# Strategic Architecture and Implementation Roadmap for Extensible Software Systems

## 1. Executive Summary

In the contemporary landscape of software engineering, the capacity for end-user modification—commonly referred to as "modding"—has transcended its origins in enthusiast gaming communities to become a critical architectural requirement for long-lived, resilient software platforms. The request to gather information for "moding" (modification) and adapt it for a new project necessitates a comprehensive analysis that bridges the gap between legacy software maintenance definitions and modern, dynamic extensibility frameworks. Historically, "moding" in software maintenance referred to the implementation of modifications to correct faults or improve capabilities after operational release.[1] Today, this concept has evolved into "User-Generated Content" (UGC) and plugin ecosystems, where the architectural goal is to empower external developers to extend the software's lifespan and utility without compromising its core stability.

The strategic imperative for adopting a moddable architecture lies in the decentralization of value creation. By exposing internal systems through secure, stable Application Programming Interfaces (APIs), an organization effectively transforms its product into a platform. This shift allows for a distributed development model where the community can implement niche features, fix minor bugs, and produce content at a scale that a centralized engineering team cannot match.[2] However, achieving this requires a fundamental shift in architectural thinking—from monolithic, tightly coupled designs to modular, event-driven, and data-oriented structures.

This report provides an exhaustive technical anThe "Modding Framework Project Plan" is a robust and comprehensive technical analysis. To transform it into a complete "reference library" and enhance its utility for future development, I recommend expanding the following key areas. These suggestions will add strategic, organizational, and deeper technical context.I. Architectural & Technical Deep Dives

These enhancements will make the core technical implementation guide more complete and resilient.

- **Formal API Stability and Versioning Policy:**
    - **Develop Further:** Dedicate a section to a formal promise about the API's

stability. For instance: "Major versions will be supported for a minimum of 2 years."
    - **Add:** A detailed process for deprecating API features (e.g., three major version warnings before removal). This is crucial for retaining the modding community's trust.
- **Dependency Resolution Deep Dive:**
    - **Develop Further:** Expand on the technical solution for **Milestone 3.3 (Dependency Resolution)**. Detail the chosen algorithm (e.g., *Topological Sort* of a dependency graph) and how conflicts (e.g., Mod A requires v1.x, Mod B requires v2.x of the same library) will be handled by the Mod Loader, especially within the **AssemblyLoadContext (ALC)** structure mentioned in Section 3.2.
- **Error Handling and Crash Reporting:**
    - **Add:** A dedicated sub-section in **Section 7 (DX and Tooling)** covering standardized error logging. Define the format for crash reports (e.g., including stack trace, mod list, and configuration) and the mechanism for mods to log warnings/errors back to the host application's console/log file.
- **Platform and Localization (L10N/I18N) Strategy:**
    - **Add:** A new section on how the modding system will support multiple operating systems/platforms and, critically, how mods can integrate with the application's localization system (e.g., defining a standard convention for mods to add their own translation files).

II. Project Management & Strategy

These additions turn the technical plan into a complete business and project document.

- **Risk Analysis Register:**
    - **Add:** A formal table listing high-impact risks and their mitigation strategies.
        - *Example Risk:* Malicious or poorly-written C# mods exploit the "Low (Process Access)" security in-process (Section 3.2).
        - *Mitigation:* A mandatory code submission/review process for C# mods, or prioritizing the **Out-of-Process Isolation** model (Section 6.1) despite complexity.
- **Team and Resource Allocation:**
    - **Add:** An outline of the required engineering roles for each phase (e.g., *Core Architect*, *Security Engineer*, *Tooling Developer*). This helps in resource planning for the project.
- **Go-to-Market / Community Adoption Plan:**
    - **Develop Further:** Expand **Phase 4** to include explicit strategies for launch:
        - **Beta/Early Access:** How to engage a small group of trusted modders (an "Alpha Modder Program") before public release.
        - **Launch Day:** The plan for announcing the SDK, hosting tutorials, and

driving initial community engagement.

III. Ecosystem & Community Development

Since the success of a modding framework depends on its community, this area needs detailed planning.

- **User-Generated Content (UGC) Legal & Licensing:**
  - **Add:** A section addressing the legal framework. Define what license (e.g., MIT, GPL, proprietary) mods must use, and the policy for intellectual property (IP) rights over mod content. This is essential for protecting the company and the community.
- **Mod Distribution and Marketplace Plan:**
  - **Develop Further:** Expand on the necessity of a central mod hub. Detail the feature set for a hypothetical "Mod Portal," including:
    - Upload and version tracking.
    - User ratings and comments.
    - **Automated Mod Compatibility Checker:** A system that reads the mod manifest (Milestone 3.3) and prevents a user from installing a mod that requires a version of the host application they do not have.
- **Mod Moderation and Reporting:**
  - **Add:** A section outlining the process for identifying and removing harmful or malicious mods (e.g., those that attempt to bypass the sandbox or contain illegal content). This directly links to the security concerns raised in **Section 6**.

alysis and project plan for building a robust modding framework. It synthesizes insights from successful extensible systems—including Visual Studio Code, Minecraft Forge, and Factorio—to establish a "Gold Standard" for modding architecture. The analysis identifies **Event-Driven Architecture (EDA)** [3], **Dependency Injection (DI)** [4], and the **Leave-and-Layer** modernization pattern [5] as the foundational pillars of such a system. Furthermore, it evaluates the trade-offs between disparate runtime environments, specifically comparing the ease of **Lua**, the raw power of **C#/.NET**, and the emerging security of **WebAssembly (WASM)**.

