Unity Architecture for Growing Projects

https://gerolds.github.io/textbook/textbook/posts/unity-architecture/

Contents

1. Premise
    1.1 The core idea: hosts as membranes
    1.2 This is not OOP architecture
    1.3 What this means for solo developers
    1.4 How dynamics change as teams grow
2. The target architecture
    2.1 Why legibility matters: the city metaphor
    2.2 What changes from prototype to production
    2.3 Host-scenes: game modes as isolated worlds
3. Unity’s instantiation reality
4. Why init-time resolution matters
5. Dependencies should always be ready
6. Explicit lifecycle beats implicit scheduling
7. Commands, queries, and events
8. The physical world as a shared channel
    8.1 Modules encode their own discovery
    8.2 Multiple modules, same physics world
    8.3 When modules coordinate spatially
    8.4 MonoBehaviours as spatial markup
9. Data ownership
10. Premature abstraction
11. What happens inside a module
12. The singleton trap
    12.1 Shells and control transfer
    12.2 Assume multiplicity
    12.3 Spawners for dynamic content
    12.4 No static references for discovery
    12.5 Actions need context
    12.6 When multiplicity matters

12.7 Design, not infrastructure

13. Cross-cutting concerns
14. Third-party code and static APIs
15. Persistence from day one
16. ScriptableObjects as symbols
17. Assembly definitions
18. Module granularity
19. DI containers
20. Runtime debugging
21. Closing
22. Quick reference
23. Implementation: the starter kit

Unity Architecture for Growing Projects

1. Premise

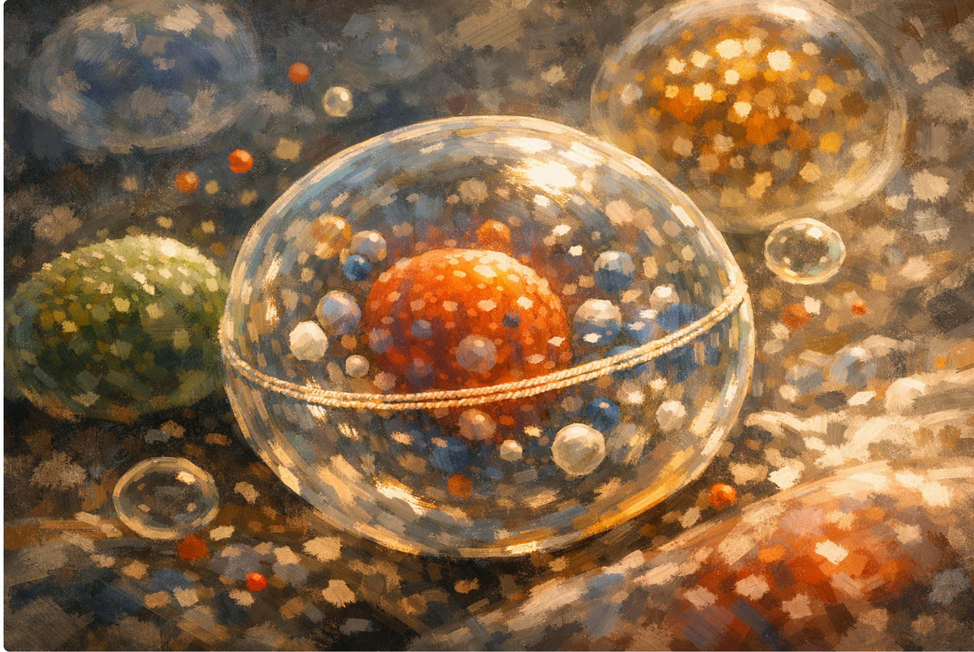
Unity's core primitives (scenes, prefabs, serialized references, `GetComponent`, lifecycle callbacks) optimize for speed of authoring. They make it cheap to get something running. They also create implicit coupling: any object can find any other object, state scatters across the hierarchy, and dependencies live in serialized fields that only the editor knows about. The longer a project runs, the more these connections accumulate. At some point the codebase works but its structure is no longer legible to the people working in it.

This is a structural tendency, not a defect. The same design that makes Unity productive for prototyping makes it quiet about the coupling it introduces. Iteration slows, onboarding takes longer, certain systems become fragile or opaque, not from any single mistake but from undeclared relationships piling up.

This article describes an architecture that works *with* Unity's grain while making ownership, boundaries, and dependencies explicit. The core structure (modules, hosts, contracts, orchestration) stays

the same from prototype to production; what changes is how strictly you enforce it. The goal is a codebase where you can reason about one module at a time, trace dependencies without archaeology, and scale rigor to match coordination cost.

1.1. **The core idea: hosts as membranes**

> 
>
> *Picture a living cell. The cell membrane isn't a wall, it is a selective boundary. It controls what enters (nutrients, signals), what exits (waste, messages), and what belongs inside (the cell's machinery). The membrane doesn't block communication; it mediates it. Without the membrane, the cell's contents would dissolve into the environment. With it, the cell maintains identity and can cooperate with other cells without losing itself.*
>
> *The cell is a module, a living unit that the membrane defines and protects. When molecules pass through the membrane, they enter the cell's domain. The cell receives them, integrates them into its machinery, and puts them to work. The cell knows what's inside, maintains its own state, and is the authority for its own function. Other cells don't reach in and manipulate its internals, they send signals through the membrane.*

In this architecture, the **host is the module's membrane**. It's the boundary you pass through to belong, the thing that equips

you when you enter, and the surface the module communicates through. The host answers “who owns this?”, “how do I join?”, and “how do I talk to this module?” If you can point to the host, you can understand the module’s shape and usually predict where its bugs live.

