### **Tier 1: Critical Architectural Refinements**

*These are the most impactful changes that address fundamental architectural issues. Tackling these first will make all subsequent refactoring easier and more effective.*

**1~~. Dismantle the Core and Global God Objects with Dependency Injection (DI)~~**

* **~~Problem:~~** ~~The Core.cs class is a classic "God Object" singleton. It creates and holds static references to every single manager and system in the game. This creates extremely tight coupling. Every system knows about every other system, making the code difficult to test, maintain, and reason about. For example, ActionExecutionSystem directly calls Core.CurrentTerminalRenderer, which it has no business knowing about directly. Global.cs has a similar, smaller-scale issue by holding direct references to CurrentGraphics and CurrentSpriteBatch.~~
* **~~Solution:~~** ~~Implement a Dependency Injection (DI) pattern.~~
  + ~~The main Core class (your Game class) should be responsible for~~ *~~instantiating~~* ~~the managers and systems.~~
  + ~~When creating a system, pass the specific dependencies it needs into its constructor. For example, new ActionExecutionSystem(componentStore, worldClockManager, entityManager) instead of the system reaching out to Core.ComponentStore.~~
  + ~~This immediately breaks the tight coupling. ActionExecutionSystem now only knows about the components and the clock, as it should. It no longer has access to the TerminalRenderer or SceneManager.~~
  + ~~To communicate between decoupled systems, introduce a simple event bus or use C# Action / event patterns. (See Tier 2, Item 5).~~
* **~~Impact:~~****~~(Highest)~~**~~. This is the single most important refactor. It will fundamentally improve the entire architecture, enabling unit testing, modularity, and parallel development.~~

**~~2. Eliminate Runtime Reflection in the Spawner~~**

* **~~Problem:~~** ~~The Spawner uses Type.GetType(string) and reflection (Activator.CreateInstance, GetProperty, SetValue) to create components from JSON definitions at runtime. While this is very flexible for rapid prototyping, it is a significant performance bottleneck. Reflection is orders of magnitude slower than direct object instantiation. Spawning many entities (e.g., bullets, particles, enemies in a wave) would severely impact performance.~~
* **~~Solution:~~** ~~Create a two-stage loading process.~~
  1. **~~Loading/Baking:~~** ~~During game startup, ArchetypeManager should still load and parse the JSON files. However, instead of storing the raw JSON, it should use that data to create "template" or "factory" objects for each archetype. These factories would know how to create the entity and its components using direct new Component() calls, not reflection.~~
  2. **~~Spawning:~~** ~~The Spawner will then ask the ArchetypeManager for an archetype's factory/template and simply Clone() or Create() a new entity from it. This is extremely fast.~~
* **~~Impact:~~****~~(High)~~**~~. Drastically improves entity creation performance, which is critical for any game where objects are created and destroyed during gameplay.~~

### **Tier 2: Significant Performance & Design Improvements**

*These items address major performance bottlenecks and design patterns that can be significantly improved.*

**3. Optimize Rendering with RenderTarget2D**

* **Problem:** MapRenderer and TerminalRenderer appear to be redrawing a large number of static elements every single frame.
  + MapRenderer generates a list of GridElements for the entire visible map grid and draws them one by one, every frame. The underlying terrain doesn't change unless the player moves to a new world tile.
  + TerminalRenderer redraws all visible lines of text, including complex color parsing and wrapping, every frame. Most of the terminal history is static.
* **Solution:** Use RenderTarget2D as a cache.
  + **For MapRenderer:** Create a RenderTarget2D for the terrain. When the player enters a new world tile (or the map view changes), draw the static terrain (water, hills, etc.) to this render target *once*. Then, in your main Draw loop, just draw this single texture. Dynamic elements like the player, path, and hover selector can be drawn on top of it.
  + **For TerminalRenderer:** Use a RenderTarget2D for the terminal history. When a new line is added, draw only that new line onto the bottom of the render target (you may need to scroll the existing content up). Then, in the main Draw loop, draw the relevant portion of this render target to the screen.
* **Impact:** **(High)**. This will massively reduce the number of draw calls per frame, significantly boosting rendering performance and lowering CPU usage.

**4. Improve ECS Query Performance**

* **Problem:** The ComponentStore uses a Dictionary<Type, Dictionary<int, IComponent>>. When a system needs to operate on entities with multiple components (e.g., PositionComponent and RenderableComponent), it has to perform multiple lookups or get all entities with one component and then check if they have the other. This can be inefficient for systems that run every frame.
* **Solution:** Implement a "View" or "Query" system.
  + When a system is initialized, it can register a query with the ECS for a specific combination of components (e.g., ecs.CreateQuery(typeof(PositionComponent), typeof(RenderableComponent))).
  + The ECS would then maintain a cached List<int> of all entity IDs that match this query. The list is updated only when components are added or removed from entities.
  + The system can then iterate over this pre-filtered, cached list, which is much faster than repeated dictionary lookups.
* **Impact:** **(Medium-High)**. Speeds up the core logic loop of your game, especially as the number of entities and systems grows.

