ChatGPT

Locality-of-Behavior (LoB) Directory Layout

```
res://
├── addons/                 # Third-party plugins (e.g. GUT)
│   └── gut/                # Godot Unit Test plugin
│       └── ...
├── features/               # Features/modules, co-locating code, assets, and
tests
│   ├── world/              # Example feature: world generation and streaming
│   │   ├── world.tscn      # Scene for the world (chunk manager)
│   │   ├── world.gd        # Controller script for world logic
│   │   ├── world_data.tres  # Resource for world settings (hex size, etc.)
│   │   ├── world_assets/    # Assets used exclusively by this world feature
│   │   │   ├── hex_tiles.png
│   │   │   └── terrain_material.tres
│   │   └── tests/          # Unit tests for this feature
│   │       ├── test_world_chunk_loading.gd
│   │       └── test_hex_grid_navigation.gd
│   ├── player/             # Feature: player entity
│   │   ├── player.tscn
│   │   ├── player.gd
│   │   ├── player_assets/
│   │   │   ├── player_model.glb
│   │   │   └── player_textures.png
│   │   └── tests/
│   │       └── test_player_movement.gd
│   ├── pathfinding/        # Feature: hex grid and pathfinding system
│   │   ├── hex_grid.gd     # Hex grid coordinate system and navigation
│   │   ├── pathfinder.gd   # A* pathfinding logic on hex grid
│   │   └── tests/
│   │       └── test_pathfinding.gd
│   ├── combat/             # Feature: combat mechanics
│   │   ├── combat.tscn     # Scene for combat encounters (if needed)
│   │   ├── combat.gd       # Combat logic (attacks, hit resolution)
│   │   ├── combat_assets/
│   │   │   └── attack_effects.png
│   │   └── tests/
│   │       └── test_damage_calculation.gd
│   └── ...                 # More features (inventory, AI, UI, etc.), each
with its own code, assets, tests
├── global/                 # Cross-cutting code (systems and singletons)
│   ├── networking/         # Network layer (network managers, RPC handlers)
```

```
│   │   ├── network_manager.gd  # Manages ENet/WebSocket connections
│   │   ├── message_definitions.tres
│   │   └── ...
│   ├── simulation/          # Tick-based simulation core
│   │   ├── tick_manager.gd  # Fixed-timestep driver
│   │   └── ...
│   └── common/              # Shared utilities and resources
│       ├── util.gd         # Miscellaneous helper functions
│       ├── data_models.gd   # Shared DataResources or definitions
│       └── ...
├── tools/                   # Editor plugins and external tooling
│   ├── map_editor/          # Custom map editor plugin (e.g. hex pattern
design)
│   │   ├── map_editor_plugin.gd
│   │   └── plugin.cfg
│   ├── chunk_loader.gd      # Standalone script or scene to preprocess chunk
data
│   └── ...
├── tests/                   # (Optional) Global test suites or integration
tests
│   ├── integration/
│   └── utility_tests/
├── project.godot            # Godot project file
└── README.md
```

- **Features/**: Under LoB, each major feature or game system (world streaming, player, combat, etc.) has its own folder. *Code, scenes, assets, and tests that implement that feature live together*, making behavior self-contained [1] [2].
- **Feature subfolders** (e.g. `world/`, `player/`, `pathfinding/`): Each contains a main scene file (`.tscn`), its controller script(s) (`.gd`), any feature-specific resource files, and an `assets/` subfolder for art/audio used only by that feature. Names use **snake_case** for files and folders as per Godot conventions [3].
- **Tests**: Following GUT guidelines [4], unit tests for a feature are placed in a `tests/` subfolder of that feature. For example, `features/world/tests/test_world_chunk_loading.gd` would contain tests extending `GutTest`. This co-location keeps tests near the code they verify, satisfying LoB. GUT's quickstart recommends separating unit and integration tests (e.g. `test/unit` vs `test/integration`), but here we localize them by feature for modularity [4].
- **addons/**: Holds third-party plugins. For example, GUT is installed here (`addons/gut/`) so the test runner is available.
- **global/**: Contains cross-cutting systems (network manager, tick manager, utility scripts) and autoload singletons. For example, `simulation/tick_manager.gd` is an autoloaded script that drives a fixed timestep simulation loop for the whole game.
- **tools/**: Contains editor scripts and external utilities. Custom editors (e.g. a hex-map editor plugin), import scripts (e.g. a tool to bake raw heightmap data into chunks), or level preprocessors go here.
- **Naming Conventions**: Use *snake_case* for directories and GDScript file names, *PascalCase* for any C# scripts. Scene files (`.tscn`) and resource files (`.tres`) are typically lower_snake (as they are files).