- **Everything participates by belonging to a host.** Scene objects, spawned prefabs, loaded content don’t just exist; they belong to a module by registering with its host. “Registration” is conceptual: for a component, it might be a formal call; for a pooled object, it might just mean “this pool belongs to Combat.”
- **Registration is the handshake.** When something joins a module, the host gives it context (dependencies, configuration). The object doesn’t fish for these later.
- **Communication crosses the membrane through contracts.** Modules don’t reach into each other’s internals. They talk through the host’s public surface.
- **The host owns what’s inside.** State, lifecycle, persistence; the host is the authority. If you need to save, enumerate, or debug, you ask the host.

This concept generates everything else in this article. Read it as orientation, not prescription; the specific mechanisms (DI containers, manual wiring, singletons-with-discipline) vary by project. The principle stays the same.

The goal is a codebase where new team members can learn one module at a time, persistence is straightforward because state has clear owners, and debugging starts with “what did this module see?” rather than “what is everything connected to?”

1.2. This is not OOP architecture

This architecture uses objects because C# and Unity use objects, but the pattern itself is not object-oriented in the design-patterns sense. Game architecture operates under constraints that enterprise patterns don’t account for:

- **Dynamic populations**: objects spawn and despawn constantly
- **Large-scale state transitions**: entire levels load/unload in real-time

- **Soft real-time**: 16ms per frame, every frame
- **Multiple modes**: menus, gameplay, cutscenes, editors (different apps sharing a process)
- **Spatial structure**: 3D space, scene graphs, streaming, physics volumes
- **Dual data domains**: authored content vs. user-generated state
- **Editor as runtime**: inspectors, gizmos, and tooling are first-class concerns
- **Scripting and systems**: high-level game logic alongside optimized infrastructure
- **Cross-cutting coherence**: a sword swing touches animation, audio, particles, damage, UI, camera
- **Performance as architecture**: allocation patterns and update frequency are design decisions, not afterthoughts

Traditional patterns (MVC, clean architecture, repository) assume stable object graphs and cheap milliseconds. Games don't have those. What works here embraces transience, space, real-time constraints, and the editor.

These principles (ownership, boundaries, communication surfaces) apply regardless of paradigm. In MonoBehaviour-heavy code, hosts coordinate other MonoBehaviours. In ECS/DOTS, systems *are* the hosts: they own processing for a slice of logic, and the membrane is the system's public queries and events. In Unreal, Subsystems fill the role. In Godot, Autoloads. The mechanisms differ but the mental model travels.

Different engines ship different "rails" for them. Unity gives you freedom, and you build the rails yourself. Treat this article as orientation, not a framework to copy wholesale.

1.3. What this means for solo developers

If you're working alone on a short project, most of this is optional. You don't need assembly separation, formal contracts, or explicit orchestration. You can use singletons freely. The cost of unwinding assumptions is bounded by the project's lifespan and your own memory.

What helps even solo:

- **Thinking in modules.** Folders that correspond to responsibilities make it easier to find code six months later.
- **Init-only dependency capture.** Storing references during `Awake` and never fishing for them at runtime prevents mysterious ordering bugs.
- **Plain data for state.** If you ever want to save, replay, or debug, having state in plain objects pays off immediately.

Solo projects can travel light. But as a project grows, you may find you can't keep all the details in your head anymore. At that point the architecture helps protect you from yourself. And once certain modules mature into reusable libraries, you can prototype quickly while leaning on production-ready infrastructure for the generic bits.

1.4. How dynamics change as teams grow

Architecture pressure comes from coordination costs, not code volume.

Solo / pair: You hold the whole system in your head. Informal conventions work. Singletons are fine because you know who uses them.

Small team (3–6): You can't assume everyone knows everything. Implicit dependencies start to surprise people: "I didn't know that singleton was being used there." Code review catches some issues, but reviewing every line is expensive. Explicit module boundaries start paying off, not for the compiler but for human communication. "Inventory is Alice's. Talk to her before touching it."

