ChatGPT

Godot Hex-Grid MMO Project Structure

Building a large open-world MMO in Godot demands a carefully organized project. Below we present two directory layouts:

- **Layout 1 (Locality of Behavior)**: co-locates related game code, assets, and tests by feature/system [1].
- **Layout 2 (Industry-Recommended)**: a more traditional separation by technical concern (e.g. scenes, scripts, assets).

Each layout is followed by detailed explanations, comparisons, and integration with key systems (hex grids, networking, streaming, etc.). Throughout, we cite Godot best practices and game-development sources.

Layout 1: Locality of Behavior (LoB)

In this **LoB** structure, each major feature or system lives in its own folder. Code, scenes, assets, and tests for that feature are grouped together. This co-location (a core LoB principle [1]) makes each feature self-contained and easier to understand. For example, the combat system has its own folder with scripts, scene files, and assets.

```
project_root/
├── addons/                       # Godot plugins (GUT, Tiled importer, etc.)
│   └── gut/                      # GUT unit-testing framework plugin
├── autoload/                     # Singletons / autoloaded scripts
│   ├── GameState.gd
│   └── NetworkManager.gd
├── combat/                       # Combat System (LoB grouping)
│   ├── Scenes/
│   │   ├── CombatArena.tscn
│   │   └── BossFight.tscn
│   ├── Scripts/
│   │   ├── CombatManager.gd
│   │   ├── Damage.gd
│   │   └── Skills/
│   │       ├── FireballSkill.gd
│   │       └── HealSkill.gd
│   ├── Assets/
│   │   ├── VFX/
│   │   │   └── FireballParticles.tres
│   │   └── SFX/
│   │       ├── sword_swing.ogg
│   │       └── fireball.wav
│   └── test/                   # GUT tests for combat
│       └── test_combat_manager.gd
├── inventory/                  # Inventory System
│   ├── Scenes/
│   │   └── InventoryUI.tscn
│   ├── Scripts/
│   │   ├── InventoryManager.gd
│   │   └── Item.gd
│   ├── Assets/
│   │   └── Icons/
│   │       ├── potion_icon.png
│   │       └── sword_icon.png
│   └── test/
│       └── test_inventory_manager.gd
├── world/                      # World/Hex Grid & Chunk Management
│   ├── HexGrid/
│   │   ├── HexCell.tscn
│   │   ├── HexGrid.gd
│   │   └── TileData.gd
│   ├── Chunks/
│   │   ├── ChunkManager.gd
│   │   └── ChunkLoader.gd
│   ├── Scenes/
│   │   └── Overworld.tscn      # Root world scene
│   ├── Assets/
│   │   ├── Terrain/
│   │   │   ├── grass.png
│   │   │   └── water.png
│   │   └── LOD/                # Level-of-Detail models
│   │       ├── tree_high.tres
│   │       └── tree_low.tres
│   └── test/
│       ├── test_hexgrid.gd
│       └── test_chunk_manager.gd
├── npc/                        # NPC and AI System
│   ├── Scenes/
│   │   ├── NPC.tscn
│   │   └── Creature.tscn
│   ├── Scripts/
│   │   ├── AIManager.gd
│   │   └── BehaviorTree.gd
│   ├── Assets/
│   │   └── Animations/
│   │       └── idle_anim.tres
│   └── test/
│       └── test_ai_manager.gd
├── player/                     # Player character, input, and stats
│   ├── Scenes/
│   │   └── Player.tscn
│   ├── Scripts/
│   │   ├── PlayerController.gd
│   │   └── PlayerStats.gd
│   ├── Assets/
│   │   └── HUD/
│   │       └── health_bar.png
│   └── test/
│       └── test_player_stats.gd
├── ui/                         # UI Views not specific to other systems
│   ├── Scenes/
│   │   ├── MainMenu.tscn
│   │   └── PauseMenu.tscn
│   ├── Scripts/
│   │   ├── MenuManager.gd
│   │   └── Button.gd
│   ├── Assets/
│   │   ├── Themes/
│   │   │   └── default_theme.tres
│   │   └── Fonts/
│   │       └── game_font.ttf
│   └── test/
│       └── test_menu_manager.gd
├── network/                    # Networking and protocol definitions
│   ├── Client.gd
│   ├── Server.gd
│   ├── Protocols/
│   │   ├── Messages.gd       # RPC and packet definitions
│   │   └── Serialization.gd
│   └── test/
│       └── test_protocols.gd
├── tools/                      # Editor plugins and conversion scripts
│   ├── HexMapEditorPlugin.gd   # Custom Godot plugin for hex map design
│   ├── DataConverters/
│   │   └── TiledImporter.gd    # Converts Tiled JSON to .tres
│   └── test/
│       └── test_tools.gd
├── data/                       # Game data definitions and persistence
│   ├── Items/
│   │   └── item_data.tres      # Godot Resource for all item stats
│   ├── TerrainTypes.tres
│   ├── SaveData.gd
│   └── test/
│       └── test_data_integrity.gd
└── project.godot              # Godot project file (engine config)
```