Crucially, this document addresses the "unspoken" challenges of modding: the complexity of dependency resolution [6], the necessity of strict data lifecycle management to prevent race conditions [7], and the security implications of executing untrusted code.[8] It concludes with a detailed, phased project plan designed to guide the engineering team from architectural inception to the deployment of a thriving developer ecosystem.

## 2. Architectural Paradigms for Extensibility

The foundation of a moddable system is the rigorous separation of concerns. In a standard monolithic application, business logic, data persistence, and presentation layers are often tightly coupled. To enable modding, these layers must be decoupled using specific architectural patterns that allow external code to intercept, modify, or replace internal behaviors.

### 2.1. The Microkernel and Plugin-Based Architecture

The most effective architectural pattern for high degrees of extensibility is the **Plugin-based Architecture**, often implemented as a variation of the Microkernel pattern.[3] In this model, the "host" application is reduced to a minimal core responsible for managing the lifecycle of the application and the plugins. The "features" of the application—whether they are inventory systems in a game or data processing modules in a business tool—are implemented as internal plugins.

This approach enforces a discipline often referred to as "eating your own dog food." If the internal development team uses the same API as external modders, the API is naturally battle-tested and robust. The core system defines the contracts (interfaces), and the plugins provide the implementations. This aligns with the **Dependency Inversion Principle**, where high-level modules (the engine) do not depend on low-level modules (specific features); both depend on abstractions.[9]

#### 2.1.1. Hexagonal Architecture (Ports and Adapters)

Complementing the plugin model is the **Hexagonal Architecture**. This pattern isolates the core domain logic from external dependencies (inputs and outputs) using "Ports" and "Adapters".[3]

- **Ports:** These are the interfaces defined by the application core (e.g., IInputProvider, IRenderer).
- **Adapters:** These are the implementations. A mod can provide a new Adapter. For instance, if the core logic requires a pathfinding algorithm (Port), a mod can supply a highly optimized A* implementation (Adapter) that replaces the default one.

This architecture is particularly valuable when "moding" involves adapting the software to new environments or hardware, as noted in legacy software maintenance contexts.[1] By treating the mod as just another adapter, the core logic remains pristine and testable.

### 2.2. Event-Driven Architecture (EDA)

Static coupling is the antithesis of modding. If Class A calls Class B directly, a modder cannot intervene without overwriting Class A. **Event-Driven Architecture (EDA)** solves this by introducing an intermediary: the Event. Services communicate by emitting events that other services (or mods) consume.[3]

### 2.2.1. Taxonomy of Event Patterns

To implement a robust modding system, one must distinguish between different categories of events, as utilizing a single type for all interactions leads to architectural fragility [12]:

- **Notification Events:** These are "fire-and-forget" signals indicating that a state change has occurred (e.g., FileSaved, EnemySpawned). Mods subscribe to these to trigger side effects, such as playing a sound or logging a metric. They are asynchronous in nature and do not block the core loop.
- **Interception (Cancellable) Events:** These are synchronous events fired *before* an action is committed (e.g., PreDamageEvent). The event object carries mutable state, such as a IsCancelled boolean or a DamageAmount float. Mods can modify these values. If a mod sets IsCancelled = true, the core system aborts the action. This pattern is the backbone of gameplay modification, allowing mods to alter rules without rewriting engine code.[13]
- **Data Transformation Events:** These events pass a data structure through a chain of subscribers, allowing each to mutate it. This is commonly used for loot generation or dynamic UI construction. For example, a LootTableGenerationEvent might be passed to five different mods, each adding their own items to the list before the chest is finally populated.

### 2.2.2. The Event Bus Implementation

The mechanism for dispatching these events is the **Event Bus**. While simple **Observer Patterns** (callbacks) can suffice for small systems [14], a dedicated Event Bus provides the necessary decoupling for a complex modding API.

In a C# environment, a naive implementation using standard.NET events (public event EventHandler ActionHappened) is often insufficient because it creates strong references, leading to memory leaks if mods are unloaded but not unsubscribed. A robust Event Bus implementation should utilize **Weak References** or a token-based subscription model to ensure that the engine does not hold onto mod memory after the mod has been unloaded.[15]

Furthermore, for networked or multi-process architectures (like Visual Studio Code's extension host), the Event Bus must be capable of serializing events and transmitting them across process boundaries via IPC (Inter-Process Communication).[16] This introduces latency, necessitating an asynchronous design (async/await) for the API to prevent the main UI thread from freezing while waiting for a mod to process an event.[5]

## 2.3. The "Leave and Layer" Pattern

For projects that involve adapting an existing codebase rather than starting from scratch, the **Leave and Layer** pattern is the strategic choice.[5] Rewriting a monolithic legacy application to be fully plugin-based is often prohibitively expensive and risky.

