

Axiom

Style Guide

How to write, test, package and ship plugins, game providers and language providers for the Axiom universal modding framework.

```
impl Plugin for BetterMinimap {  
    // Plugins are guests. Leave the game as you found it.  
    fn on_disable(&mut self, ctx: &mut Ctx) → Result<()>  
}
```

Inside this guide

01	Purpose and how to read this guide	2
02	How Axiom fits together	5
03	Choosing what to build	7
04	Toolchain and project layout	9
05	Naming	11
06	The plugin manifest	13
07	Plugin entry point and lifecycle	15
08	Calling the C ABI correctly	18
09	Events, hooks and talking to other plugins	19
10	Threading and the game loop	21
11	Memory and ownership across the boundary	22
12	Errors and crash safety	23
13	Writing a game provider	24
14	Game providers: hooking, signatures and game updates	26
15	Game API modules: exposing a game to plugins	28
16	Writing a language provider	30
17	Capabilities and security	32
18	Performance budgets	34
19	Logging and diagnostics	35
20	Testing plugins and providers	36
21	Packaging, versioning and publishing	38
22	Git, releases and review	39
A	Appendix	40

Rule levels

MUST

MUST NOT

SHOULD

SHOULD NOT

MAY

AVOID

Purpose and how to read this guide

Axiom is a universal modding framework: one core that loads plugins into many games, with plugins written in many languages. This guide is for the people who **extend** Axiom:

- **Plugin authors** write the gameplay changes, tools and overlays that players install.
- **Game provider authors** teach Axiom how to load into one game and expose that game to plugins.
- **Language provider authors** let plugins be written in another language (a WASM engine, a script VM, a native loader).

It covers how to structure, write, test, package and ship each of them so that every plugin and provider looks and behaves the same, and so that a broken one never takes the game down. Code examples use a fictional game, `mygame`, and a fictional author, `acme`.

Plugins and providers written in C or C++ also follow the [C++ Style Guide](#). This guide only adds what is specific to Axiom.

Rule levels

LEVEL	MEANING
<code>MUST</code> / <code>MUST NOT</code>	Required. The registry, CI or review rejects the plugin or provider.
<code>SHOULD</code> / <code>SHOULD NOT</code>	The default. Deviate only with a short comment explaining why.
<code>MAY</code>	Allowed when it helps; no justification needed.
<code>AVOID</code>	Legal but discouraged; expect a review question.

Principles

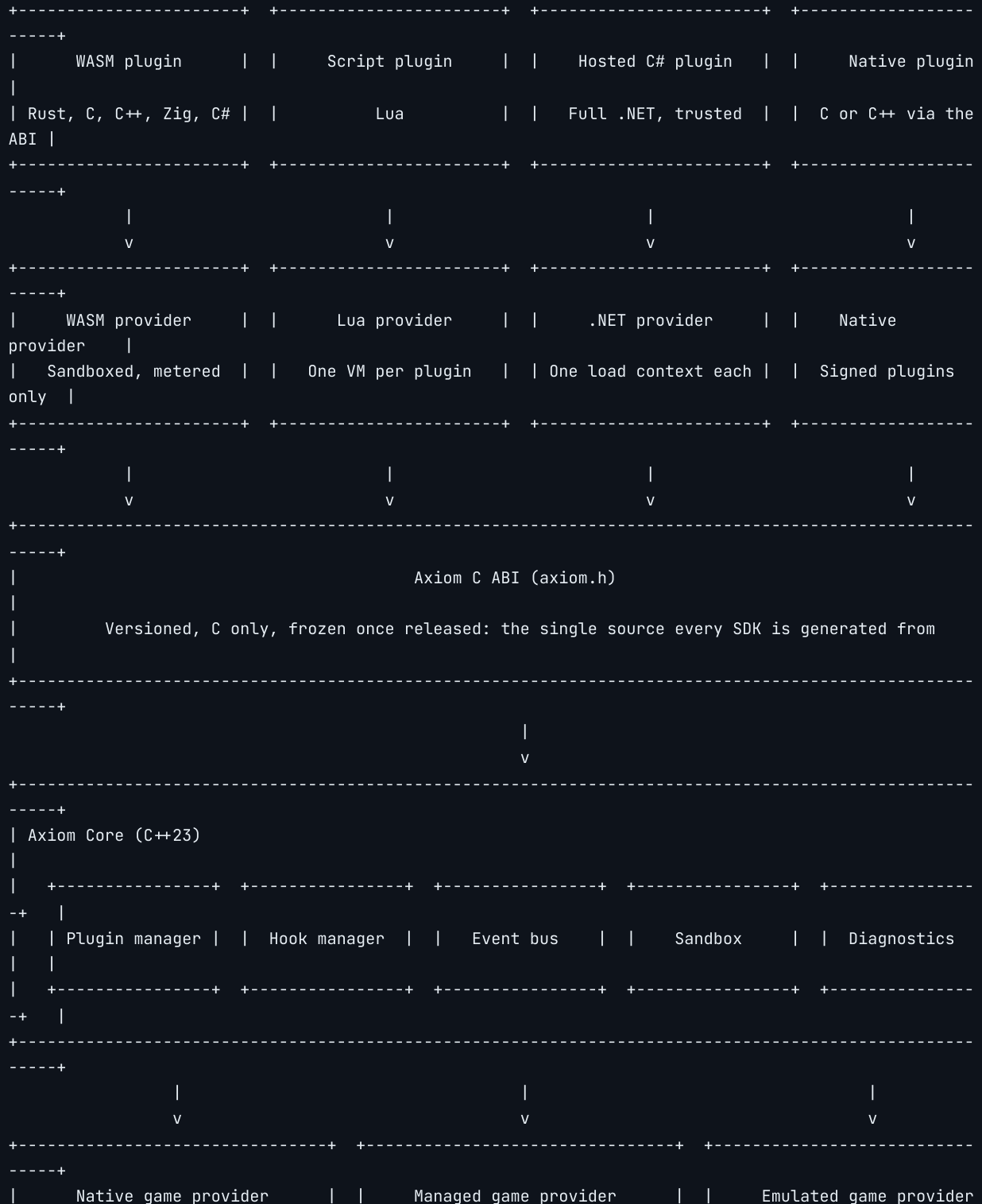
- **Plugins are guests.** They borrow the game's process and frame time. Leave everything as you found it.
- **Go through the API.** A plugin talks to the game only through the Axiom API and game API modules; a provider is the only code that touches game internals.
- **Fail small.** An error disables one plugin, never the game, the core or another plugin.
- **The lowest tier that works.** Write a WASM plugin before a script plugin, and a script plugin before a native one.

- **Version everything.** Plugins, providers, game API modules and the ABI each carry a version and declare what they need.

