AbilityBlueprintLibrary::CanActivateMassForgeAbility(
            WorldContextObject, Source, Target, Ability);
        const FMFAbilityActivationResult Activation = UMF_AbilityBlueprintLibrary::ActivateMassForgeAbility(
            WorldContextObject, Source, Target, Ability);
        const FMFAbilityRequestReceipt QueuedActivation = UMF_AbilityBlueprintLibrary::QueueMassForgeAbilityActivation(
            WorldContextObject, Source, Target, Ability);

        FMFAbilityRadiusTargetRequest RadiusRequest;
```

The compile-checked counterpart keeps the ordinary gameplay sequence in one readable frame. Every operation retains its typed result or receipt; none is collapsed into an unchecked boolean.

Apply reusable healing and effects

```
const FMFInstantEffectResult HealResult =
    UMF_AttributeBlueprintLibrary::ApplyMassForgeInstantEffect(
        this,
        Source,
        HealEffect,
        TEXT("Example.Heal"));
```

Use the queued equivalent when overlap with Mass processing is possible. Prefer an Effect over a raw attribute change when the operation has several atomic modifiers, gameplay tags, source context, or a reusable designer-authored identity.

Deal deterministic damage

```
FMFDamageRequest Request;
Request.SourceEntity = Source;
Request.TargetEntity = Target;
Request.BaseDamage = 25.0f;
Request.CriticalRoll = 1.0f;
Request.Context.Origin = GetActorLocation();
Request.Context.bHasOrigin = true;

const FMFDamageApplicationResult Damage =
    UMF_DamageBlueprintLibrary::ApplyMassForgeDamage(
        this,
        DamageDefinition,
        Request);

if (Damage.WasSuccessful())
{
    // Damage.HealthAfter, Damage.ShieldAfter, Damage.bCritical, Damage.bKilled.
}
```

The result contains the resolved formula stages, policy decision, attribute transaction, death request, and failure location. Use `QueueMassForgeDamage` outside a guaranteed Mass-idle point and correlate `OnDamageRequestCompletedNative` by request ID.

Ordinary Actors can use `ApplyMassForgeActorDamageToEntity`; entity-to-entity combat uses `ApplyMassForgeEntityDamageToEntity`. These are convenience constructors around the same canonical request.

Grant and activate an ability

```
const EMFAbilityResult GrantResult =
    UMF_AbilityBlueprintLibrary::GrantMassForgeAbility(
        this, Source, Ability);

if (GrantResult == EMFAbilityResult::Success)
{
    const FMFAbilityActivationResult Check =
        UMF_AbilityBlueprintLibrary::CanActivateMassForgeAbility(
            this, Source, Target, Ability);

    if (Check.WasSuccessful())
    {
        const FMFAbilityActivationResult Activation =
            UMF_AbilityBlueprintLibrary::ActivateMassForgeAbility(
                this, Source, Target, Ability);
    }
}
```

Activation is atomic across costs, cooldowns, charges, tags, effects, and authored execution behavior. A cast or channel returns a started lifecycle rather than pretending the final effect committed immediately.

For ordinary runtime orchestration, prefer `QueueMassForgeAbilityGrant` and `QueueMassForgeAbilityActivation`. Bind once to `UMF_AbilitySubsystem::OnAbilityOperationRequestCompletedNative`, then match the positive request ID. Cancellation and interruption share that ordered request channel.

Select actorless or represented targets

```
FMFAbilityRadiusTargetRequest Query;
Query.SourceEntity = Source;
Query.Origin = GetActorLocation();
Query.Radius = 1200.0f;
Query.MaxTargets = 16;
Query.bExcludeSource = true;

const FMFAbilityTargetSelection Selection =
    UMF_AbilityTargetingBlueprintLibrary::FindMassForgeAbilityTargetsInRadius(
        this, Query);

for (const FMFAbilityTargetCandidate& Candidate : Selection.Targets)
{
    const FMFEntityHandle TargetEntity = Candidate.Entity;
    // Candidate.RepresentedActor is optional; actorless entities remain valid targets.
}
```

Radius and segment queries are bounded and deterministic. Use `SelectMassForgeAbilityTargetsWithProvider` when a project owns a more specialized C++/Blueprint targeting rule; Mass Forge validates, de-duplicates, and bounds the returned handles.

Launch a gameplay projectile

```
const FMFProjectileLaunchReceipt Launch =
    UMF_ProjectileBlueprintLibrary::LaunchMassForgeEntityProjectile(
        this,
        ProjectileEntityConfig,
        Source,
        Target,
        MuzzleLocation,
        TargetLocation,
        1800.0f,            // speed
        120.0f,             // arc height
        0.0f,               // launch delay
        8.0f,               // maximum lifetime
        0,                  // presentation channel
        TEXT("Impact.Arc"),
        DamageDefinition,
        25.0f,
        FMFEffectContext());

if (!Launch.WasSuccessful())
{
    // No projectile exists; inspect Launch.Result.
}
```

Damage commits when Mass Forge resolves impact, not at launch. Gameplay state remains in the projectile subsystem; visuals consume `GetMassForgeProjectileSnapshots`. Use Niagara, HISM, or an Actor pool for production populations. `PresentFirstMassForgeProjectile` is intentionally only a small-scene teaching helper.

```
#if WITH_DEV_AUTOMATION_TESTS

        FMFDamageRequest DamageRequest;
        DamageRequest.SourceEntity = Source;
        DamageRequest.TargetEntity = Target;
        DamageRequest.BaseDamage = 25.0f;
        DamageRequest.CriticalRoll = 1.0f;
        DamageRequest.Context.Origin = SourceLocation;
        DamageRequest.Context.bHasOrigin = true;
        const FMFDamageApplicationResult Damage = UMF_DamageBlueprintLibrary::ApplyMassForgeDamage(
            WorldContextObject, DamageDefinition, DamageRequest);

        const EMFAbilityResult Grant = UMF_AbilityBlueprintLibrary::GrantMassForgeAbility(
            WorldContextObject, Source, Ability);
        const FMFAbilityActivationResult CanActivate = UMF_AbilityBlueprintLibrary::CanActivateMassForgeAbility(
            WorldContextObject, Source, Target, Ability);
        const FMFAbilityActivationResult Activation = UMF_AbilityBlueprintLibrary::ActivateMassForgeAbility(
            WorldContextObject, Source, Target, Ability);
        const FMFAbilityRequestReceipt QueuedActivation = UMF_AbilityBlueprintLibrary::QueueMassForgeAbilityActivation(
            WorldContextObject, Source, Target, Ability);

        FMFAbilityRadiusTargetRequest RadiusRequest;
        RadiusRequest.SourceEntity = Source;
        RadiusRequest.Origin = SourceLocation;
        RadiusRequest.Radius = 1200.0f;
        RadiusRequest.MaxTargets = 16;
        const FMFAbilityTargetSelection Selection =
        UMF_AbilityTargetingBlueprintLibrary::FindMassForgeAbilityTargetsInRadius(
            WorldContextObject, RadiusRequest);

        const FMFProjectileLaunchReceipt Launch = UMF_ProjectileBlueprintLibrary::LaunchMassForgeEntityProjectile(
            WorldContextObject,
            ProjectileEntityConfig,
            Source,
            Target,
            SourceLocation,
            TargetLocation,
            1800.0f,
            120.0f,
            0.0f,
            8.0f,
            0,
            TEXT("Impact.Arc"),
            DamageDefinition,
            25.0f,
            FMFEffectContext());

        FMFEntityInspectionSnapshot Inspection;
        const EMFEntityInspectionResult InspectionResult =
            UMF_EntityInspectionBlueprintLibrary::CaptureMassForgeEntityInspection(
```

Selection produces bounded generation-safe candidates; launch owns flight and impact resolution. The authored damage definition is passed into the projectile so damage is committed at impact instead of being applied invisibly when the button is pressed.

Inspect and persist state

```
FMFEntityInspectionSnapshot Inspection;
const EMFEntityInspectionResult InspectionResult =
    UMF_EntityInspectionBlueprintLibrary::CaptureMassForgeEntityInspection(
        this, Source, Inspection, true);

FMFEntityStateSnapshot SavedState;
const FMFEntityStateSnapshotOperationResult Capture =
    UMF_EntityStateBlueprintLibrary::CaptureMassForgeEntityState(
        this, Source, SavedState);
```

Store the snapshot in the project's save format, not the entity handle. Recreate an entity in the destination world, then restore at a known Mass-idle point with `RestoreMassForgeEntityState`, or queue `QueueMassForgeEntityStateRestore`. Active externally targeted casts/channels require the documented project-owned relationship key and lifecycle target binding.

```
#if WITH_DEV_AUTOMATION_TESTS
        RadiusRequest.MaxTargets = 16;
        const FMFAbilityTargetSelection Selection =
        UMF_AbilityTargetingBlueprintLibrary::FindMassForgeAbilityTargetsInRadius(
            WorldContextObject, RadiusRequest);

        const FMFProjectileLaunchReceipt Launch = UMF_ProjectileBlueprintLibrary::LaunchMassForgeEntityProjectile(
            WorldContextObject,
            ProjectileEntityConfig,
            Source,
            Target,
            SourceLocation,
            TargetLocation,
            1800.0f,
            120.0f,
            0.0f,
            8.0f,
            0,
            TEXT("Impact.Arc"),
            DamageDefinition,
            25.0f,
            FMFEffectContext());

        FMFEntityInspectionSnapshot Inspection;
        const EMFEntityInspectionResult InspectionResult =
            UMF_EntityInspectionBlueprintLibrary::CaptureMassForgeEntityInspection(
                WorldContextObject, Source, Inspection, true);
        FMFEntityStateSnapshot SavedState;
        const FMFEntityStateSnapshotOperationResult Capture =
            UMF_EntityStateBlueprintLibrary::CaptureMassForgeEntityState(
                WorldContextObject, Source, SavedState);

        // Keep every public result type instantiated so API drift cannot be hidden by optimization.
        (void)SourceSpawn;
        (void)TargetSpawn;
        (void)Read;
        (void)CurrentHealth;
        (void)Change;
        (void)QueuedChange;
        (void)Heal;
        (void)Damage;
        (void)Grant;
        (void)CanActivate;
        (void)Activation;
        (void)QueuedActivation;
        (void)Selection;
        (void)Launch;
        (void)InspectionResult;
        (void)Capture;
    }
```

Inspection is an observation surface; persistence captures a portable value. The current world handle is deliberately not the save identity and must not be serialized as if it survives recreation.

Blueprint and C++ parity

Task	Blueprint	Ordinary C++	Processor/high-volume C++
Spawn a small population	Entity spawn nodes	Same static public functions	Project Mass spawning path
Read/change one attribute	Direct or async nodes	Direct or queued public API	Compiled attribute + fragment view

Task	Blueprint	Ordinary C++	Processor/high-volume C++
Effects/damage/abilities	Direct or async nodes	Direct or queued public API	Bounded commands or high-volume effect table
Target selection	Radius, trace, provider nodes	Same public APIs	Project query/processor with bounded handoff
Projectile gameplay	Launch async/convenience nodes	Same public APIs/subsystem	Projectile trait + built-in processor
Presentation	Blueprint events/snapshots	Native delegates/snapshots	Collect bounded presentation state; render elsewhere
Save/restore	Snapshot nodes	Same public APIs	Perform outside processor hot loops
Inspection	Inspector and capture node	Capture API/native providers	Never inspect every entity every frame

Intentionally C++ only work

Blueprint is supported for authoring and ordinary gameplay orchestration. C++ is required or strongly preferred when the work itself is high-volume: custom Mass processors, direct fragment iteration, compiled attribute bindings, populations above the Blueprint batch limit, project representation processors, and renderer-specific Niagara/HISM upload paths.

That boundary prevents per-entity reflection, allocation, delegate broadcasts, and large Blueprint arrays from becoming the scale limit. It does not create a second gameplay model; both paths use the same schemas, Data Assets, entity identity, and result contracts.

Continue with C++ high-volume attribute access, architecture, deferred commands, Mass projectiles, and entity-state persistence.

BLUEPRINT AND C++ WORKFLOWS

Mass Forge C++ high-volume attribute access

> **INFO - Intentionally C++ only**
>
> Processor-level iteration, compiled fragment access, and million-entity hot paths stay in C++ so they can remain allocation-free and avoid moving huge handle arrays through Blueprint. Blueprint remains the supported authoring and orchestration surface for ordinary gameplay-scale operations.

Blueprint and subsystem calls are designed for gameplay orchestration. A Mass processor that touches thousands of entities should compile attribute names once and then use slot-based access inside its entity loop.

Compile before the hot loop

```
#include "Attributes/MF_CompiledAttribute.h"
#include "Subsystems/MF_AttributeSubsystem.h"

FMFAttributeId HealthId;
HealthId.Domain = EMFAttributeDomain::Vital;
HealthId.Name = TEXT("Health");

FMFCompiledAttribute HealthBinding;
const EMFAttributeAccessResult CompileResult =
        AttributeSubsystem->CompileAttribute(HealthId, HealthBinding);
```

Keep the binding only when `CompileResult` is `Success`. Compilation performs the schema/name lookup and copies the slot and clamp policy into a small value type.

Use fragment views in a processor

Require the three Mass Forge attribute fragments in the query, then resolve each entity without subsystem calls, allocation, asset access, or `FName` lookup:

```
TArrayView<FMF_VitalAttributesFragment> Vitals =
        Context.GetMutableFragmentView<FMF_VitalAttributesFragment>();
TArrayView<FMF_CombatAttributesFragment> Combat =
        Context.GetMutableFragmentView<FMF_CombatAttributesFragment>();
TArrayView<FMF_SupportAttributesFragment> Support =
        Context.GetMutableFragmentView<FMF_SupportAttributesFragment>();

for (int32 EntityIndex = 0; EntityIndex < Context.GetNumEntities(); ++EntityIndex)
{
        const FMFAttributeChangeResult Change = UE::MassForge::ApplyCompiledAttribute(
                Vitals[EntityIndex],
                Combat[EntityIndex],
                Support[EntityIndex],
                HealthBinding,
                EMFAttributeOperation::Add,
                -10.0f);
}
```

`ReadCompiledAttribute` provides the equivalent const read path. `ResolveCompiledAttribute` is available when a custom processor needs the raw value pointer.

Schema refresh safety

Every binding records the attribute subsystem's schema generation. Check it once before scheduling or entering a batch:

```
if (!AttributeSubsystem->IsCompiledAttributeCurrent(HealthBinding))
{
        // Recompile outside the Mass execution loop.
}
```

`RefreshSchemas` invalidates every earlier binding, even when the apparent slot did not change. This prevents callers from silently carrying an old clamp policy into a new schema generation.

Events and threading

Compiled access intentionally does not broadcast Blueprint delegates. Broadcasting per entity would defeat the batch path and is unsafe from worker threads. A processor that needs presentation events should collect a bounded summary and hand it to a game-thread presentation system after processing.

For game-thread orchestration, the attribute, Instant Effect, Gameplay Tag, and Persistent Effect Blueprint delegates have corresponding `Native` multicast mirrors. Ability transactions defer both forms until all participating state has committed and discard them together on rollback.

Do not call the direct Blueprint/subsystem mutation functions while Mass is processing. Use compiled fragment access from processors and the bounded queued APIs from external gameplay code.

Compiled effect application and removal

For repeated multi-operation effects, add **Mass Forge High Volume Effects Trait** and author an explicit Apply/Remove Instant Effect pair. The trait resolves attributes into slots once and stores the fixed-capacity table in a const-shared archetype fragment. A producer processor submits by cached numeric index through `RequestHighVolumeEffect`; the built-in `UMF_HighVolumeEffectsProcessor` commits the complete attribute/tag side atomically and retains one bounded completion per entity. This path also synchronizes active persistent-modifier base values.

It supports finite context-free operations only and deliberately emits no per-entity delegates. Context parameters, source/target captures, duration, stacking, handles, and presentation events remain on the normal effect APIs. See the high-volume compiled effects guide for setup, ordering, back-pressure, completion, and removal semantics.

Memory before throughput

Call `UMF_PerformanceBlueprintLibrary::GetMassForgeMemoryFootprint()` in a diagnostic or capacity-planning path before choosing a population archetype. It reports exact native payloads for the current build and distinguishes per-entity fragments from the const-shared Derived Attributes graph. See the memory and capacity guide for verified Win64 sizes, preset totals, exclusions, and the required packaged profiling workflow.

Mass Forge's built-in processors are protected by a source-level processor hot-loop performance contract. New processors must keep name lookup, asset loading, delegates, dynamic containers, allocation sites, reflection, and logging outside their marked entity-iteration region.

BLUEPRINT AND C++ WORKFLOWS

AI and StateTree ability requests

Mass Forge provides a bounded, pollable ability-request contract for StateTree tasks, behavior systems, utility AI, and project-owned C++ controllers. It does not require delegate ownership and it does not add a hard dependency on Unreal's experimental Mass AI or StateTree plugins.

The API works for actorless Mass entities and represented entities through the same stale-safe handles. A request queues normal Mass Forge activation; it never bypasses grant, target, targeting-policy, tag, cooldown, charge, requirement, cost, transaction, or one-active-lifecycle rules.

Blueprint nodes

- `Request Mass Forge Ability Activation for AI` returns an acceptance result and stable request ID.
- `Request Mass Forge Ability Activation for AI from Actors` performs the same request after resolving represented source and optional target Actors.
- `Get Mass Forge AI Ability Request State` retrieves the current state for that ID.
- `Cancel Mass Forge AI Ability Request` cancels pending admission or asks the active cast/channel to follow its authored cancellation rule.
- `Forget Mass Forge AI Ability Request` explicitly releases a terminal record.

These are world-context nodes, so Blueprint tasks do not need to cache the subsystem. Equivalent methods are available directly on `UMF_AbilitySubsystem` for C++ tasks.

StateTree task pattern

Store the returned `RequestId` as task instance data rather than on the entity.

1. **Enter State:** call `Request Mass Forge Ability Activation for AI`. Return failure immediately if the receipt was not accepted.
2. **Tick:** call `Get Mass Forge AI Ability Request State` with the stored ID.
3. Return **Running** for `Pending` or `Active`.
4. Return **Succeeded** for `Succeeded`.
5. Return **Failed** for `Failed`, `Cancelled`, or `Interrupted`, unless the project's tree deliberately routes those outcomes differently.
6. **Exit State:** if the task is leaving while the ticket is `Pending` or `Active`, call cancel. Once it is terminal, call forget.

Do not treat the poll function's `Success` result as ability success. It means only that the ticket exists. Read `State.Status`, then use `State.Activation`, `State.Lifecycle`, and `State.EndReason` for detail.

Status contract

Status	Meaning	StateTree default
`Pending`	Accepted by the bounded queue but not executed yet	Running
`Active`	A cast or channel lifecycle is active	Running
`Succeeded`	An instant ability committed, or a cast/channel completed successfully	Succeeded
`Failed`	Admission execution or lifecycle commit failed	Failed
`Cancelled`	Pending admission was suppressed or the lifecycle was cancelled	Failed or project-specific
`Interrupted`	The lifecycle was interrupted, revoked, or lost its source entity	Failed or project-specific

`Activation` retains the machine-readable ability result. While active, `Lifecycle` exposes its world-local instance ID, phase, remaining time, completed channel ticks, target, and commit state. Terminal lifecycle tickets also expose `EndReason`. This lets a task distinguish an unaffordable request from an invalid target, a user cancellation from an interruption, or a normal completion from a commit-time revalidation failure.

Timing and cancellation

New requests begin as `Pending`. The shared deferred coordinator executes ability requests in its normal final phase, after queued attribute changes, Instant Effects, and damage. Existing lifecycles advance before new requests are admitted, so a newly started cast never consumes the frame delta that admitted it.

Cancelling `Pending` changes the ticket to `Cancelled` immediately and guarantees that its queued gameplay operation is suppressed. The ticket may safely be forgotten before the queue drains; cancellation is retained internally until that queue item is consumed.

Cancelling `Active` delegates to the ability lifecycle. An ability with `Can Be Cancelled` disabled returns `Cancellation Blocked` and remains active. Cancellation does not refund an already committed channel; see ABILITY_LIFECYCLE_GUIDE.md.

Bounded retention

`Max Tracked AI Ability Requests` under **Project Settings** > **Mass Forge** > **Global Attribute Schemas** > **Runtime** limits retained tickets. Pending and active tickets are never evicted. When the limit is reached, the oldest terminal ticket is evicted to admit new work. If every slot is live, a new request returns `Queue Full` without entering the ability queue.

Call forget after consuming a terminal outcome when practical. Polling an evicted or forgotten positive ID returns `Request Not Found`; zero and negative IDs return `Invalid Request ID`. Forgetting live work returns `Request Not Terminal`.

Ticket records are runtime coordination state, not SaveGame data and are not included in replicated gameplay state. The owner-only network state stream exposes the entity's granted abilities, cooldowns, charges, and active lifecycle, but AI request tickets remain authority-local. Opt-in network Ability prediction is a separate presentation-only request ledger and never replicates or predicts AI tickets.

C++ sketch

```
if (RequestId == 0)
{
        const FMFAbilityRequestReceipt Receipt = AbilitySubsystem-
>RequestAbilityActivationForAI(Source, Target, Ability);
        if (!Receipt.WasAccepted())
        {
                return EStateTreeRunStatus::Failed;
        }
        RequestId = Receipt.RequestId;
}

FMFAbilityAIRequestState State;
if (AbilitySubsystem->GetAIAbilityRequestState(RequestId, State) != EMFAbilityResult::Success)
{
        return EStateTreeRunStatus::Failed;
}