Directory Explanations: Each major folder is a self-contained system:

- `combat/` – Contains the combat subsystem. Includes *scenes* (e.g. arenas, battles), *scripts* (combat logic like `CombatManager.gd`), *assets* (VFX, SFX used only by combat), and its own test suite. By grouping combat logic and assets together, a developer can understand and modify all combat behavior in one place [1]. Script and scene names use PascalCase matching Godot style (e.g. `CombatManager.gd`, `BossFight.tscn`), and assets are in subfolders by type.

- `inventory/`, `npc/`, `player/`, `ui/`, etc. – Similarly, each system's code, scenes, and assets are co-located. For example, `inventory/` has the inventory UI scene, item scripts, icons, and tests. This minimizes cross-folder navigation. Each subfolder (Scenes, Scripts, Assets, test) is consistently named in snake_case or PascalCase as needed.

- `world/` – Manages the hex grid and world streaming. Subfolders like `HexGrid/` contain hex-related scripts (`HexGrid.gd`, `HexCell.tscn`) and `Chunks/` has the streaming manager. Assets like terrain textures and LOD meshes reside under `world/Assets`. This keeps world-gen code (grid logic, chunking) together.
- `network/` – All networking code and protocol definitions. Client/server scripts and message formats reside here. Separating networking prevents mixing it with gameplay code.
- `tools/` – Custom Editor plugins and conversion scripts (e.g. hex map editor, data converters). Designers' tools are housed here, making version control easier for plugins.
- `data/` – Static data resources (item/terrain definitions, save formats). Using `.tres` files allows text-based version control.
- `autoload/` – Project-wide singletons (e.g. `GameState.gd`, `NetworkManager.gd`) set via Project Settings, since these are needed everywhere.
- `addons/` – Third-party plugins (e.g. GUT unit-test plugin) are placed here in Godot's convention.

Naming & Conventions: In this layout we follow Godot's styleguide: scripts and folders use `snake_case.gd`, scenes use `PascalCase.tscn`. Tests use prefix `test_` (as GUT expects [2]). For example, `CombatManager.gd` might contain `class_name CombatManager`, but the file is `combat_manager.gd`. Each test script extends `GutTest` and is named like `test_combat_manager.gd` [3]. Assets are stored as `.tres` or common formats in clearly named folders (e.g. `fireball.wav`, `water.png`). This structure scales by simply adding new feature folders; each feature grows with its own subfolders. Dependencies between systems (e.g. combat calling player stats) can refer by the autoload or via signals, but code remains organized by feature.