02

How Axiom fits together

Every plugin reaches the game only through the C ABI and the core





Axiom layers: plugins, language providers, the C ABI, the core and game providers.

A plugin never touches a game directly. It calls the C ABI through its **language provider**; the core checks and applies the call; a **game provider** turns it into hooks and reads inside one specific game. The launcher (not drawn) starts the game, loads the core and the right game provider, and manages plugin profiles.

PIECE	WHO WRITES IT	OWNS	MAY TOUCH GAME MEMORY?
Plugin	Plugin authors	Gameplay changes, tools, UI	No — only through game API modules
Language provider	Provider authors	Running plugin code in one language, safely	No
C ABI (<code>axiom.h</code>)	Axiom	The contract every plugin and provider builds against	—
Core	Axiom	Loading, ordering, hooks, events, capabilities, diagnostics	No
Game provider	Provider authors	Loading into one game; its hooks, symbols and game API module	Yes, for exactly one game

Rules

- MUST** AXIOM-02.1 keep dependencies pointing down the diagram: plugins → language providers → ABI → core → game providers.
- MUST** AXIOM-02.2 reach the game from a plugin only through the Axiom API and a game API module. There is no back door for "trusted" native plugins.
- MUST** AXIOM-02.3 add support for a game by writing a game provider, and for a language by writing a language provider. Neither requires changes to the core.

03

Choosing what to build

YOU WANT TO...	BUILD A...	TIER
Change gameplay, add UI, add commands, record stats	Plugin	WASM by default
Share code between your plugins	Library plugin (a plugin that exports services, no gameplay)	Same as its users
Make Axiom work with a new game	Game provider + its game API module	Native (C++)
Expose more of a supported game to plugins	A new version of that game's game API module	Native (C++)
Let plugins be written in a new language	Language provider + its SDK	Native (C++)

Plugin tiers

Plugins are tiered by the language provider they run on. The tier decides how much Axiom trusts the code.

TIER	LANGUAGE PROVIDER	TRUST	ISOLATION	PICK IT WHEN
1	WASM (Rust, C, C++, Zig, AssemblyScript, C#)	Untrusted	Own linear memory, fuel limits, capability-gated imports	Almost always; the default template
2	Script (Lua)	Untrusted	Own VM state, stripped standard library, instruction and memory quotas	Small scripts, config-heavy plugins, fast iteration with live reload
3	Native (C or C++ via the C ABI) or hosted .NET (C#)	Trusted	None; runs in the game process	Measured performance needs or OS access WASM cannot offer; must be signed

C# has two paths. Untrusted C# plugins compile to WASM with .NET's WASI target and run on the WASM language provider like any other tier 1 plugin. C# plugins that need full .NET (reflection-heavy libraries, large existing codebases, fast startup) run on the hosted .NET language provider. .NET cannot sandbox code

inside a process, so hosted .NET plugins are tier 3: trusted and signed, under the same rules as native plugins.

- **MUST** AXIOM-03.1 start every new plugin on tier 1 and move to a higher tier only for a reason written in the plugin's README.
- **MUST NOT** AXIOM-03.2 write a game provider to work around a missing API; ask for (or contribute) the API in the game's API module instead.
- **SHOULD** AXIOM-03.3 split a large plugin into a library plugin and thin feature plugins when other authors could reuse the library.

04

Toolchain and project layout

Start every plugin and provider from the official template (`axiom new plugin --lang rust`, `axiom new game-provider`, `axiom new language-provider`). The templates already follow this guide.

Toolchain

AREA	PLUGINS	PROVIDERS
Language	Any with an Axiom SDK (Rust, C, C++, Zig, AssemblyScript, C#, Lua)	C++23, with a C11 public surface
Build	The language's own build tool (<code>cargo</code> , <code>zig build</code> , <code>npm</code> , <code>dotnet</code>) wrapped by <code>axiom build</code>	Premake5; MUST build with MSVC on Windows; Clang/GCC for Linux and macOS
Formatting	The language's standard formatter (<code>rustfmt</code> , <code>clang-format</code> , <code>dotnet format</code> , <code>stylua</code>)	<code>clang-format</code> , 2-space indent
Packaging	<code>axiom pack</code> → <code>.axmod</code>	<code>axiom pack --provider</code> → <code>.axprov</code>
Testing	<code>axiom test</code> with the mock host	Unit tests + the provider conformance suite

A plugin project

```
better-minimap/  
  axiom.toml           # Manifest (§6)  
  src/  
    lib.rs             # Entry point: on_load / on_enable / on_disable / on_unload  
    minimap.rs  
    minimap.test.rs    # Tests beside the code  
  assets/  
  config/default.toml  # Default user settings  
  README.md  
  CHANGELOG.md
```

A game provider project

```
provider-mygame/
  axiom.toml           # Provider manifest: game IDs, supported builds
  premake5.lua
  include/axiom/mygame.h # Game API module (generated from spec/)
  spec/                # Game API definitions (IDL)
  src/
    Provider.cpp        # ax_game_provider_t implementation
    Hooks/              # One file per hooked subsystem
    Signatures/         # Patterns and offsets per game build
    Api/                # Game API module implementation
    Provider.test.cpp
  tests/conformance/    # Provider conformance suite config
```

Rules

- **MUST** AXIOM-04.1 keep one plugin or provider per repository (or per top-level folder in a monorepo), with its manifest at the root.
- **MUST** AXIOM-04.2 keep tests beside the code they test (`<file>.test.<ext>`), and shared fixtures in `tests/`.
- **MUST** AXIOM-04.3 commit the lock file of the language's package manager, and pin the Axiom SDK version.
- **MUST NOT** AXIOM-04.4 vendor or modify Axiom headers; depend on the SDK package.
- **SHOULD** AXIOM-04.5 keep game-build-specific data (signatures, offsets) in data files under `Signatures/`, not inline in code, so a game update is a data change.

05

Naming

Axiom's short code is `AX`: it prefixes every Axiom macro, and in lower case every C ABI symbol. Your own code uses your own prefix.

