### **Hello. I am a game developer working on a project in C# using the MonoGame framework. I am about to undertake a significant refactoring of my game's architecture to implement a modern, scalable Entity-Component-System (ECS).**

### **I have a detailed, multi-step roadmap for this process. I will be providing you with one step at a time. Your task is to act as an expert C# and game architecture programmer. For each step I provide, you will perform the following:**

### **1 Analyze the Request: Carefully read the Objective and Detailed Instructions for the step.**

### **2 Process the Context: I will provide you with all the necessary existing C# files as context. You must use these files to understand the current state of the project.**

### **3 Generate New Files: Create the complete, fully-functional C# code for any new files listed under "Files to Create." The code must be clean, well-commented, and adhere to standard C# conventions.**

### **4 Refactor Existing Files: Modify the provided context files exactly as described in the instructions. This may involve adding, changing, or removing methods, properties, and fields.**

### **5 Identify Deletions: If the instructions mention deleting a file, you will not provide code for it but will explicitly state, "The file [FileName.cs] should now be deleted from the project."**

### **6 Provide a Complete Response: Your final output for each step should be a single, coherent response containing all the new and modified code, clearly demarcated by file name. Do not ask me for clarification unless there is a direct and unresolvable contradiction in the instructions. Assume the instructions are the source of truth.**

### **Our goal is to collaboratively refactor my game. I will verify the output of each step before proceeding to the next. Let's begin.**

### **We are on step []**

### **Here is your step:**

### **Step 1: Establish the ECS Core Primitives**

**Objective:**Create the foundational, non-game-specific classes for our Entity-Component-System (ECS) architecture. This involves creating an entity manager, a base component interface, and a component storage system.

**Detailed Instructions:**

1. **Create IComponent.cs:**
   * This will be a simple marker interface. It will have no methods or properties. Its purpose is to enforce a type constraint on our components.
   * public interface IComponent { }
2. **Create EntityManager.cs:**
   * This class will be responsible for creating and destroying unique entity IDs. To be efficient, it should recycle destroyed IDs.
   * It will be a singleton, accessible via Instance.
   * **Fields:**
     + private Queue<int> \_availableIds = new Queue<int>();
     + private int \_nextId = 0;
   * **Methods:**
     + public int CreateEntity(): If the queue is not empty, dequeue and return an ID. Otherwise, return \_nextId++.
     + public void DestroyEntity(int entityId): Enqueue the entityId back into \_availableIds.
3. **Create ComponentStore.cs:**
   * This class will store all components for all entities, organized by component type.
   * It will be a singleton, accessible via Instance.
   * **Field:**
     + private Dictionary<Type, Dictionary<int, IComponent>> \_components = new Dictionary<Type, Dictionary<int, IComponent>>();
     + This is a dictionary where the key is the Type of the component (e.g., typeof(PositionComponent)) and the value is another dictionary that maps an entity ID (int) to its instance of that component (IComponent).
   * **Methods:**
     + public void AddComponent<T>(int entityId, T component) where T : IComponent: Adds a component for a given entity. It must first check if the outer dictionary contains the type T. If not, it creates a new inner dictionary for it.
     + public T GetComponent<T>(int entityId) where T : IComponent: Retrieves a component of type T for a given entity. Returns default(T) if not found.
     + public bool HasComponent<T>(int entityId) where T : IComponent: Checks if an entity has a specific component type.
     + public void RemoveComponent<T>(int entityId) where T : IComponent: Removes a component of type T from an entity.
     + public void EntityDestroyed(int entityId): A helper method that iterates through all component types and removes any components associated with the given entityId. This should be called when an entity is destroyed.
4. **Integrate into Core.cs:**
   * In the Core class, add static properties to access the singletons:
     + public static EntityManager EntityManager => \_entityManager;
     + public static ComponentStore ComponentStore => \_componentStore;
   * Instantiate them: private static readonly EntityManager \_entityManager = new(); and private static readonly ComponentStore \_componentStore = new();

**Files to Create:**

* IComponent.cs
* EntityManager.cs
* ComponentStore.cs

**Context Files (Provide these to the AI):**

* Core.cs

### **Step 2: Refactor the Player into an Entity**

**Objective:**Deconstruct the monolithic Player class and represent the player as a standard entity with a collection of components. This is a critical step to prove the viability of the ECS architecture.