Layout 2: Recommended (Layered/Modular)

This industry-style structure separates by **concern** rather than feature. Code and assets are organized in broad categories (Scenes, Scripts, Assets, etc.), making it easier to find all UI or all world files at once.

```
project_root/
├── addons/
│   └── gut/                    # GUT plugin
├── autoload/
│   ├── GameState.gd
│   └── NetworkManager.gd
├── Assets/                     # Global assets by type
│   ├── Art/
│   │   ├── Characters/
│   │   │   ├── player.png
│   │   │   └── goblin.png
│   │   ├── Environments/
│   │   │   └── forest_tile.png
│   │   └── UI/
│   │       └── button.png
│   ├── Audio/
│   │   ├── Music/
│   │   └── SFX/
│   ├── Fonts/
│   └── Shaders/
├── Scenes/                     # All scene files
│   ├── MainMenu.tscn
│   ├── Overworld.tscn
│   ├── CombatArena.tscn
│   └── UI/
│       ├── InventoryUI.tscn
│       └── HUD.tscn
├── Scripts/                    # All script files
│   ├── combat/
│   │   └── CombatManager.gd
│   ├── inventory/
│   │   └── InventoryManager.gd
│   ├── world/
│   │   ├── HexGrid.gd
│   │   └── ChunkManager.gd
│   ├── npc/
│   │   └── AIManager.gd
│   ├── player/
│   │   └── PlayerController.gd
│   └── ui/
│       ├── MenuManager.gd
│       └── Button.gd
├── Data/                       # Data and definitions
│   ├── items.tres
│   ├── terrain.tres
│   └── save/
│       └── SaveData.gd
├── Tools/                         # Editor plugins, custom tools
│   ├── HexMapEditorPlugin.gd
│   └── TiledImporter.gd
├── Tests/                         # Centralized tests
│   ├── combat/
│   │   └── test_combat_manager.gd
│   ├── world/
│   │   └── test_chunk_manager.gd
│   ├── network/
│   │   └── test_protocols.gd
│   └── inventory/
│       └── test_inventory_manager.gd
├── network/                       # Networking code (if separate)
│   ├── Client.gd
│   ├── Server.gd
│   └── Protocols/
│       └── Messages.gd
└── project.godot
```

Directory Explanations:

- `Assets/` – All art, audio, fonts, shaders, etc. grouped by type (sprites, tiles, sounds). For example, character sprites are in `Assets/Art/Characters/`. This makes it easy for designers to browse assets of a given type.
- `Scenes/` – All `.tscn` scene files in one place, optionally subfolders by purpose (UI scenes in `Scenes/UI/`). Scenes reference assets via relative paths. This hierarchy is flat but can scale with subfolders.
- `Scripts/` – Code is still separated by functional area (combat, world, UI) but under one Scripts folder. Each subfolder (`combat/`, `world/`, etc.) contains the GDScript logic. This approach balances feature modules with central location.
- `Data/` – Game data resources and configs (`.tres` or scripts for save/data classes). This keeps all definitions together for easy version control and editing.
- `Tests/` – GUT tests are organized by module under one `Tests` folder. For instance, `Tests/combat/` holds combat tests. GUT is pointed to this root test directory [4]. This contrasts with LoB, but still keeps test files near code types.
- `network/` – As in Layout 1, contains networking code. This could be merged into Scripts or separate if using a non-Godot server (TS backend).
- `Tools/` – Editor plugins or data converters. Having a single Tools directory simplifies plugin management.

Naming & Conventions: In Layout 2, we also follow Godot naming conventions. Folders are lower_snake or PascalCase as above. Scenes still PascalCase (e.g. `Overworld.tscn`), scripts snake_case (e.g. `combat_manager.gd` with `class_name CombatManager`). Tests use `test_` prefix [3]. For example, `Scripts/combat/CombatManager.gd` might have a corresponding test `Tests/combat/test_combat_manager.gd`.