THING	CONVENTION	EXAMPLE
Plugin ID	Lower-case <code>author.plugin-name</code>	<code>acme.better-minimap</code>
Game provider ID	<code>provider.<game></code> (official) or <code>author.provider-<game></code>	<code>provider.mygame</code>
Language provider ID	<code>lang.<language></code> or <code>author.lang-<language></code>	<code>lang.wasm</code> , <code>lang.lua</code> , <code>lang.dotnet</code>
Game ID	Lower-case, stable, chosen by the game provider	<code>mygame</code>
Package file	<code><id>-<version>.axmod</code> / <code>.axprov</code>	<code>acme.better-minimap-1.2.0.axmod</code>
Event name	<code><area>.<noun>.<verb-past></code>	<code>player.health.changed</code>
Plugin-defined event	Your plugin ID + <code>/</code> + event	<code>acme.better-minimap/zoom.changed</code>
Capability	<code><area>:<access></code>	<code>fs:read</code> , <code>net:connect</code> , <code>game:write</code>
Config key	snake_case	<code>zoom_level</code>
C ABI symbols (Axiom)	<code>ax_<area>_<verb></code> , types <code>ax_..._t</code> , constants <code>AX_...</code>	<code>ax_event_subscribe</code> , <code>AX_RESULT_OK</code>
Game API symbols	<code>ax_<game>_<area>_<verb></code>	<code>ax_mygame_player_get_health</code>
Your C/C++ macros	Your own 2–3 letter prefix, never <code>AX_</code>	<code>BMM_ASSERT</code>

Rules

- MUST** AXIOM-05.1 choose a plugin ID once; it is how saves, configs, dependencies and the registry find the plugin. **MUST NOT** rename it after the first release.

- **MUST NOT** AXIOM-05.2 use the `ax_`, `AX_`, `axiom.` or `provider.` prefixes for your own symbols or IDs; they are reserved for Axiom and official providers.
- **MUST** AXIOM-05.3 namespace every event, config key and storage key a plugin defines under its plugin ID.
- **MUST** AXIOM-05.4 keep IDs, event names and capability names lower-case ASCII so they are stable across file systems and languages.
- **SHOULD** AXIOM-05.5 use Axiom's nouns consistently in code, docs and UI: *plugin*, *game provider*, *language provider*, *game API module*, *event*, *hook*, *capability*.

06

The plugin manifest

Every plugin has an `axiom.toml` at its root. The core reads it before loading any code, so it decides load order, dependencies and permissions.

```
[plugin]
id = "acme.better-minimap"
name = "Better Minimap"
version = "1.2.0"           # Semantic version
description = "A zoomable minimap with waypoints."
authors = ["Acme Mods <mods@acme.example>"]
license = "MIT"
language = "wasm"           # Language provider: wasm | lua | dotnet | native
entry = "better_minimap.wasm"

[requires]
axiom = "≥1.4, <2"          # Core ABI version range
"game:mygame" = { api = "≥2.1, <3", builds = ["*"] }

[dependencies]
"acme.ui-kit" = "≥0.9, <1"

[optional-dependencies]
"other.waypoints-sync" = "≥1, <2"

[load]
after = ["acme.ui-kit"]
phase = "game"              # early | game | late

[capabilities]
required = ["game:read", "ui:draw", "storage:plugin"]
optional = ["net:connect"]   # Asked for at runtime with a reason

[capabilities.reasons]
"net:connect" = "Sync waypoints with your party."

[network]
hosts = ["sync.acme.example"] # The only hosts net:connect may reach (§17)
```

Rules

- **MUST** AXIOM-06.1 declare every capability the plugin uses, with a one-sentence reason for each optional one. Undeclared calls fail with `AX_RESULT_DENIED`.

- **MUST** AXIOM-06.2 declare version ranges, never exact versions, for `axiom`, game API modules and dependencies; the upper bound is the next major.
- **MUST** AXIOM-06.3 declare load order with `after` / `before` and `phase`, never by relying on file names or install order.
- **MUST** AXIOM-06.4 bump `version` for every published change, following semantic versioning (§21).
- **MUST NOT** AXIOM-06.5 put secrets, tokens or machine-specific paths in the manifest.
- **SHOULD** AXIOM-06.6 keep `description` under 120 characters; it is shown in the launcher and registry.

07

Plugin entry point and lifecycle

States

STATE	CALLBACK	DO HERE	NEVER DO HERE
Loaded	<code>on_load</code>	Read config, register events, hooks and services; allocate long-lived state	Touch game state; the game may not be ready
Enabled	<code>on_enable</code>	Start gameplay effects, show UI, start timers	Long blocking work (move it to a task)
Disabled	<code>on_disable</code>	Undo everything <code>on_enable</code> did: remove effects, hide UI, stop timers	Fail; disabling must always succeed
Unloaded	<code>on_unload</code>	Free everything allocated in <code>on_load</code> ; save state if hot-reloading	Call any Axiom API other than logging

- **MUST** AXIOM-07.1 keep every callback idempotent and safe to call in order load → enable → disable → enable → disable → unload. Players toggle plugins at runtime.
- **MUST** AXIOM-07.2 leave the game exactly as it was found after `on_disable`: every hook, event subscription, spawned entity and UI element the plugin added is gone.
- **SHOULD** AXIOM-07.3 register subscriptions and hooks through scoped handles the SDK releases automatically on disable, rather than tracking them by hand.
- **MUST** AXIOM-07.4 return from each callback within the lifecycle budget (\$18); slow initialisation goes into an async task.
- **MAY** AXIOM-07.5 implement `on_save_state` / `on_restore_state` to keep state across hot reloads during development.

Entry point (C)

```
#include <axiom/axiom.h>
#include <axiom/mygame.h>

typedef struct bmm_state { ax_subscription_t on_health; int32_t zoom; } bmm_state;

static ax_result_t AX_CALL bmm_on_load(ax_plugin_ctx_t* ctx) {
    bmm_state* s = ax_alloc(ctx, sizeof(bmm_state)); /* owned by the plugin */
    if (s == NULL) return AX_RESULT_OUT_OF_MEMORY;
    s->zoom = ax_config_get_i32(ctx, AX_STR("zoom_level"), 3);
    ax_plugin_set_user_data(ctx, s);
    return AX_RESULT_OK;
}

static ax_result_t AX_CALL bmm_on_disable(ax_plugin_ctx_t* ctx) {
    bmm_state* s = ax_plugin_get_user_data(ctx);
    ax_event_unsubscribe(ctx, s->on_health);
    return AX_RESULT_OK;
}

AX_PLUGIN_EXPORT ax_result_t AX_CALL ax_plugin_main(const ax_api_t* api, ax_plugin_desc_t* out) {
    if (api->version_major != AX_ABI_VERSION_MAJOR) return AX_RESULT_UNSUPPORTED_VERSION;
    out->struct_size = sizeof(*out);
    out->on_load = bmm_on_load;
    out->on_enable = bmm_on_enable;
    out->on_disable = bmm_on_disable;
    out->on_unload = bmm_on_unload;
    return AX_RESULT_OK;
}
```

Entry point (Rust SDK)

```
use axiom::prelude::*;
use axiom_mygame::player;

#[axiom::plugin]
struct BetterMinimap {
    zoom: i32,
}

impl Plugin for BetterMinimap {
    fn on_load(ctx: &mut Ctx) → Result<Self> {
        Ok(Self { zoom: ctx.config().get("zoom_level").unwrap_or(3) })
    }

    fn on_enable(&mut self, ctx: &mut Ctx) → Result<()> {
        // Scoped: removed automatically on disable.
        ctx.events().on(player::HealthChanged, |ev, _ctx| {
            log::debug!("health {} → {}", ev.old, ev.new);
        });
        Ok(())
    }
}
```

08

Calling the C ABI correctly

Most authors use an SDK, which handles these rules. Native plugins, providers and SDK authors work with `axiom.h` directly and **MUST** follow them.