**Detailed Instructions:**

1. **Create New Component Files:**
   * **PositionComponent.cs**:
     + Will implement IComponent.
     + Will contain public Vector2 WorldPosition { get; set; }.
   * **LocalPositionComponent.cs**:
     + Will implement IComponent.
     + Will contain public Vector2 LocalPosition { get; set; }.
   * **StatsComponent.cs**:
     + Will implement IComponent.
     + This component will hold all the data currently in PlayerStats.cs. Copy all properties from PlayerStats (e.g., MaxEnergyPoints, CurrentEnergyPoints, WalkSpeed, etc.) and all methods (ExertEnergy, Rest, CanExertEnergy).
   * **ActionQueueComponent.cs**:
     + Will implement IComponent.
     + Will contain public List<PendingAction> ActionQueue { get; } = new List<PendingAction>();.
   * **PlayerTagComponent.cs**:
     + Will implement IComponent.
     + This is a "marker" component. It will be an empty class, used only to identify which entity is the player.
2. **Refactor GameState.cs:**
   * **Remove Player Field:** Delete private Player \_player;.
   * **Add Player Entity ID:** Add public int PlayerEntityId { get; private set; }.
   * **Constructor Changes:**
     + In the GameState constructor, after initializing the managers, create the player entity:
       - PlayerEntityId = Core.EntityManager.CreateEntity();
     + Remove the line \_worldEntities.Add(\_player);.
     + Add all the new components for the player entity using Core.ComponentStore.AddComponent():
       - Add a PositionComponent with the initial position.
       - Add a LocalPositionComponent with the initial local position (32, 32).
       - Add a StatsComponent, initializing it with the same values as the old PlayerStats constructor.
       - Add an ActionQueueComponent.
       - Add the PlayerTagComponent.
   * **Update Property Accessors:** Go through every property and method in GameState that used \_player and change it to query the ComponentStore.
     + PlayerWorldPos becomes Core.ComponentStore.GetComponent<PositionComponent>(PlayerEntityId).WorldPosition.
     + PlayerLocalPos becomes Core.ComponentStore.GetComponent<LocalPositionComponent>(PlayerEntityId).LocalPosition.
     + PendingActions becomes Core.ComponentStore.GetComponent<ActionQueueComponent>(PlayerEntityId).ActionQueue.
     + PlayerStats becomes Core.ComponentStore.GetComponent<StatsComponent>(PlayerEntityId).
   * **Update Methods:** Systematically refactor methods like SimulateActionQueueEnergy, ApplyActionEffects, QueueMovementInternal, etc., to get player data from the ComponentStore instead of the \_player object. For example, \_player.SetLocalPosition(newPos) becomes Core.ComponentStore.GetComponent<LocalPositionComponent>(PlayerEntityId).LocalPosition = newPos;.
3. **Delete Obsolete Files:**
   * The Player.cs file is now redundant and should be deleted.
   * The PlayerStats.cs file is now redundant (its logic is in StatsComponent.cs) and should be deleted.
   * The Entity.cs file is no longer the pattern we are following. Delete it.

**Files to Create:**

* PositionComponent.cs
* LocalPositionComponent.cs
* StatsComponent.cs
* ActionQueueComponent.cs
* PlayerTagComponent.cs

**Context Files (Provide these to the AI):**

* GameState.cs
* Player.cs
* PlayerStats.cs
* Entity.cs
* Core.cs
* IComponent.cs
* EntityManager.cs
* ComponentStore.cs
* PendingAction.cs (from GameState.cs)

### **Step 3: Create the First Systems for Input and Action Execution**

**Objective:**Move game logic out of the monolithic GameState class and into dedicated, reusable System classes that operate on components.