Instead, the "Leave and Layer" approach involves leaving the existing legacy core unchanged. A new "Modding Layer" or "Facade" is built alongside it.

- **Mechanism:** When the legacy system performs an action, calls are injected (hooked) to divert control to the new layer.
- **Benefit:** This reduces regression risks. The legacy code is a "black box." The new layer handles the complexity of the modding API, versioning, and security. If a mod crashes the new layer, the legacy core can theoretically continue to function or degrade gracefully.
- **Migration:** Over time, functionality can be migrated from the legacy core into the new layer, gradually strangling the monolith (a variation of the **Strangler Fig Pattern**).[5]

---

## 3. Runtime Environments and Scripting Languages

The selection of the runtime environment for mods—the language in which mods are written and the engine that executes them—is the most significant technical decision in the project planning phase. It dictates performance profiles, security models, and the "Developer Experience" (DX) for the community. The three primary candidates identified in the research are **Lua**, **C#/.NET**, and **WebAssembly (WASM)**.

### 3.1. Lua: The Standard for Logic Scripting

Lua is the incumbent standard for game modding (used in World of Warcraft, Factorio, Roblox) due to its design as an embedded scripting language.[17]

**Technical analysis:**

- **Embedding:** Lua is designed to be embedded in C/C++ host applications, but libraries like MoonSharp, NLua, or Sol2 allow seamless integration with C#/.NET environments.[17]
- **Performance:** Lua is an interpreted language (though LuaJIT offers Just-In-Time compilation). It is significantly slower than native code for heavy computation but is sufficient for high-level game logic.
- **Ease of Use:** Lua's syntax is minimal and forgiving, making it accessible to non-programmers. This lowers the barrier to entry for the modding community.[19]
- **Safety:** Lua operates in a managed state (lua_State). The host application can create a "Sandbox" by removing dangerous functions (like os.execute, io.open) from the global environment (_ENV) exposed to the mod script.[20] This prevents mods from accessing the user's file system or network, providing a baseline of security.

**Best Use Case:** Projects where mods are primarily content-driven or logic tweaks (e.g., changing damage formulas, adding dialogue) and do not require heavy computation.

## 3.2. C# and.NET Assemblies: Deep Integration

For applications already written in.NET, allowing mods to be written in C# (as compiled DLLs) offers the deepest possible integration.

**Technical Analysis:**

- **Performance:** C# mods run at near-native speed, leveraging the.NET JIT compiler. This allows mods to perform complex tasks like custom physics simulations or procedural terrain generation that would be too slow in Lua.[21]
- **Tooling:** Modders can utilize the full power of the.NET ecosystem: Visual Studio, NuGet packages, strong typing, and advanced debugging.[22]
- **Architecture - AssemblyLoadContext (ALC):** In modern.NET (Core and 5+), the AppDomain system has been replaced by AssemblyLoadContext.[23] This class is crucial for mod isolation.
    - **Dependency Isolation:** Each mod can be loaded into its own ALC. This allows Mod A to use Newtonsoft.Json v12.0 and Mod B to use v13.0 without causing a FileLoadException due to version conflict. The runtime isolates the dependencies of each context.[25]
    - **Unloadability:** ALC supports unloading. This enables **Hot Reloading**, where a mod developer can recompile their code, and the engine unloads the old context and loads the new one without restarting the application.[26]

**Risks:**

- **Security:** By default, a loaded assembly has full access to the process memory and the OS user's privileges. It can read files, open network sockets, and use Reflection to access private members of the host engine.
- **Mitigation:** Security must be enforced via **Roslyn Analyzers**.[27] These are static analysis tools that run during the mod's compilation. The host developers provide an Analyzer that flags and blocks usage of banned APIs (e.g., System.IO, System.Reflection). This creates a "compile-time sandbox," though it is less secure than a runtime sandbox.[28]

## 3.3. WebAssembly (WASM): The Secure Polyglot Future

WebAssembly is emerging as a powerful alternative for modding, offering a balance between performance and strict security.

**Technical Analysis:**

- **Polyglot Support:** Mods can be written in any language that compiles to WASM (Rust, C++, Go, C#, TypeScript).[30] This widens the potential community of modders.
- **Sandboxing:** WASM executes in a strict virtual machine. It uses a capability-based security model. The WASM module cannot access any memory outside its own linear memory instance.