Each controller script/class should be named after its scene and scene root node [5] (e.g. `world.tscn` with a `world.gd` script attached to root node "World"). This makes behavior obvious from the code location [1] [5].

GUT Testing Integration

Unit and integration tests live alongside feature code: e.g. `features/player/tests/test_movement.gd`. Each test script extends `GutTest` [6] and is auto-discovered by GUT when placed in the project. Tests must be named with the `test_` prefix or configured accordingly. You should configure GUT (in the editor panel) to include all `*/tests/` directories. Optionally, have top-level `tests/unit/` and `tests/integration/` dirs for broad tests (as GUT recommends) [4]; these can import features from their folders. This structure keeps tests near code for quick understanding (LoB principle) while allowing global test runs.

Asset Pipeline and Chunk Streaming

A key workflow is building the world in chunks. The **map design pipeline** might use:

- **External tools**: Designers use a hex-map editor (Tiled, or a custom Godot tool in `tools/map_editor/`) to lay out terrain, placing hex-tile IDs on a grid. These raw maps (e.g. .tmx or JSON) are kept outside the `game/` folder (per Godot best practice) to avoid exporting source assets [7].
- **Import**: A Godot import script or autoload in `tools/` converts the raw map into Godot resources. For example, an editor plugin reads the map and splits it into region-based scenes (`scenes/levels/hex_chunk_001.tscn`, etc.), each with a TileMap or Mesh. These "chunk" scenes are saved in `scenes/levels/`.
- **Runtime streaming**: The world root node (controlled by `world.gd`) holds an instance of `ChunkManager`. As the player moves, the ChunkManager determines which adjacent chunks should be loaded (e.g. within a 3×3 grid around the player). It uses `PackedScene.instance()` or `ResourceLoader.load_interactive()` to load chunk scenes asynchronously (in a background thread) [8]. Chunks moving out of range are removed from the SceneTree and freed to save memory. The **background loading** approach (supported by Godot) ensures smooth transitions: "A background game thread could be used to manage preloading and unloading nearby scenes asynchronously based on player movement" [8]. In code, chunk scenes live under `scenes/levels/` and are instanced by path or preload.
- **LODs**: You can supply low-detail and high-detail chunk variants. Faraway chunks can be swapped with lower-detail scenes (or simply have fewer objects) and replaced with high-detail when the player is nearby [9].
- **Memory**: Each chunk scene is a moderate-size node hierarchy. Unloading is done via `queue_free()` when out of range. Use Godot's `ResourceLoader` to free unused resources. Keep dynamic data out of scenes if possible (use Resources for terrain height, etc., so it can be shared or reused). For very large worlds, consider manually freeing unused scripts and textures, or use Godot 4's **reference-counted resources** (like `ImageTexture.unload()`).

Tick-Based Simulation

An MMO needs a deterministic, fixed-timestep simulation. In this layout, a single **TickManager** (autoloaded from `global/simulation/tick_manager.gd`) drives the game loop at, say, 20 or 30 ticks per second. All game logic that needs to be deterministic (NPC AI, physics, etc.) is updated by the TickManager rather than relying on frame rate. Entities register with TickManager and implement a `tick(delta)` function. This central manager can also emit a "world_tick" signal each step. Input and rendering still happen on `_process()`, but authoritative game state changes on ticks. A possible organization: - **TickManager.gd**: emits `world_tick` every fixed interval (set by project settings or custom timer).
- Each entity node (in `features/`) has a script that listens to `world_tick` and updates position/logic.
- On the server (authoritative), the TickManager runs and sends new states to clients after each tick. Clients interpolate between states for smoothness.