Comparative Analysis

- **Locality vs Layering:** The LoB layout co-locates related code/assets/tests, making feature logic easy to navigate [1]. Team members working on one system find everything in one place. In contrast, the modular layout separates by file type; this can help reuse (e.g. shared scripts) but may force a developer to jump between folders. LoB may lead to some duplication (e.g. similar utilities in multiple modules) whereas the layered layout avoids that by centralizing common code.
- **Scalability:** Both can scale, but differently. LoB scales by adding new feature folders (self-contained). The layered layout scales by adding new categories or subfolders under existing categories. For example, a new game system under LoB is a new folder; in layered, it might add subfolders under `Scripts/` and `Scenes/`.
- **Merge Conflicts:** In LoB, code changes for one system rarely conflict with another, reducing merge issues. In layered, many devs touch common folders (`Scripts/`, `Assets/`) which could increase conflicts on shared resources.
- **Learning Curve:** LoB offers instant context for a system but requires understanding its boundaries. Layered is more familiar to programmers who think “Assets vs Code vs Scenes” but requires discipline to not mix concerns in one folder.
- **Godot Conventions:** Godot itself doesn’t force one style. Official tutorials often mix both approaches. The recommended layout aligns with Godot’s examples of grouping all scenes together, whereas LoB aligns with component-based design.

GUT Testing Integration

Both layouts use GUT for unit tests, but organization differs:

- **Layout 1 (LoB):** Each module/folder has its own `test/` subfolder. For example, `combat/test/test_combat_manager.gd`. In Project Settings → GUT, include `combat/test` (and each module’s `test`) in the **Test Directories** list [4]. This ties tests closely to code. Tests should extend `GutTest` and file names start with `test_` [5]. This ensures any change in a system’s code encourages updating its tests.
- **Layout 2 (Layered):** A single `Tests/` directory holds all tests, organized by system. GUT is pointed at `Tests/`. For example, combat tests live in `Tests/combat/`. File-naming conventions (prefix/suffix) are the same [2]. The advantage is one place to run all tests; however, it introduces an extra hop to find a module’s tests. Either way, following GUT’s conventions (prefix `test_`, class extends `GutTest`) is critical.

Integration tests (e.g. full client-server flow) can live in a dedicated `Tests/integration/` area or as part of network tests in both layouts. Performance tests can use GUT’s profiling and custom benchmarks. Automation: both can tie into CI by invoking GUT's command-line runner on the `tests/` directories.

Asset Pipeline & Level Design Workflow

An effective pipeline spans design to runtime:

- **Design Tools:** Designers can use Tiled (tile-based editor) or custom Godot tools. With Tiled, one can use plugins like *godot-tiled-importer* to convert `.tmx` or `.json` maps into Godot scenes [6]. Alternatively, a *HexMapEditorPlugin.gd* (in the `tools/` folder) could allow painting hex grids directly in the Godot editor. In both layouts, such plugins reside under a `tools/` or `addons/` directory. Non-technical designers get GUI tools (e.g. tile palettes, brush tools) so they don't manually edit data files. Everything stays in version control since Godot's scenes (`.tscn`) and resources (`.tres`) are text-based.

- **Data Conversion:** Raw design data (like Tiled JSON or spreadsheets) is converted to game format at build time or editor time. This might involve GDScript converter scripts (e.g. `TiledImporter.gd`) that output `TileSet`, `TileMap`, or custom chunk data resources. Defining tile types and properties is done in data files or Godot resources; for example, each tile can have custom properties (e.g. terrain cost) that the importer preserves. Batch processing can be set up via EditorScripts or external build scripts to regenerate all maps. Tools validate data (check missing assets, tile rules) and log errors during import.

- **Streaming & LOD:** In both layouts, streaming is handled in the world or core systems. Designers create regions or chunks (could be separate scene files or serialized chunk files). The pipeline can pack these chunks (as `.scn` or custom binary via Godot's `ResourceSaver`) for fast loading. At runtime, a ChunkManager loads/unloads based on player position [7] [8]. Level of Detail (LOD) assets should be named consistently (e.g. `tree_high.tres` vs `tree_low.tres`) and loaded automatically by scripts based on distance, to save memory.

- **Serialization:** Static data (items, tile properties) is stored in Godot Resource files (`.tres`). For dynamic world chunks or large maps, one can use Godot's binary file I/O or even Godot's `BinaryFile` API. (Some projects use a custom BinarySerializer for huge worlds, as discussed in community forums [7].) Crucially, using Godot's scene files or resources means level data remains text-searchable in Git.