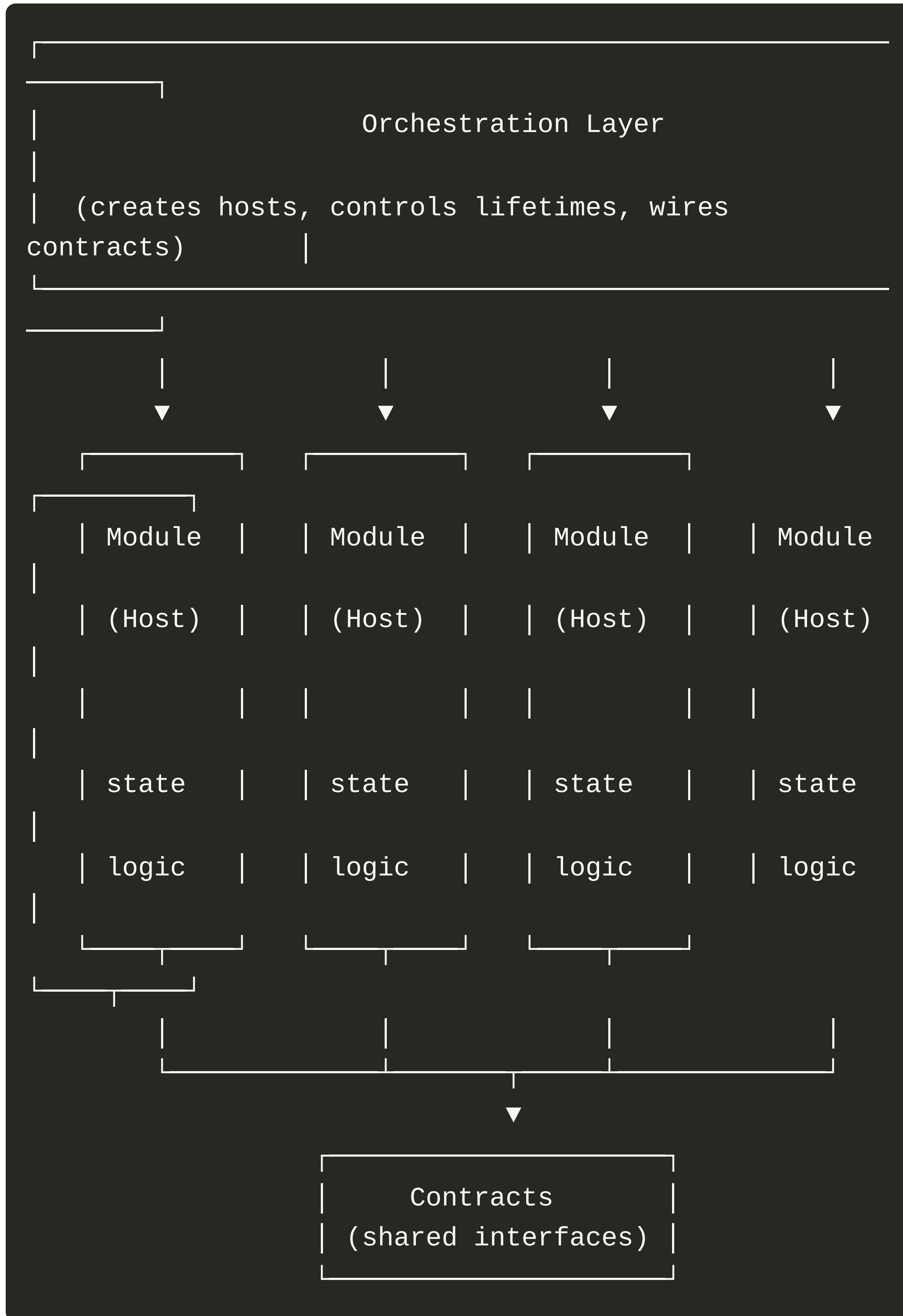
Larger team (7+): Verbal coordination doesn't scale. You need boundaries the compiler enforces: assembly definitions, explicit contracts, formal ownership. New hires learn one module at a time. The architecture becomes documentation.

Multi-team / long-lived project: Modules become team boundaries. Contracts become APIs that teams negotiate. The orchestration layer is where separately-developed modules meet.

2. The target architecture

The stable shape looks like this:

1. **Modules**: coherent slices of functionality (inventory, combat, dialogue, save/load). Each does one thing you can name in a sentence.
2. **Hosts**: the entry point and state owner for each module. A host is a long-lived object (often a MonoBehaviour, sometimes a plain C# class) that coordinates the module's internals and exposes its public surface.
3. **Contracts**: the narrow public interface of each module. Other code talks to contracts, not to internals. Contracts are typically interfaces or small data types in a shared assembly.
4. **Orchestration layer**: the composition root that creates or loads hosts, controls their lifetimes, defines startup order, and wires contracts to implementations. This is the one place that knows about everything.



Modules communicate through contracts. The orchestration layer is the only code that references all modules. This keeps coupling explicit and one-directional.

2.5. **Why legibility matters: the city metaphor**

Picture a large Unity project as a city. The problem in most Unity cities is that any building can build a private road to any other

building. At first it feels efficient. Later you discover you cannot reason about traffic, you cannot reroute, and you cannot tell which roads are essential and which are accidental.

Architecture is deciding where the districts are, what roads may cross district lines, and where the entry points sit. You're not eliminating roads; you're making the road network legible. Each district should be coherent: most of what it does should fit in one sentence.

2.6. What changes from prototype to production

The principles stay the same. What changes is how explicit you make them:

Stage	Modules	Hosts	Contracts	Enforcement
Prototype	Folders, conventions	MonoBehaviours	Implicit (public methods)	Social
Vertical slice	Named responsibilities	Own state as plain data	Deliberate interfaces	Code review
Production	Assembly-separated	Formal lifecycle	Shared assembly	Compiler

A prototype host might just be "the MonoBehaviour that holds this feature's state." A production host might have formal lifecycle methods, assembly isolation, and tooling. The *principle* is identical (this is the authority for this slice); the *rigor* scales with coordination cost.

2.7. Host-scenes: game modes as isolated worlds

In a production project, the top-level organization is **host-scenes**: Unity scenes that represent completely independent game modes. Each host-scene is a self-contained world with its own lifecycle, hosts, and orchestration. Examples:

- **App-startup**: splash screens, platform initialization, profile selection.
- **Main menu**: title screen, settings, save-slot selection.

- **Single-player**: the core gameplay loop for a solo campaign.
- **Multiplayer**: networked gameplay with its own connection lifecycle.
- **Free-roam / sandbox**: exploration mode with different rules.
- **Credits / cinematics**: linear sequences with minimal interactivity.

These modes are **conceptually decoupled**: they share no runtime lifecycle. Transitioning from menu to single-player unloads one world and loads another. State that needs to survive (player profile, selected save slot, network session) travels via a small persistent layer or initialization parameters, not shared singletons that span modes.

What lives in a host-scene:

1. **All host instances for that mode.** Inventory, combat, dialogue, UI (whatever the mode needs). Hosts load with the scene and destroy with the scene.
2. **A bootstrapper.** A single MonoBehaviour that initializes the mode. It receives minimal input (maybe a save-slot ID or a “new game” flag), then clears and populates the service locator, initializes hosts in dependency order, and triggers the initial state transition.
3. **A play-mode state machine.** An authoritative FSM for mode-wide concerns: input routing, cursor state, UI layers, time scale, pause, network readiness. The single place that knows “what phase of gameplay are we in?” Hosts query or subscribe; they don’t independently manage global state.
4. **Persistence and scenario initialization.** The mode knows how to hydrate from a save file or initialize fresh. Part of the bootstrapper’s job, not an afterthought.