This enforces synchronous simulation. All tick-related code (scheduling events, processing queued commands, advancing game time) belongs under `global/simulation/` or within the relevant feature scripts but triggered by the TickManager.

Hex Grid Navigation and Multi-Layer Movement

The **hex grid system** is typically encapsulated in a feature (e.g. `features/pathfinding/hex_grid.gd`). This code handles coordinate conversions (axial/cube coords) and neighbor offsets. For example, the GDHexGrid library shows how to map flat-topped hexes and even includes A *pathfinding on a hex grid* [10]. *We adopt a similar approach: our* `HexGrid` *class has methods to get adjacent hexes, compute distances, and convert hex coordinates to world positions. Pathfinding uses A* on a 2D grid of hex cells (flagging obstacles), as outlined in Red Blob Games guides [10].

Multi-layer movement (e.g. ground vs. mountain vs. air): We treat layers as separate path graphs or use Godot's Navigation2D with layers. In Godot 4, you can assign different tilemaps or navigation meshes to navigation layers (like physics layers for agents) to constrain movement [11]. For example, flying units might use a "sky" NavMap, while ground units use a "ground" NavMap. Godot's `NavigationRegion` and `NavigationLink` (in 3D) or TileMap navigation (in 2D) support multiple layers. The navigation logic resides in the same feature (pathfinding) – e.g. a `NavManager` script that queries `NavigationServer2D` with specific layer masks. Godot docs note that tilemap navigation layers needed a PR to work correctly, essentially placing each layer on separate navigation maps [11]. In practice, we might bake multiple navmeshes (one per layer) and switch the agent's `map` or `layer_mask` as needed.

Streaming and Memory Management

The LoB structure encourages each feature to manage its own memory. For example, the `world` feature loads/unloads chunk scenes and is responsible for freeing them. The `pathfinding` feature may load path mesh data only when needed. General strategies: - **Chunk pooling**: Instead of freeing, you could keep a pool of recently used chunks and reuse them to reduce allocation cost.
- **View frustum culling**: Non-active areas of world remain unloaded.
- **Godot Resources**: Use `@export var preload_scene: PackedScene` or `.tres` resources for lightweight data.
- **Threading**: Long operations (like baking navmesh or loading large scenes) run via `Thread` or `background_loader` to avoid hitches, as suggested by Godot background loading docs.

Memory profiling and leak testing are done via GUT's leak testing features [12] or Godot's built-in Profiler. Each feature's tests can include leak checks after loading/unloading.

Networking (Client-Server)

In LoB layout, network code (RPCs, sync) often sits in the feature or global scripts that are network-aware. For instance: - **NetworkManager** (`global/networking/network_manager.gd`): A singleton that sets up the `MultiplayerPeer` (ENet or WebSocket) and handles connection/disconnection. It may use Godot 4's `MultiplayerSynchronizer` nodes inside scenes to replicate state [13].
- **Scenes**: Networked scenes use `MultiplayerSpawner` and `MultiplayerSynchronizer` nodes as shown in Godot's official guide [13]. For example, a player character scene might have a `MultiplayerSynchronizer` to sync its position and health. The `features/player/player.gd` script can check `is_multiplayer_authority()` for server-side logic.
- **Separation**: On a dedicated server, many visuals/scripts (UI, effects) are omitted. The same project can run as headless for the server. Code that only runs on server (AI simulation) is behind `if multiplayer.is_server():` checks.

Data flow: client inputs (move, actions) are sent to server (via RPC or custom packets), server updates authoritative state on ticks, and syncs back to clients. Network messages (e.g. chat, action commands) use custom Resource data (in `global/networking/message_definitions.tres`) so they can be versioned and extended.

Godot's built-in ENet can handle up to **~4096 simultaneous clients** (practical limit imposed by the library) [14]. For a large MMO, this implies either sharding across multiple servers or using a lower-level/more scalable networking stack. In our design, the LoB structure still allows hooking in an external server: e.g. using Godot's `WebSocketClient` to connect to a Node.js backend, as done by community projects [15].