**Detailed Instructions:**

1. **Create ISystem.cs:**
   * Create an interface that all systems will implement.
   * It should contain one method: void Update(GameTime gameTime);
2. **Create PlayerInputSystem.cs:**
   * This system will handle player-specific command processing.
   * Implement ISystem.
   * The Update method will not be used for this system. Instead, it will have public methods that are called by the CommandProcessor.
   * Move the logic from the following GameState methods into this new class:
     + QueueNewPath
     + AppendPath
     + CancelPendingActions
     + QueueRest
     + QueueMovementInternal (and its public wrappers QueueRunMovement and QueueWalkMovement)
   * These methods will no longer be part of GameState. They will operate on the PlayerEntityId from GameState and access its components via the ComponentStore. You will need to pass Core.CurrentGameState or its PlayerEntityId to this system's methods.
3. **Create ActionExecutionSystem.cs:**
   * This system will be responsible for executing actions from any entity's queue and applying their effects.
   * Implement ISystem.
   * Move the logic from the following GameState methods into this class's Update method:
     + UpdateMovement
     + ApplyActionEffects
     + CalculateSecondsForAction
   * The Update method in this system will be the main driver of action execution. It will check GameState's \_isExecutingPath flag. If true, it will get the ActionQueueComponent for the PlayerEntityId and process the current action.
   * **Crucially, this system should be designed to eventually work on *any* entity, not just the player.** For now, it's acceptable to hard-code it to the player's entity ID for simplicity.
4. **Refactor Core.cs:**
   * Add a list of systems: private static readonly List<ISystem> \_updateSystems = new();
   * Instantiate your new systems and add them to the list.
   * In the Core.Update method (within the \_sceneManager.Update call for the TerminalMapScene), add a loop: foreach (var system in \_updateSystems) { system.Update(gameTime); }.
   * The CommandProcessor will now need a reference to an instance of PlayerInputSystem to call its methods.
5. **Refactor GameState.cs:**
   * Remove all the methods that were just moved to the new systems. The GameState class should now be significantly smaller, primarily holding state flags (\_isExecutingPath, CurrentMapView) and the PlayerEntityId.

**Files to Create:**

* ISystem.cs
* PlayerInputSystem.cs
* ActionExecutionSystem.cs

**Context Files (Provide these to the AI):**

* Core.cs
* GameState.cs
* CommandProcessor.cs (to see how commands are handled)
* ComponentStore.cs
* PositionComponent.cs
* LocalPositionComponent.cs
* StatsComponent.cs
* ActionQueueComponent.cs
* PlayerTagComponent.cs

**Step 4: Spatial Partitioning - The Chunk System**

**Objective:**Implement the core data structures for spatial partitioning. This involves creating a Chunk class to hold entities for a single world coordinate and a ChunkManager to organize the world. This step lays the groundwork for loading and processing only the entities relevant to the player's current world location.

**Detailed Instructions:**

1. **Create Chunk.cs:**
   * This class represents the data associated with a single **1x1 grid cell** on the world map.
   * **Properties:**
     + public Point ChunkCoords { get; }: The coordinate of this chunk, which directly corresponds to a world map position (e.g., (0,0), (0,1)).
     + public HashSet<int> EntityIds { get; } = new HashSet<int>();: A collection of entity IDs currently located within this chunk (i.e., at this world coordinate). A HashSet is used for fast additions and removals.
   * **Constructor:**
     + public Chunk(Point chunkCoords): Initializes the chunk with its coordinates.
2. **Create ChunkManager.cs:**
   * This class manages the entire collection of chunks. It will be a singleton.
   * **Fields:**
     + private Dictionary<Point, Chunk> \_chunks = new Dictionary<Point, Chunk>();: Stores all loaded chunks, keyed by their Point coordinates.
   * **Methods:**
     + public Point WorldToChunkCoords(Vector2 worldPosition): A static helper method that converts a Vector2 world position into a Point chunk coordinate. The logic is simply a cast: new Point((int)worldPosition.X, (int)worldPosition.Y).
     + public void RegisterEntity(int entityId, Vector2 worldPosition):
       - Calculates the chunk coordinates for the given worldPosition.
       - Checks if a Chunk already exists at those coordinates in the \_chunks dictionary. If not, it creates a new Chunk and adds it.
       - Adds the entityId to the EntityIds HashSet of the correct chunk.
       - Updates the entity's PositionComponent to store its current chunk coordinates.
     + public void UnregisterEntity(int entityId, Vector2 worldPosition):
       - Calculates the chunk coordinates.
       - Finds the corresponding chunk and removes the entityId from its EntityIds set.
     + public void UpdateEntityChunk(int entityId, Vector2 oldPosition, Vector2 newPosition):
       - Calculates the chunk coordinates for both oldPosition and newPosition.
       - If the coordinates are the same, do nothing.
       - If they are different, call UnregisterEntity with the oldPosition and RegisterEntity with the newPosition.
     + public List<int> GetEntitiesInChunk(Point chunkCoords): Returns a list of entity IDs from the specified chunk. Returns an empty list if the chunk doesn't exist.
     + public List<int> GetEntitiesInTetherRange(Point centralChunkCoords):
       - This method's purpose is to get all entities relevant to the player. In this design, that means **only the entities in the player's current chunk**.
       - The implementation should simply call GetEntitiesInChunk(centralChunkCoords) and return the result. The 9-chunk logic is not needed.
3. **Modify PositionComponent.cs:**
   * Add a new property: public Point CurrentChunk { get; set; }. This will be set by the ChunkManager and allows for efficient lookup of an entity's current chunk without recalculating.
4. **Integrate into Core.cs:**
   * Add a static property for the ChunkManager singleton: public static ChunkManager ChunkManager => \_chunkManager;
   * Instantiate it: private static readonly ChunkManager \_chunkManager = new();
5. **Integrate into GameState.cs:**
   * In the constructor, after creating the player entity and its PositionComponent, register it with the ChunkManager:
     + var playerPos = Core.ComponentStore.GetComponent<PositionComponent>(PlayerEntityId).WorldPosition;
     + Core.ChunkManager.RegisterEntity(PlayerEntityId, playerPos);