What does NOT live in a host-scene:

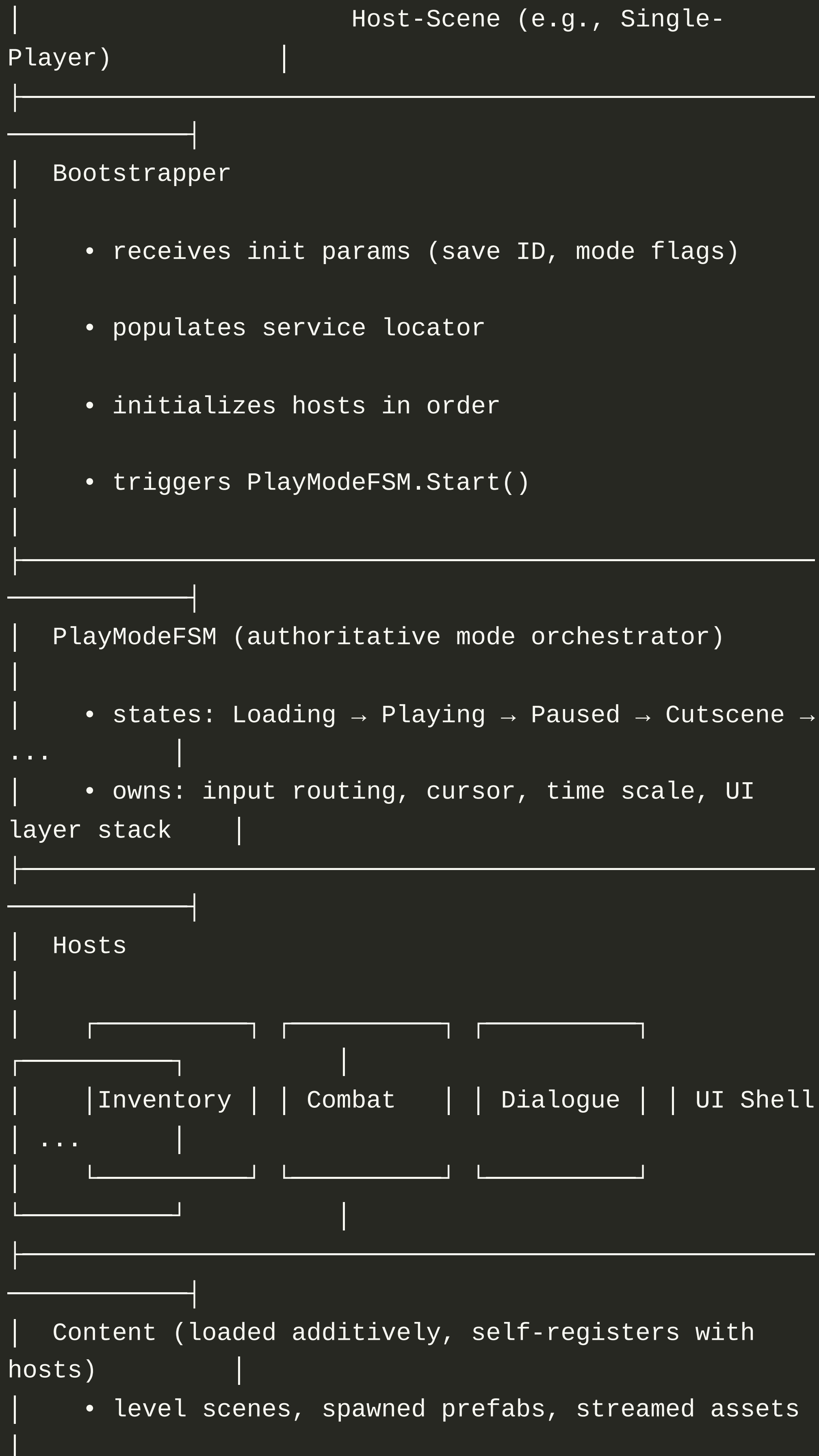
The host-scene contains only **infrastructure**: the hosts and orchestration that manage gameplay. No gameplay content.

Levels, enemies, items, NPCs, interactables are all **loaded additively** into the host-scene’s scope. Level scenes load on top. Prefabs spawn at runtime. Addressables stream as needed.

Self-registration does the heavy lifting. When a content scene loads or a prefab spawns, its components discover active hosts (via the service locator) and register. The host-scene doesn't need to know what content will arrive; it just needs to be ready to receive it.

Workflow benefits:

- **Content teams work independently.** Level designers build levels in their own scenes. Character artists set up prefabs. None of them edit the host-scene.
- **Clean integration.** A new enemy, weapon, or interactable is just a prefab that self-registers with the appropriate hosts. If it implements the right components, it works.
- **Fast iteration.** Test a single level by loading the host-scene plus that level. No menu boot, no full world load.
- **Predictable memory.** Content loads and unloads while hosts persist. Stream levels without tearing down infrastructure.



```
┌──────────────────────────────────────────────────────────────────┐
│                      Host-Scene (e.g., Single-Player)            │
├──────────────────────────────────────────────────────────────────┤
│  Bootstrapper                                                    │
│    • receives init params (save ID, mode flags)                  │
│    • populates service locator                                   │
│    • initializes hosts in order                                  │
│    • triggers PlayModeFSM.Start()                                │
├──────────────────────────────────────────────────────────────────┤
│  PlayModeFSM (authoritative mode orchestrator)                   │
│    • states: Loading → Playing → Paused → Cutscene → ...        │
│    • owns: input routing, cursor, time scale, UI layer stack     │
├──────────────────────────────────────────────────────────────────┤
│  Hosts                                                           │
│    ┌──────────┐ ┌──────────┐ ┌──────────┐ ┌──────────┐          │
│    │Inventory │ │ Combat   │ │ Dialogue │ │ UI Shell │ ...      │
│    └──────────┘ └──────────┘ └──────────┘ └──────────┘          │
├──────────────────────────────────────────────────────────────────┤
│  Content (loaded additively, self-registers with hosts)          │
│    • level scenes, spawned prefabs, streamed assets              │
```

```
│     • not part of the host-scene asset itself
│
└──────────────────────────────────────────────────────────────
─────────────┘
```