Industry-Standard Large-Scale MMO Architecture Layout

```
res://
├── addons/                  # Third-party plugins (GUT, etc.)
│   └── gut/
├── assets/                  # All raw assets (imported into Godot)
│   ├── textures/
│   ├── models/
│   ├── audio/
│   ├── fonts/
│   └── data/               # JSON/CSV config, skill/item data, etc.
├── scenes/                  # All Godot scenes (.tscn files)
│   ├── levels/             # Level and chunk scenes
│   │   ├── chunk_001.tscn
│   │   ├── chunk_002.tscn
│   │   └── ...
```

```
│   ├── entities/           # Scene prefabs for entities
│   │   ├── player/
│   │   │   ├── player_body.tscn
│   │   │   └── ...
│   │   ├── npc/
│   │   ├── items/
│   │   └── ...
│   ├── ui/                 # UI screens and windows
│   │   ├── main_menu.tscn
│   │   ├── hud.tscn
│   │   └── ...
│   ├── world/              # Global scenes (game root, server-only, etc.)
│   │   ├── server_root.tscn
│   │   └── client_root.tscn
│   └── ...
├── scripts/                 # GDScript (or C#) code, organized by category
│   ├── ai/                # AI behaviors and state machines
│   ├── gameplay/          # Core mechanics (combat, inventory, skills)
│   ├── navigation/        # Pathfinding and navmesh scripts
│   │   └── hex_nav.gd     # Hex grid navigation helper
│   ├── networking/        # Low-level network code
│   │   ├── client.gd      # Client-side net manager
│   │   ├── server.gd      # Server-side net manager (headless)
│   │   └── protocol.gd    # Message serialization/deserialization
│   ├── systems/           # ECS-like or manager systems (if used)
│   ├── ui/                # UI controllers
│   └── util/              # Utility scripts (logger, math, etc.)
├── resources/              # Custom resources (.tres) and configs
│   ├── player_stats.tres
│   ├── world_settings.tres
│   └── ...
├── tests/                  # Global test suites
│   ├── unit/
│   ├── integration/
│   └── scenarios/
├── tools/                  # Build scripts, level converters, external tools
│   ├── map_importer.py
│   ├── chunk_baker.gd
│   └── ...
├── project.godot
└── README.md
```

- **Assets/**: Organized by type (textures, models, audio) for easy reuse and processing. All imported Godot assets reside here. Raw source files (e.g. 3D models `.blend`, concept art) are kept outside (e.g. in a separate `art/` folder) to avoid exporting non-game files [7].

- **Scenes/**: Types of scenes are grouped: level/chunk scenes under `levels/`, entity prefabs under `entities/`, UI under `ui/`, etc. This mirrors many large projects and the GitHub advice ("scene-based assets folder") [16]. For example, `scenes/levels/chunk_001.tscn` includes its TileMap and any local resources. Scenes that are reused across the game (like an `enemy_type_a.tscn` used in many levels) live in `entities/`. Global root scenes (`server_root.tscn` vs `client_root.tscn`) separate server-only logic.
- **Scripts/**: Pure code is separated by system or feature **type** (not by game feature). For example, all AI scripts go under `scripts/ai/`, all navigation/pathfinding code in `scripts/navigation/`. This is a common large-project pattern: it aids IDE navigation and reuse. However, it means one may have to open multiple folders to see all parts of a feature's behavior.
- **Resources/**: Any custom `.tres` resources (data-only files) such as skill definitions, item stats, or world constants. These are kept in one place, rather than spread in feature folders. For example, `world_settings.tres` might define hex spacing, chunk size, tick rate, etc.
- **Global Network and Simulation**: Here, network and simulation code are explicitly separated: e.g. `scripts/networking/client.gd` and `scripts/networking/server.gd` drive the two roles. This makes it clear what runs on each.
- **Tests/**: All tests are outside the main code (unlike LoB). GUT is configured to scan `tests/unit` and `tests/integration`, which reference code from anywhere. This makes running full-suite tests simple, at the cost of jumping around to find code under test.
- **Naming**: Same conventions: snake_case for files/folders (PascalCase for C#). Controller scripts are named after the scenes they govern [5], but here scene files themselves are in separate folders. For example, `entities/player/player_body.tscn` with `scripts/gameplay/player_body.gd` controlling it.
- **Autoloads & Singletons**: May have a top-level `scripts` or `autoloads` folder. For instance, `scripts/systems/logging.gd` as an autoload (Logger), or `scripts/systems/tick_system.gd` managing ticks. These are often in a `systems` or `util` folder.