- **Optimization:** During build, assets can be preprocessed (e.g. lightmaps baked, meshes simplified). Godot's import pipeline (resource importer settings) should use lossless formats for source assets and compressed formats for exports. Version control tracking of import settings ensures consistency across the team.

Overall, the asset pipeline should be as automated as possible: designers push new map files or art, an editor script/importer updates the Godot project, and then the team can immediately playtest with the latest content.

Chunk-Based Streaming System

Both layouts rely on a **chunk manager** that loads world sections as needed:

- **Loading Logic:** Using VisibleOnScreenNotifiers or distance checks, the game tracks which hex-chunk the player is in [9]. When the player nears a chunk boundary, `ChunkManager` loads the adjacent chunk scene (instancing it under the world root) and queues freeing far-away chunks [9] [7]. For example, if chunks are 100×100 hex tiles, moving into the edge of chunk (X) triggers loading of chunk (X+1). The code for this lives in `world/Chunks/` in Layout 1 or in `Scripts/world/ChunkManager.gd` in Layout 2.

- **Memory Efficiency:** Chunks are freed with `queue_free()` when no longer needed [9]. Resources inside chunks (textures, meshes) should use `free()` after chunk unload or rely on Godot's resource cache to drop unused assets. Limiting the number of loaded chunks (e.g. a 3×3 grid around the player) keeps memory low. Using `.tscn` for chunk data (text scenes) allows fast incremental loading without parsing giant files.

- **Networking:** On the server side, only chunks around active players are simulated and sent to clients. Network messages include only the visible chunk data for that client. This minimizes bandwidth by not streaming the entire world at once.

- **Consistency:** Both layouts incorporate the same streaming logic; only the code location differs. In LoB, streaming code is in `world/Chunks/`. In the layered layout, it might be in `Scripts/world/`. The migration between these is straightforward (see below).

- **Edge Cases:** The streaming system may include a "buffer" so the player never sees an empty void (for example, load far ahead of movement or show a loading animation). Designers should design chunk boundaries along logical barriers to hide load delays (e.g. a narrow tunnel or door).

In summary, chunk streaming is architecture-agnostic: it is implemented once (in the World system) and used in either structure. The key is modularity: the `ChunkManager` should work with whatever folder it resides in.

Tick-Based Simulation Architecture

A lockstep tick system is critical for MMO consistency:

- **Fixed Ticks:** The server runs a fixed tick rate (e.g. 20 ticks per second) to advance game state. In Godot, this can be implemented with an `Engine.get_physics_ticks_per_second()` or a custom timer. The `World` or `GameState` autoload invokes a `_physics_process()` or a custom `_tick()` on all active entities each tick. Clients apply inputs immediately and also predict motion, but ultimately interpolate/extrapolate state from server updates.

- **Entity Updates:** Entities in loaded chunks register for tick events. For performance, loop through only active entities (e.g. track them in the chunk scripts). This avoids iterating over thousands of off-screen objects. In Layout 1, chunk scenes may contain all entities and handle their ticks internally; in Layout 2, a central simulation manager might call `Entity.update_tick()` on each. Either way, design each entity script (e.g. `Enemy.gd`) with a `tick(delta)` function rather than relying solely on `_process()`. This ensures deterministic updates.

- **Performance:** To maintain a stable tick, heavy computations (pathfinding, AI) should be spread across ticks or done asynchronously. Use fixed arrays or object pools where possible. Since Godot's `_physics_process` runs on the main thread, ensure it completes quickly; offload pathfinding to a separate system (see next section). The fixed tick rate means all clients see the same timeline, aiding anti-cheat (clients can't speed up).

- **Synchronization:** The server state (position, health, etc.) is sent to clients each tick (or as needed). Clients apply these states in order, minimizing correction. In both layouts, networking code (client/server) ties into the tick loop to send/receive updates. State interpolation (smoothing) happens on the client using the last few ticks to avoid jitter.