- MUST** AXIOM-08.1 check the `ax_result_t` of every call that returns one. `AX_RESULT_DENIED` means a capability is missing; `AX_RESULT_NOT_READY` means the game is not in a state where the call is valid.
- MUST** AXIOM-08.2 set `struct_size = sizeof(...)` on every struct passed to Axiom. This is how newer cores stay compatible with your plugin; a zero `struct_size` is rejected.
- MUST** AXIOM-08.3 zero-initialise structs before filling them (`ax_event_desc_t d = {0};`) so fields added in later minor versions start at their defaults.
- MUST** AXIOM-08.4 pass strings as `ax_str_t` (UTF-8 pointer + length; use `AX_STR("literal")`), never assume Axiom returns null-terminated strings.
- MUST** AXIOM-08.5 use only fixed-width types at the boundary (`int32_t`, `uint64_t`, `float`, `ax_bool_t`); never `bool`, `long`, `wchar_t` or a C++ type.
- MUST** AXIOM-08.6 treat every handle (`ax_entity_t`, `ax_subscription_t`) as opaque and possibly stale: check results when using a handle you stored across frames.
- MUST** AXIOM-08.7 mark callbacks `AX_CALL` and keep them non-throwing (C++: `noexcept`), and pass a `user_data` pointer instead of using globals.
- MUST** AXIOM-08.8 check the API version you were given at the entry point and call newer functions only when `api→version_minor` says they exist (the SDK's `ax_has(api, fn)` macro does this).
- MUST NOT** AXIOM-08.9 call Axiom from threads Axiom did not give you unless the function is documented as thread-safe (§10).

```
ax_event_desc_t desc = {0};
desc.struct_size = sizeof(desc);
desc.name = AX_STR("player.health.changed");
desc.priority = 0;
desc.callback = bmm_on_health;
desc.user_data = state;

ax_result_t r = api→event_subscribe(ctx, &desc, &state→on_health);
if (r != AX_RESULT_OK) {
    ax_log(ctx, AX_LOG_ERROR, "subscribe player.health.changed failed: %s", ax_result_name(r));
    return r;
}
```

09

Events, hooks and talking to other plugins

Events

- **SHOULD** AXIOM-09.1 react to events instead of polling game state every frame.
- **MUST** AXIOM-09.2 keep event handlers short and non-blocking; heavy work is scheduled as a task (§10).
- **MUST** AXIOM-09.3 treat event payloads as borrowed: valid only during the callback. Copy what you need.
- **MUST** AXIOM-09.4 use priorities only to express a real ordering need, and document it. Lower numbers run first; the default is `0`.
- **MUST** AXIOM-09.5 cancel a cancellable event (`ax_event_cancel`) only when the plugin's purpose is to block that action, and log why at `debug` level.

Hooks

Hooks are the plugin-facing way to change game behaviour at a point the game API module exposes (`mygame.damage.apply`). Plugins never patch game code themselves.

KIND	RUNS	CAN
<code>pre</code>	Before the game's code	Read and change arguments, or skip the original
<code>post</code>	After the game's code	Read and change the result
<code>observe</code>	After, read-only	Read arguments and result; cheapest

- **MUST** AXIOM-09.6 use the weakest hook kind that works: `observe` over `post` , `post` over `pre` .
- **MUST** AXIOM-09.7 skip the original in a `pre` hook only when the plugin replaces that behaviour completely; two plugins skipping the same original is a conflict the core reports.
- **MUST NOT** AXIOM-09.8 keep a hook's argument pointers after the hook returns.

Services between plugins

- **MUST** AXIOM-09.9 talk to other plugins through **services** (a versioned interface a plugin registers under its ID) or plugin-defined events, never by reaching into another plugin's memory or files.
- **MUST** AXIOM-09.10 version every service interface with semantic versioning, and declare the dependency in the manifest.

- **MUST** AXIOM-09.11 handle an optional dependency being absent or disabled at any time; check `ax_service_get` every time you need it, or listen for `plugin.disabled`.
- **SHOULD** AXIOM-09.12 pass plain data (structs, strings, byte buffers) through services, not handles that only make sense inside one language provider.

10

Threading and the game loop

Axiom runs plugin callbacks on the **game thread** during defined phases of each frame, because most game state is only safe to touch there.

PHASE	WHEN	TYPICAL WORK
<code>frame.begin</code>	Start of the game's update	Read input, start per-frame work
<code>game</code> events and hooks	During the game's own update	React to gameplay
<code>frame.end</code>	After the game's update, before rendering	Apply queued changes
<code>render.ui</code>	During UI rendering	Draw overlays and windows
Task pool	Worker threads, any time	File I/O, networking, heavy computation

- **MUST** AXIOM-10.1 touch game state and call game API functions only from the game thread, inside a callback or hook, unless the function is documented as thread-safe.
- **MUST NOT** AXIOM-10.2 block the game thread: no sleeping, no synchronous network or disk I/O, no waiting on locks held by worker threads.
- **MUST** AXIOM-10.3 move slow work to the task pool with `ax_task_spawn` and bring results back with `ax_task_post_to_game` (or the SDK's `ctx.spawn(...).then_on_game(...)`).
- **MUST NOT** AXIOM-10.4 create raw OS threads in a plugin. Native plugins that need a dedicated thread request one through `ax_thread_create` so the core can name, track and stop it on disable.
- **MUST** AXIOM-10.5 cancel and wait for every task a plugin started before `on_disable` returns; the SDK's scoped tasks do this.
- **MUST** AXIOM-10.6 keep state shared between a task and the game thread behind the SDK's channels or a mutex, never as unsynchronised globals.

WASM and Lua plugins are single-threaded per plugin: their callbacks never run concurrently, and tasks are run as async calls on the plugin's own queue.

11

Memory and ownership across the boundary

Every pointer and handle that crosses the C ABI has one of four ownership rules, and the header says which.