GUT Testing Structure

Following industry practice, place tests in dedicated `tests/` directories: e.g. `tests/unit/test_hexgrid.gd` and `tests/integration/test_world_load.gd`. Configure GUT to include these directories (as per its quickstart guide [4]). Tests import and instantiate scenes or scripts from the main project structure, rather than being co-located. This makes test execution language-agnostic (all in one place) and easier to run via CI, at the expense of scattering context.

Asset Pipeline Workflow

Asset workflow is more formalized:

- **Content Creation**: Artists save raw files (e.g. `art/terrain.eps`, `art/models/*.blend`) in a source repo outside the Godot project.
- **Import and Setup**: Use scripts (possibly in `tools/`) to batch-import or convert assets into `assets/` (textures, models). For example, a Python or GDScript tool (`tools/map_importer.gd`) may read Tiled `.tmx` hex maps and generate `.tscn` chunk scenes into `scenes/levels/`.
- **Chunk Preparation**: An external or in-editor "chunk baker" tool may preprocess navmeshes and object placement for each chunk. Processed chunks (PackedScenes) live in `scenes/levels/`.

- **Runtime Streaming**: The game's world manager (e.g. in `scenes/world/world_root.tscn` with `scripts/gameplay/world_manager.gd`) loads chunk scenes around the player. Code for asynchronous loading (using `Thread` or `ResourceLoader`) is found in `scripts/gameplay/` or `scripts/systems/`.

In this layout, assets are decoupled from specific features, so the pipeline may use naming conventions (prefix assets with scene names [17]) to find relevant assets. But since scenes have their own directories, each chunk's local assets can still live in `assets/levels/` or inline.

Tick-Based Simulation Organization

Tick logic is typically implemented in a "system" script (e.g. `scripts/systems/simulation.gd`) or as an autoload. For example, a `FixedTickSystem` autoload drives world ticks. All game systems subscribe to it. Unlike LoB, there isn't a single feature folder; instead, tick handling functions are in scripts under `scripts/systems/` or relevant subsystems. For instance, `scripts/systems/ai_manager.gd` may be called by the tick system to update NPCs. The fixed timestep is configured in Project Settings or in code. The important part is that all ticked code is organized under `scripts/`, not spread next to scenes.

Hex Grid and Pathfinding

The hex grid code lives in `scripts/navigation/` (e.g. `hex_grid.gd`, `pathfinder.gd`). The logic to convert coordinate systems and run A* is shared by all features that need pathfinding. For example, an NPC's movement code (in `scripts/ai/`) would call `HexGrid.get_neighbor_coords()` or `Pathfinder.find_path(start, goal)`. Any pathfinding settings (like movement costs) might be stored in `resources/pathfinding_settings.tres`. Multi-layer navigation (bridges, tunnels) is handled by loading multiple `NavigationPolygon`s into `NavigationRegion2D` nodes in scene (e.g. in `scenes/world/chunk_001.tscn`) and using Godot's Navigation layers to select appropriate graphs.

Streaming and Memory Management

Chunk streaming code (e.g. a `ChunkLoader` script) is placed under `scripts/gameplay/` or `scripts/systems/`. It monitors the player's position (via a global autoload or the main scene) and loads/unloads from `scenes/levels/`. Memory is freed by calling `queue_free()` on old chunk instances. Global pooling systems (also under `scripts/systems/`) might reuse node instances for frequently spawned objects (bullets, enemies). Large static resources (heightmaps, biomes) are stored as Godot Resources (`.tres`) under `resources/`, which can be `preload`ed or `load()`d on demand; unloading them is automatic when no scene references them.