Why this matters:

- **Clean unload.** Mode ends, scene destroys. All hosts, registered components, and scoped state are gone. No lingering singletons.
- **Parallel development.** Teams work on different modes without conflicts.
- **Testability.** Load a host-scene in isolation with test parameters.
- **Mode-specific optimization.** Each mode loads only the hosts it needs.

Cross-mode communication has to be explicit. You can't call "the inventory" from the menu unless you design the handoff. That's the point: it forces you to decide what state survives transitions.

---

3. Unity's instantiation reality

Unity instantiates from content: scenes, prefabs, additive loading, Addressables. Insisting that a central orchestrator explicitly construct every gameplay object means writing spawners for everything and fighting the engine's iteration speed.

The compromise that scales:

- The orchestration layer creates or loads **hosts** and establishes **scopes** to which they bind.
- Unity-instantiated components **self-register** with the appropriate host during initialization.
- After registration, the host injects them with dependencies, and the components never look up dependencies on their own again.

This keeps workflows editor-friendly while making ownership and boundaries explicit.

4. **Why init-time resolution matters**

The membrane model implies a constraint: **resolve dependencies only during initialization, store them, and never resolve again.**

If components could resolve dependencies at any time, they'd bypass the host and the membrane would become porous. Runtime dependency fishing reintroduces hidden coupling, just behind a nicer API.

> Scene objects find their host and resolve contracts **only during initialization**. After that, they use stored references.

This blocks invisible dependencies that appear mid-gameplay, untraceable in code review, brittle to ordering changes.

5. **Dependencies should always be ready**

What if a dependency isn't ready yet? It should be.

The orchestration layer ensures that by the time anything initializes, the hosts it depends on already exist. Predictable memory, predictable timing, fewer frame spikes.

If something needs to load asynchronously, orchestration awaits it *before* allowing dependents to initialize. Avoid architectures where "asking for a thing" silently constructs it. Requests cascade, lifecycle reasoning gets murky, and dependencies turn brittle.

6. **Explicit lifecycle beats implicit scheduling**

Many Unity projects get into trouble not because Unity has callbacks, but because callbacks become an implicit scheduler. Nobody knows what runs when, and everyone is afraid to change the order.

Hosts make lifecycle explicit:

- `Awake` / `OnEnable` is for finding the host and registering.
- Registration triggers dependency injection.
- A deliberate initialization phase transitions modules to "running".

- Update order is owned by a single place, not emergent behavior.

When order is explicit, sequencing bugs stop being mysterious.

7. **Commands, queries, and events**

Global event buses let you communicate without designing contracts. They also create hidden dependencies and brittle order constraints.

Keep these separated:

- **Command:** do something (may fail, may have ordering).
- **Query:** ask for data (no side effects).
- **Event:** announce something happened (observers optional).

Let modules accept commands/queries through their contracts. Expose events deliberately and typed, as part of the contract, not as a global publish-anything bus.

A well-designed event:

```
public interface ICombatEvents
{
    event Action<DamageEvent> OnDamageDealt;
    event Action<Entity> OnEntityDied;
}
```

Subscribers are explicit: a module that cares about combat outcomes takes a dependency on `ICombatEvents` during initialization and subscribes. No anonymous broadcast. The dependency is visible.

8. **The physical world as a shared channel**

Beyond host contracts, there's another communication channel: the physical world itself. Think of it as a **shared telephone line** any module can use, but with a specific protocol.

Unity's physics engine (triggers, raycasts, overlap queries, collision callbacks) is a built-in spatial index. It answers "what is near this point?" or "what entered this volume?" efficiently, with designer-

friendly tooling. Most games should use it. So how does spatial discovery fit into modular architecture?

8.8. Modules encode their own discovery

Each module that wants to be discoverable through physics **defines its own marker components**. The queries that look for them also live inside the module. The module owns both sides of the conversation.

Example: the **Combat module** wants projectiles to find damageable targets.

1. Combat defines `DamageReceiver`: a MonoBehaviour that marks “I can take damage” and holds a reference to the entity’s health data.
2. Combat’s projectile logic performs a raycast or overlap query.
3. The query looks for `DamageReceiver` components (via `GetComponent` or layer filtering).
4. When found, the projectile calls methods on `DamageReceiver`, a component Combat owns.
5. `DamageReceiver` updates the entity’s health through Combat’s internal systems.

The querying code and the discovered component belong to the same module. Combat queries for Combat’s own marker, not some global `IDamageable` interface. The module encodes its discovery protocol and operates within its own boundaries.

8.9. Multiple modules, same physics world

Different modules attach different marker components to the same GameObject. A character might have:

- `DamageReceiver` (Combat) for taking damage
- `InteractionTarget` (Interaction) for player prompts
- `AIPerceptionTarget` (AI) for enemy detection

Each module queries for its own markers. Combat raycasts find `DamageReceiver`. Interaction finds `InteractionTarget`. AI finds

`AIPerceptionTarget`. Same physics world, separate component vocabularies.