MARKED	MEANING	YOUR JOB
<code>/* borrowed */</code>	Axiom owns it; valid only during this call or callback	Copy what you keep
<code>/* valid until <event> */</code>	Axiom owns it; valid until a named point (end of frame, unload)	Do not use it after that point
<code>/* caller owns */</code>	You allocated it and pass it in; Axiom does not keep it	Free it when you are done
<code>/* transfers ownership */</code>	Axiom allocated it for you	Release it with the matching <code>ax_*_release</code> function

- **MUST** AXIOM-11.1 free memory with the allocator that allocated it. Memory from `ax_alloc` goes back through `ax_free`; never mix with `free`, `delete` or another language's allocator.
- **MUST NOT** AXIOM-11.2 pass ownership of a C++ object, a `std::` type, a Rust `Box` or a Lua table across the ABI; pass plain data or a handle.
- **MUST** AXIOM-11.3 release every handle and buffer a plugin owns by `on_unload`. The core reports leaks per plugin in development builds.
- **MUST** AXIOM-11.4 treat handles to game objects (`ax_entity_t`) as weak: the object can be destroyed between frames. Re-validate with `ax_<game>_entity_is_valid` or listen for the destroy event.
- **SHOULD** AXIOM-11.5 use the SDK's RAII or scoped wrappers (`ax::Handle<T>` in C++, `Drop` types in Rust) instead of manual release calls.
- **MUST** AXIOM-11.6 in providers: validate every pointer and length that comes from a plugin before reading it, and copy guest data out of WASM linear memory before using it.

12

Errors and crash safety

In plugins

- **MUST** AXIOM-12.1 handle every error result. Expected failures (missing config, denied capability, entity gone) are logged at `warn` or `debug` and the plugin carries on.
- **MUST** AXIOM-12.2 return a failure from `on_load` or `on_enable` when the plugin cannot work, with a message that tells the player what to do ("requires acme.ui-kit 0.9 or newer").
- **MUST NOT** AXIOM-12.3 let an exception, Rust panic or Lua error escape a callback. SDKs catch them at the boundary, log them with a stack trace and disable the plugin; native plugins **MUST** do the same (`noexcept` callbacks with a `try` / `catch` at the edge).
- **MUST NOT** AXIOM-12.4 call `abort`, `exit`, `std::terminate` or anything that ends the process.

In providers

- **MUST** AXIOM-12.5 isolate every call into a plugin: catch C++ exceptions, and on Windows use structured exception handling around native plugin calls, so a crash in a plugin disables that plugin instead of the game.
- **MUST** AXIOM-12.6 disable a plugin that exceeds its limits (instruction budget, memory quota, frame-time budget repeated across frames) and tell the player which plugin and why.
- **MUST** AXIOM-12.7 write a crash report on a real crash (minidump on Windows) that names the plugins and providers loaded and which one was running, and **MUST NOT** upload it without the player's consent.
- **MUST** AXIOM-12.8 fail closed when a game build is unknown: refuse to install hooks and report "unsupported game version" rather than patching memory that may have moved.

Messages

- **MUST** AXIOM-12.9 write player-facing errors as: what happened, which plugin, what to do. "Better Minimap was disabled: it needs the `ui:draw` permission. Allow it in the launcher."
- **SHOULD** AXIOM-12.10 give every error a stable code (`BMM-0012`) so players can search for it and you can find it in logs.

13

Writing a game provider

A game provider is the only code in Axiom that knows about one specific game. It implements the provider contract, installs the low-level hooks, and backs the game API module that plugins call.

The provider contract

FUNCTION	CALLED	MUST
<code>detect</code>	By the launcher, before starting the game	Identify the game and its exact build from files only (hashes, version resources); no side effects
<code>attach</code>	Once, early in the game process	Resolve signatures, verify the build, prepare hooks; fail closed on unknown builds
<code>start</code>	When the game is ready for plugins	Install hooks, register the game API module, emit <code>game.ready</code>
<code>frame</code>	Each frame, from the game thread	Drive the frame phases (§10)
<code>stop</code>	Before unload or on shutdown	Remove every hook, release every resource
<code>detach</code>	Last	Leave the process as it was before <code>attach</code>

```

namespace MyGame {

class Provider final : public Axiom::GameProvider {
public:
    ax_result_t Detect(const ax_game_files_t& files, ax_game_match_t& out) noexcept override;
    ax_result_t Attach(const ax_api_t& api) noexcept override;
    ax_result_t Start() noexcept override;
    void Frame(ax_frame_phase_t phase) noexcept override;
    void Stop() noexcept override;
    void Detach() noexcept override;

private:
    Signatures m_Signatures;
    HookSet m_Hooks;
    Api::Module m_Api;
};

} // namespace MyGame

AX_GAME_PROVIDER(MyGame::Provider)

```

Living inside the game's process

- **MUST NOT** AXIOM-13.1 do work in `DllMain` or a shared-library constructor beyond recording the module handle: no allocation, threads, locks, logging or library loads under the loader lock.
- **MUST NOT** AXIOM-13.2 rely on static initialisation order or non-trivial global constructors; create and destroy state in `attach` / `detach`.
- **MUST** AXIOM-13.3 link the C runtime statically on Windows (`/MT`) so the provider never conflicts with the game's CRT.
- **MUST NOT** AXIOM-13.4 change process-wide state the game depends on (FPU control word, locale, working directory, signal or exception handlers, COM apartment) without restoring it on the same thread.
- **MUST** AXIOM-13.5 be able to stop and detach cleanly at runtime: every hook removed, every thread joined, every allocation returned.
- **MUST** AXIOM-13.6 use `std::expected<T, Axiom::Error>` for recoverable failures and keep every function the core or a hook can call `noexcept`.

Managed and emulated games

- **SHOULD** AXIOM-13.7 hook managed games (.NET, JVM) at the runtime level through the runtime's supported profiling or agent interfaces before resorting to native patches.
- **MUST** AXIOM-13.8 in emulated targets, hook the emulator's guest-facing layer (CPU, memory, HLE calls), not the emulator's UI or host code, and translate guest addresses through the emulator's memory map every time.

14

Game providers: hooking, signatures and game updates

Hooking

- **MUST** AXIOM-14.1 install every hook through Axiom's hook manager (`Axiom::Hooks`), never with a private detour library, so hooks from different providers are ordered, tracked and removable.
- **MUST** AXIOM-14.2 suspend or synchronise game threads while patching code, and verify the bytes being replaced match what the signature expected.
- **MUST** AXIOM-14.3 keep each native hook as thin as possible: capture arguments, call into the core's dispatch, return. Game logic lives in the game API module, not in the trampoline.
- **MUST** AXIOM-14.4 preserve the hooked function's calling convention, registers and stack alignment exactly; write a test for each hook that calls the original through it.
- **SHOULD** AXIOM-14.5 hook the narrowest stable function that gives plugins what they need, and prefer one hook that feeds many events over many hooks.