- **WASI (WebAssembly System Interface):** To perform useful work (like reading files), the host implements WASI. This allows granular permission control. The host can grant a mod read/write access *only* to a specific directory (e.g., /mods/my-mod-data/) and block all other file access. This makes it safe to run untrusted code from the internet.[8]

**Trade-offs:**

- **Interop Overhead:** Calling between the Host (C#) and the Guest (WASM) involves marshalling data across the memory boundary. This "Context Switching" cost can be high if the API is "chatty" (many small calls). APIs must be designed to be "chunky" (fewer calls, larger data payloads) to mitigate this.[32]

## 3.4. Comparative Analysis Table

The following table summarizes the trade-offs to assist in the selection process:

| **Feature** | **Lua** | **C# (.NET Core)** | **WebAssembly (WASM)** |
|---|---|---|---|
| **Performance** | Low (Interpreted/JIT) | High (Native JIT) | High (Near-Native) |
| **Security** | Medium (Env Sandboxing) | Low (Process Access) | High (VM Sandboxing) |
| **Isolation** | Logical (State) | Loader (ALC) | Memory (VM) |
| **Ease of Use** | High (Simple Syntax) | Medium (Requires Toolchain) | Medium (Requires Compiler) |
| **Hot Reload** | Excellent (Text Eval) | Good (Unloadable ALC) | Excellent (Module Swap) |
| **Tooling** | Basic (Text Editors) | Advanced (VS/Rider) | Fragmented (Language dependent) |
| **Data Marshalling** | Slow (Proxy Objects) | Fast (Shared Memory) | Slow (Serialization) |

# 4. Data Lifecycle and State Management

A recurring failure mode in modding architectures is the conflation of **Data Definition** (what an object is) and **Control Logic** (what an object does). Successful engines like Factorio and Unity utilize a strict separation of these stages to ensure stability.

## 4.1. The "Data Stage" vs. "Control Stage" Pattern

The research highlights Factorio's "Data Lifecycle" as a best practice for managing modded content.[7]

1. **The Data Stage (Prototype Definition):**
    - This occurs strictly at application startup.
    - Mods load scripts that define "Prototypes." These are data structures representing entities, items, or recipes.
    - **Constraint:** No game logic runs here. Mods cannot verify player positions or spawn entities. They can only populate the global data table.
    - **Validation:** Once all mods have loaded, the engine "bakes" this data. It resolves inheritance (e.g., SuperSword inherits from BaseSword), validates references (does the texture exist?), and optimizes the data for runtime.[34]
    - **Outcome:** A read-only, optimized database of all game objects is created.
2. **The Control Stage (Runtime Execution):**
    - This occurs during the gameplay loop.
    - Mods access the baked data as read-only.
    - Scripts respond to events (OnTick, OnPlayerInteract).
    - **Constraint:** Mods cannot define new Prototypes here. This prevents "race conditions" where one mod expects an item to exist that another mod hasn't created yet.

**Strategic Insight:** This separation is critical for multiplayer synchronization. It ensures that all clients have an identical definition of the game universe before simulation begins. If runtime definition were allowed, clients could drift out of sync, causing desynchronization bugs that are notoriously difficult to debug.[35]

## 4.2. Save Data Serialization and Versioning

Handling save data in a modded environment introduces the "Missing Mod" problem.

- **The Problem:** A user saves the game with "Mod A" installed, which adds a "Magic Sword." The user then uninstalls "Mod A" and loads the save. The save file contains data for an item that no longer exists in the engine's Prototype database.
- **Solution Strategies:**
    1. **Manifest Verification:** The save file header must include a list of all active mods and their versions. The engine checks this against the currently installed mods upon load.[36]

2. **Graceful Degradation:** The deserializer must implement a "Try-Load" pattern. If it encounters a chunk of data for an unknown ID, it should either discard it (logging a warning) or wrap it in a "Placeholder" object. The Placeholder preserves the data (as a binary blob) so that if the mod is reinstalled later, the item is restored.[37]
3. **Migration Scripts:** Mods must support versioning. When a mod updates from v1.0 to v2.0, it may need to restructure its save data. The modding API should provide a hook (e.g., OnMigration(oldVersion, newVersion)) where the mod can run logic to convert old data structures to the new format.[38]

---

## 5. Resource Management: Virtual File Systems (VFS)

Mods are not just code; they are often assets—textures, models, audio, and configuration files. A robust architecture must allow mods to inject these assets into the game without modifying the original game files on disk. This is achieved via a **Virtual File System (VFS)**.

### 5.1. The VFS Layering Mechanism

The VFS acts as an abstraction layer between the game engine and the physical disk. It aggregates multiple physical directories into a single logical directory tree.[40]

Implementation:
The VFS maintains a list of "Mount Points," each with a priority.

1. **Mount:** BaseGame/Assets/ -> Priority 0
2. **Mount:** Mods/TexturePackA/Assets/ -> Priority 10
3. **Mount:** Mods/MyMod/Assets/ -> Priority 20

Resolution Algorithm:
When the engine requests a file, e.g., Load("textures/player.png"), the VFS iterates through the mount points in descending order of priority:

1. Check Priority 20 (MyMod): File not found.
2. Check Priority 10 (TexturePackA): **File Found!** Return this file stream.
3. Check Priority 0 (BaseGame): Skipped.