Networking and Entities

Networked scenes and entities follow Godot's recommended pattern [13]. For example, the `scenes/world/client_root.tscn` contains a `MultiplayerSpawner` node configured to spawn player and world scenes. Each spawned scene has a `MultiplayerSynchronizer` for its relevant properties. Code controlling networking (e.g. on join, on spawn) resides under `scripts/networking/`. Separation is explicit: e.g. `scripts/networking/server.gd` runs on the headless server and has authority, while `client.gd` on clients applies updates and sends inputs. Shared code (like the data model for an entity) may be in `scripts/util/` or a `resources/` file so both sides use the same definitions.

If using an external (e.g. TypeScript) server, the Godot project's network layer (in `scripts/networking/`) would use `WebSocketClient` or `WebRTCMultiplayer` to talk to it. The separation is clearer: Godot is purely client, and server code (not in this project) runs in Node/TS.

Comparative Analysis

- **Organization by Feature vs. by Type**: The LoB layout groups everything (scenes, scripts, assets, tests) for a feature in one place [1] [16]. The industry layout separates by asset type or system. LoB is more self-contained (good for small teams or feature owners) [1], while the industry layout can be more scalable for large teams specializing in art, code, etc. For example, in LoB you might have `features/combat/` with its own scripts and assets, whereas in the industry layout combat scripts live in `scripts/gameplay/combat/` and combat assets in `assets/ui/` or similar.
- **Ease of Navigation**: LoB makes it easy to find all parts of a feature (just look under one folder), reducing "jumping around" to piece behavior [1]. The type-based approach requires knowing where code lives globally (e.g. AI code under `scripts/ai/`). However, type-based can make it easier to reuse code and track usage of assets across features (all sounds in one place, etc.).
- **Modularity and Dependencies**: LoB encourages minimal external dependencies: each feature relies mainly on its own resources. This can improve cohesion but sometimes leads to duplication (two features might import the same texture into their own asset folders). The industry layout naturally shares resources: all features draw from the same `assets/` pool and the same code libraries in `scripts/`. This reduces redundancy but can increase coupling (a change in a global script affects many features).
- **Testing**: LoB puts tests next to code, making developers more likely to write tests (they see them) and simplifying understanding. Industry approach centralizes tests; running a subset of tests per feature is harder, but global QA is streamlined. GUT supports both styles (tests can call into code anywhere) [4].
- **Pipeline and Collaboration**: For LoB, artists/designers might need to navigate into feature folders to add assets, which can get cluttered if not managed carefully. Industry layout's separation of `assets/` and `scenes/` can be clearer for roles (artists put art in `assets/`, designers edit `scenes/levels/`). On the other hand, LoB's feature folders can contain feature-specific art pipelines (e.g. a custom map editor for the world feature in `features/world/`).

Both structures can support an MMO, but differ in emphasis. LoB is often easier for initial development and small teams (everything for "World" is in one place), while the industry-type layout is how many large projects scale (shared systems, clear separation of code vs content, and complex build pipelines).

Best Practices and Conventions

- **Naming**: Use `snake_case` for all file and folder names (Godot standard) [3]. For example: `hex_grid.gd`, `player_model.glb`, `world_settings.tres`. Use `PascalCase` for class names (via `class_name`) and C# scripts. Name scene files after their root node and controller (e.g. `player.tscn` with root node "Player" and script `player.gd`).
- **File Placement**: Keep GUI scenes in a `ui/` folder, levels in `scenes/levels/` or feature-level folders, etc. Autoload singletons (e.g. `TickManager.gd`) can reside in a top-level `scripts/systems/` or in an `autoloads/` folder. Keep `.gd` scripts close to what they control, or in a

dedicated `scripts/` area if you prefer. Use descriptive folder names (`world/`, `player/`, `ui/`) rather than generic ones.