Signatures and offsets

- **MUST** AXIOM-14.6 locate game code and data by byte-pattern signatures, exported symbols, RTTI or metadata — never by hard-coded absolute addresses.
- **MUST** AXIOM-14.7 keep signatures and offsets in data files per game build (`Signatures/<build>.toml`), each with a name, pattern, expected match count and the build it was verified on.
- **MUST** AXIOM-14.8 fail closed: if any required signature has zero or more than one match, refuse to start and report which one.
- **SHOULD** AXIOM-14.9 resolve all signatures once in `attach`, in parallel, and cache the result per build hash.

Game updates

- **MUST** AXIOM-14.10 detect the game build exactly (file hash or version resource) and list supported builds in the provider manifest.
- **MUST** AXIOM-14.11 treat a new game build as unsupported until the signatures are verified on it; the launcher tells players to wait for a provider update.
- **SHOULD** AXIOM-14.12 run the provider conformance suite against every supported build in CI, and keep at least the last two builds of a live-service game supported while players update.

```
# Signatures/1.18.4-0b7c2e.toml
[build]
version = "1.18.4"
sha256 = "0b7c2e..."

[[signature]]
name = "Player::ApplyDamage"
pattern = "48 89 5C 24 ?? 57 48 83 EC 30 8B FA"
matches = 1
```

15

Game API modules: exposing a game to plugins

The game API module is the public face of a game provider: the functions, types, events and hook points plugins use for that game. It is a public API, held to the same standard as `axiom.h`.

One definition

- **MUST** AXIOM-15.1 define the module in `spec/` (IDL). `AxiomGen` generates the C header (`axiom/mygame.h`), the size asserts, every SDK binding and the reference pages from it. Generated files **MUST NOT** be edited by hand.
- **MUST** AXIOM-15.2 land a spec change, its generated output and an example plugin that uses it in the same pull request.

Design

- **MUST** AXIOM-15.3 expose game concepts, not game internals: `ax_mygame_player_get_health(player)` rather than a pointer to the game's `Player` struct or an offset.
- **MUST** AXIOM-15.4 use stable identifiers that survive game updates (names, hashed names, provider-assigned IDs), never raw addresses.
- **MUST** AXIOM-15.5 separate queries (no side effects) from commands (change state), and state which phases each may be called in.
- **SHOULD** AXIOM-15.6 publish events for state changes plugins would otherwise poll (`player.health.changed`), and hook points for behaviour they may change (`mygame.damage.apply`).
- **SHOULD** AXIOM-15.7 offer batched calls for things plugins do in loops (`ax_mygame_entities_query` filling a caller buffer), because each boundary crossing from WASM or Lua has a fixed cost.
- **MUST** AXIOM-15.8 gate writes and anything that affects multiplayer or other players behind capabilities (`game:write`, `game:network`).
- **MUST** AXIOM-15.9 keep the module the same across supported game builds; differences between builds are the provider's problem, hidden behind the module.

Versioning

CHANGE	BUMP	EXISTING PLUGINS
New function, event, hook point or optional trailing struct field	Minor	Keep working
Bug fix that keeps the documented behaviour	Patch	Keep working
Remove, rename, reorder or change the meaning of anything released	Major	Must be updated; keep the previous major available for at least one year

- **MUST** AXIOM-15.10 version the game API module independently of the provider binary and of Axiom; a game update that only needs new signatures does not change the module version.
- **MUST** AXIOM-15.11 deprecate before removing: mark it in the spec, warn once per plugin at load, and keep it working until the next major.

16

Writing a language provider

A language provider lets plugins be written in one language. It loads plugin code, runs it safely, and bridges every Axiom and game API call.

The contract

FUNCTION	MUST
<code>load</code>	Validate and load the plugin's code; no plugin code runs yet
<code>call</code>	Invoke a plugin callback with arguments marshalled from the C ABI
<code>limits</code>	Apply the plugin's memory, instruction and time quotas
<code>usage</code>	Report memory, CPU time and call counts per plugin
<code>save_state</code> / <code>restore_state</code>	Support hot reload where the language allows it
<code>unload</code>	Free every resource the plugin had, even if it misbehaved

Rules

- MUST** AXIOM-16.1 expose exactly the API the C ABI defines, generated by AxiomGen from the same spec. A function available to native plugins but missing in your language is a bug.
- MUST** AXIOM-16.2 give every plugin its own isolated instance (a WASM instance, a Lua state); plugins never share globals.
- MUST** AXIOM-16.3 enforce quotas: memory caps, and fuel, epoch or instruction-count interruption so an infinite loop is stopped and the plugin disabled, not the game frozen.
- MUST** AXIOM-16.4 remove ambient authority from the language: no WASI file, clock, random or network imports and no Lua `os`, `io`, `debug`, `package.loadlib` or FFI, except as granted by capabilities.
- MUST** AXIOM-16.5 bounds-check every (pointer, length) pair a guest passes, and copy data out of guest memory before using it.
- MUST** AXIOM-16.6 convert language-level failures (WASM traps, Lua errors, panics) into an error result plus a log entry with the plugin's own stack trace.
- MUST** AXIOM-16.7 ship an SDK for the language with idiomatic wrappers: scoped handles, typed events, the language's normal error type and async model.

- **SHOULD** AXIOM-16.8 support hot reload in development builds: save state, unload, load the new code, restore.
- **MUST** AXIOM-16.9 embed the engine (WASM runtime, Lua VM) behind your provider's own interface and pin its version, so it can be upgraded without changing plugins.

The hosted .NET language provider

- **MUST** AXIOM-16.10 host the .NET runtime once per game process (through `hostfxr`), pinned to one supported .NET LTS version that the provider ships with; plugins never bring their own runtime.
- **MUST** AXIOM-16.11 load each plugin into its own collectible `AssemblyLoadContext` so it can be unloaded and hot-reloaded, and **MUST** release every GC handle and delegate pointing into a plugin before unloading it, or the context will never collect.
- **MUST** AXIOM-16.12 marshal across the C ABI with `[UnmanagedCallersOnly]` entry points and blittable structs generated by AxiomGen; no `Marshal.GetDelegateForFunctionPointer` or COM interop at the boundary.
- **MUST** AXIOM-16.13 catch every exception at the boundary and never let one unwind into native frames.
- **MUST** AXIOM-16.14 treat hosted .NET plugins as trusted (tier 3): signed packages only, `process:native` capability required.
- **SHOULD** AXIOM-16.15 run plugin callbacks on the game thread without allocating, keeping per-frame paths free of LINQ, closures and boxing so the .NET GC does not cause frame spikes.