return State.Status == EMFAbilityAIRequestStatus::Succeeded
        ? EStateTreeRunStatus::Succeeded
        : State.IsTerminal() ? EStateTreeRunStatus::Failed : EStateTreeRunStatus::Running;
```

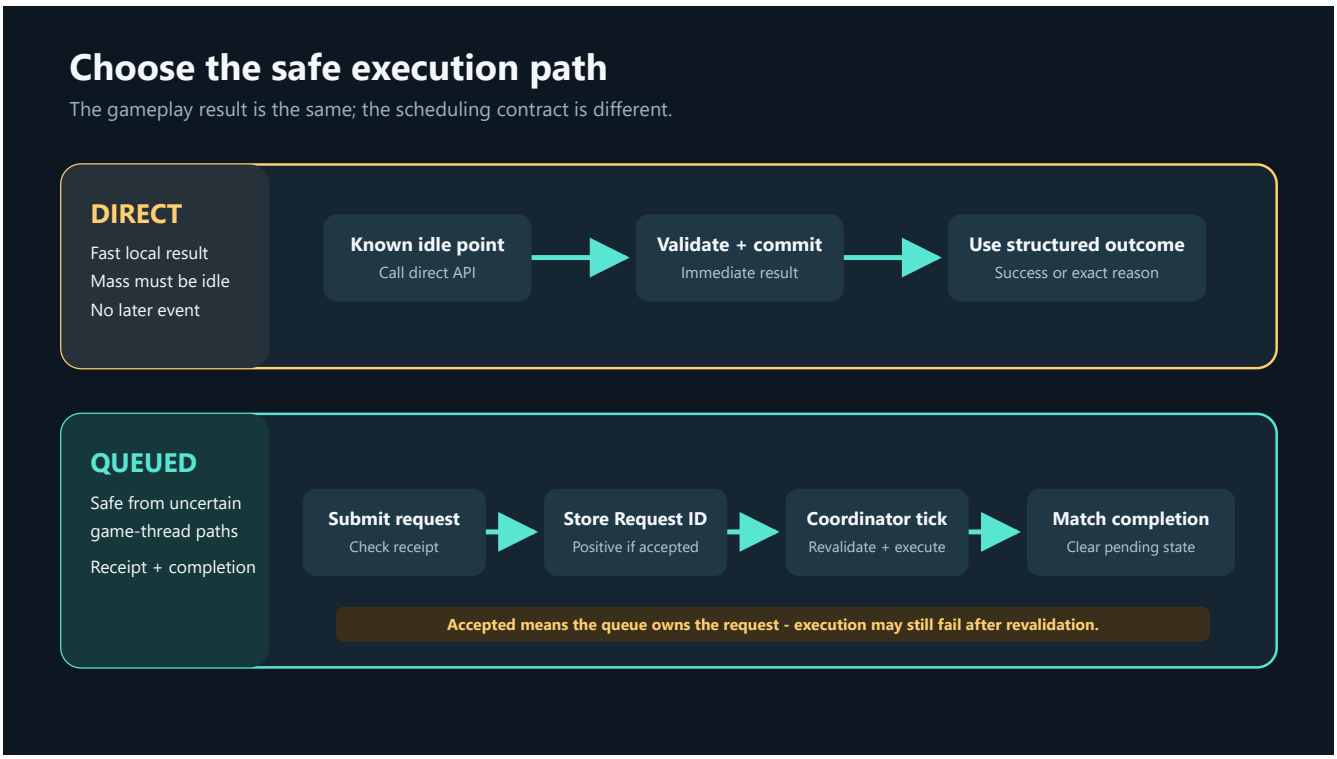
The sketch intentionally leaves task-exit policy to the host project. A task that owns the gameplay request should cancel it on early exit; an observer-only task should not.

BLUEPRINT AND C++ WORKFLOWS

Deferred command execution guide

Mass Forge rejects unsafe direct entity mutations while Mass is processing. External gameplay, Blueprint callbacks, async handoffs returning to the game thread, and presentation systems can instead submit bounded deferred commands and match completion events by request ID.

All external queues and the Persistent Effect scheduler are executed by one **Mass Forge Deferred Command Subsystem**. This removes dependence on incidental Unreal subsystem tick order.



Use a direct call only at a lifecycle point where Mass is known to be idle. Use the queued path when timing is uncertain: acceptance returns a positive Request ID, then the coordinator revalidates and publishes exactly one correlated completion unless the world tears down.

Fixed phase order

Every safe coordinator tick runs these phases:

1. **Entity State Restores** - queued complete durable snapshots atomically replace supported attributes, tags, grants, cooldowns, charges, regeneration phase, active Persistent Effects, and one active Ability Lifecycle.
2. **Attribute Snapshot Restores** - queued focused attribute-only snapshots establish saved attribute values.
3. **Attribute Changes** - queued Set/Add/Multiply requests.
4. **Gameplay Tag Changes** - queued raw counted-tag changes.
5. **Instant Effects** - queued atomic attribute and counted-tag transactions.
6. **Persistent Effects** - queued Apply/Remove requests, then deterministic duration/period advancement, due pulses, and expiration.

7. **Projectile Impacts** - completed Mass projectile motion publishes its terminal result and optionally applies impact-driven damage.
8. **Damage** - queued Damage Definition requests and project policy evaluation.
9. **Death Handling** - bounded Mass-entity destruction requested by successful lethal damage, after kill listeners and queued-damage completion events.
10. **Ability Lifecycles** - existing casts/channels advance in stable source-entity order.
11. **Ability Requests** - one FIFO for queued Activate, Grant, Revoke, Cancel, and Interrupt operations. Activation admission can commit costs, effects, cooldowns, and charges; lifecycle control is correlated to the exact captured lifecycle ID.

Advancing existing lifecycles before admitting queued Ability operations ensures a queued cast never consumes the delta time from the frame that accepted it. It also means a lifecycle that naturally ends in phase ten completes before a queued phase-eleven Cancel/Interrupt is evaluated. Directly started lifecycles advance at the next coordinator tick. Channel catch-up remains bounded per active ability and carries earned pulses forward.

`Get Deferred Command Phase Order` returns these names for Blueprint diagnostics and tools. Category phase order deliberately takes precedence over cross-category submission order. Within each category, requests remain FIFO.

For example, if a Blueprint queues an Ability, Damage, an Instant Effect, a raw Gameplay Tag change, an Attribute Change, an Attribute Snapshot Restore, and finally an Entity State Restore before the next tick, Mass Forge still executes the complete Entity State Restore first, then the focused Attribute Snapshot Restore, raw Attribute Change, raw Gameplay Tag change, Instant Effect, scheduled Persistent Effects, resolved Projectile Impacts, Damage, optional Death Handling, existing Ability Lifecycles, and new Ability Request. This is intentional and covered by automated phase-order tests.

The complete same-boundary pulse, expiration, cancellation, cost, lifecycle, and death precedence rules are defined in the deterministic gameplay ordering contract.

Phase-start snapshot rule

Each phase snapshots how many requests are eligible when that phase begins, capped by its per-frame budget. A completion callback can enqueue more work, but it cannot extend the phase currently being flushed.

- Work queued into the current phase waits for the next safe coordinator tick.
- Work queued into a phase that already ran waits for the next safe tick.
- Work queued into a later phase may execute in the current tick when that later phase begins and has budget.

Calling the coordinator recursively from a completion callback is ignored while a flush is active. This prevents unbounded recursion and preserves the phase-start rule.

Blueprint production pattern

1. Call the appropriate Queue node and store its positive `Request Id` when the receipt reports success.
2. Treat a queue-full or validation result as an immediate submission failure; no completion event will follow for rejected submission.
3. Bind the corresponding completion event on the owning subsystem.
4. Match the event's `Request Id` to the stored ID.
5. Branch on the complete execution result. Acceptance means only that the bounded queue owns the request; entities and assets are validated again when it executes.
6. Remove UI pending state on every completion outcome, not only success.

Available deferred entry points are:

Category	Queue node	Completion event
Entity State Restore	**Queue Mass Forge Entity State Restore**	`On Entity State Restore Completed`
Attribute Snapshot Restore	**Queue Mass Forge Attribute Snapshot Restore**	`On Attribute Snapshot Restore Request Completed`
Attribute	**Queue Mass Forge Attribute Change**	`On Attribute Request Completed`
Gameplay Tag	**Queue Add Mass Forge Gameplay Tag** / **Queue Remove Mass Forge Gameplay Tag**	`On Gameplay Tag Request Completed`
Instant Effect	**Queue Mass Forge Instant Effect**	`On Instant Effect Request Completed`
Persistent Effect	**Queue Mass Forge Persistent Effect** / **Queue Remove Mass Forge Persistent Effect**	`On Persistent Effect Request Completed`
Damage	**Queue Mass Forge Damage**	`On Damage Request Completed`
Ability	**Queue Mass Forge Ability Activation**, **Queue Mass Forge Ability Grant**, **Queue Mass Forge Ability Revocation**, **Queue Active Mass Forge Ability Cancellation**, or **Queue Active Mass Forge Ability Interruption**	`On Ability Operation Request Completed` ; activation also emits the compatibility `On Ability Request Completed`

Use the dedicated tag queue for one raw counted-tag mutation. Use a tag-only or mixed queued Instant Effect when multiple tag/attribute modifiers must succeed or fail as one entity transaction.

Budgets and back-pressure

Configure limits under **Project Settings > Mass Forge > Global Attribute Schemas > Runtime**:

Queue	Per-frame setting	Pending setting
Entity State Restore	Max Queued Entity State Restores Per Frame	Max Pending Queued Entity State Restores
Attribute Snapshot Restore	Max Queued Snapshot Restores Per Frame	Max Pending Queued Snapshot Restores
Attribute	Max Queued Changes Per Frame	Max Pending Queued Changes
Gameplay Tag	Max Queued Gameplay Tag Changes Per Frame	Max Pending Queued Gameplay Tag Changes
Instant Effect	Max Queued Effects Per Frame	Max Pending Queued Effects
Persistent Effect	Max Queued Persistent Effect Requests Per Frame	Max Pending Queued Persistent Effect Requests
Damage	Max Queued Damage Requests Per Frame	Max Pending Queued Damage Requests
Death destruction	Max Death Destructions Per Frame	Max Pending Death Destructions
Ability	Max Queued Abilities Per Frame	Max Pending Queued Abilities

Every setting has a minimum effective value of one. Pending limits provide deterministic back-pressure instead of allowing unbounded memory growth. Per-frame limits prevent a backlog from monopolizing one game frame. Tune them from measured project load rather than raising them blindly.

Completion and failure behavior

- Request IDs are positive and monotonic within their owning world subsystem. They identify a category request; they are not a global ordering number across categories.
- A stale generation-checked entity completes with that operation's explicit invalid-entity or invalid-target result.
- Queued definition assets preserve a soft reference plus their submitted stable ID. An ordinarily unloaded asset is synchronously resolved when its request executes; a missing/unloadable asset, broken redirect, or identity-mutated asset is rejected under the submitted identity.
- Successful application events occur before that request's completion event.
- World teardown discards pending commands and resets counters. Teardown does not broadcast gameplay completion events.
- A queue request is not a multi-request transaction. Separate requests can succeed or fail independently even when submitted together.

Recovery matrix

Submission and execution are separate decisions. A rejected receipt always has request ID `0`, owns no queue slot, and produces no later completion. An accepted receipt has a positive ID, and exactly one completion follows unless the owning world tears down first. Blueprint uses the assignable events shown below; C++ has native mirrors for every completion event.

Queue	Immediate receipt failures	Execution-time failures	Correlation preserved	Required recovery
Entity State Restore	Unsupported format, schema mismatch, malformed/duplicate/over-capacity attributes, tags, grants, cooldowns, charges, regeneration, Persistent Effects, or lifecycle state; unset/stale entity when safely inspectable; invalid world; missing/mismatched external lifecycle-target binding; legacy-format active temporal state; or `QueueFull`.	Stale full generation, changed schema/fragments/definitions, an invalid recreated lifecycle target, format-1/2 active Persistent Effect, format-1/2/3 active lifecycle, or another complete preflight failure.	Request ID, complete entity index/serial, source name, nested attribute result, failed tag/ability/group/target-reference, and restored section counts including active effects and lifecycle. The queue owns a deep copy of the snapshot and optional target binding.	Migrate the nested attribute snapshot; reacquire stale entities; use format 4 for active-lifecycle continuity; resolve an external target key to its recreated full-generation handle; repair payload/archetype failures. Never apply a partial save.
Attribute Snapshot Restore	Unsupported binary format, schema-version mismatch, empty payload, unset entity, invalid/duplicate attribute ID, non-finite value, invalid world, or `QueueFull`. The nested validation result includes received/expected versions and the failing attribute when applicable.	Stale entity generation, schema-version drift, missing schema/fragment, retired/renamed attribute, or other complete restore validation failure.	Request ID, complete entity index/serial, source name, received/expected versions, failed attribute, and applied count. The queue owns a copied snapshot.	Migrate version mismatches before resubmission; reacquire stale entities; repair payload/schema failures. Treat `QueueFull` as back-pressure. Never apply only part of a failed save.
Attribute Change	Invalid world, unset entity, invalid attribute ID, non-finite magnitude, or `QueueFull`.	Stale entity, missing schema/fragment, retired attribute, invalid calculation, or another explicit attribute result.	Request ID, complete entity index/serial, attribute ID, submitted operation and logical source, old/new values, and clamp state.	Clear pending UI on completion. Reacquire stale entities; repair schema/archetype/ID failures. On `QueueFull`, defer or shed work and tune limits only from measurements.

Queue	Immediate receipt failures	Execution-time failures	Correlation preserved	Required recovery
Gameplay Tag	Invalid world, unset entity, invalid/unregistered tag, invalid count/operation, or `QueueFull`.	Stale entity, missing fragment, insufficient removal count, count overflow, or no free tag slot.	Request ID, complete entity index/serial, operation, tag, and structured count change.	Reacquire stale entities or correct the reported tag/count/capacity issue. On `QueueFull`, defer or shed work rather than spinning.
Instant Effect	Invalid world, unset entity, invalid/empty effect, or `QueueFull`.	`InvalidEffect` for missing/unloadable, redirected-to-incompatible, or stable-ID-mutated definitions; otherwise precise entity, attribute, parameter, or tag failure.	Request ID, source/target entity generations, original Effect ID, logical source ID, and structured application result-even when execution fails before mutation.	Do not retry an invalid definition blindly. Keep the original ID in diagnostics, repair/re-cook the asset or redirect, run Project Health, then submit a new request.
Persistent Effect	Invalid world, unset apply target, invalid definition, invalid removal handle, or `QueueFull`.	Changed/unloadable definition, stale owner, missing fragments/schema, stack/tag/attribute transaction failure, or already-absent exact handle.	Request ID, Apply/Remove operation, complete entity generation, submitted/resolved Effect ID, exact handle, captured effect context, and full apply/removal result. Apply context is retained through pre-mutation failures; removal captures the active context at queue admission when available.	Reacquire stale targets; discard terminal handles; repair the precise asset/archetype/transaction failure. On `QueueFull`, defer or shed work rather than spinning.
Damage	Invalid world, invalid definition, unset target, required source omission, non-finite request, or `QueueFull`.	`InvalidDefinition` for missing/unloadable or ID-mutated definitions; `InvalidSource` / `InvalidTarget` for stale endpoints; otherwise the full policy, attribute, tag, and transaction diagnostics.	Request ID plus original Damage ID, source, target, requested magnitude, and context.	Re-resolve endpoints or correct the reported definition/policy/attribute/tag. Never reinterpret a failed hit as success.
Ability	Invalid world, unset source/entity, invalid Grant/Activate definition, empty Revoke ID, no active lifecycle, blocked Cancel/Interrupt, or `QueueFull`.	`InvalidAbility` for missing/unloadable or ID-mutated Grant/Activate definitions; `InvalidEntity` / `InvalidTarget` for stale endpoints; `NoActiveAbility` / `LifecycleChanged` for safely stale lifecycle control; otherwise exact ownership, cooldown, charge, targeting, requirement, cost, effect, and lifecycle diagnostics.	Request ID, operation, complete source/target handles, original Ability ID, captured lifecycle ID, terminal result, and activation details. Ability check/activation results themselves retain the submitted source and optional target on success or failure. AI tickets retain activation data while tracked.	Replan from the terminal result. Reacquire entities, repair the asset, or inspect the current lifecycle before issuing new control; do not reuse a completed request ID.
Death Destruction	Created only by successful lethal damage. `Death Handling Queue Full` is returned in the killing damage result with death-handling request ID `0`.	`Entity Already Invalid` when another owner already retired the exact generation.	Positive death-handling request ID, complete entity handle, Damage ID, and context.	Fall back to the project's pooling/deactivation cleanup when scheduling is rejected; treat already-invalid completion as safe terminal cleanup.

`QueueFull` is a load-shedding signal, not permission to spin. A producer should stop submitting that burst, aggregate replaceable work where gameplay rules allow, spread independent work over later frames, and record queue pressure. Only then should a project raise **Max Pending** or **Per Frame** limits and re-profile frame time and

memory.

When a world is replaced or shut down, its entity handles and request IDs are permanently invalid. Clear project-owned pending indicators from `EndPlay`, world-cleanup, or equivalent ownership teardown; never wait for a completion from a destroyed world or carry its ID into a replacement world.

The complete generation-reuse, representation-change, and teardown ownership rules are defined in the entity lifecycle safety contract.

Timing boundary

The coordinator waits whenever the Mass entity manager is processing. It resumes on a later tick when direct access is safe. The Persistent Effect subsystem does not tick independently; the coordinator advances it as the fifth phase. Regeneration remains a Mass processor whose ordering is governed by Mass execution phases rather than this world-subsystem sequence.

Paused and dilated gameplay time, world replacement, and the current durable-save boundary follow the time, travel, and save contract. Network authority and cross-request atomic batches remain separate roadmap items and must not be inferred from queue acceptance.

Automated proof

The automation suite verifies per-queue back-pressure and zero-ID rejection, positive-ID completion correlation, deep snapshot-copy ownership, complete-state/Attribute/tag phase order, counted-tag, Persistent Effect, and unified Ability-operation FIFO/per-frame budgeting, stale source/target generation safety for every queue family, execution-time schema drift, atomic failed restores, exact lifecycle-ID protection, raw-tag-before-Instant-Effect ordering, queued persistent removal before same-boundary pulses, projectile-impact-before-damage ordering, submitted stable-ID preservation when definition identities change, per-frame damage and death-destruction budgeting, callback-enqueued next-tick behavior, same-phase ability conflicts, independent-source requests, coordinator-owned Persistent Effect scheduling, safe stale-destruction completion, teardown discard, and the fixed eleven-phase cross-category order.

Transaction and event contract

Mass Forge uses explicit, bounded transaction scopes. Atomic means every state write inside the named scope succeeds together or none of those writes remain. It does not mean unrelated calls, an arbitrary list of entities, project-owned Actor state, or later lifecycle work are silently included.

Supported atomic scopes

Operation	Atomic state boundary	Outside the boundary
Raw attribute change	One attribute on one Mass entity.	Other calls, even in the same frame.
Instant Effect	Every authored attribute and counted Gameplay Tag modifier on one recipient entity. Repeated modifiers for one attribute use projected authored order.	Other effect calls and ordinary Actor receiver callbacks.
Damage	Health, optional Shield, and the first-death tag on one Mass target.	Kill listeners and the later keep/deactivate/destroy policy are post-commit consequences; a destruction-queue failure does not undo lethal damage.
Ability activation	Reserved charge, individual/shared cooldown, costs on the source, self/target Instant Effects, and self/target Persistent Effect applications or cancellations across the source and one target.	Later channel pulses, unrelated activations, and any third or subsequent target.
Channel pulse	All self/target Instant Effects authored for that one pulse.	Earlier/later pulses and the original activation cost.
Persistent Effect application	The selected target's stacking decision, cancellations, lifecycle tags, persistent modifiers, and optional initial pulse.	Future periodic pulses, expiration, and other entities.
Snapshot restore	Every restorable attribute in one entity snapshot after version/schema validation.	Other entities and project-owned save data.

Every atomic operation prevalidates against projected copies or captures reversible state before publishing success. A failure returns its detailed result without publishing provisional success events. Ability rollback also restores provisional persistent handles, cancellations, tag counts, attributes, cooldowns, charges, and handle allocation.

Multi-entity rule

One ability may atomically coordinate exactly its source and one target because its payment and outcome policy are authored together. Mass Forge deliberately has no implicit transaction over an arbitrary target-selection array.

Radius, trace, collision-hit, and custom-provider selection are read-only. If a Blueprint or project system loops through ten selected candidates, each activation, effect, or damage call is a separate transaction. Candidate one can succeed and candidate two can fail; earlier successful targets are not rolled back. Queued requests are also independent, even when submitted consecutively in the same frame.

Choose the policy explicitly for an area or chain mechanic:

- **Independent outcomes:** submit one operation per target and retain every receipt. This permits partial success and is the normal scalable choice.
- **Preflight then execute:** validate every target first, then submit in deterministic order. State can still change between validation and execution, so this reduces predictable failures but is not atomic.
- **Project-owned all-or-nothing batch:** implement a bounded native coordinator that captures and validates all participating state, commits it without callbacks, and publishes only after success. Do not emulate this by applying public operations and trying to reverse them; listeners may already have observed those commits.

An area ability must also decide whether cost/cooldown is paid once or once per target, how targets are ordered, and what happens when one becomes stale. Mass Forge does not guess these game-specific rules.

Event visibility

For one successful Instant Effect, all attribute and tag fragments are committed before the first `OnAttributeChanged`, `OnGameplayTagChanged`, or `OnInstantEffectApplied` listener runs. Authored result arrays retain operation order, while a listener reading current state sees the final transaction state rather than a partially applied intermediate value.

Ability activation extends that protection across the source and target. Attribute, tag, and Persistent Effect events are buffered while the transaction is provisional. The committed lifecycle flag, costs, cooldowns, charges, self/target attributes and tags, and persistent state are all visible before buffered listeners run. A rollback discards the buffered events.

Callbacks may query the complete committed state, enqueue follow-up work, cancel the exact active lifecycle, or begin a replacement lifecycle. Identity checks prevent the outer execution frame from continuing against a callback-created replacement. Work enqueued from a completion callback cannot recursively flush the coordinator; it starts on the next eligible safe tick.

Direct operations invoked by a listener are new transactions. They never join or retroactively extend the transaction whose event triggered the callback.

Deterministic publication order

Within an Instant Effect, changed-attribute events follow authored modifier order, changed-tag events follow authored tag order, and the complete Instant Effect event follows them. No-op attribute writes do not emit a changed-value event.

Within an ability commit, buffered attribute events are released first, then counted-tag events, then Persistent Effect application/removal/stack/pulse events, followed by ability activation and committed-lifecycle events. This order describes observation after state is complete; it is not the order in which consumers should recreate gameplay state.

The world coordinator processes **Entity State Restores** → **Attribute Snapshot Restores** → **Attribute Changes** → **Gameplay Tag Changes** → **Instant Effects** → **Persistent Effects** → **Projectile Impacts** → **Damage** → **Death Handling** → **Ability Lifecycles** → **Ability Requests**. Requests are FIFO inside each queue. Each request is its own transaction, so a later request observes earlier successful commits. Snapshot attribute events precede restore completion, and a same-tick raw change observes the restored base state. The deterministic gameplay ordering contract defines same-boundary restore, projectile, pulse, expiration, cancellation, lifecycle, and death precedence.

Actor adapters

A represented Actor resolves to its current Mass entity and uses the same Mass transaction as the entity-handle API. An ordinary Actor receiver is project-owned code called through the receiver interface/component. Mass Forge can report acceptance or rejection, but it cannot snapshot or roll back arbitrary Actor, Blueprint, inventory, animation, or external-system state. Combine cross-system state only in a project coordinator with an explicit compensation or authoritative commit design.

Verification contract

The automation suite proves:

- an invalid modifier or tag operation leaves a multi-attribute Instant Effect unchanged and publishes no provisional events;
- the first direct attribute listener sees all attributes in the completed Instant Effect;
- an ability listener sees source cost, every target attribute, and committed lifecycle state together;
- late ability failure restores source/target attributes, tags, persistent effects, cooldowns, shared cooldowns, charges, and provisional events;
- callback-enqueued work waits for a later coordinator tick;
- callback cancellation/replacement cannot let an old lifecycle frame mutate the replacement;
- two queued operations against one target commit independently in FIFO order.

Treat a change to any boundary above as a public behavioral change: update tests, this guide, the relevant system guide, the changelog, and the versioning decision together.

GAMEPLAY SYSTEMS

Abilities

Mass Forge abilities are shared Data Assets executed by actorless or represented Mass entities. Per-entity state contains only stable ability IDs, cooldown timestamps, and optional charge state.

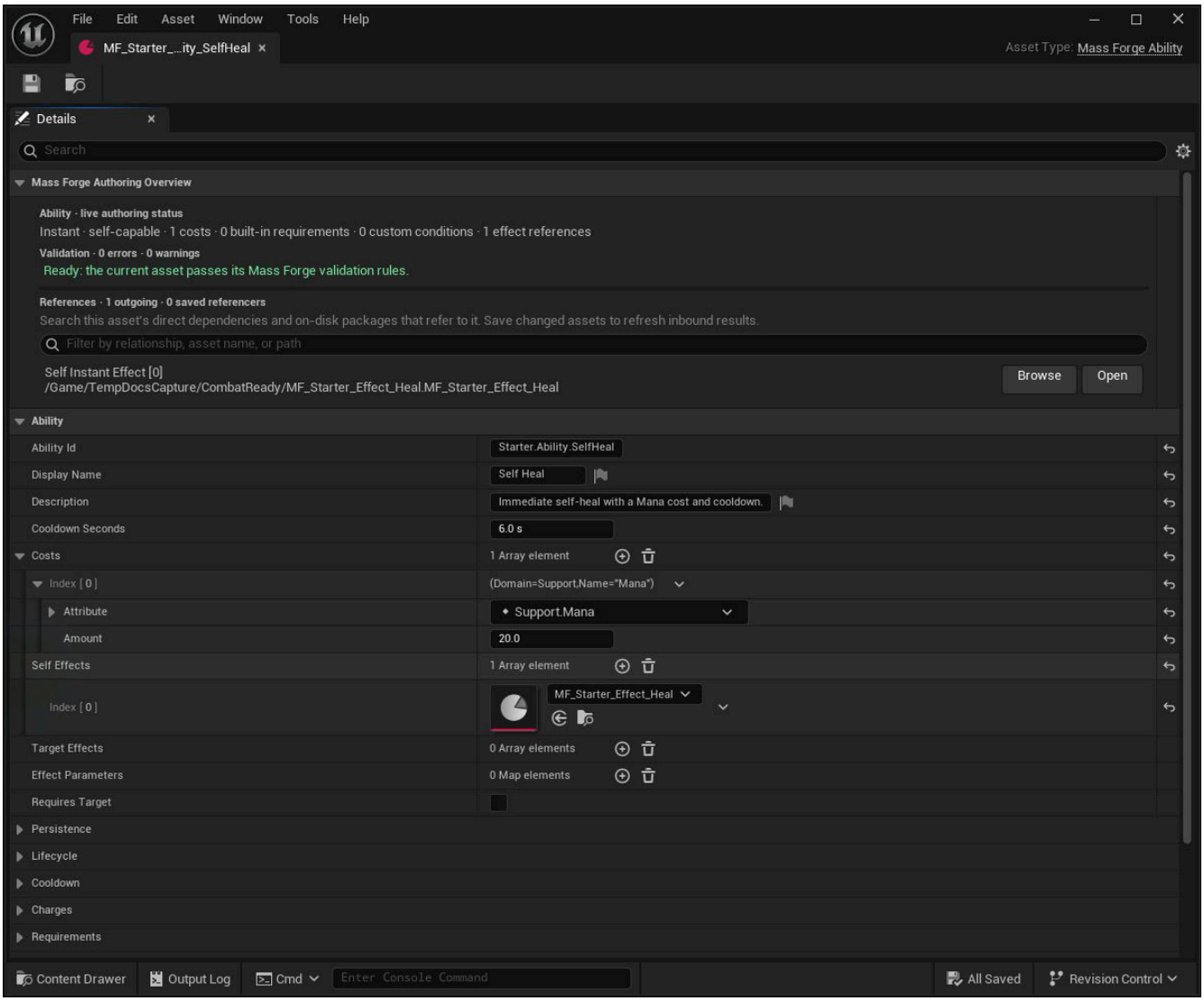
For project-specific read-only admission rules beyond built-in attributes, tags, targeting, cooldowns, charges, and costs, see the ability condition extensions guide.

For project-specific dynamic effect selection that still uses the normal bounded atomic transaction, see the ability execution plans guide.

Opening the asset shows a live execution summary, validation results, and searchable policy/effect dependencies plus saved referencers. See GAMEPLAY_ASSET_AUTHORING_GUIDE.md.

Setup

1. Create **Mass Forge Instant Effect** and/or **Mass Forge Persistent Effect** assets for the ability's outcomes.
2. Create a **Mass Forge Ability** Data Asset and assign a stable `AbilityId`.
3. Configure costs, individual/shared cooldowns, execution type, self/target instant effects, self/target persistent effects, optional channel effects, default effect parameters, and an optional execution plan.
4. Add **Mass Forge Ability Trait** to the Entity Config and assign starting abilities, or grant them at runtime. Entities receiving persistent outcomes also need **Mass Forge Persistent Effects Trait** and, when tags are used, **Mass Forge Gameplay Tag Trait**.



This generated example is a complete small ability: it costs 20 Mana, applies the authored Heal effect to the source, and starts a six-second cooldown. Change the assets and numbers; the execution contract stays the same.

`Can Activate Mass Forge Ability` reports grant, target, requirement, tag-gate, cooldown, charge, affordability, and active-lifecycle failures without changing state. `Activate Mass Forge Ability` commits Instant abilities immediately or begins a configured Cast/Channel lifecycle. The eventual activation transaction commits attribute, counted Gameplay Tag, and persistent-effect changes atomically. A failed transaction restores individual cooldown, shared cooldown, charges, attributes, tags, active persistent instances, cancellations, and deterministic handle allocation. See ABILITY_LIFECYCLE_GUIDE.md for timing and stop semantics.

*A granted ability is compact per-entity ownership state; the shared Data Asset remains the authored definition. Add the side-effect-free **Can Activate** query when UI or AI needs the exact rejection reason before activation.*

Casts and channels

Set `ExecutionType` to `Cast` with a positive cast time, or `Channel` with positive duration and period values. Cast/channel requests are checked when they start and fully revalidated before costs, charges, cooldowns, and activation effects commit. One lifecycle can be active per entity.

Channel self/target effects are separate from activation effects. They execute as an atomic transaction on every scheduled pulse without repeating the ability cost. The lifecycle API exposes state, remaining time, pulse count, cancel and interrupt operations, Actor convenience nodes, and presentation-neutral events. The unified entity inspection snapshot and editor inspector show the active lifecycle.

Attribute requirements

Add entries to `Requirements` when activation depends on current gameplay state rather than spending a resource. Each entry selects the source or target entity, an attribute, one of six comparisons, and the required value. Equal and Not Equal comparisons also expose a non-negative tolerance.

Examples include source `Health > 0`, source `Mana >= 20`, target `Health <= 30`, or target `Shield == 0` within a chosen tolerance. A target-side requirement automatically makes the target pin mandatory even when the ability has no target effects.

A failed comparison returns `Requirement Not Met` without reserving cooldowns or charges and without applying costs or effects. The activation result identifies the zero-based requirement index, source/target subject, attribute, observed value, and required value, allowing Blueprint UI or AI logic to explain the failure without parsing text. An

unresolvable requirement attribute returns the corresponding attribute failure with the same requirement identity fields.

Gameplay Tag requirements

Add **Mass Forge Gameplay Tag Trait** to entities that own tags. Ability assets expose required and blocked tag containers for both source and target. Every required tag must match; any blocked tag rejects activation. Target tag rules automatically require a valid target. Failures return `Missing Gameplay Tag Fragment`, `Missing Required Tag`, or `Blocked By Tag` plus the exact tag and source/target subject.

Matching is hierarchical, so requiring a parent tag accepts an owned child. Tag gates run before any cooldown, charge, cost, or effect state is changed. See GAMEPLAY_TAGS_GUIDE.md.

Individual and shared cooldowns

`CooldownSeconds` controls only the activated ability. To lock several abilities together, give them the same non-empty `CooldownGroup` and a positive finite `CooldownGroupSeconds` value. Activating any member starts that named group; another member returns `On Cooldown Group`, the group ID, and its remaining time until the lock expires.

- Leave the group name empty and its duration at zero to disable shared cooldowns.
- Use one common group ID on every ability to author a global cooldown.
- Different abilities in a group may author different durations; the ability that successfully activates supplies the new group duration.
- `Get Mass Forge Cooldown Group Remaining` lets Blueprint UI query a group without referencing a particular ability.
- An entity tracks at most eight named groups at once. Expired records are reused deterministically, so historic groups do not grow per-entity memory.
- A ninth group returns `No Free Cooldown Group Slot` only while all eight tracked groups are still active.

Individual cooldowns, shared cooldowns, and charges are independent gates. All enabled gates must be ready before an activation can succeed.

Charges

Charges are optional. Leave `MaxCharges` at zero to disable them. To enable charges, set `MaxCharges` from 1 to 255 and set `ChargeRecoverySeconds` to a finite value greater than zero.

- A grant begins at full capacity.
- Each successful activation consumes one charge.
- Recovery proceeds one charge at a time using absolute world time.
- Several elapsed recovery intervals are processed deterministically during the next query or activation.
- The recovery clock stops when capacity is full.
- An exhausted ability returns `No Charges` and reports the next recovery time.
- `Get Mass Forge Ability Charge State` exposes current charges, maximum charges, and recovery remaining to Blueprint.

Charges can be combined with both cooldown types. This permits burst charges with a short per-use cooldown, a shared global lockout, or slower strategic recovery.

`Get Granted Mass Forge Abilities` enumerates compact runtime state in lexical ability-ID order without loading definition assets. `Get Active Mass Forge Cooldown Groups` lists only groups with positive remaining time. Both are also included in the unified entity inspection snapshot.

Targets and effects

Enable `Requires Target` when the ability must receive a valid target distinct from its source. Target instant, persistent, or channel effects also imply that a target is needed. Self effects apply to the source entity; target effects apply to the selected target.

`Targeting Policies` provide reusable validation after basic handle checks and before tags, cooldowns, charges, requirements, costs, or effects. Every policy must allow the pair. The built-in range policy works with actorless Mass transforms and optional represented-Actor fallback; Blueprint policy subclasses can return a stable project-specific rejection reason. Policy failures identify the authored index, policy ID, detailed result, and evaluated distance where applicable. See TARGETING_POLICIES_GUIDE.md.

Selection is a separate concern. Mass Forge supplies bounded actorless radius and transform-segment searches, collision-hit conversion for represented Actors, Actor-based ability nodes, and a sanitized custom provider interface. See TARGET_SELECTION_GUIDE.md.

Requirements are read-only gates. Costs and all referenced attribute/tag modifiers form one transaction after every requirement passes. If a requirement fails, a cost is unaffordable, an attribute is unknown, a tag operation cannot complete, an effect is invalid, or a required magnitude parameter is absent, no cost or effect state is committed. Tag-operation failures identify the tag, source/target subject, and exact reason. See INSTANT_EFFECTS_GUIDE.md.

Instant and persistent modifiers may also derive magnitude from one source or recipient attribute. Every modifier in the atomic cost/Instant Effect stage samples the stage's pre-commit state. Persistent captures occur when that persistent instance reaches its documented self-then-target application position; earlier provisional ability outcomes are therefore visible consistently, while the incoming instance's own lifecycle changes are not. Persistent periodic snapshots retain that value and live periodic captures reevaluate at each pulse. Failures expose the sampled attribute, Source/Target subject, and underlying access result.

Dynamic execution plans

An optional **Mass Forge Ability Execution Plan** can append to or replace the fixed effect lists for activation and channel pulses. It receives source/target entity handles, optional represented Actors, stable Ability/lifecycle identity, phase, one-based pulse number, and authored parameters. It returns only a bounded transaction description; Mass Forge validates every effect and parameter and commits it through the existing atomic path.

Plans are preflighted by `Can Activate` and re-evaluated at actual commit, so Blueprint/C++ implementations must be deterministic, side-effect-free, and safe to call more than once. Target output requires **Requires Target**. Channel-pulse output is Instant-only. See ABILITY_EXECUTION_PLANS_GUIDE.md for merge order, limits, failure diagnostics, and the data-only Static plan.

Persistent outcomes and transaction order

Use `SelfPersistentEffects` and `TargetPersistentEffects` for damage-over-time, healing-over-time, timed buffs/debuffs, temporary state tags, immunity-aware applications, and cleanse or dispel abilities. The ability's effect parameters and source identity are copied into every persistent context for later pulses.

Successful activation uses a deterministic order: costs plus self/target Instant Effects are calculated first, followed by self Persistent Effects in authored order, then target Persistent Effects in authored order. Persistent-effect handles are returned in that same self-then-target order. The result also reports the total number of existing instances cancelled by the activation.

This is one observable transaction. If any later persistent application fails-including a missing trait, immunity blocker, stack/capacity rejection, invalid asset, tag-capacity failure, or failed execute-on-application pulse-Mass Forge restores every earlier part of the activation. No provisional attribute, tag, persistent apply/removal/stack, or pulse event is published. The failure returns `Persistent Effect Failed`, the underlying persistent result, the source/target subject, the zero-based list index, and relevant tag or magnitude-parameter details.

On success, all state is committed before buffered events are released. Blueprint delegates remain available, and the attribute, tag, and persistent subsystems expose native C++ mirrors for direct engine-side binding.

Safe execution

Direct mutations are rejected while Mass is processing. Use the queued Activate, Grant, Revoke, Cancel, or Interrupt node when a Blueprint or external system can overlap a Mass phase. All five operations share one bounded FIFO and configurable back-pressure limit. Accepted receipts return a positive request ID; `On Ability Operation Request Completed` reports the operation and terminal result. Queued Cancel/Interrupt captures an exact lifecycle ID and cannot stop a later replacement lifecycle. The activation-only `On Ability Request Completed` event remains available for compatibility.

StateTree and tick-driven AI can instead use `Request Mass Forge Ability Activation for AI`. It adds a bounded pollable ticket over the same queue, with `Pending`, `Active`, `Succeeded`, `Failed`, `Cancelled`, and `Interrupted` states plus detailed activation/lifecycle data. Pending admission and active lifecycles can be cancelled through the ticket; terminal records can be explicitly forgotten and are otherwise evicted oldest-first when capacity is needed. See AI_STATETREE_GUIDE.md.

Abilities are the final phase of the shared external-command coordinator, after queued raw changes, Instant Effects, Persistent Effects, damage, and death handling. Existing ability lifecycles advance before queued Activate/Grant/Revoke/Cancel/Interrupt operations begin, preventing a newly admitted cast from consuming an earlier frame's delta and resolving natural lifecycle completion before queued control. See the deferred command contract for same-frame and callback-enqueue behavior.

Requests admitted to one ability phase execute FIFO. Each request validates against the state committed by every earlier request: two same-source abilities sharing a cooldown therefore produce one success followed by `On Cooldown Group`, while independent sources may both affect the same target in deterministic request order. A completion listener may queue more ability work, but that follow-up waits for the next safe coordinator tick even if the listener calls the coordinator recursively.

Successful ability state is complete before attribute, Gameplay Tag, persistent-effect, or ability commit listeners are notified. Cast/channel state carries a positive world-local `Lifecycle Id`; internal lifecycle processing checks that identity after every callback-capable operation. If a listener cancels a lifecycle and starts a replacement on the same entity, the outer cast completion or channel loop cannot consume, complete, or otherwise mutate the replacement.

Current limits

Each entity can hold 16 granted abilities, track eight named cooldown groups, run one active cast/channel lifecycle, and-when opted in-hold 16 active persistent instances. Multiplayer transport and prediction are outside the initial supported release.

GAMEPLAY SYSTEMS

Ability lifecycle

Mass Forge supports instant abilities, delayed casts, and periodic channels without requiring an Actor or UObject per entity. One cast or channel may be active on an ability-enabled entity at a time; compact grant, cooldown, and charge data remains in the Mass fragment, while only currently active lifecycle definitions are retained by the world subsystem.

Authoring modes

Choose `Execution Type` on a **Mass Forge Ability** Data Asset.

Mode	Required settings	Commit point	Periodic output
Instant	No lifecycle timing	Inside `Activate Mass Forge Ability`	None
Cast	Positive `Cast Time Seconds`	When the cast finishes	None
Channel	Positive `Channel Duration Seconds` and `Channel Period Seconds`; cast time may be zero	Before channeling begins	`Channel Self Effects` and `Channel Target Effects`

Instant preserves the original synchronous behavior. A cast/channel activation request first runs the complete grant, target, policy, tag, cooldown, charge, requirement, and affordability check. If accepted, `bLifecycleStarted` is true in the activation result and the active state begins in `Casting`.

Commit and cost rules

Costs, charges, individual/shared cooldowns, activation Instant Effects, and activation Persistent Effects commit together only after the cast delay ends. The complete activation is revalidated at that point. A target that vanished, a newly blocking tag, an exhausted resource, or another changed rule produces `Commit Failed` without a partial transaction.

- Cancelling or interrupting before commit spends nothing.
- After a channel commits, stopping it prevents future pulses but does not refund its already committed cost, charge, cooldown, or activation outcomes.
- A cast completes immediately after its one atomic commit.
- A channel commits once, then each pulse is a separate atomic self/target Instant Effect transaction. Ability costs are not repeated per pulse.
- `Execute Channel Tick On Start` adds a pulse immediately after a successful channel commit. It does not replace the regularly scheduled pulses.

This contract deliberately avoids implicit refunds after gameplay effects may already have been observed.

Blueprint control and state

Use these entity-handle nodes:

- `Get Active Mass Forge Ability Lifecycle`
- `Cancel Active Mass Forge Ability`
- `Interrupt Active Mass Forge Ability`
- `Queue Active Mass Forge Ability Cancellation`
- `Queue Active Mass Forge Ability Interruption`

Matching `...from Actor` nodes resolve represented Mass entities for Actor-driven UI and gameplay. The lifecycle state reports a positive world-local lifecycle ID, ability ID, target handle, `Casting` or `Channeling` phase, remaining phase time, completed channel-pulse count, and whether the activation transaction has committed. Idle entities return `No Active Ability` with lifecycle ID zero. The ID distinguishes successive instances of the same ability on the same entity; it is runtime coordination data, not a persistent or replicated identifier.

Safe queued ownership and lifecycle control

Every common mutating Ability operation has a bounded deferred counterpart:

Direct operation	Safe queued operation
`Grant Mass Forge Ability`	`Queue Mass Forge Ability Grant`
`Revoke Mass Forge Ability`	`Queue Mass Forge Ability Revocation`
`Activate Mass Forge Ability`	`Queue Mass Forge Ability Activation`
`Cancel Active Mass Forge Ability`	`Queue Active Mass Forge Ability Cancellation`
`Interrupt Active Mass Forge Ability`	`Queue Active Mass Forge Ability Interruption`

All five operations share one FIFO, one pending limit, one per-frame budget, and positive world-local request IDs. Bind `On Ability Operation Request Completed` and match its request ID. The completion identifies the operation, full source/target generations, submitted Ability ID, captured lifecycle ID where applicable, terminal result, and detailed activation result for Activate. The older `On Ability Request Completed` event remains available for Activate requests only.

Grant and Activate retain a soft definition reference plus its stable ID, then reload and revalidate both at execution. Revoke retains the submitted stable ID. Cancel and Interrupt capture the exact positive lifecycle ID observed at submission, so delayed control cannot end a later replacement cast/channel. `No Active Ability` means the captured lifecycle ended before execution; `Lifecycle Changed` means another lifecycle replaced it. Rejected submissions have request ID zero and never emit completion.

Successful grant/revoke/lifecycle events publish before the operation completion. Work enqueued by an Ability-operation completion callback waits for the next Ability Requests phase even when budget remains, preventing an unbounded callback flush.

For a progress bar, VFX controller, audio controller, or Blueprint-only HUD, use the async `Observe Mass Forge Ability Lifecycle` node with an entity handle and optional ability ID. It observes an already-active matching lifecycle or waits for the next one, then exposes Started, Committed, Channel Tick, Ended, and Failed execution pins. It automatically releases its subsystem bindings after the matching lifecycle ends. Calling `Cancel Observation` only releases the observer; it never cancels the gameplay ability.

Starting another ability while a lifecycle is active returns `Active Ability In Progress`. `Can Be Cancelled` and `Can Be Interrupted` are independent authoring switches; blocked requests return `Cancellation Blocked` or `Interruption Blocked`. Revoking the definition currently being used force-ends that lifecycle with the `Ability Revoked` reason regardless of the interruption switch.

Events

Bind on **Mass Forge Ability Subsystem** when presentation or project rules need lifecycle notifications:

- `On Ability Lifecycle Started` fires after an accepted cast/channel is registered.
- `On Ability Activated` fires only when its atomic activation transaction commits.
- `On Ability Committed` is the detailed companion event and includes the complete activation result plus an immutable lifecycle snapshot captured at commit.
- `On Ability Channel Tick` fires after each attempted atomic pulse and includes the pulse number and detailed result.
- `On Ability Lifecycle Ended` fires after state is removed, with `Completed`, `Cancelled`, `Interrupted`, `Commit Failed`, `Ability Revoked`, or `Source Invalid`.
- `On Ability Activation Failed` still reports failed checks and failed cast-completion commits.
- `On Ability Operation Request Completed` reports every queued Activate, Grant, Revoke, Cancel, or Interrupt terminal outcome; `On Ability Request Completed` remains the activation-only compatibility event.

Every subsystem event has a native C++ mirror with the same payload. The Blueprint event is emitted first and its native mirror immediately afterward.

Lifecycle state is removed before the ended event is broadcast. An event listener can therefore start another ability safely. Presentation remains optional: Actorless entities receive the same events, handles, state, and outcomes.

All costs, cooldowns, charges, attribute/tag outcomes, persistent state, and the lifecycle's committed phase are visible before successful transaction events are released. Callback-created replacement lifecycles are identity-protected: after any event or project-extensible operation, the outer lifecycle loop continues only if the same lifecycle ID is still active. A listener may therefore cancel the committing cast or current channel pulse and start another cast without the old execution frame completing the new one by mistake.

The activation boundary coordinates the source and one target. Each later channel pulse is its own transaction, and arbitrary target-selection arrays are not one implicit transaction. The transaction and event contract defines the complete scope, publication order, and multi-target partial-success rule.

Same-tick costs, effects, channel pulses, cancellation, completion, Damage, and death handling follow the deterministic gameplay ordering contract. In particular, every earned final channel pulse publishes before natural lifecycle completion.

Source destruction, target destruction before commit, recycled generations, representation reassignment, and world shutdown follow the entity lifecycle safety contract.

Durable save and restore

Entity State Snapshot format 4 can preserve one active cast or channel exactly. It stores the guarded Ability asset reference and stable ID, **Save Compatibility Version**, execution type and authored timings, current phase, remaining gameplay time, committed state, completed channel pulses, and earned pulse backlog. Restore allocates a fresh world-local lifecycle ID and does not emit synthetic lifecycle-started or lifecycle-ended events; re-query the active lifecycle and rebind presentation observers after the whole-entity restore boundary.

Keep **Save Compatibility Version** at `1` while saved active phases remain behavior-compatible. Increment it when an Ability change must invalidate old saved lifecycle state. Stable ID, version, execution type, cast time, channel duration, and channel period all have to match at restore, preventing an old phase from continuing under materially changed authoring.

- Targetless and self-targeted lifecycles use the ordinary Entity State capture/restore nodes; self targets rebind to the restored entity.
- For an external target, call `Capture Mass Forge Entity State with Lifecycle Target Reference` with a project-owned relationship key. After recreating both entities, call `Restore Mass Forge Entity State with Lifecycle Target` or its queued variant with that same key and the new target handle.
- Never store the former target handle or lifecycle ID in the project SaveGame. Missing, mismatched, unset, stale, or self-collapsed external bindings fail before mutation.
- Formats 1-3 remain readable but cannot describe an active lifecycle; they reject an active destination lifecycle instead of erasing it.

The complete transaction and migration boundary is defined in the durable entity-state persistence guide.

Channel failures and timing

`Channel Failure Policy` controls a failed pulse:

- `Continue Channel` reports the failed pulse and keeps scheduling.
- `Cancel Channel` ends with `Cancelled`.
- `Interrupt Channel` ends with `Interrupted` and is the default.

The shared deferred-command coordinator advances existing lifecycles before it starts newly queued ability requests. A queued cast therefore never consumes the frame delta that admitted it. Timing uses world delta time, so normal pause and time-dilation behavior follows the host world.

The precise gameplay-clock, Actor custom-dilation, seamless-travel, focused attribute snapshot, and durable entity-state SaveGame boundaries are defined in the time, travel, and save contract.

Large frame gaps preserve earned pulses. `Max Ability Channel Ticks Per Frame` under **Project Settings > Mass Forge > Global Attribute Schemas > Runtime** bounds catch-up work per active ability; remaining earned pulses carry into later frames instead of being discarded. Active entities are processed in stable entity-handle order.

Practical recipes

For a cast-time projectile or heal, use `Cast`, put the final effects in the normal self/target lists, and drive a progress bar from `Remaining Time`. Cancel from player input and interrupt from damage, stun, or project-owned crowd-control logic.

For a beam, drain, or channeled heal, use `Channel`, put the one-time setup in normal effects and repeated output in channel effects. Use `On Ability Channel Tick` for presentation; simulation authority remains in the atomic effect result.

For a wind-up followed by a channel, set both a positive cast time and the channel duration/period. The state transitions from `Casting` to `Channeling` only after the activation transaction succeeds.

The lifecycle is authoritative within its owning world. Multiplayer transport, prediction, reconciliation, and replicated baselines are outside the initial supported release and must be supplied and validated by the customer project.

GAMEPLAY SYSTEMS

Ability condition extensions

Mass Forge ability conditions are reusable Blueprint/C++ Data Assets for project rules that do not belong in the framework's fixed attribute, tag, target, cooldown, charge, or cost gates. Examples include match phase, quest state, equipment ownership, line-of-command approval, or a project-specific faction relationship.

Create a Blueprint derived from **Mass Forge Ability Condition**, assign a stable `Condition Id`, and implement **Evaluate Condition**. Add condition assets to an Ability's **Activation Conditions** array in the order they should run.

Query and response

The immutable query contains:

- the current world context;
- source and optional target entity handles;
- optional represented source and target Actors;
- the stable Ability ID.

Return a response with `Passed` only when the rule allows activation. Any other condition result rejects the ability as `Condition Rejected`. Set a stable `Reason` for UI, AI, logs, and support diagnostics. Mass Forge fills an omitted response `Condition Id` from the asset, then reports the response and zero-based `Failed Condition Index` in the ordinary activation result. Event History records the top-level `ConditionRejected` outcome while the full activation result carries the detailed condition response.

The native base implementation returns `Invalid Configuration` with reason `NoConditionImplementation`. This fail-closed behavior makes an accidentally unimplemented condition visible instead of silently allowing gameplay. It can also be used intentionally as a data-authored kill switch.

Evaluation contract

Mass Forge evaluates built-in target, tag, cooldown, charge, attribute-requirement, and affordability gates first. It then evaluates custom conditions in authored order. The first failure stops evaluation. A successful activation re-evaluates every condition during the final commit check, including after a cast delay, so a changed project state cannot reuse an earlier approval.

Condition implementations must be:

- deterministic for the supplied query and current authoritative world state;
- side-effect free-do not spend resources, add tags, activate abilities, dispatch effects, or alter external state;
- safe to execute more than once;
- bounded and fast enough for the expected activation rate;
- explicit about unset targets and actorless entities.

Re-entering ability admission from condition evaluation is rejected as `Condition Evaluation In Progress`. This guard prevents recursive evaluation; it does not make side effects safe. Use the queued Mass Forge APIs after the current activation completes when a later gameplay action is required.

Blueprint workflow

1. Create a Blueprint class based on **Mass Forge Ability Condition**.
2. Implement **Evaluate Condition** and branch only on read-only project state.
3. Return `Passed`, or a specific failure result plus a stable reason such as `Match.NotStarted`.
4. Create a Data Asset from that class and give it a stable `Condition Id` such as `Condition.MatchActive`.
5. Add the asset to the Ability's **Activation Conditions** list.
6. In failure UI or AI, inspect `Result`, `Failed Condition Index`, `Condition Response.Condition Id`, and `Condition Response.Reason`.
7. Use the gameplay-asset Details panel to browse the outgoing condition dependency and run **Validate Assets** before packaging.

If a condition requires a target, also enable the Ability's existing **Requires Target** setting. Conditions may inspect a supplied target, but they do not implicitly change the Ability's target requirement or target-selection contract.

C++ workflow

Derive from `UMF_AbilityCondition` and override `EvaluateCondition_Implementation`. Keep the class shared as a definition asset; do not attach one UObject per Mass entity. Query entity state through the public generation-checked subsystem APIs and return stable diagnostics rather than logging as the only failure channel.

Current extension boundary

Reusable targeting-policy assets, custom target-provider objects, inspection providers, damage-policy providers, and ability activation conditions are implemented. Custom magnitude-calculation and transaction-safe execution-plan extensions remain roadmap work; do not mutate gameplay from a condition to emulate either feature.

GAMEPLAY SYSTEMS

Ability execution plans

Ability Execution Plans are optional Blueprint/C++ Data Assets that choose a bounded effect payload for one ability activation or one channel pulse. They extend authored abilities without bypassing Mass Forge validation, rollback, cooldown, charge, cost, event-order, or source-plus-one-target transaction rules.

Use a plan when an ability must choose effects or parameter values from current source/target state. Keep fixed outcomes directly on the Ability asset.

Quick setup

1. Create a **Mass Forge Static Ability Execution Plan** for data-only branching, or derive a Blueprint from **Mass Forge Ability Execution Plan** for project logic.
2. Give it a unique `PlanId`.
3. Choose **Append After Authored Effects** or **Replace Authored Effects**.
4. For a Static plan, fill `ActivationPayload` and/or `ChannelPulsePayload`. For a Blueprint plan, override **Build Execution Plan** and return a response.
5. Assign the plan to the Ability's **Execution Plan** field.
6. Run Unreal Data Validation and **Tools > Mass Forge Project Health**.

Any plan that can return target effects must be used by an Ability with **Requires Target** enabled. This makes targeting requirements visible to designers, AI, UI, and preflight checks before plan evaluation.

Query available to Blueprint and C++

Every evaluation receives read-only context:

- world context;
- source and target entity handles;
- optional represented source and target Actors;
- stable Ability ID;
- `Activation` or `ChannelPulse` phase;
- lifecycle ID for casts/channels;
- one-based channel-pulse number, or zero for activation;
- the Ability's authored effect-parameter map.

Actor pointers may be null for actorless Mass entities. Logic must use the entity handles when Actor representation is optional.

Returned transaction payload

A successful response may return:

- self and target Instant Effects;
- self and target Persistent Effects for the activation phase;

- named effect-parameter overrides.

Output is limited to 64 combined effects and 64 parameter overrides per evaluation. Channel-pulse output supports Instant Effects only; Persistent Effects belong to the activation transaction. Empty output is valid.

With **Append After Authored Effects**, fixed Ability effects execute first in authored order, followed by plan effects in returned order. With **Replace Authored Effects**, plan lists replace all fixed effect lists for that phase. In both modes, plan parameters override authored parameters with the same name and leave other authored parameters intact.

Evaluation timing

- `Can Activate Mass Forge Ability` evaluates the activation plan as a side-effect-free preflight.
- An Instant ability evaluates again immediately before its atomic commit.
- A Cast or Channel evaluates at admission, then evaluates again when the delayed activation transaction commits.
- A Channel evaluates its channel-pulse payload once for every earned pulse. `ChannelPulse` is one-based and remains stable during bounded catch-up.

Plans must therefore be deterministic for the same query and safe to evaluate more than once. Do not spend resources, apply effects, grant abilities, start another ability, broadcast gameplay events, or otherwise mutate gameplay from **Build Execution Plan**. Return a description and let Mass Forge commit it.

Validation and atomic failure behavior

Mass Forge validates the plan and every returned reference before changing gameplay state. It rejects:

- missing, unloadable, or unidentified plans/effects;
- invalid worlds, sources, targets, merge modes, names, or non-finite parameter values;
- recursive plan evaluation;
- output beyond either fixed limit;
- target output without a valid required target;
- Persistent Effect output during a channel pulse.

Activation failures return `ExecutionPlanRejected` with an `ExecutionPlanResponse` containing `PlanId`, the detailed result, and optional stable `Reason`. Recursive admission returns `ExecutionPlanEvaluationInProgress`. A rejected activation commits no costs, effects, cooldowns, charges, persistent instances, lifecycle state, or gameplay events. If a later effect in a valid payload fails while staging, the existing ability transaction rollback rules apply to both authored and planned output.

A failed channel pulse follows the Ability's `ChannelFailurePolicy`; earlier completed pulses are separate transactions and are not rolled back.

Blueprint implementation pattern

In a Blueprint-derived plan:

1. Override **Build Execution Plan**.
2. Branch on `Phase` first.
3. Read only the supplied handles/Actors and stable project state.

4. Build one payload in deterministic array order.
5. Return `Success`, the plan's stable ID, and the payload.
6. For expected gameplay denial, return `Rejected` with a stable reason. For bad setup, return `InvalidConfiguration`.

The base class intentionally returns `InvalidConfiguration` with `NoExecutionPlanImplementation`, so an unimplemented asset fails closed. The Static subclass is the recommended no-code example and provides separate activation and channel payloads.

Scope and high-volume boundary

One plan describes one source plus at most one target. Radius, segment, provider, or collision selection remains a separate targeting step, and each selected activation remains an independent transaction. A plan is shared definition logic, not a per-entity UObject.

Execution plans run at ability admission/commit or channel-pulse frequency. They are not a Mass processor hot loop and should not be used to iterate an entire population. For uniform work across large archetypes, use compiled slots and Mass processors described in CPP_HIGH_VOLUME_GUIDE.md and HIGH_VOLUME_EFFECTS_GUIDE.md.

Support checklist

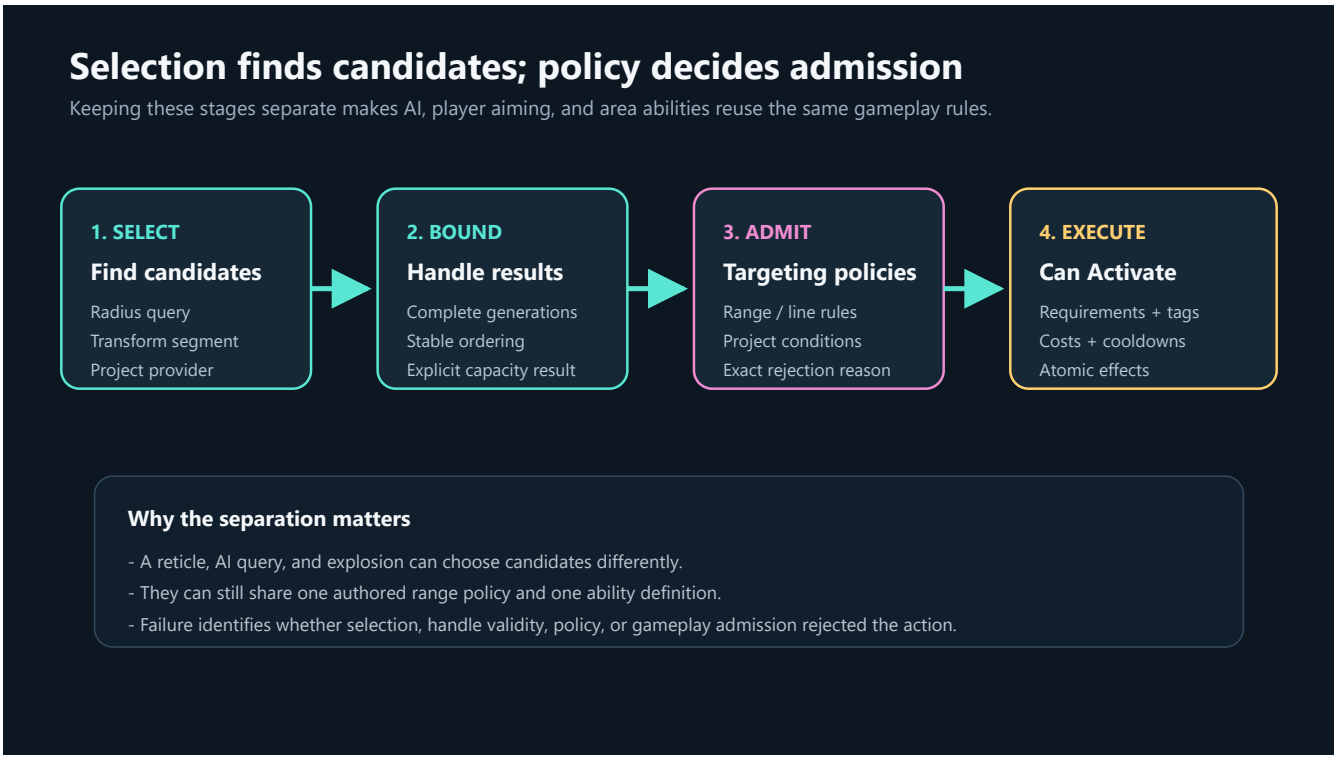
- Give every plan and returned effect a unique stable ID.
- Keep output under both limits and use finite parameter values.
- Enable **Requires Target** when target output is possible.
- Treat Blueprint callbacks as pure decision functions.
- Test both preflight and delayed commit when state may change during a cast.
- Test every channel failure policy used by the project.
- Inspect nested execution-plan diagnostics before reporting a transaction bug.
- Include the Ability, plan, effects, Project Health report, and reproduction query in support requests.

See ABILITIES_GUIDE.md, ABILITY_LIFECYCLE_GUIDE.md, GAMEPLAY_ASSET_AUTHORING_GUIDE.md, and TRANSACTION_AND_EVENT_CONTRACT.md.

GAMEPLAY SYSTEMS

Ability target selection

Mass Forge separates target selection from target validation. Selection adapters produce stale-safe Mass entity handles; an Ability Data Asset's targeting policies, tags, requirements, cooldowns, and costs decide whether a chosen pair can activate.



*Selection answers "which candidates?" while a Targeting Policy and **Can Activate** answer "is this source-target pair allowed now?" Keeping those stages separate lets player input, AI, and area effects reuse the same authored gameplay rules.*

All built-in outputs use `Mass Forge Ability Target Selection`. The result contains a bounded candidate array, total pre-truncation match count, truncation flag, and provider diagnostics where applicable. A candidate carries its entity handle, optional represented Actor, location, distance, and trace miss distance.

Radius selection

Use **Find Mass Forge Ability Targets in Radius** with:

- A valid Mass source entity.
- An explicit world-space origin and finite non-negative radius.
- A result limit from 1 to 256; 32 is the default.
- Optional XY-only distance and source exclusion.

The adapter searches entities with Unreal's standard `FTransformFragment`, so it works for actorless Mass entities. Results are nearest-first. Equal distances are tie-broken by entity index and serial number, making the retained set deterministic even when the output limit truncates a larger match set.

This convenience scan is linear in the number of transform-bearing entities. Use a project spatial index through the custom-provider route for frequent high-volume queries.

Trace selection

Trace for Mass Forge Ability Targets performs an actorless-capable thick line-segment query against Mass transform locations. Set the segment start/end, a perpendicular `Trace Radius`, and the same 1-256 result bound. Results are ordered along the segment, then by miss distance and entity identity.

This is a geometric point trace. It does not test world collision, entity radius, cover, or line of sight. For a normal Blueprint or collision-system trace, pass its `Hit Result` to **Get Mass Forge Ability Target from Hit**. That adapter resolves the hit Actor through `UMassActorSubsystem` and returns both its represented Mass handle and Actor. An Actor without current Mass representation is rejected explicitly.

Represented-Actor ability nodes

When both participants are represented Actors, use:

- **Can Activate Mass Forge Ability from Actors**
- **Activate Mass Forge Ability from Actors**
- **Queue Mass Forge Ability Activation from Actors**

The source Actor must currently represent a valid Mass entity. The target Actor may be null for a targetless ability; a supplied target must also have valid representation. After resolution, these nodes use the exact same validation, transaction, queue, and failure contracts as the entity-handle nodes. Queued requests store entity handles and revalidate them at execution time.

Ordinary player Actors that do not represent Mass entities cannot own Mass Forge abilities through these nodes. They can still affect Mass entities through the Actor-to-Entity damage and Instant Effect adapters described in the combat integration guide.

Custom C++ and Blueprint providers

Implement **Mass Forge Ability Target Provider** on a C++ UObject or Blueprint class when selection comes from a spatial database, formation, threat table, encounter manager, party system, or other project-owned source. **Mass Forge Ability Target Provider Component** is a drop-in Blueprint-spawnable host intended for one manager Actor, not every Mass entity.

The provider query supplies the world, valid source entity, optional represented source Actor, origin, trace end, radius, result limit, source-exclusion flag, and optional named scalar parameters. Return a stable provider ID, result/reason, and ordered entity handles.

Call **Select Mass Forge Ability Targets with Provider** rather than trusting the raw provider array. Mass Forge:

- Rejects invalid providers, cross-world provider objects, invalid requests, and recursive provider evaluation.
- Rejects unnamed provider responses.
- Removes stale or unset handles and reports their count.
- Removes duplicates and reports their count.
- Optionally removes the source entity.

- Preserves the provider's order for valid targets.
- Enforces the request's 1-256 bound and reports truncation.
- Resolves each retained candidate's Mass transform or represented-Actor location when one is available.

Provider response fields are Blueprint-writeable, so an override can construct the response directly. Providers must be deterministic and side-effect free; they should select, not activate abilities or mutate entity state.

Production flow

1. Select candidates with a radius, trace, collision hit, or custom provider.
2. Run `Can Activate Mass Forge Ability` for the candidate you intend to use. This reevaluates the ability's current targeting policies and gameplay rules.
3. Activate directly while Mass is idle, or enqueue the activation when the caller may overlap Mass processing.
4. Treat every handle as a snapshot reference. Selection success does not reserve the entity or guarantee it remains valid.

Selection is read-only and returns `Mass Is Processing` rather than racing a Mass phase. There is no queued selection path; schedule the read after Mass work or perform high-volume selection inside a project Mass processor/provider.

Mass Forge does not automatically activate an ability once per radius result. Doing so would silently choose project-specific cost, cooldown, ordering, and multi-target atomicity semantics. Area abilities should select targets, define their intended payment policy, and then apply an authored area effect or explicit project batch operation.

Looping through selected candidates is explicitly non-atomic across the list: every target receives an independent detailed result, and a later failure does not roll back earlier successful targets. See the transaction and event contract before implementing an area, chain, or project-owned all-or-nothing batch.

See TARGETING_POLICIES_GUIDE.md for reusable validation policies and DEFERRED_COMMANDS_GUIDE.md for safe activation ordering.

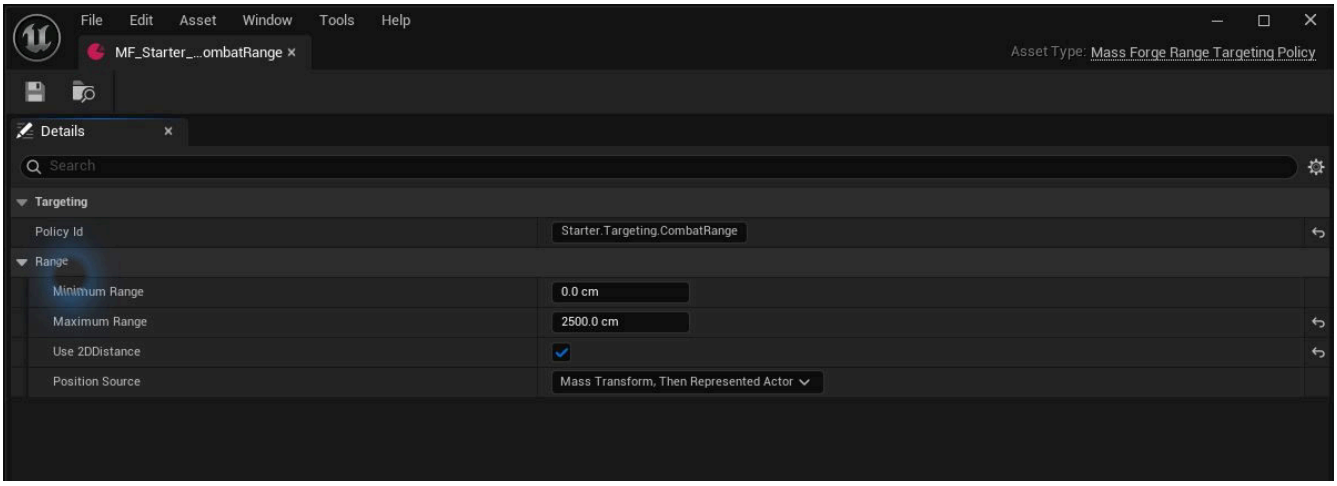
GAMEPLAY SYSTEMS

Ability targeting policies

Mass Forge targeting policies are reusable Data Assets that validate a chosen source/target pair before an ability changes any state. They do not select targets. Projects remain free to choose targets through AI, player input, traces, Mass queries, or their own C++ systems, then pass the selected entity to the normal ability check or activation node.

Authoring workflow

1. For a distance gate, create a **Mass Forge Range Targeting Policy** Data Asset.
2. Give it a stable `PolicyId`, then set inclusive minimum and maximum ranges.
3. Choose three-dimensional or XY-only distance and the desired position source.
4. Add the policy to an Ability Data Asset's `Targeting Policies` list.
5. Call `Can Activate Mass Forge Ability` to preview the complete result, or use the direct/queued activation nodes as usual.



The policy stores only reusable admission rules. This actorless-safe setup measures XY distance from Mass transforms first and falls back to represented Actors; target discovery remains a separate bounded selection step.

For a project-specific rule, create a Blueprint subclass of **Mass Forge Ability Targeting Policy** and override `Evaluate Target`. The query provides the world context, stale-safe source and target entity handles, and optional represented Actors. Return **Allowed** to continue or a non-allowed result, stable policy ID, and optional reason for UI, logs, Behavior Trees, or StateTree decisions.

Policy assets are shared. They add no UObject to each Mass entity.

Composition and transaction contract

An ability may reference several policies. Mass Forge evaluates them in authored array order and stops at the first rejection. All assigned policies must allow the pair.

Targeting policies run after the source grant and basic target-handle checks, but before tag gates, cooldowns, charges, attribute requirements, costs, or effects. A rejection therefore consumes nothing and publishes no successful ability/effect event.

The activation result returns:

- `Targeting Policy Rejected` as the ability result.
- `Failed Targeting Policy Index` for the authored array entry.
- The policy's stable ID and optional project reason.
- A machine-readable policy result such as `Below Minimum Range`, `Beyond Maximum Range`, or `Missing Target Position`.
- `Evaluated Distance` when the range policy reached its distance calculation.

An unassigned policy reference or a loaded policy without an ID makes the ability invalid. Editor validation reports unassigned references, missing policy IDs, non-finite ranges, negative ranges, and inverted minimum/maximum bounds.

Built-in range policy

The range policy resolves positions using one of these modes:

- **Mass Transform, Then Represented Actor** - prefer the entity's standard `FTransformFragment`; fall back to its represented Actor.
- **Mass Transform Only** - require standard Mass transform fragments on both entities.
- **Represented Actor Only** - require both entities to have represented Actors.

The default mode works for actorless entities and stays stable across representation changes when Mass transforms are present. Actor-only mode is useful when the project's authoritative interaction position intentionally comes from Actor representation.

Both range bounds are inclusive. Set `MinimumRange` to zero when only a maximum matters. XY-only distance ignores vertical separation; three-dimensional distance includes it.

Blueprint safety

Custom policy overrides must be deterministic and side-effect free. Do not grant abilities, apply effects, change attributes, or activate another ability from `Evaluate Target`. Mass Forge rejects recursive policy evaluation with `Targeting Policy Evaluation In Progress` instead of allowing unbounded recursion.

Direct checks and activations still return `Mass Is Processing` while Mass owns the entity manager. A queued ability reevaluates every targeting policy when its ability phase executes, so movement or representation changes between submission and execution cannot bypass the current rules. See the deferred command contract.

Scope and current limits

Target validation and target selection are deliberately separate. Use the built-in radius, trace-segment, represented-Actor, collision-hit, or custom-provider adapters described in the target selection guide. Multiplayer is outside the initial supported release; a networked project must revalidate untrusted target requests on its own authority path.

For attribute and Gameplay Tag gates that are intrinsic to the ability, use the existing ability requirements rather than recreating them in a custom policy. See the abilities guide.

GAMEPLAY SYSTEMS

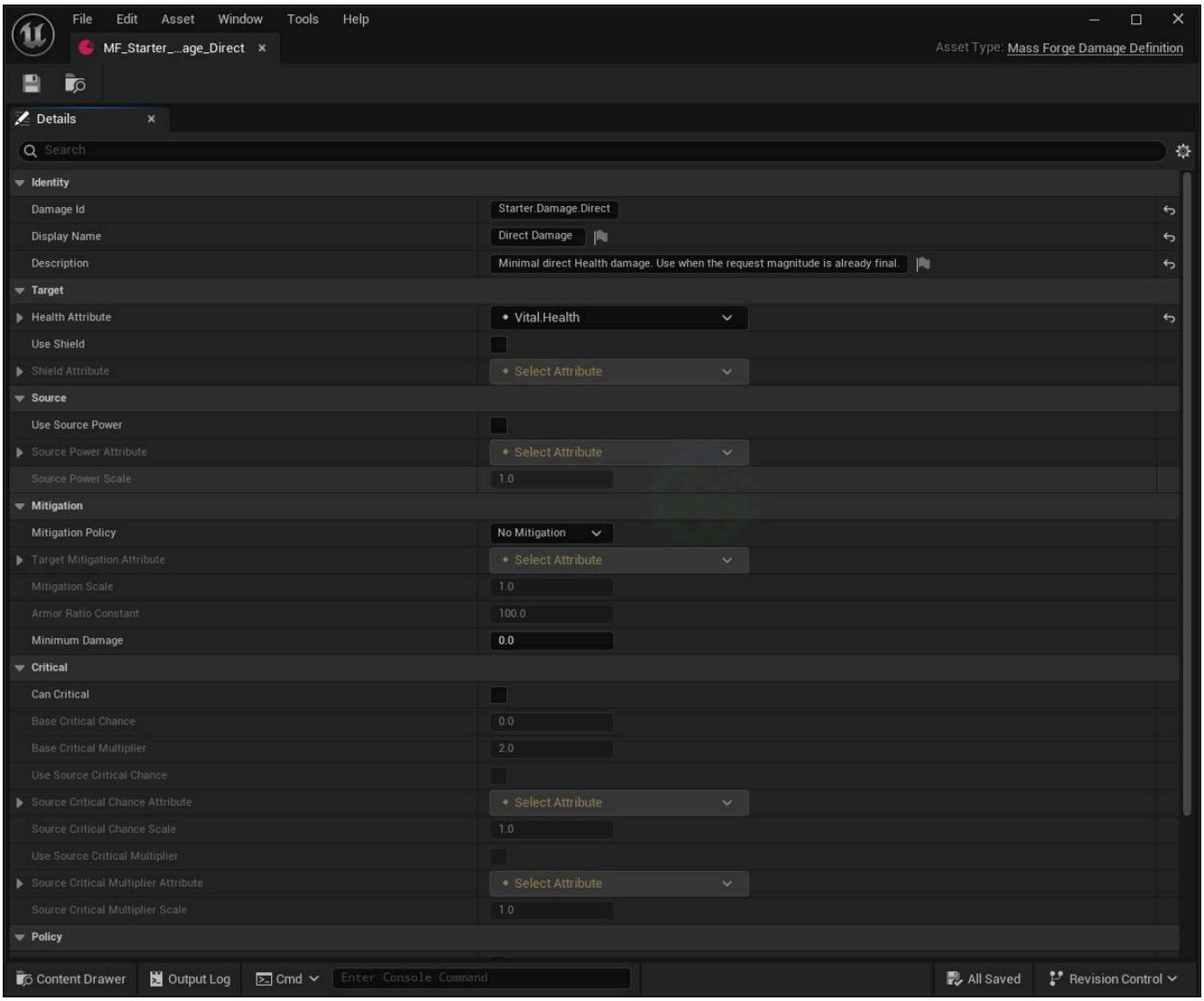
Damage system guide

Mass Forge Damage Definitions turn project-owned attributes into a reusable, deterministic combat calculation. They are intended for damage against actorless or represented Mass entities. Attribute names are never reserved: `Health`, `Shield`, `Power`, and `Armor` below are examples selected in the asset.

Ordinary Actor targets such as a non-Mass player continue to use the generic Effect Receiver adapter described in the combat integration guide. This keeps Mass Forge compatible with a project's existing GAS, health component, or other Actor-side state instead of creating a competing store.

One-time setup

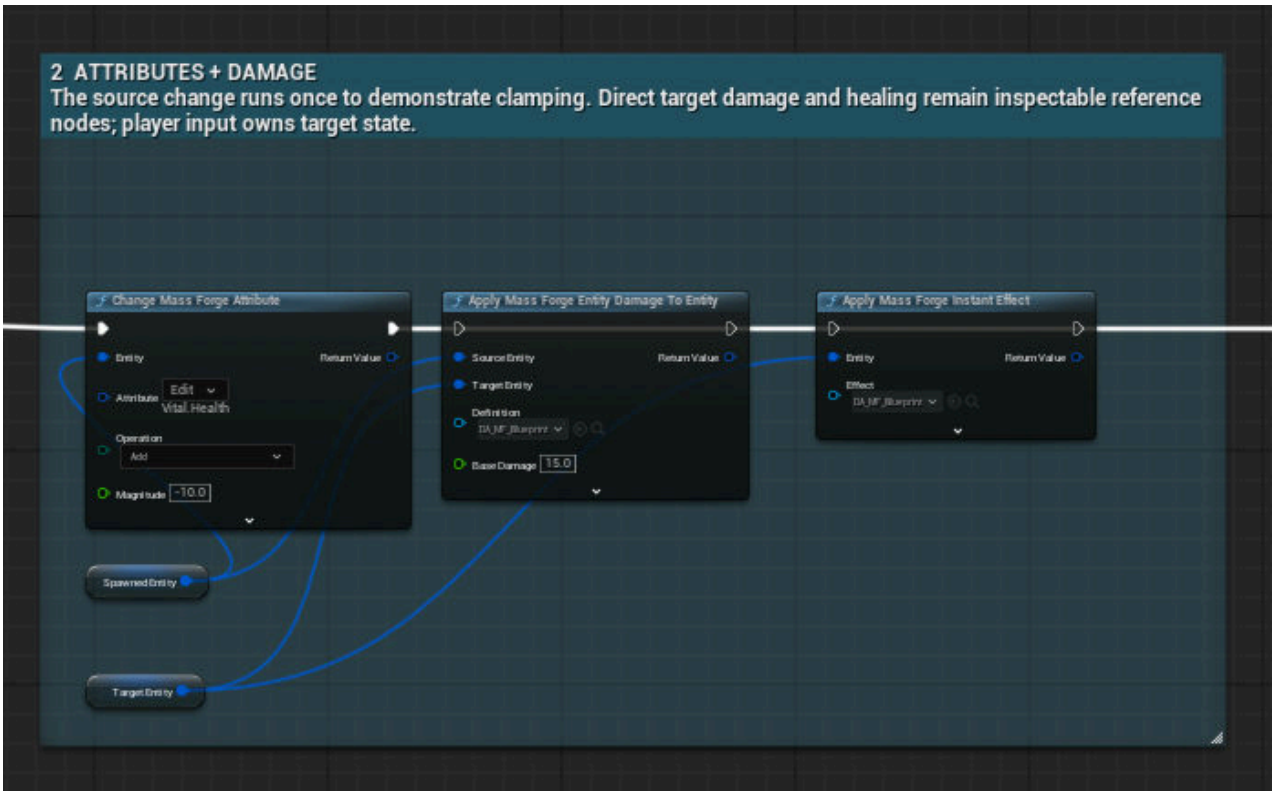
1. Add current Health to a Vital, Combat, or Support schema and give it a non-negative lower bound if zero means dead.
2. Optionally add Shield, source Power, target mitigation, Critical Chance, and Critical Multiplier attributes.
3. Assign the schemas under **Project Settings > Mass Forge > Global Attribute Schemas**.
4. Add **Mass Forge Attributes Trait** to every participating Mass Entity Config.
5. Add **Mass Forge Gameplay Tag Trait** when the definition uses required, blocked, or death tags.
6. Create a **Mass Forge Damage Definition** Data Asset and select those attributes.



This minimal authored definition turns the request magnitude into final Health damage. Enable shield routing, source power, mitigation, or critical rules only when the game needs those stages.

The Damage Definition validates its ID, attribute references, finite scales, critical bounds, positive armor constant, distinct Health/Shield fields, and contradictory required/blocked tags. The selected attributes still resolve against the configured schemas at runtime, so a renamed or missing project attribute returns a structured failure instead of silently changing another slot.

Blueprint entry points



The middle node is the common entity-to-entity damage path. The surrounding nodes show why raw stat changes, damage formulas, and reusable healing/effects remain separate public operations.

- **Apply Mass Forge Damage** accepts a complete request and is the most flexible node.
- **Apply Mass Forge Entity Damage To Entity** fills source and target entity handles for common Mass combat.
- **Apply Mass Forge Actor Damage To Entity** accepts an ordinary or represented source Actor. If the Actor represents a Mass entity, its handle is resolved automatically. Source-attribute formulas require that representation; an ordinary Actor can still deal definition-based damage when source attributes are disabled.
- **Queue Mass Forge Damage** safely defers a complete request when the call may overlap Mass processing.

The three Apply nodes are immediate. If Mass is processing, direct application returns **Mass Is Processing** before invoking policy providers or reading entity state. Queue the request in callbacks that may overlap Mass work.

An accepted queued request returns a positive monotonic `Request Id`. Requests execute FIFO while Mass is idle, up to **Max Queued Damage Requests Per Frame** under **Project Settings > Mass Forge > Global Attribute Schemas > Runtime**. **Max Pending Queued Damage Requests** supplies hard back-pressure; overflow returns **Damage Queue Full** and consumes no ID. Each request is revalidated at execution, so a stale entity or changed/unloaded definition completes with a structured failure rather than touching state. Bind **On Damage Request Completed** on the Damage Subsystem and match its ID with the enqueue receipt. Successful damage/death events occur during application, followed by the request-completion event. Cross-type ordering follows the deferred command contract.

Calculation order

One request is evaluated in this order:

1. Validate the world, definition, generation-checked entities, numeric inputs, and self-damage rule.
2. Evaluate required target tags, blocked target tags, and optional dead-target rejection.

3. Read the configured Health, optional Shield, source attributes, and mitigation attribute.
4. Calculate `Base Damage + Source Power x Source Power Scale`, clamped to at least zero.
5. Resolve critical chance and multiplier. A critical occurs only when the caller's `Critical Roll` is lower than the resolved chance.
6. Apply the selected mitigation policy and caller-provided penetration.
7. Apply Minimum Damage when pre-mitigation damage was positive.
8. Send the requested bypass fraction directly toward Health; Shield absorbs the remaining shieldable amount up to its current value.
9. Clamp projected Health and Shield through their schema entries.
10. Atomically commit every changed attribute and the optional first-death tag.
11. If this is a first-death transition, schedule the selected Death Handling Policy, then publish damage, kill, and optional project-deactivation events.

The caller supplies `Critical Roll` in `[0, 1]`. This makes the calculation deterministic and replayable; the project can use its own seeded random stream, authoritative server roll, or fixed test value. Mass Forge does not hide an uncontrolled random-number source inside the subsystem.

Mitigation policies

Policy	Formula after critical	Meaning of penetration
No Mitigation	`Damage`	Ignored
Flat Reduction	`max(0, Damage - max(0, Mitigation x Scale - Penetration))`	Flat mitigation points ignored
Percentage Reduction	`Damage x (1 - clamp(Mitigation x Scale - Penetration, 0, 1))`	Fractional reduction ignored
Armor Ratio	`Damage x K / (K + max(0, Armor x Scale - Penetration))`	Flat armor points ignored

`K` is **Armor Ratio Constant**. At effective Armor equal to `K`, damage is halved. Minimum Damage is evaluated after the selected mitigation formula.

Tags, immunity, and death

Required Target Tags are useful for eligibility such as `State.Alive` or `Faction.Damageable`. Blocked Target Tags are useful for invulnerability, damage-type immunity, protected phases, or project-specific faction policy. Queries are hierarchical, so a parent blocker can reject owned child tags.

When **Reject Dead Targets** is enabled, zero Health or an already-owned configured Death Tag returns **Target Already Dead**. When positive Health reaches zero, `bKilled` is true, the optional Death Tag is added in the same transaction, and **On Entity Killed** is broadcast once for that transition.

Each Damage Definition selects one explicit post-kill policy:

Death Handling Policy	Behavior
Keep Entity	Default. Publish the normal damage/kill events and leave lifecycle state unchanged. This supports corpses, resurrection, and project-owned state machines.

Death Handling Policy	Behavior
Request Project Deactivation	Assign a positive correlation ID and publish **On Entity Deactivation Requested** after **On Entity Killed**. The entity remains valid; the project decides how to disable AI/movement, pool representation, reward, ragdoll, respawn, or remove it.
Destroy Mass Entity (Deferred)	Queue generation-safe Mass-entity destruction. Kill listeners and queued-damage completion run first; the bounded Death Handling phase destroys the entity before the Ability phase and publishes **On Death Destruction Completed**.

The killing result exposes the policy, scheduling result, and request ID. **Queue Full** applies only to the optional destruction request: lethal damage and its atomic Health/death-tag transaction still succeed, the request ID remains zero, and the still-valid dead entity is available for recovery. Configure **Max Death Destructions Per Frame** and **Max Pending Death Destructions** under the Runtime settings.

Deferred destruction removes only the Mass entity. It never destroys or hides a represented Actor and never invents pooling, rewards, ragdoll, respawn, or replication behavior. Use **Request Project Deactivation** when any of those project-specific actions are required. A direct lethal call queues destruction for the next coordinator tick; a lethal request executed in the Damage phase is destroyed later in that same tick. If another system removes the entity first, completion reports **Entity Already Invalid** without touching a reused generation.

For exact same-tick precedence between Persistent Effects, Damage events, request completion, destruction, active Ability lifecycles, and new Ability requests, see the deterministic gameplay ordering contract.

For the complete destruction, generation reuse, Actor representation, pooling, and world-teardown rules, see the entity lifecycle safety contract.

Friendly fire, factions, ownership, and safe zones

These rules depend on project data, so Mass Forge exposes a policy contract instead of inventing a team model. The easiest Blueprint setup is:

1. Create a Blueprint subclass of **Mass Forge Damage Policy Component**.
2. Add one instance to a persistent world manager Actor or Game State. **Auto Register** is enabled by default.
3. Override **Evaluate Mass Forge Damage Policy**.
4. Read `Source Entity`, `Target Entity`, resolved source/target Actors, `Damage Id`, and the context payload from the query.
5. Query the project's team, faction, owner, safe-zone, or diplomacy data.
6. Return **Deny** with a stable `Policy Id` and machine-readable `Reason` when damage is forbidden. Return **Abstain** when this provider has no opinion, or **Allow** to record an affirmative decision while still letting other providers veto.

A simple friendly-fire Blueprint compares the project team IDs for the resolved source and target. If both are valid and equal, return `Deny`, `Policy Id = TeamPolicy`, and `Reason = FriendlyFire`; otherwise return `Abstain`.

Projects can also implement **Mass Forge Damage Policy Provider** on another UObject and use **Register Mass Forge Damage Policy Provider** and **Unregister Mass Forge Damage Policy Provider** manually. Registration is weak, world-scoped, duplicate-aware, and bounded to sixteen providers. Invalid objects, cross-world objects, duplicate registration, overflow, missing removal, and evaluation-time mutation all return distinct result values.

Providers run in registration order after endpoints and numeric inputs are validated but before tags or attributes are read. Any **Deny** stops the request immediately; no Health/Shield/tag mutation or damage/death event occurs. An **Allow** never bypasses a later provider or the Damage Definition's built-in tag and dead-target rules. Provider

evaluation is synchronous and must be side-effect free. Recursive damage and provider-list mutation attempted from inside evaluation are rejected rather than allowing an unstable policy chain.

Reading the result

Mass Forge Damage Application Result contains:

- the exact result enum and failed attribute or tag diagnostics;
- requested damage, source-power contribution, pre-mitigation damage, mitigation value, and final damage;
- resolved critical chance/multiplier and whether the supplied roll was critical;
- Shield/Health before, applied damage, after values, and overkill;
- whether this application caused the first positive-to-zero transition;
- selected death policy, its scheduling result, and the positive request ID when one was produced;
- the fully resolved effect context and underlying atomic Instant Effect transaction.

Successful zero-damage requests are valid and publish a damage-applied result without inventing an attribute change. Failed requests publish neither damage nor death events.

Context and presentation

The applied context can carry source/target entity handles, represented Actors, a logical Source ID, Ability ID, Damage Definition ID as Effect ID, origin, optional `FHitResult`, and project scalar parameters. Bind **On Damage Applied** for floating numbers, hit reactions, threat, audio, analytics, or combat logs. Bind **On Entity Killed** for common death presentation, **On Entity Deactivation Requested** for project-owned retirement, and **On Death Destruction Completed** for deferred-destruction bookkeeping. These listeners are not required by the simulation and should not mutate the same target recursively.

Safety boundaries

- Health, Shield, and a first-death tag commit through one prevalidated transaction. A late tag-capacity or missing-fragment failure leaves attributes unchanged.
- Non-finite damage, penetration, roll, bypass, asset scales, or calculated results are rejected.
- Entity handles include serial numbers; stale source and target handles fail explicitly.
- Source and target Actors must belong to the same world.
- Direct calls are rejected while Mass is processing; queued calls wait for a safe tick and apply bounded work per frame.
- Queued definitions are expected to be immutable during play. Mass Forge stores a soft reference and original Damage ID, rejecting an unloaded/replaced asset or identity change at execution.
- This release supplies the policy-provider contract for project-defined friendly fire, teams, ownership, and protected zones, but deliberately does not define the project's relationships. Multiplayer damage can use Mass Forge's owner-bound authority transport and replicated state, but project policies must still define who may damage whom. Presentation prediction is available only for Ability activation; damage prediction, resistance by damage type, and represented-Actor lifecycle remain project integration work.

Automated proof

`MassForge.Damage.DeterministicPolicyAndAtomicCommit` runs in an isolated world with real Mass entities and configured schemas. It verifies formula order, deterministic criticals, armor ratio, Shield absorption and bypass, required and blocked tags, self-damage rejection, source requirements, minimum damage, first-death behavior, context identity/hit payloads, non-finite rejection, generic healing, full rollback when a lethal death-tag mutation cannot be committed, ordinary Actor sources, policy-provider behavior, queued FIFO budgeting/back-pressure, project-deactivation events, non-immediate Mass destruction, destruction request IDs, per-frame and pending limits, queue-full recovery, stale destruction, and completion results.

GAMEPLAY SYSTEMS

Combat and cross-ownership effect guide

Mass Forge provides two complementary layers: project-authored **Instant Effects** for direct state changes, and reusable **Damage Definitions** for common deterministic combat formulas against Mass entities. Neither layer hard-codes an attribute name, player class, or despawn implementation.

This guide covers the four Blueprint dispatch nodes that connect Mass entities and ordinary Actors without project-specific casts.

Before dispatching

For a Mass target:

1. Define the target attribute in a Mass Forge schema. `Vital.Health` is a common example, not a required name.
2. Add **Mass Forge Attributes Trait** to the target Entity Config.
3. Create a **Mass Forge Instant Effect** and add the desired attribute or Gameplay Tag modifiers.

For an ordinary Actor target such as a player:

1. Add **Mass Forge Effect Receiver Component** to the Actor or implement **Mass Forge Effect Receiver** directly on the Actor.
2. Bind the component's **On Effect Received** event.
3. Translate the generic effect and context into the Actor's project-owned health, GAS, inventory, or other system.

The stock receiver component is an adapter, not a second attribute store. Its **Accept Effects** switch can reject incoming effects temporarily. A direct Actor interface implementation is useful when acceptance needs custom synchronous logic.

Choosing the node

Source	Target	Blueprint node
Mass entity	Mass entity	**Apply Mass Forge Entity-to-Entity Effect**
Actor/player	Mass entity	**Apply Mass Forge Actor-to-Entity Effect**
Mass entity	Actor/player	**Apply Mass Forge Entity-to-Actor Effect**
Actor/player	Actor/player	**Apply Mass Forge Actor-to-Actor Effect**

The entity nodes work for actorless and represented entities. The Actor nodes accept ordinary Actors and represented Mass Actors. A represented Actor target is automatically routed to its Mass entity before any Actor receiver is considered.

Each node accepts an optional **Context** advanced pin. Set `Source Id` to a weapon, hazard, ability, or other stable logical identity. Add `Origin`, optional hit data, and custom scalar `Parameters` when needed. Mass Forge resolves source and target identities and stamps the selected effect identity while preserving the authored payload.

Dispatch precedence

An Actor target is resolved in this order:

1. represented Mass entity;
2. receiver interface implemented directly by the Actor;
3. exactly one Actor Component implementing the receiver interface.

This order prevents a represented entity from receiving the same effect twice. If an Actor has multiple receiver components, dispatch returns **Multiple Receiver Components** rather than guessing or double-applying. An Actor with no supported route returns **Unsupported Target**.

Reading the receipt

Every cross-ownership node returns a **Mass Forge Effect Dispatch Receipt**:

- `Result` reports routing success or the exact routing failure;
- `Route` identifies Mass Entity, Actor Interface, or Component Interface;
- `Resolved Source Entity` is populated for an entity source or represented Actor source;
- `Resolved Entity` is the resolved target entity for the Mass route;
- `Applied Context` is the fully resolved context delivered to the selected route;
- `Entity Application` contains every attribute/tag result for a Mass target;
- `Receiver Object` identifies the Actor or Actor Component used by a receiver route.

`Success` means the selected route accepted and completed the immediate dispatch. `Entity Application Failed` means routing succeeded but the Mass effect transaction failed; inspect `Entity Application.Result` and its detailed fields. `Receiver Rejected` means the ordinary Actor adapter deliberately declined the effect.

Blueprint recipes

Actorless Mass entity damages another actorless entity

Use **Apply Mass Forge Entity-to-Entity Effect** with both generation-checked handles and a damage Instant Effect. The source handle is copied into `Applied Context.Source Entity`; no Actor is required.

Represented Mass entity damages another represented entity

Use the same entity-to-entity node. Mass Forge resolves the optional source and target Actors into the applied context, while the Mass handles remain authoritative.

Player damages an actorless entity

Use **Apply Mass Forge Actor-to-Entity Effect** with the player as Source Actor and the Mass handle as Target Entity. If the player also represents a Mass entity, the receipt exposes that resolved source handle automatically.

Player damages a represented Mass entity

Use **Apply Mass Forge Actor-to-Actor Effect** with the player and represented target Actor. The target's Mass handle is resolved and the effect runs through the atomic Mass path.

Actorless Mass entity damages a player

Add **Mass Forge Effect Receiver Component** to the player, bind **On Effect Received**, and call **Apply Mass Forge Entity-to-Actor Effect**. The event receives the actorless source handle, effect asset, source ID, origin, and parameters.

Represented Mass entity damages a player

Use the same entity-to-Actor node. The applied context contains both the source entity and its represented Actor when representation is currently available.

Add project-owned factions and friendly-fire rules

Mass Forge does not prescribe `Team 0`, `Team 1`, or a fixed faction hierarchy. Put your own counted Gameplay Tags on the Entity Configs, such as `Faction.Player`, `Faction.Guard`, or `Faction.Monster`, and choose one of these policy paths:

1. For the common same-faction rule, add **Mass Forge Gameplay Tag Damage Policy Component** to one world manager or Game State.
2. Populate **Faction Tags** with the exact tags your project owns.
3. Leave **Auto Register** enabled. When two Mass endpoints share one configured tag, the component returns **Deny** with policy ID `GameplayTagFriendlyFire` and reason `SharedFaction`.
4. Actor-only sources and entities without a configured faction make the component **Abstain**, allowing later project policies to decide.

For alliances, ownership, safe zones, PvP flags, or asymmetric rules, subclass **Mass Forge Damage Policy Component** in Blueprint and override **Evaluate Mass Forge Damage Policy**. Read the resolved source/target handles and return **Deny**, **Allow**, or **Abstain**. A denial is visible in `Mass Forge Damage Application Result` through **Blocking Policy Id**, **Blocking Policy Reason**, and **Evaluated Policy Count**.

The shipped Battle Simulator makes the reusable path inspectable: its arena root carries `FactionDamagePolicy`, configured with the sample Mint and Berry tags. Replace those sample tags with project-owned tags rather than adopting the sample taxonomy.

Safety and ownership boundaries

- Entity handles are validated by index and serial number. Stale sources return **Invalid Source**; stale targets return **Invalid Target**.
- Actors must belong to the same world as the call. Cross-world endpoints are rejected.
- The four convenience nodes are immediate operations. If a Mass entity call may overlap Mass processing, use the existing queued entity Instant Effect node and provide the same context; completion is reported by request ID.
- Receiver events are synchronous and run on the caller's game-thread path. Keep them bounded and avoid recursively dispatching the same effect.

- Damage Definitions provide opt-in mitigation, critical, shield, tag-gate, self-damage, minimum-damage, and first-death policies for Mass targets. Post-death handling can keep the entity, request project-managed deactivation, or queue Mass-entity destruction. Mass Forge supplies the policy extension point and a configurable tag-based friendly-fire component; the actual faction tags, alliances, ownership, represented-Actor lifecycle, and presentation remain project integrations.

Verification

`MassForge.Effects.CrossOwnershipDispatch` creates real Mass entities, real representation mappings, ordinary Actors, and the receiver component in an isolated world. It verifies all six recipes above plus disabled receivers, duplicate receiver components, invalid sources, invalid targets, resolved routes, and applied source/target context. `MassForge.Damage.DeterministicPolicyAndAtomicCommit` verifies configurable shared-faction denial, opposing-faction damage, stable policy diagnostics, registration bounds, and atomic health behavior.

For effect authoring, atomicity, parameters, and attribute-backed magnitudes, continue with the Instant Effects guide.

For the configurable damage formula and Blueprint nodes, continue with the Damage system guide.

Instant effects

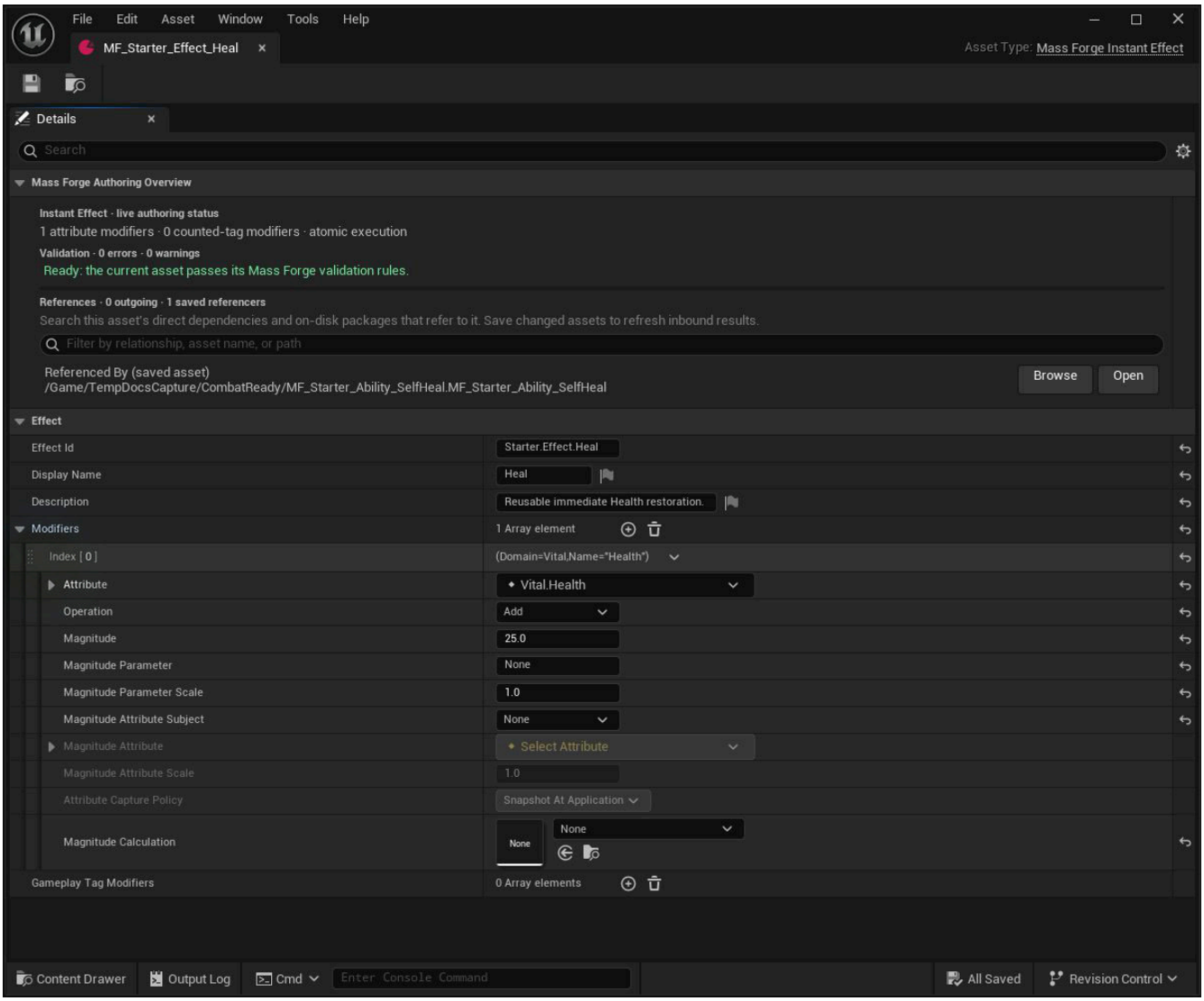
For Entity-to-Entity, Actor-to-Entity, Entity-to-Actor, and Actor-to-Actor routing recipes, see the combat and cross-ownership effect guide.

Mass Forge Instant Effect Data Assets are reusable, game-agnostic transactions for immediate attribute and counted Gameplay Tag changes. They work on actorless Mass entities, represented entities, queued callers, and ability self/target effect lists.

Opening the asset shows live validation plus searchable saved referencers, making it easy to audit every Ability or Persistent Effect affected by a change. See GAMEPLAY_ASSET_AUTHORING_GUIDE.md.

Create an effect

1. Create a **Mass Forge Instant Effect** Data Asset.
2. Assign a stable `EffectId`; this becomes its Primary Asset identity and the fallback mutation source.
3. Add one or more attribute modifiers, Gameplay Tag modifiers, or both.
4. Apply it directly while Mass is idle, queue it when work can overlap Mass processing, or reference it from a Mass Forge Ability.



This reusable effect adds 25 to `Vital.Health`. The schema clamps the result, and the inbound reference shows that the generated Self Heal ability consumes this asset.

An asset with neither kind of modifier is invalid. Editor validation also rejects unnamed attributes, non-finite magnitudes or scales, invalid tags, and tag counts outside 1-255.

Attribute modifiers

Each entry selects an attribute domain and stable name, then applies one operation in authored order:

- **Set** replaces the working value.
- **Add** offsets the working value. Negative values are suitable for damage or resource spending.
- **Multiply** scales the working value.

The target schema clamps every intermediate result. Multiple entries may address the same attribute; each later entry observes the prior entry's projected value.

An optional magnitude parameter resolves as:

```
Magnitude + Context Parameter x Magnitude Parameter Scale
```

Context-aware calls supply parameters at runtime. Ability assets provide defaults through `EffectParameters`. A missing or non-finite parameter rejects the whole transaction and returns its name.

An optional source- or target-attribute term extends that formula:

`Magnitude + Parameter Term + Sampled Attribute x Magnitude Attribute Scale`

Source reads `Context.SourceEntity`; **Target** reads the effect recipient. For an immediate Instant Effect, both capture policies sample once at the start of that atomic execution. When the Instant Effect is used as a Persistent Effect pulse, **Snapshot At Application** freezes the value when the persistent instance is created, while **Evaluate On Execution** reads the current value before every pulse. All operations in one atomic instant/ability transaction sample the pre-commit attribute state rather than observing earlier projected writes.

Snapshot contexts are bounded to 32 unique subject/attribute pairs. Invalid entities, missing schemas/fragments, unknown attributes, non-finite values, and missing source context fail explicitly and identify the capture subject and attribute.

Gameplay Tag modifiers

Each entry selects an exact project Gameplay Tag, **Add Count** or **Remove Count**, and a count from 1 to 255. Entries run in authored order against projected tag state.

Adding a new tag needs one of the entity fragment's 32 slots. Adding an existing tag increments its count up to 255. Removing requires the complete requested count; removing the final count releases the slot. The effect result distinguishes invalid tags/counts, missing Gameplay Tag Trait, absent tags, insufficient counts, overflow, and full capacity.

The plugin does not define game-specific tags. Define them in the project's normal Gameplay Tags dictionary and add **Mass Forge Gameplay Tag Trait** to every entity that may receive tag modifiers.

Atomic transaction contract

Mass Forge validates and calculates every attribute and tag operation before changing fragment state. If any operation fails:

- no attribute value changes;
- no tag count changes;
- the returned `Changes` and `GameplayTagChanges` arrays are empty;
- no attribute, tag, or successful-effect event is published.

On success, all fragment state is committed before listeners run. `Changes` and `GameplayTagChanges` preserve authored operation order and report previous/new values or counts. Attribute changes publish `OnAttributeChanged`; tag changes publish `OnGameplayTagChanged`; the complete transaction then publishes `OnInstantEffectApplied`.

Ability costs, self effects, and target effects extend the same atomic boundary. A tag failure returns `Gameplay Tag Effect Failed`, the failed tag, its source/target subject, and the detailed tag result. The ability system restores any reserved individual cooldown, shared cooldown, and charge state.

This scope covers one Instant Effect recipient. Separate effect calls-even consecutive queued calls-are independent transactions. For the source-plus-one-target ability boundary, multi-target partial-success rule, ordinary Actor limitation, and re-entrant listener guarantees, see the transaction and event contract.

Blueprint entry points

- `Apply Mass Forge Instant Effect` applies immediately with a simple source name.
- `Apply Mass Forge Instant Effect With Context` carries source/target entities and Actors, source identity, origin, and scalar parameters.
- `Queue Mass Forge Instant Effect` accepts work for the next safe world tick and returns a request ID; observe `OnInstantEffectRequestCompleted`.
- Queued Instant Effects run after queued raw attribute changes and before queued damage, death handling, and abilities; see the deferred command contract.
- `Dispatch Mass Forge Instant Effect To Actor` applies to a represented Mass entity or calls the generic **Mass Forge Effect Receiver** interface on an ordinary Actor.

Direct entity application returns `Gameplay Tag Failure` for a tag-operation rejection. Read `GameplayTagResult` and `FailedGameplayTag` for the exact cause. Successful tag details are in `GameplayTagChanges`.

Example recipes

- **Damage plus state:** add `-25` to Vital `Health` and add one project-defined `State.Hit` tag.
- **Healing and cleanse:** add `+40` to Vital `Health` and remove one `State.Debuff.Poisoned` count. If the target is not poisoned, the heal also rolls back; use two effects when independent success is desired.
- **Ability mark:** create a tag-only effect that adds `State.Target.Marked`, then place it in an ability's target effects.
- **Resource conversion:** subtract from one attribute and add to another in the same effect, ensuring neither side can commit alone.

Current scope

Instant effects are immediate transactions. Wrap one in a **Mass Forge Persistent Effect** for finite or infinite periodic execution, application-time or live source/target capture, lifecycle-granted tags, immunity gates, classification cancellation, persistent attribute aggregation, and configurable independent, capped, refresh, extend, replace, or strongest-wins behavior; see PERSISTENT_EFFECTS_GUIDE.md. Multiplayer mutations use the owner-bound authority request layer and authoritative state stream. Prediction remains presentation-only for Ability activation; direct Instant Effects are never silently predicted.

GAMEPLAY SYSTEMS

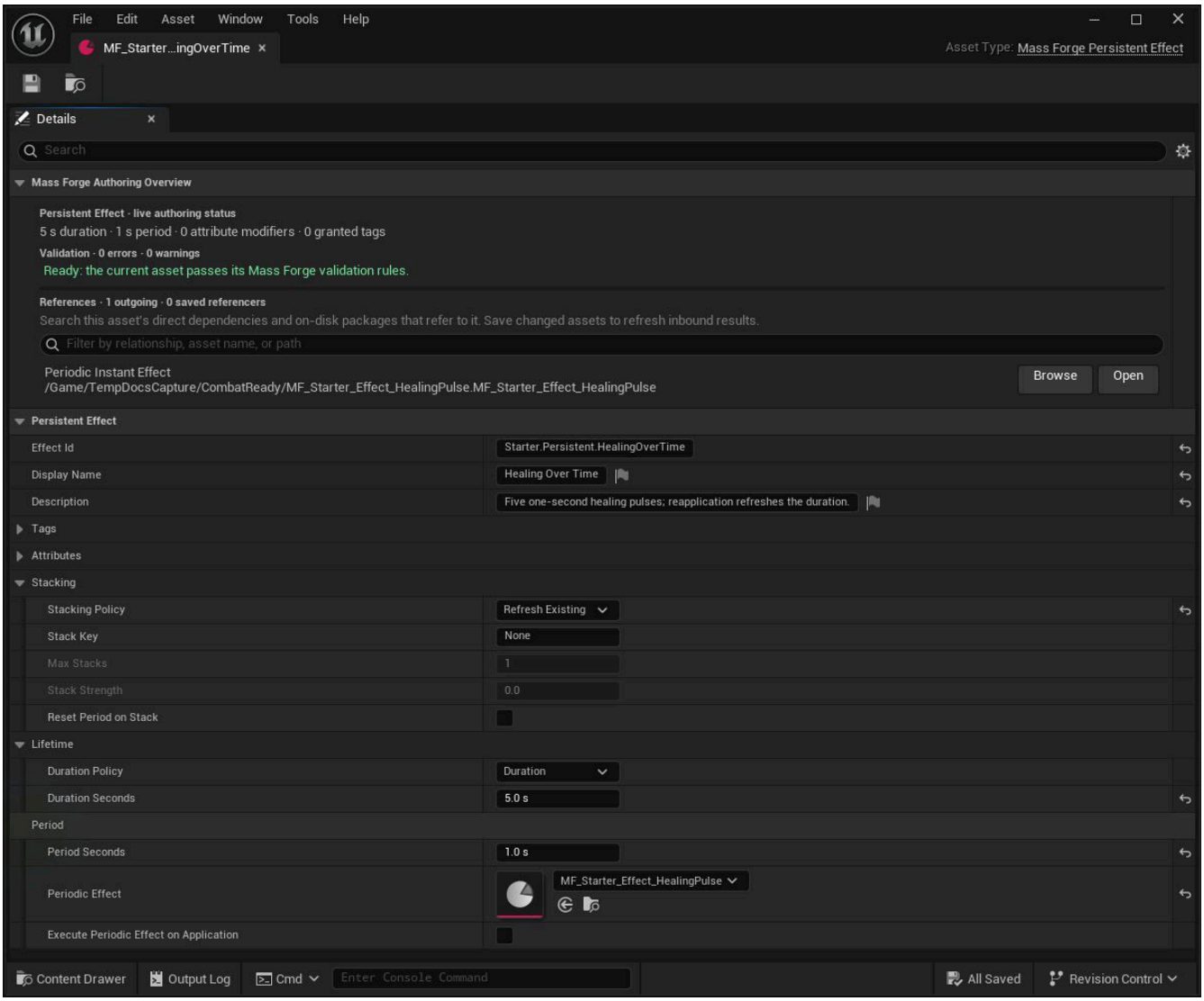
Persistent effects

Mass Forge Persistent Effects provide opt-in finite or infinite lifecycle instances for actorless and represented Mass entities. An instance can act as a queryable timer/marker, execute an Instant Effect periodically, or do both.

Opening the asset shows live validation, its periodic dependency, and searchable saved referencers with direct navigation. See GAMEPLAY_ASSET_AUTHORING_GUIDE.md.

Setup

1. Add **Mass Forge Persistent Effects Trait** to every Mass Entity Config that may receive persistent effects. Also add **Mass Forge Gameplay Tag Trait** when using requirements, immunity blockers, or granted tags.
2. For periodic behavior, create a **Mass Forge Instant Effect** containing the attribute and/or counted Gameplay Tag pulse.
3. Create a **Mass Forge Persistent Effect** Data Asset and assign a stable `EffectId`.
4. Choose **Duration** or **Infinite**, select a stacking policy, then optionally configure persistent attribute modifiers, a period, and a pulse effect.
5. Call `Apply Mass Forge Persistent Effect` at a safe point, or `Queue Mass Forge Persistent Effect` when the caller may overlap Mass processing.



This complete asset is the reusable authoring unit: reapplication refreshes the existing handle, the lifecycle lasts five seconds, and the referenced Instant Effect executes once per second. The green authoring overview proves the saved asset and dependency validate together.

Each participating entity has 16 fixed active-instance slots. The trait is opt-in, so entities that do not need lifecycle effects pay no fragment cost.

Lifetime policies

- **Duration** requires a positive finite `DurationSeconds` value. The instance expires automatically after its active time and publishes an expiration event.
- **Infinite** has no automatic expiration. Remove it explicitly by its returned handle or let entity destruction invalidate it.

An effect with a zero period is a lightweight lifecycle marker. It remains queryable, can own lifecycle tags, and emits apply/removal events without periodically changing attributes.

Persistent attribute modifiers

Add entries to `AttributeModifiers` when an effect should contribute a reversible stat change for its complete lifetime. Each contribution targets a normal schema attribute and uses one aggregation channel:

- **Add** contributions are summed.
- **Multiply** contributions are multiplied together.
- **Override** selects one winner. Higher `OverridePriority` wins; equal priority is resolved by the newest active handle, making the result stable and explainable.

The effective value is calculated as `(Base + Additive Sum) * Multiplicative Product`, followed by the winning override when one exists, then the attribute schema clamp. Removing, replacing, cancelling, refreshing, or expiring an instance rebuilds the remaining aggregate from snapshotted per-handle contributions; Mass Forge never tries to reverse a modifier with lossy inverse arithmetic.

Modifier magnitudes can use the same context-parameter formula as Instant Effects and can add one sampled source or target attribute term. Persistent lifetime contributions always resolve that term when the instance applies, preserving a compact allocation-free aggregate. Later source/target changes, parameter changes, or Data Asset edits cannot mutate an existing contribution. Missing and non-finite parameters, invalid capture entities, unknown attributes, missing schema/attribute fragments, aggregate overflow, and capacity exhaustion return explicit apply results before lifecycle state changes.

The Persistent Effects Trait includes storage for 24 distinct modified attributes per entity in addition to the 16 lifecycle slots. Multiple active contributions to the same attribute consume one aggregate slot. This fixed bound keeps the hot path allocation-free; exceeding it returns `Modifier Capacity Exceeded`.

The final PrePhysics modifier processor reconciles direct Mass writes after regeneration and reapplies the active aggregate. The derived-attribute processor writes its calculated result into the aggregate's base, so persistent modifiers remain effective on derived targets and dependent derived rules observe the effective modified value. Blueprint/subsystem changes continue to operate on the visible effective value; an active override keeps presentation fixed while preserving underlying base movement for when the override leaves.

Periodic execution

Set `PeriodSeconds` to at least 0.001 seconds and assign a valid **Mass Forge Instant Effect** to `PeriodicEffect`. Every pulse uses the Instant Effect's complete atomic transaction contract, including attribute bounds, counted tag operations, structured failures, and change events.

Enable `Execute Periodic Effect On Application` to run pulse 1 during application. If that first pulse fails, the persistent application returns `Initial Application Failed`, exposes the complete Instant Effect result, and leaves no active instance. A successful first pulse is followed by the next pulse one full period later.

Scheduled pulse failures are reported through `On Persistent Effect Period`; they do not silently remove the lifecycle instance. This lets a project decide whether a temporary failure should cancel, pause, or simply skip that pulse.

Deterministic timing and catch-up

Finite effects count only time within their authored lifetime. If a frame crosses several period boundaries, every due pulse is scheduled in order. A pulse exactly on the duration boundary is included before expiration.

To prevent one hitch from monopolizing the game thread, one instance executes at most 64 catch-up pulses per subsystem tick. Remaining pulses stay in the compact slot as backlog and execute on later ticks; they are not discarded. A finite instance whose lifetime has ended remains internally pending until its earned backlog is drained, then emits one expiration event.

If a scheduler tick encounters active Mass processing, elapsed time is accumulated and applied on the next safe tick rather than lost.

The scheduler uses a world-local deadline min-heap. It advances only effects whose next pulse, expiration, or catch-up deadline is due, and it never queries or iterates the complete Mass entity population. A separate generation-safe ownership check scales with active persistent handles so destroyed owners are still retired on the next phase. Refresh and extension use revisioned lazy rescheduling; obsolete heap entries are harmless and compacted when their ratio grows.

Use **Get Persistent Effect Scheduler Stats** on the Persistent Effect subsystem to inspect active handles, schedules, heap entries, last-phase due work, ownership checks, and stale entries from Blueprint. See the persistent scheduler architecture for the exact runtime, determinism, compaction, diagnostics, and verification contract.

Stacking policies

`StackKey` controls which active instances interact. If it is empty, Mass Forge uses `EffectId`, so the default setup groups applications of the same asset. An explicit key lets different effect assets participate in one shared stack.

- **Independent** always creates a new handle, subject to the entity's 16-slot capacity.
- **Capped Independent** creates separate handles until `MaxStacks` is reached, then returns `Stack Limit Reached` without executing an initial pulse or changing state.
- **Refresh Existing** reuses the first matching handle and resets its remaining duration to the incoming duration. It requires a finite Duration effect.
- **Extend Existing** reuses the first matching handle and adds the incoming duration to its remaining lifetime, saturating safely. It requires a finite Duration effect.
- **Replace Existing** removes every matching active instance and creates one incoming instance with a new handle.
- **Strongest Wins** compares `StackStrength`. A strictly stronger incoming effect replaces every matching instance; an equal or weaker application returns `Stronger Or Equal Effect Active` and identifies the winning handle without changing state.

Refresh and extend update the active definition, strength, periodic effect, copied context, and snapshotted modifier payload. Enable `Reset Period On Stack` when a successful refresh or extension should discard partial progress toward the next pulse. Already-earned catch-up backlog is preserved when the incoming effect remains periodic; switching to a non-periodic marker clears obsolete pulse timing and backlog. If execute-on-application is enabled, the initial pulse must succeed before any merge or replacement commits; a failure restores the exact active stack.

The application result exposes `StackDisposition` and the resulting `StackCount`. Successful applications publish `On Persistent Effect Stacked`; new and replacement instances also publish `On Persistent Effect Applied`. Handles retired by replacement publish `On Persistent Effect Removed` with **Replaced**.

Lifecycle tags, immunity, and cancellation

Persistent Effect assets can participate in the project's normal Gameplay Tag vocabulary without imposing plugin-specific tag names:

- `EffectTags` classify an active effect for cancellation queries, such as `Effect.Debuff` or `Effect.Debuff.Fire`.
- `RequiredTargetTags` must all match before application. Parent requirements match owned child tags.
- `BlockedTargetTags` reject application when any matches. These are the normal authoring point for immunity tags such as `State.Immune.Fire`.
- `GrantedTags` add one counted tag contribution per active instance. The exact contribution is removed on manual removal, expiration, replacement, or cancellation.
- `CancelEffectsWithTags` cancels every active instance whose classification matches any requested tag. A broad parent therefore supports dispel or cleanse families.

Application gates run before any state changes. Cancellations, old grant removal, incoming grants, and an optional execute-on-application pulse are projected as one transaction. A missing required tag, immunity blocker, tag-capacity error, invalid pulse, or other transaction failure leaves the original handles, attributes, and counted tags intact.

Refresh and extend reconcile their reused handle's grants against the incoming definition without removing shared tags. Independent instances each contribute their own count, so removing one source preserves the other sources. Granted and classification containers are snapshotted per active handle; later edits to the Data Asset cannot cause cleanup to remove a different tag set.

The apply result identifies a failed tag and underlying tag result when relevant, and reports `CancelledEffectCount` on success. Cancelled handles publish `On Persistent Effect Removed` with **Cancelled**.

Ability integration

Ability Data Assets can author `SelfPersistentEffects` and `TargetPersistentEffects`. Add the Persistent Effects Trait to every possible recipient; target persistent outcomes automatically make the ability require a valid target. Ability parameters and source identity flow into each stored effect context.

Costs, self/target Instant Effects, self Persistent Effects, and target Persistent Effects share one rollback boundary. Applications execute in that deterministic order. If any persistent outcome fails, Mass Forge restores cooldowns, shared cooldowns, charges, attributes, counted tags, cancelled or replaced instances, compact timer slots, payload snapshots, and the next handle ID. Provisional events are discarded. A successful ability result returns persistent handles in authored order and the aggregate cancellation count; a failure identifies the list index, source/target subject, underlying persistent result, and relevant tag details.

Handles, queries, and removal

Every successful application returns a world-unique `Mass Forge Persistent Effect Handle`. Independent policies create handles; refresh and extend deliberately return the reused handle.

- `Query Mass Forge Persistent Effect` returns identity, stack key, strength, current stack count, snapshotted classification/granted tags, persistent modifier contribution count, duration policy, remaining time, period, time to the next period, and completed pulse count.
- `Get Active Mass Forge Persistent Effects` lists every visible instance in compact fragment order.
- `Remove Mass Forge Persistent Effect` removes exactly one instance by handle.

The matching queued nodes defer application or exact-handle removal into one bounded FIFO stream. An accepted receipt contains a positive request ID; bind `On Persistent Effect Request Completed` and match its unified completion payload, which identifies Apply versus Remove and carries the full corresponding result. A rejected submission uses request ID `0` and never emits a completion. Definition identity and complete entity generation are revalidated at execution.

Queued requests execute at the start of the Persistent Effects phase, before that phase advances durations, due pulses, and expiration. Consequently, a queued removal accepted before a boundary can suppress that handle's otherwise-due pulse. Work enqueued by a Persistent Effect completion callback waits for the next coordinator tick. Configure **Max Queued Persistent Effect Requests Per Frame** and **Max Pending Queued Persistent Effect Requests** under **Project Settings** > **Mass Forge** > **Global Attribute Schemas** > **Runtime**.

Removal publishes `On Persistent Effect Removed` with **Manual**, **Expired**, **Replaced**, **Cancelled**, or **Entity Invalidated**. Applying beyond the 16-instance capacity returns `No Free Persistent Effect Slot` without changing state.

Scheduling is owned by the shared coordinator rather than an independent subsystem tick. Due work is sorted by complete entity handle and persistent handle; earned pulses run before same-boundary expiration, and carried pulse backlog delays expiration until it drains. The complete cancellation and callback precedence is defined in the deterministic gameplay ordering contract.

Persistent handles contain the owning entity generation. Destroyed owners are retired on the next valid subsystem pass, including a zero-delta cleanup pass, and cannot leak into a recycled entity index. See the entity lifecycle safety contract.

Durations and periods use scaled gameplay delta and stop under normal world pause. Entity-state snapshots serialize exact active instances from format 3 onward with projected duration, period phase/backlog, reversible modifier bases, owned tags, portable context, and soft definition identities; format 4 additionally preserves an active Ability Lifecycle. Restore allocates fresh handles; external effect-source entity handles and Actor/hit payloads do not cross worlds. See the durable persistence guide and time, travel, and save contract.

For a combined debug view, `Capture Mass Forge Entity Inspection` includes all visible persistent instances in monotonically increasing handle order alongside the entity's attributes, tags, abilities, and cooldown groups. See ENTITY_INSPECTION_GUIDE.md.

Context policy

The supplied effect context is copied when the instance is applied. Source/target entity handles, Actor references, source identity, origin, and scalar parameters therefore remain stable inputs for later pulses. Persistent modifier magnitudes are resolved at application.

Periodic Instant Effect modifiers can add an attribute-backed magnitude term and select one explicit policy:

- **Snapshot At Application** captures the source or target attribute before lifecycle tags, initial pulses, cancellations, or other state changes commit. The captured values remain in the stored context for every later pulse.
- **Evaluate On Execution** validates the capture at application, then reads the current source or recipient value before each pulse. If an entity or attribute later becomes unavailable, that pulse reports the normal detailed failure and the lifecycle instance remains active.

Snapshot storage is bounded to 32 unique subject/attribute pairs per active context. Duplicate captures share one stored value. Capture failures identify Source or Target, the attribute ID, and the underlying attribute access result. All modifiers in one pulse observe capture state from before that pulse's atomic writes.

The subsystem owns the shared Data Asset and context payload centrally while active. The Mass fragment stores only bounded handles and timing state, so no Actor or UObject is required per entity.

Blueprint events

- `On Persistent Effect Applied` fires after an instance becomes visible.
- `On Persistent Effect Stacked` fires after every successful add, refresh, extension, or replacement and reports the disposition and resulting stack count.
- `On Persistent Effect Period` fires after every pulse attempt and includes its sequence number and full Instant Effect result.
- `On Persistent Effect Removed` fires once for manual removal, expiration, replacement, cancellation, or invalidated entity ownership.
- `On Persistent Effect Removed With Context` is its additive, source-aware companion. It fires for the same removal and includes the application context snapshotted before the instance is retired. Existing bindings to `On Persistent Effect Removed` remain compatible.
- `On Persistent Effect Request Completed` fires once after each accepted queued Apply or Remove reaches a terminal result; successful lifecycle events are published first.

Get the world subsystem through `Get Mass Forge Persistent Effect Subsystem` to bind these delegates. C++ integrations can bind the corresponding `Native` delegate mirrors without dynamic dispatch.

Current scope

Persistent effects support independent, capped, refresh, extend, replace, and strongest-wins behavior, reversible additive/multiplicative/priority-override attribute aggregation, target tag gates, lifecycle-granted counted tags, classification-based cancellation, transactional initial-pulse rollback, fully transactional ability-authored self/target applications, bounded deferred Apply/Remove commands, durable entity-state save/load, and owner-only authoritative network baselines. Ability prediction is presentation-only and never predicts a Persistent Effect mutation. Additional game-specific Mass LOD timing policies remain project work.

GAMEPLAY SYSTEMS

Custom effect magnitude calculations

Mass Forge magnitude calculations are reusable Primary Data Assets for project-owned damage, healing, scaling, and resource formulas. Each receives the fully resolved built-in magnitude before returning the final value. They extend normal Instant Effects and reversible Persistent Effect contributions without putting a UObject on each entity or changing the compact Mass fragments.

Blueprint setup

1. Create a Blueprint derived from **Mass Forge Effect Magnitude Calculation**.
2. Assign a durable, unique **Calculation Id**.
3. Implement **Calculate Magnitude** and return a **Mass Forge Effect Magnitude Calculation Response**.
4. Assign the asset to an Instant Effect modifier or Persistent Effect attribute modifier under **Magnitude Calculation**.
5. Run Unreal Data Validation and **Tools > Mass Forge Project Health**.

The query contains the world context, complete read-only effect context, target attribute, zero-based modifier index, usage (`Instant Modifier` or `Persistent Contribution`), and `Input Magnitude`. The effect context includes source/target entity handles, optional represented Actors, source/effect/ability IDs, origin, optional hit data, scalar parameters, and any captured attributes.

Return `Success` only with a finite final `Magnitude`. Set a stable `Reason` for game UI, AI, analytics, and support. Mass Forge fills an omitted response `Calculation Id` from the asset. The native base class deliberately returns `Invalid Configuration` with `NoCalculationImplementation`, so an accidentally unimplemented asset fails closed.

For a data-only calculation, **Mass Forge Linear Magnitude Calculation** ships as a reusable example:

```
Output = Input Magnitude * Scale + Bias
```

Exact evaluation order

For each modifier, Mass Forge resolves values in this order:

1. Authored base `Magnitude`.
2. Optional named context parameter multiplied by `Magnitude Parameter Scale`.
3. Optional source/target attribute multiplied by `Magnitude Attribute Scale`, using the authored capture policy.
4. The assigned custom calculation receives that finite total as `Input Magnitude`.
5. The returned finite magnitude is used by the modifier's Set/Add/Multiply or persistent Add/Multiply/Override operation.
6. Schema clamping and the enclosing atomic transaction rules apply normally.

Every calculation in a transaction runs against committed pre-transaction entity state. Earlier staged modifiers are not visible through subsystem reads. This prevents authored ordering from exposing half-committed state.

Transaction and lifecycle guarantees

- **Direct and queued Instant Effects:** calculations run during execution-time staging. If any calculation fails, no attribute or counted-tag change commits and the result identifies the failed modifier.
- **Abilities:** calculations in self/target Instant Effects run inside the same transaction as costs, tags, cooldowns, charges, persistent effects, and lifecycle admission. Failure rolls the entire ability transaction back and reports the effect ID, modifier index, calculation ID, result, and reason.
- **Periodic effects:** the referenced Instant Effect is executed normally at each earned pulse, so its calculations evaluate once per pulse using that pulse's context and capture policy.
- **Reversible Persistent Effect contributions:** calculations run once when the incoming instance or replacement is staged. The finite output is snapshotted with that active handle, remains stable if the asset later changes in memory, and reverses exactly on removal/expiration.
- **Stack refresh/replace:** the incoming definition is calculated before projected aggregate validation. A failed calculation cannot partially refresh, replace, cancel, grant tags, or alter attributes.

Failure handling

Branch first on the enclosing result:

- Instant Effect: `MagnitudeCalculationFailed` in `EMFAttributeAccessResult`, then inspect `FailedModifierIndex` and `MagnitudeCalculationResponse`.
- Persistent Effect: `MagnitudeCalculationFailed` in `EMFPersistentEffectResult`, then inspect the same nested diagnostics.
- Ability: `MagnitudeCalculationFailed` in `EMFAbilityResult`, then inspect `FailedEffectId`, `FailedModifierIndex`, and the nested response.

Missing assets/IDs, the base class, intentional rejection, recursive evaluation, and non-finite output all fail without gameplay mutation. Do not retry in a tight loop; fix configuration or wait until the game rule represented by a deliberate rejection changes.

Performance boundary

Calculation assets are shared definitions, never per-entity state. They execute only on the direct/queued transaction path outside Mass processing. Keep Blueprint implementations bounded, deterministic, side-effect-free, and safe to evaluate repeatedly. Do not start effects, abilities, or other custom magnitude work from inside `Calculate Magnitude`; recursion is rejected.

The High Volume Effects Trait intentionally rejects modifiers with custom calculation assets. Its Mass processor hot loop accepts only finite, context-free, precompiled numeric operations and performs no asset loading, Blueprint dispatch, UObject work, allocation, or name lookup. Precompute high-volume values into explicit compiled effects when that path is required.

C++ implementation

Derive from `UMF_EffectMagnitudeCalculation` and override `CalculateMagnitude_Implementation`. Read `Query.InputMagnitude` and any immutable context needed, then return a finite response with a stable diagnostic. Use public, generation-checked subsystem reads for source/target state; do not retain query pointers or mutate gameplay.

The built-in `UMF_LinearEffectMagnitudeCalculation` is the smallest reference implementation. Use a distinct Primary Data Asset for each stable formula identity so Project Health can detect duplicate IDs and saved authored references remain traceable.

GAMEPLAY SYSTEMS

Gameplay tags

Mass Forge provides opt-in, counted Gameplay Tags for actorless and represented Mass entities. Projects use their normal Unreal Gameplay Tags dictionary; the plugin does not impose combat-specific tag names.

Setup

1. Define project tags under **Project Settings > Gameplay Tags**.
2. Add **Mass Forge Gameplay Tag Trait** to the Mass Entity Config.
3. Optionally place initial tags in `StartingTags`.
4. Use **Add Mass Forge Gameplay Tag**, **Remove Mass Forge Gameplay Tag**, or **Query Mass Forge Gameplay Tag** at a safe point. Use **Queue Add Mass Forge Gameplay Tag** or **Queue Remove Mass Forge Gameplay Tag** when the caller may overlap Mass processing.

The compact fragment stores at most 32 distinct exact tags per entity. Each tag has an unsigned count from 1 to 255, allowing independent systems and future effect stacks to share ownership. Adding an existing tag increments its count. Removing the final count releases the slot. Overflow, over-removal, invalid tags, missing traits, invalid entities, and full capacity return distinct result values.

`Query Mass Forge Gameplay Tag` returns a structured result, the exact count, and whether the requested match exists. Exact matching checks only the selected tag. Hierarchical matching also treats an owned child as matching its parent; for example, owning `State.Control.Stunned` satisfies a query for `State.Control`.

`Get Owned Mass Forge Gameplay Tags` enumerates every exact tag and count in stable lexical order. The same array appears in the unified entity inspection snapshot.

Successful count changes publish `On Gameplay Tag Changed` with entity, tag, previous count, and new count.

The two Queue nodes return a receipt immediately. Success means the bounded world queue accepted the request, not that the mutation has already happened. Keep the positive `Request Id`, bind `On Gameplay Tag Request Completed` on the Gameplay Tag Subsystem, and match the completion ID before consuming the final structured change. Rejected submissions use ID `0` and never publish a completion. Accepted requests execute FIFO, revalidate the complete entity generation, and publish the ordinary change event before their completion event.

Configure **Max Queued Gameplay Tag Changes Per Frame** and **Max Pending Queued Gameplay Tag Changes** under **Project Settings > Mass Forge > Global Attribute Schemas > Runtime**. Queue overflow returns `Gameplay Tag Queue Full`; spread or shed the burst instead of retrying in a tight loop.

Effect-granted tags

Mass Forge Instant Effect assets can add or remove counted tags alongside attribute modifiers. The plugin projects the entire effect first, so a missing tag fragment, absent tag, insufficient removal count, count overflow, or full 32-slot capacity rejects every attribute and tag change. Successful effect results include the ordered tag changes; failures expose `Gameplay Tag Failure`, the failed tag, and the exact tag result.

Tag-only effects are supported. This is useful for marks, immediate state transitions, cleanses, and ability self/target state changes. Ability transactions include costs and every referenced attribute/tag modifier in the same atomic boundary; failure also restores reserved cooldowns and charges. See INSTANT_EFFECTS_GUIDE.md.

Persistent lifecycle integration

Persistent Effect assets can require target tags, reject matching immunity tags, grant counted tags for exactly as long as an instance remains active, and cancel classified active effects. Independent sources contribute independent counts; expiration or removal of one instance decrements only one contribution. Refresh and extend reconcile the reused handle's grants, while replace and cancellation clean up every retired handle.

Add both the Gameplay Tag Trait and Persistent Effects Trait to participating entities. Use `EffectTags` to classify effects and `CancelEffectsWithTags` for hierarchical dispel/cleanse behavior. The persistent application result identifies requirement, blocker, and counted-tag transaction failures without partial state. See PERSISTENT_EFFECTS_GUIDE.md.

Ability gates

Every Mass Forge Ability Data Asset can configure:

- `RequiredSourceTags`: all must match the activating entity.
- `BlockedSourceTags`: any match rejects the activating entity.
- `RequiredTargetTags`: all must match the target.
- `BlockedTargetTags`: any match rejects the target.

Target tag rules automatically make a valid target mandatory. Required tags on an entity without the Gameplay Tag Trait return `Missing Gameplay Tag Fragment`. A missing requirement returns `Missing Required Tag`; a blocker returns `Blocked By Tag`. The activation result identifies the exact failed tag and whether it came from the source or target.

Ability tag checks happen before cooldown/charge reservation, costs, or effects, so rejection changes no state. Hierarchical matching is used, letting an ability require a broad parent category while project tags remain specific.

Safe access and current scope

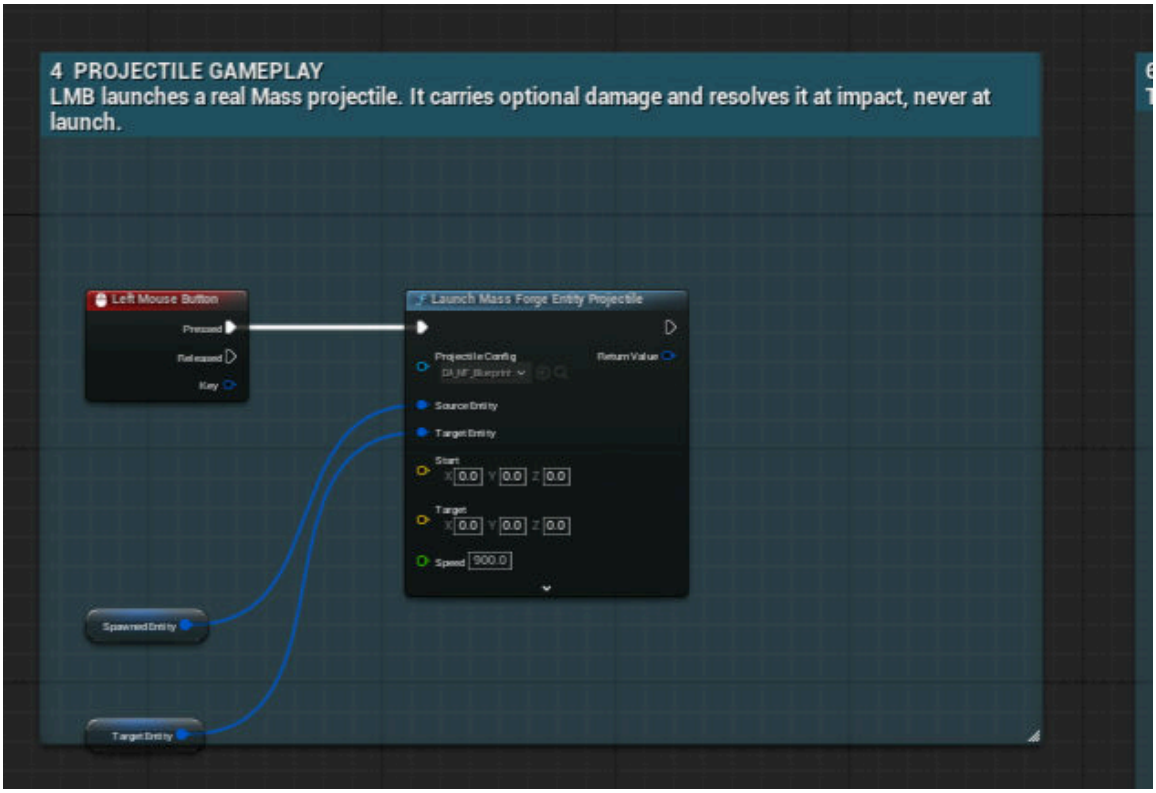
Direct tag mutation and query are rejected while Mass is processing. Queue direct count changes through the dedicated Add/Remove nodes; use a queued Instant Effect when the tag change must share one atomic transaction with attributes. Persistent effects and abilities provide transactional lifecycle grants, immunity gates, and cancellation. Durable entity-state snapshots preserve exact counted tags, and the owner-only network state stream replicates their authoritative baseline. Direct local tag mutations are never silently turned into RPCs; remote players must use the owner-bound network request component.

GAMEPLAY SYSTEMS

Mass Forge Projectiles

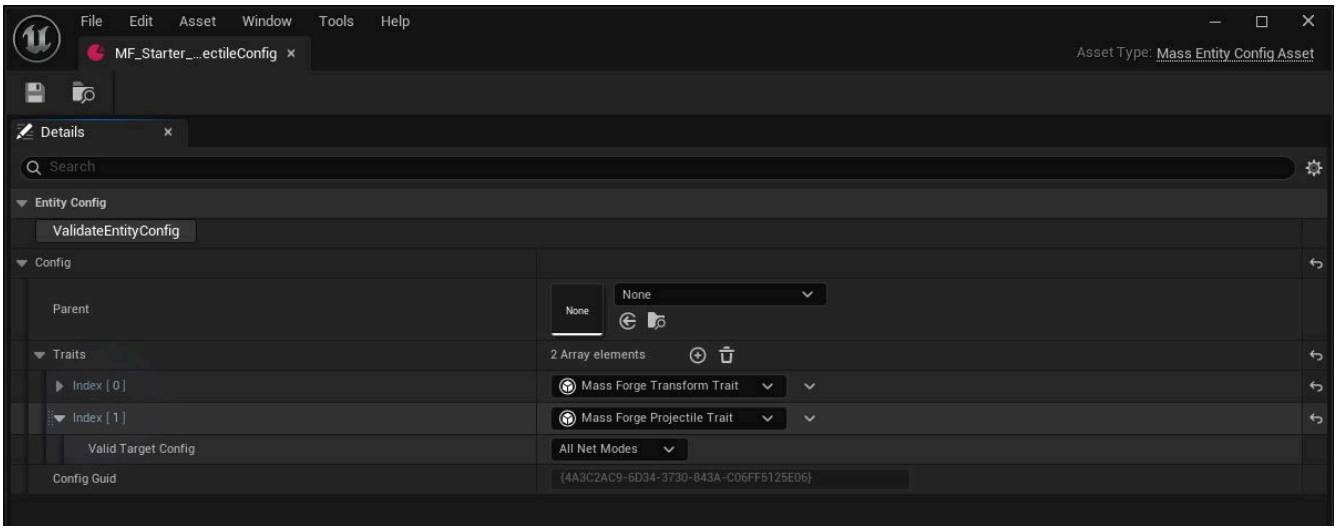
Mass Forge projectiles are short-lived Mass entities. Movement runs in `UMF_ProjectileProcessor`; `UMF_ProjectileSubsystem` creates projectile entities, exposes renderer-neutral snapshots, and crosses back to gameplay only after Mass processing has finished. This keeps projectile presentation replaceable and damage authoritative.

Setup



The convenience node launches a real Mass projectile. Its optional damage resolves from the projectile's impact path, never from the input event shown at the left.

1. Add **Mass Forge Projectile Trait** to a Mass Entity Config.



The generated projectile config is intentionally small. Transform stores its world state; Projectile stores the actorless flight and impact contract. Presentation remains a replaceable snapshot consumer. 2. Build an `FMFProjectileLaunchRequest` with that config, start/target locations, speed, optional arc, and optional authored release delay. 3. For a direct hit, assign a Damage Definition and `FMFDamageRequest`. For an area spell or project-owned effect, leave the definition empty and assign a stable `ImpactCue`. 4. Call `Launch Projectile` on `UMF_ProjectileSubsystem`. 5. Render `Get Projectile Snapshots` with Niagara, HISM, actors, or no visual representation. Gameplay does not depend on the renderer. 6. Listen to `On Projectile Resolved` for area queries, sound/VFX, healing, or other project-specific impact work.

Timing contract

`LaunchDelaySeconds` is the animation release point. A projectile remains hidden and stationary until that delay expires, then travels through the Mass processor. Optional damage is committed at impact-not at button press and not when an attack animation begins.

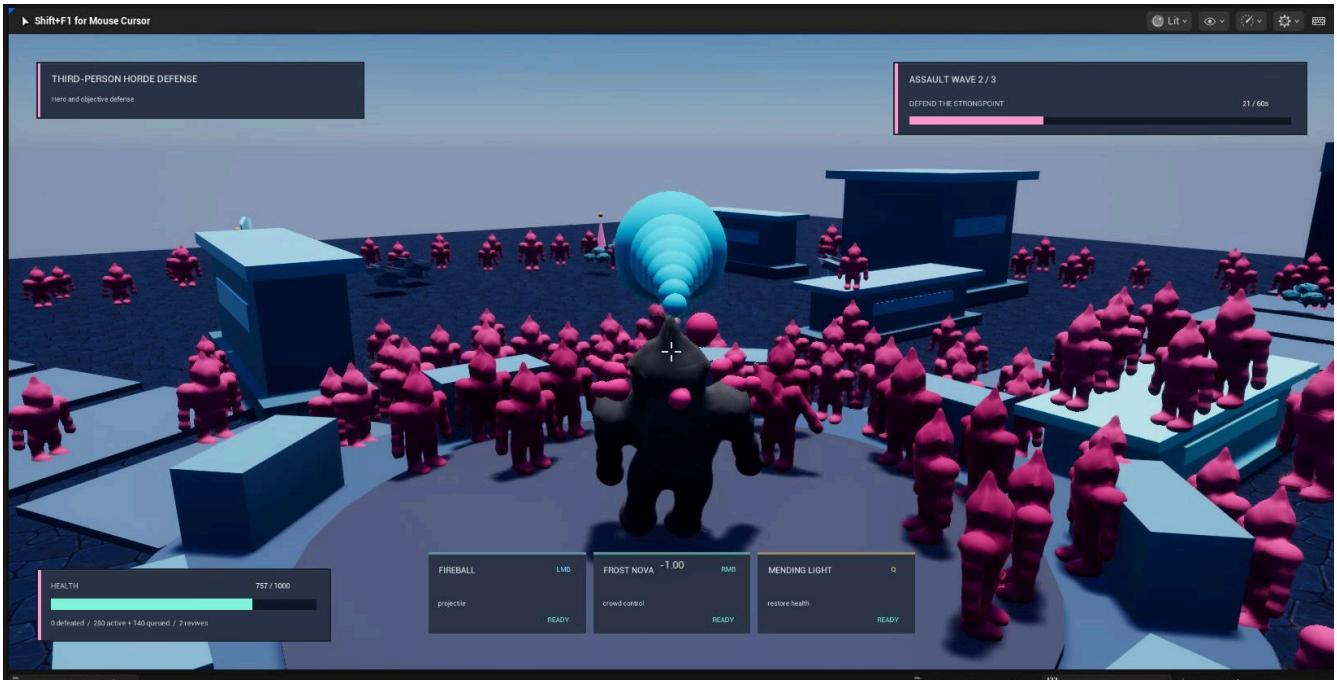
The included Horde Defense, Survivor Roguelite, and Battle Simulator examples all use this path:

- player fireballs, nova shards, and meteors;
- hostile horde bolts;
- battle ranger volleys;
- battle support/healing bolts.

Battle melee contacts also use the same impact queue on a renderer-free presentation channel. That keeps damage ordering symmetric between the two armies without drawing a fake projectile for a sword strike.

Their sample renderer uses three HISM batches and a bounds-stable sphere reference mesh. Each visible snapshot becomes a large core plus seven tapering trail beads, so direction remains readable without stretching a sphere into a distorted capsule. The renderer keeps instance slots stable and updates their transforms in place; it does not clear and recreate the HISM hierarchy each frame. Player shots select an enemy near the reticle ray instead of stopping on decorative props, clamp travel to at least 0.90 seconds, and leave an expanding 0.70-second twelve-bead impact ring. Hostile impacts use a much smaller six-bead ring so incoming fire remains legible without covering the player. Damage and the impact presentation are both triggered by the same resolved Mass projectile; neither occurs at button press. Renderer-free channel 3 carries deferred melee/contact transactions without pretending that a sword strike is a ranged shot. The same projectile entities can instead feed a project Niagara renderer without changing combat code.

The examples bind authored animation events to the launch delay: the player Cast clip releases at frame 13/30, while the 10 Hz enemy simulation quantizes its frame-10/30 Attack contact to the closest fixed step. Animation begins first, the projectile becomes visible at release, and damage is committed only at impact.



The large core and tapering beads make travel direction readable in the sample HISM renderer. This presentation can be replaced without changing launch timing, impact authority, or Damage Definition evaluation.

The Survivor example uses the same projectile API for automatic attacks. Allocation, active simulation, representation, and rendering remain separate scale decisions.

Scale and boundaries

This first-release implementation is a deterministic point-to-point/arc flight model. It is intentionally not a physics projectile or a swept collision system. Projects that need walls, ricochet, prediction, homing, or spatial collision can extend the processor/fragment while retaining the same launch, snapshot, safe-impact, and damage contracts.

Projectile entities are destroyed after impact or lifetime expiry. Pending impacts are bounded, map/reset boundaries clear active projectile entities, and gameplay damage is processed only while the Mass entity manager is idle.

The showcase audit publishes `ProjectileVisibleSamples` and `PlayerProjectileImpacts`. Both must be nonzero in Third-Person Horde Defense, its VAT duplicate, and Survivor Roguelite, preventing a transaction-only implementation from passing when the player cannot actually see projectile travel and impact feedback.

GAMEPLAY SYSTEMS

Regeneration and degeneration

Mass Forge regeneration profiles provide passive, linear attribute changes for actorless or represented Mass entities. They are opt-in and do not impose a combat formula.

Setup

1. Define the target and optional rate attributes in the global Vital, Combat, or Support schemas.
2. Create a **Mass Forge Regeneration Profile** Data Asset.
3. Give the profile a stable `ProfileId` and add up to eight rules.
4. Add **Mass Forge Attributes Trait** and **Mass Forge Regeneration Trait** to the same Mass Entity Config.
5. Select the profile on the regeneration trait.

The regeneration trait resolves attribute names while building the entity template. Each entity then stores only domains, slots, numeric rate settings, bounds, and timing state. The processor performs no name lookup, asset loading, allocation, or delegate dispatch inside its entity loop.

Rule formula

Each completed update applies:

`NewValue = CurrentValue + (ConstantRatePerSecond + RateAttributeValue x RateAttributeScale) x ElapsedSeconds`

The rate-attribute term is included only when **Use Rate Attribute** is enabled. A negative total rate produces degeneration. If the target schema enables clamping, the new value is restricted to that schema's minimum and maximum.

Examples:

- Health regeneration at 5 points per second: constant rate `5`, no rate attribute.
- Health regeneration driven by a stat: constant rate `2`, rate attribute `RegenerationPower`, scale `0.25`.
- Hunger drain at 1 point per second: constant rate `-1`.

Tick interval

- `0` applies the rule on every regeneration processor run.
- A positive value accumulates time and applies complete intervals in one deterministic linear update.
- Remainder time is retained, so variable frame lengths do not discard elapsed time.

The rate attribute is sampled when the completed interval is processed. This is intentionally a linear, low-cost policy; it does not reconstruct historical rate changes within an accumulated interval.

Validation and failure behavior

Profiles report invalid data when the ID is missing, no rules exist, more than eight rules are authored, multiple rules target the same attribute, an attribute is missing, an authored float is non-finite, or an interval is negative.

At template construction, rules whose attributes are absent from the configured schemas are skipped with an actionable log warning. If any global schema is missing, the regeneration fragment remains attached but inactive. At runtime, invalid slots or non-finite source state are ignored rather than contaminating the simulation.

Performance contract

The feature adds a fixed-capacity fragment only to entities with **Mass Forge Regeneration Trait**. Each fragment supports eight rules; use shared profiles to standardize authoring, but remember timing accumulators are intentionally per entity.

For adaptive scheduling, add Unreal's **Simulation LOD** trait and enable variable ticking. Non-due LOD chunks are skipped before entity iteration; due entities consume Unreal's full accumulated delta, while Entity Configs without variable ticking retain the ordinary per-frame path. High, Medium, Low, and Off guarantees, sampled-rate limitations, and setup are defined in the Regeneration Simulation LOD policy.

GAMEPLAY SYSTEMS

Derived attributes

Mass Forge derived-attribute profiles calculate deterministic linear stats for actorless or represented Mass entities. A profile can express formulas such as maximum health from vitality, armor from strength, or combat power from other calculated attributes.

Setup

1. Define every source and target attribute in the global Vital, Combat, or Support schemas.
2. Create a **Mass Forge Derived Attribute Profile** Data Asset.
3. Give the profile a stable `ProfileId` and add up to eight rules.
4. Add **Mass Forge Attributes Trait** and **Mass Forge Derived Attributes Trait** to the same Mass Entity Config.
5. Select the profile on the derived-attributes trait.

Each rule supports a base value and up to eight source terms:

`Target = BaseValue + sum(SourceAttribute x Coefficient)`

Examples:

- `MaxHealth = 100 + Vitality x 10`
- `Armor = 5 + Strength x 0.5`
- `CombatPower = AttackPower + Armor x 0.25`

Negative coefficients are valid. An empty term list produces a constant derived value. The target attribute's current schema bounds are compiled into the rule and applied after calculation.

Dependency graph

A source may itself be the target of another derived rule. Authors do not need to arrange rules manually: the profile builds a deterministic topological order and preserves authored order whenever multiple rules are otherwise independent.

Duplicate targets and circular dependencies are rejected. For example, `Armor` depending on `CombatPower` while `CombatPower` depends on `Armor` produces a validation error instead of frame-dependent results.

Derived outputs should be treated as processor-owned values. A direct change to a derived target can be overwritten on the next processor run.

Runtime behavior

The trait resolves attribute names and dependencies while building the entity template. The compiled formula graph is stored as a Mass const-shared fragment, so every entity in the archetype uses the same immutable rules without carrying a per-entity rule copy.

The processor evaluates the graph during PrePhysics and is explicitly ordered before Mass Forge regeneration. A regeneration rule can therefore sample a freshly calculated derived attribute in the same processing phase.

Calculations use double precision internally. If a runtime source or result is non-finite, that rule leaves its target unchanged rather than contaminating entity state.

Validation and limits

A profile reports invalid data for a missing ID, no rules, more than eight rules, duplicate or missing targets, more than eight terms in a rule, missing source attributes, non-finite numeric values, or any dependency cycle.

Schema membership is checked when the trait compiles the profile. A profile that references an attribute absent from the configured schemas is rejected for that entity template with an actionable log error.

Performance contract

Compiled rules contain only domains, slots, coefficients, base values, and target bounds. The entity loop performs no name lookup, asset loading, allocation, graph traversal, or delegate dispatch. The graph consumes shared archetype memory and adds zero bytes of rule storage per entity.

The current formula model is intentionally linear. Nonlinear and project-defined calculations are not part of the supported formula contract in this release.

EDITOR AND DEBUGGING

Gameplay asset authoring

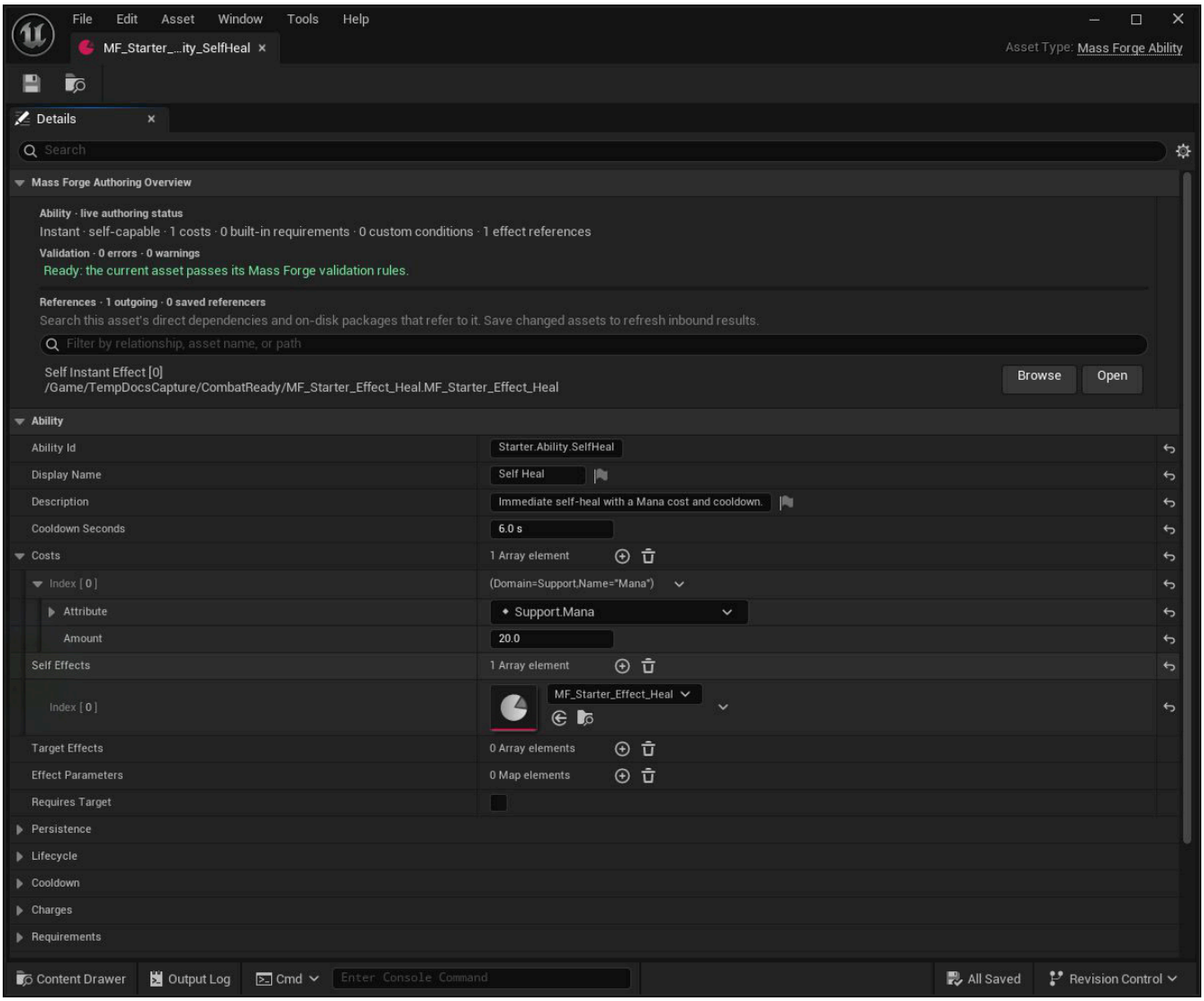
Mass Forge adds a live **Mass Forge Authoring Overview** to every **Mass Forge Instant Effect**, **Mass Forge Persistent Effect**, **Mass Forge Ability**, and **Mass Forge Ability Execution Plan** Data Asset. It keeps the normal Unreal Data Asset editor and property controls while adding the product-specific information designers need beside them.

Project-specific read-only admission rules are covered by the ability condition extensions guide. Condition assets appear as searchable outgoing dependencies of each Ability that uses them.

Project-owned damage, healing, and scaling formulas are covered by the custom effect magnitude calculations guide. Assigned calculation assets appear as searchable outgoing dependencies of Instant and Persistent Effects.

Dynamic effect selection is covered by the ability execution plans guide. Ability-to-plan and Static-plan-to-effect relationships appear as searchable outgoing dependencies.

Live authoring status



The overview makes readiness and dependencies visible above the ordinary properties. Here the ability validates cleanly and links directly to its Heal effect while the cost and cooldown remain editable below.

The overview updates when an editable property or array child changes. It shows:

- The asset kind and a compact execution summary.
- Every current Mass Forge validation error and warning inline.
- A green readiness message when the asset passes its current validation rules.
- Direct dependencies such as an Ability's targeting policies/execution plan/effects, a Static plan's phase payloads, a Persistent Effect's periodic Instant Effect, or a modifier's custom magnitude calculation.
- Saved assets that refer to the open asset according to Unreal's Asset Registry.

Inline results use the same `IsDataValid` contract as Unreal content validation. The overview is fast feedback during authoring; teams should still run their normal Validate Assets and packaged-build gates before release.

Searchable references

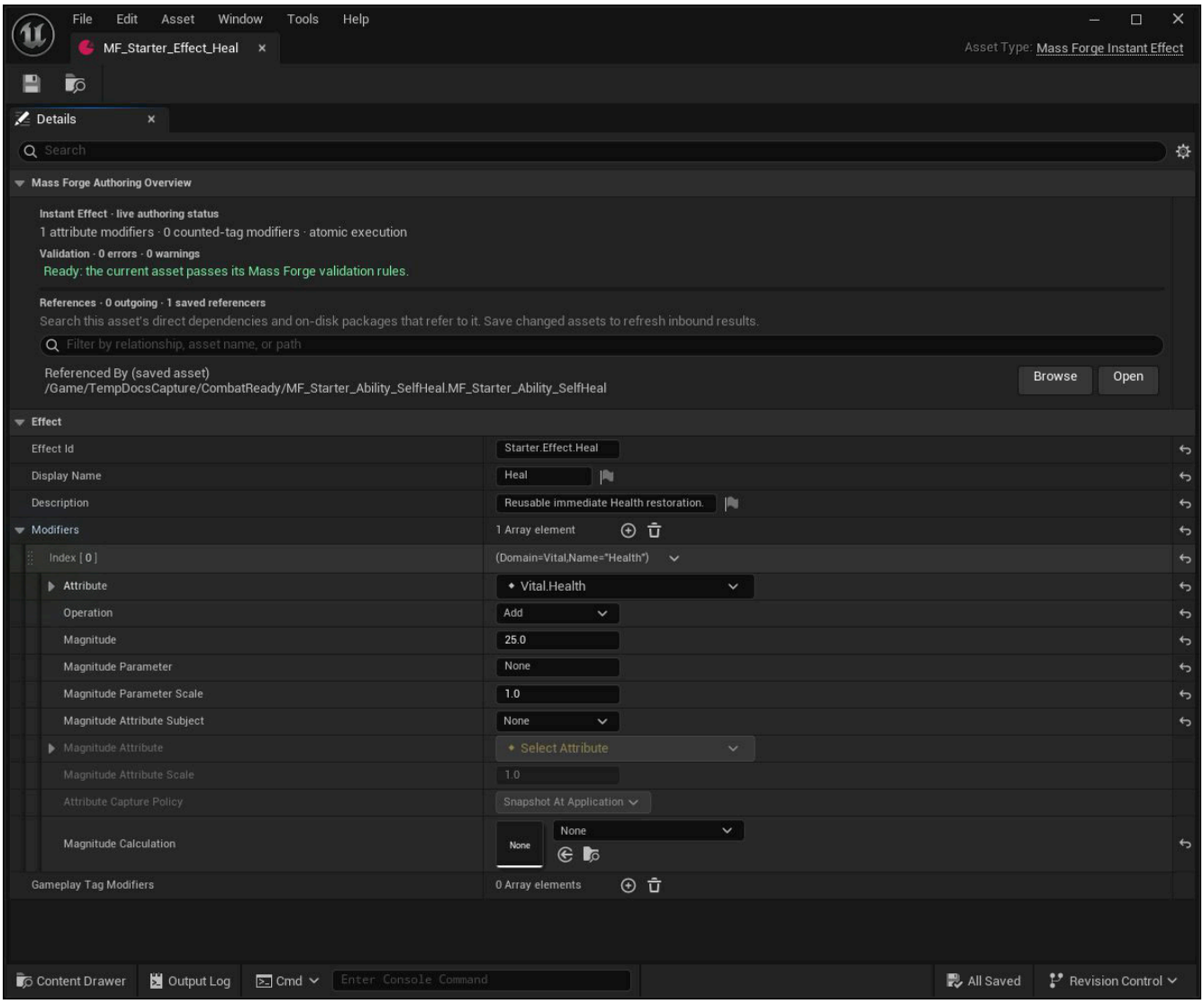
Use the reference search field to filter by relationship, asset name, or object path. Each resolved row has:

- **Browse** to select the asset in the Content Browser.
- **Open** to open it in its editor.

Outgoing references reflect the open in-memory asset immediately. Inbound **Referenced By (saved asset)** rows come from on-disk Asset Registry dependencies. Save changed assets before relying on the inbound list. The panel bounds inbound output to 128 rows so an unusually connected asset cannot make the Details view unresponsive.

An Instant Effect with no custom calculations normally has no outgoing asset dependencies, but its inbound list is particularly useful for finding Abilities and Persistent Effects that consume it.

Instant Effect workflow



The reference row answers "what uses this?" while the modifier shows exactly what will change. Both are visible in the same standard Data Asset editor.

1. Assign a stable `EffectId`.
2. Add at least one attribute modifier or counted Gameplay Tag modifier.
3. Optionally assign a validated custom magnitude calculation after configuring base, parameter, and captured-attribute contributions.
4. Resolve every inline error, including invalid identities, tags, counts, captured-attribute pairs, calculation references, and non-finite values.
5. Save the asset, then review its inbound referencers before changing behavior used by other assets.
6. Validate the asset and exercise its intended direct, queued, or cross-ownership route.

Persistent Effect workflow

1. Assign a stable `EffectId` and choose the lifetime and stacking policies.
2. Configure lifecycle tags and/or reversible attribute modifiers; optional custom magnitudes are evaluated and snapshotted once per admitted contribution.

3. For periodic behavior, set a valid positive period and choose a configured Instant Effect.
4. Use the reference row to open the periodic effect and verify its atomic payload.
5. Resolve all lifetime, stacking, capacity, tag-overlap, modifier, and period diagnostics before use.

Ability workflow

1. Assign a stable `AbilityId` and choose Instant, Cast, or Channel execution.
2. Configure costs, requirements, targeting policies, target rules, cooldowns, and charges.
3. Add activation and channel effects in the intended source/target collections.
4. Supply every context parameter required by referenced Instant and Persistent Effects in `EffectParameters`.
5. Use the dependency rows to inspect every referenced policy/effect without searching the Content Browser manually.
6. Resolve all inline errors and warnings, then save and validate the complete dependency chain.

The Ability validator loads configured effect dependencies to detect missing parameter defaults. This catches a common integration failure before a runtime activation request can encounter it.

Ability Execution Plan workflow

1. Assign a unique `PlanId` and choose Append or Replace merge behavior.
2. Prefer **Mass Forge Static Ability Execution Plan** when separate fixed activation/channel payloads are sufficient.
3. Derive a Blueprint/C++ plan only when output must depend on the supplied read-only query.
4. Keep target-producing plans on Abilities with **Requires Target**, keep channel payloads Instant-only, and stay within 64 effects and 64 parameters.
5. Confirm the overview resolves every phase-specific effect dependency and Project Health reports no duplicate plan ID.

Identity and change discipline

`AbilityId` and `EffectId` are gameplay identities used in runtime results, events, inspection, and Primary Asset IDs. Treat released IDs as stable even though Unreal object references can survive an asset rename. Before changing an ID, audit Blueprint logic, AI, UI, telemetry, save data, and project code that may compare or store it.

Release checklist

- The overview reports zero errors on every shipped Ability, Execution Plan, Instant Effect, and Persistent Effect.
- Warnings have been reviewed and intentionally accepted or corrected.
- Every outgoing dependency resolves and opens.
- Changed assets were saved before reviewing inbound referencers.
- Referencer assets were checked for behavior changes caused by the edit.
- Ability effect parameters cover every parameterized referenced modifier.
- Every assigned magnitude calculation resolves, has a unique stable ID, returns finite results, and has failure-path coverage.
- Every assigned execution plan resolves, has a unique stable ID, stays bounded, and has preflight, commit, and channel-pulse failure coverage where applicable.

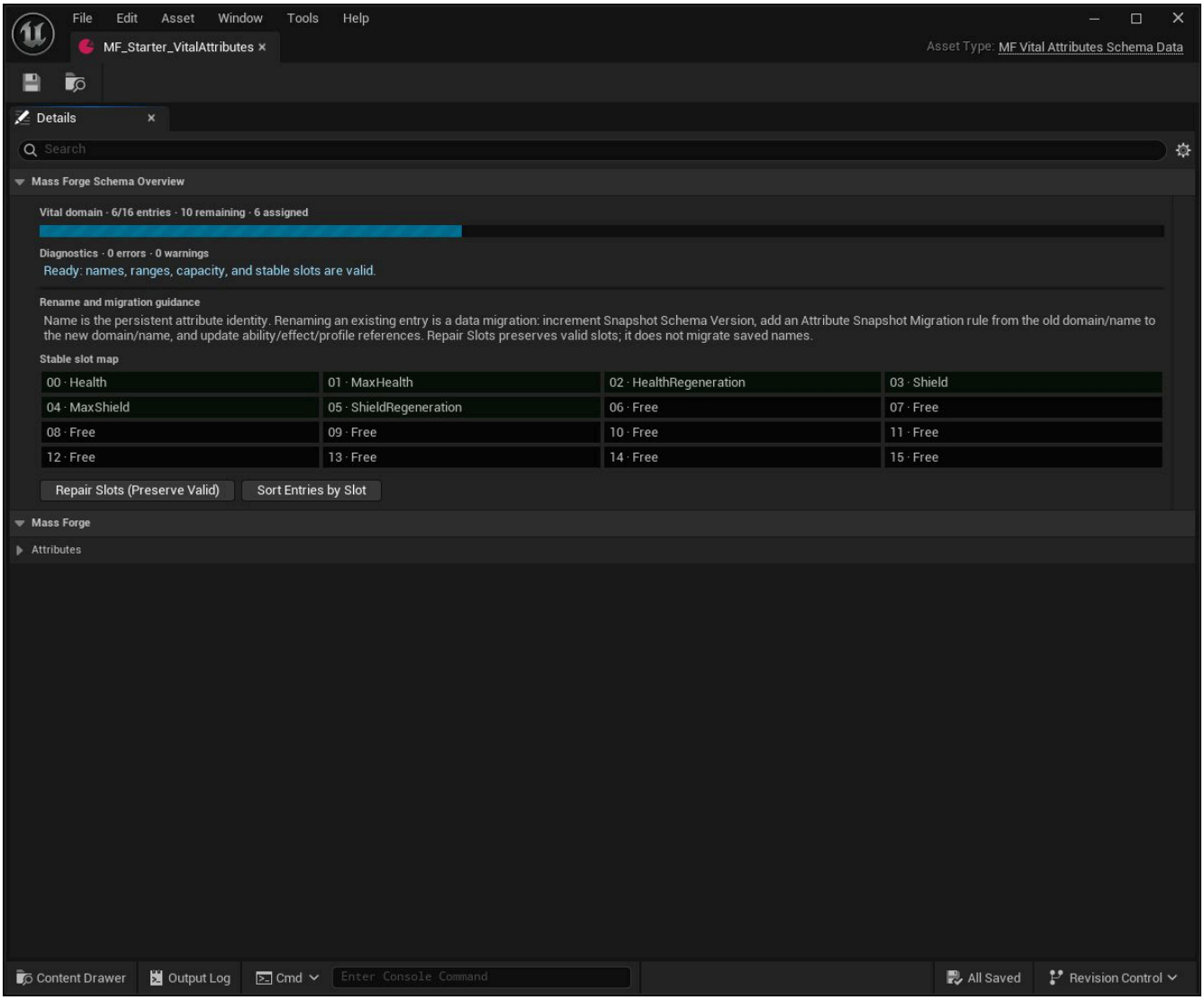
- Content validation succeeds for the full asset folder.
- Runtime automation, representative gameplay scenarios, and packaged builds pass.

EDITOR AND DEBUGGING

Attribute schema editor

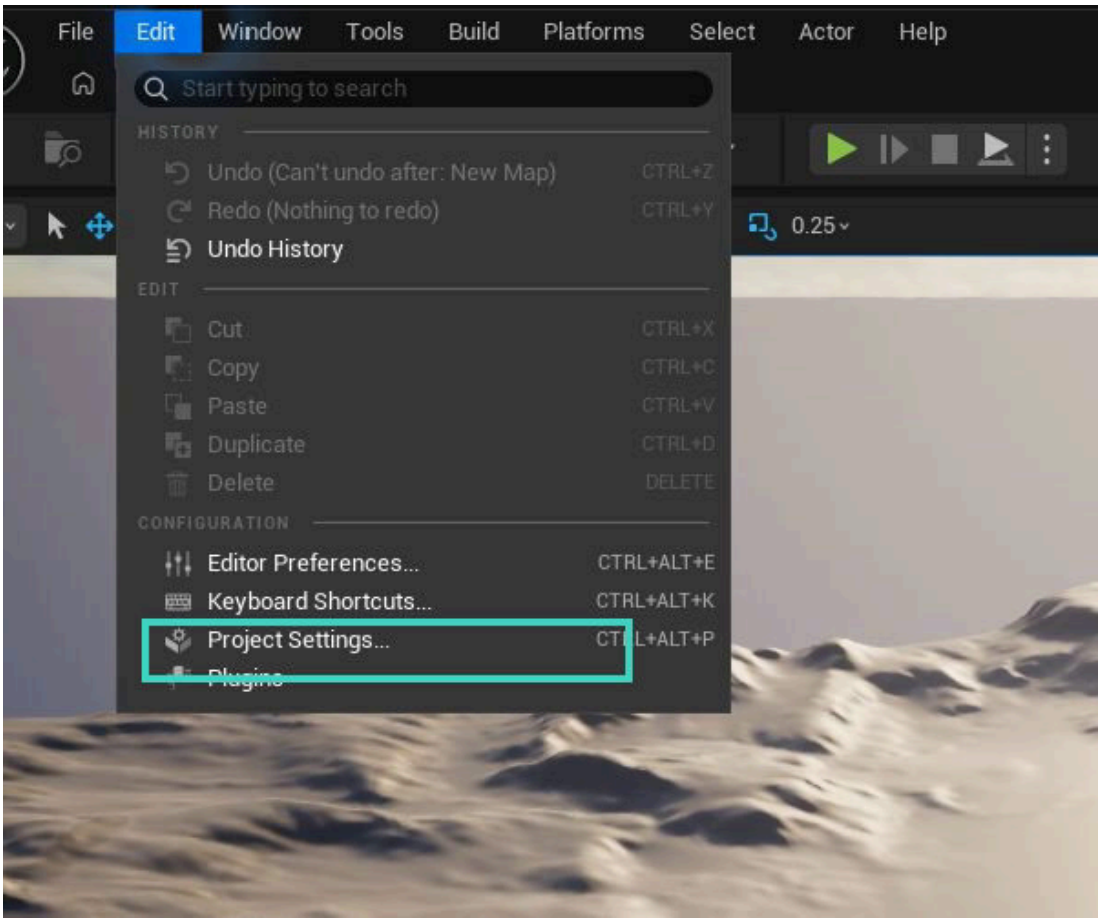
Mass Forge adds a focused authoring overview to every **Mass Forge Vital Attributes Schema**, **Mass Forge Combat Attributes Schema**, and **Mass Forge Support Attributes Schema** Data Asset. Open one of those assets and use the **Mass Forge Schema Overview** section at the top of the Details panel.

The overview is an authoring aid; the schema asset remains ordinary Unreal data that can be reviewed, versioned, and referenced by the project.

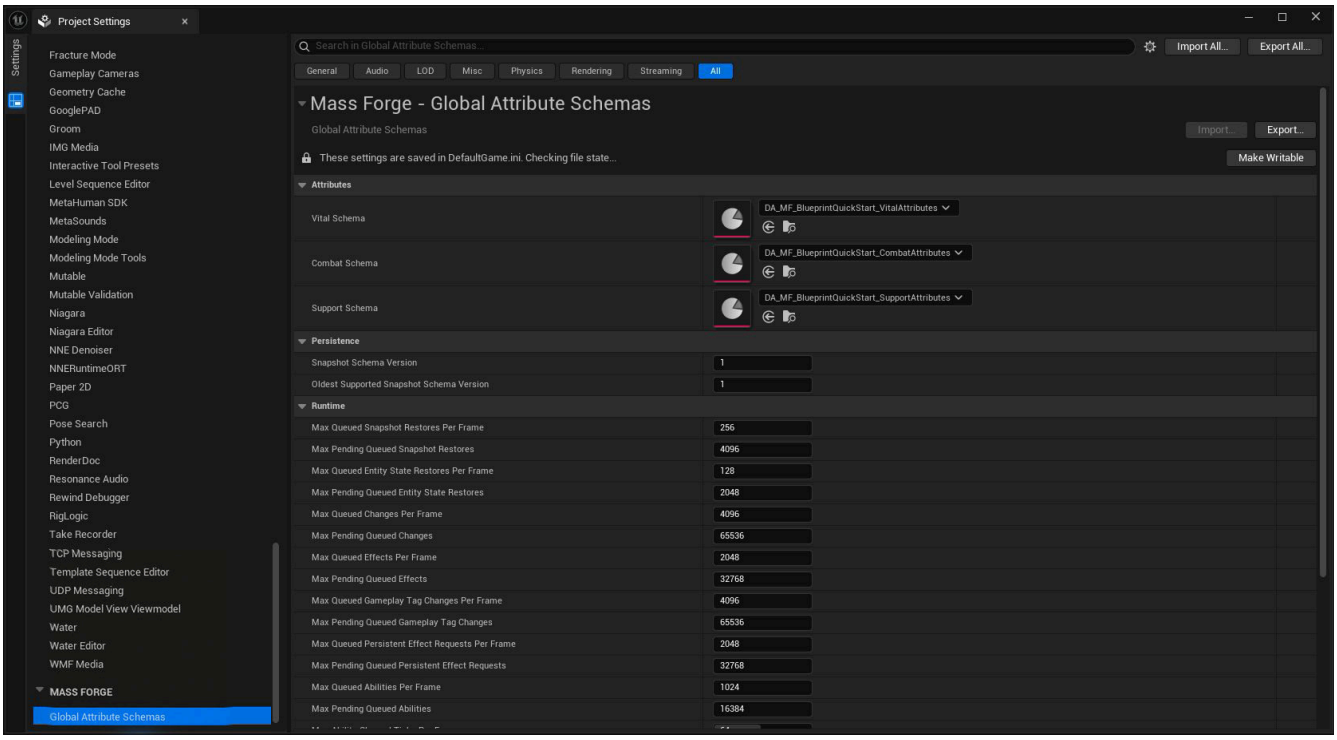


This is the healthy state to look for before dependent assets are authored: no errors or warnings, every entry assigned once, and unused slots still available.

For a first-time project, **Tools > Mass Forge Entity Config Wizard** can create valid starter schemas and register missing domains without modifying schemas that are already configured. See ENTITY_CONFIG_WIZARD_GUIDE.md.



*Open the standard project settings from **Edit > Project Settings** after generating or manually creating schemas.*



The three domain assignments are the project-wide source of truth used by schema-backed Blueprint pickers, Data Assets, bindings, and runtime resolution.

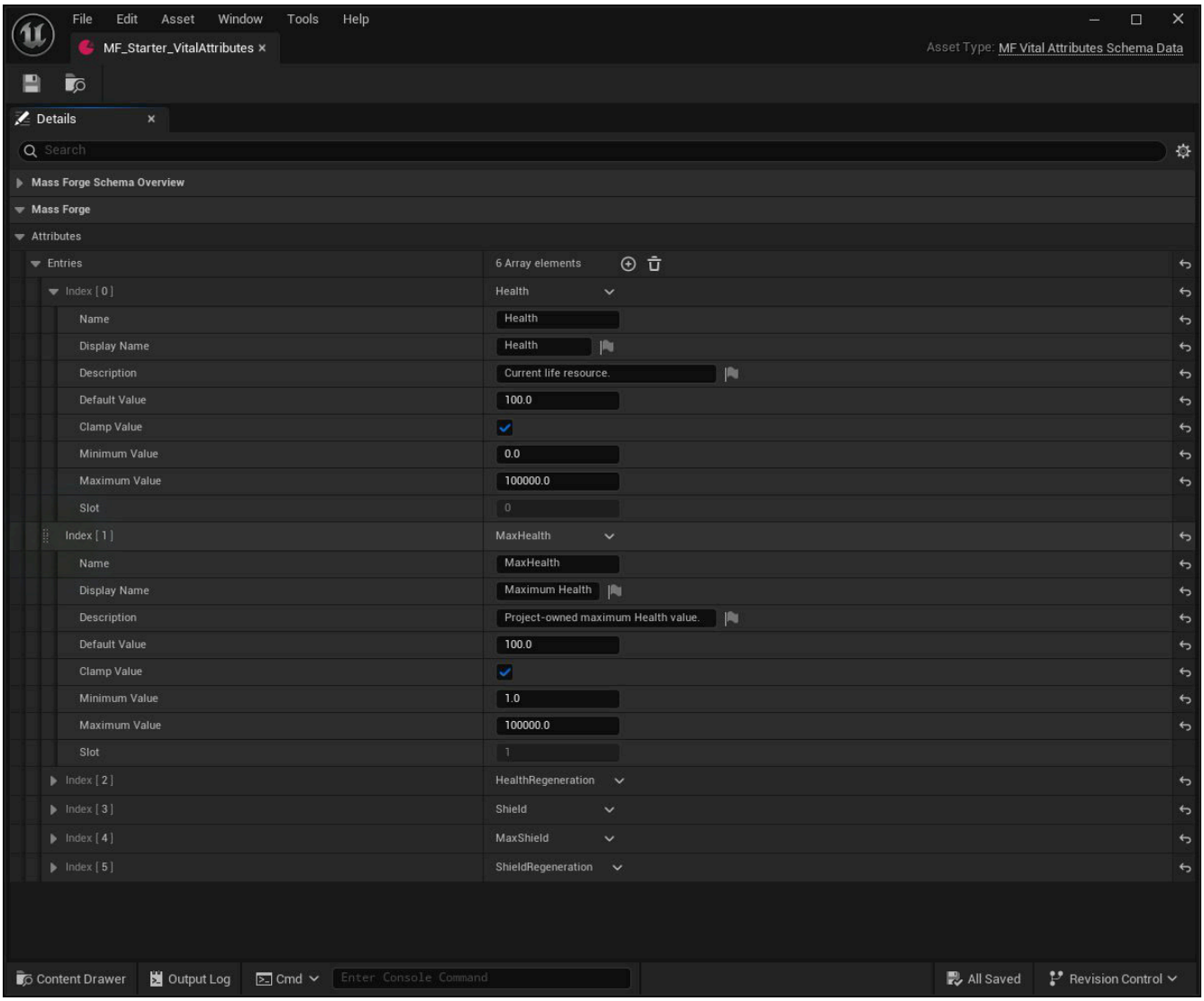
What the overview shows

- The schema domain and its fixed runtime capacity: Vital has 16 slots, Combat has 48, and Support has 32.
- Authored entry count, assigned-slot count, remaining capacity, and a color-coded capacity meter.
- A complete slot map, including unused slots and collisions.
- Blocking errors for missing or duplicate names, duplicate or out-of-range slots, invalid clamp ranges, defaults outside their authored clamp range, non-finite values, and over-capacity schemas.
- A warning when the schema is close to capacity.
- Rename guidance beside the repair controls so identity changes are not mistaken for layout cleanup.

The normal Unreal validation command remains authoritative for cook and release checks. The overview brings the same high-value problems into the asset authoring loop before packaging.

Recommended authoring workflow

1. Add an entry and give it a unique, stable `Name`. Treat this name plus its domain as the attribute's persistent identity.
2. Set the display name, description, default value, and optional minimum/maximum bounds.
3. Use **Repair Slots (Preserve Valid)** if the entry has no slot or the panel reports a collision.
4. Confirm the capacity meter and diagnostic list are healthy.
5. Assign the schema under **Project Settings > Mass Forge > Global Attribute Schemas**.
6. Run Unreal asset validation before release.



Edit the fields in the normal Details panel. The domain plus `Name` is persistent identity; `Slot` is compact runtime layout, while display name and description are designer-facing metadata.

Repair Slots (Preserve Valid)

Repair keeps every valid, unique existing slot. It assigns only missing, invalid, or colliding entries to the lowest available slot. This makes routine edits predictable and avoids needless runtime-layout churn.

Perform repair and other schema-layout work before spawning gameplay entities. Existing live fragments retain their old compact array layout; a schema refresh invalidates compiled bindings but intentionally does not guess how project-owned live entities should be rebuilt. For a running/save-loaded game, capture or retain an identity-based snapshot, recreate the affected entity after the new schema is active, and restore the snapshot. Restore resolves domain/name identities into the new slots.

Entries with an empty name or a duplicate name remain unassigned until their identity error is corrected. If the schema has more valid named entries than its domain can hold, the extra entries remain unassigned and the asset continues to report a blocking capacity error.

The repair action is undoable.

Sort Entries by Slot

Sorting changes only the order of rows in the Details panel. Assigned entries appear in ascending slot order and unresolved entries follow them. It does not change any slot number or attribute identity, and it is undoable.

Sorting is useful after repair because the authored list then mirrors the compact runtime layout.

Rename and migration rules

Changing `Name` or moving an attribute to another schema domain changes its persistent identity. Slot repair does not create a compatibility alias and sorting does not make a rename safe for existing saves.

For a released project with stored snapshots:

1. Increment **Snapshot Schema Version** in **Project Settings > Mass Forge > Global Attribute Schemas > Persistence**.
2. Create a **Mass Forge Attribute Snapshot Migration** Data Asset for the old version to the new version.
3. Add a Rename/Move rule from the exact old domain and name to the exact new domain and name. Add an explicit Remove rule when retiring an attribute.
4. Update ability, effect, damage, regeneration, derived-attribute, UI, AI, and project Blueprint references.
5. Migrate old snapshots through every required version before calling restore.
6. Test both a new entity and a migrated saved entity.

See PERSISTENCE_GUIDE.md for the full atomic migration and restore contract.

Configured, unambiguous schema entries automatically populate the Attribute ID picker used by gameplay Data Assets and literal Blueprint pins. Duplicate names, slot collisions, missing slots, and out-of-range slots are excluded until repaired. See BLUEPRINT_AUTHORING_GUIDE.md.

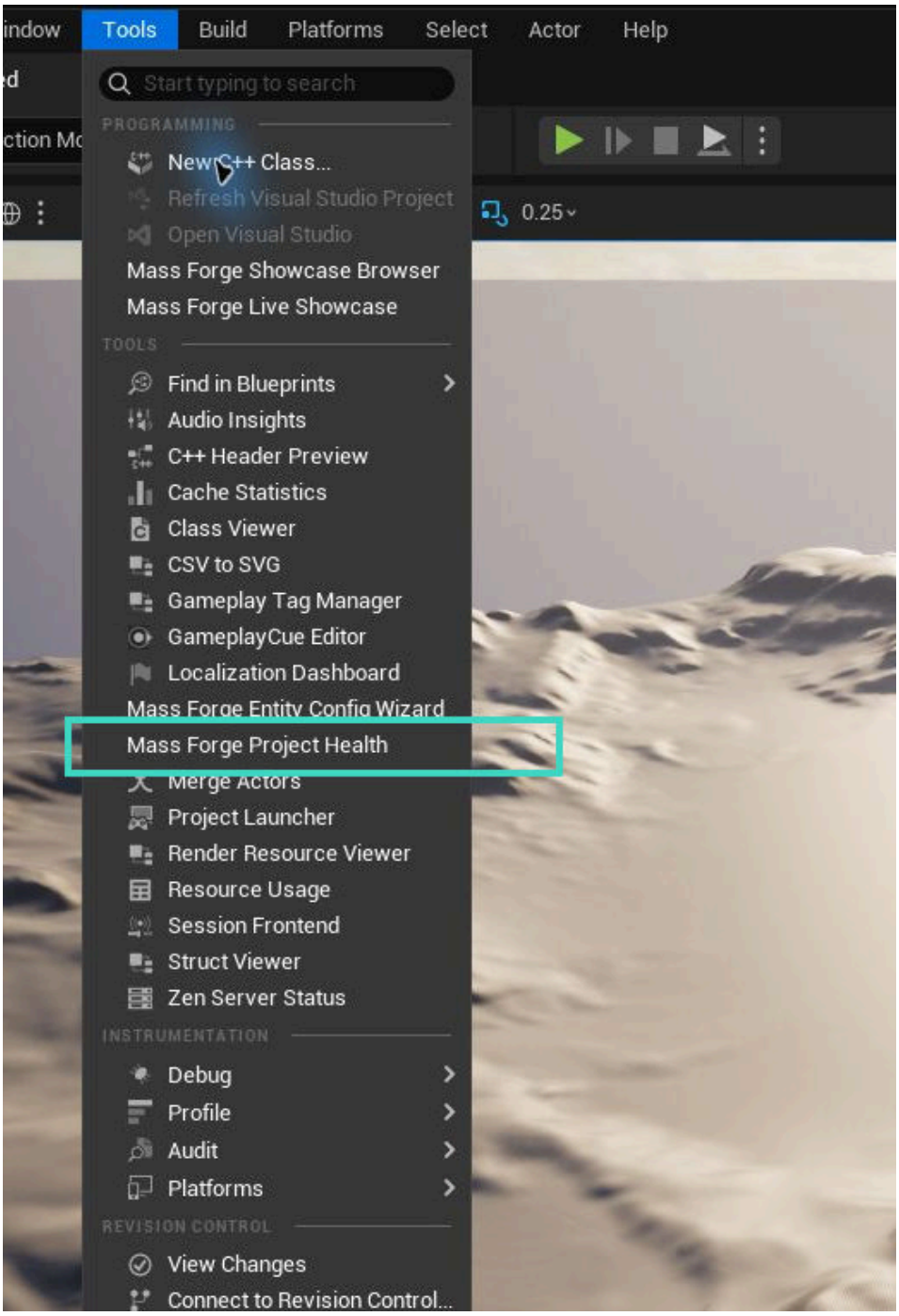
Release checklist

- Every entry has a unique, stable name.
- Every valid entry has one unique in-range slot.
- The authored row count is within the domain capacity.
- Defaults and clamp bounds are finite; every minimum is less than or equal to its maximum.
- Intended defaults lie inside their clamp range.
- The capacity meter leaves enough room for the project's planned expansion.
- Intentional renames or removals have a schema-version bump and migration asset.
- All dependent Data Assets and Blueprints have been updated.
- Content validation, automation tests, and a packaged build pass.

EDITOR AND DEBUGGING

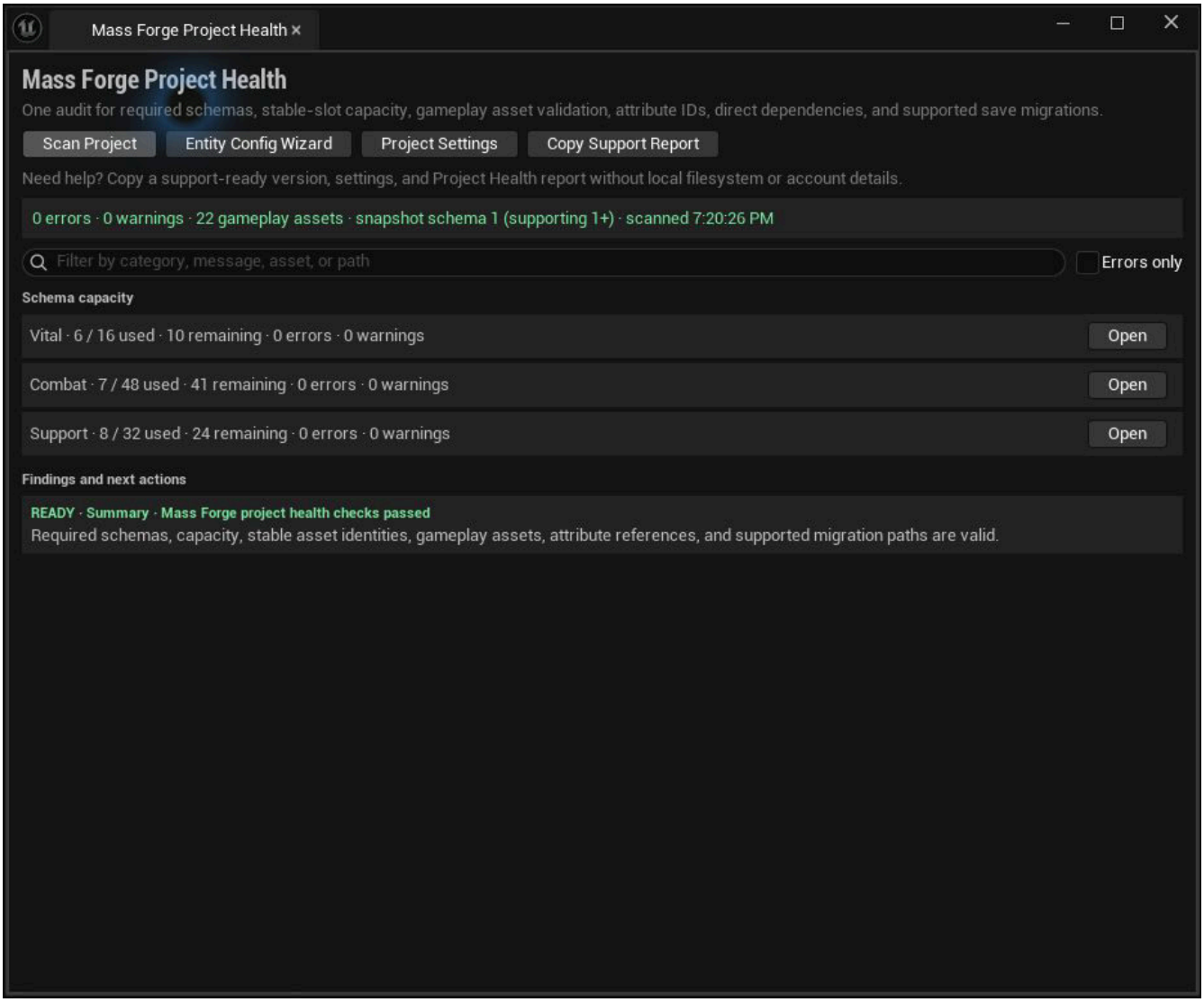
Project Health dashboard

The **Mass Forge Project Health** dashboard is the project-wide preflight for Mass Forge authoring. Open it from **Tools > Mass Forge Project Health**. It scans project-owned Mass Forge assets beneath `/Game`, checks the configured global schemas, and turns validation failures into searchable rows with direct **Browse** and **Open** actions.



*Open the dashboard from Unreal's standard **Tools** menu. Run it after changing schemas or gameplay assets and before producing a release build.*

The dashboard is an editor-only tool. It does not add runtime fragments, load gameplay assets in packaged builds, or change an entity. A scan is read-only and runs once when the panel opens or when **Scan Project** is selected.



A release-ready scan is explicit: zero errors, zero warnings, every configured schema within capacity, and a green summary. This disposable UE 5.8.2 validation host contains 22 project-owned starter gameplay assets beneath `/Game`; shipped plugin examples beneath `/MassForge` are still validated separately by the release commandlet.

Support report

Select **Copy Support Report** to perform a fresh scan and copy a support-ready text report. The same action is available through the editor console command `MassForge.CopySupportReport`.

The report includes the Mass Forge and Unreal Engine versions, platform/OS/build configuration, project name, health summary, schema capacities, complete findings, snapshot-version window, and configured runtime queue limits. It does not collect local filesystem paths, usernames, machine names, account data, saved-game contents, or live gameplay state. Windows absolute paths and exact machine-identity strings are redacted even when they occur inside project-authored validation text. Asset names and remaining validation details are included, so review the report before sharing it.

Use the report with SUPPORT_REQUEST_TEMPLATE.md and follow SUPPORT_POLICY.md. Current product boundaries are tracked in KNOWN_ISSUES.md.

What a scan checks

Required schema setup

Vital, Combat, and Support schemas are all required. A missing or unloaded configured schema is an error. Use **Project Settings** to open **Project Settings > Mass Forge > Global Attribute Schemas**, or use **Entity Config Wizard** to create and register only the missing schemas.

Every configured schema receives a capacity row showing:

- entries used and the domain maximum;
- remaining stable slots;
- blocking schema errors;
- near-capacity warnings.

The dashboard reuses the same analysis shown in the schema's Details panel, so duplicate names, empty names, duplicate or invalid slots, non-finite values, invalid clamp ranges, defaults outside bounds, and over-capacity schemas have one definition across both tools. A schema warns when two or fewer slots remain.

Gameplay asset validation

The scan discovers these project assets recursively beneath `/Game`:

- Mass Forge Ability;
- Mass Forge Instant Effect;
- Mass Forge Persistent Effect;
- Mass Forge Damage Definition;
- Mass Forge Derived Attribute Profile;
- Mass Forge Regeneration Profile;
- Mass Forge Attribute Snapshot Migration;
- Mass Forge Ability Targeting Policy subclasses.

Each asset runs its native Mass Forge validation. Ability, Instant Effect, and Persistent Effect references are also resolved so stale effect or policy paths become direct **Asset References** errors.

Stable asset identities

Project Health groups every discovered Mass Forge Primary Data Asset by its complete Primary Asset ID: asset type plus its authored stable ID. Two assets of the same kind may not claim the same identity. A collision is a blocking **Stable IDs** error on every conflicting asset and lists the full conflict set, so searching by stable ID or by any asset name finds the problem.

The check covers abilities, Instant Effects, Persistent Effects, Damage Definitions, derived-attribute profiles, regeneration profiles, snapshot migrations, and targeting policies. The same text may intentionally appear in different Primary Asset types-for example, an Instant Effect and Persistent Effect can both use `Status.Burning`-because the complete identities remain distinct.

An asset's Content Browser name and path are presentation/organization; its authored `AbilityId`, `EffectId`, `DamageId`, `ProfileId`, `MigrationId`, or `PolicyId` is its durable gameplay identity. Renaming or moving the Unreal asset therefore preserves its Primary Asset ID. Changing the authored ID is an identity migration and must follow VERSIONING_AND_UPGRADES.md, including all project-owned save, UI, telemetry, Blueprint, and code references.

Attribute references

The dashboard checks active attribute-ID fields against the currently configured schemas. Coverage includes:

- ability costs and requirements;
- instant-effect target modifiers and optional source/target magnitude captures;
- persistent-effect lifetime modifiers and optional magnitude captures;
- damage Health and every enabled Shield, Power, mitigation, and critical attribute;
- derived-rule targets and source terms;
- regeneration targets and enabled rate attributes;
- rename targets in snapshot migrations.

Inactive conditional fields are intentionally ignored. For example, an empty Shield attribute is valid while **Use Shield** is disabled. Migration source IDs are also intentionally checked only for structural validity by the migration asset: they describe an older schema and are normally absent from the current schemas. Rename targets must resolve in the current schemas.

An invalid row includes its authored property path, such as `Costs[0].Attribute` or `Rules[2].Terms[1].SourceAttribute`, so the designer can go straight to the affected field.

Snapshot migration coverage

Two project settings define the supported save window:

- **Snapshot Schema Version** is the current version written into new snapshots.
- **Oldest Supported Snapshot Schema Version** is the oldest released integer version the project promises to load.

For every integer version from the oldest supported version up to-but not including-the current version, the dashboard requires a directed chain of valid migration assets that reaches the current version. Chained and intentional jump migrations are supported. A `1 → 3` asset covers version 1, but it does not cover version 2 unless a separate path from 2 to 3 exists.

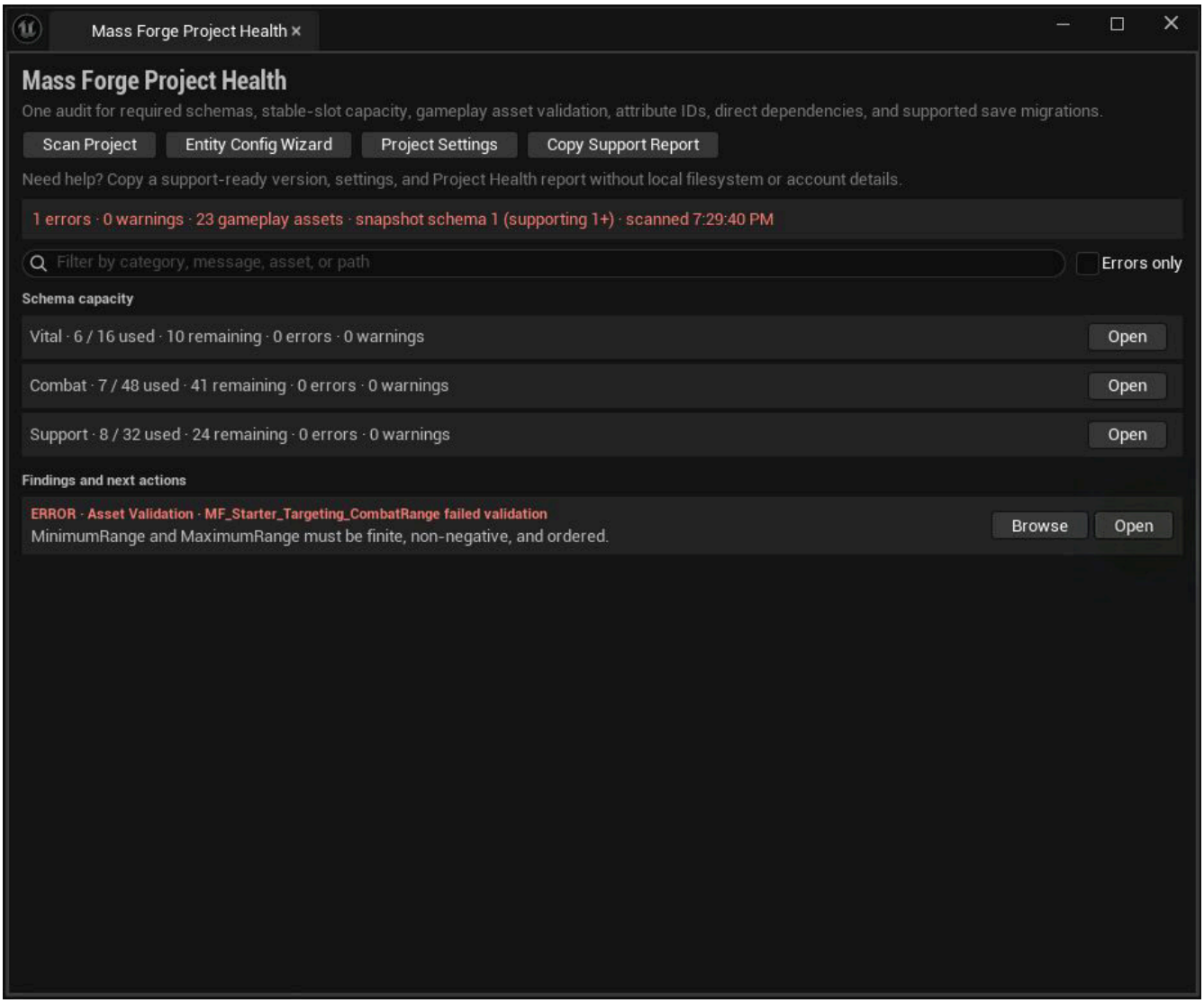
Set the oldest supported version equal to the current version only when the release intentionally retires every older save. A migration outside the declared window is a warning, because it may be obsolete or may indicate that the support window was configured incorrectly. Invalid migration assets never count toward coverage.

Working through findings

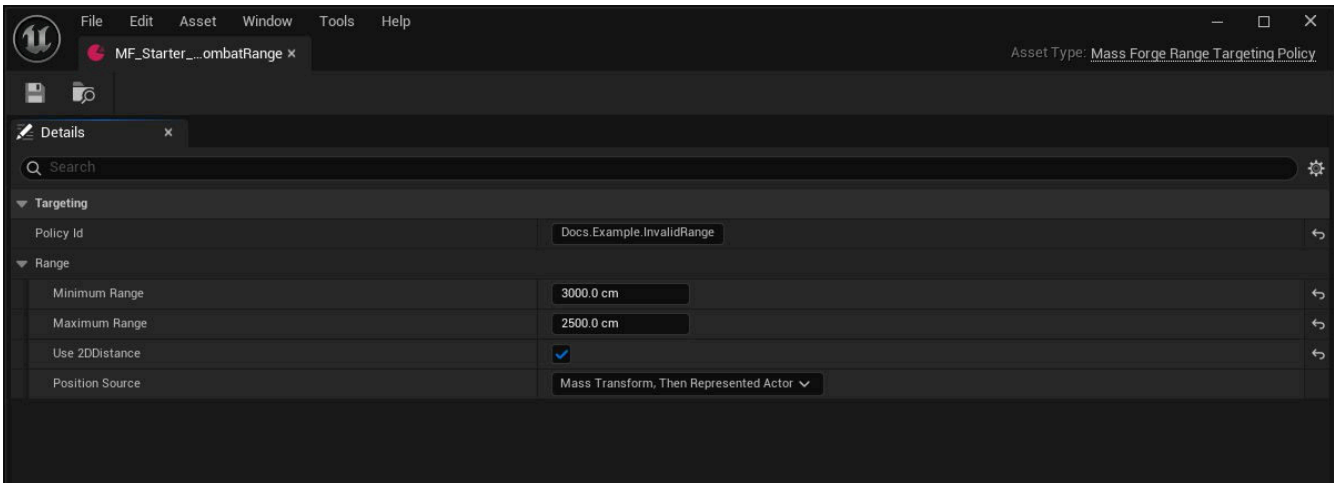
Findings sort with errors first, then warnings, then informational results. Search matches category, summary, details, asset name, and object path. **Errors only** hides warnings while preparing a blocking release pass.

- **Browse** selects the asset in the Content Browser.
- **Open** opens the asset editor.
- **Project Settings** opens the schema and persistence configuration.
- **Entity Config Wizard** opens the non-destructive starter/configuration workflow.

After correcting an asset, select **Scan Project** again. The report is a point-in-time result; it does not silently rescan while assets are being edited.



*This deliberately invalid disposable asset sets its minimum range above its maximum. The finding names the asset, states the violated invariant, and exposes **Browse** and **Open** instead of leaving the user to search manually.*



*Selecting **Open** routes directly to the affected Data Asset. Correct the ordered range, save, then rescan until the dashboard returns to the green release-ready state shown above. The invalid asset existed only in the disposable documentation host.*

Recommended release workflow

1. Save every changed Mass Forge asset.
2. Open **Mass Forge Project Health** and scan.
3. Resolve every error and deliberately review each capacity or version-window warning.
4. Run Unreal's project-wide Data Validation for non-Mass-Forge content and project-specific validators.
5. Run the packaged project's automated tests and shipping build.

The dashboard complements Unreal Data Validation; it does not replace validators owned by the host project. It specifically adds cross-asset stable-ID uniqueness, schema resolution, conditional Attribute ID checks, stable-slot capacity, and migration-graph coverage that a single asset cannot prove in isolation.

Scope and performance

The asset scan is intentionally limited to `/Game` and configured schema paths. Plugin example assets are not treated as project configuration unless the project references them. The scan may synchronously load discovered Mass Forge assets because it is a deliberate editor command; no equivalent loading occurs in runtime modules.

EDITOR AND DEBUGGING

Entity inspection guide

Mass Forge exposes a single presentation-neutral snapshot for building runtime debug widgets, editor tooling, logs, and automated diagnostics. It works for actorless Mass entities and represented entities, and it does not add a UObject to each entity.

Capturing an entity

Use `Capture Mass Forge Entity Inspection` with an `FMFEntityHandle`. For a represented entity, `Capture Mass Forge Actor Inspection` first resolves the Actor to its current generation-checked Mass handle.

The overall `Result` answers whether the entity could be inspected safely:

- `Success`: the entity was valid and Mass was idle when the core snapshot was captured.
- `Invalid World`: the supplied world context has no usable Mass world.
- `Invalid Entity`: the handle is unset, stale, or the Actor has no represented Mass entity.
- `Mass Is Processing`: retry from a safe game-thread point after Mass processing finishes.

An overall success does not imply that every optional Mass Forge trait exists. Check the section results separately:

- `Attribute Result`
- `Gameplay Tag Result`
- `Ability Result`
- `Persistent Effect Result`

For example, a valid entity without the Ability Trait still returns a useful snapshot of its attributes and tags while `Ability Result` reports `Missing Ability Fragment`.

Built-in sections

Attributes

`Attributes` contains every configured schema entry in stable domain and schema-slot order. Each row includes:

- stable domain/name identity;
- current value;
- display name and description;
- schema slot;
- optional minimum and maximum;
- whether the value currently sits at either bound.

This section uses the same finite-value and fragment validation as save snapshot capture. It never exposes unused fixed-array slots.

Counted Gameplay Tags

`Gameplay Tags` contains exact owned tags and their counts in lexical tag order. Parent-query expansion is intentionally not duplicated in the snapshot; use the existing hierarchical tag query when a UI needs to answer a specific parent-tag question.

`Get Owned Gameplay Tags` is also available independently of the inspector.

Abilities, active lifecycle, and cooldown groups

`Abilities` contains stable granted ability IDs, individual cooldown remaining time, compact stored charges, and scheduled charge-recovery time. Enumeration does not load definition assets. Because maximum charges and presentation text belong to the Ability Data Asset rather than the fragment, consumers that need those fields should resolve the ID through their own ability catalog.

`Active Cooldown Groups` contains only named groups whose remaining time is positive. Both arrays use lexical ID ordering. `Get Granted Abilities` and `Get Active Cooldown Groups` are available as standalone subsystem calls.

`Active Ability Lifecycle` reports `No Active Ability` while idle. During a cast/channel it includes the definition ID, target entity, phase, remaining phase time, completed channel-pulse count, and whether the atomic activation transaction has committed. This state is also available through `Get Active Mass Forge Ability Lifecycle` for entity handles and represented Actors.

Persistent effects

`Persistent Effects` contains the public state for each visible instance, ordered by monotonically assigned handle ID. It includes effect and stack identity, duration, next period, completed periods, granted/classification tags, stack count, strength, and persistent attribute-modifier count.

Adding project-specific “mind” data

Implement `Mass Forge Entity Inspection Provider` on a long-lived world object such as a manager Actor or Actor Component, then register it with `Register Mass Forge Inspection Provider`. Its execution result distinguishes success, an invalid world/provider, a missing interface, an already registered provider, and exhausted capacity. Unregistration separately reports a provider that was never registered; do not infer either operation’s outcome from the provider count.

The provider returns custom sections. A section has:

- a stable `Provider Id` namespace;
- a stable `Section Id` within that namespace;
- a localized display name;
- a sort order;
- label/value fields with stable keys and sort orders.

Example sections might be `AI.Intent`, `AI.Target`, `Faction.Relationship`, or a project-owned utility planner trace. Providers are held weakly, so registration does not extend their lifetime. Unregister them during teardown when practical; expired providers are ignored automatically.

Provider output is bounded to protect debug tools from accidental growth:

- 16 registered providers per world;

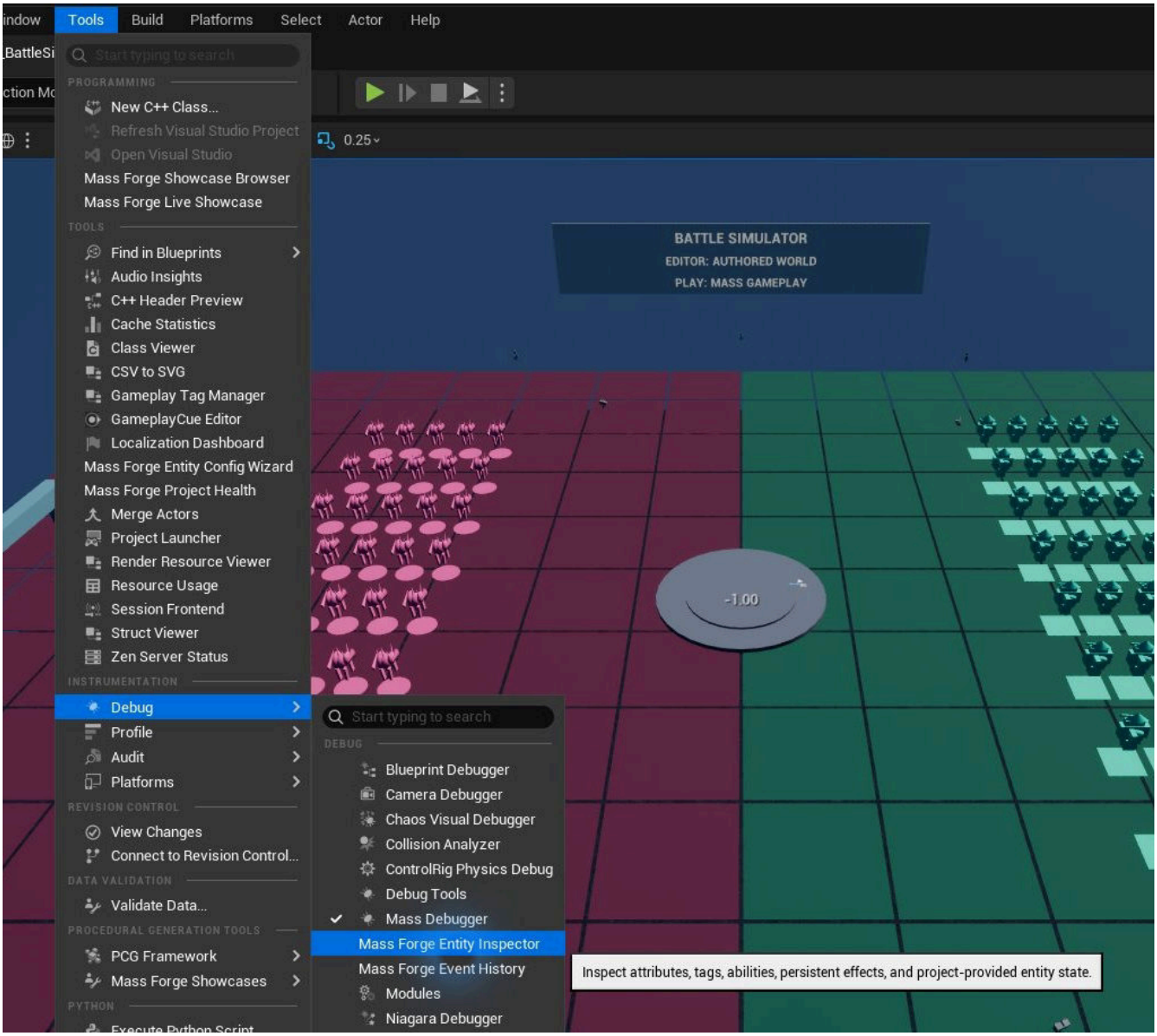
- 32 accepted custom sections per snapshot;
- 64 accepted fields per section.

Sections missing either stable ID, duplicate provider/section identities, missing or duplicate field keys, and excess entries are omitted. The snapshot reports truncation plus discarded section and field counts. Core Mass Forge sections are never displaced by custom output.

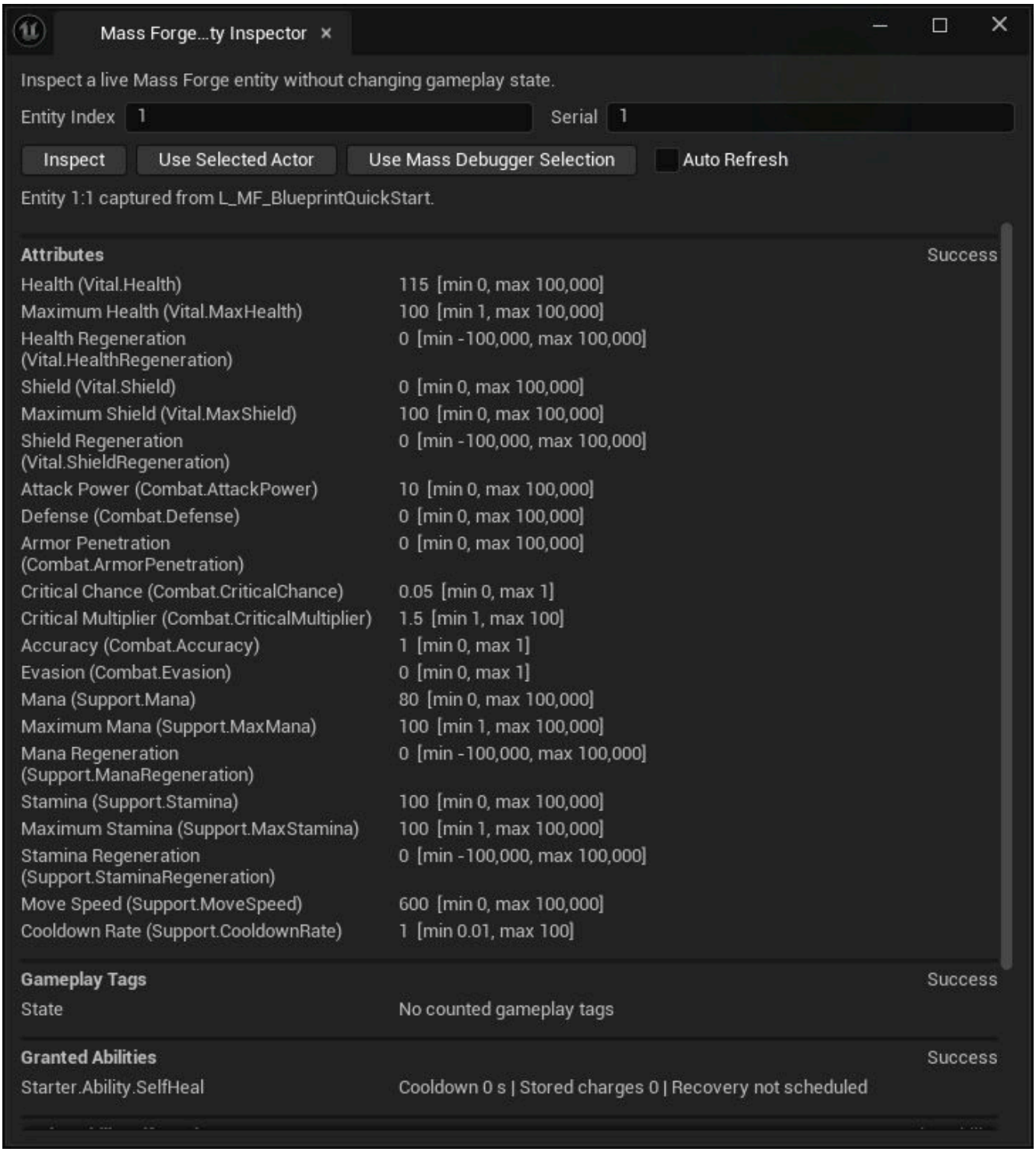
Custom providers run only after the core snapshot has been captured. Keep provider queries side-effect free. A provider mutation cannot retroactively change the core values already stored in that snapshot.

UI guidance

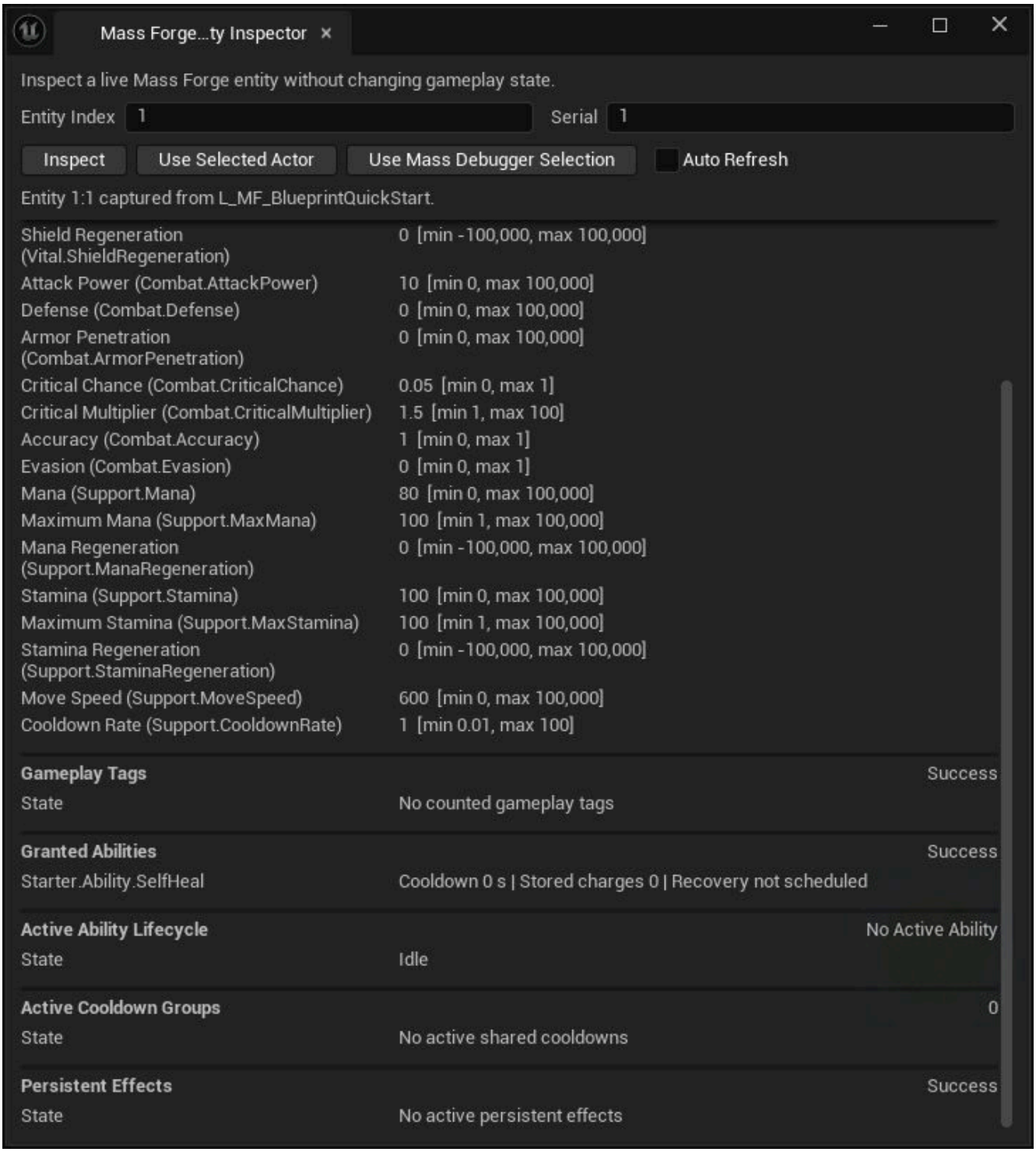
Mass Forge includes an editor-only **Mass Forge Entity Inspector** tab. Open it from **Tools** > **Debug** > **Mass Forge Entity Inspector**.



Open the inspector after entering Play in Editor so the tab can select the active PIE or game-preview world.



The header records the exact `Index:Serial` handle and source map. Attribute rows show the resolved identity, current value, and authored bounds; the ability section shows compact runtime ownership state.



An idle entity still reports each supported subsystem explicitly. During gameplay these sections expose counted tags, cooldowns, active lifecycle state, and persistent instances without adding debug fragments to the entity.

The tab supports three lookup paths:

- Select a represented Actor in the editor or during PIE and choose **Use Selected Actor**. The inspector resolves the Actor's generation-checked entity handle and uses that Actor's world.
- Select an actorless or represented entity in Unreal's Mass Debugger and choose **Use Mass Debugger Selection**. The inspector reads the full selected index/serial pair from the active world's entity manager.
- Enter an entity index and serial number, then choose **Inspect**. The inspector prefers the active PIE world, followed by game preview and editor worlds.

Enable **Auto Refresh** to update the chosen entity four times per second. The panel displays schema-enriched attributes, exact counted tags, granted ability runtime state, the active cast/channel and target, active shared cooldowns, persistent effects, and registered project-specific sections. Missing optional traits appear as section-level results instead of hiding the rest of a valid entity. Invalid or stale handles remain explicit failures.

The editor panel is a consumer of the public runtime snapshot, not a second source of truth. Its Slate and UnrealEd dependencies live in the separate `MassForgeEditor` module and are excluded from game targets. Projects can still build a compact UMG selected-entity panel, an in-world debug overlay, logs, or custom editor tools from the same Blueprint/C++ snapshot API.

Capture only on demand or at a modest refresh rate for selected entities. Do not inspect every entity every frame; simulation processors should continue using compiled attribute slots and fragments directly.

Event-history companion

The Entity Inspector remains a point-in-time view. Open **Tools > Debug > Mass Forge Event History** to record a bounded timeline, watch multiple entities, filter events, and reuse the selected Actor or Mass Debugger entity. The two tools consume the same runtime identities but keep current-state inspection separate from diagnostic history. See the gameplay event-history guide.

EDITOR AND DEBUGGING

Gameplay event history

Mass Forge provides one optional world-level event stream for debugging, UI prototypes, telemetry adapters, and future editor tooling. It normalizes the existing attribute, whole-entity state restore, Instant Effect, counted Gameplay Tag, Damage, Persistent Effect, and Ability delegates without replacing them. Existing subsystem delegates remain the authoritative integration points for gameplay code that needs a specific strongly typed payload.

Recording is disabled by default and adds no per-entity fragment or UObject. When enabled, records are retained in a bounded circular history. The oldest record is overwritten when the configured capacity is reached.

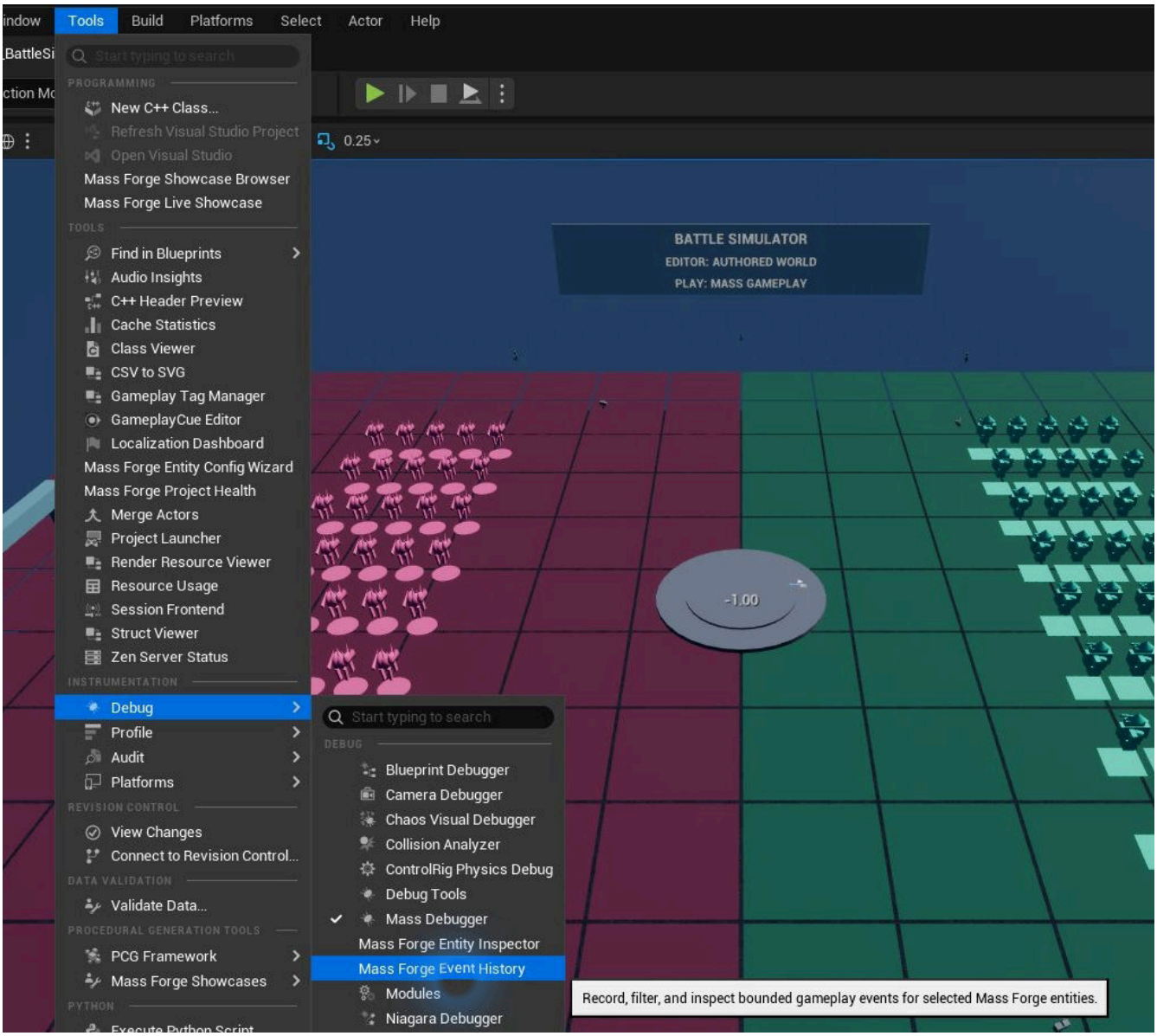
Blueprint setup

1. Get the **Mass Forge Gameplay Event Subsystem** for the active world.
2. Call **Set Gameplay Event Recording Enabled** with `Enabled = true`. Set `Clear Existing History` when starting a new diagnostic session.
3. Optionally call **Add Gameplay Event Watch** for up to 64 live entity handles. With no watched entities, every supported event is eligible. With one or more watches, an event is captured only when a watched entity is its source or target.
4. Bind **On Gameplay Event Recorded** for live presentation, or call **Get Gameplay Event History** with a filter and a positive maximum result count.
5. Disable recording when the diagnostic session ends. Use **Clear Gameplay Event History** and **Clear Gameplay Event Watches** when their state should also be reset.

The history capacity is **Project Settings > Mass Forge > Global Attribute Schemas > Debugging > Max Recorded Gameplay Events**. The supported range is 32-65,536 and the default is 2,048. Size it from measured diagnostic needs; each retained record contains copied identifiers and handles.

Editor event browser

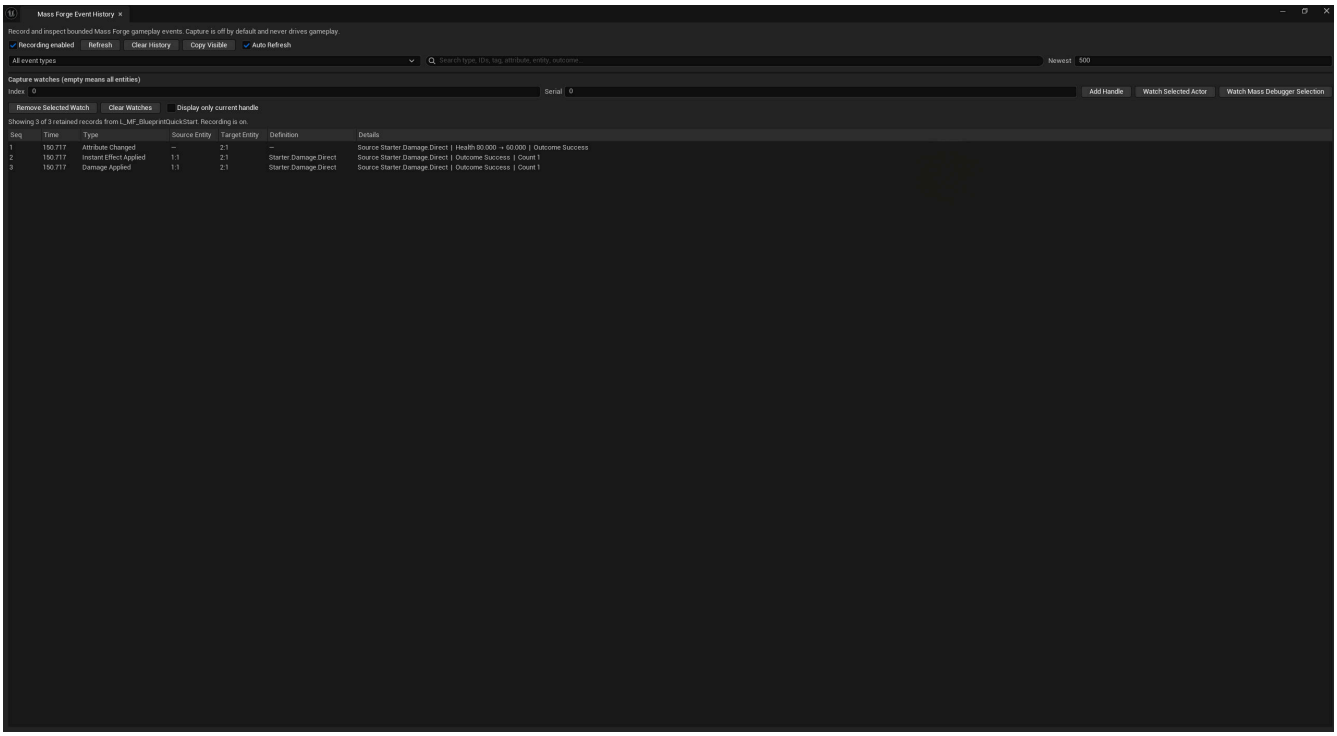
Open **Tools > Debug > Mass Forge Event History**. The editor-only browser exposes the runtime contract without requiring a debug Blueprint:



Open the tab after Play begins, then enable recording only for the diagnostic session that needs a timeline.

- enable or disable recording and clear retained history;
- auto-refresh or manually refresh the active PIE, game-preview, or editor world;
- select one event category, search across identities/tags/attributes/entities/outcomes, limit the newest returned records, optionally display only the current handle, and copy the visible rows as tab-separated diagnostic text;
- add a validated handle manually, resolve the selected represented Actor, or use the actorless/represented entity selected in Unreal's Mass Debugger;
- inspect and remove individual watches or clear the complete watch list;
- read sequence, world time, type, source/target handles, definition, request/attribute/tag/outcome, instance, and ordinal details in one structured timeline table.

The browser owns no gameplay state. Closing it does not disable recording automatically because Blueprint tooling or another editor view may share the same world subsystem. Recording and history are world-local and naturally reset when PIE ends. An empty watch list means capture all supported events; it is not the same as capturing none.



The screenshot is a live Blueprint Quick Start impact: one traveling projectile resolved `Starter.Damage.Direct`, changed Health from `80` to `60`, and retained source `1:1`, target `2:1`, and the successful enclosing Effect and Damage records. The ordered rows make it possible to distinguish visual launch, impact, transaction changes, and the final gameplay result.

Record contract

Every captured record receives a world-local monotonic `Sequence` and `World Time Seconds`. Records returned by a query are chronological. When more records match than `Max Results`, the newest matching records are returned while preserving chronological order.

Field	Meaning
`Type`	Stable normalized category.
`Request Id`	Positive only on terminal queued-request records; direct and lifecycle records use zero.
`Source Entity` / `Target Entity`	Generation-checked endpoints when the originating payload supplies them.
`Definition Id`	Effect, Damage Definition, or Ability identity.
`Source Id`	Weapon, system, snapshot, or project-defined logical source.
`Attribute`, `Previous Value`, `New Value`, `Was Clamped`	Attribute-change data when applicable.
`Gameplay Tag`, `Previous Count`, `New Count`	Counted Gameplay Tag data when applicable.
`Outcome`	Stable enum-value name for success, failure, disposition, or lifecycle end reason.
`Operation`	Stable enum-value name for a queued operation or lifecycle phase.

Field	Meaning
`Instance Id`	Persistent Effect handle ID or Ability lifecycle ID.
`Ordinal`	Persistent period number or Ability channel tick.
`Item Count`	Changed attributes/tags, restored durable-state items, stack count, or another type-specific bounded count. For a whole-entity restore it is the sum of attribute, exact counted-tag, ability-grant, and cooldown-group entries actually reported by the restore result.

Fields that do not apply to an event type remain unset. Attribute and tag changes are published before their enclosing successful Instant Effect record, matching the transaction publication contract. Failed atomic gameplay does not manufacture provisional change records.

The underlying typed results preserve the same diagnostic identity used by this stream. `FMFAttributeChangeResult` retains operation and logical source; `FMFInstantEffectResult` retains source entity, target generation, Effect ID, and logical source through pre-mutation failure; `FMFAbilityActivationResult` retains the submitted source and optional target; and `FMFPersistentEffectRequestCompletion` retains submitted Apply context or the active removal context captured at queue admission. This prevents a terminal failure from becoming less diagnosable than a success merely because no application event was emitted.

Field-coverage matrix

Event family	Guaranteed applicable identity and change fields
Attribute change / request	Target, attribute, old/new values, clamp state, logical source; queued completion also has request ID, operation, and outcome.
Attribute or entity-state restore	Target, logical source, request ID, outcome, and restored count including active Persistent Effects and an active Ability Lifecycle; successful constituent changes separately retain exact attribute/tag/ability details.
Instant Effect	Source, target, Effect ID, logical source, outcome, and changed-item count; queued completion also has request ID, including failures before mutation.
Counted Gameplay Tag	Target, tag, old/new counts, and outcome; queued completion also has request ID and operation. The tag API has no source parameter to invent.
Damage and death	Source, target, Damage ID, logical source, and outcome; queued/death-destruction records retain their request IDs.
Persistent Effect	Source context when supplied, target, Effect ID, handle, outcome/disposition, and stack or period count; queued completion also has request ID and operation, including failed Apply context. Period and removal records retain source identity even when recording starts after application.
Ability	Source, optional target, Ability ID, and outcome; lifecycle records add lifecycle ID/phase/tick, and queued completions add request ID and operation. Failed direct activation retains its attempted target.
Custom	The project supplies whichever generic fields describe its decision or mind breadcrumb; Mass Forge assigns sequence/time and sanitizes type/request ID.

The stream includes:

- direct changes plus terminal queued completions for attributes, attribute snapshots, whole-entity state snapshots, Instant Effects, and counted Gameplay Tags;
- Damage application, terminal Damage requests, kill, deactivation request, and destruction completion;
- Persistent Effect apply, period, remove, stack, and terminal queued-operation records;

- Ability grant, revoke, activation, commit, failure, request completion, lifecycle start, channel tick, and lifecycle end.

`Entity State Restore Completed` is emitted exactly once for every accepted queued whole-entity restore, including an execution-time failure such as a retired entity generation. It retains the positive request ID, exact target handle, logical restore source, stable result name, and restored-item total. That total includes the lifecycle as one item when format 4 restores one. A restore rejected before admission has no request ID and does not manufacture a terminal history record.

`After Sequence` supports incremental polling without reprocessing records already consumed. Filters may combine event types, either endpoint entity, definition ID, attribute, Gameplay Tag, and sequence. An unset filter field matches all values. `Invalid Limit` means `Max Results` was outside 1-65,536.

Project-defined mind and decision records

Use **Record Custom Gameplay Event** to place bounded project data in the same stream. This supports AI intent, target choice, state changes, or other “what is this entity thinking?” breadcrumbs without making Mass Forge own the project’s AI model. Fill only the relevant generic fields. Mass Forge forces `Type = Custom`, assigns sequence/time, and clears caller-supplied request correlation.

Custom records obey recording state, watch filtering, and capacity exactly like built-in records. For richer point-in-time AI state, use the Entity Inspection Provider interface; event history is the time-ordered breadcrumb view.

Boundaries and limitations

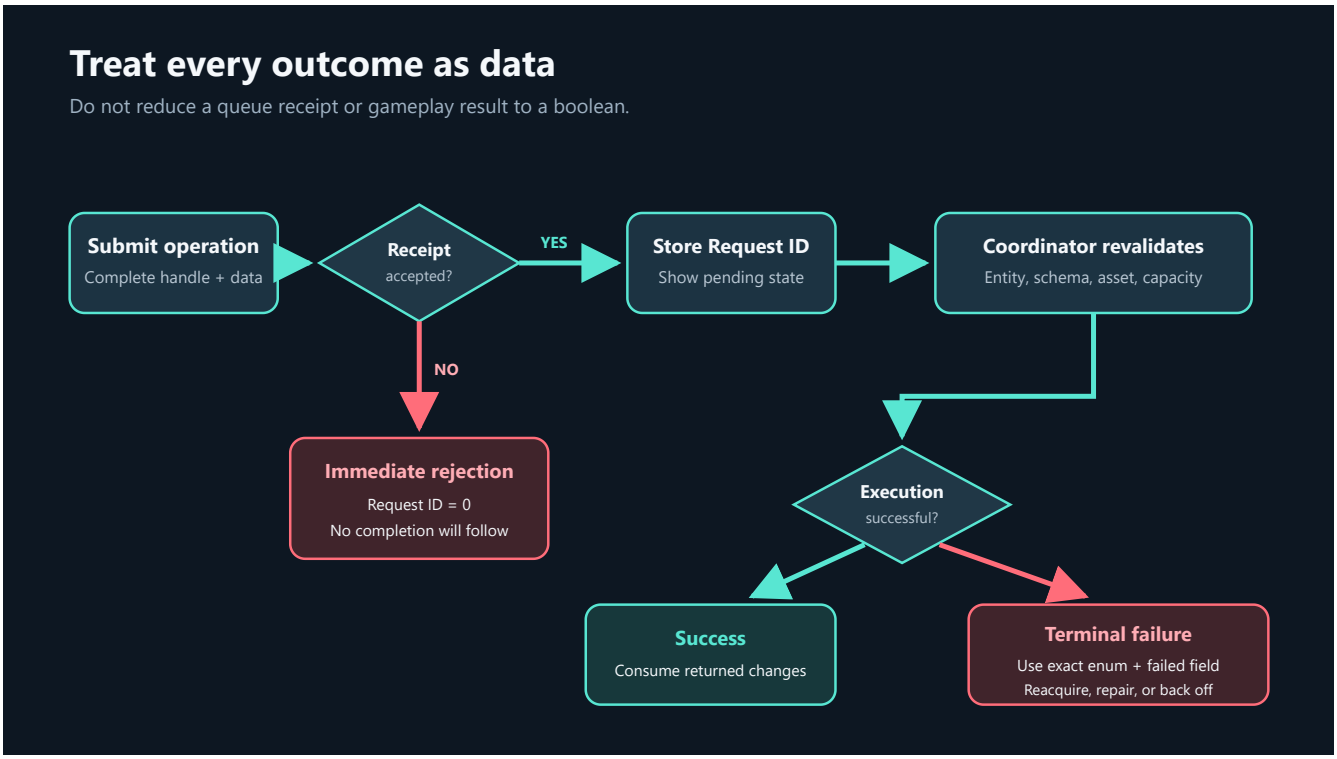
- This is diagnostic and presentation data, not gameplay authority, replication, prediction, rollback, or save-game state.
- Listening to `On Gameplay Event Recorded` must not be required for a gameplay transaction to succeed.
- Watch handles are generation-sensitive. Reacquire a handle after destruction, pooling, representation replacement, world travel, or PIE restart.
- Persistent Effect period results and the additive context-aware removal event carry source identity directly. Starting recording after an effect was applied does not erase source fields from its later period or removal records; no global active-effect scan is required when recording is enabled.
- High-volume compiled processor commands intentionally do not broadcast one UObject/delegate event per entity. Use their bounded completion mailbox and population benchmark instead of event history for million-entity throughput analysis.
- World replacement destroys the subsystem and its history. Copy/export any project telemetry before travel if it must survive.

The Entity Inspector complements this browser: use inspection for current state and Event History for the ordered changes that produced it.

EDITOR AND DEBUGGING

Error and result-code reference

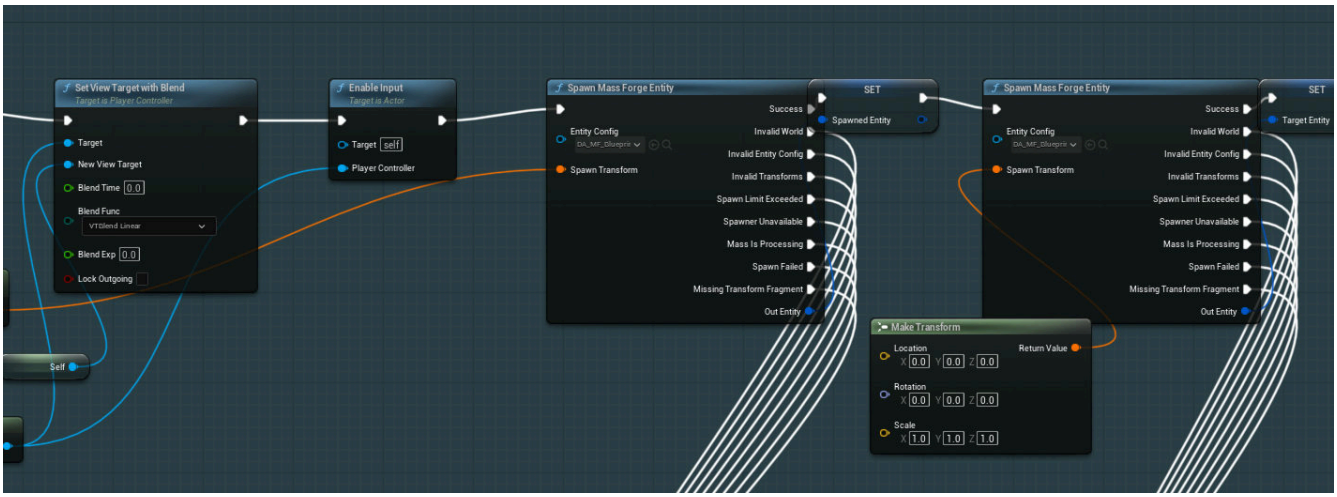
Mass Forge uses explicit enums and result structures so Blueprint and C++ callers can distinguish expected gameplay outcomes from setup errors. This reference covers every public result, status, disposition, and terminal-reason enum. Treat the enum as the primary branch condition; inspect the surrounding result structure for the failed attribute, tag, policy, effect, entity, or request ID.



A rejected receipt owns no queue slot and will not complete later. An accepted request still needs execution-time validation, so retain its Request ID and handle every terminal result instead of converting the workflow to a boolean.

Handling rules

- `Success`, `Allowed`, and accepted/terminal-success states are the only general success paths. A receipt is accepted only when its `Was Accepted` helper is true and its request ID is positive.
- `MassIsProcessing` means the direct operation was attempted during a Mass processing phase. Use the corresponding queued operation when one exists; do not retry in a tight Blueprint loop.
- `InvalidWorld` commonly means the world-context object is null, destroyed, or belongs to a world that is shutting down. Supply a live object from the intended game world.
- `InvalidEntity` and stale-representation outcomes are safe failures. Re-resolve represented Actors or reacquire actorless handles instead of modifying an old index/serial pair.
- `QueueFull` is bounded back-pressure, not silent loss. Reduce burst size, distribute work across frames, or raise the documented project setting after measuring memory and frame cost.
- Nested results matter. For example, `EntityApplicationFailed`, `GameplayTagFailure`, `PersistentEffectFailed`, and `TransactionFailed` identify the failed layer; inspect the nested result payload for the exact cause.



Blueprint exposes expected spawn outcomes as named execution branches instead of hiding them behind a boolean. For compound gameplay operations, branch on the top-level enum first, then split the returned result struct to inspect the nested entity, attribute, tag, policy, effect, or request-ID details described below.

`EMFProjectileLaunchResult`

Returned immediately when a project asks the world projectile subsystem to create one actorless Mass projectile.

Value	Cause or meaning	Recommended response
`Success`	A projectile entity was created, initialized, and assigned a positive projectile ID.	Store the receipt if correlation is needed; render snapshots or wait for the resolved event.
`InvalidWorld`	The subsystem has no live world, or required Mass spawning/entity services are unavailable.	Launch from a live gameplay world after subsystem initialization and stop during teardown.
`InvalidConfig`	The request has no Mass Entity Config.	Assign an Entity Config containing Mass Forge Projectile Trait.
`InvalidRequest`	A location contains NaN, or speed, arc, release delay, or maximum lifetime is non-finite/out of range.	Sanitize authored/runtime values and keep speed/lifetime positive and delays/arcs non-negative.
`MassIsProcessing`	Immediate entity creation was requested while the Mass entity manager owned processing.	Queue the project-side launch until the next safe world tick; never retry inside the processor loop.
`SpawnFailed`	The Mass spawner did not return exactly one projectile entity.	Validate the Entity Config/archetype and world capacity, then retry only from a later safe tick.
`MissingProjectileFragment`	The spawned config did not contain `FMF_ProjectileFragment`.	Add Mass Forge Projectile Trait to the exact Entity Config supplied in the request.

`EMFProjectileResolveReason`

Published when an accepted Mass projectile leaves simulation.

Value	Cause or meaning	Recommended response
`Impact`	The point-to-point/arc flight reached its authored target and optional damage was attempted.	Use the resolved location/cue and inspect the nested Damage result before applying project-specific follow-up behavior.

Value	Cause or meaning	Recommended response
`LifetimeExpired`	Maximum lifetime elapsed before travel reached the target.	Treat it as a miss/cleanup result; increase lifetime or reduce travel distance only when the authored design requires it.

`EMFEntityResolveResult`

Returned when converting a represented Actor into a generation-checked Mass Forge entity handle.

Value	Cause or meaning	Recommended response
`Success`	The Actor has a current, valid Mass representation.	Use the returned handle; reacquire it after representation or lifecycle changes.
`InvalidWorld`	The world context did not resolve to a live world.	Pass a live Actor, component, subsystem, or other object from the intended world.
`InvalidActor`	The Actor is null, pending destruction, or otherwise invalid.	Stop the operation and obtain a live Actor.
`DifferentWorld`	The Actor belongs to a different world than the selected subsystem.	Use a context object from the Actor's world and never carry handles across worlds.
`RepresentationUnavailable`	The world's Mass Actor representation subsystem is unavailable.	Enable/configure the required Mass representation support or use an actorless handle source.
`NotRepresented`	The Actor is valid but is not currently mapped to a Mass entity.	Use an ordinary-Actor adapter or wait for representation before resolving it.
`StaleRepresentation`	The Actor mapping returned an entity index/serial that is no longer valid.	Recreate or refresh representation; do not reuse the cleared output handle.

`EMFEntitySpawnResult`

Returned by the Blueprint-first single and bounded-batch entity spawn nodes.

Value	Cause or meaning	Recommended response
`Success`	Every requested entity was created, received its requested transform, and returned a generation-safe handle.	Store handles only for the lifetime needed and revalidate them after destruction, pooling, or world changes.
`InvalidWorld`	The supplied context did not resolve to a live gameplay world.	Call from an object owned by the intended game world.
`InvalidEntityConfig`	The Entity Config is null or invalid.	Assign a saved Mass Entity Config, preferably one generated by the Mass Forge wizard.
`InvalidTransforms`	The transform array is empty or contains a non-finite transform.	Supply at least one finite world transform.
`SpawnLimitExceeded`	One Blueprint batch requested more than 10,000 returned handles.	Split ordinary Blueprint work across bounded batches or use a measured C++ population path for larger allocations.
`SpawnerUnavailable`	The world does not provide the required Mass entity/spawner subsystems.	Enable the Mass dependencies and call after world subsystem initialization.

Value	Cause or meaning	Recommended response
`MassIsProcessing`	Immediate entity creation would overlap Mass processing.	Schedule the spawn from a later safe game-thread boundary; do not retry inside a processor loop.
`SpawnFailed`	The Mass spawner did not create the complete requested batch.	Validate the config/archetype and world capacity; no partial output is retained.
`MissingTransformFragment`	The spawned archetype lacks Unreal's standard transform fragment, so placement could not be guaranteed.	Add Mass Forge Transform Trait or another trait that supplies `FTransformFragment`; the failed batch is destroyed.

`EMFEntityDestroyResult`

Returned by the Blueprint-first generation-safe destroy node.

Value	Cause or meaning	Recommended response
`Success`	The exact live entity generation was submitted to the Mass spawner for destruction.	Clear project-held references and reacquire any replacement entity.
`InvalidWorld`	The supplied context did not resolve to a live gameplay world.	Stop during teardown or supply a live context from the correct world.
`InvalidEntity`	The handle is unset, stale, destroyed, or belongs to another world.	Treat destruction as already complete or reacquire the intended entity; never reuse only the index.
`SpawnerUnavailable`	The Mass entity/spawner services are unavailable.	Call after world subsystem initialization and before teardown.
`MassIsProcessing`	Immediate destruction would overlap Mass processing.	Use an existing queued death/destruction path or schedule the direct destroy at the next safe boundary.

`EMFAttributeAccessResult`

Used by attribute reads, writes, instant-effect transactions, and queued attribute/effect receipts.

Value	Cause or meaning	Recommended response
`Success`	The requested read or transaction completed.	Consume the values and any clamp/operation details in the result.
`InvalidWorld`	No live Mass Forge world subsystem was available.	Supply a valid world context and avoid calls during world teardown.
`InvalidEntity`	The handle is unset, stale, or belongs to another world.	Reacquire the entity handle and verify its lifecycle.
`MassIsProcessing`	Direct fragment access would overlap Mass processing.	Submit the matching queued request and handle its completion event.
`MissingSchema`	The requested attribute domain has no configured runtime schema.	Assign the domain schema in Mass Forge project settings and rebuild the Entity Config.
`UnknownAttribute`	The stable attribute ID is absent from its configured schema.	Select a current schema entry or migrate the old ID deliberately.

Value	Cause or meaning	Recommended response
`MissingFragment`	The entity archetype does not contain the required attribute fragment.	Add the Mass Forge Attributes Trait and the needed domain to the Entity Config.
`InvalidMagnitude`	A supplied or calculated float is NaN, infinite, or otherwise invalid.	Sanitize gameplay inputs and validate every custom calculation before submission.
`MissingMagnitudeParameter`	An effect requested a named context scalar that was not supplied.	Add the parameter to the effect context or remove/correct the authored parameter name.
`InvalidEffect`	The Instant Effect asset is null, invalid, or changed incompatibly.	Repair the asset in its authoring overview and rerun Project Health.
`GameplayTagFailure`	The atomic effect's tag portion could not validate or commit.	Inspect the effect's failed tag/result, trait capacity, and tag counts.
`QueueFull`	The bounded attribute or instant-effect queue has no free request slot.	Throttle the producer or raise the matching pending-request limit after profiling.
`MagnitudeCalculationFailed`	An assigned custom magnitude calculation rejected, was invalid/unloadable, recursed, or returned a non-finite value.	Inspect `FailedModifierIndex` and `MagnitudeCalculationResponse`; repair the stable calculation asset or its inputs before retrying.

`EMFEffectMagnitudeCalculationResult`

Returned by reusable Blueprint/C++ effect-magnitude calculation assets.

Value	Cause or meaning	Recommended response
`Success`	The calculation returned a finite final magnitude.	Use the returned magnitude; Mass Forge continues staging the enclosing transaction.
`Rejected`	Project logic intentionally refused to calculate for this context.	Inspect the stable calculation ID/reason and treat expected gameplay denial separately from setup failure.
`InvalidWorld`	The query has no live world or crosses world ownership.	Supply context from the same active world as the transaction.
`InvalidConfiguration`	The calculation is missing, unloadable, has no stable ID, or uses the unimplemented fail-closed base class.	Repair/implement the asset and rerun Data Validation and Project Health.
`RecursiveEvaluation`	Calculation code started another custom magnitude evaluation before returning.	Keep calculations side-effect-free; defer gameplay operations until the outer transaction completes.
`NonFiniteMagnitude`	The input, configured math, or returned magnitude is NaN or infinite.	Sanitize inputs and guard division/overflow before returning Success.

`EMFAttributeSnapshotResult`

Used when capturing or atomically restoring attribute snapshots.

Value	Cause or meaning	Recommended response
`Success`	Capture or restore completed atomically.	Persist the snapshot or continue from the applied values.

Value	Cause or meaning	Recommended response
`InvalidWorld`	No live attribute subsystem was available.	Perform save/load against the correct active game world.
`InvalidEntity`	The target handle is unset or stale.	Reacquire the entity before capture or restore.
`MassIsProcessing`	Direct snapshot access would overlap Mass processing.	Capture on a safe lifecycle tick; for restore, use **Queue Mass Forge Attribute Snapshot Restore**.
`MissingSchema`	A required domain schema is not configured.	Restore the schema configuration before handling snapshots.
`MissingFragment`	The entity lacks a fragment required by a snapshot attribute.	Use a compatible Entity Config or migrate into an appropriate archetype.
`EmptySnapshot`	The snapshot contains no values.	Treat it as absent save data or populate it through a successful capture.
`UnsupportedVersion`	`FormatVersion` is not supported by this plugin build.	Compare `SnapshotFormatVersion` with `ExpectedFormatVersion`; upgrade through a supported plugin path or reject the save with a user-facing message.
`SchemaVersionMismatch`	The project schema generation differs from the snapshot.	Compare `SnapshotSchemaVersion` with `ExpectedSchemaVersion`, then apply each required migration asset in order before restore.
`DuplicateAttribute`	The snapshot contains the same stable attribute ID more than once.	Repair the producer or migration; never choose one duplicate implicitly.
`UnknownAttribute`	A saved stable ID is not present in current schemas.	Add an explicit rename/remove migration rule.
`InvalidValue`	A saved value is NaN or infinite.	Reject or repair the corrupted save; do not inject the value into Mass state.
`QueueFull`	The bounded deferred snapshot-restore queue has no pending capacity.	Defer or shed the load; raise **Max Pending Queued Snapshot Restores** only after measuring copied-payload memory and frame cost.

`EMFAttributeSnapshotMigrationResultCode`

Returned by one explicit snapshot migration step.

Value	Cause or meaning	Recommended response
`Success`	The migration produced a complete target-version snapshot.	Continue the migration chain or restore the final snapshot.
`InvalidMigration`	The migration asset is null or fails its own validation.	Correct its ID, adjacent versions, rules, and duplicate mappings.
`UnsupportedSnapshotFormat`	The input binary format is not understood.	Load it with a compatible plugin version or provide an external format converter.
`SourceVersionMismatch`	The input schema version does not equal the migration's source version.	Select the next adjacent migration in the chain.
`EmptySnapshot`	The migration input has no attribute values.	Treat the save as missing/corrupt or start from a valid captured snapshot.
`InvalidAttribute`	A source or rename target ID in the snapshot/rules is invalid.	Repair the stable IDs and validate the migration asset.

Value	Cause or meaning	Recommended response
`InvalidValue`	An input value is NaN or infinite.	Reject or repair the corrupted save before migration.
`DuplicateAttribute`	Migration would produce duplicate target identities.	Remove conflicting rules or resolve converging renames explicitly.

`EMFEntityStateSnapshotResult`

Used when capturing, validating, directly restoring, or queuing the versioned durable entity-state snapshot.

Value	Cause or meaning	Recommended response
`Success`	Capture, validation, or restore completed atomically.	Persist the snapshot or continue from the fully restored state.
`InvalidWorld`	No live Mass Forge world or required subsystem was available.	Use a live object from the intended gameplay world and avoid world teardown.
`InvalidEntity`	The entity handle is unset, stale, or belongs to another world.	Reacquire the complete entity index/serial before capture or restore.
`MassIsProcessing`	A direct operation would read or mutate fragments during Mass processing.	Capture at a safe lifecycle point or use **Queue Mass Forge Entity State Restore** for restore.
`UnsupportedVersion`	The snapshot format is not understood by this plugin build.	Load it with a compatible plugin or migrate it through an explicitly supported format path.
`SchemaVersionMismatch`	The nested attribute schema generation differs from the project schema.	Apply the required ordered attribute migrations before restoring the complete snapshot.
`InvalidAttributeSnapshot`	The nested attribute payload failed format, schema, identity, value, capacity, or fragment validation.	Inspect `AttributeResult`, repair or migrate the entire payload, and retry without partial application.
`MissingGameplayTagFragment`	The snapshot includes counted tags but the target archetype has no tag fragment.	Use a compatible Entity Config with the Gameplay Tag Trait.
`InvalidGameplayTag`	A saved Gameplay Tag is empty or not registered.	Repair the save producer or restore the required project tag registration.
`InvalidGameplayTagCount`	A saved exact tag count is outside the supported positive compact range.	Reject or repair the corrupted count before restore.
`DuplicateGameplayTag`	The same exact Gameplay Tag occurs more than once.	Merge it into one exact count; do not choose a duplicate implicitly.
`GameplayTagCapacityExceeded`	Saved exact tags exceed the fixed fragment capacity.	Migrate to a compatible bounded payload or redesign the target state budget.
`MissingAbilityFragment`	The snapshot includes ability state but the target archetype has no ability fragment.	Use a compatible Entity Config with the Ability Trait.
`InvalidAbilityId`	An ability grant has an empty stable ID or ability data appears while abilities are excluded.	Repair the snapshot producer and resolve every grant through a current stable ability ID.
`DuplicateAbilityId`	The same stable Ability ID occurs more than once.	Keep one authoritative grant record per ability.

Value	Cause or meaning	Recommended response
`AbilityCapacityExceeded`	Saved grants exceed the target's fixed ability capacity.	Migrate or trim deliberately before restore; never accept an implicit partial grant list.
`InvalidAbilityTime`	An individual cooldown remaining value is negative, NaN, or infinite.	Reject or repair the save and persist remaining gameplay seconds only.
`InvalidChargeState`	Stored charge recovery has an invalid time or inconsistent scheduled flag.	Repair the charge record so unscheduled recovery is zero and every time is finite/non-negative.
`InvalidCooldownGroup`	A shared/global cooldown has an empty ID or non-positive/non-finite remaining time.	Remove expired groups and repair the remaining duration or stable group identity.
`DuplicateCooldownGroup`	The same shared/global cooldown ID occurs more than once.	Merge it into one authoritative remaining duration.
`CooldownGroupCapacityExceeded`	Saved active groups exceed the target's fixed group capacity.	Migrate to a compatible bounded payload; do not partially restore groups.
`ActiveAbilityLifecycle`	A format-1/2/3 restore targeted an entity with an active cast/channel that the legacy payload cannot describe.	Reach an idle ability boundary or restore a compatible format-4 snapshot; legacy restore never erases the lifecycle implicitly.
`ActivePersistentEffects`	A format-1/2 restore targeted an entity with active Persistent Effects that the legacy payload cannot describe.	End those effects deliberately or restore a compatible format-3 snapshot; legacy restore never erases them implicitly.
`TransactionInProgress`	A re-entrant direct restore was requested while another entity-state restore was publishing its transaction.	Defer the new operation through the bounded queue and handle its terminal completion.
`QueueFull`	The bounded entity-state restore queue has no pending capacity.	Throttle or defer the producer; raise **Max Pending Queued Entity State Restores** only after profiling memory and frame cost.
`InvalidRegenerationState`	Regeneration timing contains an invalid/duplicate target, non-finite or out-of-range remainder, changed schema identity, excess rules, or format-1-only forbidden fields.	Reject or explicitly migrate the payload; keep continuous-rule remainder at zero and interval remainder below its tick interval.
`MissingRegenerationFragment`	A format-2-or-newer snapshot includes regeneration timing but the destination archetype has no regeneration fragment.	Restore into a compatible Entity Config with the Regeneration Trait.
`RegenerationProfileMismatch`	The destination's unique regeneration target set or tick intervals differ from the format-2-or-newer snapshot.	Use the matching profile or explicitly migrate/reset timing before restore; Mass Forge will not remap a phase by array position.
`InvalidPersistentEffectState`	A format-3-or-newer effect section has inconsistent duration/period/backlog state, invalid portable context/tags/modifiers, missing reversible bases, or effective attributes that do not match those bases and contributions.	Reject or explicitly migrate the save; do not patch runtime fragment slots or accept partial effect ownership.
`MissingPersistentAttributeModifierFragment`	The destination or captured active-effect archetype lacks the companion fragment required for reversible contributions.	Use the Persistent Effects Trait on a compatible Entity Config and repair any custom archetype construction.
`MissingPersistentEffectDefinition`	A saved persistent or periodic soft asset reference is absent or cannot be loaded.	Restore the referenced asset at its stable path or explicitly migrate/remove that saved effect before restore.

Value	Cause or meaning	Recommended response
`PersistentEffectDefinitionMismatch`	A loaded persistent or periodic asset's stable Effect ID differs from the ID guarded by the save.	Restore the identity-compatible asset or run an explicit project SaveGame migration; never reinterpret it silently.
`PersistentEffectCapacityExceeded`	The save exceeds 16 active effects or 24 reversibly modified attributes.	Migrate or reduce the payload deliberately; Mass Forge refuses partial restoration.
`InvalidAbilityLifecycleState`	The format-4 lifecycle section has inconsistent include flags, grants, phase, timing, commit state, pulse count/backlog, target mode, or forbidden legacy-format data.	Reject or explicitly migrate the complete save; never infer missing lifecycle state or patch the payload partially.
`MissingAbilityLifecycleDefinition`	The saved active lifecycle's soft Ability reference is empty, missing, or cannot be loaded.	Restore the referenced Ability asset at its stable path or explicitly migrate/remove that lifecycle before restore.
`AbilityLifecycleDefinitionMismatch`	The loaded Ability differs from the saved stable ID, Save Compatibility Version, execution type, cast time, channel duration, or channel period.	Restore a behavior-compatible asset or run an explicit project SaveGame migration; do not continue old timing under changed authoring.