- **Modularity:** We recommend a dedicated TickManager node (autoload) that dispatches tick events. In LoB, this might be in a core folder; in layered, perhaps under `Scripts/core/`. Tests can simulate ticks using GUT by manually calling `tick()` methods, ensuring logic correctness.

In either structure, the tick system is placed within the core simulation code. The directory layout does not affect runtime tick performance; it affects maintainability. Keeping tick logic separate (e.g. in a `tick/` or `world/` folder) clarifies responsibilities.

Pathfinding & Navigation

For hex-grid movement, efficient pathfinding is crucial:

- **Hex A***: We use an A** implementation on a hex grid. The `HexGrid` library (see [20]) includes an A* class where you set obstacles and call `find_path(start, goal)` [10]. This handles flat or pointy hexes in axial/cube coords [11] [12]. Tile movement cost and barriers are encoded (e.g. terrain cost, one-way paths). For multi-layer (ground vs flying), maintain separate grids or weight attributes. The pathfinding code belongs in the World/HexGrid subsystem in LoB or in `Scripts/world/` in the layered layout.

- **Navigation Meshes:** In addition to grid-based A*, for complex 3D navigation one can use Godot's `NavigationRegion3D`. However, for a large hex world, bake smaller navmeshes per chunk to avoid whole-map rebuilds [13]. For 3D or mixed 2D/3D, slice the map: each chunk can have its own `NavigationMesh`. Agents switch contexts when moving between chunks, as recommended for performance [13].

- **Performance Optimizations:** To handle many simultaneous paths, we can cache computed paths (especially common start/goal pairs). Spatial partitioning (chunk-level or hierarchical coarse grid) reduces A *search space. If NPCs flock, use steering behaviors with local avoidance rather than full A* every tick. The LoB layout naturally isolates pathfinding to the HexGrid module; in the layered layout, it would be a `Navigation/` or `Pathfinding/` script set.

- **Tick Integration:** Pathfinding runs on demand (when AI seeks a target). To avoid stalling the tick, path requests can be scheduled over multiple ticks or run in a background thread (Godot 4 threads or GDExtension). The path results then inform entity movement on subsequent ticks. Both layouts should implement it similarly; directory grouping does not affect the algorithm.

- **Navigation Agent:** In Godot 4, use `NavigationAgent2D/3D` to follow computed paths. The agent's settings (e.g. max speed, repath rate) live in scene files in `Scenes/` or `world/Chunks`. The directory layout only affects where those scene files are stored (feature folder vs central scenes folder).

Citing Red Blob Games and the Godot hex library underpins these systems [11] [10]. The organization of code (either co-located or centralized) should facilitate adjustments to pathfinding and obstacle data as design evolves.

Migration Guide (Between Layouts)

Converting from one structure to the other involves reorganizing files and updating references:

- **LoB → Layered:** Move scenes from feature folders into `Scenes/`, rename file paths in scripts. For example, `combat/Scenes/CombatArena.tscn` goes to `Scenes/CombatArena.tscn`. Adjust `load()` and `$` paths accordingly. Move scripts into `Scripts/` (e.g. `combat/combat_manager.gd` to `Scripts/combat/CombatManager.gd`). If tests were in `combat/test/`, move them to `Tests/combat/`. Update GUT settings to point at the new `Tests/` directory. Assets from `combat/Assets/` can be consolidated into `Assets/Art/Combat/` or similar. Because Godot scenes reference paths as strings, carefully search for old paths. Using Godot's "Find in Files" helps update references across the project. Git commit in stages (e.g. move code first, test, then update references) can reduce merge issues.

- **Layered → LoB:** Essentially the reverse. Create a folder for each major system (e.g. create `combat/`), move the related scene, script, asset, and test files into it. For example, move `Scenes/CombatArena.tscn` into `combat/Scenes/`. Update references and autoloads accordingly. Remove now-empty central folders if they become unused. Again, updating resource paths in scripts is necessary.

In both cases, maintain the same naming conventions so that scripts' `class_name` and file names stay consistent. Ensure the Godot import paths (res://) still match the file locations. Since Godot uses relative paths, careful refactoring is needed. Ideally, do this before a major feature freeze to minimize disruptions.