C# SDK

- **MUST** AXIOM-16.16 target the same API surface for both C# paths: one `Axiom` NuGet package whose code builds for the WASI target and for hosted .NET.
- **MUST** AXIOM-16.17 follow the standard .NET conventions in C# plugins: PascalCase public members, `async` / `await` for tasks (mapped onto Axiom's task pool), nullable reference types enabled, warnings as errors.
- **SHOULD** AXIOM-16.18 expose events as C# events or `IObservable<T>`, and scoped resources as `IDisposable`, so `using` releases them automatically.

17

Capabilities and security

Players install plugins from strangers. Capabilities let them see and limit what each plugin can do.

CAPABILITY	GRANTS
<code>game:read</code>	Read game state through game API modules
<code>game:write</code>	Change game state and use <code>pre</code> / <code>post</code> hooks
<code>game:network</code>	Send or alter multiplayer traffic (denied on servers that forbid it)
<code>ui:draw</code>	Draw overlays and windows
<code>input:read</code>	Read keyboard, mouse and controller input
<code>storage:plugin</code>	Read and write the plugin's own data folder
<code>fs:read</code> / <code>fs:write</code>	Read or write files outside the plugin's folder, through a picker the player approves
<code>net:connect</code>	Open outbound connections to the hosts the manifest lists
<code>process:native</code>	Run as a native (tier 3) plugin

For plugin authors

- **MUST** AXIOM-17.1 request the fewest capabilities that work, and explain every optional one in the manifest.
- **MUST** AXIOM-17.2 handle `AX_RESULT_DENIED` gracefully: turn the feature off and tell the player, never retry in a loop.
- **MUST** AXIOM-17.3 list the exact hosts a plugin connects to (`[network] hosts = ["api.acme.example"]`) and use TLS.
- **MUST NOT** AXIOM-17.4 collect personal data, telemetry or crash reports without the player's opt-in, and **MUST** say what is sent in the README.
- **MUST NOT** AXIOM-17.5 include obfuscated code, downloaders that fetch new code at runtime, or anything that bypasses a game's anti-cheat or multiplayer rules.

For provider authors

- **MUST** AXIOM-17.6 enforce capabilities in the provider for every call that needs one; never trust the plugin's SDK to have checked.
- **MUST** AXIOM-17.7 load native (tier 3) plugins only when signed by a trusted key or explicitly allowed by the player in developer mode.
- **MUST** AXIOM-17.8 verify package signatures and hashes before loading any plugin or provider.
- **MUST** AXIOM-17.9 refuse to run plugins with `game:write` or `game:network` in online modes the game provider marks as protected.

18

Performance budgets

A plugin shares the frame with the game and every other plugin. Budgets keep one plugin from costing everyone their frame rate. The numbers below are starting defaults; a game provider may tighten them for its game.

WHERE	BUDGET PER PLUGIN	WHAT HAPPENS WHEN EXCEEDED
Work on the game thread per frame	0.5 ms average, 2 ms peak	Warning in the overlay; disabled after sustained overruns
A single event handler or hook	0.1 ms	Logged with the handler name in development builds
<code>on_load</code> / <code>on_enable</code>	50 ms	Warning; move work to a task
Memory (WASM and Lua)	64 MiB default, raised in the manifest up to 512 MiB	Allocation fails; plugin disabled if it cannot recover
Hooks installed by a game provider	No measurable cost when no plugin uses the hook point	Provider bug

- **MUST** AXIOM-18.1 stay within budget on the reference machine the game's minimum specs describe.
- **MUST** AXIOM-18.2 measure before optimising, with `axiom profile` or the in-game overlay, and include before and after numbers in performance pull requests.
- **SHOULD** AXIOM-18.3 cache results that do not change every frame, and spread periodic work across frames instead of doing it all on one.
- **SHOULD** AXIOM-18.4 use batched game API calls in WASM and Lua plugins; each boundary crossing has a fixed cost.
- **MUST NOT** AXIOM-18.5 allocate on hot paths in native plugins and providers (per-frame handlers, hook trampolines); reuse buffers.
- **MUST** AXIOM-18.6 in providers: make a hook point cost nothing when no plugin subscribes to it, by checking a subscriber count before dispatching.

19

Logging and diagnostics

- **MUST** AXIOM-19.1 log through Axiom's logger (`ax_log` , or the SDK's `log` macros); never `printf` , `std::cout` , `print` or the game's own console. Axiom tags each line with the plugin or provider ID.
- **MUST** AXIOM-19.2 use levels consistently: `trace` and `debug` for development detail, `info` for lifecycle and rare player-relevant events, `warn` for handled problems, `error` for failures that disabled a feature.
- **MUST NOT** AXIOM-19.3 log every frame at `info` or above. Rate-limit repeated messages (`ax_log_once` , `log::warn_every!(5s, ...)`).
- **MUST NOT** AXIOM-19.4 log secrets, tokens, or personal data such as player names or chat in non-debug levels.
- **SHOULD** AXIOM-19.5 add structured fields instead of formatting them into text: `log::info!(zoom = self.zoom; "minimap enabled")` .
- **SHOULD** AXIOM-19.6 expose a debug panel in the in-game overlay (`ax_debug_panel_add`) for plugin state, instead of verbose logging.
- **MUST** AXIOM-19.7 in providers: report per-plugin frame time, memory and hook counts to Axiom's diagnostics so the overlay and `axiom profile` can show them, and include the provider version and game build in every crash report.

20

Testing plugins and providers

Plugins

- **MUST** AXIOM-20.1 unit-test plugin logic with the language's normal test runner, tests beside the code (`minimap.test.rs`, `minimap.test.lua`).
- **MUST** AXIOM-20.2 test behaviour against the **mock host** (`axiom test`): a fake core and fake game API module that let tests fire events, call hooks, advance frames and assert on what the plugin did.
- **MUST** AXIOM-20.3 test the full lifecycle at least once: load → enable → disable → enable → disable → unload, asserting nothing leaks and nothing stays hooked.
- **MUST** AXIOM-20.4 test the denied-capability and missing-optional-dependency paths.
- **SHOULD** AXIOM-20.5 run an in-game smoke test (`axiom run --game mygame --headless --script smoke.toml`) before each release when the game supports it.

```
#[axiom::test]
fn zoom_resets_on_disable(host: &mut MockHost) → Result<()> {
    let plugin = host.load::<BetterMinimap>()?;
    host.enable(&plugin)?;
    host.emit(player::HealthChanged { old: 100, new: 40 });
    host.advance_frames(3);
    host.disable(&plugin)?;
    assert!(host.ui().windows_of(&plugin).is_empty());
    assert_eq!(host.hooks_of(&plugin).len(), 0);
    Ok(())
}
```

Game providers

- **MUST** AXIOM-20.6 pass the **provider conformance suite**: lifecycle, attach and detach, hook install and removal, event delivery, capability checks and crash isolation.
- **MUST** AXIOM-20.7 verify every signature file against its game build in CI (match counts and expected bytes) and fail on any drift.
- **MUST** AXIOM-20.8 test each hook by calling the original through it with known arguments and comparing the result to an unhooked call.
- **SHOULD** AXIOM-20.9 keep a set of real community plugins as a compatibility corpus and run it against every provider release.