- **Scenes and Scripts**: Each scene has one "controller" script attached to its root, named the same as the scene [5]. Additional scripts (for child nodes) use descriptive names and are kept minimal. Follow the recommended GDScript ordering (signals, enums, consts, vars, _init, _ready, etc.). Group related scripts (AI states, utilities) together in their subfolders.
- **Modular Design**: Design each scene to be as self-contained as possible [18]. Inject external dependencies via signals or setter methods. Use custom Resource classes to store data (e.g. stats or tile definitions) so scenes load data without hardcoding. Use Godot's composition (SceneTree, instancing) instead of deep inheritance to avoid rigid coupling.
- **Version Control**: Follow Godot docs: ignore `.import/` folders and `.fscache` files [19]. Keep project-wide config (`project.godot`) at root. Use consistent case (all lowercase) to avoid case-sensitivity issues [3].
- **Naming Assets**: For scene-specific assets, some developers prefix asset files with the scene name for searchability (e.g. `player_idle.png` for the Player scene) [17]. Keep shared assets (like common shaders or fonts) in global folders.
- **Modular Pipelines**: If you have tools (e.g. a map generator), integrate them as either editor plugins (`addons/`) or external scripts (`tools/`). Document their usage. For example, you might include a Python script `tools/convert_hex_map.py` and note in README that designers should run it to generate Godot chunks.
- **Networking**: Follow Godot's high-level multiplayer best practices [13]. All RPC methods should be `@rpc` annotated. Use authoritative server logic and reject client discrepancies. Avoid calling `change_scene()` in multiplayer; instead use `MultiplayerSpawner` to switch levels so all clients sync [20].
- **Performance**: Aim to keep tick/update logic fast. Profile using Godot's built-in monitors. Avoid heavy processing on clients for non-visible objects. LOD and culling are essential for large worlds. On the server, limit the number of active entities per tick. Typical targets: sub-0.01s per tick for 1000 active objects on a modern CPU. Use Godot's Profiling and GUT's leak tests to check memory.
- **File Examples**: Use clear names like `hex_grid.gd`, `tick_manager.gd`, `multiplayer_manager.gd`, `chunk_loader.gd`, `inventory_panel.tscn`, `enemy_fsm.gd`. For data resources: `unit_stats.tres`, `world_geometry.csv`.
- **Error Handling**: Place error-checks when loading resources (e.g. verify `ResourceLoader.load()` does not return `null`). In multiplayer, gracefully handle disconnects (use Godot's signals). Log important events (use a custom Logger singleton if needed).
- **Troubleshooting Tips**: If chunks fail to load, check resource paths in the scene files. Broken links in scenes often break streaming – use "Open in FileSystem" in editor to find missing files. For networking issues, ensure peers use the same networked scenes (same path) and that `MultiplayerSynchronizer` is set on identical nodes. Use Godot's **Verbose Print** mode to see RPC calls. In GUT tests, if a test hangs, it may be waiting on a signal; consider using `await` or timeouts.
- **Performance Benchmarks**: As an order-of-magnitude guide, Godot 4's ENet can handle thousands of connections [14], but practical player counts per server will be lower depending on message rate. For tick processing, test how many entities you can update per frame – a simple test could be instantiating 10,000 moving sprites and measuring FPS. Optimize until performance is acceptable (e.g. >30 FPS). Use LOD, pooling, and threading to meet goals.

Transition Plan Between Layouts

If moving from one structure to another, use an incremental refactor:

1. **Version Control**: Ensure the project is under git or similar. Create a new branch for restructuring.
2. **Move Directories**: For LoB→type layout, create the new top-level folders (`scenes/`, `scripts/`, `assets/`, etc.). Move files accordingly. Update any hardcoded paths in code (`load("res://...")`). For type→LoB, group related scenes/scripts under a common feature folder. Godot 4 allows moving files in the editor (FileSystem dock) which updates internal references.
3. **Fix References**: After moving, open the project in Godot and re-save any scene that lost links. Godot will auto-fix moved script paths if possible, but double-check `@export(String)` resource paths. Run the game to test.
4. **Tests**: Move test files to new locations and update GUT settings. Run all tests to catch missing imports.
5. **Stubs/Deprecations**: Maintain old paths as aliases if needed (e.g. a stub script at old path that re-directs to new location) during transition. Remove them once everything is stable.
6. **Communicate**: Update team on changes (README, docs). If multiple developers, ensure everyone updates their local workspace.