### **Step 5: Activating Entities via the Chunk System**

**Objective:**Make the game systems aware of the spatial partitioning. The ActionExecutionSystem will now update an entity's chunk registration upon movement. We will also introduce the concept of "active entities" (those within the player's tether range) to limit processing.

**Detailed Instructions:**

1. **Refactor ActionExecutionSystem.cs:**
   * Modify the logic that handles WalkMove and RunMove actions.
   * After calculating the nextPosition and before updating the PositionComponent, store the oldPosition.
   * After updating the PositionComponent with the newPosition, call the ChunkManager:
     + Core.ChunkManager.UpdateEntityChunk(entityId, oldPosition, newPosition);
   * This ensures that every time an entity moves, its chunk registration is correctly updated.
2. **Modify GameState.cs:**
   * Add a new property to hold the list of entities that should be processed this frame:
     + public List<int> ActiveEntities { get; private set; } = new List<int>();
   * Create a new method UpdateActiveEntities():
     + This method will be called once per game tick.
     + It gets the player's current PositionComponent.
     + It converts the player's world position to chunk coordinates using ChunkManager.WorldToChunkCoords.
     + It calls ChunkManager.GetEntitiesInTetherRange with the player's chunk coordinates.
     + It sets the ActiveEntities list to the result of this call.
     + **High-Importance Entities:** This is where you implement the "always processed" feature from your roadmap.
       - Create a new marker component: HighImportanceComponent.cs which implements IComponent.
       - In UpdateActiveEntities, after getting the tethered entities, query the ComponentStore for all entities with a HighImportanceComponent.
       - Add these entities to the ActiveEntities list (use Union or a loop with a check to avoid duplicates).
3. **Refactor Core.cs:**
   * In the main Update loop (inside the TerminalMapScene logic), before the systems are updated, call the new method on GameState:
     + CurrentGameState.UpdateActiveEntities();
   * Now, all systems updated after this point will have access to a fresh list of active entities for the current frame.
4. **Update Systems to Use Active Entities:**
   * For now, this is a conceptual change. In later steps, systems like AISystem and RenderSystem will be modified to iterate over GameState.ActiveEntities instead of all entities in the world. The ActionExecutionSystem is a special case because it only needs to act on the player, which is always active.

**Files to Create:**

* HighImportanceComponent.cs

**Context Files (Provide these to the AI):**

* ActionExecutionSystem.cs
* GameState.cs
* Core.cs
* ChunkManager.cs
* PositionComponent.cs
* ComponentStore.cs
* IComponent.cs

### **Step 6: Tiered Update Frequencies & System Management**

**Objective:**Implement the "Tiered Update Frequencies" from the roadmap. This will allow different systems to run at different intervals (e.g., AI every 0.5 seconds, rendering every frame), saving significant CPU resources. We will create a SystemManager to orchestrate this.