Language providers

- **MUST** AXIOM-20.10 run the shared API conformance tests through your language's SDK, proving every ABI function is reachable and behaves the same as from C.
- **MUST** AXIOM-20.11 test quota enforcement: an infinite loop, a memory bomb and a deep recursion are each stopped and the plugin disabled, with the game still running.
- **MUST** AXIOM-20.12 fuzz the marshalling layer with malformed guest pointers and lengths.

CI

- **MUST** AXIOM-20.13 build and test on every supported platform, and run providers' tests under ASan and UBSan builds as well as the release build.

Packaging, versioning and publishing

Versioning

CHANGE TO A PLUGIN	BUMP
Bug fix, no change to config, saves or services	Patch
New feature, new config key with a default, new service method	Minor
Removed feature, renamed config key, changed save format or service contract	Major

- **MUST** AXIOM-21.1 follow semantic versioning for plugins, providers, game API modules and services.
- **MUST** AXIOM-21.2 migrate a plugin's saved data and config forward automatically on upgrade, and keep a schema version in every saved file.
- **MUST** AXIOM-21.3 keep a `CHANGELOG.md` generated from Conventional Commits, with player-facing notes at the top of each release.

Packages

- **MUST** AXIOM-21.4 build packages with `axiom pack` only. A `.axmod` contains the manifest, the compiled plugin (`.wasm`, Lua sources, or one native library per platform), assets, licences and a file hash list.
- **MUST** AXIOM-21.5 sign every published package with the author's registry key; native plugins and all providers **MUST** be signed.
- **MUST** AXIOM-21.6 make builds reproducible: pinned toolchains, lock files, no timestamps or absolute paths in the output.
- **MUST NOT** AXIOM-21.7 include debug symbols, source maps or test data in the published package; publish symbols separately for crash reports.

Publishing

- **MUST** AXIOM-21.8 fill in the registry listing: description, screenshots, supported games and builds, capabilities with reasons, source link and licence.
- **MUST** AXIOM-21.9 publish open-source plugins with an OSI licence file at the root; closed-source plugins are allowed but cannot request `process:native`.
- **SHOULD** AXIOM-21.10 release to a `beta` channel first and promote to `stable` after it has run without new crash reports.

22

Git, releases and review

Commits and branches

- **MUST** AXIOM-22.1 follow **Conventional Commits**: `type(scope): summary`, imperative, lower case, no trailing period, under 72 characters. Scope is the subsystem: `feat(waypoints): add party sync`, `fix(signatures): update for 1.18.4`.
- **MUST** AXIOM-22.2 mark breaking changes with `!` and a **BREAKING CHANGE:** footer; these require a major version bump.
- **MUST** AXIOM-22.3 keep commits small and focused, each building and passing tests.
- **MUST** AXIOM-22.4 branch from `main` with a typed name (`feat/party-sync`, `fix/1.18.4-signatures`), and tag releases `v<semver>`.

Review checklist: plugins

- Does `on_disable` undo everything `on_enable` did, with no leaks or hooks left behind?
- Are capabilities minimal, explained, and is denial handled?
- Does every handler stay within its budget and keep heavy work off the game thread?
- Are events, config keys and services namespaced under the plugin ID?
- Are errors handled, and are player-facing messages clear?

Review checklist: providers

- Does it fail closed on unknown game builds and ambiguous signatures?
- Are all hooks installed through the hook manager and removed on stop?
- Is every plugin call isolated, and are capabilities enforced in the provider?
- Are game API changes made in the spec, generated, versioned and covered by an example plugin?
- Does the conformance suite pass on every supported build and platform?

A

Appendix

A. Minimal WASM plugin (Rust)

```
// src/lib.rs
use axiom::prelude::*;

#[axiom::plugin]
struct HelloAxiom;

impl Plugin for HelloAxiom {
    fn on_load(_ctx: &mut Ctx) → Result<Self> {
        Ok(Self)
    }

    fn on_enable(&mut self, ctx: &mut Ctx) → Result<()> {
        log::info!("hello from {}", ctx.plugin_id());
        Ok(())
    }
}
```

```
# axiom.toml
[plugin]
id = "acme.hello-axiom"
name = "Hello Axiom"
version = "0.1.0"
description = "Logs a greeting when enabled."
authors = ["acme"]
license = "MIT"
language = "wasm"
entry = "hello_axiom.wasm"

[requires]
axiom = "≥1.4, <2"

[load]
phase = "game"

[capabilities]
required = []
```

B. Minimal Lua plugin

```
-- main.lua
local plugin = axiom.plugin()

function plugin:on_enable(ctx)
    ctx.events:on("player.health.changed", function(ev)
        if ev.new < 20 then
            ctx.ui:toast("Low health!")
        end
    end)
end

return plugin
```

B2. Minimal C# plugin

```
// HelloAxiom.cs
using Axiom;
using Axiom.MyGame;

[Plugin("acme.hello-axiom")]
public sealed class HelloAxiom : Plugin
{
    private IDisposable? _onHealth;

    public override void OnEnable(PluginContext ctx)
    {
        // Removed automatically on disable; disposing early is fine too.
        _onHealth = ctx.Events.On<Player.HealthChanged>(ev =>
        {
            if (ev.New < 20)
            {
                ctx.Ui.Toast("Low health!");
            }
        });
    }

    public override void OnDisable(PluginContext ctx) => _onHealth?.Dispose();
}
```

C. New plugin checklist

- ☐ Started from `axiom new plugin`; WASM unless the README says why not.
- ☐ Plugin ID chosen and final; events, config and services namespaced under it.
- ☐ Manifest declares version ranges, load order and minimal capabilities with reasons.
- ☐ Lifecycle test passes: enable/disable twice, no leaks, no hooks left.

- ☐ Within the frame budget on minimum-spec hardware.
- ☐ README, CHANGELOG and licence present; package signed.

D. New game provider checklist

- ☐ Detects the game and exact build from files only.
- ☐ Signatures in data files per build, verified in CI, failing closed.
- ☐ All hooks through the hook manager, removed on stop; process state restored on detach.
- ☐ Game API module defined in `spec/`, generated, versioned, with an example plugin.
- ☐ Capabilities enforced for writes and network; protected online modes respected.
- ☐ Conformance suite passing on every supported build and platform.

E. Sources

- [Semantic Versioning 2.0.0](#)
- [Conventional Commits 1.0.0](#)
- [WebAssembly specification](#)
- [Lua 5.4 reference manual](#)