This mechanism allows for **Asset Overrides**. A mod can replace a core asset simply by providing a file with the same relative path. It also supports "Partial Overrides," where a mod replaces only specific files while leaving the rest of the directory structure intact.[41]

### 5.2. Handling "Search Customs" and Mod Isolation

Tools like the Source Engine or specialized mod managers utilize "Search Paths" (Search Customs) to isolate mods. Instead of dumping all files into a single Data folder, each mod resides in its own isolated folder or compressed archive (e.g., .zip, .vpk).[41]

Project Plan Integration:
The VFS must be implemented early in the project lifecycle (Phase 2). Legacy file loading code (e.g., File.Open("C:/Game/Data/config.txt")) must be refactored to use the VFS API (e.g., VFS.Open("config.txt")). This effectively "virtualizes" the entire IO layer of the application.42

---

# 6. Security and Process Isolation

Allowing users to run arbitrary code is an inherent security risk. The architecture must prioritize the containment of this risk through process isolation and sandboxing.

## 6.1. The VS Code Model: Out-of-Process Isolation

For maximum stability, the "VS Code Model" is the industry benchmark. It isolates the mod execution environment from the main application process.[44]

- **Architecture:**
  - **Main Process:** Handles UI rendering, input, and OS interactions.
  - **Extension Host Process:** A separate child process (e.g., Node.js for VS Code, or a separate.NET process) where *all* extensions run.
- **Communication:** The two processes communicate via **RPC (Remote Procedure Call)** or **IPC (Inter-Process Communication)** using JSON or binary protocols over named pipes/standard IO.[45]
- **Fault Tolerance:** If a mod enters an infinite loop or causes a segmentation fault, it crashes only the Extension Host. The Main Process remains responsive, displays an error message to the user, and can restart the Extension Host without losing unsaved work.[16]

Adapting for Your Project:
Implementing this model requires defining a strict asynchronous API. Synchronous hooks (e.g., "OnKeyDown" preventing a character from appearing) are difficult to implement because the round-trip latency to the external process is perceptible to the user. This architecture forces the adoption of an asynchronous, event-driven design pattern.5

## 6.2. Sandboxing Implementation Details

If Out-of-Process isolation is too complex or performance-prohibitive, in-process sandboxing is the fallback.

- **Lua Sandboxing:**
  - Create a "Safe Environment" table containing only whitelisted functions (math, string, table, game_api).
  - Use setfenv (Lua 5.1) or _ENV (Lua 5.2+) to apply this environment to loaded chunks.
  - Explicitly exclude os.*, io.*, debug.*, and package.loadlib to prevent OS-level access.[20]

- **C# Roslyn Analyzers (Compile-Time Sandboxing):**
  - Since the.NET runtime does not support strict security boundaries within a single process (Code Access Security is deprecated), security relies on **Static Analysis**.
  - Develop a custom **Roslyn Analyzer** that scans mod source code during compilation.
  - The Analyzer checks the Semantic Model for invocations of banned symbols (e.g., System.IO.File, System.Net.Sockets).
  - If a banned usage is detected, the build fails. This prevents well-intentioned developers from accidentally using dangerous APIs, though it does not stop a determined attacker who constructs malicious IL code directly.[27]

---

# 7. Developer Experience (DX) and Tooling

A modding system is a product for developers. Its success depends entirely on the quality of the Software Development Kit (SDK) and documentation.

## 7.1. API Design: The Facade Pattern

Do not expose internal engine classes directly to mods. Internal code changes frequently, and such changes would break mods.

- **The Facade Pattern:** Create a dedicated API namespace. Classes in this namespace (e.g., API.Item) act as wrappers or proxies around the internal engine classes (e.g., Internal.Core.GameItem).
- **Stability:** This decouples the internal implementation from the public contract. You can refactor the engine entirely, and as long as the API.Item Facade maintains its interface, mods will not break.[47]

## 7.2. Documentation Generation

Documentation must be accurate and versioned.

- **Tools:**
  - **DocFX:** The modern standard for.NET. It generates static HTML documentation from XML code comments (/// <summary>) and Markdown files. It supports custom templates and versioning, allowing you to host docs for API v1.0 and v2.0 simultaneously.[49]
  - **Sandcastle:** An older alternative, primarily used for generating offline help files (CHM), but largely superseded by DocFX for web-based docs.[51]
- **Content:** Good documentation includes not just API references (auto-generated) but also conceptual guides ("How to create a Block," "Understanding the Event Loop") written in Markdown.[52]

### 7.3. Debugging and Hot Reload

To foster a vibrant community, the "Edit-Build-Run" loop must be minimized.

- **Hot Reload:** Implement FileSystemWatcher to monitor mod folders. When a script file changes, the engine should automatically reload the mod context. For C# mods using AssemblyLoadContext, this involves unloading the old ALC and loading the new DLL.[53]
- **Debugging:**
  - For Lua: Integrate a socket-based debug server (e.g., mobdebug) that allows VS Code to attach to the running game and hit breakpoints in Lua scripts.[55]
  - For C#: Ensure debugging symbols (PDBs) are loaded so modders can attach the Visual Studio debugger to the game process.[56]

---

## 8. Comprehensive Project Plan and Roadmap

The following roadmap outlines the phased implementation of a modding system, moving from architectural restructuring to community ecosystem building.