**Detailed Instructions:**

1. **Create SystemManager.cs:**
   * This class will replace the simple List<ISystem> in Core.cs.
   * **Fields:**
     + private List<SystemEntry> \_systems = new List<SystemEntry>();
   * **Methods:**
     + public void RegisterSystem(ISystem system, float updateIntervalSeconds = 0f):
       - This method adds a system to the manager.
       - An updateIntervalSeconds of 0 means the system should run every frame.
       - It creates a SystemEntry object and adds it to the \_systems list.
     + public void Update(GameTime gameTime):
       - This is the main update loop for all systems.
       - It iterates through the \_systems list.
       - For each SystemEntry, it adds gameTime.ElapsedGameTime.TotalSeconds to the entry's \_accumulator.
       - If the \_accumulator is greater than or equal to the UpdateInterval, it calls the system's Update method and then subtracts the UpdateInterval from the \_accumulator. This handles both per-frame and periodic updates correctly.
2. **Create SystemEntry.cs (as a private nested class or a separate file):**
   * This is a small data-holding class used by SystemManager.
   * **Properties:**
     + public ISystem System { get; }
     + public float UpdateInterval { get; }
     + public float Accumulator { get; set; }
   * **Constructor:**
     + Takes the system and interval as parameters.
3. **Refactor Core.cs:**
   * Remove the \_updateSystems list and the foreach loop that updates them.
   * Add a static property for the SystemManager: public static SystemManager SystemManager => \_systemManager;
   * Instantiate it: private static readonly SystemManager \_systemManager = new();
   * In the Initialize or LoadContent method, register your existing systems with the new manager:
     + \_systemManager.RegisterSystem(new ActionExecutionSystem(), 0f); // Runs every frame
     + (The PlayerInputSystem doesn't have an Update loop, so it doesn't need to be registered here).
   * In the Core.Update method (for the TerminalMapScene), call the manager's update:
     + \_systemManager.Update(gameTime);

**Files to Create:**

* SystemManager.cs
* SystemEntry.cs (can be inside SystemManager.cs)

**Context Files (Provide these to the AI):**

* Core.cs
* ISystem.cs
* ActionExecutionSystem.cs
* PlayerInputSystem.cs

Step 7: Generalizing Actions  
 Objective:  
 Refactor the PendingAction class and ActionType enum into a more flexible, object‑oriented system using an IAction interface. This will allow us to easily define new, complex actions like Attack, Interact, or CastSpell in the future.

Detailed Instructions:  
 1.  Create IAction.cs:  
 o  This interface will be the base for all actions.  
 o  It will implement IComponent. This is a key design choice: an action being executed is treated as a temporary component on an entity.  
 o  Properties:  
 §  public int ActorId { get; } : The ID of the entity performing the action.  
 §  public bool IsComplete { get; set; } : A flag to signal when the action is finished.

2.  Create Concrete Action Classes:  
 o  MoveAction.cs:  
 §  Implements IAction.  
 §  Properties: ActorId, IsComplete, public Vector2 Destination { get; }, public bool IsRunning { get; }.  
 o  RestAction.cs:  
 §  Implements IAction.  
 §  Properties: ActorId, IsComplete, public RestType RestType { get; }.

Files to Create:  
 · IAction.cs  
 · MoveAction.cs  
 · RestAction.cs

Context Files (Provide these to the AI):  
 · ActionQueueComponent.cs  
 · PlayerInputSystem.cs  
 · ActionExecutionSystem.cs  
 · GameState.cs (to remove PendingAction and ActionType)  
 · ComponentStore.cs  
 · IComponent.cs

Step 7.5: Integrate & Clean Up New Actions  
 Objective:  
 Wire the new IAction‑based classes into your systems, replace the old PendingAction queue, and remove obsolete code.

Detailed Instructions:  
 1.  Refactor ActionQueueComponent.cs:  
 o  Change the ActionQueue property from List<PendingAction> to public Queue<IAction> ActionQueue { get; } = new Queue<IAction>();

2.  Refactor PlayerInputSystem.cs:  
 o  Update all methods that created PendingAction objects. They should now create instances of MoveAction or RestAction and enqueue them into the player's ActionQueueComponent.  
 o  For example, QueueRunMovement will now create a new MoveAction(playerId, destination, true).

3.  Refactor ActionExecutionSystem.cs:  
 o  Dequeue & Attach: look for entities that have an ActionQueueComponent with items but no IAction component; dequeue the next action and Core.ComponentStore.AddComponent(entityId, nextAction).  
 o  Process Each Action Type:  
 §  Query all entities with a MoveAction component, run movement logic, set action.IsComplete = true when done.  
 §  Query all entities with a RestAction component, run rest logic, set action.IsComplete = true.  
 o  Cleanup Completed Actions: query for all IAction components where IsComplete is true and remove them from their entities.