Everyone uses the same wire (Unity's physics), but each module speaks its own language.

8.10. When modules coordinate spatially

Sometimes spatial events cross module boundaries. A projectile (Combat) hits something that triggers dialogue. The spatial discovery still happens within Combat; it finds its `DamageReceiver`. Cross-module communication happens through contracts: if the target dies, Combat raises an event the dialogue system observes.

Physics queries don't directly invoke another module's code. They invoke the querying module's own component, which may then communicate through contracts.

8.11. MonoBehaviours as spatial markup

Even with most logic in hosts, MonoBehaviours remain essential for spatial games because they attach data to positions. For physics-based discovery:

- **Trigger volumes** with `OnTriggerEnter/Exit` callbacks
- **Colliders** queryable via raycast or overlap
- **Marker components** identifying what a GameObject means to a module

These MonoBehaviours are thin: data, callbacks, delegation to the host. "Spatial markup" is the module's way of saying "this point in space participates in my domain."

Physics isn't a back door around the architecture; it's a discovery mechanism each module uses on its own terms.

9. Data ownership

When two parts of the game care about the same data, pick an owner.

If both combat and UI need health, one owns the canonical value. Combat issues commands (apply damage). UI observes changes. Shared mutable state that anyone can access is how coupling sneaks back in.

If two modules both need to write the same field, your boundaries are wrong, or you need a third module to own it.

Entities are not modules. A character might have `DamageReceiver` (Combat), `InventoryHolder` (Inventory), `CharacterMotor` (Movement). Each component belongs to its module; the entity is an assemblage, not an owner. “Which module owns the player?” is the wrong question. Modules own *aspects* of entities.

10. **Premature abstraction**

Abstractions have a cost. Every interface, every indirection, every “what if we need to swap this later” adds cognitive load. Contracts and interfaces should be *earned*, not speculative.

Start concrete, abstract when you understand the compression.

A good abstraction compresses meaning: it finds the narrow API that captures what several concrete cases share while hiding what differs. You can’t find that compression until you’ve seen the cases.

Abstracting too early creates interfaces that are either too narrow (and you bypass them) or too wide (and they constrain nothing).

- **First implementation:** concrete code. `InventoryHost` has methods that do exactly what inventory needs. No interface.
- **Second use case:** if another module needs to talk to inventory, extract an interface with only the methods actually used. The interface is discovered, not designed upfront.
- **Reusable systems:** when you see the same pattern three times, extract a generic system. Now abstraction earns its keep.

Contracts between modules aren't premature abstraction; they're boundaries. `IInventoryQueries` exists to define what inventory exposes, not to enable hypothetical swapping.

Within a module, resist the urge to abstract early. Internals can be messy, concrete, pragmatic. The architecture protects the rest of the codebase from that mess.

11. **What happens inside a module**

Inside a module, do whatever solves the problem. Inheritance, composition, ECS-style data layouts, state machines, coroutines, Jobs, Burst.

A module is a protected space. Respect the contract at the boundary and you're free to experiment or rewrite without coordinating with the rest of the team. Combat might use tight data layouts; dialogue might be a state machine; save might be pure functions.

Contracts are the membrane. Inside, do what works.

12. **The singleton trap**

One of the most expensive early assumptions: there's one player, one UI, one camera, one inventory. It simplifies everything until it doesn't.

Single-player games routinely break singleton assumptions:

- **Vehicles and shells.** Player enters a mech, drives a car, possesses an NPC. “The player” becomes two things: identity and current shell. If PlayerController is a singleton, control transfer gets messy.
- **Mode changes.** JRPG switches between traversal and combat. Each mode has different input, UI, camera. Singletons that assume they’re always active become a tangle of enable/disable flags.
- **Minigames.** Player enters a fishing minigame. Different systems handle input and UI. Main systems assuming they’re always active means more time suppressing them than building the minigame.
- **Companions and AI.** NPCs using player-like systems. If those are hardcoded to “the player,” you duplicate code.
- **Multiple views.** Split-screen, minimaps, picture-in-picture. Singleton Camera or UI makes each additional view a special case.

What seemed unique is an instance of a category.

12.12. Shells and control transfer

Avatar ≠ player. A player is an identity (save slot, profile, input source) that controls different avatars over time. The avatar, or “shell,” is the current vehicle: character, mech, interface.

Systems that conflate player and avatar break on control transfer. Health belongs to the avatar. Input routes from player identity to current shell. When the player enters a vehicle, input reroutes; the character’s controller goes quiet.

Matters in single-player: piloting mechs, possessing enemies, switching party members, in-game terminals.

12.13. Assume multiplicity

Structure systems to operate on *a* thing, not *the* thing. Health isn’t `PlayerHealth.Instance.Current`; it’s a component on an entity that might be player-controlled. Inventory isn’t a singleton; it’s owned by something queryable. Input doesn’t go to “the player

controller" directly; it goes to whichever entity is currently receiving from that source.

Runtime might have only one instance. The code shouldn't assume that. Pass references; don't call singletons.

12.14. **Spawners for dynamic content**

Runtime-created objects (enemies, projectiles, items, effects) need a spawner that owns their lifecycle: creating, tracking, and destroying them. The spawner answers "how many active enemies?" or "all projectiles in this area."

Without spawners, dynamic content is untracked. Try to "pause all enemies" or "serialize active projectiles" and you discover nothing knows what's alive.

An escalation, not foundational. Early on, Unity's built-in instantiation is fine; prefabs spawn, self-register, participate. Formalize spawners when you need enumeration, persistence, or pooling.

For high-frequency content (thousands of projectiles), the *pool* belongs to the host, not each instance.