Naming Conventions & Guidelines

Adhere to Godot's style for clarity:

- **Files & Classes:** Use **snake_case** for file names and folder names (e.g. `combat_manager.gd`), and **PascalCase** for class names (with `class_name`) and scene names (e.g. `CombatArena.tscn`) [2]. Tests prefix with `test_` (e.g. `test_combat_manager.gd`) [5].
- **Folders:** Lowercase with underscores or PascalCase for major systems (both layouts use this consistently).
- **Scene Nodes:** Name the root node after the scene (e.g. a `Player.tscn` root node named "Player").
- **Resources:** Give `.tres` resources descriptive names (e.g. `sword_item.tres`).
- **Paths:** Prefer Godot's resource paths (`res://`) and avoid hardcoding OS paths. Use `preload("res://path/file.tres")` for constants to have them loaded ahead of time. Godot docs advise preloading static resources to avoid runtime hitches.
- **Constants & Config:** Store game-wide constants in a config file (`.cfg`) or autoload (e.g. `GameConfig.gd`), not as magic numbers in code. This directory structure keeps config in `data/` or `autoload/`.

Following these conventions improves readability and reduces Git merge conflicts. For example, scenes and resources are text files by default, so small changes merge easily. Grouping by object (LoB) vs by type (layered) is a matter of preference, but in all cases use clear, consistent naming.

Example Files (Naming and Minimal Content)

Below are sample file declarations illustrating structure (not full code):

```
# File: player/PlayerController.gd or Scripts/player/PlayerController.gd
extends CharacterBody2D
class_name PlayerController
# ... player movement and input handling ...

# File: player/test/test_player_stats.gd or Tests/player/test_player_stats.gd
extends GutTest
func test_player_level_up():
    var stats = PlayerStats.new()
    assert_eq(stats.level, 1)
    stats.gain_xp(1000)
    assert_eq(stats.level, 2)

# File: world/HexGrid.gd or Scripts/world/HexGrid.gd
# Implements conversion between hex coordinates and screen (using flat-topped
axial coords as per [64†L296-L299]).
var hex_size = Vector2(1, 0.866)
func get_hex_center(axial):
    # returns pixel position from axial coords
    return Vector2(
        hex_size.x * (axial.x + axial.y/2),
        hex_size.y * axial.y
    )

# File: network/Client.gd
extends Node
func _ready():
    var peer = NetworkedMultiplayerENet.new()
    peer.create_client("server.address", 4000)
    Multiplayer.multiplayer.peer = peer

# File: Tools/HexMapEditorPlugin.gd
tool
extends EditorPlugin
# Provides a custom node for designing hex maps in the editor.
```

Each file shows the naming style and purpose. Tests (beginning with `test_`) extend `GutTest` and include assertions. Scenes (not shown) would have matching `.tscn` filenames and root node names (e.g. a `Player.tscn` with root named `Player`).

Performance Expectations

While directory layout itself doesn't impact in-game speed, the organizational choices support performance goals:

- **Load Times:** With chunk streaming, initial load is small (just the first chunk and UI), so startup is fast. As the world grows, memory stays bounded by loaded chunks (e.g. 3×3 around player). Both layouts implement identical streaming, so load times should be comparable. However, loading many scenes at once (monolithic world) is avoided. Benchmark: loading a new chunk (instancing ~100 tiles) should take on the order of milliseconds, keeping gameplay smooth [9] .

- **Memory Use:** Assets are loaded on demand. Unused chunks are freed, so memory usage grows linearly with number of active chunks and cached assets. For example, with 9 chunks (each 100×100 tiles) and moderate asset reuse, memory might be a few hundred MB. Godot's VideoMemoryChart and Profiler can be used to measure this.

- **CPU:** Fixed-tick simulation means CPU usage scales with entity count. An optimized architecture (e.g. updating only active entities) can handle hundreds of units. Pathfinding is on-demand; caching should allow computing thousands of path nodes per second. If needed, heavy tasks can run in threads or be rate-limited. In practice, a few hundred simultaneous pathfinding calls or AI updates should still run at 20+ ticks/sec on modern hardware.