4.  Delete Obsolete Code:  
 o  Remove the PendingAction class and ActionType enum (likely inside GameState.cs).

Files to Create:  
 · (none – you’re using the IAction implementations from Step 7)

Context Files (Provide these to the AI):  
 · ActionQueueComponent.cs  
 · PlayerInputSystem.cs  
 · ActionExecutionSystem.cs  
 · GameState.cs  
 · ComponentStore.cs

### **Step 8: Implementing Basic AI**

**Objective:**Bring the first non-player entity to life. We will create a simple "Wanderer" NPC that moves randomly around the world, proving that our systems can handle multiple entities.

**Detailed Instructions:**

1. **Create AI-Related Components:**
   * **AIComponent.cs**: Implements IComponent. Will hold the AI's state. For now, it can be simple: public AIState CurrentState { get; set; } = AIState.Idle;, public float StateTimer { get; set; }.
   * **AIState.cs**: An enum with initial states: Idle, Wandering.
   * **NPCTagComponent.cs**: An empty marker component, similar to PlayerTagComponent, to identify NPCs.
2. **Create AISystem.cs:**
   * Implements ISystem. This system is the "brain" for our NPCs.
   * In its Update method:
     + It iterates through all ActiveEntities from GameState.
     + For each entity, it checks if it has both an AIComponent and an NPCTagComponent.
     + It also checks that the entity does *not* have an active IAction component (meaning it's not busy).
     + **FSM Logic:**
       - If the AIComponent.CurrentState is Idle, it decrements the StateTimer. When the timer hits zero, it transitions to Wandering, sets a new timer, and picks a random adjacent passable tile. It then creates a new MoveAction and enqueues it in the entity's ActionQueueComponent.
       - If the AIComponent.CurrentState is Wandering, it does nothing (it's waiting for the ActionExecutionSystem to complete the move). Once the move is done and the MoveAction component is removed, on the next AISystem tick, it will see the entity is not busy and transition it back to Idle with a new random timer.
3. **Refactor ActionExecutionSystem.cs:**
   * The system must be generalized to work for any entity, not just the player.
   * Its main loop should iterate over all ActiveEntities, check for actions to start, and process them. The logic is the same, but the entityId is no longer hard-coded to the player.
4. **Spawn an NPC in GameState.cs:**
   * In the constructor, after creating the player, create a new NPC entity.
   * int npcId = Core.EntityManager.CreateEntity();
   * Give it the necessary components: PositionComponent (spawn it near the player), ActionQueueComponent, AIComponent, and NPCTagComponent.
   * Register it with the ChunkManager.
5. **Register the AISystem in Core.cs:**
   * In the SystemManager registration block, add the new system. It should run at a lower frequency.
   * \_systemManager.RegisterSystem(new AISystem(), 0.5f); // AI thinks twice a second

**Files to Create:**

* AIComponent.cs
* AIState.cs
* NPCTagComponent.cs
* AISystem.cs

**Context Files (Provide these to the AI):**

* GameState.cs
* Core.cs
* ActionExecutionSystem.cs
* SystemManager.cs
* ChunkManager.cs
* ComponentStore.cs
* ActionQueueComponent.cs
* MoveAction.cs
* IAction.cs

### **Step 9: Data-Driven Archetypes**

**Objective:**Stop hard-coding entity creation in GameState.cs. Instead, define entity "templates" in external JSON files. This allows for rapid content creation and is the final major architectural piece.