`MissingAbilityLifecycleTargetBinding`	An externally targeted lifecycle was captured without a stable relationship key or restored without its runtime target binding.	Store a project-owned stable target key during capture, recreate the target, and supply the matching key plus new entity handle during restore.
`AbilityLifecycleTargetReferenceMismatch`	The supplied runtime target binding's project-owned key does not equal the key saved with the lifecycle.	Resolve the exact saved relationship and resubmit with the identical key; do not guess by Actor, position, or array order.
`InvalidAbilityLifecycleTarget`	The external target binding is unset, stale, not a live full-generation entity, or collapses the external relationship onto the restored source.	Recreate and resolve the intended distinct target entity, then retry with its current handle.

`EMFEntityStateLifecycleTargetMode`

Describes the portable target relationship stored for one format-4 active Ability Lifecycle.

Value	Cause or meaning	Recommended response
`None`	The active cast/channel has no target and stores no target reference.	Use the ordinary capture and restore nodes; do not invent a binding.
`Self`	The active cast/channel targeted the saved entity itself.	Use the ordinary restore path; Mass Forge rebinds it to the restored destination entity.
`External`	The active cast/channel targeted another entity, represented only by a project-owned stable relationship key in the save.	Recreate the target and supply the matching key plus its current full-generation handle to the lifecycle-target restore variant.

`EMFGameplayTagResult`

Used by counted tag mutations and queries.

Value	Cause or meaning	Recommended response
`Success`	The tag operation or query completed.	Use the exact count and hierarchical-match fields as appropriate.
`InvalidWorld`	No live Gameplay Tag subsystem was available.	Supply a valid world context.
`InvalidEntity`	The entity handle is unset or stale.	Reacquire the entity handle.
`MissingFragment`	The entity lacks the Mass Forge Gameplay Tag fragment.	Add the Gameplay Tag Trait to its Entity Config.
`InvalidTag`	The Gameplay Tag is empty or not registered.	Select a registered project tag.
`InvalidCount`	The requested add/remove count is outside the supported positive range.	Pass a positive bounded count.
`TagNotFound`	A remove/query expected an exact owned tag that is absent.	Treat absence as gameplay state or correct the requested tag.
`InsufficientCount`	Removal exceeds the exact owned count.	Clamp the request deliberately or correct the producer.
`CountOverflow`	Adding would exceed the compact counter's supported maximum.	Reduce stacking or redesign the counter semantics.
`NoFreeTagSlot`	The entity's bounded tag fragment is full.	Remove unused tags or redesign the archetype/state budget.
`MassIsProcessing`	Direct tag access would overlap Mass processing.	Use the queued Add/Remove node for one raw count change, or a queued Instant Effect for an atomic multi-change transaction.
`QueueFull`	The bounded counted-tag queue has no pending capacity.	Defer or shed the burst; tune queue limits only after profiling.
`InvalidOperation`	Native code supplied an out-of-range tag-operation enum value.	Pass only Add or Remove; treat other values as corrupted input.

`EMFEffectDispatchResult`

Returned by Entity/Actor cross-ownership Instant Effect dispatch.

Value	Cause or meaning	Recommended response
`Success`	Exactly one supported target route accepted the effect.	Inspect `Route`, resolved endpoints, and the nested application result.
`InvalidWorld`	No live world/subsystem could be resolved.	Pass a context object from the intended game world.
`InvalidSource`	The explicit source Actor/entity is invalid or cross-world.	Correct or omit the optional source as the node contract allows.
`InvalidTarget`	The target Actor/entity is null, stale, or cross-world.	Reacquire a live target.
`InvalidEffect`	The Instant Effect asset is null or invalid.	Repair the effect asset and validate its dependencies.
`UnsupportedTarget`	The Actor is neither Mass-represented nor an effect receiver.	Add the receiver interface/component or use a supported Mass entity target.

Value	Cause or meaning	Recommended response
`MultipleReceivers`	More than one receiver component would make routing ambiguous.	Keep exactly one receiver component or implement the Actor interface directly.
`EntityApplicationFailed`	The Mass route was selected but its atomic effect transaction failed.	Inspect `EntityResult` and `EntityApplication` for the precise failure.
`ReceiverRejected`	The selected ordinary-Actor receiver declined the effect.	Inspect project receiver logic; retry only when its rejection condition changes.

`EMFPersistentEffectResult`

Used by persistent-effect application, removal, query, stacking, and periodic execution.

Value	Cause or meaning	Recommended response
`Success`	The requested persistent-effect operation completed.	Use the handle, stack disposition, and timing data returned.
`InvalidWorld`	No live persistent-effect subsystem was available.	Supply a valid world context.
`InvalidEntity`	The target handle is unset or stale.	Reacquire the entity.
`MassIsProcessing`	Direct fragment access would overlap Mass processing.	Schedule the operation from a safe phase.
`InvalidEffect`	The Persistent Effect asset is null or fails validation.	Repair its ID, duration, stacking, modifiers, tags, and dependencies.
`MissingFragment`	The entity lacks persistent-effect instance storage.	Add the Persistent Effects Trait.
`MissingAttributeModifierFragment`	Reversible attribute modifiers require storage absent from the archetype.	Enable persistent attribute modifiers in the Entity Config.
`MissingAttributeFragment`	An authored modifier or pulse targets a missing attribute domain fragment.	Add the Attributes Trait/domain or change the effect.
`MissingAttributeSchema`	A required attribute domain schema is not configured.	Configure the schema in project settings.
`UnknownAttribute`	An authored stable attribute ID no longer resolves.	Repair the asset or migrate the attribute identity.
`MissingMagnitudeParameter`	A context-backed magnitude lacks its named scalar.	Supply the parameter or correct the effect asset.
`InvalidModifierMagnitude`	A resolved persistent modifier is NaN or infinite.	Correct authored values, context inputs, or custom calculations.
`AttributeCaptureFailed`	Snapshot-at-application could not read a required source/target attribute.	Inspect the failed binding, fragments, entity, and processing phase.
`AttributeCaptureCapacityExceeded`	The effect needs more captured values than its bounded instance storage permits.	Reduce unique captures or split the effect.
`ModifierCapacityExceeded`	Applying the effect would overflow bounded persistent modifier storage.	Remove/split modifiers or budget more storage in a future archetype design.
`NoFreeEffectSlot`	The entity has no free persistent-effect instance slot.	Remove/expire effects or reduce simultaneous effect variety.

Value	Cause or meaning	Recommended response
`InvalidHandle`	A remove/query handle is unset, malformed, or for another entity.	Retain the exact returned handle and validate ownership before use.
`NotFound`	The handle was once plausible but no active instance matches it.	Treat it as already removed/expired and discard the handle.
`InitialApplicationFailed`	The authored apply-on-start periodic/instant payload failed atomically.	Inspect the nested Instant Effect inputs and target configuration.
`StackLimitReached`	The selected stacking rule cannot add another stack.	Treat this as expected gameplay or choose a refresh/replace policy.
`StrongerOrEqualEffectActive`	Strongest-wins rejected a weaker or equal incoming effect.	Keep the current effect or submit a genuinely stronger value.
`StackOperationInProgress`	Re-entrant listener code attempted to alter the same stack transaction.	Defer the nested operation and avoid recursive mutation.
`IncompatibleStack`	An existing stable stack identity has incompatible authored/runtime structure.	Keep stack definitions structurally consistent and use a new ID for a different contract.
`MissingGameplayTagFragment`	Effect tag gates/grants require tag storage absent from the entity.	Add the Gameplay Tag Trait.
`MissingRequiredTag`	The target lacks an authored prerequisite tag.	Treat as a gameplay gate or grant the prerequisite first.
`BlockedByTag`	The target owns an authored immunity/blocking tag.	Respect the immunity or remove it through an explicit cleanse/dispel flow.
`GameplayTagFailure`	Lifecycle tag validation or atomic tag mutation failed.	Inspect tag validity, counts, capacity, and the entity's tag fragment.
`QueueFull`	The bounded persistent Apply/Remove queue has no pending capacity.	Defer or shed the burst; tune limits only after measuring queue pressure.
`MagnitudeCalculationFailed`	A persistent attribute contribution's custom calculation failed before the instance committed.	Inspect the failed modifier and nested calculation response; the effect and all lifecycle changes remain unapplied.

`EMFPersistentEffectStackDisposition`

Describes what a successful persistent-effect application did to stack state.

Value	Cause or meaning	Recommended response
`AddedNewInstance`	A new independent or first stack instance was created.	Track the returned handle when later removal/query is needed.
`RefreshedExisting`	Existing duration/timing was refreshed without creating a new instance.	Refresh presentation timing for the existing handle.
`ExtendedExisting`	Existing duration was increased by the stacking rule.	Update duration UI from the returned live state.
`ReplacedExisting`	The previous compatible instance was removed and replaced.	Stop old presentation and bind to the returned replacement handle.
`ReplacedWithStronger`	Strongest-wins accepted the incoming stronger instance.	Present the upgraded magnitude and replacement lifecycle.

`EMFPersistentEffectRemovalReason`

Published when a persistent effect leaves active state.

Value	Cause or meaning	Recommended response
`Manual`	A caller explicitly removed or dispelled the instance.	End presentation normally and record the manual source if needed.
`Expired`	Finite duration reached zero.	End duration UI/VFX and release the handle.
`Replaced`	A stacking rule replaced the instance.	Transition presentation to the replacement instance.
`EntityInvalidated`	The owning entity was destroyed or became invalid.	Drop all project-side references without targeting the old handle.
`Cancelled`	A classification cleanse/cancel or lifecycle rollback removed it.	Use project context/events to explain the cancellation when relevant.

`EMFDamageResult`

Used by immediate damage applications, queued receipts, and queued completions.

Value	Cause or meaning	Recommended response
`Success`	Damage calculated and committed atomically.	Use the complete calculation, shield/health, critical, death, and context fields.
`InvalidWorld`	No live damage subsystem was available.	Supply a valid world context.
`MassIsProcessing`	Immediate damage would overlap Mass processing.	Use `Queue Mass Forge Damage`.
`InvalidDefinition`	The Damage Definition is null or fails validation.	Repair it in the authoring overview and rerun Project Health.
`InvalidSource`	A required source entity/Actor is invalid.	Reacquire the source or use an environmental/Actor request allowed by the definition.
`InvalidTarget`	The target entity is unset, stale, or lacks required state.	Reacquire a valid damageable target.
`SelfDamageDisallowed`	Source and target match while the definition forbids self damage.	Choose another target or explicitly enable self damage.
`TargetAlreadyDead`	The target already satisfies the configured death condition.	Suppress repeat hits/rewards and wait for lifecycle handling.
`MissingRequiredTag`	The target lacks a tag required by the definition.	Treat as a gate or establish the prerequisite tag first.
`BlockedByTag`	An authored immunity/blocking tag is present.	Respect or explicitly remove the immunity.
`BlockedByPolicy`	A project damage-policy provider vetoed the request.	Inspect `BlockingPolicyId` and `BlockingPolicyReason`.
`PolicyEvaluationInProgress`	A provider re-entered damage while policies were already being evaluated.	Defer nested damage and keep providers side-effect-free.
`GameplayTagFailure`	Required/death tag work failed validation or atomic commit.	Inspect `GameplayTagResult`, failed tag, trait, counts, and capacity.

Value	Cause or meaning	Recommended response
`AttributeReadFailed`	A configured formula attribute could not be read.	Inspect `AttributeResult` and `FailedAttribute`; repair schema/archetype setup.
`InvalidMagnitude`	Base damage or a resolved calculation became NaN, infinite, or invalid.	Sanitize inputs and definition values before applying damage.
`TransactionFailed`	The final atomic Health/Shield/death-tag effect failed.	Inspect the nested transaction result and failed operation.
`QueueFull`	The bounded pending-damage queue is full.	Throttle damage producers or raise the measured queue budget.

`EMFDamagePolicyDecision`

Each policy provider returns one decision; any `Deny` vetoes the damage request.

Value	Cause or meaning	Recommended response
`Abstain`	This provider has no opinion for the request.	Continue evaluating later providers.
`Allow`	This provider permits the request but does not override another provider's veto.	Continue evaluation; use it for diagnostics if desired.
`Deny`	This provider blocks the request.	Supply stable policy/reason IDs and present the relevant gameplay feedback.

`EMFDamagePolicyProviderResult`

Returned by damage-policy provider registration and unregistration.

Value	Cause or meaning	Recommended response
`Success`	Registration/unregistration changed the live provider registry.	Continue normal lifecycle handling.
`InvalidWorld`	No live damage subsystem was available.	Register from a live object in the intended world.
`InvalidProvider`	The provider is null, destroyed, or lacks the required interface.	Pass a live implementing object/component.
`WrongWorld`	The provider belongs to a different world.	Register it with its own world's subsystem.
`AlreadyRegistered`	The same live provider is already present.	Treat setup as idempotent; do not add another registration.
`CapacityReached`	The bounded world provider registry is full.	Consolidate policy rules or remove unused providers.
`EvaluationInProgress`	Registry mutation was attempted during a policy callback.	Defer registration changes until evaluation completes.
`NotRegistered`	Unregistration found no matching live provider.	Treat it as already removed or correct the lifecycle pairing.

`EMFDeathHandlingRequestResult`

Describes the immediate post-death lifecycle request attached to successful killing damage.

Value	Cause or meaning	Recommended response
`NotRequested`	The target survived or the definition keeps dead entities.	Apply project lifecycle behavior only if separately configured.
`ProjectDeactivationRequested`	Mass Forge emitted a request for project-owned pooling/deactivation.	Handle the request ID once and report completion in project logic.
`DestructionQueued`	Safe deferred Mass-entity destruction was accepted.	Correlate the later destruction completion with the request ID.
`QueueFull`	Deferred destruction could not be scheduled.	Keep the dead entity inert, throttle deaths, and retry through controlled project logic if appropriate.

`EMFDeathDestructionResult`

Final result published for queued Mass-entity destruction.

Value	Cause or meaning	Recommended response
`Destroyed`	The generation-checked entity was still valid and was destroyed.	Release project-side presentation and references.
`EntityAlreadyInvalid`	The entity was already destroyed/reused before the request executed.	Treat destruction as terminal and never act on the old handle.

`EMFAbilityResult`

Shared by ability ownership, checks, activation, lifecycle control, AI requests, and queued operation receipts/completions.

Value	Cause or meaning	Recommended response
`Success`	The operation completed or an activation/lifecycle request was accepted.	Inspect the enclosing receipt/result for request ID and committed effects.
`InvalidWorld`	No live ability subsystem was available.	Supply a valid world context.
`InvalidEntity`	The source entity is unset or stale.	Reacquire the source handle.
`MissingAbilityFragment`	The source archetype lacks ability state.	Add the Mass Forge Ability Trait.
`InvalidAbility`	The Ability asset is null, invalid, or changed incompatibly.	Repair it in the authoring overview and rerun Project Health.
`NotGranted`	The source does not own the requested stable ability ID.	Grant it first or correct the ID.
`AlreadyGranted`	A grant request targeted an already-owned ability.	Treat the grant as unnecessary or revoke deliberately before replacement.

Value	Cause or meaning	Recommended response
`NoFreeAbilitySlot`	The compact ability fragment has no free grant slot.	Revoke an ability or reduce the loadout.
`QueueFull`	The shared bounded Activate/Grant/Revoke/Cancel/Interrupt or AI-request capacity is exhausted.	Throttle producers or increase the measured budget.
`TargetRequired`	The ability requires a target but none was supplied.	Select and pass a valid target.
`InvalidTarget`	The supplied target is stale or otherwise invalid.	Reacquire the target.
`SelfTargetNotAllowed`	Source and target match while self-targeting is disabled.	Select another target or change the authored targeting rule.
`TargetingPolicyRejected`	A targeting policy denied the source/target pair.	Inspect the policy result/index/ID and correct range or project conditions.
`TargetingPolicyEvaluationInProgress`	Policy code re-entered activation during evaluation.	Keep policies side-effect-free and defer nested activation.
`OnCooldown`	The individual ability cooldown has not expired.	Wait for the reported remaining time.
`OnCooldownGroup`	A shared/global cooldown group is active.	Query the group remaining time and wait.
`NoFreeCooldownGroupSlot`	Starting a new named cooldown would exceed bounded group storage.	Consolidate group names or reduce simultaneous groups.
`NoCharges`	The ability has no available charge.	Wait for charge recovery or change the loadout rules.
`CannotAfford`	One or more cost attributes are below their required values.	Inspect the failed cost/attribute and restore resources.
`RequirementNotMet`	An authored source/target attribute comparison failed.	Inspect the requirement index, subject, observed value, and threshold.
`MissingGameplayTagFragment`	Ability tag gates/effects require tag storage absent from an entity.	Add the Gameplay Tag Trait to the relevant archetype.
`MissingRequiredTag`	A required source/target tag is absent.	Establish the prerequisite or treat this as an expected gate.
`BlockedByTag`	A source/target blocking tag is present.	Respect or explicitly cleanse the blocking state.
`InvalidAttribute`	A cost, requirement, or nested effect references an unresolved attribute.	Repair the stable attribute ID/schema dependency.
`MissingMagnitudeParameter`	A nested effect needs a context scalar missing from the ability defaults/context.	Add the named default/input parameter.
`GameplayTagEffectFailed`	A nested tag transaction failed.	Inspect failed tag details, counts, capacity, and fragment setup.
`PersistentEffectFailed`	A referenced persistent effect could not apply.	Inspect the nested persistent result and asset/archetype setup.
`TransactionInProgress`	Re-entrant code attempted a conflicting ability transaction.	Defer nested work until the current transaction/event returns.

Value	Cause or meaning	Recommended response
`ActiveAbilityInProgress`	The source already has an active cast/channel.	Wait, cancel/interrupt it when allowed, or reject the new input.
`NoActiveAbility`	A lifecycle query/control request found no active cast/channel.	Treat it as already ended and clear stale presentation.
`CancellationBlocked`	The active ability does not permit voluntary cancellation.	Continue it or use a permitted interruption path.
`InterruptionBlocked`	The active ability is immune to interruption.	Respect the authored rule.
`InvalidRequestId`	A supplied AI request ID is non-positive or malformed.	Retain the positive ID returned by an accepted receipt.
`RequestNotFound`	No tracked AI request matches the ID.	Correct the ID or treat it as expired/forgotten history.
`RequestNotTerminal`	Forget was requested while the AI ticket is still pending/active.	Cancel or wait for a terminal state first.
`RequestAlreadyTerminal`	A control operation targeted a ticket that has already finished.	Read its terminal state and stop issuing control requests.
`RequestCancelled`	A queued AI activation was cancelled before execution.	Treat it as a terminal cancellation, not a system failure.
`InvalidEffect`	A referenced Instant/Persistent Effect is invalid at execution time.	Repair the dependency and rerun Project Health.
`MassIsProcessing`	Direct ability state access would overlap Mass processing.	Use the matching queued operation or invoke the direct operation from a safe phase.
`LifecycleChanged`	A queued Cancel/Interrupt captured one lifecycle ID, but a different lifecycle is now active on that source.	Treat the request as safely stale; inspect the new lifecycle and submit a new request only if it should also stop.
`ConditionRejected`	A project-authored activation condition returned a non-passing response.	Inspect the failed condition index, stable condition ID, result, and reason; treat expected game-rule denial separately from invalid configuration.
`ConditionEvaluationInProgress`	Condition code re-entered ability admission while another condition was being evaluated.	Keep conditions side-effect-free and defer nested gameplay work until the current activation completes.
`MagnitudeCalculationFailed`	A nested Instant Effect custom magnitude failed while the ability transaction was staged.	Inspect `FailedEffectId`, `FailedModifierIndex`, and `MagnitudeCalculationResponse`; costs, effects, cooldown, charges, and lifecycle state remain uncommitted.
`ExecutionPlanRejected`	The assigned execution plan rejected evaluation or returned invalid, unsafe, or over-limit output.	Inspect `ExecutionPlanResponse`, its stable plan ID/result/reason, and the plan's returned references and parameters; activation state remains uncommitted.
`ExecutionPlanEvaluationInProgress`	Plan code re-entered ability admission while another plan evaluation was active.	Keep execution plans deterministic and side-effect-free; defer nested gameplay operations until the current evaluation returns.

`EMFAbilityExecutionPlanResult`

Detailed response from an optional Blueprint/C++ Ability Execution Plan.

Value	Cause or meaning	Recommended response
`Success`	The plan returned a payload that may proceed to framework validation and transaction staging.	Let Mass Forge validate and commit it; do not apply the described effects manually.
`Rejected`	Project logic intentionally declined to produce an activation or channel-pulse payload.	Inspect the stable plan ID/reason and treat expected gameplay denial separately from setup failure.
`InvalidWorld`	The query lacks a live world or crosses world ownership.	Evaluate from the same active world as the source and target.
`InvalidSource`	The query's source entity or Actor is no longer valid.	Reacquire the source and retry only through normal ability admission.
`InvalidTarget`	Target output requires a valid target, or the supplied target/Actor is stale or cross-world.	Enable the Ability's target contract and pass a current target from the same world.
`InvalidConfiguration`	The plan is missing/unloadable/unidentified, has an invalid merge mode, or uses the unimplemented fail-closed base class.	Repair or implement the plan, assign a unique `PlanId`, and rerun Data Validation and Project Health.
`RecursiveEvaluation`	Plan code triggered another plan evaluation before returning.	Remove gameplay mutations and nested activation from the callback; return only a transaction description.
`OutputLimitExceeded`	The payload exceeds 64 combined effects or 64 parameter overrides.	Split the design into bounded operations or reduce the payload; do not raise hidden per-activation work.
`InvalidEffect`	A returned effect is missing, unloadable, unidentified, or Persistent Effect output was returned for a channel pulse.	Repair the reference/ID and keep channel payloads Instant-only.
`InvalidParameter`	A returned parameter has no name or a NaN/infinite value.	Supply a stable non-empty name and finite scalar.

`EMFAbilityRequestOperation`

Identifies which operation produced an `FMFAbilityRequestCompletion`.

Value	Cause or meaning	Recommended response
`Activate`	The request attempted an Ability activation or cast/channel admission.	Inspect `Result` and the nested activation diagnostics.
`Grant`	The request attempted to add a validated Ability definition to the entity.	On failure, inspect the entity, definition, and bounded grant capacity.
`Revoke`	The request attempted to remove one submitted Ability ID and its runtime state.	Treat `NotGranted` as an already-absent ownership state when appropriate.
`Cancel`	The request attempted voluntary cancellation of the exact captured lifecycle.	Handle blocked, ended, or replaced lifecycles explicitly.
`Interrupt`	The request attempted external interruption of the exact captured lifecycle.	Handle immunity, ended, or replaced lifecycles explicitly.

`EMFAbilityTargetingPolicyResult`

Detailed response from a reusable ability targeting policy.

Value	Cause or meaning	Recommended response
`Allowed`	The policy accepts this source/target pair.	Continue evaluating later policies.
`Rejected`	Custom policy logic rejected the pair.	Inspect the policy's stable reason and provide gameplay feedback.
`InvalidWorld`	The policy query has no valid world.	Supply a live world context.
`InvalidSource`	The source entity/Actor is invalid for this policy.	Reacquire or correct the source.
`InvalidTarget`	The target entity/Actor is invalid for this policy.	Reacquire or correct the target.
`MissingSourcePosition`	The selected position source cannot resolve a source location.	Add a Mass transform/representation or change the policy position mode.
`MissingTargetPosition`	The selected position source cannot resolve a target location.	Add a Mass transform/representation or change the policy position mode.
`BelowMinimumRange`	Distance is smaller than the authored minimum.	Move apart or adjust the validated minimum.
`BeyondMaximumRange`	Distance exceeds the authored maximum.	Move closer or adjust the validated maximum.
`InvalidConfiguration`	Policy settings are contradictory or non-finite.	Repair and validate the policy asset.

`EMFAbilityConditionResult`

Detailed response from a reusable project-authored activation condition.

Value	Cause or meaning	Recommended response
`Passed`	The condition allows activation to continue.	Continue evaluating later conditions.
`Rejected`	Project logic intentionally denied activation.	Inspect the stable condition ID and reason, then provide appropriate gameplay feedback.
`InvalidWorld`	The condition cannot resolve required authoritative world state.	Supply a live world or repair the condition's world-state dependency.
`InvalidSource`	The source entity/Actor is unusable for this condition.	Reacquire the source or handle actorless entities explicitly.
`InvalidTarget`	The optional/required target is unusable for this condition.	Require and reacquire a target or handle an unset target explicitly.
`InvalidConfiguration`	The condition asset is unimplemented or configured incorrectly.	Implement `Evaluate Condition`, repair the asset, and rerun validation/Project Health.

`EMFAbilityTargetSelectionResult`

Top-level result for built-in and provider-driven target selection.

Value	Cause or meaning	Recommended response
`Success`	Selection completed; an empty target array can still be a valid result.	Use bounded targets and inspect truncation/diagnostic counts.
`InvalidWorld`	No live Mass world could be resolved.	Supply a valid world context.
`MassIsProcessing`	The query cannot safely inspect Mass entity data now.	Run selection from a safe phase.
`InvalidSource`	A required source entity/Actor is invalid.	Reacquire or correct the source.
`InvalidTarget`	A collision hit did not resolve to a valid Mass target.	Treat it as a miss or resolve another target.
`InvalidRequest`	Radius, trace, bounds, or parameter inputs are invalid/non-finite.	Sanitize origin/end/radius/max-target inputs.
`InvalidProvider`	The custom provider is null, invalid, or lacks the interface.	Pass a live implementing provider.
`ProviderEvaluationInProgress`	Provider selection was re-entered recursively.	Keep the provider side-effect-free and defer nested selection.
`ProviderFailed`	The provider returned a non-success response.	Inspect `ProviderId` and `Reason`, then correct project logic or inputs.

`EMFAbilityTargetProviderResult`

Authored response returned by a custom target provider before Mass Forge sanitizes its handles.

Value	Cause or meaning	Recommended response
`Success`	The provider produced a candidate list for validation.	Return stable provider identity and let Mass Forge bound/de-duplicate it.
`Rejected`	Project logic intentionally declined the query.	Supply a stable reason and treat it as a controlled selection failure.
`InvalidRequest`	Project logic considers the supplied query malformed/unsupported.	Validate query parameters before invoking the provider.

`EMFAbilityAIRequestStatus`

Pollable lifecycle status for a tracked AI activation request.

Value	Cause or meaning	Recommended response
`Invalid`	No valid tracked state is represented.	Check the enclosing query result and request ID.
`Pending`	Accepted work is waiting in the bounded activation queue.	Continue polling without resubmitting.

Value	Cause or meaning	Recommended response
`Active`	The request started a live cast/channel lifecycle.	Read lifecycle progress and continue polling.
`Succeeded`	Activation or lifecycle completed successfully.	Consume the terminal result and forget the ticket when no longer needed.
`Failed`	Activation or commit ended with a non-success result.	Inspect the nested activation/lifecycle diagnostics.
`Cancelled`	The request/lifecycle ended through allowed cancellation.	Treat as a terminal controlled outcome.
`Interrupted`	The active lifecycle was interrupted.	Treat as terminal and inspect interruption context in project events.

`EMFAbilityLifecycleEndReason`

Terminal reason published for a cast/channel lifecycle.

Value	Cause or meaning	Recommended response
`Completed`	The lifecycle reached its authored end and committed successfully.	Finish presentation and consume the activation result.
`Cancelled`	A permitted voluntary cancellation ended it.	Roll back/stop presentation according to project UX.
`Interrupted`	A permitted external interruption ended it.	Present interruption feedback and clear active state.
`CommitFailed`	Final or periodic atomic work could not commit.	Inspect the detailed ability/effect result and repair runtime prerequisites.
`AbilityRevoked`	The granted ability was removed while its lifecycle was active.	Treat the lifecycle as terminated and update loadout UI.
`SourceInvalid`	The source entity became invalid before completion.	Drop the lifecycle and all stale source references.

`EMFHighVolumeEffectResult`

Admission or terminal outcome for the per-entity compiled Apply/Remove mailbox.

Value	Cause or meaning	Recommended response
`None`	The mailbox has no admission or execution result.	Submit a command or wait for a completed mailbox before consuming.
`Accepted`	A valid command now owns the entity's single mailbox.	Wait for the high-volume processor and consume the correlated completion.
`Success`	Every compiled attribute/tag operation committed atomically.	Consume the completion and release the mailbox.
`Busy`	A pending command or unconsumed completion already owns the mailbox.	Do not overwrite or spin; consume completion before resubmitting.

Value	Cause or meaning	Recommended response
`InvalidRequestId`	The project supplied correlation ID `0` or a negative value.	Allocate a positive per-producer correlation ID.
`InvalidEffectIndex`	The numeric effect index is outside the archetype's compiled table.	Resolve/cache the ID from the matching shared fragment before entity iteration.
`InvalidOperation`	The operation is neither Apply nor Remove.	Pass a declared `EMFHighVolumeEffectOperation` value.
`InvalidDefinition`	The selected compiled side is empty or exceeds its fixed capacity.	Repair the trait/effect assets and recreate the entity template.
`MissingAttributeSlot`	A compiled domain/slot cannot resolve against the entity fragments.	Verify the Attributes Trait/schema and rebuild stale templates after schema changes.
`InvalidMagnitude`	Existing state, the compiled magnitude, or projected arithmetic is non-finite.	Repair the source asset/state; the command changed nothing.
`MissingGameplayTagFragment`	The selected side changes tags but this archetype has no tag fragment.	Add/configure the Gameplay Tag Trait or remove tag operations from the pair.
`GameplayTagFailure`	A projected tag operation failed validation, count, or capacity rules.	Inspect `GameplayTagResult` and `FailedOperationIndex`; the command changed nothing.

`EMFEntityInspectionResult`

Returned by point-in-time entity inspection captures.

Value	Cause or meaning	Recommended response
`Success`	A bounded snapshot was captured.	Render its built-in/custom sections without treating them as simulation authority.
`InvalidWorld`	No live inspection subsystem was available.	Select an object/entity from the intended active world.
`InvalidEntity`	The requested handle or represented Actor is invalid/stale.	Reacquire the selection.
`MassIsProcessing`	Snapshotting would overlap Mass processing.	Refresh on a later editor/game tick.

`EMFInspectionProviderResult`

Returned by custom inspection-provider registration and unregistration.

Value	Cause or meaning	Recommended response
`Success`	The provider registry changed as requested.	Pair registration with lifecycle-safe unregistration.
`InvalidWorld`	No live inspection subsystem was available.	Register from a live object in the intended world.
`InvalidProvider`	The provider is null or no longer valid.	Pass a live object.
`UnsupportedProvider`	The object does not implement the inspection-provider interface.	Implement the interface or use the supplied component pattern.

Value	Cause or meaning	Recommended response
`AlreadyRegistered`	The provider is already in the registry.	Treat setup as idempotent and avoid duplicate registration.
`CapacityReached`	The bounded provider registry is full.	Consolidate sections/providers or unregister unused providers.
`NotRegistered`	Unregistration found no matching provider.	Treat it as already removed or correct lifecycle pairing.

`EMFGameplayEventHistoryResult`

Returned by bounded gameplay-event history queries and entity watch-list operations.

Value	Cause or meaning	Recommended response
`Success`	The query or watch-list operation completed.	Consume the chronological records or continue the diagnostic session.
`InvalidWorld`	The gameplay-event subsystem no longer belongs to a live world.	Reacquire the subsystem from the intended active world.
`InvalidEntity`	A watch handle is unset, stale, or belongs to another entity generation.	Reacquire a live entity handle before adding the watch.
`AlreadyWatched`	The entity is already present in the bounded watch list.	Treat the watch setup as idempotent; do not add a duplicate.
`NotWatched`	Removal found no matching entity in the watch list.	Treat it as already removed or verify the generation-sensitive handle.
`WatchCapacityReached`	The watch list already contains its maximum 64 entities.	Remove unused watches or divide diagnostics into smaller sessions.
`InvalidLimit`	A history query requested fewer than 1 or more than 65,536 records.	Supply a positive bounded maximum within the documented range.

`EMFNetworkEntityReferenceResult`

Returned when translating between a world-local Mass handle and an Unreal Mass session network identity.

Value	Cause or meaning	Recommended response
`Success`	The entity and session reference resolved to one current full generation.	Use the reference only within this replicated world/session.
`InvalidWorld`	The world or Mass entity subsystem is unavailable.	Reacquire the subsystem from the live network world.
`InvalidEntity`	The supplied Mass handle is unset or stale.	Reacquire the complete current entity generation.
`MissingNetworkIdentity`	The entity has no valid Unreal Mass network-ID fragment.	Add/configure the Mass Replication trait and wait for identity initialization.
`InvalidNetworkReference`	The incoming session reference is zero, negative, or larger than the native unsigned 32-bit Mass network-ID range.	Send a valid replicated network identity, never an entity index or an unchecked Blueprint integer.

Value	Cause or meaning	Recommended response
`UnknownNetworkReference`	No authoritative registration or client bubble mapping currently owns the ID.	Wait for relevance/registration or discard a reference from another session.
`StaleNetworkReference`	The cached mapping no longer matches a live entity with the same ID.	Discard the mapping and reacquire current replicated state.

`EMFNetworkEntityRegistryResult`

Returned by the authority-side constant-time network identity registry.

Value	Cause or meaning	Recommended response
`Success`	The authority registry changed as requested.	Pair registration with entity retirement/unregistration.
`InvalidWorld`	No live network world is available.	Register through the intended world's authority subsystem.
`NotServerAuthority`	Registration was attempted on a remote client.	Register only where the authoritative Mass entity exists.
`InvalidEntity`	The supplied handle is unset or stale.	Reacquire the server's current entity generation.
`MissingNetworkIdentity`	The entity has no initialized Mass network ID.	Configure Mass Replication before registering gameplay identity.
`DuplicateNetworkIdentity`	The ID already maps to a different live entity.	Repair entity creation/identity assignment; never overwrite the earlier mapping.
`AlreadyRegistered`	The same ID already maps to the same full entity generation.	Treat setup as idempotent.
`NotRegistered`	Unregistration found no entry for that entity generation.	Treat it as already retired or repair lifecycle pairing.

`EMFNetworkDefinitionRegistryResult`

Returned by the authority-owned catalog of definitions approved for future network dispatch. Client input never loads assets into this catalog.

Value	Cause or meaning	Recommended response
`Success`	The catalog operation completed.	Continue with the returned kind/ID or resolved server object.
`InvalidWorld`	No live network world is available.	Register during authoritative world or match setup.
`NotServerAuthority`	A remote client attempted to maintain or resolve the trusted catalog.	Perform catalog work only on server authority.
`InvalidDefinition`	The supplied Primary Data Asset is null, destroyed, or otherwise invalid.	Retain and pass a live server-owned definition.
`UnsupportedDefinitionType`	The asset is not a Mass Forge Ability, Instant Effect, Persistent Effect, or Damage Definition.	Register only an explicitly supported gameplay definition family.

Value	Cause or meaning	Recommended response
`MissingDefinitionId`	The definition or lookup has no stable authored ID.	Assign and validate the asset's stable ID before match setup.
`DuplicateDefinitionId`	Another object already owns that ID within the same definition family.	Rename or remove the conflicting server definition; never replace it implicitly.
`AlreadyRegistered`	The same object already owns that family/ID entry.	Treat repeated setup as idempotent.
`NotRegistered`	Explicit unregistration found no entry for that object.	Treat it as already released or repair setup/teardown pairing.
`UnknownDefinition`	The requested family/ID was never admitted by the server.	Reject the client request or explicitly register the preloaded asset during setup.
`StaleDefinition`	A registered object became invalid or its authored ID changed after admission; the stale entry was removed.	Restore immutable runtime definitions and explicitly register the corrected object again.

`EMFNetworkRequestResult`

Returned by server-side validation of an untrusted client request descriptor. Success authorizes later dispatch but is not a gameplay receipt.

Value	Cause or meaning	Recommended response
`Success`	Identity validation passed and at least one project policy allowed the request without a veto.	Dispatch through the operation's bounded authoritative queue and correlate its separate receipt.
`InvalidWorld`	The authority subsystem has no live world.	Reacquire it from the current server world.
`NotServerAuthority`	A remote client attempted to evaluate authority policy.	Send through an owner-bound server RPC adapter instead of evaluating locally.
`InvalidRequestId`	The client-local correlation ID is zero or negative.	Allocate a positive client-local ID before submission.
`InvalidOperation`	The operation is unset, structurally incomplete, has a non-finite magnitude, or has a negative count.	Build a valid operation-specific descriptor before transport.
`InvalidRequester`	The requester Player Controller is null/destroyed.	Invoke from the owning live connection.
`WrongRequesterWorld`	The requester belongs to a different world.	Reject cross-world input and reacquire the correct connection.
`InvalidSourceReference`	The request supplied no source network identity.	Send the authoritative source entity's session reference.
`InvalidTargetReference`	Damage or Persistent Effect Remove omitted its required target.	Supply the intended target's current session reference.
`SourceEntityUnavailable`	The source reference is unknown or stale on authority.	Stop retrying stale IDs; wait for current ownership/relevance state.
`TargetEntityUnavailable`	The optional/required target reference is unknown or stale.	Reacquire the target or reject the action in UI.

Value	Cause or meaning	Recommended response
`NoAllowingPolicy`	No live project policy explicitly allowed the request.	Install/repair a bounded ownership policy; do not change the default to implicit allow.
`DeniedByPolicy`	A project policy vetoed the request.	Use the deciding policy ID/reason for diagnostics and do not dispatch.
`InvalidPolicyResponse`	A non-Abstain response omitted its stable policy ID or returned an unknown decision.	Repair the provider implementation; the request changed nothing.
`EvaluationInProgress`	A policy recursively evaluated or changed the registry during evaluation.	Remove policy re-entry and defer follow-up work until evaluation returns.

`EMFNetworkStateResult`

Returned when maintaining or immediately refreshing one Player Controller's owner-only gameplay-state relevance set.

Value	Cause or meaning	Recommended response
`Success`	Relevance changed or the requested capture completed without truncation.	Consume the initial/current state or wait for later semantic update events.
`InvalidWorld`	The component or required world subsystem has no live world.	Reacquire the owning Player Controller and component from the active network world.
`InvalidOwner`	The component is not the sole Mass Forge state component on a live Player Controller.	Install exactly one state component as a default Player Controller component.
`NotServerAuthority`	A client attempted to add, remove, clear, or force authority relevance.	Make relevance decisions on the server; clients only query replicated state.
`InvalidEntity`	The authoritative Mass handle is unset, stale, or was invalidated during refresh.	Reacquire the current generation or allow automatic relevance removal to complete.
`MissingNetworkIdentity`	The entity does not yet carry an initialized Mass network ID.	Configure Mass Replication and add relevance only after identity initialization.
`AlreadyRelevant`	The same session identity is already in this connection's relevance set.	Treat setup as idempotent and query its existing state.
`NotRelevant`	Removal or forced refresh found no matching session identity.	Treat removal as already complete or add the entity on authority first.
`CapacityReached`	This connection reached its configured relevant-entity bound.	Remove stale interest entries or raise the profiled bound deliberately.
`CaptureDeferred`	Mass was processing, so relevance was retained without reading unstable fragments.	Wait for the bounded scheduled retry and its Added/Updated event.
`CaptureFailed`	Registry setup, a section result combination, or a fixed payload capacity prevented a complete baseline.	Inspect entity traits/capacities and correct setup; Mass Forge never sends a truncated state.
`StaleNetworkIdentity`	A relevant entity's Mass network ID changed after its connection binding was created.	Retire the stale relevance entry and register/add the entity under its current immutable session identity.
`InvalidAttributeRule`	An attribute rule has an unset ID, a negative/non-finite step, a duplicate Attribute ID, or the rule array exceeds the fixed attribute capacity.	Correct the complete rule set and configure it again; the previous runtime policy remains unchanged.

`EMFNetworkRequestPolicyProviderResult`

Returned when maintaining the bounded weak server request-policy chain.

Value	Cause or meaning	Recommended response
`Success`	The policy registry changed as requested.	Keep lifecycle-safe registration/unregistration pairing.
`InvalidWorld`	No live authority world exists.	Register from a world-owned server object.
`NotServerAuthority`	A remote client attempted to own server request policy.	Install the provider on Game State or another authority-owned object.
`InvalidProvider`	The object is null or does not implement the policy interface.	Pass a live `IMF_NetworkRequestPolicy` object/component.
`WrongWorld`	The provider belongs to another world.	Register with its own world's subsystem.
`AlreadyRegistered`	The provider is already present.	Treat setup as idempotent.
`NotRegistered`	Unregistration found no matching live provider.	Treat it as already removed or correct teardown pairing.
`CapacityReached`	Sixteen live providers are already registered.	Consolidate related project checks or unregister unused providers.
`EvaluationInProgress`	Registration changed while a request policy chain was executing.	Defer registry mutation until the evaluation has returned.

`EMFNetworkClientSubmissionResult`

Returned immediately on the owning machine when handing a descriptor to the Player Controller request component. `Submitted` means only that the unreliable RPC was emitted or processed locally; wait for the server receipt before treating the request as accepted.

Value	Cause or meaning	Recommended response
`Submitted`	The owner-bound component assigned a positive correlation ID and handed the request to transport.	Store the ID and correlate the later server receipt and, if accepted, completion.
`InvalidWorld`	The component has no live world.	Submit only after the owning Player Controller and component have entered a world.
`InvalidOwner`	The component is not owned by a live Player Controller.	Add it as a default component of the project Player Controller class.
`NotLocallyControlled`	Code tried to submit through another connection's controller.	Submit only through the local owning Player Controller.
`RequestIdExhausted`	The component exhausted its positive signed 64-bit client request sequence.	Recreate the connection/component; never wrap or reuse correlation IDs.

`EMFNetworkTransportResult`

Returned by the server receipt after owner, payload, replay, rate, outstanding-capacity, policy, catalog, and backend-queue admission checks.

Value	Cause or meaning	Recommended response
`Accepted`	The request entered the matching bounded authoritative gameplay queue and has a positive backend ID.	Wait for the correlated terminal completion; activation completion may mean a cast/channel lifecycle started, not ended.
`InvalidOwner`	The replicated component is not owned by a live Player Controller.	Install the component on the Player Controller class and do not reparent it at runtime.
`NotServerAuthority`	Processing occurred outside authoritative server context.	Repair component replication/ownership and submit through its generated Server RPC.
`InvalidPayload`	Operation fields, bounds, scalar count, identities, or finite-number requirements failed.	Rebuild the operation-specific descriptor; the project policy and gameplay queues were not called.
`ReplayOrOutOfOrder`	The client ID was not greater than the last structurally valid ID received on this connection.	Allocate IDs monotonically through `Submit Network Request`; never retry with an old ID.
`RateLimited`	The connection exceeded its server-owned real-time request-window limit.	Back off and configure an intentional per-game limit on the component class defaults.
`TooManyOutstanding`	Too many accepted requests still await backend terminal events for this connection.	Wait for completion; investigate stalled world processing if capacity does not recover.
`AuthorizationRejected`	The authority subsystem or project policy chain rejected the request.	Inspect `AuthorizationResult`, `PolicyId`, and `Reason`; never bypass the deny-by-default policy.
`DefinitionRejected`	The stable ID did not resolve through the expected server-approved definition family.	Preload/register the correct definition on authority; never load using the client name.
`BackendUnavailable`	The required Mass Forge world subsystem was unavailable.	Verify plugin/world initialization and the operation's subsystem dependency.
`BackendRejected`	The matching bounded gameplay queue rejected admission or returned no valid backend ID.	Inspect the operation-specific result on the receipt, correct the request, or wait for queue capacity.
`UnsupportedOperation`	The common transport intentionally does not carry this operation, currently Entity State Restore.	Keep save restoration on a trusted project-owned server path until a separately bounded snapshot channel is implemented.

Reading compound results

- Attribute changes: branch on `FMFAttributeChangeResult.Result`, then use previous/new value and clamp state only on success.
- Instant Effects: read the top-level attribute result first, then failed-operation/tag fields; successful operation arrays describe the atomic commit.
- Damage: read `FMFDamageApplicationResult.Result` first. On layered failures, inspect `AttributeResult`, `GameplayTagResult`, `FailedAttribute`, `FailedTag`, policy identity/reason, and the nested transaction.
- Abilities: read `FMFAbilityActivationResult.Result` first, then requirement, targeting-policy, cost, gameplay-tag, Instant Effect, and Persistent Effect details relevant to that code.
- Queues: acceptance means `Success` plus a positive request ID. Rejection returns ID `0` and no completion. The later completion event is authoritative for execution-time success because complete entity generations and submitted definition identities are revalidated. See the deferred recovery matrix.

- Actor dispatch: `FMFEffectDispatchReceipt.Route` identifies whether Mass, an Actor interface, or a receiver component was selected; never assume an Actor uses only one ownership path.
- Projectiles: branch first on `FMFProjectileLaunchReceipt.Result`; after acceptance, use `FMFProjectileResolveResult.Reason`, then inspect the optional nested Damage result before treating impact as applied damage.

EXAMPLES AND PRESENTATION

Playable showcases guide

Mass Forge ships its reusable framework, editor teaching tools, and examples in one plugin. The companion Unreal project is only a customer-style launcher and verification host. The examples do not depend on Game Master.

Launcher

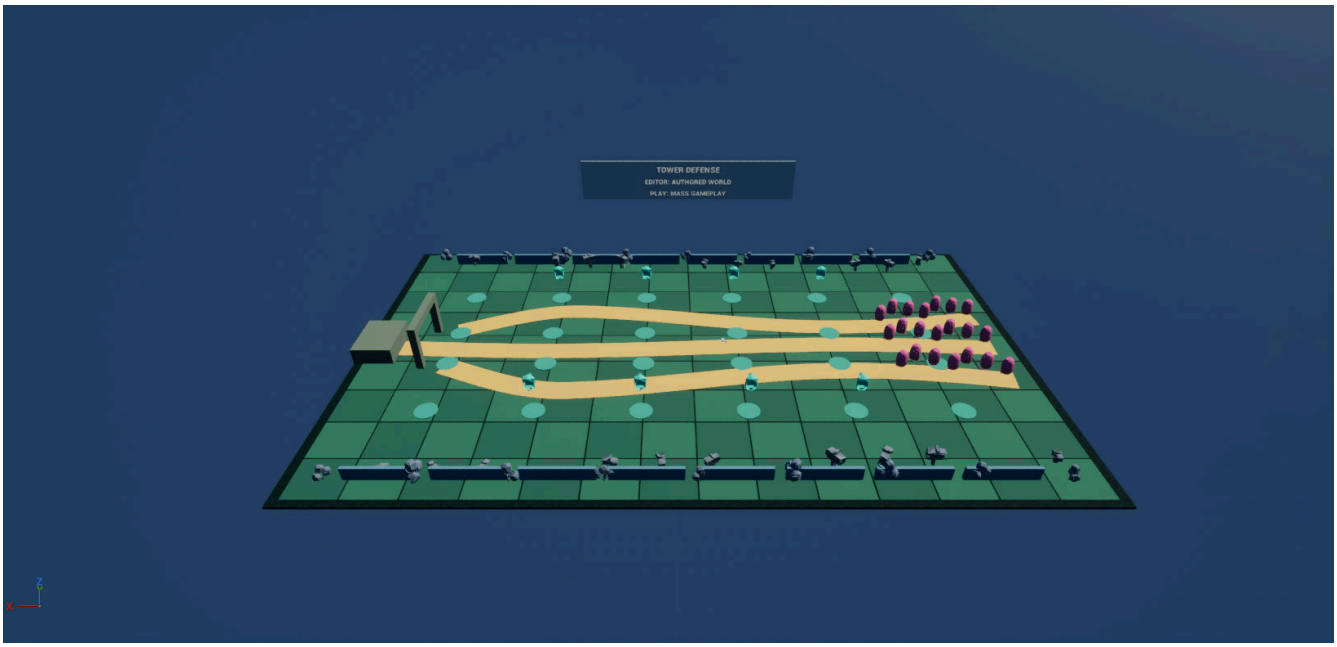
The project starts in `/MassForge/MassForge/L_MF_ShowcaseMenu`, located at the root of the plugin's `Content/MassForge` pack. Click a card to open its standalone map. Press **Escape** from an example to return to the launcher.

The initial release has five entries. Multiplayer is intentionally excluded from this sample suite.

Tower Defense

Map: `L_MF_TowerDefense`

Three routes feed mixed enemies toward a base. Actor towers acquire Mass targets and apply direct or area damage, slow effects, and upgrades through Mass Forge state.



The saved-world image shows construction before Play; the live image shows the same three routes under pressure. Towers are Actors for convenient placement, while target Health, effects, damage, and enemy lifecycle remain Mass Forge state.

This example teaches:

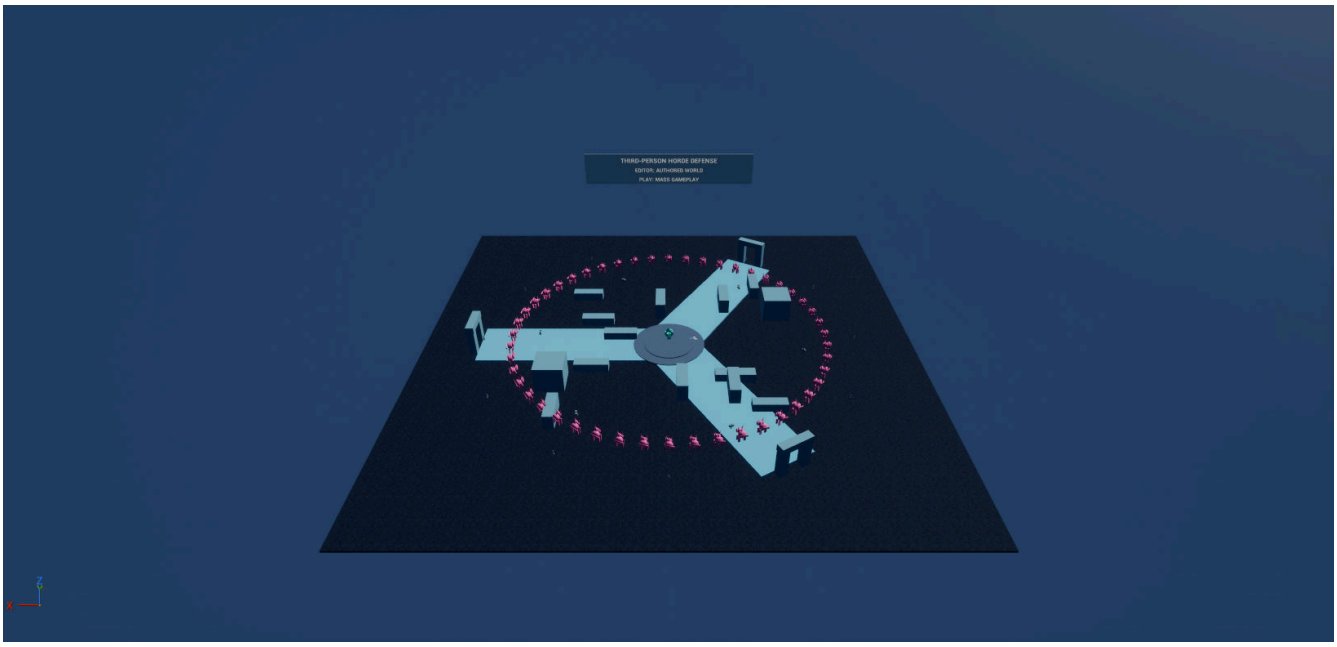
- schema-backed Health, Move Speed, Attack Power, Defense, and combat attributes;
- Actor-to-entity targeting and damage;
- effect application and removal;
- economy and upgrade state;
- stable entity presentation without origin stacking; and
- how wave roles change targeting decisions.

Third-Person Horde Defense

Map: `L_MF_ThirdPersonHordeDefense`

A third-person arcanist protects a fortified strongpoint through three finite assault waves. The player uses manual abilities while Mass enemies pursue, attack the player, and pressure the objective.

Fireball, frost-nova shards, hostile bolts, and VAT/skeletal attack timing use the same Mass projectile path. The attack pose begins first, the projectile releases at the authored frame, and damage commits on impact.



The expanding cyan bead trail is the example renderer reading one Mass projectile's snapshots. The attack starts before release, and the target loses Health only when the impact record resolves.

This example teaches:

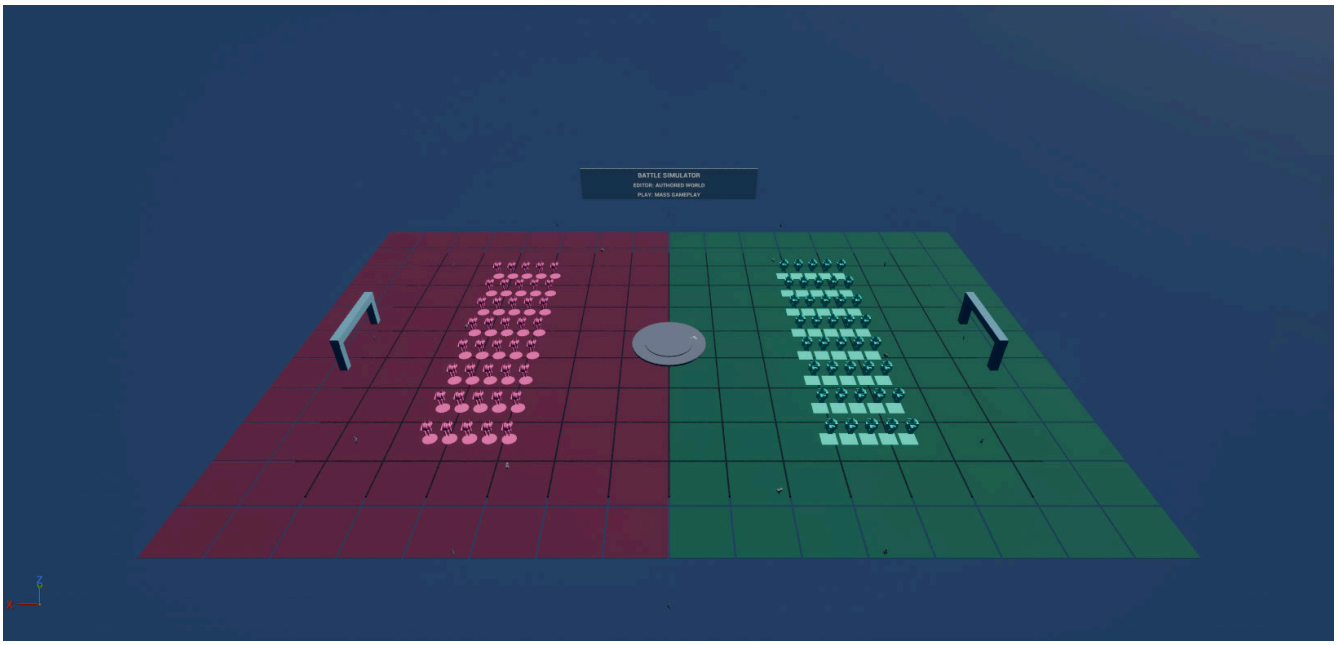
- player-to-entity and entity-to-player damage;
- abilities, costs, cooldowns, charges, and tags;
- objective attributes and failure/success conditions;
- blocking obstacles, jump-aware pursuit, and bounded local representation; and
- Actor presentation that observes Mass-owned gameplay state.

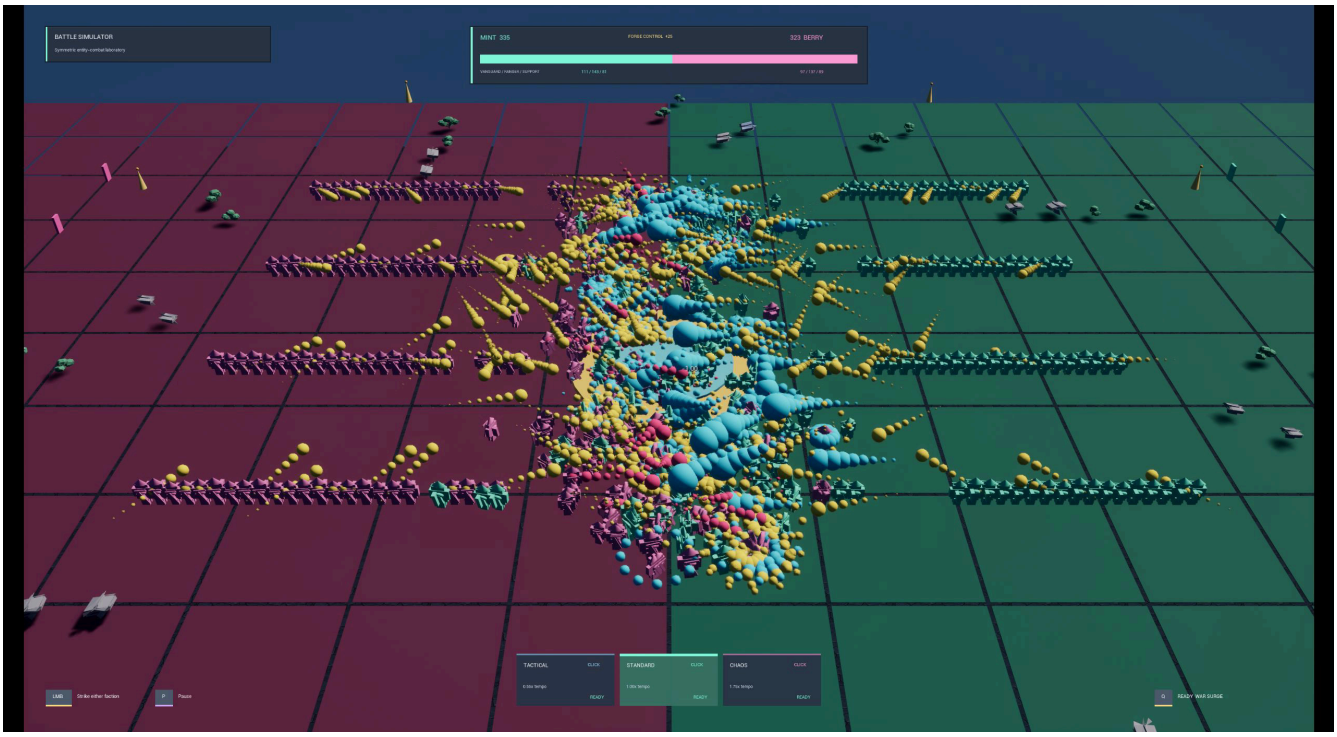
Battle Simulator

Map: `L_MF_BattleSimulator`

Vanguards resolve melee damage on their authored strike frame. Rangers and supports launch Mass-owned projectile entities; the sample HISM renderer is presentation-only, and damage/healing resolves when the projectile impact record returns to the safe gameplay phase.

Two symmetric 1,000-entity armies fight until one side wins. Roles include melee, ranged, support, and healing behavior. The post-battle tableau retains the result instead of freezing in an unexplained intermediate state.





The saved map above shows how the example is constructed; this Play-in-Editor capture shows the same map after both armies have entered combat. The visible projectile field is presentation for real Mass projectile entities whose damage or healing resolves on impact.

This example teaches:

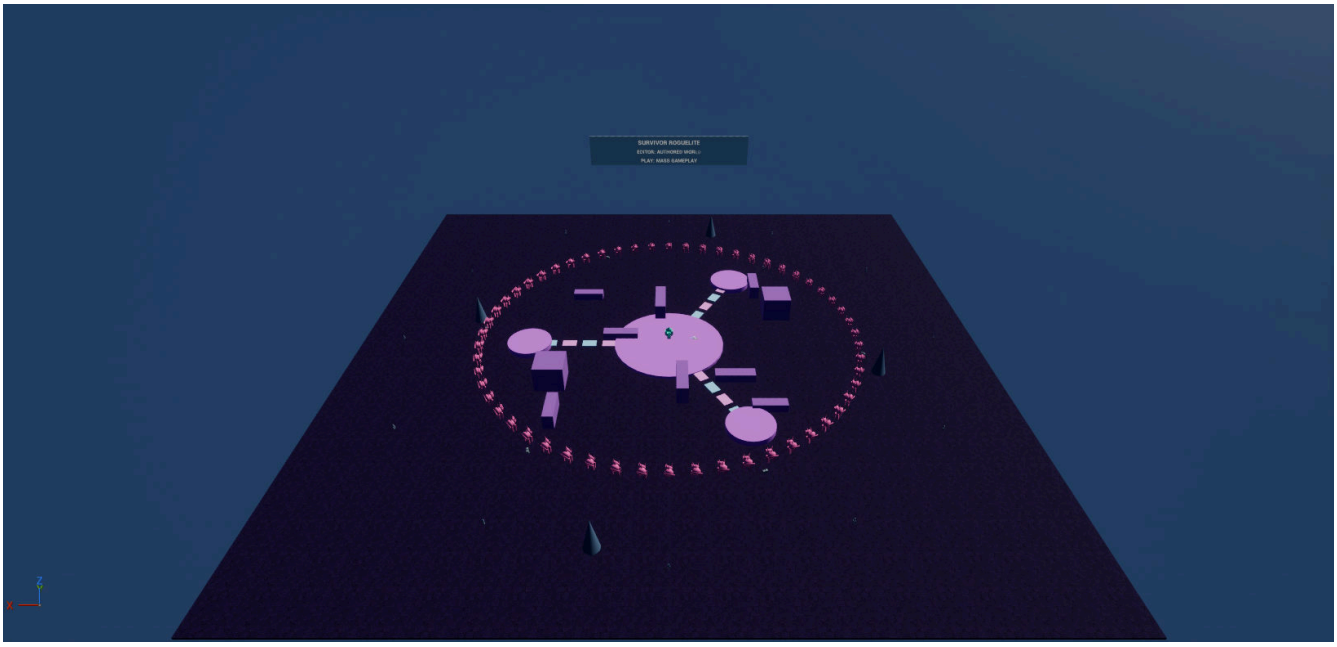
- project-owned faction tags, role-specific attributes, and the inspectable `FactionDamagePolicy` component that denies only shared configured tags;

- Mass-to-Mass target acquisition and retargeting;
- direct, ranged, and healing effects;
- deterministic balance inputs; and
- large-population outcomes without a per-unit Actor requirement.

Survivor Roguelite

Map: `L_MF_SurvivorRoguelite`

A third-person survivor loop combines automatic and aimed abilities, navigation around real collision, jumping, XP, ranks, and mouse-selected upgrade cards. Population presets are explicit about allocation versus active simulation and representation.



The example separates a 10,000-entity allocation from the bounded active set. The visible Arc Bolt, ability cards, Health, level, and XP are game presentation over Mass Forge state rather than hidden substitute counters.

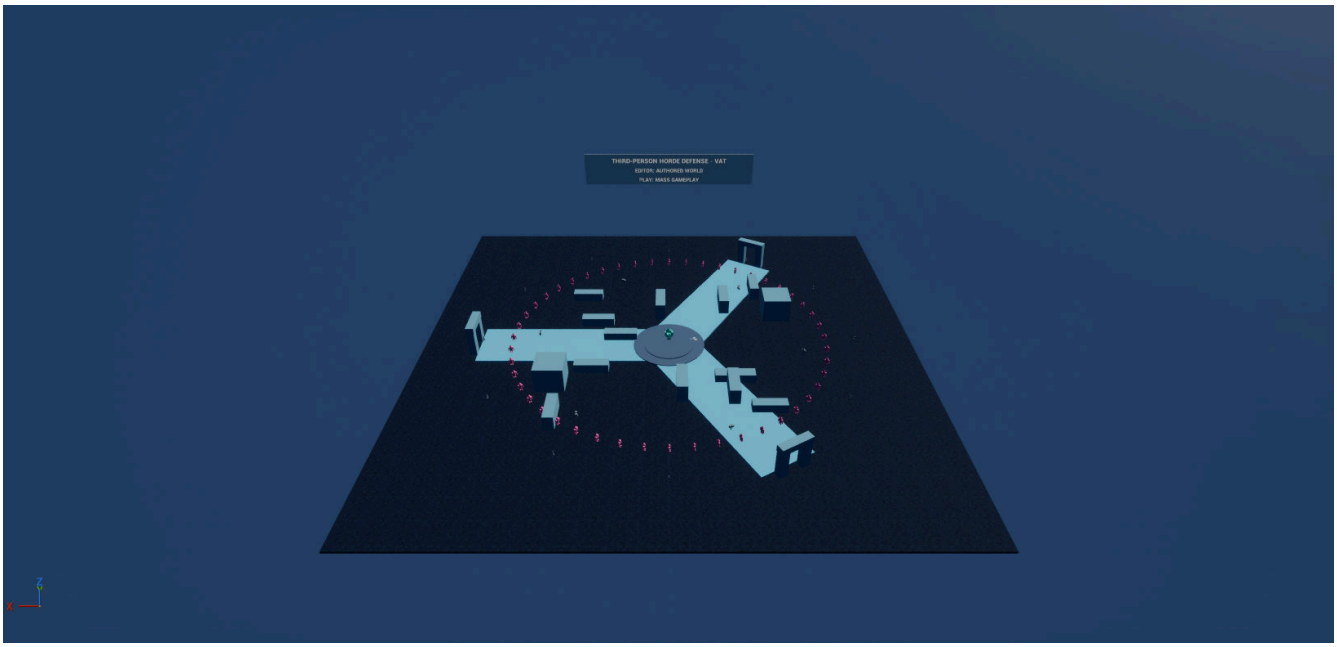
This example teaches:

- scalable enemy allocation and bounded active work;
- player attributes, incoming damage, recovery, XP, and ranks;
- data-driven upgrade choices;
- ability evolution during a run; and
- readable pursuit across a 3D arena.

Third-Person Horde Defense - VAT

Map: `L_MF_ThirdPersonHordeDefenseVAT`

This is the same finite game as Third-Person Horde Defense: the same 420 Mass enemies, player controls, balance, obstacles, abilities, damage, objectives, and three-phase win condition. Its only intentional difference is presentation: the horde is rendered with the project-owned Arcane Sentinel VAT mesh, position/normal textures, and material state machine instead of the bounded skeletal near tier.



The twin map keeps the same 420-enemy gameplay contract and swaps only the enemy presentation backend. Cyan and purple states come from per-instance gameplay/material data, not per-enemy dynamic material instances.

The VAT textures are sampled from the same Blender rig and clips used by the skeletal map. Gameplay states select Idle, Move, Attack, Cast, Hit, Jump, and Death clip ranges through a versioned twelve-float per-instance payload. Damage flash, frozen, burning, shielded, selected, and dissolve flags originate in Mass gameplay state/tags; they do not own damage or death.

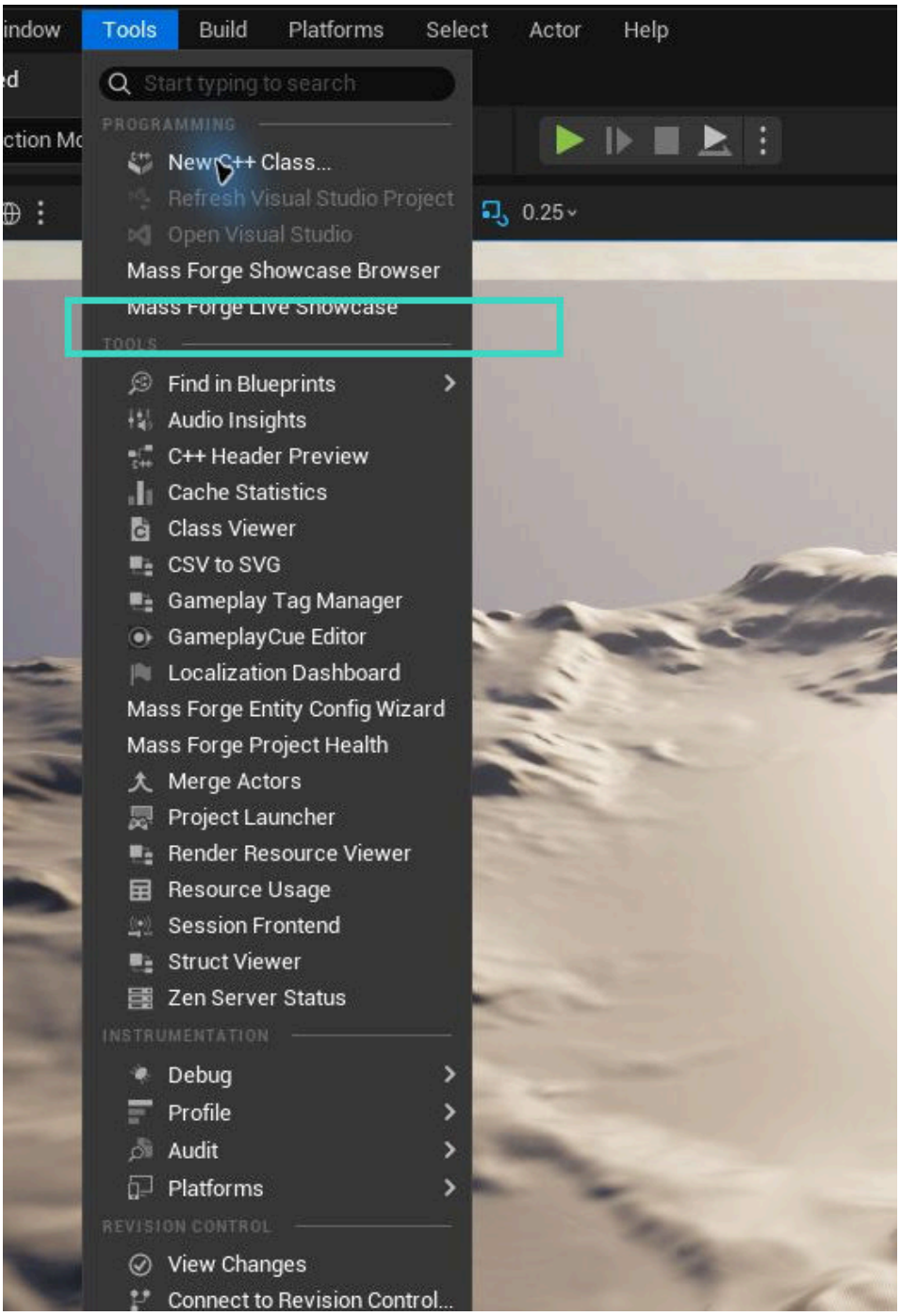
This example teaches:

- how one gameplay implementation can select skeletal or VAT presentation;
- genuine position/normal VAT playback with stable instance identity;
- animation transition and material-state data in a fixed custom-data layout;
- team palette and temporary status effects without per-entity dynamic material instances; and
- the separation between Mass-owned gameplay authority and replaceable GPU presentation.

The older `L_MF_VertexAnimation` procedural-WPO crowd map is retained only as an internal migration/performance regression fixture. It is not linked from the customer launcher and is never described as VAT.

Editor monitor

During PIE, **Mass Forge Live Showcase** opens automatically. Its pages are:



*If the monitor was closed, reopen it from **Tools > Mass Forge Live Showcase** while an example is running.*

Page	Purpose
Overview	Allocated, active, simulated, represented, animated, and resolved populations plus the map's proof statement
Gameplay	Current game metrics, concise controls, and safe pause/reset/scale actions
Performance	Frame observation and representation health without marketing unsupported benchmark claims
Recipe	Public integration steps and the ownership boundary between Mass Forge and sample presentation

The monitor observes gameplay at a bounded rate. It does not own the simulation and is excluded from Shipping runtime logic.

Inspectability rule

What customers see in the editor must be the real level or a clearly labeled, editable generator. Substitute preview scenery is not used. Any runtime-authored environment must expose its inputs in inspectable assets/components and be documented as runtime generation.

Every public map stores its Blueprint gameplay root, Blueprint world-layout actor, and Player Start. The running GameMode reuses those actors; it creates a fallback only when a developer deliberately runs the sample GameMode in a blank map. See How the examples are authored for the exact assets, editable definition references, and the Blueprint/C++ ownership boundary.

Verification

Maintainers run `Examples/MassForgeShowcases/Scripts/Verify-Showcases.ps1 -RequireSecondPassGameplayProof -KeepWorkspace`, then `Capture-Showcases.ps1`. The two character backends also pass `Verify-CharacterPresentationPackage.ps1`; its optional `-CaptureVATProfile` switch records a frame-bounded packaged observation. Passing automation is necessary but not sufficient: every final rendered frame and every saved map must receive human review before release.

See the character animation and VAT evidence for exact assets, counts, visual review, and benchmark boundaries, and how the examples are authored for the editable Blueprint and C++ ownership split.

EXAMPLES AND PRESENTATION

How the examples are authored

The five public examples are real plugin maps, not screenshots. Each map stores a Blueprint world-layout actor, a Blueprint gameplay-root actor, and a Player Start. The running game reuses the saved gameplay root, lighting, camera, and environment settings. The world-layout actor also owns an explicitly labelled editor-only composition preview: route/deployment geometry, obstacles, landmarks, and representative teams remain visible before Play, then hide while the runtime collision geometry and Mass population take authority.

This division is intentional: designers can inspect and change composition and gameplay definitions in Blueprint and Data Assets, while the high-volume entity loops remain in compiled Mass processors and bounded sample backends.

Find the editable content

Enable **Show Plugin Content** in the Content Browser, then open `/MassForge/MassForge/L_MF_ShowcaseMenu` or one of the maps below.

Example	Map	Blueprint gameplay root	Blueprint world layout
Tower Defense	`/MassForge/MassForge/MassForgeShowcases/Maps/L_MF_TowerDefense`	`BP_MF_TowerDefense_WorldRoot`	`BP_MF_TOWER_DEFENSE_WorldLayout`
Third-Person Horde Defense	`/MassForge/MassForge/MassForgeShowcases/Maps/L_MF_ThirdPersonHordeDefense`	`BP_MF_ThirdPersonHordeDefense_WorldRoot`	`BP_MF_WIZARD_HORDE_WorldLayout`
Third-Person Horde Defense - VAT	`/MassForge/MassForge/MassForgeShowcases/Maps/L_MF_ThirdPersonHordeDefenseVAT`	`BP_MF_ThirdPersonHordeDefenseVAT_WorldRoot`	`BP_MF_WIZARD_HORDE_VAT_WorldLayout`
Battle Simulator	`/MassForge/MassForge/MassForgeShowcases/Maps/L_MF_BattleSimulator`	`BP_MF_BattleSimulator_WorldRoot`	`BP_MF_BATTLE_SIMULATOR_WorldLayout`
Survivor Roguelite	`/MassForge/MassForge/MassForgeShowcases/Maps/L_MF_SurvivorRoguelite`	`BP_MF_SurvivorRoguelite_WorldRoot`	`BP_MF_ARCANE_STAMPEDE_WorldLayout`

The shared editable gameplay definitions are under `/MassForge/MassForge/BlueprintExamples/Data`. They include the three attribute schemas, entity and projectile configs, damage definitions, instant and persistent effects, targeting policy, and abilities used by the examples.

Plugin content is a versioned reference. Duplicate an asset into `/Game` before changing it for a real project, then assign the duplicate on the project's Blueprint root.

What exists before Play

Select the `MF_AUTHORED_*_WORLD_LAYOUT` actor in the World Outliner. Its inherited Blueprint components contain the ground, routes or deployment markers, obstacles, landmarks, representative team staging, lighting, atmosphere, post process, overview camera, and concise teaching labels. Change **Showcase**, toggle the labels, or use **Rebuild Authored Layout** to rebuild the saved editor composition. The preview geometry has **Hidden in Game** enabled by design: it explains the level before Play without duplicating, flickering against, or pretending to own the runtime Mass presentation.

Select `MF_AUTHORED_*_GAMEPLAY_ROOT` to inspect the definitions used by that game. The **Mass Forge Showcases > Authoring** categories expose schema, Entity Config, ability, damage, projectile, and effect references. The root is the actor the GameMode reuses when Play begins; the GameMode does not spawn a hidden replacement.

Open the root Blueprint's Event Graph. Every published root contains a visible, commented startup flow:

- Tower Defense calls **Initialize Example** from **Event BeginPlay**.
- The other four roots call **Start Mode** with the map's explicit mode preset.
- **On Tower Defense Example Ready** or **On Showcase Example Ready** is an empty project extension event for UI, audio, tutorials, objectives, or game-specific rules.

The startup calls are idempotent and retain a native fallback for C++ only or blank-map use. Automation rejects a public root whose graph, preset, comments, extension event, or editable definition references are missing.

The Player Start is also saved and labelled. Representative friendly and hostile formations are present before Play to communicate scale, routes, and faction intent. They are previews rather than simulated entities and are replaced by the real Mass population when Play begins. The gameplay-root Blueprint and editor monitor remain the sources for live entity state.

Blueprint and C++ ownership boundary

Responsibility	Blueprint/Data Asset	Native backend
Saved editor composition, light, camera, teaching markers	Yes	Reusable component implementation
Editor-only representative teams and route preview	Blueprint component instances	Hidden and replaced by runtime Mass presentation
Attribute names, defaults, bounds, and stable slots	Yes	Fixed-capacity fragments and lookup
Entity/projectile trait composition	Yes	Trait build implementation
Damage, effects, targeting, abilities, cooldowns, costs	Yes	Validation and atomic execution
Per-example definition references and balance inputs	Yes	Safe startup and fallback diagnostics
Mass population iteration and fixed-step combat	No	Yes
Batched skeletal/VAT/HISM presentation updates	Profile/material assets	Yes
Project-specific UI, objectives, progression, and controls	Replaceable sample Blueprint/native surface	Sample implementation

Mass Forge does not pretend that a Blueprint actor is simulating thousands of entities one by one. The public Blueprint and Data Asset surface owns authoring and orchestration. The C++ path owns the operations where Blueprint iteration would be misleading, unsafe during Mass processing, or unnecessarily expensive.

Make a project-owned variant

1. Duplicate the example map and both referenced Blueprints into `/Game`.
2. Duplicate only the Data Assets you want to change from `/MassForge/MassForge/BlueprintExamples/Data`.

3. Assign those project-owned definitions on the duplicated gameplay root.
4. Edit the layout Blueprint or replace the placed world-layout actor with project scenery. Keep one actor tagged `MassForge.AuthoredEnvironment` if the sample GameMode should use its overview camera.
5. Press Play and open **Tools > Mass Forge Live Showcase**. Confirm the active, simulated, represented, and rendered counts mean what your design expects.
6. Open **Mass Forge Entity Inspector** and **Gameplay Event History** to verify attributes, tags, grants, effects, damage, and lifecycle events.
7. Run **Mass Forge Project Health** before packaging.

Example-specific authoring surfaces

Tower Defense

The Blueprint root references the public starter schemas, a combat-ready Entity Config, direct and mitigated damage definitions, and a persistent slow. Runtime duplicates the definitions before applying match-specific values, so the shipping reference assets are never mutated.

Horde Defense and Survivor Roguelite

The Blueprint root references the schemas, unit and projectile configs, four ability templates, and entity/Actor damage definitions. Horde Defense selects the skeletal character backend; its VAT twin selects the VAT backend while preserving the same gameplay setup. Survivor changes population, objective, and progression behavior while using the same Mass Forge contracts.

Battle Simulator

The same root exposes symmetric faction combat definitions. Melee and ranged/support actions resolve through the shared damage/effect/projectile systems; the fixed-step army loop stays native because it is the high-volume portion being demonstrated.

Runtime fallback

If a developer launches a supported GameMode in a blank map, the sample can still spawn a temporary root and environment so the map fails usefully instead of crashing. Published maps must not use that fallback. Automation requires exactly one saved Blueprint gameplay root and one saved authored layout in every public map, and runtime audits require `ActiveRoots=1` with `StaleRemoved=0`.

Rebuild the authored reference content

The checked-in `.uasset` files are the shipping source of truth. Maintainers can reproduce them in an isolated host in this order:

1. Run `Examples/MassForgeShowcases/Scripts/CreateShowcaseMap.py` through the rendered Unreal Editor Python runner to create or refresh the maps, layout Blueprints, gameplay-root Blueprints, and Player Starts.
2. Run `UnrealEditor-Cmd <Project> -run=MFGenerateShowcaseBlueprintGraphs` to author the five inspectable startup graphs.

3. Run `UnrealEditor-Cmd <Project> -run=MFGenerateBlueprintExamples -RebuildGeneratedAssets` to rebuild the Blueprint Quick Start, 27 Data Assets, and four material instances (32 assets total).
4. Run `UnrealEditor.exe <Project> -ExecutePythonScript=<absolute path to CreateBlueprintQuickStartMap.py> -Unattended -NoSound -NoSplash` to rebuild the authored learning map. Do not add `-NullRHI`: UE 5.6 actor placement consults the level viewport and its NullRHI hit-proxy path is not safe.
5. Run `MassForgeShowcases.Editor.StandaloneMaps` and `MassForge.Editor.BlueprintExamples.QuickStartGraph` before promoting generated assets.
6. Generate and save assets with Unreal Engine 5.6, the oldest supported version, then cross-load and test them in 5.7 and 5.8.

Continue with Playable Showcases, Blueprint Quick Start, and C++ High-Volume Integration.

EXAMPLES AND PRESENTATION

Character animation, VAT, and material-state evidence

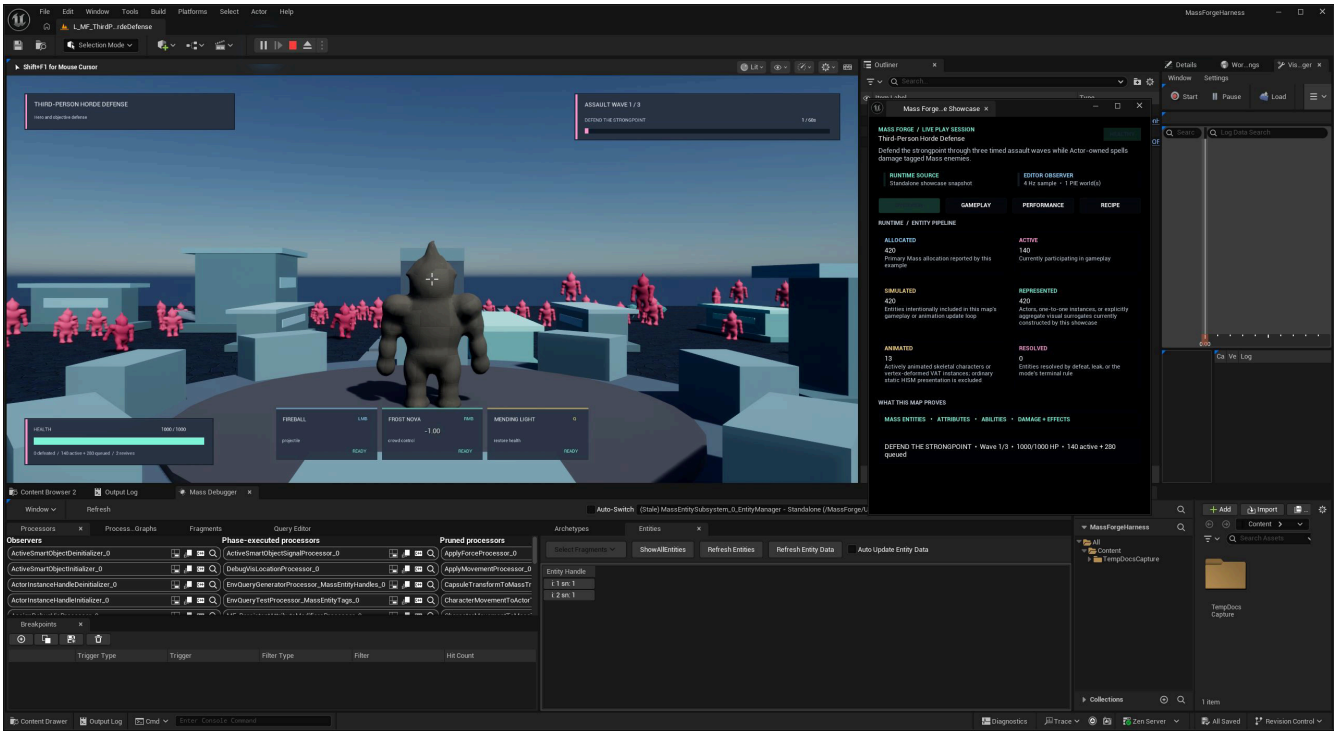
This document records the concrete implementation and verification evidence for the two published Third-Person Horde Defense presentation paths. It is not a claim that every Mass entity owns an Actor or skeletal mesh.

Release shape

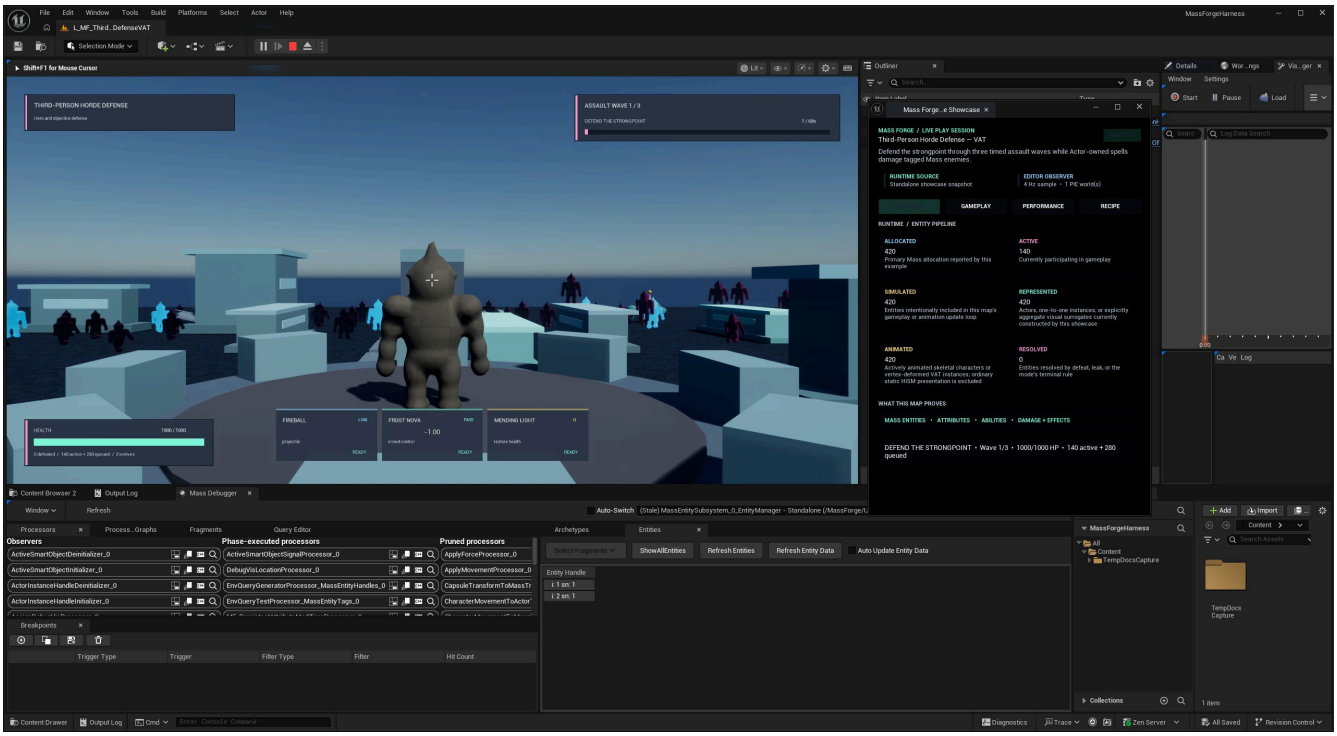
Map	Gameplay	Character presentation
`L_MF_ThirdPersonHordeDefense`	Shared 420-enemy, three-phase horde-defense rules	Rigged player plus a bounded twelve-enemy skeletal near tier; coherent single-piece static far tier
`L_MF_ThirdPersonHordeDefenseVAT`	The same population, controls, abilities, balance, obstacles, damage, and objectives	One batched VAT/material presentation path for represented enemies; no per-enemy Actor requirement

Both maps use `FMFCharacterPresentationFragment` as presentation-neutral Mass state. Gameplay authority remains in Mass Forge attributes, abilities, effects, tags, damage, and the sample's deterministic movement/objective code.

Matched runtime comparison



Skeletal presentation sample: Wave 1, `1 / 60 s`, default camera and untouched controls. The monitor reports 420 allocated, 140 currently active, 420 simulated, 420 represented, and 13 animated-the rigged player plus the bounded twelve-enemy skeletal near tier.



VAT presentation sample: the same Wave 1, `1 / 60 s`, camera, game rules, arena, and population. The monitor reports the same 420 allocated, 140 active, 420 simulated, and 420 represented entities, while all 420 enemies are animated through the batched VAT path.

Mass Forge...e Showcase
MASS FORGE / LIVE PLAY SESSION
Third-Person Horde Defense
Defend the strongpoint through three timed assault waves while Actor-owned spells damage tagged Mass enemies.
RUNTIME SOURCE
Standalone showcase snapshot
EDITOR OBSERVER
4 Hz sample • 1 PIE world(s)
GAMEPLAY
PERFORMANCE
RECIPE
RUNTIME / ENTITY PIPELINE
ALLOCATED
420
Primary Mass allocation reported by this example
ACTIVE
140
Currently participating in gameplay
SIMULATED
420
Entities intentionally included in this map's gameplay or animation update loop
REPRESENTED
420
Actors, one-to-one instances, or explicitly aggregate visual surrogates currently constructed by this showcase
ANIMATED
13
Actively animated skeletal characters or vertex-deformed VAT instances; ordinary static HISM presentation is excluded
RESOLVED
0
Entities resolved by defeat, leak, or the mode's terminal rule
WHAT THIS MAP PROVES
MASS ENTITIES • ATTRIBUTES • ABILITIES • DAMAGE + EFFECTS
DEFEND THE STRONGPOINT • Wave 1/3 • 1000/1000 HP • 140 active + 280 queued

The focused monitor crops make the accounting legible. "Allocated," "active," "simulated," "represented," and "animated" are separate workload statements; neither backend relabels allocation as visible or active enemies.

This packaged evidence frame is the accepted post-fix VAT presentation: whole connected bodies, correct frame-row orientation, purple/cyan palette variation, and no diagonal shards or unintended partial meshes. The player remains rigged; the enemy crowd uses batched VAT instances.

Original source and provenance

- Asset: **Mass Forge Arcane Sentinel**.
- Author/tool: original deterministic Blender 5.2.1 LTS source and scripts owned by the Mass Forge project. Morrow/Trellis was not used for this character pack.
- Rights: original project-owned work, cleared for redistribution with Mass Forge; no third-party or paid assets.
- Authoritative source: `Examples/MassForgeShowcases/ArtSource/CharacterAnimation/Generated/MF_ArcaneSentinel_Source.blend`.
- Rebuild/import scripts: `build_character_animation_pack.py` and `import_character_animation_pack.py` beside the source.
- Receipts: `CharacterAnimationAssetReceipt.json`, `CharacterAnimationBakeMetadata.json`, and `CharacterAnimationUnrealImportReceipt.json` under `Generated`.
- Unreal destination: `/MassForge/MassForge/MassForgeShowcases/CharacterAnimation`.
- Provenance gate: `Verify-ArtProvenance.ps1` matched the original source plus 14/14 UE 5.8 skeletal/VAT packages.

The model contains 3,512 source vertices and 7,020 rendered triangles, uses one section/material and one fixed topology for skeletal and VAT output, has no unweighted vertices, uses at most four influences per vertex, and recorded a maximum normalized-weight error of `4.8e-08`. The skeletal mesh has three LODs. Release 1 deliberately constrains VAT to one measured fixed-topology LOD and keeps explicit skeletal/static fallbacks.

Animation and VAT bake

The runtime contract contains seven states at 30 samples per second:

State	VAT frames	Loop	Authored event
Idle	0-29	yes	-
Move	30-53	yes	-
Attack	54-71	no	impact frame 10
Cast	72-95	no	release frame 13
Hit	96-105	no	-
Jump	106-129	no	apex 12, land 23
Death	130-159	no	completion frame 29

Victory and Stun source clips are also retained. All clips are in-place; gameplay transforms remain authoritative.

The position and normal data textures are each 4096x160, linear HDR, nearest-filtered, mip-free, non-streaming, and imported as RGBA16F. A direct `(vertex_id + 0.5) / 4096` UV lookup failed packaged visual QA even with full-precision UV storage: cooked output could render stretched diagonal shards. An intermediate split using low bits in U and high bits in V also failed because the FBX/Unreal V-axis convention changed the high field. The accepted `Split6x6TwoUChannels` mesh stores `low6 / 63` in UV1.U and `high6 / 63` in UV2.U; both V components are deliberately ignored. The VAT material rounds both U fields, reconstructs `high6 * 64 + low6`, and only then calculates the texel center. Blender EXR scanline zero is at the lower edge, so the generated material explicitly applies `OneMinus` to its frame V coordinate before sampling in Unreal; without that conversion Idle reads Death and Move reads Jump. Editor automation decodes every render vertex, requires the unique reconstructed set to contain all 3,512 source IDs despite render-vertex duplication, and rejects out-of-range or non-integer U fields. Full-precision UV storage remains enabled as defense in depth. Raw runtime texel payload is `5,242,880` bytes per texture (`10,485,760` bytes total) before platform/container overhead. The imported `.uasset` files are 4,513,552 and 5,033,637 bytes respectively; the fixed-topology VAT mesh package is 400,333 bytes. VAT bounds are extended by +/- `(200, 200, 220)` cm.

Half-precision comparison sampled 561,920 vertex poses:

- maximum position error: `0.087183 cm` (acceptance `0.1 cm`);
- mean position error: `0.0031324 cm`;
- maximum normal error: `0.0370094 degrees` (acceptance `0.1 degrees`); and
- mean normal error: `0.0058121 degrees`.

Shared runtime contract

`UMFCharacterPresentationProfile` validates required assets, compatible skeletons, all seven unique non-overlapping clips, texture properties/dimensions, mesh render data, a positive finite sample rate, and the exact versioned twelve-float payload:

1. phase offset;
2. play rate;
3. current start frame;
4. current end frame;

5. previous start frame;
6. previous end frame;
7. transition alpha;
8. animation state;
9. health ratio;
10. palette index;
11. material-state flags; and
12. effect intensity.

The material flags are Damage Flash, Frozen, Burning, Shielded, Selected, and Dissolving. Death deterministically resolves to Dissolving. Team palette and temporary states use instance custom data; the VAT crowd does not allocate a dynamic material instance per enemy.

Move playback remains in-place and presentation-only, but its cadence is scaled from authoritative gameplay velocity for the rigged player, bounded skeletal enemies, and VAT instances. This reduces visible foot sliding without moving collision or gameplay authority into animation, and it preserves the frozen twelve-float layout by multiplying the existing play-rate slot. Arena decisions remain on a bounded 10 Hz fixed step; presentation retains the previous/current authoritative positions and interpolates between them every rendered frame. Movement direction drives facing while travelling, stopped units select Idle and face their target, and attack wind-up owns facing until its authored release. This removes visible 10-FPS stepping and sideways travel without multiplying the Mass simulation workload.

Lifecycle presentation is explicit. Newly engaged units use a 0.28-second smooth scale-in. Defeated units retain the authored Death clip for a 1.35-second retirement window, with EffectIntensity driving dissolve only after 0.88 seconds. The animation transition value no longer controls opacity, so a death reads as a fall followed by a late fade rather than an unexplained despawn.

Automated evidence - UE 5.8

The clean customer-style verifier passed on the i9-14900K Windows 11 25H2 reference machine:

- UE 5.8 Editor build;
- 9/9 Mass Forge showcase automation tests, warning-free;
- sample-boundary and character-art provenance gates;
- root menu plus all five published maps;
- Tower Defense 1,000-entity accepted VAT scale vista;
- Survivor Roguelite 1,000,000 allocated enemies with the explicitly bounded 1,200 simulated/represented pool; and
- the internal procedural-WPO fixture at 1K, 5K, 10K, and 25K, still labeled as procedural WPO rather than VAT.

At 28 simulated seconds both horde maps reported 420 allocated/simulated enemies, 413 represented survivors, seven defeated enemies, positive hero health, no game-over, zero obstacle penetrations, all seven animation-state evidence bits (`StateMask=127`), all six material-state evidence bits (`MaterialMask=63`), 61 Mass projectile launches, 61 matching impacts, and no spawn failure. The skeletal map reported `NearSkeletal=12 CharacterBackend=SkeletalMesh`; the twin reported `NearSkeletal=0 CharacterBackend=VertexAnimationTexture`. The full skeletal objective gate completed all three phases with a living hero and 385 launches/385 impacts.

Packaged Win64 cooking included 564 packages with zero reported cook warnings/errors. Separate packaged skeletal and VAT runs loaded all cooked assets, retained the complete 420-entity population, selected their intended backends, remained at positive health with `GameOver=0`, and reported zero penetrations or spawn failures. `Verify-CharacterPresentationPackage.ps1` makes this two-map package gate repeatable after `Verify-Showcases.ps1 -KeepWorkspace`.

Representation accounting

The release evidence distinguishes Mass allocation/simulation from rendering. It does not add the rigged player to the enemy counts.

Observation	Allocated enemies	Simulated enemies	Represented enemies	Skeletal enemies	VAT enemies	Static far enemies	Representation-culled
Skeletal map, initial	420	420	420	12	0	408	0
VAT map, initial	420	420	420	0	420	0	0
Either map, gameplay audit after seven defeats	420	420	413	backend-specific survivors	backend-specific survivors	backend-specific survivors	0

The skeletal map additionally renders one rigged player. The VAT map renders its enemy population through one Hierarchical Instanced Static Mesh component/material batch rather than one Actor or dynamic material instance per enemy. The audit records all seven gameplay-driven animation states and all six combinable material states; defeated instances are retired rather than counted as live represented enemies.

Visual and performance observation

The current packaged visual evidence is `.ci-artifacts/CharacterAnimationVAT/Evidence/ThirdPersonHordeDefenseVAT_VAxisFixed.jpg`. A prior capture exposed that valid vertices were reading vertically mirrored animation rows: living idle enemies used Death poses and moving enemies used Jump poses. After adding the explicit EXR-to-Unreal V conversion and rebuilding the material, an independent controls-only release QA pass observed initial spawn through assault wave three. Living purple/cyan enemies remained upright connected whole bodies, advanced toward and generally faced the player, and showed no diagonal shards, false death poses, partial bodies, or unexplained hovering. The UI read `ASSAULT WAVE` and `DEFEND THE STRONGPOINT`, Escape returned to the hub, and the packaged application closed normally. Projectile release timing was not visually asserted from the wide shot; it remains covered by the deterministic launch/release/impact audit instead.

The deterministic Blender 5.2.1 source audit is recorded at `.ci-artifacts/CharacterAnimationVAT/Evidence/CharacterAnimationQualityAudit.json`. Across all nine clips it reports zero root drift, zero loop-seam delta, zero one-shot final-pose delta, finite deformation, one connected surface, normalized weights, no unweighted vertices, no more than four influences, maximum adjacent-frame motion of `78.9011 cm` below the explicit `85 cm` pop guard, and maximum planted-foot drift of `8.8095 cm` below the `12 cm` authored-loop guard. Visual QA remains the acceptance source for unintended self-intersection and bounds artifacts that this numeric audit cannot fully classify.

`Verify-CharacterPresentationPackage.ps1 -SkipBuildCookRun -CaptureVATProfile` produced a reproducible packaged Development observation on Windows 11 Pro 25H2, Intel Core i9-14900K, and NVIDIA GeForce RTX 5090. The VAT horde ran at 1600x900 with 420 allocated, simulated, represented, and VAT-animated enemies plus one rigged player. After discarding the first 100 of 600 captured frames:

Metric	Average	Median	95th percentile	Maximum
Frame time	6.621 ms	5.145 ms	11.983 ms	19.788 ms
Game thread	4.534 ms	3.561 ms	8.979 ms	17.195 ms

Metric	Average	Median	95th percentile	Maximum
Render thread	4.979 ms	3.365 ms	10.398 ms	12.378 ms
GPU	0.688 ms	0.666 ms	0.873 ms	1.177 ms
RHI draw calls	183	183	183	183
RHI primitives	37,765	37,765	37,765	37,765

The CSV and machine-readable summary remain under `.ci-artifacts/CharacterAnimationVAT/PackagedWin64`. This is a reference-machine observation, not a minimum-hardware guarantee and not a claim that one million enemies are visible or animated. The enemy VAT mesh is 7,020 triangles; the two uncompressed runtime VAT payloads total 10,485,760 bytes.

Measured boundaries

- “One million” in Survivor Roguelite means one million allocated enemy Mass entities, not one million visible, simulated, or VAT-animated enemies.
- The skeletal map does not create 420 skeletal Actors. It uses one rigged player and a bounded twelve-enemy skeletal tier; distant enemies remain an instanced presentation tier.
- The VAT map is genuine baked position/normal animation. The older `L_MF_VertexAnimation` map is a separate internal procedural-WPO regression fixture and is not linked from the customer launcher.
- A common data contract and palette rules are shared, while the skeletal stylized material and VAT deformation material remain separate masters because their vertex paths are materially different.
- Performance numbers describe the exact packaged reference-machine run above; customers should profile their own content, platform, resolution, population, and material complexity.

ARCHITECTURE, SCALE, AND PERSISTENCE

Mass Forge architecture guide

> **INFO - Ownership boundary**
>
> Mass Forge owns generation-safe entity handles and the attributes, effects, damage, tags, abilities, projectiles, queues, persistence, and diagnostics built on them. A game project owns spawning policy, movement/navigation, AI decisions, factions, player framework, presentation, UI, balance, and any supported multiplayer model.

Mass Forge adds attributes, effects, damage, abilities, and inspection to Unreal Engine Mass entities without making Actors or per-entity UObjects authoritative. This guide explains which layer owns each responsibility, how data reaches runtime fragments, and which entry path to use when extending the plugin.

System at a glance