### Phase 1: Architectural Foundation (Months 1-3)

**Goal:** Decouple the core engine and prove the architecture with internal implementations.

| Milestone | Deliverable | Technical Focus |
|---|---|---|
| **1.1 Event Bus** | IEventBus interface and implementation. | Implement Pub/Sub pattern. Replace hardcoded calls with events. Focus on Cancellable event support. |
| **1.2 Dependency Injection** | DI Container integration. | Refactor Singleton Managers (GameManager.Instance) into injected services (IGameManager). |
| **1.3 Internal "Mod"** | A functional non-critical feature (e.g., Chat). | Implement a feature strictly using the new Event/DI system to validate the API ("Dog-Fooding"). |

## Phase 2: Data Pipeline and VFS (Months 4-6)

**Goal:** Enable asset modification and define the Data Lifecycle.

| Milestone | Deliverable | Technical Focus |
|---|---|---|
| **2.1 Virtual File System** | VFS Manager and Mount Points. | Refactor all File.Open calls to use VFS. Implement asset overriding logic. |
| **2.2 Data Loader** | Prototype Definition System. | Create the "Data Stage" logic. Define JSON/Script schemas for items/entities. |
| **2.3 Registry System** | GameRegistry for lookups. | Implement the "Baking" process that validates and freezes data after the Data Stage. |

## Phase 3: Scripting Runtime and Security (Months 7-9)

**Goal:** Enable dynamic logic execution and secure the environment.

| Milestone | Deliverable | Technical Focus |
|---|---|---|
| **3.1 Runtime Integration** | Embedded Lua/C# Host. | Integrate the chosen runtime (MoonSharp/ALC). Create the "Facade" API layer. |
| **3.2 Sandboxing** | Security Enforcement Module. | **C#:** Implement Roslyn Analyzers for banned APIs. **Lua:** Implement _ENV masking. |
| **3.3 Mod Lifecycle** | IMod Interface (Load/Unload). | Implement Mod Loader logic: Discovery, Dependency Resolution (Topological Sort), |

|  |  |  |
|---|---|---|
|  |  | Initialization. |

### Phase 4: Developer Ecosystem (Months 10-12)

**Goal:** Empower the community with tools and documentation.

| Milestone | Deliverable | Technical Focus |
|---|---|---|
| **4.1 Documentation** | DocFX Pipeline & Website. | Set up auto-generation from XML comments. Write "Getting Started" guides. |
| **4.2 Reference Mod** | Open Source Example Mod. | Release a "Hello World" mod and a complex "Feature" mod to Github as learning resources. |
| **4.3 Hot Reload** | Development Tools. | Implement file watchers to reload assets/scripts at runtime. |

---

## 9. Conclusion

Transforming a closed software project into an extensible platform is a rigorous architectural challenge that yields significant long-term dividends. By analyzing industry leaders like Factorio and VS Code, the path forward becomes clear: it requires a commitment to **Event-Driven Architecture**, a strict separation of **Data and Control**, and a robust **Plugin** model.

The recommended path is to adopt the **Leave and Layer** strategy, building a modern, moddable API layer alongside the existing core. The choice of runtime—**C#** for power and integration, or **Lua** for safety and ease of use—must align with the target audience's technical proficiency. Ultimately, the success of a modding project rests not just on the code, but on the ecosystem; prioritizing **Tooling**, **Documentation**, and **Hot Reload** capabilities is as vital as the engine architecture itself. This roadmap provides the structural framework to navigate that complexity and deliver a thriving, user-extended platform.