Given that both structures can coexist for a while, one could adopt a hybrid: group by feature, but still have a central `assets/` and `scripts/`. Eventually, pick one consistent style.

Recommendations: Tooling & Server Architecture

- **Custom Map Tools**: Integrate editor plugins for level design. For example, a *Hex Map Editor* plugin under `addons/` can allow painting hex biomes or placing prefabs in-editor, outputting a Godot TileMap or custom resource. Alternatively, support external tools (like Tiled): write import scripts to convert `.tmx` into Godot scenes. Document the pipeline (e.g. in README: "Run `map_converter.py` to import `.tmx` files into `scenes/levels/`").
- **Server-Side Separation**: For true MMO scale, offload the game simulation to a backend server architecture. You can still use Godot headless as a server (great for prototyping), but for massive scale use a dedicated solution. One option is a **TypeScript/Node.js** server (e.g. using WebSocket or custom TCP protocol). The Godot client uses `WebSocketClient` or `WebRTCMultiplayer` to communicate. We saw an example where a developer built a Godot client and a TS server connected by WebSockets [15]. With TS, you can leverage web tech (load balancers, database integrations).
- **External Server Patterns**: Implement a lobby/matchmaking system on the TS side, then spawn instances (perhaps still Godot headless servers) for each game session. Or write most logic in TS and have Godot as a lightweight display client. Use message schemas (JSON or binary) and keep them versioned. Use SSL/TLS if needed for security (Godot supports SSL certificates [21]).
- **Authoritative Design**: Regardless of server language, keep server authoritative. Clients send input commands; server computes physics/simulation. For example, clients send `move_unit(direction)` and server replies with `unit_position` updates. Do not trust client state.
- **Scaling**: Use multiple server instances behind a gateway. If using ENet, consider the 4096 limit [14] as the max per server. Otherwise, implement sharding or regional servers.

- **Monitoring**: Integrate logging (Moonwards built a custom logger addon [22]). Collect metrics (tick time, network lag) to guide scaling.

In summary, **both layouts can support all requirements**. The LoB design emphasizes ease of understanding and modular development [1], while the industry-style layout emphasizes clear boundaries and reuse. The final choice depends on team size and preferences; the above guide provides a blueprint for each.

---

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

[2] [7] [19] Project organization — Godot Engine latest documentation
https://docs.huihoo.com/godotengine/godot-docs/godot/tutorials/engine/project_organization.html

[3] [5] [16] [17] [18] GitHub - abmarnie/godot-architecture-organization-advice: Advice for architecting and organizing Godot projects.
https://github.com/abmarnie/godot-architecture-organization-advice

[4] [6] [12] Quick Start — GUT 9.3.1 documentation
https://gut.readthedocs.io/en/9.3.1/Quick-Start.html

[8] [9] How would you approach an seamless open-world map in Godot 4? : r/godot
https://www.reddit.com/r/godot/comments/198tsu8/how_would_you_approach_an_seamless_openworld_map/

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

[11] Godot 4 2D. Only first tilemap navigation layers works. : r/godot
https://www.reddit.com/r/godot/comments/11z9ntk/godot_4_2d_only_first_tilemap_navigation_layers/

[13] [20] Multiplayer in Godot 4.0: Scene Replication – Godot Engine
https://godotengine.org/article/multiplayer-in-godot-4-0-scene-replication/

[14] Can I start worrying about the 4K user limit on Godot Multiplayer already? : r/godot
https://www.reddit.com/r/godot/comments/1jobzmg/can_i_start_worrying_about_the_4k_user_limit_on/

[15] I have been working on a simple online multiplayer game, with a Godot client and a Typescript server, all connected by websockets : r/godot
https://www.reddit.com/r/godot/comments/tpudwy/i_have_been_working_on_a_simple_online/

[21] Is there any viable backend for Godot? - Help
https://forum.godotengine.org/t/is-there-any-viable-backend-for-godot/41218

[22] Guest post - “Small Team, Big Project”: Building Moonwards – Godot Engine
https://godotengine.org/article/guest-post-small-team-big-project-building-moonwards/