```
Project settings + validated Data Assets
                 |
                 v
        Entity Config traits
                 |
                 v
    compact fragments / shared graphs
         |                |
         v                v
 automatic Mass processors  world subsystems and bounded queues
         |                |
         +--------+--------+
                  v
       committed entity state and result events
                  |
                  v
  optional Actors, UI, audio, VFX, AI, and inspection
```

The upper layers compile designer-facing names and asset references. The entity-processing layer uses stable slots, fixed-capacity storage, and numeric iteration. Presentation consumes results afterward and is never the source of simulation truth.

Ownership boundaries

Concern	Authoritative owner	Important boundary
Attribute definitions	Global schema Data Assets selected in project settings	A stable domain/name identity compiles to a runtime slot; deployed renames require migration
Per-entity values	Mass fragments	Actors and UI are views or request sources, not value owners
Derived values and regeneration	Automatic Mass processors	Authoring data is compiled before entity iteration
Instant transactions	Attribute subsystem	All modifiers and counted-tag changes for one target commit atomically
Persistent lifetime	Persistent Effect subsystem plus compact entity fragments	The world subsystem owns definitions, contexts, handles, and deadline scheduling
Damage policy and calculation	Damage subsystem	Project policy may veto; committed attribute/tag changes use the shared transaction path
Mass projectiles	Projectile fragment, processor, and subsystem	Compact motion remains on Mass entities; optional damage commits only after a queued impact reaches a Mass-idle boundary
Ability grants, clocks, and lifecycle requests	Ability fragment plus Ability subsystem	One shared bounded FIFO handles Activate/Grant/Revoke/Cancel/Interrupt; one activation transaction spans its source and at most one target
Durable entity persistence	Entity State subsystem plus project-owned SaveGame	Format 4 commits attributes, exact tags, grants, cooldowns, charges, regeneration timing, active Persistent Effects, and one active cast/channel together; external lifecycle targets require an explicit stable-reference remap
Unsafe external calls	Deferred-command coordinator	Queue acceptance is not execution success; completion events carry the final result
Presentation and ordinary Actors	Blueprint libraries, receiver/provider interfaces, and project code	Representation may appear, disappear, or change without changing Mass identity
Debugging	On-demand inspection subsystem and editor-only inspector	No continuously mirrored debug UObject is stored per entity

Modules and folder layout

- `MassForge` is the Runtime module. Public headers contain the supported C++ surface; private code contains subsystem, processor, trait, and automation implementations.
- `MassForgeEditor` is Editor-only. It owns schema and gameplay-asset Details customizations, the Entity Config wizard, Project Health, the Entity Inspector, the Event History browser, and editor automation.
- `MassForgeSamples` is the shipped Runtime showcase module. It owns the six playable sample modes and their reusable sample-facing presentation layer; it depends on the core while the core never depends on it.
- `MassForgeSamplesEditor` is the shipped Editor showcase companion. It owns the Showcase Browser and play-session telemetry monitor described in `SHOWCASES_GUIDE.md`.
- `Content` is optional and reserved for curated, namespaced distributable content. Core runtime behavior does not depend on example assets.

- `Config` contains packaging filters. `Docs`, `Resources`, the README, and changelog are shipped and hash-audited with the plugin. Maintainer-only `Scripts` and `Examples` verification fixtures remain in the source repository and are excluded from customer packages.

The runtime depends on Unreal's MassGameplay family. Simulation LOD interoperability uses the engine's `MassLOD` module; Mass Forge does not define a competing LOD classifier.

Schemas and attributes

`UMF_MassForgeAttributesSchemaSettings` selects one global schema for each storage domain:

- `UMF_VitalAttributesSchemaData` maps up to 16 Vital attributes.
- `UMF_CombatAttributesSchemaData` maps up to 48 Combat attributes.
- `UMF_SupportAttributesSchemaData` maps up to 32 Support attributes.

Each schema entry has a stable slot, name, display metadata, default, and optional clamp. `UMF_AttributesTrait` creates `FMF_VitalAttributesFragment`, `FMF_CombatAttributesFragment`, and `FMF_SupportAttributesFragment`, then copies schema defaults into the fixed arrays when the Entity Config template is built.

Blueprint calls use `FMFAttributeId` domain/name identities. C++ high-volume code compiles those identities into generation-checked `FMFCompiledAttribute` bindings before iteration and operates through numeric slots. Refreshing schemas increments the generation and invalidates old bindings instead of silently redirecting them.

Opt-in fragments and traits

Trait	Runtime storage added	Use it when
`UMF_TransformTrait`	Unreal's standard `FTransformFragment`	The entity is spawned, placed, targeted, or presented through the Blueprint-first workflow
`UMF_AttributesTrait`	`FMF_VitalAttributesFragment`, `FMF_CombatAttributesFragment`, `FMF_SupportAttributesFragment`	The entity owns any Mass Forge attribute
`UMF_GameplayTagTrait`	`FMF_GameplayTagFragment`	The entity needs counted tags, gates, immunity, or effect-owned tags
`UMF_AbilityTrait`	`FMF_AbilityFragment`	The entity grants or activates abilities, cooldowns, groups, charges, casts, or channels
`UMF_PersistentEffectsTrait`	`FMF_PersistentEffectsFragment`, `FMF_PersistentAttributeModifiersFragment`	The entity receives finite/infinite effects, stacking, periodic work, or reversible modifiers
`UMF_RegenerationTrait`	`FMF_RegenerationFragment`	The entity runs compiled passive regeneration or degeneration rules
`UMF_DerivedAttributesTrait`	`FMF_DerivedAttributesSharedFragment`	The archetype evaluates a shared compiled derived-attribute graph
`UMF_HighVolumeEffectsTrait`	`FMF_HighVolumeEffectCommandFragment`, `FMF_HighVolumeEffectSetSharedFragment`	C++ producer processors apply/remove a small context-free effect pair across large populations without lookup or allocation
`UMF_ProjectileTrait`	`FMF_ProjectileFragment`	The entity represents a Mass-owned projectile with compact launch delay, travel, arc, lifetime, impact, and presentation-channel state

Traits are deliberately opt-in. The Entity Config wizard supplies dependency-safe common presets, but profile-backed traits still require project-specific validated assets. Exact fragment bytes and preset totals are in the memory and capacity guide.

Authored Data Assets

- Attribute schemas define identity, storage, defaults, bounds, and editor metadata.
- `UMF_InstantEffectData` defines an ordered, atomic set of attribute and counted-tag mutations.
- `UMF_PersistentEffectData` adds lifetime, period, stacking, immunity, cancellation, granted tags, and reversible modifiers around optional Instant Effects.
- `UMF_DamageDefinitionData` defines presentation-neutral Health, Shield, power, mitigation, critical, gating, and death-handling policy.
- `UMF_AbilityData` defines built-in and custom requirements, costs, cooldowns, charges, targeting policies, instant/persistent payloads, an optional execution plan, and optional cast/channel behavior.
- `UMF_RegenerationProfileData` compiles passive linear rules into each participating entity.
- `UMF_DerivedAttributeProfileData` compiles a dependency-sorted formula graph into const-shared archetype data.
- `UMF_AttributeSnapshotMigrationData` declares explicit saved-attribute rename, move, or removal steps between schema versions.
- `UMF_AbilityTargetingPolicy` and its built-in range implementation validate a selected target before reservation or commit.
- `UMF_AbilityCondition` provides ordered, fail-closed, project-authored Blueprint/C++ admission rules that are re-evaluated at commit without adding per-entity objects.
- `UMF_AbilityExecutionPlan` provides bounded fail-closed Blueprint/C++ transaction descriptions; `UMF_StaticAbilityExecutionPlan` supplies data-only activation and channel-pulse payloads.
- `UMF_EffectMagnitudeCalculation` provides a shared, fail-closed Blueprint/C++ formula after built-in parameter/capture resolution; reversible persistent contributions snapshot its finite output, while periodic Instant Effects recalculate per pulse.

Data validation and Project Health reject malformed IDs, missing attributes, non-finite numeric values, incompatible references, duplicate durable Primary Asset IDs, and unsupported migration graphs before release.

Runtime subsystems

Subsystem	Responsibility
`UMF_AttributeSubsystem`	Schema caches, entity/Actor resolution, attribute access, guarded custom magnitude evaluation, atomic Instant Effects, versioned snapshots, compiled bindings, bounded restore/attribute/effect queues, and change/completion publication
`UMF_GameplayTagSubsystem`	Counted-tag access, mutation, hierarchical queries, and tag-change publication
`UMF_PersistentEffectSubsystem`	Apply, stack, query, remove, rollback, reversible modifiers, periodic execution, deadline scheduling, and lifecycle events
`UMF_DamageSubsystem`	Damage validation/calculation, project policy providers, atomic commit, queues, death policy, and bounded destruction requests
`UMF_ProjectileSubsystem`	Bounded projectile launch, renderer-neutral snapshots, impact queuing, optional impact-driven damage, cleanup, and resolution events

Subsystem	Responsibility
`UMF_AbilitySubsystem`	Grants, built-in/custom requirements, target policies, bounded execution-plan evaluation, activation, cooldowns, charges, casts/channels, AI tickets, queues, rollback, and lifecycle events
`UMF_EntityStateSubsystem`	Versioned durable-state capture, complete validation, atomic attribute/tag/ability restore, remaining-time reconstruction, and bounded restore completion
`UMF_DeferredCommandSubsystem`	The only gameplay queue/scheduler coordinator and its fixed cross-system phase order
`UMF_EntityInspectionSubsystem`	On-demand unified snapshots and bounded project-supplied inspection sections
`UMF_GameplayEventSubsystem`	Opt-in normalization, bounded history, entity watch filtering, and project-defined debug/telemetry events without changing legacy delegates
`UMF_NetworkAuthoritySubsystem`	Session identity, authoritative entity resolution, server definition catalog, and fail-closed request policies

Blueprint function libraries provide convenient static entry points, but state and transaction ownership remains in these world subsystems.

Owner-bound Player Controller components form the connection edge. `UMF_NetworkRequestComponent` authenticates and dispatches bounded client mutation requests; its optional bounded Ability-prediction ledger emits presentation-only intent and reconciles reliable authority receipts/completions without changing client gameplay state. `UMF_NetworkStateComponent` captures project-selected entities into an owner-only Fast Array containing filtered/quantized stable attributes, counted tags, ability/cooldown/lifecycle state, and persistent effects. It emits an immediate baseline when relevance begins, including for a late join, then refreshes a bounded round-robin subset and advances revisions only on semantic post-quantization changes.

Automatic Mass processors

All Mass Forge processors run in PrePhysics and use marked, verifier-enforced hot loops:

1. `UMF_HighVolumeEffectsProcessor` consumes one bounded precompiled Apply/Remove mailbox per opted-in entity before derived evaluation.
2. `UMF_DerivedAttributesProcessor` evaluates the const-shared graph in compiled dependency order.
3. `UMF_RegenerationProcessor` applies compiled passive rules after derived values and after Unreal's Simulation LOD timing update.
4. `UMF_PersistentAttributeModifiersProcessor` reconciles external Mass writes and reapplies active additive, multiplicative, and priority-override aggregates after regeneration.

`UMF_ProjectileProcessor` is an independent PrePhysics processor. It advances each `FMF_ProjectileFragment` through launch delay and bounded travel, then queues one impact record with `UMF_ProjectileSubsystem`. The subsystem resolves the optional damage payload only after Mass processing is idle, so visible travel and gameplay impact share one authoritative lifecycle without dispatching delegates or carrying UObject payloads inside the entity hot loop.

Entity loops may use only fragment views, stable slots, fixed-capacity records, and bounded numeric scans. Name lookup, asset loading, dynamic containers, allocation, reflection, logging, and event dispatch are excluded from these loops by the release verifier.

Custom magnitude assets belong to the direct/queued transaction path and never execute in a Mass processor hot loop. The High Volume Effects Trait rejects them at compilation; high-volume producers must supply already compiled numeric operations.

Direct, queued, and processor paths

Caller situation	Correct path	Result contract
Blueprint or game-thread C++ while Mass is idle	Direct Blueprint library or subsystem operation	Returns the final structured result synchronously
Callback, timer, async completion, or any caller that may overlap Mass processing	Corresponding bounded queue operation	Returns an acceptance receipt; observe the matching completion event for the final result
A Mass processor already holding fragment views	Compile IDs before execution and use the documented compiled fragment helpers	No asset lookup or delegate broadcast inside entity iteration
A producer processor applying/removing the same context-free effect across a large population	Cache the const-shared high-volume effect index and use the entity command/completion mailbox	One atomic command in flight per entity, explicit back-pressure, no per-entity event dispatch
Ordinary Actor that is represented by Mass	Actor convenience adapter	Resolves the full Mass generation and then uses the same entity operation
Ordinary Actor that is not a Mass entity	Effect receiver/provider interface where supported	Project-owned state remains a separate transaction boundary

Direct operations reject `MassIsProcessing`. Queued operations are bounded and revalidate entity generation plus stable definition identity at execution. Never treat a nonzero request ID as proof that gameplay was applied.

Deferred phase order

On each safe, unpaused world tick, `UMF_DeferredCommandSubsystem` snapshots and executes work in this order:

1. Entity State Restores
2. Attribute Snapshot Restores
3. Attribute Changes
4. Raw counted Gameplay Tag Changes
5. Instant Effects
6. Persistent Effects (queued Apply/Remove, then scheduled lifecycle work)
7. Projectile Impacts
8. Damage
9. Death Handling
10. Ability Lifecycles
11. Ability Requests (one FIFO for Activate, Grant, Revoke, Cancel, and Interrupt)

Work enqueued into the current or an earlier phase waits for the next coordinator tick. This bounds re-entrant callbacks and makes same-boundary outcomes deterministic. Persistent periods and expiration, damage/death publication, final channel pulses, and new activations follow the complete deterministic ordering contract.

Transactions and events

An Instant Effect is atomic for one Mass entity. An Ability activation may atomically span its source and one target, including costs, cooldowns, charges, tags, and authored effects. Arbitrary target arrays, separate queue entries, later channel pulses, ordinary Actor receiver state, and post-damage entity destruction are intentionally separate operations.

Subsystems finish validation and commit before publishing observable events. Re-entrant listeners therefore see the completed supported transaction, and rolled-back operations publish no provisional gameplay changes. Exact boundaries and partial-success guidance are in the transaction and event contract.

`UMF_GameplayEventSubsystem` is an opt-in observer over those native subsystem delegates. It copies normalized records into one bounded world-level circular history, can restrict capture to 64 generation-checked source/target handles, and accepts bounded project-defined AI/mind breadcrumbs. It does not add entity fragments, participate in transactions, or become gameplay authority. See the gameplay event history guide.

Multiplayer networking is outside the initial supported release. The gameplay subsystems remain world-owned and deterministic, but a customer project must supply and validate its own authority, replication, relevance, prediction, and late-join model before using Mass Forge state in a networked game.

Identity, representation, and lifetime

`FMFEntityHandle` carries both Mass index and serial number. Every public operation and deferred request validates the full generation, preventing a destroyed entity's queued work from reaching a recycled index. Actor representation is a temporary association; attaching, detaching, or changing LOD does not transfer ownership away from the Mass fragments.

World replacement invalidates world-local handles, requests, clocks, and subsystem metadata. The focused Attribute Snapshot persists only configured attributes; durable Entity State format 4 additionally preserves exact tags, grants, cooldowns, charges, regeneration partial intervals, active Persistent Effects, and one active cast/channel as portable gameplay time. Restored effects and lifecycles receive fresh handles; effect self-source context rebinds, external effect-source handles are cleared, and an external lifecycle target requires a project-owned stable key plus its recreated destination-world handle. Formats 1-3 remain supported inputs under their documented temporal-state rules. See the durable persistence guide, entity lifecycle safety, and time, travel, and save contract.

Extension boundaries

- Add project attributes through schemas, not by renaming compact slots in runtime code.
- Add reusable gameplay through validated Data Assets and existing transaction APIs.
- Add faction, ownership, safe-zone, or friendly-fire rules through the Damage Policy Provider.
- Add client ownership, loadout, rate-limit, and match-state authorization through a Network Request Policy; authorization remains separate from gameplay dispatch.
- Add project target discovery through the bounded Ability Target Provider; add target validity through Targeting Policy assets.
- Add dynamic per-activation or per-pulse effect selection through bounded Ability Execution Plans; returned payloads remain inside the source-plus-one-target atomic transaction.
- Add AI intent or other "mind" data through an Entity Inspection Provider.
- Add ordinary Actor interoperability through the Effect Receiver interface/component.

- Add a custom Mass processor only when work genuinely belongs in an entity batch; compile all names/assets first and follow the C++ high-volume guide and processor contract.

Mass Forge does not own navigation, faction storage, AI decision-making, animation, project ownership rules, player progression, UI, pooling, or game-specific combat balance. These remain project systems connected through explicit adapters and result events. The networking stack supplies session identity, authorization, owner-bound mutation transport, explicit per-connection relevance, per-attribute filtering/quantization, replicated gameplay state, late-join baselines, and opt-in presentation-only Ability prediction. Normal and hostile dedicated/listen multi-process certification passes on every supported engine baseline with `80 +/- 20` ms latency/jitter emulation; raw packet fuzzing, packet-loss recovery, and bandwidth saturation remain project-specific validation boundaries.

Recommended reading path

1. Blueprint-only quick start
2. Starter attributes and schema editor
3. Instant Effects, Persistent Effects, and Damage
4. Abilities, lifecycles, and target selection
5. Durable persistence, deferred commands, ordering, and error codes
6. Gameplay event history, C++ high-volume integration, memory/capacity, and verification

Entity lifecycle safety

Mass Forge treats entity identity, Mass representation, and gameplay presentation as separate concerns. This contract defines what happens when entities are destroyed or recycled, Actors enter or leave representation, LOD changes, and a world shuts down.

Identity and generation rule

`Mass Forge Entity Handle` stores both the Mass entity index and serial number. The pair is the identity. An index alone is never sufficient.

- Every public operation revalidates the full index/serial pair against the operation's world.
- Destroying an entity invalidates every copy of its old handle.
- If Mass later reuses the same index, the serial number changes. An old request, ability target, persistent-effect handle, or Blueprint variable cannot address the replacement generation.
- Handles are world-local runtime references. Do not save, replicate, compare, or reconstruct them as durable IDs across world travel.

Project code must reacquire a handle after spawn, pooling/reactivation that creates a new Mass entity, representation reassignment, seamless travel, or load.

Queued requests

Queue acceptance transfers a generation-checked source/target handle and the authored definition identity into a bounded world queue. Acceptance does not reserve the entity or guarantee later success.

At execution, complete Entity State Restores, focused Attribute Snapshot Restores, Attribute Changes, counted Gameplay Tag changes, Instant Effects, Persistent Effect requests, Damage, Death Handling, and Abilities revalidate the identities they use. A destroyed or recycled entity produces the operation's explicit invalid-entity or invalid-target completion. It is never redirected to an entity that later occupies the same index. Queued restores own their copied payload; definition assets are revalidated against the stable ID captured when queued.

Separate requests remain independent transactions. If one request destroys or invalidates an entity, later requests for the old generation fail normally and keep their own request IDs.

Active ability lifecycles

Active casts and channels are keyed by the source's complete entity handle.

- If the source is destroyed, the next safe lifecycle pass removes the stored lifecycle and publishes one end event with `Source Invalid` and `Invalid Entity`.
- If a required target is destroyed before cast commit, commit fails atomically with `Invalid Target`: costs, cooldowns, charges, effects, and persistent effects do not commit.
- If a target becomes invalid during a channel, that pulse reports its normal target failure. The Ability Data Asset's Channel Failure Policy then continues, cancels, or interrupts the channel.

- A replacement generation never inherits the old source's lifecycle, cooldown state, grants, or target identity.
- Queued Ability Grant/Revoke retains the complete entity generation. Queued Cancel/Interrupt also captures the exact active lifecycle ID, so a delayed request cannot affect either a recycled entity generation or a replacement lifecycle on the same source.

Lifecycle observers should clear presentation on every end reason, including Source Invalid and Commit Failed. UI should not wait for an invalid source to become queryable again.

Persistent-effect handles

A persistent-effect handle combines a world-local effect instance ID with the owning generation-checked entity handle.

- Query or manual removal through a stale owner returns `Invalid Entity` and never searches a replacement generation.
- The persistent-effect subsystem performs invalid-owner retirement on every valid tick, including a zero-delta cleanup pass. Retirement removes all subsystem metadata and publishes one removal event with `Entity Invalidated`.
- Authoritative per-entity slots disappear with the Mass entity. Replacement entities begin with empty effect slots and cannot inherit grants or modifier handles.
- World teardown clears all remaining persistent metadata without gameplay callbacks.

Project-side maps keyed by persistent handles must remove entries when they receive any removal event or when a query reports an invalid owner.

Actor representation and LOD

Mass Forge attributes, tags, effects, abilities, cooldowns, charges, and lifecycles live on Mass fragments. Actor representation is an optional lookup and presentation route; it is not simulation authority.

- Removing an Actor-to-Mass mapping makes Actor resolution return `Not Represented` and clears its output handle. The existing Mass entity and its handle remain valid.
- Assigning the Actor to a different Mass entity makes the next resolve return the new complete handle. Previously resolved handles are not rewritten.
- Representation LOD changes that preserve the Mass entity do not change gameplay identity or state because core systems do not key state by Actor, mesh, visual LOD, or representation fragment.
- If a project implements an LOD or pooling transition by destroying one Mass entity and creating another, that is an identity change. Reacquire the new handle and deliberately initialize or restore state.

The base plugin does not destroy, hide, pool, or respawn represented Actors. Damage Definitions can request project deactivation or generation-safe Mass destruction, but presentation ownership stays with the host project. The published skeletal and VAT horde twins demonstrate that gameplay identity survives a presentation-backend change: the skeletal map uses one rigged player and a bounded near tier, while the VAT map uses one batched HISM presentation for represented enemies. The older procedural-WPO crowd harness remains an internal regression fixture and is never reported as VAT.

World teardown

Each world owns independent schemas, queues, request counters, lifecycles, AI tickets, persistent-effect metadata, tag-event buffers, and inspection providers. During subsystem deinitialization Mass Forge:

- drains pending Entity State Restore, Attribute Snapshot Restore, Attribute Change, Gameplay Tag, Instant Effect, Persistent Effect, Damage, Death Handling, and unified Ability-operation queues;
- clears active ability lifecycles, tracked AI requests, persistent-effect metadata, deferred events, providers, and transaction scratch state;
- resets world-local request and handle counters; and
- does not emit gameplay completion, lifecycle-end, or persistent-removal callbacks for discarded teardown state.

This no-callback rule prevents shutdown code from mutating a world that is being destroyed. A loading screen, pending-action UI, or save system must clear its own project state from world/level lifecycle events rather than waiting for Mass Forge completions.

The gameplay clock and the exact state that does or does not cross save/load and seamless-travel boundaries are defined in the time, travel, and save contract.

Safe project recipes

Pool presentation while preserving simulation

1. Keep the Mass entity alive.
2. Remove or change only its representation.
3. Retain the complete entity handle.
4. Re-resolve the Actor whenever presentation returns.

Attributes, effects, abilities, and identity remain intact.

Retire and later create a fresh entity

1. Stop submitting work for the old handle.
2. Destroy or deactivate the old Mass entity through the chosen death policy.
3. Clear UI, targeting, AI, and project maps for the old generation.
4. Create a new entity and acquire its full handle.
5. Initialize from traits or restore a validated stable-ID snapshot when continuity is intended.

Never copy only the old index and never assume recycled memory implies gameplay continuity.

Automated proof

The isolated-world suite cross-checks the contract at several boundaries:

- a queued damage request cannot touch the replacement created after its target is destroyed, including when Mass recycles the index;
- Actor representation can be removed and reassigned while authoritative entity-handle operations continue correctly;

- destroyed cast sources end once with Source Invalid, and destroyed cast targets fail commit without costs or effects reaching a replacement;
- persistent owner destruction retires metadata and emits Entity Invalidated on a zero-delta pass; and
- a request deliberately left pending at world teardown produces no gameplay completion callback.

All other world tests also destroy their isolated worlds after exercising real subsystem state, providing repeated teardown coverage across the supported engine matrix.

ARCHITECTURE, SCALE, AND PERSISTENCE

Durable entity-state persistence

Mass Forge provides a versioned, SaveGame-safe `Mass Forge Entity State Snapshot` for durable gameplay state that belongs to one entity. Projects still own the surrounding `USaveGame`, slot naming, player/profile identity, world population records, and entity recreation policy.

Format 4 scope

Format version `4` captures:

- every configured Vital, Combat, and Support attribute by stable domain/name identity;
- exact counted Gameplay Tags when the source archetype has the Gameplay Tag fragment;
- ability grant IDs when the source archetype has the Ability fragment;
- remaining individual cooldown seconds for each grant;
- raw stored charges and remaining time until the next charge recovery; and
- active shared/global cooldown group IDs and their remaining seconds;
- the unconsumed partial interval for every compiled regeneration/degeneration rule, identified by its stable target attribute and guarded by its authored tick interval; and
- active Persistent Effects, including remaining duration, periodic phase and earned backlog, stack identity/strength, completed-period count, snapshotted tags and resolved attribute contributions, reversible attribute bases, and portable application context; and
- one active cast/channel phase, including guarded ability identity/version, execution type, remaining phase time, earned channel backlog, completed pulse count, committed state, and portable target relationship.

The format never stores a destination Mass entity handle, Actor pointer, request ID, lifecycle ID, persistent-effect handle, hit result, or absolute `UWorld` timestamp. Restoring in another world rebuilds deadlines from saved remaining gameplay seconds and allocates fresh world-local Persistent Effect and Ability Lifecycle handles.

Format 4 does not serialize queued work, AI tickets, event history, representation, navigation, or project-owned state. AI tickets and async lifecycle observers are world-local orchestration; reacquire or rebind them after restoration.

Formats `1`, `2`, and `3` remain readable. Because format 1 predates regeneration timing, restoring it resets every destination regeneration remainder to zero rather than retaining unrelated live timing. Format 2 retains regeneration timing but has no Persistent Effect section. Formats 1 and 2 reject a destination with active Persistent Effects, and formats 1-3 reject a destination with an active cast/channel; a legacy payload never erases temporal state it cannot describe. A format-2-or-newer regeneration restore requires the same unique target set and tick intervals as the captured profile; a changed profile returns **Regeneration Profile Mismatch** before any state changes.

Released format matrix

Entity-state format	Durable sections	Compatibility rule on load
1	Attributes, counted tags, grants, individual/shared cooldowns, charges and charge recovery	Migrate the nested attribute payload when its schema is old. Regeneration remainder becomes zero. Active destination Persistent Effects or an Ability Lifecycle reject the restore.

Entity-state format	Durable sections	Compatibility rule on load
2	Format 1 plus regeneration remainder	The current regeneration target/interval profile must match. Active destination Persistent Effects or an Ability Lifecycle reject the restore.
3	Format 2 plus exact active Persistent Effects	Soft definitions and stable Effect IDs must still resolve and agree. An active destination Ability Lifecycle rejects the restore.
4	Format 3 plus one exact active cast/channel	Soft Ability identity, Save Compatibility Version, execution/timing guards, and any external target binding must agree. Existing supported destination state is replaced atomically.

Do not edit or blindly replace `FormatVersion` after deserialization. It describes which fields the saved record was capable of representing and selects the compatibility rules above. New saves use the current format automatically. Attribute schema evolution is separate: migrate the nested `Attributes` member through project-authored migration assets, without changing the outer entity-state format.

Portable Persistent Effect context

Each active effect stores its persistent definition and periodic definition as soft asset references guarded by their stable Effect IDs. It also stores logical Source ID, Ability ID, optional origin, up to 64 finite named scalar parameters, and up to 32 application-time attribute captures. Actor pointers and hit results are deliberately excluded.

`SourceEntity` is rebound to the newly restored destination only when the captured effect was self-sourced. An external source entity belongs to the old world and is therefore left unset; its logical Source ID and captured numeric inputs remain available. `TargetEntity` always becomes the destination. Save files must never assume that an old Persistent Effect handle will remain valid after restore-enumerate the restored entity's active effects when a new handle is needed.

Portable Ability Lifecycle targets

A targetless cast/channel uses the ordinary capture and restore nodes. A lifecycle that targeted the saved entity itself records **Saved Entity** and rebinds automatically. A lifecycle targeting another entity records **External Entity** and requires an explicit project-owned relationship key:

1. Call **Capture Mass Forge Entity State with Lifecycle Target Reference** and supply a stable key meaningful to the outer SaveGame, such as a party-member record ID-not a Mass index/serial.
2. Save the returned snapshot and the project's relationship records.
3. Recreate both entities in the destination world and resolve their new full-generation handles.
4. Build **Mass Forge Entity State Lifecycle Target Binding** with the identical key and the recreated target handle.
5. Call **Restore Mass Forge Entity State with Lifecycle Target** or its queued variant.

Missing keys, mismatched keys, unset targets, stale targets, and a binding that collapses an external relationship onto the restored source all fail before mutation. Mass Forge does not guess a target from spatial proximity, Actor representation, or a recycled entity index.

Every Ability asset exposes **Save Compatibility Version**, default `1`. Increment it when a behavior change must invalidate previously saved active phases. Restore also guards the stable Ability ID, execution type, cast time, channel duration, and channel period. A mismatch fails closed rather than continuing old timing under changed authored behavior.

Project-owned SaveGame envelope

Mass Forge deliberately provides the durable value type, not a universal `USaveGame` class. The owning game must retain the information required to recreate its population and relationships. A minimal C++ record can follow this shape:

```
USTRUCT(BlueprintType)
struct FProjectMassForgeEntityRecord
{
    GENERATED_BODY()

    UPROPERTY(SaveGame)
    FName EntityRecordId;

    UPROPERTY(SaveGame)
    FPrimaryAssetId EntityConfigId;

    UPROPERTY(SaveGame)
    FMFEntityStateSnapshot State;
};

UCLASS()
class UProjectWorldSaveGame : public USaveGame
{
    GENERATED_BODY()

public:
    UPROPERTY(SaveGame)
    int32 ProjectSaveVersion = 1;

    UPROPERTY(SaveGame)
    TArray<FProjectMassForgeEntityRecord> Entities;
};
```

`EntityRecordId` is a project identity, not an `FMFEntityHandle`. If an active lifecycle targets another saved entity, pass that target's record ID to **Capture Mass Forge Entity State with Lifecycle Target Reference**. Store whatever additional transform, spawning, ownership, faction, inventory, quest, or world-partition information is needed beside the Mass Forge state. Keep referenced Ability, Instant Effect, and Persistent Effect assets cooked and preserve their paths through Unreal redirectors when moving them.

Blueprint projects use the same envelope: create a project `SaveGame` Blueprint with a project version and an array of project entity records containing a stable record name, an Entity Config reference/ID, and **Mass Forge Entity State Snapshot**. The Mass Forge snapshot remains a normal Blueprint struct.

End-to-end load and migration sequence

Use this ordering for travel, slot loads, and released-save upgrades:

1. Deserialize the project `USaveGame` record. Treat a missing/corrupt project record as a project load failure; do not construct a default Mass Forge snapshot and call it restored state.
2. Upgrade the outer `ProjectSaveVersion` first. Perform project-specific unit conversions, archetype substitutions, or record-shape changes here.
3. Leave every Mass Forge `State.FormatVersion` unchanged. Reject formats outside `OldestSupportedFormatVersion` through `CurrentFormatVersion` rather than guessing.
4. Compare `State.Attributes.SchemaVersion` with the current Mass Forge project setting. Apply the complete ordered chain of **Mass Forge Attribute Snapshot Migration** assets to `State.Attributes`; replace only the nested member after each successful step. Stop the load if any step fails.
5. Recreate every destination entity from the project record's Entity Config/archetype information before restoring cross-entity relationships. Build a map from each `EntityRecordId` to its new full-generation `FMFEntityHandle`.
6. For a targetless or self-targeted lifecycle, use the ordinary restore path. For **External Entity**, resolve `State.AbilityLifecycle.TargetReference` through that map and build **Mass Forge Entity State Lifecycle Target Binding** with the identical reference plus the recreated target handle. Never substitute an index match or nearest entity.
7. At a guaranteed Mass-idle point, restore directly. If loading can overlap Mass processing, submit the queued restore, retain the positive request ID, and wait for its one terminal completion. Any rejected zero-ID receipt is a load failure that will not later complete.
8. Only after successful restoration, rebuild project-owned Actor representation, AI/state-machine bindings, targeting caches, UI observers, and persistence bookkeeping. Re-query active Persistent Effects and the active Ability Lifecycle because restored world-local handles are new.
9. On the next successful save, capture the current format and current attribute schema. Do not rewrite the user's old slot until the upgraded world has restored and passed project validation.

For large populations, recreate all entities and resolve the relationship map in bounded batches, then use the bounded queued restore API. A single entity-state transaction is atomic for one source plus its supported temporal state; a project save containing many entities is not one global Mass Forge transaction. If record 37 fails after records 1-36 restore, the project decides whether to discard the new world, retry, or report a partial project-load failure.

Failure recovery

- **Unsupported Snapshot Version**: keep the slot unchanged and route it through an older supported build or an explicit project converter. Never relabel the format.
- **Schema Version Mismatch**: run the missing attribute migration chain and retry with its output; do not restore the unmigrated values by slot.
- **Missing/Changed Persistent Effect or Ability definition**: restore the referenced asset path/redirector or perform an explicit project upgrade. Do not silently drop the active state.
- **Regeneration Profile Mismatch**: recreate the entity with the matching released profile or intentionally reset/rebuild the record in a documented project migration.
- **Missing Lifecycle Target Binding**: finish population recreation and relationship resolution before retrying.
- **Queue Full**: retain the project record, wait for capacity, and resubmit. A zero request ID promises no completion.
- **Invalid Entity**: discard the stale handle, resolve the intended record to its current generation, and retry only if the project record still identifies the same logical entity.

Blueprint workflow

1. At a Mass-idle checkpoint, call **Capture Mass Forge Entity State** with the source entity. Actor-owned graphs may use **Capture Mass Forge Entity State from Actor**. Use the target-reference variant when the active lifecycle points to another entity.
2. Require `Result == Success`, then place the returned struct in the project's SaveGame record beside its own stable entity/archetype identity.
3. On load, recreate or resolve the correct destination entity. Never save and replay the former index/serial handle.
4. Upgrade `Snapshot.Attributes` through the required **Mass Forge Attribute Snapshot Migration** assets when its schema version is older than the current project schema.
5. Call **Restore Mass Forge Entity State** only from a guaranteed Mass-idle point. Use **Queue Mass Forge Entity State Restore** when load orchestration can overlap Mass processing. Supply the explicit target-binding variant for an external active-lifecycle relationship.
6. For a queued restore, retain the positive request ID and match it against **On Entity State Restore Completed**. A rejected receipt has request ID `0` and produces no completion.

The represented-Actor restore nodes resolve the current entity generation at submission. A representation swap cannot redirect accepted queued work to another entity.

Exact restore semantics

Restore validates the complete payload before changing state. It then commits attributes, tags, grants, cooldowns, charges, regeneration timing, Persistent Effect ownership/scheduling, and the active Ability Lifecycle before publishing any change event. An attribute-change callback therefore observes complete restored tag, ability, regeneration, effect, and lifecycle state rather than a partial intermediate state. Restore itself does not emit synthetic effect-applied/removed or ability-lifecycle started/ended events; the whole-entity restore completion is the authoritative boundary. Re-query restored effect/lifecycle handles and rebind world-local observers after that boundary.

The `Includes Gameplay Tags` and `Includes Abilities` flags describe whether the captured archetype owned those fragments:

- when a section is included, the destination must have its matching fragment and the saved list replaces that section exactly;
- entries absent from an included list are removed;
- when a section was not present on the source, an optional matching fragment on the destination is cleared; and
- non-empty section data with its include flag disabled is rejected instead of being silently ignored.

Successful tag count changes publish the normal tag-change events in lexical tag order. Removed ability grants publish before newly added grants, each in lexical ID order. Cooldown/charge-only changes do not masquerade as grant changes.

Current attribute clamps remain authoritative. Ability definitions remain authoritative when a restored grant is later queried or activated: for example, charge queries clamp raw stored charges to the current definition's maximum. Stable Ability IDs must retain compatible meaning across releases or be upgraded by the project surrounding the Mass Forge snapshot.

Validation and bounded data

Restore rejects the complete transaction for:

- an unsupported entity-state or nested attribute format;
- an unmigrated attribute schema version;
- invalid, duplicate, retired, missing, or non-finite attribute values;
- invalid, duplicate, zero/overflowing, or over-capacity counted tags;
- empty or duplicate Ability IDs, over-capacity grants, or invalid cooldown/recovery times;
- invalid, duplicate, or over-capacity cooldown groups;
- malformed, duplicate, non-finite, over-interval, or over-capacity regeneration timing entries;
- a missing regeneration fragment or changed target/interval profile for a format-2-or-newer timing section;
- malformed, over-capacity, missing, or changed Persistent Effect definitions; invalid duration/period/backlog state; invalid portable context; missing resolved-modifier bases; or a saved effective value inconsistent with its reversible base and contributions;
- malformed lifecycle phase/timing/backlog/commit state, an absent or changed Ability definition/version, or an active lifecycle whose Ability is not in the saved grants;
- missing, mismatched, unset, stale, self-collapsed, or otherwise invalid external lifecycle-target bindings;
- missing destination fragments, a stale entity generation, Mass processing, lifecycle processing, or a nested transaction; and
- active casts/channels for formats 1-3. Formats 1 and 2 additionally reject active Persistent Effects because those formats cannot replace them exactly.

Capacities match the runtime fragments: 32 exact counted tags, 16 ability grants, 8 active cooldown groups, 8 regeneration rules, 16 active Persistent Effects, and 24 attributes with reversible Persistent Effect contributions. The attribute section remains bounded by the configured schema capacities.

Queued restore contract

The queue owns a deep copy of the snapshot and any lifecycle-target binding. **Max Queued Entity State Restores Per Frame** limits work per safe tick and **Max Pending Queued Entity State Restores** provides back-pressure. Entity-state restores run before narrower Attribute Snapshot Restores and before all ordinary gameplay mutation queues.

Submission validates format, schema, IDs, finite values, section consistency, capacity, and the supplied entity identity when safe access is available. Execution revalidates the complete generation, fragment compatibility, schema, and temporal-state preconditions. An accepted request receives exactly one terminal completion unless its world tears down first. Work enqueued from a completion callback waits for the next eligible phase.

Active Ability Lifecycle is expected only when a format-1/2/3 payload targets an active destination lifecycle. **Active Persistent Effects** is expected only for a format-1/2 payload targeting active effect state. A valid format-4 restore replaces destination effects and lifecycle state atomically with the saved sections.

Attribute-only alternative

Use the older **Capture/Restore Mass Forge Attribute Snapshot** nodes when the project intentionally wants to restore only attributes while leaving tags, grants, cooldowns, charges, regeneration, Persistent Effects, and active Ability Lifecycles unchanged. The detailed schema migration and partial-attribute rules remain in the attribute persistence guide.

Verified released-format scenarios

The packaged world regression serializes and deserializes the public struct through Unreal's tagged SaveGame archive before restoration. It covers format 1 plus a nested schema-2-to-3 Health rename, format 2 regeneration remainder, format 3 soft-referenced Persistent Effect state and exact timing, and format 4 soft-referenced externally targeted channel state with a project relationship remap. The strict release verifier executes those scenarios twice on UE 5.6.1, 5.7.1, and 5.8.2 together with Editor, Development Game, Shipping Game, package-integrity, and zero-warning gates.

This proves the Mass Forge payload and restore contract. Each customer project must still test its own `USaveGame` envelope, population recreation, migrations, relationship resolver, cooked assets, platform storage, and failure UI before release. The time, travel, and save contract defines the supported runtime boundaries. Multiplayer persistence remains outside the initial supported release.

ARCHITECTURE, SCALE, AND PERSISTENCE

Attribute persistence and schema migration

> **DANGER - Released identity changes require migration**
>
> Renaming an Attribute ID or moving it to another domain changes its durable identity. Once snapshots have shipped, never make that change in place without incrementing the schema version, authoring a complete migration path, and testing old-save restoration on a copy.

Mass Forge exposes portable snapshots instead of owning a game-specific SaveGame class. Projects decide when, where, and with what other game state each snapshot is stored.

Save stable gameplay identity, not a raw Mass index. Loading recreates an entity with a new generation-safe handle, optionally migrates the stored identities, remaps external targets, and atomically restores supported state; the pre-save handle is deliberately stale.

This guide covers the focused `Mass Forge Entity Attribute Snapshot` and its schema-migration assets. For a single transaction that also preserves exact counted tags, ability grants, individual/shared cooldowns, charge recovery, regeneration phase, active Persistent Effects, and one active cast/channel, use the durable entity-state persistence guide.

Attribute snapshot format version 1 is deliberately attribute-only. Loading it onto an existing entity leaves other systems unchanged; recreating an entity resets those systems to trait/project defaults. The full pause, dilation, travel, and reset policy is defined in the time, travel, and save contract.

Blueprint workflow

1. Call **Capture Mass Forge Attribute Snapshot** with a valid Mass Forge entity handle.
2. Store the returned `Mass Forge Entity Attribute Snapshot` in a project-owned SaveGame object, database payload, or other persistence layer.
3. Recreate or resolve the correct entity after loading.
4. If the saved `SchemaVersion` is older than the current project setting, apply the required **Mass Forge Attribute Snapshot Migration** assets in version order.
5. At a guaranteed Mass-idle lifecycle point, call **Restore Mass Forge Attribute Snapshot**. From callbacks, async handoffs, or other timing-sensitive Blueprint paths, call **Queue Mass Forge Attribute Snapshot Restore** instead.
6. For a queued restore, store the positive request ID and match it once through **On Attribute Snapshot Restore Request Completed**.
7. Branch on the direct result or queued terminal `Restore` result and handle every migration or restore failure explicitly.

Capture remains a direct read and returns **Mass Is Processing** during Mass processing; perform capture from a safe project lifecycle point. Direct restore has the same guard. The queued restore is the production-safe mutation path when timing can overlap Mass: submission validates the copied payload's binary/schema versions, emptiness, duplicate IDs, IDs, finite values, entity presence, and bounded capacity. A rejected receipt has request ID `0` and no completion. An accepted positive ID receives exactly one terminal completion unless its world ends.

Queued restores execute FIFO under **Max Queued Snapshot Restores Per Frame** and **Max Pending Queued Snapshot Restores**. They are the first sub-phase of the coordinator's public **Attribute Changes** phase, before queued raw attribute changes and counted Gameplay Tag changes. Therefore restore establishes saved base values and same-tick queued gameplay mutations apply afterward, independent of cross-category submission order. Work enqueued by a restore completion waits until the next coordinator tick.

Format contract

Format version `1` stores a project-owned `SchemaVersion` plus an array of stable attribute IDs and finite float values. Capture reads the schema version from **Project Settings > Mass Forge > Global Attribute Schemas > Persistence**. Capture order is deterministic: Vital, Combat, and Support domains, each in ascending stable schema-slot order. Unused capacity slots are not serialized.

Both fields of the reusable `Mass Forge Attribute ID`-domain and name-are SaveGame properties. This is required because Unreal applies SaveGame filtering recursively inside nested structs. The release automation performs an actual tagged-archive round trip and a rename migration, rather than treating the outer snapshot property's SaveGame flag as sufficient proof.

Restore resolves every saved ID by domain and name against the current schemas. This allows slot layout repair without reinterpreting a saved value. Current schema clamping is applied during restore, so a newly tightened valid range remains authoritative.

Slots are compact runtime locations, not save identities. An identity-preserving slot repair or deliberate offline layout change therefore needs no snapshot conversion: restore resolves each saved domain/name into the current slot. Do not change a deployed slot layout while affected entities are alive; live fragments already contain values in the old layout. Apply schema changes before creating/restoring entities, or recreate the affected entities and restore their identity-based snapshots. Any domain/name rename, move, or removal is a persistent identity change and requires the versioned migration workflow below.

Restore requires the binary format to equal the build's `CurrentFormatVersion` and the snapshot schema version to match the current project setting. `Unsupported Version` returns both `SnapshotFormatVersion` and `ExpectedFormatVersion`; `Schema Version Mismatch` returns both `SnapshotSchemaVersion` and `ExpectedSchemaVersion`. Both checks occur before any entity value changes, so a Blueprint can display an exact upgrade message instead of a generic load failure. Snapshots written before schema versioning was introduced load as schema version `1`.

Set **Oldest Supported Snapshot Schema Version** to the oldest integer schema version whose saves the current release promises to load. The **Mass Forge Project Health** dashboard requires a valid directed migration path from every version in that inclusive support window to the current **Snapshot Schema Version**. Set both values equal only when older saves are intentionally unsupported. See PROJECT_HEALTH_DASHBOARD_GUIDE.md for the exact coverage rule and remediation workflow.

Atomicity and failures

Restore validates the complete snapshot before changing fragment state. Unsupported versions, duplicate IDs, retired or renamed attributes, missing schemas/fragments, and non-finite values reject the whole operation. The result reports the failing attribute when applicable. Successful changes publish the normal attribute-change event with the supplied source name. For queued restores, all attribute-change events publish after the atomic state commit and before the one request-completion event.

The queue owns a deep copy of the submitted snapshot and preserves the full entity index/serial generation plus source name. It revalidates versions, stable IDs, schema/fragments, finite values, and entity generation at execution. A destroyed target completes as **Invalid Entity** and can never redirect the restore into a replacement entity that reuses its index. **Queue Full** is explicit back-pressure: defer or shed load, then tune the two measured queue limits only when necessary.

A snapshot may contain a subset of configured attributes, which is useful for project-owned save policies. Capture itself always emits every configured attribute found on the entity.

Migration assets

Mass Forge intentionally does not guess when an attribute has been renamed, moved to another domain, or retired. Create a **Mass Forge Attribute Snapshot Migration** Data Asset for each supported schema transition:

The schema asset's **Mass Forge Schema Overview** repeats this warning beside its repair tools. Slot repair preserves attribute identities and does not require migration; editing an attribute name or changing its domain does. See SCHEMA_EDITOR_GUIDE.md for the complete authoring workflow.

1. Give the asset a stable `MigrationId`.
2. Set `SourceSchemaVersion` to the version stored by old saves.
3. Set `TargetSchemaVersion` to a greater version.
4. Add a Rename/Move rule for every changed stable ID and a Remove rule for every intentionally retired value.
5. Increment `SnapshotSchemaVersion` in project settings to the final target version.
6. Before restore, call **Migrate Mass Forge Attribute Snapshot** once per transition, in ascending version order.

When the attribute snapshot is nested inside `Mass Forge Entity State Snapshot`, migrate a temporary copy of `State.Attributes` through the complete chain and replace that nested member only after each migration succeeds. Do not change the surrounding entity-state `FormatVersion`: format 1-4 compatibility and attribute-schema migration

are independent axes. The complete population and relationship ordering is documented in ENTITY_STATE_PERSISTENCE_GUIDE.md.

Rename/Move may change both the name and the domain. Rules use exact stable IDs; missing rules leave unrelated snapshot entries unchanged, which keeps subset snapshots valid. Migration validates the complete input before publishing output. Duplicate source values, two rules converging on one target, collisions with an already-present target, invalid IDs, non-finite values, wrong source versions, and unsupported formats fail atomically with a machine-readable result.

Migration changes identity only; it does not reinterpret or scale values. The final restore resolves the migrated IDs against the live schemas and applies the live bounds. If a gameplay update changes the meaning or units of a value, perform that project-specific conversion in the surrounding SaveGame upgrade before restore.

Scope

This focused snapshot covers attribute values only. Ability grants, cooldown clocks, Persistent Effects, active Ability Lifecycles, regeneration timing remainders, ownership, and external Actor state are not silently included. Use durable entity-state format 4 for tag/ability/cooldown/charge, regeneration-interval, active Persistent Effect, and one active cast/channel continuity. External lifecycle targets require the explicit project-owned relationship-key remap described in the durable guide.

ARCHITECTURE, SCALE, AND PERSISTENCE

Time, travel, and save contract

Mass Forge uses Unreal gameplay time for simulation. It does not use wall-clock, platform, or UTC time, and it does not silently carry world-owned runtime state into another world or SaveGame.

Clock sources

The systems intentionally use two equivalent forms of the host world's gameplay clock:

State	Clock	Consequence
Per-ability cooldowns	`UWorld` game-time deadline	Remaining time is deadline minus current game time.
Shared/global cooldown groups	`UWorld` game-time deadline	Every ability in the group observes the same scaled clock.
Charge recovery	`UWorld` game-time deadline	Lazy queries and activations recover every elapsed interval deterministically.
Cast and channel phases	Scaled world delta supplied to the coordinator	Zero delta preserves remaining time; large deltas use bounded channel catch-up.
Persistent durations and periods	Scaled world delta supplied to the subsystem	Zero delta performs cleanup but consumes no duration or period time.
Regeneration and degeneration	Scaled Mass processing delta	Interval accumulation and continuous rates follow the Mass simulation clock.

No Mass Forge duration advances from real time while the game simulation clock is stopped.

Pause behavior

The deferred-command coordinator explicitly does not tick while the game world is paused. Persistent Effect scheduling is invoked only by that coordinator, rather than through an independent tick, so Unreal's normal paused-world scheduling produces these results:

- casts, channels, persistent durations, periodic pulses, regeneration, cooldowns, and charge recovery do not advance;
- queued Entity State Restores, focused Attribute Snapshot Restores, Attribute Changes, counted Gameplay Tag changes, Instant Effects, Persistent Effect requests, Damage, Death Handling, and Abilities remain pending; and
- no gameplay completion or lifecycle event is generated merely because real time passed.

On the first normal tick after resume, queues continue under their ordinary budgets and timed state receives only the gameplay delta supplied by Unreal. Mass Forge does not synthesize the wall-clock pause interval as catch-up time.

A native caller can invoke a subsystem method directly, including the coordinator's public C++ `Tick(0)` used by automation. A zero-delta coordinator pass may flush queued work and lifecycle validity cleanup, but it does not consume cast/channel time. A zero-delta persistent pass may retire invalid owners, but it does not consume duration or period time. Projects should use the public operation APIs instead of manually ticking subsystems during gameplay.

Time dilation behavior

Global/world time dilation affects Mass Forge because Unreal supplies scaled gameplay delta and advances `UWorld` game time by that scaled amount.

- At 0.5 global dilation, one real second normally consumes about 0.5 seconds of cooldown, recovery, cast, channel, persistent-effect, and regeneration time.
- At 2.0 global dilation, one real second normally consumes about 2.0 gameplay seconds.
- Variable frame steps preserve earned persistent periods and channel pulses; their configured per-frame caps carry excess work forward.
- Actor `CustomTimeDilation` is not a Mass Forge clock. Actorless entities have no Actor, so per-Actor dilation does not alter their cooldowns or durations.

If a project needs faction-, entity-, or effect-specific time scales, it must author that as an explicit gameplay rule. Do not infer it from representation Actors.

Seamless travel and world replacement

Mass Forge subsystems and Mass entities are owned by a `UWorld`. On world teardown, pending queues, request counters, active lifecycles, AI tickets, persistent-effect metadata, inspection providers, and deferred event buffers are discarded without gameplay callbacks. Entity, request, lifecycle, death-handling, and persistent-effect handles from the former world are invalid in the destination world.

Seamless travel may preserve selected project Actors, but Mass Forge does not interpret that as preservation of their former Mass entity or representation mapping. In the destination world:

1. recreate/configure the Mass entity;
2. reacquire its complete generation-checked handle;
3. re-resolve any represented Actor mapping; and
4. restore only the state covered by an explicit supported snapshot or project-owned reconstruction.

Never copy an entity index/serial pair or world-local request/effect/lifecycle ID across travel.

Save and load behavior in 0.2.0

Mass Forge exposes two distinct versioned formats:

- `Mass Forge Entity Attribute Snapshot` contains configured Vital, Combat, and Support values plus the project schema version. Restore changes only those attributes and leaves other live state unchanged.
- `Mass Forge Entity State Snapshot` format 4 composes that attribute payload with exact counted tags, ability grants, remaining individual/shared cooldowns, stored charges, remaining charge-recovery time, regeneration interval remainder, active Persistent Effects, and active cast/channel phase. Restore replaces those included sections as one observable transaction.

Neither format contains world-local entity/request/lifecycle/effect handles or absolute world timestamps. The durable entity-state format rebuilds cooldown, recovery, duration, cast/channel, and period deadlines from remaining gameplay seconds in the destination world. Restored Persistent Effects and Ability Lifecycles always receive fresh handles.

Entity-state format 4 retains an active cast/channel's guarded definition identity/version, phase, remaining gameplay time, committed state, completed pulses, and earned channel backlog. A targetless phase restores directly; a self target rebinds to the destination; an external target requires a matching project-owned stable reference plus the recreated destination-world target handle. Old target/lifecycle handles never enter the SaveGame. AI tickets, async observers, deferred requests, event history, Actor representation, pooling, navigation, faction, inventory, quest, and other project state remain excluded.

Formats 1, 2, and 3 remain accepted as explicit compatibility paths. Format 1 has no regeneration section, so restoration resets destination regeneration partial intervals to zero. Format 2 identifies each remainder by stable target attribute and requires the destination to have the same unique target set and tick intervals; incompatible profiles fail atomically rather than shifting a timing phase to another rule. Formats 1 and 2 cannot describe active effects; formats 1-3 cannot describe active Ability Lifecycles. Each rejects incompatible active destination state instead of erasing it.

Projects can save format-4 durable combat state at Mass-idle checkpoints. Persistent Effect self-source contexts rebind to the destination; external source handles are cleared while logical Source ID and captured numeric inputs survive. Active Ability assets are guarded by stable Ability ID, **Save Compatibility Version**, execution type, and authored timing. Projects must reacquire restored handles through query APIs, rebind lifecycle observers, and never patch compact fragment slots from Blueprint.

Project policy choices

Choose and document one policy per game mode:

- **Reset temporal state** - use the focused attribute snapshot and recreate all other state from traits.
- **Preserve durable combat state** - use the format-4 entity-state snapshot at a Mass-idle point; grants, cooldowns, charges, counted tags, regeneration phase, Persistent Effects, and one active cast/channel are restored. Resolve external lifecycle-target references before restore.
- **Project-owned reconstruction** - retain game-specific state outside the Mass Forge snapshot and deliberately reconnect it after destination entities exist.
- **Offline progression** - store an external timestamp and deliberately convert elapsed real time into game outcomes during load. Mass Forge never performs this conversion automatically.

Do not mix absolute `UWorld` timestamps from one world with another world's game clock.

Automated proof

The isolated automation suite proves the currently advertised contract:

- the sole gameplay coordinator opts out of paused-world ticking and owns Persistent Effect scheduling;
- zero game-time delta preserves cast and persistent-effect remaining time;
- synthetic unchanged and half-speed game-clock samples preserve and proportionally consume cooldown time;
- variable-step persistent/channel behavior remains deterministic and bounded; and
- world teardown discards a deliberately pending ability request without invoking its completion callback.

Attribute snapshot tests separately prove deterministic capture, atomic restore, stable-ID slot remapping, format/schema rejection, and versioned rename migration. Durable entity-state tests prove deterministic composition, format-1/2/3 compatibility, stable-target regeneration timing, exact active Persistent Effect continuity, targetless cast continuation, targeted channel reference/remapping, fresh effect and lifecycle handles, resumed periodic phase, changed-definition rejection, complete-state visibility during callbacks, bounded queue ordering/back-pressure, full-

generation safety, and teardown discard. Formats 1-4 are also serialized through Unreal's tagged SaveGame archive, deserialized, migrated where required, and restored on every supported engine line; projects remain responsible for testing their surrounding SaveGame envelope and population recreation.

ARCHITECTURE, SCALE, AND PERSISTENCE

Deterministic gameplay ordering contract

Mass Forge turns same-frame gameplay into explicit ordered transactions. “Simultaneous” means that two or more operations become eligible for the same safe coordinator tick or the same scheduler boundary; it does not mean they execute concurrently or share one implicit transaction.

World coordinator order

Every safe Mass Forge coordinator tick executes these phases in this exact order:

1. **Entity State Restores** - queued complete durable snapshots atomically replace supported attributes, tags, grants, cooldowns, and charges.
2. **Attribute Snapshot Restores** - queued focused attribute-only snapshots establish saved attribute values.
3. **Attribute Changes** - queued raw Set/Add/Multiply requests.
4. **Gameplay Tag Changes** - queued raw counted-tag mutations.
5. **Instant Effects** - queued atomic attribute/tag effects.
6. **Persistent Effects** - execute queued Apply/Remove requests in FIFO order, then advance durations and periods, execute due pulses, and publish expirations.
7. **Projectile Impacts** - completed Mass projectile motion publishes its terminal result and optionally applies impact-driven damage.
8. **Damage** - queued Damage Definition transactions and their application/death events.
9. **Death Handling** - bounded deferred Mass-entity destruction.
10. **Ability Lifecycles** - advance existing casts and channels in stable source-entity order.
11. **Ability Requests** - execute queued Activate, Grant, Revoke, Cancel, and Interrupt operations through one FIFO.

`Get Deferred Command Phase Order` exposes the eleven stable names. Category order takes precedence over cross-category submission order. Complete Entity State Restores run before focused Attribute Snapshot Restores, followed by raw attribute requests and then raw tag requests; all four run before Instant Effects. Each queue is FIFO and snapshots eligible work when its phase begins; each request is an independent transaction. Work queued into the current or an earlier phase waits for the next safe tick, while work queued into a later phase may run when that phase begins in the current tick.

A successful queued snapshot restore commits its complete validated attribute set before publishing any attribute-change event; those events precede its terminal restore completion. A raw attribute change queued for the same coordinator tick observes the restored state and applies afterward. Completion-enqueued restores wait until the next tick. Stale full-generation entity handles, execution-time schema changes, or invalid payloads produce one terminal failure without partial writes or mutation of a replacement generation.

The Persistent Effect subsystem does not tick independently. The coordinator owns its invocation so persistent requests, pulses, and expirations cannot race the Damage or Ability phases through incidental Unreal subsystem registration order. Its request queue snapshots eligible work before scheduler advancement: a queued removal can suppress a same-boundary pulse, while a request enqueued by a request-completion callback waits until the next coordinator tick.

Direct calls and queued calls

Successful direct operations are synchronous and follow caller order. A direct cancellation completed before the coordinator reaches the Persistent Effect phase prevents that instance from pulsing or expiring later. A queued request follows the phase and FIFO rules above, regardless of when another category was submitted.

Callbacks may make a new direct call or enqueue new work, but the callback does not join the transaction that published it. Recursive coordinator ticks are ignored. A callback-created replacement lifecycle or effect keeps its own identity; the outer operation never continues against the replacement by name alone.

Persistent pulse, cancellation, and expiration order

At the Persistent Effect phase, the scheduler first captures all due work without publishing callbacks. It then sorts work by complete target entity handle-index followed by serial number-and by world-unique persistent handle ID. This removes Mass chunk/archetype traversal order from observable gameplay.

The order is:

1. entities in ascending complete handle order;
2. instances on one entity in ascending persistent handle order;
3. earned period numbers on one instance in ascending order;
4. all eligible period attempts for the phase; then
5. all pending expirations in the same entity/handle order.

If a finite duration and period land on the same boundary, the final earned pulse runs before `Expired`. When catch-up is capped, earned periods remain pending and expiration waits until that backlog drains. Once the scheduler has captured an exact-boundary expiration, that expiration owns the terminal reason; trying to remove the same instance from its final-period callback returns `Not Found` and does not produce a second removal event.

If an earlier period callback synchronously removes a different, later scheduled non-expiring instance, cancellation wins before that later instance becomes observable and its captured pulse is skipped. Manual removal publishes its removal synchronously. Applying an effect that cancels classified effects commits the entire replacement transaction first; direct publication then reports cancelled removals in compact existing-instance order, followed by the new apply event when a new handle is created, the stack disposition, and the optional initial period. Inside an Ability transaction these events instead follow the buffered Ability publication order described below.

Costs, effects, and Ability lifecycle order

An Ability activation validates and commits as one bounded source-plus-one-target transaction. Charge reservation, individual/shared cooldowns, source costs, self effects, target effects, and persistent state are all committed before the first success listener can observe them. Authored cost/effect arrays retain their authored order, but callbacks see the complete committed result rather than intermediate writes.

Successful Ability event publication is:

1. buffered attribute changes;
2. buffered counted-tag changes;
3. buffered persistent removal/apply/stack/initial-period events;
4. `On Ability Activated`; then
5. `On Ability Committed` with the immutable committed lifecycle snapshot.

A failed commit restores all provisional state and publishes only the detailed Ability failure. A pre-commit cast cancellation ends the lifecycle without paying costs or applying outcomes. A committed channel is never refunded by later cancellation or interruption.

Existing lifecycles advance before new queued Ability requests. Existing lifecycle sources are sorted by complete entity handle. On a channel, pulse numbers execute in ascending order. A pulse callback may cancel or replace the current lifecycle; the lifecycle ID is revalidated after every callback so no later pulse or completion from the old lifecycle leaks through. At the natural duration boundary, every earned pulse allowed by the catch-up budget publishes before the single `Completed` lifecycle event. Any carried pulse backlog is drained on later safe ticks before completion.

The Ability Requests phase snapshots one shared FIFO workload for Activate, Grant, Revoke, Cancel, and Interrupt. Successful operation-specific events publish before the unified request completion; Activate also publishes its legacy activation-only request completion before the unified completion. Cancel/Interrupt requests capture the exact lifecycle ID at submission. If phase-nine advancement ends that lifecycle, execution returns `NoActiveAbility`; if a listener or direct call replaces it, execution returns `LifecycleChanged` and cannot stop the replacement. Requests enqueued by any phase-ten completion callback wait for the next coordinator tick.

Damage and death order

One Mass-target Damage transaction atomically commits Shield, Health, and the optional first-death tag. Its attribute/tag events observe the complete state. The Damage layer then publishes:

1. `On Damage Applied`;
2. `On Entity Killed` for a first transition to zero Health; then
3. `On Entity Deactivation Requested` when that policy is selected.

For queued Damage, those application events occur before `On Damage Request Completed`. Deferred destruction occurs only in the later Death Handling phase, after the killing request has completed and before existing Ability lifecycles advance. Destruction uses FIFO death-request order and its configured per-frame budget. Consequently, an entity killed and scheduled for destruction by queued Damage cannot commit a later same-tick Ability lifecycle or queued Ability request.

The `Keep Entity` and project-deactivation policies leave the entity valid. Later phases may therefore interact with it, subject to its committed Health/death tag and ordinary Ability requirements. Mass Forge does not silently cancel unrelated active effects or abilities merely because Health reached zero; author a death tag gate, project callback, or destruction policy when that behavior is required.

What is and is not deterministic

Mass Forge guarantees ordering for its own world-local operations when inputs, entity generations, authored assets, queue contents, frame deltas, and configured budgets are the same. Stable order is not a networking authority model, rollback prediction system, cross-world order, or deterministic floating-point promise across different platforms.

Blueprint listener registration order is not a gameplay priority API. Listeners must read committed state and use request, lifecycle, effect, and death-handling IDs for correlation. Projects needing priority between unrelated external systems should enqueue explicit Mass Forge work or implement one project-owned coordinator instead of relying on delegate binding order.

Automated proof

The automation suite locks the contract by proving:

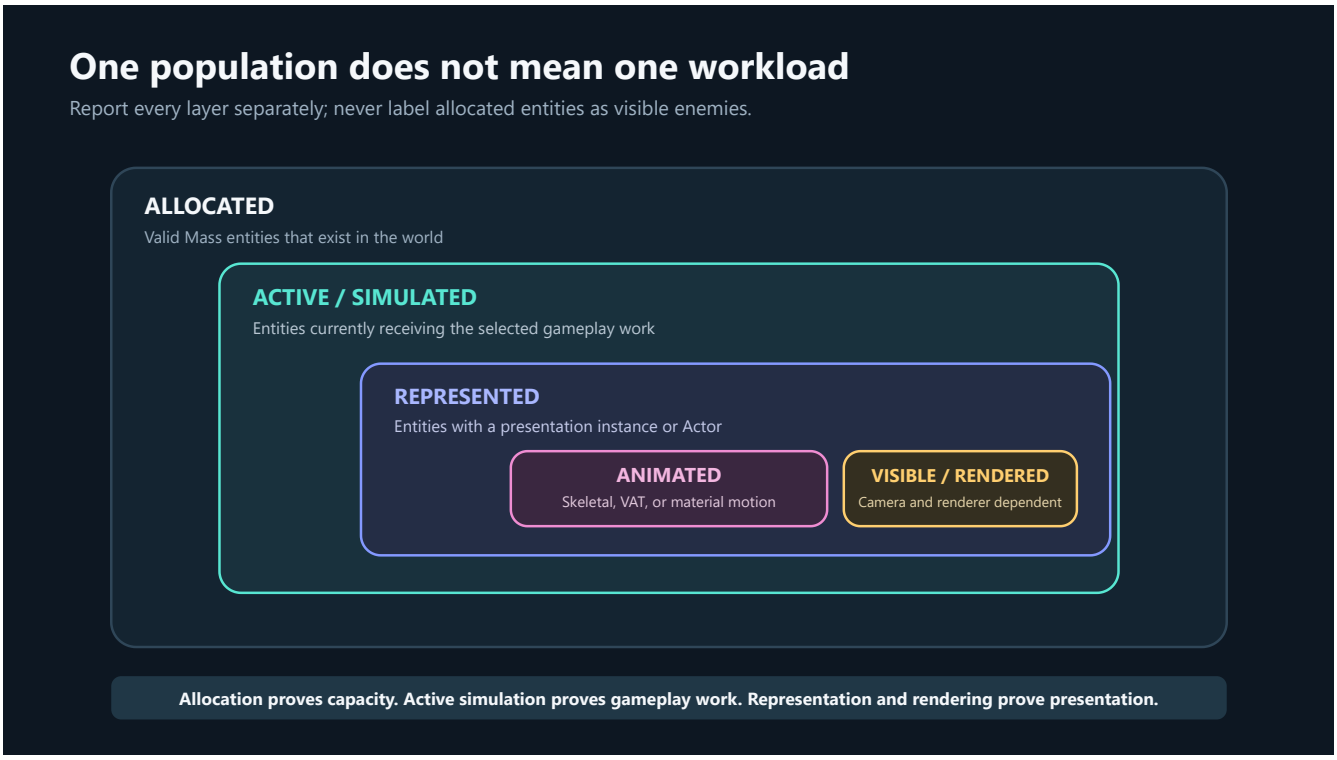
- the coordinator exposes all ten phases and executes cross-category work in that order;
- queues remain FIFO, complete entity-state restores precede focused attribute snapshots and raw attribute/tag changes, raw tag changes precede Instant Effects, and callback-enqueued same-phase work waits for a later tick;
- persistent request processing and scheduling are coordinator-owned rather than independently ticked;
- reverse-applied effects on two entities pulse and expire in stable entity-handle order;
- exact-boundary persistent pulses precede expiration and cannot emit a second cancellation removal;
- channel pulse sequences and backlogs complete before lifecycle completion;
- Ability listeners see complete costs and source/target effects, while rollback emits no provisional events;
- callback cancellation/replacement cannot advance a stale lifecycle; and
- Damage Applied, Entity Killed, request completion, and deferred destruction retain their published order.

Changing any rule above is a public behavioral change. Update this contract, relevant system guides, automation expectations, changelog, and versioning decision together.

ARCHITECTURE, SCALE, AND PERSISTENCE

Memory and capacity guide

Mass Forge keeps gameplay state in fixed-capacity Mass fragments. It does not require one UObject, Actor, Actor Component, Ability object, Effect object, targeting object, or debugger object per entity. Definition Data Assets and project provider objects are shared; inspection snapshots are built only when requested.



These counts answer different questions and must be reported separately. Allocation proves that Mass entities exist; active simulation proves gameplay work; representation, animation, and visibility describe progressively narrower presentation workloads.

Use **Get Mass Forge Memory Footprint** in Blueprint or C++ to obtain the native fragment sizes for the exact running build. The same values are included in **Copy Support Report**. The node is world-independent and allocates only its returned diagnostic array; it never adds state to an entity.

Verified Win64 fragment payloads

The following Unreal reflected structure sizes are the Mass allocation payloads. Plain C++ `sizeof` is not authoritative for inherited `USTRUCT` fragments and can be smaller. Automation checks every reported value against the running engine's reflected size and fails if a Mass Forge fragment is added without entering the report. The cross-version values below are considered verified only after the full Unreal Engine 5.6.1, 5.7.1, and 5.8.2 matrix passes.

Fragment	Added or required by	Storage scope	Bytes
`MF_VitalAttributesFragment`	Mass Forge Attributes Trait	Per entity	64
`MF_CombatAttributesFragment`	Mass Forge Attributes Trait	Per entity	192

Fragment	Added or required by	Storage scope	Bytes
`MF_SupportAttributesFragment`	Mass Forge Attributes Trait	Per entity	128
`MF_GameplayTagFragment`	Mass Forge Gameplay Tag Trait	Per entity	420
`MF_PersistentEffectsFragment`	Mass Forge Persistent Effects Trait	Per entity	1,288
`MF_PersistentAttributeModifiersFragment`	Mass Forge Persistent Effects Trait	Per entity	1,736
`MF_AbilityFragment`	Mass Forge Ability Trait	Per entity	472
`MF_RegenerationFragment`	Mass Forge Regeneration Trait	Per entity	260
`MF_HighVolumeEffectCommandFragment`	Mass Forge High Volume Effects Trait	Per entity	16
`MF_ProjectileFragment`	Mass Forge Projectile Trait	Per projectile entity	144
`MF_DerivedAttributesSharedFragment`	Mass Forge Derived Attributes Trait	Per archetype	676
`MF_HighVolumeEffectSetSharedFragment`	Mass Forge High Volume Effects Trait	Per archetype	2,388

The Derived Attributes graph and high-volume effect table are const-shared fragments. Matching entities in an archetype share one compiled 676-byte maximum-capacity derived graph and, when opted in, one 2,388-byte maximum-capacity effect table. Neither repeats its compiled rules per entity. Their required Vital, Combat, and Support attribute bags are already counted under the Attributes Trait.

Configuration totals

These totals are exact Mass Forge-owned native payload sums, not estimates of complete process memory.

Configuration	Included Mass Forge state	Bytes/entity	10,000 entities	100,000 entities	1,000,000 entities
Attributes Only wizard preset	Vital + Combat + Support	384	3.66 MiB	36.62 MiB	366.21 MiB
Combat Ready wizard preset	Attributes + counted tags + persistent timers/modifiers + abilities	4,300	41.01 MiB	410.08 MiB	4.00 GiB
Full example configuration	Combat Ready + Regeneration + high-volume command mailbox	4,576	43.64 MiB	436.40 MiB	4.26 GiB
Actorless projectile	Projectile motion/impact state	144	1.37 MiB	13.73 MiB	137.33 MiB

Formula: `Mass Forge payload = entity count x selected per-entity fragment bytes`. Projectile entities normally occupy a separate 144-byte Mass Forge archetype rather than carrying the 4,576-byte combatant preset. Add 676 bytes for each distinct maximum-capacity Derived Attributes graph/archetype and 2,388 bytes for each distinct maximum-capacity High Volume Effects table/archetype.

These figures deliberately exclude Mass entity/chunk metadata, alignment between structure-of-arrays regions, allocator slack, transforms, representation, LOD, navigation, collision, movement, project fragments, rendering, animation, assets, queues, world subsystems, and engine overhead. Measure the packaged project to obtain total resident memory.

Choosing a scalable archetype

- Use **Attributes Only** for populations that need stats but no active ability/effect state.
- Add Gameplay Tags or Abilities only to archetypes that execute those features.
- The Persistent Effects Trait adds both timer-instance and reversible-modifier storage. Do not attach it to background populations that never receive duration-based effects.
- Add Regeneration only where passive rules are required. A project can instead schedule lower-frequency aggregate updates for distant or dormant populations when its gameplay permits that approximation.
- Derived Attributes are appropriate for large matching populations because their compiled rules are shared per archetype.
- High Volume Effects adds one 16-byte command/completion mailbox per opted-in entity; its Apply/Remove table is shared per archetype. Use it only when a C++ producer can consume its explicit bounded workflow.
- Target selection reads existing transforms/representation and uses bounded world-level providers; it adds no Mass Forge fragment.
- Entity inspection is on-demand; custom providers are bounded, weak world-level references rather than per-entity storage.

A million-entity headline is not equivalent to a million fully featured combatants. On Win64, Mass Forge's Full Example payload alone is about 4.26 GiB before any engine or project state. A credible high-count design separates total simulated entities, entities carrying combat features, entities represented or animated, and entities updated during a frame.

Profiling workflow

1. Call **Get Mass Forge Memory Footprint** in the packaged target and record its totals with the engine, platform, and plugin version.
2. Count entities by actual archetype, not by the largest possible configuration.
3. Estimate the Mass Forge payload from each archetype's selected fragments.
4. Measure process memory before spawning, after spawning and settling, and at the peak gameplay population.
5. Subtract the baseline only as a diagnostic comparison; keep the absolute peak in release evidence.
6. Record total, represented, visible, animated, and actively processed populations separately.
7. Capture average and worst-frame game-thread, worker-thread, rendering, and memory figures on named hardware.

This guide records storage only. It does not claim 10k, 100k, or 1M throughput; those benchmark results remain a separate roadmap gate until measured in packaged benchmark maps on documented reference hardware.

ARCHITECTURE, SCALE, AND PERSISTENCE

Processor hot-loop performance contract

Mass Forge's automatic entity processors operate only on compiled, fixed-capacity fragment data while iterating entities. Designer-facing names and assets are resolved when traits initialize entities or when a request enters a subsystem; they are not resolved again inside a processor hot loop.

This contract covers every runtime class derived directly from `UMassProcessor` in `Source/MassForge/Public/Processors`. The release verifier discovers those classes rather than relying on a manually maintained expected count.

Audited processors

Processor	Hot-loop input	Per-entity access
Mass Forge Derived Attributes Processor	Const-shared compiled rule graph	Numeric domain and stable slot indices
Mass Forge Regeneration Processor	Fixed-capacity compiled regeneration rules and optional standard Simulation LOD timing	Numeric domain and stable slot indices
Mass Forge Persistent Attribute Modifiers Processor	Fixed-capacity aggregate records	Numeric domain and stable slot indices

Each processor obtains fragment view ranges once per chunk, walks entities through `FMassExecutionContext::FEntityIterator`, and resolves attribute storage through `ResolveAttributeSlot`. Derived rules are compiled once into dependency order and shared by the archetype. Regeneration and persistent-modifier records use bounded inline arrays; they do not grow containers while processing. Regeneration optionally uses Unreal's standard Simulation LOD due-chunk filter, so non-due variable-tick chunks never enter the marked loop; due entities consume the accumulated per-entity delta.

Forbidden work inside marked hot loops

The verifier rejects the following from each bounded `MF_HOT_LOOP_BEGIN` / `MF_HOT_LOOP_END` region:

- `FName` construction or name/schema lookup;
- synchronous or asynchronous asset loading;
- Blueprint/native delegate broadcast or interface dispatch;
- dynamic `TArray`, `TMap`, or `TSet` declarations and container-growth calls;
- UObject, shared-pointer, explicit heap, or text allocation sites;
- runtime class/structure reflection; and
- logging.

The detector self-checks every forbidden-pattern family against a representative sentinel before auditing the source. Adding another `UMassProcessor` without a matching implementation and one bounded hot-loop region fails verification. The marked region must also retain entity-index iteration and compiled slot access.

Processor constructors may use reflected class names to declare scheduling relationships. Query configuration may declare required fragment types. Those setup operations occur outside entity iteration and are not part of the hot-loop restriction.

Runtime behavior

The processors perform arithmetic and fixed-bound scans only. Invalid floating-point inputs are skipped rather than formatted or reported from the loop. Gameplay events remain subsystem transaction output and are never broadcast by a processor. Debug snapshots are captured on demand by the inspection subsystem, outside these loops.

The verifier proves the source-level architectural boundary and strict compilation on every advertised engine. It does not replace packaged performance measurement: throughput, latency, thread time, worst-frame behavior, and complete process memory remain separate benchmark gates.

The regeneration integration's exact High, Medium, Low, Off, and no-LOD behavior is documented in the Regeneration Simulation LOD policy.

Persistent-effect lifetime scheduling is subsystem work rather than a `UMassProcessor`. It has its own deadline-heap/no-population-scan source gate, scaling regression, and public diagnostics described in the persistent scheduler architecture.

Extension rules

When adding a processor:

1. Compile designer names and asset references before entity iteration.
2. Store only fixed-capacity runtime state or const-shared compiled data in the queried fragments.
3. Acquire fragment views once per chunk.
4. Mark the complete entity-iteration region with the required hot-loop markers.
5. Use numeric domains, stable slots, and bounded integer loops inside that region.
6. Queue presentation or gameplay notifications for publication outside Mass processing.
7. From a complete Mass Forge source checkout, run the repository-only `Scripts/Verify-MassForge.ps1` on every advertised engine baseline.

ARCHITECTURE, SCALE, AND PERSISTENCE

High-volume compiled effects

Mass Forge's normal Instant Effect and Persistent Effect APIs prioritize flexible Data Assets, structured context, events, and detailed dynamic results. When one Mass processor must apply or remove the same simple effect across thousands of entities, use the opt-in **Mass Forge High Volume Effects Trait** instead. It compiles effect names and schema lookups once at Entity Config build time, then runs fixed-capacity numeric operations inside Mass entity iteration.

This path is C++ first by design. It does not load assets, allocate containers, perform `FName` lookup, call a world subsystem, or broadcast delegates in the entity loop.

Supported contract

Each trait supports up to four stable effect pairs. Every pair has:

- an **Effect ID** used only during setup to resolve a numeric index;
- an **Apply Effect** containing the forward attribute and counted-tag changes;
- an explicit **Remove Effect** containing the inverse or cleanup changes.

Each side supports at most eight attribute operations and eight counted Gameplay Tag operations. Attribute operations may use finite constant Set, Add, or Multiply magnitudes. Context parameters, source/target attribute-backed magnitudes, and custom magnitude calculations are rejected because they require per-request contextual resolution or UObject/Blueprint dispatch. Use the normal Instant/Persistent Effect path for those features.

Removal is deliberately authored, not inferred. Negating Add is sometimes correct, but Set, Multiply, clamps, derived values, stacking, and intervening gameplay changes make automatic reversal ambiguous. The pair tells Mass Forge exactly what "remove" means for that project.

Entity Config setup

1. Create and validate the normal Instant Effect assets for the Apply and Remove sides.
2. Add **Mass Forge Attributes Trait** to the Entity Config.
3. Add **Mass Forge High Volume Effects Trait**.
4. Add one to four uniquely named effect pairs.
5. If either side changes counted tags, also configure the Gameplay Tag behavior expected by the entity. The high-volume trait requires the compact tag fragment for that template.
6. Resolve every compile or validation error before spawning the template.

The trait adds a 16-byte `FMF_HighVolumeEffectCommandFragment` per opted-in entity. Its compiled maximum-capacity table is `FMF_HighVolumeEffectSetSharedFragment`, stored once per matching archetype rather than once per entity.

Producer-processor workflow

Resolve the authored ID once outside the producer's entity loop:

```
const int32 DamageIndex = EffectSet.FindEffectIndex(TEXT("Crowd.Damage"));
if (DamageIndex == INDEX_NONE)
{
        return;
}
```

Inside the producer processor, submit by numeric index with a positive project-owned correlation ID:

```
const EMFHighVolumeEffectResult Admission = UE::MassForge::RequestHighVolumeEffect(
        CommandFragment,
        EffectSet,
        static_cast<uint8>(DamageIndex),
        EMFHighVolumeEffectOperation::Apply,
        CorrelationId);
```

`Accepted` means the built-in processor owns the command. It is not proof of application. One entity has one command/completion mailbox. `Busy` means the previous request is pending or its completion has not been consumed; do not overwrite it or spin.

After **Mass Forge High Volume Effects Processor** runs, consume the completion from a later ordered processor:

```
FMFHighVolumeEffectCompletion Completion;
if (UE::MassForge::ConsumeHighVolumeEffectCompletion(CommandFragment, Completion))
{
        // Completion.RequestId correlates the exact command.
        // Completion.Result is the terminal outcome.
}
```

Consuming resets the mailbox to Idle. Projects should order producer processors before `UMF_HighVolumeEffectsProcessor` and completion consumers after it. A command submitted after the built-in processor has run naturally waits for the next processing pass.

Atomicity and interaction with other systems

Every selected Apply or Remove side is projected completely before mutation. Missing slots, non-finite math, missing tag storage, tag capacity, count overflow, absent tags, and insufficient removal counts reject the command without partial attribute or tag changes. The completion identifies the failing operation index and preserves the nested Gameplay Tag result when applicable.

When an attribute currently has a Persistent Effect aggregate, the compiled operation updates the reversible base through the same persistent-attribute math used by normal mutations. Removing the persistent modifier later therefore does not discard the high-volume change.

The processor runs before Mass Forge's Derived Attributes processor in PrePhysics. Derived outputs are recalculated afterward; author high-volume operations against independent/base inputs when the value is supposed to survive derivation.

Events and presentation

The high-volume path intentionally emits no Blueprint or native per-entity delegates. A million event objects would defeat the purpose and could not be dispatched safely from worker processing. Use the completion mailbox for simulation coordination, or collect a bounded aggregate/presentation request in a later processor. Use the normal queued Instant/Persistent Effect API when individual gameplay events and full dynamic result arrays are required.

Choosing the correct path

Need	Use
One flexible effect with context, source/target captures, or delegates	Normal Instant Effect API
Timed, periodic, stacking, dispellable, or handle-based state	Persistent Effect API
Same bounded constant Apply/Remove pair over a large Mass population	High-volume compiled effect processor
Blueprint orchestration outside Mass processing	Direct or queued Blueprint APIs

This path removes avoidable per-entity lookup and allocation work; it does not by itself prove a supported entity count. The separate benchmark gate still requires packaged 10k, 100k, and 1M measurements on documented hardware.

Persistent-effect scheduler architecture

Mass Forge schedules finite lifetimes and periodic effects through one world-local deadline min-heap. The scheduler is owned by the fixed-order gameplay coordinator's **Persistent Effects** phase; it is not an independently ticking system and does not walk every Mass entity or every entity carrying the Persistent Effects fragment.

Runtime model

Each active persistent handle has one central schedule record containing its last logical update time and a monotonically changing revision. A finite expiration, a periodic pulse, or pending catch-up work contributes one next-deadline entry to the heap. Infinite non-periodic markers retain a schedule record for query and ownership consistency but require no heap entry.

The authoritative lifecycle slot remains fixed-capacity fragment data on the entity. Time is advanced lazily when its deadline is reached, when a refresh or extension must modify it, or when a public query projects its visible state. The query projection uses a copy and never changes the schedule.

The schedule clock advances only from valid scaled gameplay delta supplied by the coordinator. When Mass owns the entity manager, elapsed delta is deferred to the next safe phase. World pause, global time dilation, teardown, and save behavior therefore remain those defined in the time, travel, and save contract.

Deadline and rescheduling rules

The next deadline is the earliest of:

- the next periodic boundary;
- the finite expiration boundary; or
- the current scheduler time when earned pulse backlog or expiration cleanup is pending.

Refresh, extension, replacement, cancellation, manual removal, and ability rollback all update or retire the central schedule atomically with the lifecycle state. Rescheduling increments the handle's revision and pushes a new entry. An older entry can remain in the heap temporarily, but its stale revision can never execute gameplay. Stale entries are discarded when reached, and the heap is rebuilt when lazy entries exceed a bounded ratio to active schedules.

One due instance executes at most 64 earned pulses in a phase. Any remaining backlog receives an immediate next deadline and drains over later phases. A finite effect expires only after all pulses earned inside its lifetime have been attempted.

Determinism and callbacks

Heap layout is not observable gameplay order. Due handles are sorted by complete entity identity-index and serial number-and then persistent handle ID before work is published. Period numbers remain ascending, earned pulses precede same-boundary expiration, and a listener that removes a later captured instance suppresses that instance's remaining publication. The full cross-system order is in the deterministic gameplay ordering contract.

Destroyed-owner cleanup intentionally checks active persistent handles once per scheduler phase, including a zero-delta phase. This is proportional to active effects, not total Mass population, and preserves generation-safe retirement without requiring an entity-world scan. The lifecycle rules are in the entity lifecycle safety contract.

Blueprint diagnostics

Use **Get Persistent Effect Scheduler Stats** on the Mass Forge Persistent Effect subsystem to read an on-demand `Mass Forge Persistent Effect Scheduler Stats` value:

Field	Meaning
`ActiveEffectCount`	Active lifecycle handles.
`TimerHeapEntryCount`	Current heap entries, including lazy stale entries.
`ScheduledEffectCount`	Active central schedule records.
`DueEffectCountLastTick`	Current-revision handles whose deadline was processed in the most recent phase.
`ValidityCheckCountLastTick`	Active owner handles checked in the most recent phase.
`StaleHeapEntryCountLastTick`	Obsolete revisions discarded in the most recent phase.

These counters are diagnostics, not a frame-rate benchmark. A refresh can temporarily make `TimerHeapEntryCount` greater than `ScheduledEffectCount`; that is expected lazy invalidation. Sustained growth beyond the documented compaction ratio is a support signal.

Verification contract

Release verification applies three independent checks:

1. A source gate requires the deadline-heap operations and rejects Mass entity query, chunk traversal, or entity-iterator APIs inside the bounded scheduler region.
2. `MassForge.PersistentEffects.DeadlineHeapScaling` creates 512 unrelated entities and proves one active effect performs one ownership-validity check, not population-sized work. It also covers refresh rescheduling, stale-entry rejection, query projection, exact pulse execution, removal, and lazy cleanup.
3. The complete suite runs twice from the packaged plugin under every advertised Unreal Engine baseline, in addition to strict Editor, Development Game, and Shipping Game builds.

This architecture removes population-wide timer scans. It does not by itself claim a particular 10k, 100k, or 1M throughput; those figures require the separate reference-hardware benchmark gate.

Regeneration Simulation LOD policy

Mass Forge regeneration interoperates with Unreal Engine's standard **Simulation LOD** variable-tick path. Mass Forge does not define a second distance model or hard-code update rates. The project's Simulation LOD trait owns classification, distances, population limits, and the High, Medium, Low, and Off tick rates.

Opt-in setup

1. Configure attributes and a Mass Forge Regeneration Profile normally.
2. Add **Mass Forge Attributes Trait** and **Mass Forge Regeneration Trait** to the Entity Config.
3. Add Unreal's **Simulation LOD** trait to that Entity Config.
4. Enable variable ticking on the Simulation LOD trait.
5. Set deliberate High, Medium, Low, and Off tick rates for the project's accuracy and frame-budget needs.

No Mass Forge setting is required. An entity without the Simulation LOD variable-tick fragments continues to regenerate on every Mass Forge processor run using the ordinary world simulation delta.

Behavior and accuracy by LOD

Simulation LOD	Scheduling source	Mass Forge behavior	Accuracy guarantee
High	Project High tick rate; Unreal's default is every frame	Runs only when the standard Simulation LOD chunk is due	Constant-rate rules integrate the complete elapsed Simulation LOD delta, within floating-point precision
Medium	Project Medium tick rate	Skips non-due chunks and catches up on the next due update	No elapsed time is intentionally discarded; visible changes arrive at Medium cadence
Low	Project Low tick rate	Skips non-due chunks and catches up on the next due update	No elapsed time is intentionally discarded; larger value steps are expected at Low cadence
Off	Project Off tick rate	Follows the configured Off cadence exactly	A finite Off cadence continues batched regeneration; a project that prevents Off updates intentionally pauses regeneration until the entity becomes due again
No variable ticking	Mass Forge PrePhysics processor cadence	Uses the current world simulation delta every processor run	Existing frame-by-frame behavior is unchanged

The processor is explicitly ordered after Unreal's Simulation LOD processor. On a due chunk it reads the accumulated `DeltaTime` stored for each entity by Unreal, rather than substituting the current frame delta.

Interaction with rule Tick Interval

Simulation LOD cadence and a regeneration rule's **Tick Interval** are independent layers:

- Simulation LOD decides whether the chunk is visited.
- The accumulated entity delta advances the rule.
- A zero rule interval applies the entire accumulated delta immediately.
- A positive rule interval applies all complete authored intervals in one linear update and retains the remainder.

For example, an entity due after `1.25` seconds with a rule interval of `0.5` seconds applies `1.0` second of work and retains `0.25` seconds. This preserves the existing deterministic catch-up contract.

Deliberate sampling boundary

Constant-rate rules reproduce the same linear result regardless of LOD cadence, subject to normal floating-point precision and target clamping. Attribute-driven rates sample the rate attribute when the due update runs. Mass Forge does not reconstruct historical rate values between updates, so a frequently changing sampled rate may produce a different approximation at Medium, Low, or Off cadence.

For gameplay-critical rapidly changing rates, use High LOD, shorten the configured tick rate, or use a constant rate. Clamping occurs when the update is applied, not during skipped frames, so presentation and threshold listeners observe batched changes at the selected cadence.

Safety and verification

Invalid, zero, or negative accumulated deltas are ignored for that entity. Paused worlds therefore do not advance regeneration. The standard Simulation LOD chunk filter prevents entity iteration for non-due chunks; due chunks retain the allocation-free, compiled-slot hot loop.

`MassForge.Abilities.WorldActivation` constructs real Mass archetypes with and without Simulation LOD data. It proves that a non-due variable-tick chunk is skipped, a due entity consumes its full accumulated delta, and an entity without variable ticking still uses the world simulation delta. The release verifier also locks the optional fragment requirements, processor ordering, standard chunk filter, and both delta paths.

ARCHITECTURE, SCALE, AND PERSISTENCE

Population performance benchmark

Mass Forge ships an opt-in, reproducible headless benchmark for its simulation payload. It measures 10k, 100k, and 1M actorless Mass entities using the compiled high-volume effect path. It does not create Actors, render meshes, run navigation, animate VAT assets, or claim total-game capacity.

Published measurements for all supported engine baselines and their raw CSV files are in Population performance reference results.

What is measured

For each population, the commandlet records:

- batch entity creation wall time;
- the exact reflected Mass Forge fragment payload for Vital, Combat, Support, and the high-volume command mailbox;
- the observed process physical-memory change during batch creation, after preallocating the benchmark's handle array;
- mean and worst wall time for direct compiled Apply traversal;
- mean and worst wall time for mailbox submission, built-in processor execution, and their end-to-end sum;
- nanoseconds per entity and a correctness result.

One untimed warm-up pass runs before the requested measured iterations. Every direct application and every queued completion must cover the exact population and return `Success`; otherwise the commandlet fails.

The observed process-memory delta includes Mass chunk/allocator overhead and may include unrelated allocator reuse. The reflected payload is the deterministic capacity-planning value. Neither number includes project fragments, render resources, navigation, representation, animation, or gameplay outside this benchmark.

Run it

Maintainers should close other heavy applications and use a stable power/performance mode. Run this repository-only command from the root of a complete Mass Forge source checkout; the script and benchmark fixture are excluded from the distributable plugin package:

```
pwsh -NoLogo -NoProfile -File Scripts/Run-MassForgeBenchmarks.ps1 `
    -EngineRoot 'C:\Program Files\Epic Games\UE_5.7' `
    -OutputPath 'C:\Benchmarks\MassForge_UE_5_7.csv'
```

The script builds a strict, isolated plugin package, creates a disposable blank host project, runs the benchmark under `UnrealEditor-Cmd` with NullRHI, validates every requested row, copies only the CSV to the chosen output path, and removes its temporary package. Existing reports are never overwritten.

Use `-Iterations` to select 1-50 measured samples. The runner accepts comma-separated diagnostic populations through `-Counts`; it safely translates them to the commandlet's plus-delimited argument. A publishable reference run must retain `10000,100000,1000000`.

Reading and comparing results

Compare like with like: identical engine patch, plugin commit, build configuration, command-line counts, iterations, hardware, OS, and power state. Use the worst sample for frame-budget risk and the mean for sustained cost. An editor commandlet is a reproducible engineering baseline, not a packaged-game frame-time promise.

Record regressions against a pinned reference report. A release should be investigated when the same machine/configuration shows either a correctness failure or a sustained timing/memory regression outside the project's documented tolerance. Do not turn one machine's measurements into a universal supported-entity guarantee.

Presentation and worker-thread boundary

This benchmark isolates Mass Forge simulation and reports wall time for the serial query/processor path. VAT, Mass Representation LOD, visibility, materials, shadows, draw calls, GPU time, navigation, and project AI require the separate showcase benchmark. Worker-thread attribution requires an Unreal Insights capture; capture the commandlet or a packaged benchmark map and report game-thread, worker-thread, and worst-frame tracks separately before making a complete game-performance claim.

ARCHITECTURE, SCALE, AND PERSISTENCE

Population performance reference results

These measurements are the first published Mass Forge simulation baseline. Each report was produced by the packaged `Run-MassForgeBenchmarks.ps1` workflow with five measured iterations after one warm-up, the default 10k/100k/1M populations, NullRHI, and exact per-entity result validation.

Reference machine and conditions

- Windows 11, OS build reported by Unreal as `10.0.26200.1.256.64bit`.
- Intel Core i9-14900K, 24 physical cores and 32 logical cores.
- 102,747,594,752 bytes of physical memory reported by Unreal.
- Development Editor commandlet in a disposable host project, immediately after strict Editor, Development Game, and Shipping Game package builds.
- Actorless Mass simulation only. No rendering, VAT, navigation, representation, project AI, or gameplay outside the measured Mass Forge path.

The implementation under measurement is the benchmark/high-volume-effects source shipped with these reports. The raw CSV metadata records exact engine versions and timestamps. Future comparisons must also record the plugin commit because timings can change independently of the public plugin version.

Published results

Times are milliseconds. Payload and observed physical delta are MiB. `Queued mean/worst` is mailbox submission plus built-in processor execution.

Engine	Entities	Payload	Observed delta	Create	Direct mean	Direct worst	Queued mean	Queued worst
UE 5.6.1	10,000	3.82	6.05	1.551	0.322	0.331	0.374	0.379
UE 5.6.1	100,000	38.15	41.30	11.710	3.312	3.486	3.887	3.933
UE 5.6.1	1,000,000	381.47	402.65	169.607	34.464	37.072	43.464	47.039
UE 5.7.1	10,000	3.82	6.11	1.527	0.420	0.852	0.362	0.374
UE 5.7.1	100,000	38.15	41.41	10.856	3.145	3.147	3.603	3.641
UE 5.7.1	1,000,000	381.47	401.57	116.537	33.935	34.953	39.011	39.845
UE 5.8.2	10,000	3.82	5.93	1.462	0.313	0.318	0.368	0.377
UE 5.8.2	100,000	38.15	42.96	11.099	3.157	3.196	3.689	3.704
UE 5.8.2	1,000,000	381.47	402.52	115.527	32.698	34.584	38.166	38.503

Every row passed. The deterministic payload is about 400 bytes per entity for Vital, Combat, Support, and the high-volume mailbox in this benchmark archetype, plus one 2,388-byte shared compiled-effect table. The observed delta includes Mass chunk and allocator overhead and is not a replacement for the deterministic fragment-capacity report.

At one million entities, the serial compiled pass costs roughly 33-34 ms mean on this machine and therefore does not fit a 60 Hz frame if every entity receives an effect every frame. That is a useful capacity boundary, not a failure: real projects should distribute effect work, use Mass LOD/significance, and measure their actual archetypes and presentation workload. The 100k path remained below 4 ms mean and worst in all three engine runs.

The UE 5.7 10k direct worst sample is an isolated outlier relative to its mean and the other engines. It is retained rather than edited out; longer runs are appropriate when setting a production budget.

Raw reports

- UE 5.6.1 CSV
- UE 5.7.1 CSV
- UE 5.8.2 CSV

See the benchmark guide for reproduction, interpretation, and the explicit rendering/worker-thread boundary.

RELEASE AND SUPPORT

Engine version support

Mass Forge currently supports Unreal Engine 5.6, 5.7, and 5.8 on Win64. Each supported engine line is compiled from the same plugin source and must pass the complete release verifier before a Mass Forge release is published.

Verified matrix

Unreal Engine line	Verified baseline	Platform	Editor	Development Game	Shipping Game	Automation
5.6	5.6.1 (CL 44394996)	Win64	Pass	Pass	Pass	39/39 twice
5.7	5.7.1 (CL 48512491)	Win64	Pass	Pass	Pass	39/39 twice
5.8	5.8.2 (CL 56702186)	Win64	Pass	Pass	Pass	39/39 twice

The current final-source matrix was executed on 2026-09-21. Every row used strict includes, disabled unity/shared PCH assumptions through `BuildPlugin`, required zero Mass Forge/test warnings or errors, and compared all 494 authored package files byte-for-byte. Each package was then installed into clean Blueprint and external C++ hosts; all three engines passed 115-asset validation, authored-content automation, Blueprint Quick Start runtime smoke, and isolated C++ Editor/Game builds.

Multiplayer is intentionally outside the supported initial-release scope. Existing networking source is available for evaluation but is not represented by this matrix and is not an advertised compatibility or support promise.

The exact patch versions above are the verified baselines. Later patch releases within the same engine line are intended to remain compatible, but are not described as verified until the matrix has run against them. Unreal Engine versions before 5.6, versions after 5.8, and non-Win64 targets are currently unadvertised rather than implicitly supported.

Required Unreal plugins

Every supported package declares the same built-in dependencies and enables them automatically:

Plugin	Why Mass Forge requires it
MassGameplay	Owns MassCommon, MassActors, MassSpawner, MassLOD, and MassReplication. Its standard Unreal dependency chain supplies StateTree, ZoneGraph, Smart Objects, Pose Search, and validation support.
EnhancedInput	Supplies input for the included playable examples.
Niagara	Supplies the included example effects.

`MassEntity` and `GameplayTags` are engine modules on the supported baselines, not separate marketplace downloads. Customers do not need to obtain any third-party dependency.

Compatibility approach

Mass Forge keeps one source tree for all supported engine lines. The compatibility boundary is isolated in `Source/MassForge/Public/Compatibility/MF_MassCompatibility.h` and the runtime module rules. UE 5.8 moved foundational Mass types into `MassCore`; earlier versions continue to use the original `MassEntity` headers. Public gameplay behavior and Blueprint APIs do not change across the matrix.

Source plugins compile against the consuming project's engine. Precompiled distributable binaries are engine-line-specific and are produced separately for 5.6, 5.7, and 5.8. Content originates from the UE 5.6 compatibility baseline; the 5.7 and 5.8 distribution copies are resaved by the matching engine in an isolated host and strictly revalidated, so customers do not receive an avoidable asset-update prompt. The normalized copies never overwrite the source-of-truth assets.

Release policy

Before advertising a new engine line or patch baseline:

1. From a complete Mass Forge source checkout, run the repository-only `Scripts/Verify-MassForge.ps1` against that exact engine installation.
2. Require strict Editor, Development Game, and Shipping Game success on Win64.
3. Require the complete automation suite to pass twice with no Mass Forge/test warnings or errors.
4. Confirm the package payload and cleanup audits pass.
5. Record the exact engine version, changelist, date, and result in this matrix and the retained release evidence.

An engine line remains supported until a Mass Forge release note explicitly deprecates it. Deprecation must be announced before removal, include a migration recommendation, and never silently change an existing release's stated support.

Reporting compatibility issues

Include the exact Unreal Engine version and changelist from `Engine/Build/Build.version`, Windows version, Mass Forge version, build target, relevant log excerpt, and a minimal reproduction. If the issue occurs only on a newer engine patch, also state the nearest verified baseline that succeeds.

RELEASE AND SUPPORT

Versioning, compatibility, and upgrades

Mass Forge uses Semantic Versioning 2.0 for `VersionName` and a monotonically increasing positive integer for the Unreal plugin descriptor's `Version`. The changelog describes every customer-visible change, and the engine support matrix records the exact Unreal Engine baselines used for release verification.

Compatibility promise

The `0.x` line is a beta: a minor release may change an API or authored-data contract when required to reach a stable design. Every known breaking beta change must still be called out in the changelog, include a migration note, and increment the minor version. Patch releases must remain backward compatible and are reserved for fixes, documentation, tests, and additive behavior.

Starting with `1.0.0`:

- patch releases preserve public C++ headers, Blueprint node names and pins, reflected type/property names, enum names and values, Primary Asset types, configuration keys, and serialized formats;
- minor releases are backward-compatible and additive;
- major releases may remove or redesign public behavior only with explicit release notes and a migration path;
- a public Blueprint/C++ API scheduled for removal remains deprecated for at least two minor releases or six months, whichever is longer;
- security, platform, or upstream Unreal Engine constraints may require a shorter window, which must be identified prominently in the release notes.

An API is public when it is exported from `Source/MassForge/Public`, Blueprint-visible, serialized in an authored asset or snapshot, listed in documentation, or returned as a Primary Asset ID. Private implementation details, editor layout, log wording, internal processor composition, and undocumented private headers are not compatibility contracts.

Frozen sample-facing API baseline

Baseline ID: `MassForge.SampleAPI.0.2.v1`

This is the frozen integration surface for the first Mass Forge showcase projects. A showcase may use only this documented surface, ordinary Unreal Mass/representation APIs, and project-owned adapters. It must not include a private Mass Forge header, address an internal processor directly, depend on an editor widget implementation, or modify plugin source. Additive APIs may join a later baseline; renaming, removing, or behaviorally repurposing an item below requires a new baseline ID, changelog entry, migration note, compatibility-test update, and coordinated sample upgrade.

The baseline covers these authoring contracts:

- Vital, Combat, and Support Attribute Schema assets and stable `Mass Forge Attribute ID` domain/name identities;
- Mass Forge Instant Effect, Persistent Effect, Ability, Damage Definition, Derived Attribute Profile, Regeneration Profile, Attribute Snapshot Migration, Ability Condition, Ability Targeting Policy, Ability Execution Plan, and Effect Magnitude Calculation assets;

- Mass Forge Attributes, Gameplay Tags, Abilities, Persistent Effects, Derived Attributes, Regeneration, and High Volume Effects Entity Config traits; and
- the Entity Config wizard's Attributes Only and Combat Ready presets as optional setup accelerators rather than mandatory schemas.

The baseline covers these Blueprint/runtime workflows:

- resolve represented Actors to generation-checked entity handles; read/change attributes; capture, migrate, and direct/queued restore attribute or complete entity state;
- add/remove/query counted Gameplay Tags; direct/queued apply Instant and Persistent Effects; query/remove Persistent Effect instances;
- direct/queued grant, revoke, check, activate, cancel, and interrupt Abilities; observe cast/channel lifecycles; submit and poll AI activation tickets;
- choose targets through represented-Actor, radius, segment-trace, collision-hit, or custom-provider adapters;
- dispatch Entity-to-Entity, Actor-to-Entity, Entity-to-Actor, and Actor-to-Actor effects; apply direct/queued Damage Definitions and receive typed completion/death results;
- inspect actorless or represented entity state, record bounded gameplay history and project-supplied mind breadcrumbs, run Project Health, and retrieve memory/scheduler diagnostics; and
- evaluate the included networking source only as an experimental, unsupported integration surface for the initial release; it is not used by the five public examples or promised by the advertised compatibility matrix.

Networking types remain reflected so existing evaluation projects can compile, but they are outside the initial-release support and sample promise; no multiplayer example is included in the public launcher. A later release that promotes multiplayer to supported status must publish a dedicated compatibility baseline, example, bandwidth scope, topology matrix, and migration policy first.

The compatibility automation reflects the customer-critical functions, properties, structs, enum ordinals, serialization versions, Primary Asset identities, and required documentation clauses behind this baseline. The repository verifier separately guards the full public Blueprint surface, node metadata, bounded runtime contracts, package payload, and all advertised engine/network matrices. Passing those gates is required before a change can continue to call itself `MassForge.SampleAPI.0.2.v1`.

Protected identifiers

Do not rename or reuse these after release without the versioning process:

- schema attribute domain/name pairs and their stable slots;
- `AbilityId`, instant/persistent `EffectId`, `DamageId`, targeting `PolicyId`, and `MigrationId` values;
- Mass Forge Primary Asset type names;
- Blueprint function, pin, struct, property, enum value, config section, and config property names;
- snapshot `FormatVersion`, `SchemaVersion`, and stored attribute IDs.

New enum values belong at the end. Existing enum values must keep their names and numeric ordinals. Snapshot and migration result ordinals are explicitly assigned in source; the compatibility automation test pins the broader customer-critical baseline.

Upgrade checklist

1. Upgrade on a source-control branch and retain a restorable project/save backup.

2. Read the complete changelog range and ENGINE_VERSION_SUPPORT.md; update Unreal Engine separately when practical so failures have one cause.
3. Compile every C++ target and load/resave all affected Blueprints to surface deprecated or missing nodes.
4. Open **Mass Forge Project Health**, run a fresh scan, and resolve every new error before changing schema settings or assets.
5. Compare project configuration with any new defaults. Existing explicit settings remain authoritative unless the release notes say otherwise.
6. Validate representative abilities, effects, damage definitions, targeting policies, and entity configs through Unreal Data Validation and Project Health.
7. Test an unchanged save from every supported `SchemaVersion`, migrate it through an explicit complete chain, restore it, and verify the values before shipping.
8. Exercise direct and queued paths, actorless entities, represented Actors, entity destruction/reuse, world travel, pause/time dilation, and the project's highest entity-count tier.
9. Maintainers working from the complete Mass Forge source checkout run the project's own functional tests plus the repository-only `Scripts/Verify-MassForge.ps1` on every engine/platform combination the project ships.
10. Copy and archive the Project Health support report with the release evidence. Roll back the plugin and project together if a release gate fails.

0.1.x to 0.2.0 Blueprint migration

`Get Mass Forge Entity From Actor` now returns `Mass Forge Entity Resolve Result` execution pins instead of a boolean. Move the former true path to **Success** and handle or deliberately ignore the specific invalid-world, invalid-Actor, different-world, unavailable-representation, unrepresented, and stale-representation pins. The output handle is cleared on every failure.

`Register Mass Forge Inspection Provider` and `Unregister Mass Forge Inspection Provider` now return `Mass Forge Inspection Provider Result` execution pins. Move the former true path to **Success**. Registration now reports **Already Registered** explicitly rather than treating a duplicate request as a second success; unregistration reports **Not Registered** when appropriate. Invalid providers and provider-capacity exhaustion are also distinct outcomes.

The `Mass Forge Effect Receiver` interface callback remains implementable but is no longer exposed as a direct Blueprint call. Route effects through the Entity/Actor dispatch nodes instead; their dispatch receipt identifies the chosen receiver route and reports rejection without allowing a direct boolean-only call to bypass validation.

The schema `Find Entry By Name` and `Find Entry By Slot` helpers are now pure query nodes. Existing value wiring is unchanged; remove obsolete execution-pin wiring if an early 0.1 Blueprint used these helpers directly.

Attribute schema and saved snapshots

Moving a schema entry to a different slot without changing its domain/name identity does not reinterpret a snapshot, but deployed slot layouts should still remain stable because live fragment storage uses them. Renaming, moving to another domain, or removing an attribute requires a higher project `SnapshotSchemaVersion` and a validated Mass Forge Attribute Snapshot Migration asset. Keep `OldestSupportedSnapshotSchemaVersion` at the oldest save generation the release promises to load and test a complete path from every integer version in that window. See PERSISTENCE_GUIDE.md and SCHEMA_EDITOR_GUIDE.md.

Changing a value's unit or meaning is a project data conversion, not an identity rename. Perform that conversion in the surrounding SaveGame upgrade before Mass Forge restore, then verify current-schema clamping and derived attributes.

Effects, damage, abilities, and targeting policies

Mass Forge durable entity-state format 4 serializes grant IDs, cooldown and charge-recovery remainder, exact active Persistent Effects, and one active cast/channel. Formats 1-3 remain supported under their documented section boundaries. Projects own the surrounding `USaveGame`, stable population records, entity recreation, external lifecycle-target relationship resolution, and any game-specific conversion. Never manually bump the Mass Forge outer `FormatVersion`; migrate only the nested attribute snapshot through Mass Forge migration assets and perform other semantic conversions in the project's own save version.

An active Ability lifecycle is guarded by the Ability's stable ID, soft asset path, execution/timing fields, and **Save Compatibility Version**. Increment that authored version when an old active phase must fail instead of continuing under new behavior. Active Persistent Effects similarly require their soft definition paths and stable Effect IDs to resolve and agree. Preserve moved assets through Unreal redirectors, or ship an explicit project save upgrade; silently dropping unresolved temporal state is not a compatible migration.

Authored Data Assets are Unreal assets: retain their object paths when possible, use Unreal redirectors or Core Redirects for intentional class/property moves, and keep their stable Mass Forge IDs unchanged. When an ID must change, update every direct reference and every project-owned external/save reference as one migration, then run Project Health and Reference Viewer before deleting the old identity. Duplicate IDs are invalid even if object paths differ.

Behavioral balance changes do not require a plugin major version, but they do require project release notes and regression tests when they change damage totals, costs, cooldowns, stacking, tag gates, targeting, or lifecycle timing.

Automated compatibility gate

`MassForge.Editor.Compatibility.PublicApiBaseline` fails when the descriptor stops using semantic-version syntax or when a protected customer-critical function, recursively required SaveGame property, enum ordinal, snapshot format, or Primary Asset ID contract changes. World automation additionally performs tagged SaveGame serialization/deserialization and runtime migration/restore scenarios for durable formats 1-4. A deliberate breaking release must update the implementation, compatibility baseline, released-format fixtures, changelog, this guide, and the plugin major/minor version in the same reviewed change; changing the test alone is not a migration.

RELEASE AND SUPPORT

Third-party content notices

This page records the origin of redistributable content included with Mass Forge. It does not replace the Mass Forge product license or the Unreal Engine license agreement.

Shipped showcase content

The showcase meshes, rigs, animation clips, VAT textures, materials, VFX, textures, maps, UI, and authoring scripts are project-owned content created for Mass Forge. No purchased Marketplace pack is included.

- **Arcane Sentinel character, rig, skeletal clips, and VAT bake:** original deterministic Blender-authored output for Mass Forge.
- **Tower Defense mesh set:** original project-owned assets processed and validated by the Blender/Unreal authoring pipeline.
- **Tower Defense VFX:** original Unreal material and Niagara work created for the example.
- **Shared arena ground texture:** original project-owned texture imported through the Unreal authoring script.
- **Shared materials and procedural crowd fixtures:** original project-authored procedural output.

The authoritative file-level receipts, hashes, destinations, and redistribution decisions are maintained in `SHOWCASE_ASSET_MANIFEST.yaml` and the repository authoring manifests/receipts referenced by that file.

Unreal Engine resources

The examples may reference Unreal Engine basic-shape, default material, font, and runtime/editor resources through `/Engine` paths. Those resources are not redistributed as extracted source assets by Mass Forge; their use remains governed by the applicable Unreal Engine terms.

Documentation toolchain

The documentation source uses its declared npm dependencies to build the local website and generated PDF. Installed `node_modules`, browser binaries, and other dependency caches are excluded from the plugin package. Their individual licenses remain available through the npm packages installed from `DocumentationSite/package-lock.json`.

Verification

`Examples/MassForgeShowcases/Scripts/Verify-ArtProvenance.ps1` rejects paid assets, unknown redistribution rights, missing receipts, and unapproved origins. Any future third-party asset must be added to both provenance manifests with its exact license and redistribution evidence before it can enter a release package.

RELEASE AND SUPPORT

Mass Forge support policy

Mass Forge support covers reproducible defects in the distributed plugin, incorrect or incomplete documentation, packaging failures on advertised platforms, and compatibility regressions on supported Unreal Engine versions. The exact supported matrix is maintained in ENGINE_VERSION_SUPPORT.md.

Before requesting support

1. Update to the latest patch release in the same Mass Forge version line when practical.
2. Open **Tools > Mass Forge Project Health** and run **Scan Project**.
3. Resolve setup and asset-validation errors that explain the behavior.
4. Select **Copy Support Report**, or run `MassForge.CopySupportReport` in Unreal Editor's console.
5. Reproduce the issue in the smallest project or asset setup possible.
6. Complete SUPPORT_REQUEST_TEMPLATE.md and review every attachment for private information.

The generated report contains plugin, engine, platform, build, project-health, schema-capacity, and configured queue-limit information. It does not collect filesystem paths, usernames, machine names, account data, saved-game contents, or gameplay state. Windows absolute paths and exact machine-identity strings are redacted even inside project-authored validation text. Asset names and remaining validation messages are included; always review the text before sharing it.

Supported requests

- Crashes, assertions, corrupt state, or deterministic runtime failures attributable to Mass Forge.
- Blueprint nodes or C++ APIs that contradict their documented result contract.
- Editor tools that fail to open, validate, generate, or inspect supported assets.
- Source or packaged builds that fail on an advertised engine/platform baseline.
- Save snapshot or migration behavior that contradicts the documented atomicity and version rules.
- Documentation, accessibility, or localization defects.

Outside the support boundary

- Designing a project's combat balance, AI, UI, replication model, or proprietary gameplay architecture.
- Debugging modified Mass Forge source until the behavior also reproduces against an unmodified release.
- Third-party plugin defects or Unreal Engine defects that do not originate in Mass Forge.
- Platforms and engine lines not listed as supported.
- Multiplayer, replication, prediction, or network transport behavior in the initial `0.2.x` release; the retained networking code and guide are post-v1 experimental research.
- Recovery of project data when no backup or source-control history exists.

Support may still offer guidance for an out-of-scope case, but it is not treated as a release defect.

Severity and handling

Severity	Meaning	Expected handling
Critical	Repeatable data loss, security exposure, or a crash blocking normal editor/project use	Triage first; publish a mitigation or corrective-release plan as soon as the report is reproducible.
High	Advertised core behavior or a supported build is unusable with no reasonable workaround	Prioritize for the next corrective release.
Normal	Incorrect behavior with a safe workaround, editor usability defect, or documentation error	Schedule by impact and reproducibility.
Question	Integration or usage guidance within the documented product boundary	Answer or route to the relevant guide.

The `0.2.x` beta line is supported on a best-effort basis and does not promise a response-time SLA. A public support destination and any commercial response targets must be published in the plugin descriptor's `SupportURL` before the beta flag is removed.

Fix and compatibility policy

- Corrective releases preserve serialized IDs and documented Blueprint/C++ behavior whenever possible.
- Breaking changes require release notes, a migration path, and the deprecation policy defined by a future stable-version compatibility document.
- A defect is closed only after a regression test or an equally strong reproducible verification covers the corrected behavior.
- Reports are never required to include a complete proprietary project. A minimal reproduction, sanitized report, relevant logs, and affected assets are preferred.

RELEASE AND SUPPORT

Mass Forge support request

Copy this template into the support destination listed by the installed release. Replace bracketed prompts, remove sections that do not apply, and review all text and attachments before sending.

Summary

[One sentence describing the problem and its impact.]

Severity

[Critical / High / Normal / Question, using SUPPORT_POLICY.md.]

Reproduction

1. [Start from a blank project, minimal reproduction, or clearly described existing setup.]
2. [List exact assets, traits, nodes, settings, or C++ calls involved.]
3. [Describe the action that triggers the problem.]

Reproduction frequency: [Always / Intermittent, with an approximate rate]

Expected result

[What should happen, citing the relevant guide or API contract when possible.]

Actual result

[What happens instead, including the exact Mass Forge result enum/message.]

Environment

- Mass Forge version: [VersionName from the report]
- Unreal Engine version and changelist: [Exact value from the report]
- Platform and operating system: [Exact values from the report]
- Build target/configuration: [Editor, Development, or Shipping]
- Project type: [Blueprint / C++ / mixed]
- Entity type: [Actorless Mass / represented Mass / ordinary Actor]
- Network context: [Standalone / server / client / not applicable]
- Reproduces with an unmodified Mass Forge release: [Yes / No / Not tested]

Support report

[Paste the reviewed output from **Copy Support Report** or `MassForge.CopySupportReport`.]

Relevant assets and configuration

- Entity Config traits: [List]
- Schema domains/attributes involved: [List names and stable slots]
- Effect, damage, ability, targeting, or migration asset IDs: [List]
- Queue/direct execution path: [Describe]
- Recent schema or plugin upgrade: [Describe]

Logs and evidence

[Paste the smallest relevant log excerpt with timestamps. Attach a minimal project, screenshots, or short capture only when needed. Remove credentials, personal data, proprietary content, and unrelated logs.]

Workaround

[Known workaround, or “None.”]

Additional context

[Timing, time dilation, pause/travel state, entity destruction/reuse, LOD/representation changes, or callback/re-entrancy details if relevant.]

Submission checklist

- [ ] Project Health was run immediately before copying the report.
- [ ] The reproduction uses the smallest practical setup.
- [ ] Exact result enums/messages and asset IDs are included.
- [ ] The report and attachments were reviewed and sanitized.
- [ ] No passwords, tokens, private keys, personal data, or unnecessary proprietary files are included.

RELEASE AND SUPPORT

CI and release verification

The Mass Forge source repository owns two maintainer verification entry points: `Scripts/Verify-MassForge.ps1` for static, build, automation, and package gates; and `Scripts/Test-MassForgeMultiplayer.ps1` for real dedicated/listen server processes. The same commands are used locally and by GitLab CI so the paths cannot silently drift. They depend on repository-only fixtures and are intentionally excluded from the distributable plugin package; run them only from a complete source checkout.

What the verifier proves

One successful run performs these gates in order:

1. Rejects generated source-tree directories, build/log debris, root-level or prototype-named plugin content, conflict markers, and unresolved task markers.
2. Checks local Markdown links between documentation files.
3. Cross-checks every customer-facing node, component, and completion event named by the Blueprint quick start against its public source declaration.
4. Verifies the isolated population-benchmark runner, its 10k/100k/1M defaults, correctness gates, memory/timing fields, and interpretation boundaries remain present.
5. Discovers every Mass Forge `UMassProcessor`, self-tests forbidden-pattern detection, and enforces bounded index/slot hot loops without name lookup, asset loading, dispatch, dynamic containers, allocation sites, reflection, text construction, logging, or container growth.
6. Requires the persistent-effect deadline heap and rejects Mass entity queries, chunk traversal, or entity iterators inside its bounded scheduler phase.
7. Runs Unreal Automation Tool `BuildPlugin` with strict includes and unity/shared PCH disabled by the packaging path.
8. Compiles Win64 Editor, Development Game, and Shipping Game targets using the engine supplied through `-EngineRoot`.
9. Creates a disposable blank host project from the packaged plugin.
10. Runs every `MassForge` automation test twice in separate unattended editor sessions.
11. Requires the exact expected test count, zero failures, zero per-test warnings/errors, and no Mass Forge or Automation Controller error/warning log lines.
12. Audits the distributable for forbidden build/test debris and unnamespaced root content, rejects repository-only maintainer Scripts, requires `Installed=true` and the Runtime/Editor module types, and verifies byte-exact authored Source, Content, Docs, Resources, README, and changelog payloads.

By default the verifier creates one uniquely named directory beneath the Windows temporary directory and removes it in a guarded `finally` block whether the run passes or fails. It does not create checkpoint folders beside the plugin.

What the multiplayer verifier proves

The multiplayer command strictly packages the plugin, builds the checked-in disposable C++ host under `Tests/NetworkHarness`, and launches four scenarios through separate Unreal processes:

1. one dedicated server, a first remote client, then a second late-joining remote client; and
2. one listen host followed by one late-joining remote client;

3. one dedicated server plus a hostile-input remote client; and
4. one listen host plus a hostile-input remote client.

The first participant receives a non-empty baseline with Health, a counted Gameplay Tag, a charged granted Ability, and an active Persistent Effect carrying an owner-scoped opaque reference. It submits a real owner RPC, observes the reliable receipt and backend completion, and receives the resulting Fast Array delta. The second participant must join after that mutation, receive the changed value as its initial baseline, submit another request, and make the first participant observe the shared authoritative update. The dedicated server must also observe connection teardown. Each hostile client submits an unknown/unowned source reference and must receive an authorization rejection, attempts an oversized nine-scalar wire payload, and confirms unchanged state; its server independently audits unchanged authoritative Health. Missing markers, wrong values, harness failure markers, crashes, timeouts, or build failures fail the command.

Every process uses deterministic latency/jitter emulation. The release defaults are `-PktLag=80` and `-PktLagVariance=20`, configurable through `-NetLagMilliseconds` and `-NetLagVarianceMilliseconds`. This validates the exposed RPC and serializer boundary under delayed/variable delivery; it does not claim packet-loss recovery, raw socket fuzzing, or bandwidth saturation.

The fixture is test infrastructure and is excluded from the customer plugin package. By default its generated host, compiled files, and logs live under one uniquely named temporary directory and are removed through guarded, retrying cleanup.

Maintainer use from a source checkout

From the root of a complete Mass Forge source checkout in PowerShell 7, select one installed engine. These commands are not customer-package entry points:

```
pwsh -NoLogo -NoProfile -File Scripts/Verify-MassForge.ps1 `
    -EngineRoot 'C:\Program Files\Epic Games\UE_5.7'
```

The verifier reads `Engine/Build/Build.version`, reports the exact version, and creates its disposable host project with the matching major/minor engine association. Run it once for every baseline listed in ENGINE_VERSION_SUPPORT.md before release.

Use `-ExpectedTestCount` only when the suite intentionally changes, and update the CI default and roadmap evidence in the same commit. To inspect a package or report after a run, provide a new empty `-OutputRoot` outside the Mass Forge source checkout and `-KeepArtifacts`. The script refuses to overwrite an existing output directory or place active BuildPlugin output beneath the checkout, preventing recursive staging of earlier artifacts.

Run the real network gate with four free adjacent ports:

```
pwsh -NoLogo -NoProfile -File Scripts/Test-MassForgeMultiplayer.ps1 `
    -EngineRoot 'C:\Program Files\Epic Games\UE_5.7' `
    -BasePort 17881
```

Use a new empty `-OutputRoot` plus `-KeepArtifacts` when logs must be retained. Otherwise the command removes its workspace after success or failure.

Clean install and engine-specific distribution

After a BuildPlugin package passes the main verifier, prove the customer install boundary in fresh Blueprint and C++ hosts:

```
pwsh -NoLogo -NoProfile -File Scripts/Test-MassForgeCleanInstall.ps1 `
    -EngineRoot 'C:\Program Files\Epic Games\UE_5.8' `
    -PluginPackageRoot 'C:\MassForgeValidation\UE58\Package'
```

This loads and strictly validates all 115 shipping assets, runs authored-content automation, launches the Blueprint Quick Start runtime smoke, and compiles external Editor and Game targets against packaged public headers only with unity and PCH assumptions disabled.

Create a customer archive only after both gates pass:

```
pwsh -NoLogo -NoProfile -File Scripts/New-MassForgeDistribution.ps1 `
    -EngineRoot 'C:\Program Files\Epic Games\UE_5.8' `
    -PluginPackageRoot 'C:\MassForgeValidation\UE58\Package' `
    -OutputRoot 'C:\MassForgeDistribution\UE_5.8'
```

The distribution command refuses existing outputs, requires final non-beta legal/support metadata, copies the package into an isolated host, resaves all shipping content with the matching engine, repeats strict 115-asset validation, creates a versioned Win64 ZIP, and writes a SHA-256 manifest. `-AllowReleaseCandidateMetadata` exists only for clearly labelled non-publishable review builds, and the manifest records the override. Run it independently for UE 5.6, 5.7, and 5.8; never distribute one precompiled archive across engine lines.

GitLab runner contract

`.gitlab-ci.yml` defines one job per supported engine line. Each job expects a self-managed Windows runner with the common tags `windows` and `unreal-engine`, plus one version tag:

- `ue-5-6`
- `ue-5-7`
- `ue-5-8`

Each runner must use a PowerShell-capable executor and have Git and PowerShell 7 available. Define protected `UE_5_6_ROOT`, `UE_5_7_ROOT`, and `UE_5_8_ROOT` CI/CD variables pointing to the corresponding licensed engine installations. Engines are installed by the runner owner and are never downloaded by a job.

The three `verify` jobs and three dependent `network` jobs run for merge requests and the default branch. Each job builds in a runner-temporary directory outside the checkout, then copies the completed package, automation reports, network-process logs, and summaries into pipeline/job-specific `.ci-artifacts/` directories for seven-day retention. This ordering prevents BuildPlugin from recursively copying its own output.

Current runner status

As of 2026-09-11, the private GitLab project exposes zero project runners, zero group runners, and zero instance runners. The pipeline definition is ready, but the roadmap CI gate remains open until compatible runners execute all three jobs successfully. Attaching runners changes external infrastructure and is intentionally not simulated by local success.

The final-source local matrix passed on 2026-09-21 against UE 5.6.1, UE 5.7.1, and UE 5.8.2. Every engine compiled the complete runtime/editor plugin in strict Editor, Development Game, and Shipping Game targets; ran two independent 39/39 automation passes with zero Mass Forge/test warning or error lines; matched exact hashes for all 494 authored payload files; and completed guarded cleanup. Clean-install validation then loaded 115/115 assets, passed authored-content automation and the Blueprint Quick Start runtime smoke, and compiled isolated external C++ Editor and Game targets on every baseline. Existing multiplayer certification remains engineering evidence but is outside the initial-release product promise and public example catalogue.