**Works cited**

1. Guidance and Control Software, - DTIC, accessed December 13, 2025, https://apps.dtic.mil/sti/tr/pdf/ADA088631.pdf
2. How to build "moddable" architecture into a Unity game (and games generally)? - Reddit, accessed December 13, 2025, https://www.reddit.com/r/gamedev/comments/vp3i11/how_to_build_moddable_architecture_into_a_unity/
3. 6 Software Architectural Patterns You Must Know - ByteByteGo, accessed December 13, 2025, https://bytebytego.com/guides/6-software-architectural-patterns-you-must-know/
4. A better architecture for Unity projects - Game Developer, accessed December 13, 2025, https://www.gamedeveloper.com/business/a-better-architecture-for-unity-projects
5. Modernizing Legacy Applications with Event-Driven Architecture: The Leave-and-Layer Pattern - AWS, accessed December 13, 2025, https://aws.amazon.com/blogs/migration-and-modernization/modernizing-legacy-applications-with-event-driven-architecture-the-leave-and-layer-pattern/
6. Algorithm for dependency resolution - Stack Overflow, accessed December 13, 2025, https://stackoverflow.com/questions/28099683/algorithm-for-dependency-resolution
7. Tutorial:Modding tutorial - Official Factorio Wiki, accessed December 13, 2025, https://wiki.factorio.com/Tutorial:Modding_tutorial/Gangsir
8. WebAssembly and Web Security: What Developers Should Know - PixelFreeStudio Blog, accessed December 13, 2025, https://blog.pixelfreestudio.com/webassembly-and-web-security-what-developers-should-know/
9. Functional Unity Architecture: A Developer's Guide | by Bruno Mikoski | Medium, accessed December 13, 2025, https://medium.com/@bada/functional-unity-architecture-a-developers-guide-359e5111c5c2
10. Build Your Own Dependency Injection in less than 15 Minutes | Unity C# - YouTube, accessed December 13, 2025, https://www.youtube.com/watch?v=PJcBJ60C970
11. 10 Event-Driven Architecture Examples: Real-World Use Cases - Estuary, accessed December 13, 2025, https://estuary.dev/blog/event-driven-architecture-examples/
12. The Ultimate Guide to Event-Driven Architecture Patterns - Solace, accessed December 13, 2025, https://solace.com/event-driven-architecture-patterns/
13. Creating an Event Handler - Forge Documentation, accessed December 13, 2025, https://mcforge-documantation-jp.github.io/events/intro/
14. Event-driven architecture and hooks in PHP - Stack Overflow, accessed December 13, 2025, https://stackoverflow.com/questions/6846118/event-driven-architecture-and-hoo

ks-in-php
15. What are some gameplay architecture best practices? : r/Unity3D - Reddit, accessed December 13, 2025, https://www.reddit.com/r/Unity3D/comments/gj0nwu/what_are_some_gameplay_architecture_best_practices/
16. From Learner to Contributor: Navigating the VS Code Extensions Structure | by Cha Jesse, accessed December 13, 2025, https://medium.com/@chajesse/from-learner-to-contributor-navigating-the-vs-code-extensions-structure-ed150f9897e5
17. A curated list of awesome Lua frameworks, libraries and software. - GitHub, accessed December 13, 2025, https://github.com/forhappy/awesome-lua
18. is lua really that good for modding? : r/gamedev - Reddit, accessed December 13, 2025, https://www.reddit.com/r/gamedev/comments/189hgsx/is_lua_really_that_good_for_modding/
19. What is the best lua framework for making websites in 2024? - Reddit, accessed December 13, 2025, https://www.reddit.com/r/lua/comments/1cl2xa3/what_is_the_best_lua_framework_for_making/
20. How to make creating mods safe for my game while allowing lua scripts to refer to other lua scripts without using package or require (Unity engine, blua addon) - Reddit, accessed December 13, 2025, https://www.reddit.com/r/lua/comments/1505yyi/how_to_make_creating_mods_safe_for_my_game_while/
21. WebAssembly tools, frameworks, and libraries for .NET Developers - Uno Platform, accessed December 13, 2025, https://platform.uno/blog/webassembly-tools-frameworks-and-libraries-for-net-developers/
22. Building a mod toolkit for Edge Of Eternity - Game Developer, accessed December 13, 2025, https://www.gamedeveloper.com/programming/building-a-mod-toolkit-for-edge-of-eternity
23. System.Runtime.Loader.AssemblyLoadContext class - .NET - Microsoft Learn, accessed December 13, 2025, https://learn.microsoft.com/en-us/dotnet/fundamentals/runtime-libraries/system-runtime-loader-assemblyloadcontext
24. AssemblyLoadContext in C#. The AssemblyLoadContext class in .NET... | by Vikas Jindal | Medium, accessed December 13, 2025, https://medium.com/@vikkasjindal/assemblyloadcontext-in-c-bbaacd692989
25. About AssemblyLoadContext - .NET - Microsoft Learn, accessed December 13, 2025, https://learn.microsoft.com/en-us/dotnet/core/dependency-loading/understanding-assemblyloadcontext
26. How to dynamically load and unload (reload) a .dll assembly - Stack Overflow, accessed December 13, 2025,