**Detailed Instructions:**

1. **Create Archetype.cs:**
   * This class will be a direct mapping of our JSON structure. It should be designed for easy deserialization (e.g., using System.Text.Json).
   * **Properties:**
     + public string Id { get; set; } (e.g., "player", "goblin\_warrior")
     + public string Name { get; set; }
     + public List<Dictionary<string, object>> Components { get; set; }
       - This is the tricky part. Each item in the list represents a component. The dictionary will have a "type" key (e.g., "type": "PositionComponent") and other keys for the component's properties (e.g., "x": 0, "y": 0).
2. **Create ArchetypeManager.cs:**
   * A singleton responsible for loading and storing archetypes.
   * **Fields:**
     + private Dictionary<string, Archetype> \_archetypes = new Dictionary<string, Archetype>();
   * **Methods:**
     + public void LoadArchetypes(string directoryPath):
       - Loads all .json files from a given directory.
       - Deserializes each file into an Archetype object.
       - Stores it in the \_archetypes dictionary, keyed by its Id.
     + public Archetype GetArchetype(string id): Retrieves a loaded archetype.
3. **Create Spawner.cs:**
   * A static helper class to create entities from archetypes.
   * **Method:**
     + public static int Spawn(string archetypeId, Vector2 position):
       - Gets the Archetype from the ArchetypeManager.
       - Creates a new entity ID using EntityManager.
       - Iterates through the Components list in the archetype data.
       - For each component definition, it uses reflection (Type.GetType and Activator.CreateInstance) to create an instance of the component.
       - It then uses reflection again to set the properties of the component instance from the JSON data.
       - It adds the fully-formed component to the ComponentStore for the new entity.
       - It sets the PositionComponent's value to the passed-in position.
       - It registers the new entity with the ChunkManager.
       - Returns the new entity's ID.
4. **Create JSON Archetype Files:**
   * Create a new folder in your project, Content/Archetypes.
   * **player.json**: Define the player archetype with its components (PositionComponent, StatsComponent, PlayerTagComponent, etc.).
   * **wanderer\_npc.json**: Define the NPC archetype with its components (PositionComponent, AIComponent, NPCTagComponent, etc.).
5. **Refactor GameState.cs and Core.cs:**
   * In Core.LoadContent, call ArchetypeManager.Instance.LoadArchetypes("Content/Archetypes").
   * In GameState.constructor, remove the manual entity creation and component-adding. Replace it with:
     + PlayerEntityId = Spawner.Spawn("player", new Vector2(0, 0));
     + Spawner.Spawn("wanderer\_npc", new Vector2(5, 5));

**Files to Create:**

* Archetype.cs
* ArchetypeManager.cs
* Spawner.cs
* Content/Archetypes/player.json
* Content/Archetypes/wanderer\_npc.json

**Context Files (Provide these to the AI):**

* GameState.cs
* Core.cs
* All Component class files (PositionComponent.cs, StatsComponent.cs, etc.)
* EntityManager.cs
* ComponentStore.cs
* ChunkManager.cs

### **Step 10: Modular Interaction - A Simple Combat System**

**Objective:**Demonstrate the power and scalability of the completed architecture by adding a new gameplay feature—combat—without majorly altering existing systems.

**Detailed Instructions:**

1. **Create New Combat Components:**
   * **HealthComponent.cs**: Implements IComponent. Properties: public int CurrentHealth { get; set; }, public int MaxHealth { get; set; }.
   * **CombatantComponent.cs**: Implements IComponent. Properties: public int AttackPower { get; set; }, public float AttackRange { get; set; }.
2. **Create AttackAction.cs:**
   * Implements IAction.
   * **Properties:** ActorId, IsComplete, public int TargetId { get; }.
3. **Create CombatSystem.cs:**
   * Implements ISystem.
   * In its Update method:
     + It queries for all entities that have an AttackAction component.
     + For each AttackAction:
       - It gets the ActorId and TargetId.
       - It verifies both actor and target are still valid and have CombatantComponent and HealthComponent.
       - It gets the actor's AttackPower from its CombatantComponent.
       - It gets the target's HealthComponent and subtracts the AttackPower from CurrentHealth.
       - It prints a message to the terminal: "[COMBAT] Entity {ActorId} attacked Entity {TargetId} for {AttackPower} damage!"
       - If the target's health is <= 0, it can fire an EntityDiedEvent (future step) or simply destroy the entity. For now, just log it.
       - It sets action.IsComplete = true.
4. **Update AISystem.cs:**
   * Add a new state to the AIState enum: Chasing.
   * In the AISystem.Update method, add logic for the AI to detect the player.
   * If an NPC is Idle or Wandering, it should check the distance to the player.
   * If the player is within a certain "aggro range", the AI state changes to Chasing.
   * In the Chasing state, the AI will enqueue MoveActions toward the player's position.
   * If the AI is in range to attack (check AttackRange from its CombatantComponent), instead of moving, it will enqueue an AttackAction targeting the PlayerEntityId.
5. **Update Archetypes:**
   * Modify player.json and wanderer\_npc.json to include HealthComponent and CombatantComponent with some default values. Rename wanderer\_npc to goblin\_warrior to be more thematic.
6. **Register CombatSystem in Core.cs:**
   * \_systemManager.RegisterSystem(new CombatSystem(), 0f); // Combat resolves instantly