12.15. **No static references for discovery**

Outside host-to-host communication, nothing should rely on static references. A component needing health receives it through injection, not `Player.Instance`.

This forces context-aware APIs. A UI panel receives a reference to *an* inventory when opened, not "the inventory." A damage system operates on whatever entity was hit. Reusable because it assumes nothing about the global shape.

12.16. **Actions need context**

Action-performing systems should assume they need:

- **Actor**: who is performing
- **Target**: who is affected

- **View**: how is this presented, to whom

“Deal 10 damage” → “actor A deals 10 damage to target B, with view context for feedback.” “Show health” → “show this entity’s health in this panel for this observer.”

Works well at scale. Player enters vehicle: actor changes, system doesn’t care. Companion health bar: view routes to a different panel. AI uses the same combat system; the actor slot accepts AI controllers.

12.17. When multiplicity matters

Not all code needs multiplicity. The question is whether the code will be reused.

Project-specific scripting (glue code, one-off behaviors) can assume singletons. Disposable code. Assumptions break, you rewrite.

Reusable systems (stat systems, damage pipelines, ability frameworks) should support multiplicity from the start. If they assume singularity, every project inherits the assumption.

One-off prototype? Use singletons. Expect to reuse? Design for multiplicity. Unsure? Notice when the singleton assumption starts hurting; that’s the signal.

12.18. Design, not infrastructure

None of this requires building vehicle systems or multiplayer up front. It requires **not cementing assumptions**. Ship a single-player game with one player, one inventory, one camera. But if systems accept context rather than assuming globals, adding a vehicle later is “create a shell and reroute input,” not “rewrite everything that touches player state.”

Cost of multiplicity: passing a reference instead of calling a singleton. Cost of singularity: weeks of rewriting when the assumption breaks.

13. **Cross-cutting concerns**

Logging, input, time, and analytics are services many modules use but that don't belong inside any domain module. Treat them as **infrastructure contracts**: interfaces that the orchestration layer provides at startup.

Infrastructure contracts differ from domain contracts (`IInventory`, `ICombat`). Domain contracts come from hosts. Infrastructure contracts come from the orchestration layer itself and exist before any domain host initializes.

- **Input** translates raw Unity input into game-meaningful actions
- **Time** provides pausable, scalable game time
- **Logging** and **Analytics** accept calls without modules knowing how data is stored

These live in a shared assembly. Cross-cutting concerns become explicit dependencies, not invisible singletons.

The payoff is testability. You can record a replay by swapping input for playback. You can fast-forward time in a test by swapping the time contract. You can ship to a platform without analytics by swapping for a no-op. None of this touches the modules consuming the contracts.

14. **Third-party code and static APIs**

Unity's own APIs are static: `Time.deltaTime`, `Input.GetKey`, `Physics.Raycast`. Third-party assets often use singletons. This isn't something to fight; it's just reality.

Static APIs give global access to things that are genuinely global: the physics engine, the render pipeline, platform services. Don't reject them. Concentrate them.

Wrap external singletons in modules that adapt them to your contracts. An input module wraps Unity's input system and exposes a contract. A time module wraps `Time.deltaTime` and provides pausable game time. An audio module wraps FMOD or Wwise and exposes your game's audio vocabulary.

```
// Without wrapping: scattered static calls
float delta = Time.deltaTime;
FMODUnity.RuntimeManager.PlayOneShot("event:/UI/Click"
);

// With wrapping: game code talks to contracts
float delta = _time.DeltaTime;  // IGameTime, pausable
_audio.Play(Sounds.UIClick);    // IAudio, your
vocabulary
```

Two purposes:

1. **Isolation.** Change input systems, change one module. Everything else talks to the contract.
2. **Translation.** The wrapper translates engine-level concepts into game-level concepts. Raw input becomes “jump pressed.” Raw time becomes “game time, paused during menus.”

Static calls happen inside modules that own the relationship. Outside, code talks to contracts.

15. Persistence from day one

Almost every game saves state. Treat persistence as an afterthought and you’ll find state scattered across MonoBehaviours, buried in scene hierarchies, tangled with runtime-only data. Retrofitting hurts.

The modular architecture makes persistence straightforward. Hosts own persistent state in plain data objects (simple fields, no Unity references, no circular graphs). A save module asks each host for serializable state, bundles it, writes it. Loading reverses the flow.

Design for persistence early. It costs almost nothing if hosts already own plain data. It costs months if you add it after state has leaked everywhere.

16. ScriptableObjects as symbols

Modules multiply; you need shared identity without shared code: stat names, damage types, item categories.

ScriptableObjects work well as **symbols**, assets that represent identity without behavior. A damage type, a stat name, an item category, no logic attached. Modules define symbol vocabularies. A host holds a serialized reference declaring “I fill this role.” Other modules share the symbol without communicating directly; they agree on identity through asset references.

Why not strings or enums? Strings break silently on typos. Enums force recompilation when values change. ScriptableObject symbols: rename without breaking references, serialize, inspect, query at

runtime. Designer adds damage type? Create an asset. Programmer checks if damage is fire? Compare references.

17. **Assembly definitions**

When you want the compiler to enforce boundaries, separate each module into its own assembly. Contracts in a small shared assembly everyone references. Each module references contracts, not other modules' internals. Orchestration references everything.

Dependency graph becomes explicit. Accidental coupling becomes a compile error.

Typical layout:

- `Contracts.asmdef`: interfaces and data types for module boundaries
- `Inventory.asmdef`: references Contracts only
- `Combat.asmdef`: references Contracts only
- `Orchestration.asmdef`: references everything

Someone calls Combat internals from Inventory? Compiler says no. Conversation at compile time, not code review.