- **Networking:** Data-per-tick depends on how much state changes. With an efficient protocol (delta-compression, priorities), expect a few KB per tick per client. The structure (scripts, message definitions) does not affect runtime networking speed. Instead, using `rpc` and `multiplayer_api` efficiently (Godot 4's MultiplayerAPI) determines performance.

Overall, neither layout has a clear advantage in raw performance; both can be optimized to meet MMO requirements. The key is using fixed ticks, efficient chunking, and asset streaming to keep CPU/memory in check.

Troubleshooting & Best Practices

- **Test Configuration:** If GUT reports "no directories set" [4], ensure you've listed your `test/` or `Tests/` folder in the GUT panel. Remember GUT only runs tests in specified paths.
- **Missing Scenes/Resources:** Moving files can break links. Use Godot's "Find in Files" to update `load("res://old/path")` references after reorganizing. Prefer `preload` or `$NodePath` with exported variables for resources, to get load-time errors instead of silent failures.
- **Pathfinding Errors:** If A *returns no path, check that cube coordinates satisfy* q+r+s=0* [12]. Also ensure collision shapes (if using Navigation) align across chunk boundaries. For grid pathfinding, remember to convert offset coordinates to axial before searching.
- **Chunk Loading Stutter:** If moving between chunks causes a framerate hitch, preload chunks slightly early (e.g. when player enters the last row of the current chunk). VisibleOnScreenEnabler2D (or 3D) nodes can help preload chunk scenes just before they enter view [9].
- **Merge Conflicts:** Scenes (`.tscn`) are text; small edits can still conflict. Adopt conventions like one per file or frequent commits. In LoB, feature folders reduce cross-team conflicts. In layered layout, consider splitting very large scenes into sub-scenes to minimize overlap.
- **Networking Sync:** Ensure client and server use the same tick logic. Desyncs often come from non-deterministic code (floating-point differences, random calls). Mitigate by keeping an authoritative server and validating client inputs.

Sources: We based this structure on Godot's best-practices (e.g. using autoloads for globals [14], GDScript style guide, and official tutorials), community examples (e.g. hex-grid libraries [11] [10]), and general game engineering principles (Locality of Behavior [1], MMO chunking [7], etc.). Our design strives for scalability, clarity, and performance while being designer-friendly and version-control-safe.

---

[1] Locality of behavior - DEV Community
https://dev.to/ralphcone/new-hot-trend-locality-of-behavior-1g9k

[2] [3] [4] [5] Unit testing GDScript with GUT. Who says you can't unit test your game? | by Stephan Bester | Medium
https://stephan-bester.medium.com/unit-testing-gdscript-with-gut-01c11918e12f

[6] GitHub - vnen/godot-tiled-importer: Plugin for Godot Engine to import Tiled Map Editor tilemaps and tilesets
https://github.com/vnen/godot-tiled-importer

[7] Godot 4+ Multiplayer Seamless Open-World Chunks · Issue #8981 · godotengine/godot-docs · GitHub
https://github.com/godotengine/godot-docs/issues/8981

[8] [9] How to build a 2D Open World? - Help - Godot Forum
https://forum.godotengine.org/t/how-to-build-a-2d-open-world/105697

[10] [11] GitHub - romlok/godot-gdhexgrid: A GDScript hexagonal grid implementation for Godot.
https://github.com/romlok/godot-gdhexgrid

[12] Implementation of Hex Grids
https://www.redblobgames.com/grids/hexagons/implementation.html

[13] How to handle new NavigationRegion3D on large maps? : r/godot
https://www.reddit.com/r/godot/comments/10zzb65/how_to_handle_new_navigationregion3d_on_large_maps/

[14] Best Practices for Godot Project Structure and GDScript? : r/godot
https://www.reddit.com/r/godot/comments/1g5isp9/best_practices_for_godot_project_structure_and/