**5. Formalize an Event-Driven Architecture**

* **Problem:** Systems communicate by directly calling methods on each other via the Core god object (e.g., Core.CurrentTerminalRenderer.AddOutputToHistory). This is a direct result of the tight coupling mentioned in Tier 1.
* **Solution:** Create a simple, centralized event bus or use C# events more formally.
  + Instead of calling a method, a system should raise an event. For example, ActionExecutionSystem would raise an ActionCompleted(action) event. CombatResolutionSystem would raise a DamageDealt(attacker, target, damage) event.
  + Other systems, like TerminalRenderer, CombatLogPanel, or a future SoundManager, would subscribe to these events and react accordingly. The TerminalRenderer would listen for ActionCompleted and format a message. The SoundManager would listen for DamageDealt and play a hit sound.
* **Impact:** **(High)**. This is the other half of solving the God Object problem. It fully decouples systems, making them independent and reusable modules.

### **Tier 3: Code Quality & Maintainability**

*These changes will make the code cleaner, easier to understand, and less prone to bugs.*

**6. Refactor GameState**

* **Problem:** GameState.cs is becoming a secondary God Object, managing player state, world state, combat state, and UI state. This violates the Single Responsibility Principle. The combat-related properties (IsInCombat, Combatants, InitiativeOrder, etc.) do not belong in the main GameState.
* **Solution:** Break GameState into more focused classes.
  + Create a CombatState class to hold all combat-specific data. The main GameState could hold an instance of CombatState which is only non-null when in combat.
  + Player-specific data like PlayerEntityId and PendingActions should ideally be accessed through a dedicated PlayerManager or directly via the player's components, rather than being top-level properties on GameState.
* **Impact:** **(Medium)**. Improves code organization and makes it clearer where to find and modify specific parts of the game's state.

**7. Refine Input Handling Logic**

* **Problem:** The input handling is spread across InputHandler, MapInputHandler, and PlayerInputSystem. The logic in PlayerInputSystem for modifying the action queue by converting a Queue<T> to a List<T> and back is highly inefficient.
* **Solution:**
  + Clarify the responsibilities: InputHandler for raw device state, MapInputHandler for translating clicks to world actions, and PlayerInputSystem for processing those actions into the queue.
  + Change the player's action queue from Queue<IAction> to List<IAction>. A List provides the flexibility needed for appending, removing, and inserting actions, which you are already trying to do. A Queue is the wrong data structure for a path that needs to be modified.
* **Impact:** **(Medium)**. Simplifies input logic and fixes a significant inefficiency in action queue management.

**8. Consolidate and Abstract UI Components**

* **Problem:** You have many UI panel classes (CombatLogPanel, ActionMenuPanel, TargetInfoPanel, etc.) and UI element classes (Button, Slider, Dialog). The panels have repeated code for drawing borders and backgrounds. The Button class has a Function string property which is used in a large switch statement in MapInputHandler - this is brittle.
* **Solution:**
  + Create a base UIPanel class that handles common functionality like drawing a frame, border, and title. The specific panels can then inherit from this.
  + Instead of a Function string, have the MapInputHandler (or a new UIManager) directly subscribe to the OnClick event of each button with a specific lambda or method. This is type-safe and less error-prone. \_buttonMap.TryGetValue("go", out Button goButton) is a good start, but wiring the event is better.
* **Impact:** **(Medium)**. Reduces code duplication in UI classes and makes the UI interaction logic more robust.

### **Tier 4: Future-Proofing & Best Practices**

*These are good practices that will set you up for future success.*

**9. Use the MonoGame Content Pipeline for Archetypes**

* **Problem:** JSON files are being loaded directly from the Content directory using File.ReadAllText. This won't work on all platforms (like consoles or mobile) and is less efficient than using pre-processed binary assets.
* **Solution:** Create a custom Content Pipeline processor for your .json archetype files. The processor would read the JSON during the build process and write it out as a compiled .xnb binary asset. Your ArchetypeManager would then load it using Content.Load<Archetype>. This integrates seamlessly with the "baking" process from Tier 1.
* **Impact:** **(Low now, High later)**. Ensures cross-platform compatibility and improves loading times.

**10. Introduce Unit Testing**

* **Problem:** There are no tests. It's impossible to verify that complex logic like the Pathfinder, WorldClockManager time calculations, or StatsComponent secondary stat calculations are correct without running the entire game.
* **Solution:** Once DI is implemented (Tier 1), systems and logic classes will be testable. Create a separate test project and start writing unit tests for your pure logic classes.
* **Impact:** **(Medium)**. Dramatically increases code quality and confidence, preventing regressions as you add new features.

### **Summary**

Your current codebase is a solid, functional prototype. The next evolutionary step is to address the architectural patterns that limit scalability and performance.

1. **Start with Dependency Injection.** Break apart the Core God Object. This is the key that unlocks most of the other improvements.
2. **Tackle Performance.** Eliminate reflection in the Spawner and use RenderTarget2Ds for your static UI and map rendering.
3. **Refine the Design.** Formalize your event system, clean up the GameState and input handlers, and improve your ECS queries.

By following this roadmap, you will transform your project from a promising prototype into a clean, performant, and highly maintainable game foundation. Excellent work so far, and I'm ready for the first step when you are