18. **Module granularity**

When is a module the right size? Boundaries come from **domain coherence** and **operational abstraction**.

Domain coherence: module corresponds to a slice of game meaning. Inventory = what the player has. Combat = damage and resolution. When the one-sentence description uses “and,” probably two responsibilities.

Operational abstraction: code stops caring what data *means*, implements a pattern for how meaning is *created*. A stat system doesn’t care that “Strength” is strength; it cares about named values that can be queried and modified. Systems-level code becomes reusable; domain-level stays specific.

19. **DI containers**

Use Zenject or VContainer? You can. They automate creation and resolution, manage scopes. With discipline, they reduce boilerplate.

The danger: easy resolution means developers stop thinking about dependency direction. A container that resolves anything from anywhere is just a service locator with extra steps. Configure for scoped lifetimes, init-only resolution, contracts over implementations.

Some teams find a hand-rolled locator sufficient (one static class holding hosts for the scene's lifetime). Others find Zenject's installers match their thinking. Choice matters less than discipline.

Unsure? Start hand-rolled. Explicit, debuggable, teaches constraints. Upgrade when you need child scopes, async installers, or factory patterns.

20. **Runtime debugging**

Explicit architecture pays off in debugging. Hosts own state and expose contracts. Build editor tools to inspect any host, log calls at boundaries, visualize module status.

Some teams build debug panels that list registered hosts with live data. That's straightforward when state lives in plain objects owned by known hosts. It's nearly impossible when state scatters across hundreds of MonoBehaviours.

Contract-call tracing: log every command or query a module handles. When something breaks, you have a timeline of cross-module communication.

21. **Closing**

Architecture responds to needs and is applied incrementally. Prototype: informal modules. Vertical slice: lightweight hosts. Production: assembly separation, explicit contracts.

The principles stay the same: coherent modules, clear boundaries, one owner per piece of state, explicit lifecycle. What changes is the

rigor. Add structure when it solves problems the team actually feels.

If there's one takeaway: **everything participates by registering with a host; the host injects dependencies and owns what's inside; communication crosses the boundary through contracts.** That's how hidden dependencies become visible relationships.

---

22. Quick reference

For returning readers:

The core idea:

> **The host is the module's membrane.** Everything participates by registering with a host. Registration injects dependencies. Communication crosses the membrane through contracts. The host owns what's inside.

The shape:

- **Modules** own coherent slices of functionality. One sentence per module.
- **Hosts** are the membrane; the entry point, state owner, and authority for each module.
- **Contracts** are the membrane's external surface. Other code talks to contracts, not internals.
- **Orchestration** creates hosts, controls lifetimes, wires contracts.

Registration is the handshake:

- Scene objects and spawned content **register with their host** during initialization.
- The host **injects dependencies** upon registration.
- After registration, components use stored references; **no runtime resolution**.

Communication:

- Host-to-host: through contracts (commands, queries, events).

- Spatial: through physics (triggers, raycasts, `GetComponent`). Components bridge to hosts.

Ownership:

- Every piece of state has one owner (a host).
- If two modules need to write the same field, the boundaries are wrong.

Inside vs. outside:

- Inside a module, do whatever works. The architecture governs boundaries, not internals.
- Contracts are the membrane. Respect them; internals are free.

Multiplicity:

- Design systems to operate on *a* thing, not *the* thing.
- Player ≠ avatar. Pass references; avoid singletons in reusable code.

Enforcement escalation:

- Solo: folders, conventions, social.
- Small team: named responsibilities, code review.
- Larger team: assembly definitions, compiler-enforced contracts.

Persistence:

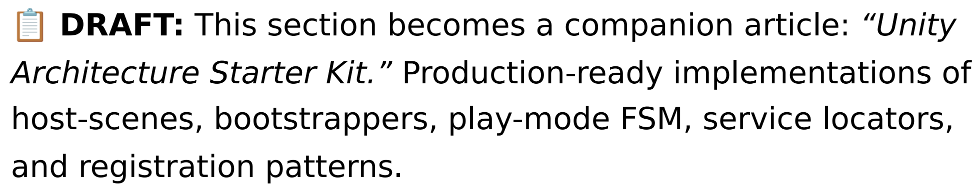
- Hosts own state in plain data objects. Saving is asking each host for serializable state.

Symbols:

- ScriptableObjects as identity tokens. No strings, no enums. Assets you can rename without breaking references.

---

23. Implementation: the starter kit



> 📋 **DRAFT:** This section becomes a companion article: *“Unity Architecture Starter Kit.”* Production-ready implementations of host-scenes, bootstrappers, play-mode FSM, service locators, and registration patterns.

To bootstrap, you need: a service locator, host base classes, a bootstrapper pattern, contract interfaces, and folder structure mapping to eventual assembly separation.

Companion article will cover:

1. **Service locator**: scoped container holding host references for a scene’s lifetime
2. **Host base classes**: consistent initialization, registration, shutdown
3. **Bootstrapper**: how the host-scene entry point initializes hosts, populates the locator, triggers the FSM
4. **Play-mode FSM**: authoritative orchestrator for input routing, time scale, UI layers
5. **Contracts**: narrow public surfaces modules expose
6. **Symbols**: ScriptableObjects as identity tokens
7. **Folder and assembly structure**: organizing for compiler-enforced boundaries

What you can do now:

- Organize into module folders (no assembly definitions needed yet)
- Create a bootstrapper MonoBehaviour that initializes hosts in order
- Store dependencies during `Awake` / `Start`, never resolve again at runtime
- Define contracts as interfaces in a shared folder
- Use ScriptableObjects instead of strings or enums for identity

---

Drafting assistance: Claude Opus. All claims mine; errors my responsibility.