https://stackoverflow.com/questions/63616618/how-to-dynamically-load-and-unload-reload-a-dll-assembly
27. Tutorial: Write your first analyzer and code fix - C# - Microsoft Learn, accessed December 13, 2025, https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/tutorials/how-to-write-csharp-analyzer-code-fix
28. How I Built a Roslyn Analyzer to Save Developers from Azure Durable Functions Bugs | by Viktor Ponamarev | Oct, 2025 | Medium, accessed December 13, 2025, https://medium.com/@vikpoca/how-i-built-a-roslyn-analyzer-to-save-developers-from-azure-durable-functions-bugs-ab2da8a9cc49
29. Restricted IVT analyzer: exposed internals should be accessible · Issue #2648 - GitHub, accessed December 13, 2025, https://github.com/dotnet/roslyn-analyzers/issues/2648
30. WebAssembly Language Support Matrix - Developer, accessed December 13, 2025, https://developer.fermyon.com/wasm-languages/webassembly-language-support
31. Can We Achieve Secure and Measurable Software Using Wasm? - Fermyon, accessed December 13, 2025, https://www.fermyon.com/blog/can-we-achieve-secure-and-measurable-software-in-the-real-world
32. Using WASM as a mod system for a game built in rust · wasmerio wasmer · Discussion #2627 - GitHub, accessed December 13, 2025, https://github.com/wasmerio/wasmer/discussions/2627
33. Data lifecycle - Auxiliary Docs | Factorio, accessed December 13, 2025, https://lua-api.factorio.com/latest/auxiliary/data-lifecycle.html
34. [Modding] Is it not possible to update an entity's fields during control.lua runtime? : r/factorio, accessed December 13, 2025, https://www.reddit.com/r/factorio/comments/cw3amp/modding_is_it_not_possible_to_update_an_entitys/
35. [Meta] Unify naming between data and control stage. - Factorio Forums, accessed December 13, 2025, https://forums.factorio.com/viewtopic.php?t=60662
36. How do you handle saves? : r/gamedev - Reddit, accessed December 13, 2025, https://www.reddit.com/r/gamedev/comments/x5srzu/how_do_you_handle_saves/
37. "Save contains missing mods" but all mods are there - Discussion - Nexus Mods Forums, accessed December 13, 2025, https://forums.nexusmods.com/topic/13302726-save-contains-missing-mods-but-all-mods-are-there/
38. Data linking between stages/layers - Factorio Forums, accessed December 13, 2025, https://forums.factorio.com/viewtopic.php?t=97618
39. How to avoid breaking player's saved games while continuing development and changing format of saved games? : r/gamedev - Reddit, accessed December 13, 2025, https://www.reddit.com/r/gamedev/comments/fftpsh/how_to_avoid_breaking_pla

yers_saved_games_while/
40. Lurler/TrimKit.VirtualFileSystem: Virtual File System (VFS) is a lightweight C# framework that allows merging folders and zip-archives into one unified virtual hierarchy, enabling seamless modding, DLC, and content overrides for games. - GitHub, accessed December 13, 2025, https://github.com/Lurler/TrimKit.VirtualFileSystem
41. Overriding Custom Assets - Momentum Mod Documentation, accessed December 13, 2025, https://docs.momentum-mod.org/guide/override_custom_assets/
42. I just released a VFS system for adding easy Modding/DLC/Addon support into your games!, accessed December 13, 2025, https://www.reddit.com/r/gamedev/comments/10qfapg/i_just_released_a_vfs_system_for_adding_easy/
43. Modding Guide for Developers - Steam Community, accessed December 13, 2025, https://steamcommunity.com/sharedfiles/filedetails/?id=3212055987
44. VS Code Architecture Overview - Skywork.ai, accessed December 13, 2025, https://skywork.ai/skypage/en/VS-Code-Architecture-Overview/1977611814760935424
45. Our Approach to Extensibility - vscode-docs, accessed December 13, 2025, https://vscode-docs.readthedocs.io/en/stable/extensions/our-approach/
46. How can I create a secure Lua sandbox? - Stack Overflow, accessed December 13, 2025, https://stackoverflow.com/questions/1224708/how-can-i-create-a-secure-lua-sandbox
47. How to Build an API: A Step-by-Step Guide - Postman Blog, accessed December 13, 2025, https://blog.postman.com/how-to-build-an-api/
48. How to build an SDK from scratch: Tutorial & best practices - liblab, accessed December 13, 2025, https://liblab.com/blog/how-to-build-an-sdk
49. Generating docs: Enhancing DocFx and migrating from Sandcastle (SHFB) - Reddit, accessed December 13, 2025, https://www.reddit.com/r/csharp/comments/1ov0nlx/generating_docs_enhancing_docfx_and_migrating/
50. .NET documentation generator alternatives to Sandcastle? - Software Recommendations Stack Exchange, accessed December 13, 2025, https://softwarerecs.stackexchange.com/questions/39227/net-documentation-generator-alternatives-to-sandcastle
51. DocProject vs Sandcastle Help File Builder GUI - Stack Overflow, accessed December 13, 2025, https://stackoverflow.com/questions/319632/docproject-vs-sandcastle-help-file-builder-gui
52. Developing SDK Mods - Borderlands Python SDK, accessed December 13, 2025, https://bl-sdk.github.io/developing/
53. Write and debug code by using Hot Reload - Visual Studio (Windows) - Microsoft Learn, accessed December 13, 2025, https://learn.microsoft.com/en-us/visualstudio/debugger/hot-reload?view=visuals

tudio
54. C# Scripting Engine Part 7 – Hot Reloading - Kah Wei, Tng, accessed December 13, 2025, https://kahwei.dev/2023/08/07/c-scripting-engine-part-7-hot-reloading/
55. Debugging and Profiling Factorio Mods with VSCode - YouTube, accessed December 13, 2025, https://www.youtube.com/watch?v=oNfMNFxy2X4
56. Setting up a Minecraft mod environment in VSCode (It's easier than you think!), accessed December 13, 2025, https://dev.to/drazisil/setting-up-a-minecraft-mod-enviroment-in-vscode-it-s-easier-than-you-think-5bfc