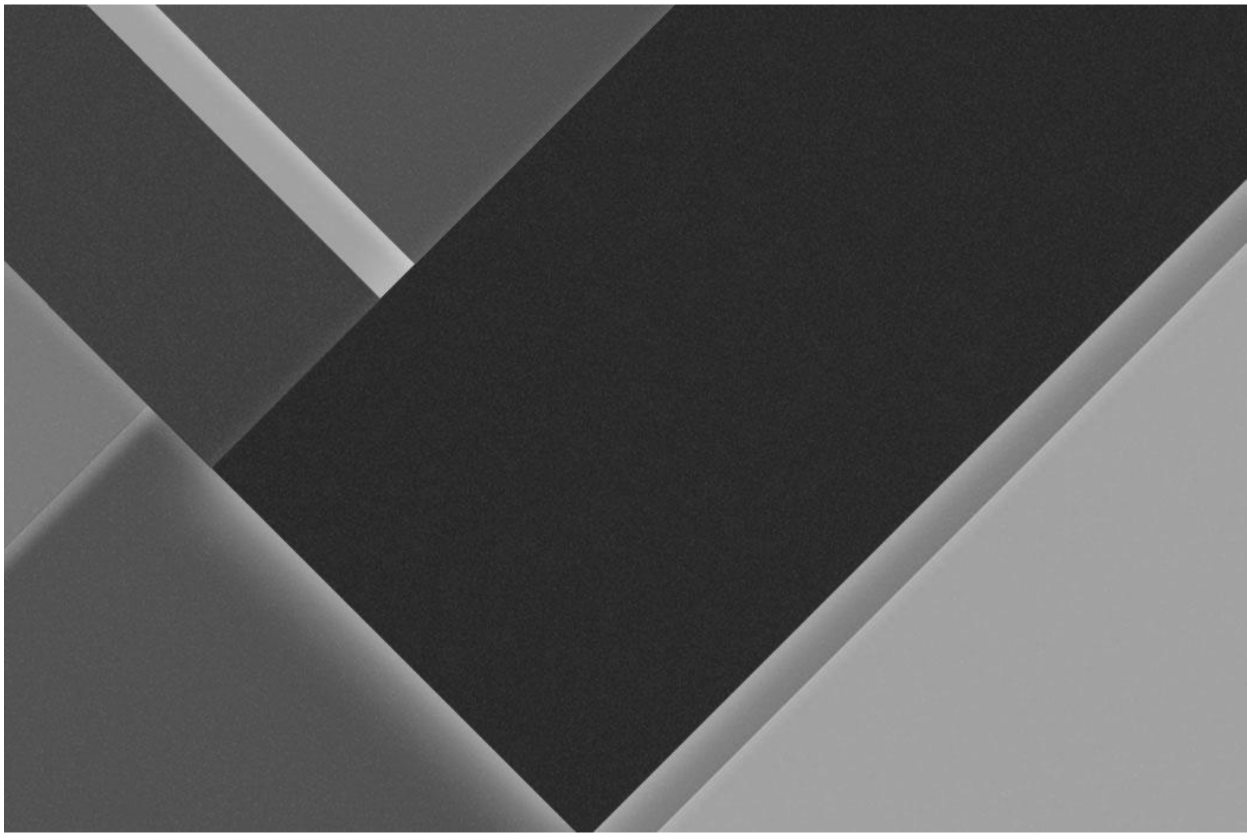


Project Design & Specifications

11.18.2025

"NecronomiCore: The Elder Bloom"

Alexandra Curry

Landon Coonrod

Noah Valdez

Overview

This document serves as a comprehensive design and planning guide for our custom C++ Godot 2D module. It outlines the purpose, functionality, and technical architecture of the module, details the core design and gameplay elements, and establishes our implementation strategy, development timeline, and team workflow. It also defines the roles and responsibilities of each team member, the tools and processes we will use throughout development, and the risks we anticipate.

Roles

- **Alexandra Curry**
  - Sprite Design/Animator/Tileset Design
    - *Responsible for creating all character sprites, animations, environmental tilesets, and visual assets.*
  - Story Writer
    - *Will contribute to writing the game's overarching narrative and lore.*
  - Custom Module: AI-Powered Emotion Dialog
    - *Will develop a custom module that assigns personality traits to NPCs and generates emotionally consistent dialogue using AI.*
- **Landon Coonrod**
  - Story Writer
    - *Will assist in developing the main storyline, environmental lore, and character narratives.*
  - Custom Module: Random Item Generation
    - *Will implement a custom module that procedurally generates items, including stats, descriptions, and rarity classifications.*
- **Noah Valdez**
  - Level Design

        - *Will design and construct levels using custom assets, ensuring proper layout, pacing, and collision physics.*
    - Story Writer
        - *Will contribute to writing narrative elements and worldbuilding.*
    - Custom Module: Random Roll Generation
        - *Will create a custom module that generates randomized rolls for in-game gambling mechanics and chance-based systems.*

General Game Concept

Our project is a top-down 2D dungeon crawler that blends Lovecraftian horror with an overgrown, decaying dungeon environment. Core gameplay focuses on careful exploration, resource management, environmental hazards, and high-tension encounters with otherworldly creatures. The experience is enhanced by our custom module: a seamless integration of OpenAI's ChatGPT that dynamically generates random items, random rolls for in-game gambling, in-game lore, reactive NPC dialogue, cryptic environmental messages, and evolving narrative elements based on player actions. This creates a personalized, unpredictable atmosphere that reinforces the game's themes of madness, mystery, and the unknown. Designed for PC players who enjoy atmospheric dungeon crawlers, narrative-rich indie titles, and cosmic-horror experiences, our game will use fully custom tilemaps, hand-drawn sprites, and bespoke animations to create a dense, overgrown aesthetic where nature has reclaimed forgotten ruins. The visual direction leans into muted palettes, glowing fungal tones, and unsettling organic textures to emphasize a world where rot, ruin, and cosmic dread intertwine.

Module Purpose

The purpose of this custom module is to integrate OpenAI's ChatGPT directly into the Godot engine, enabling developers to enhance and expand player experiences through intelligent, dynamic, and context-aware interactions. By providing a native C++ interface with a simple, well-structured C# API, the module allows games to leverage advanced language models for special features. This integration removes the need for external scripts or manual API handling, giving developers a seamless and powerful tool for creating richer, more immersive, and highly responsive gameplay experiences that evolve in real time based on player behavior and in-game events.

Anticipated Technical Challenges

I. Integrating AI API

A major challenge will be integrating ChatGPT into the game in a stable and reliable way. Player interactions, NPC dialogue, random luck rolls, and loot outcomes will depend on AI-generated results rather than predefined code paths. This requires careful handling of API requests, error cases, response timing, and maintaining consistent gameplay even when external calls are involved.

II. Level Creation & Connectivity

Designing the game's levels will be time-intensive and is likely to occur later in the development cycle, after core systems and assets are in place. Because we may encounter new interactions between mechanics, environments, and AI systems for the first time during this stage, debugging and refining level behavior will present additional challenges.

III. Integrating Team Contributions Into a Unified System

Each team member is developing a different component, whether it be art assets, narrative content, custom modules, or level design. Bringing all these pieces together into cohesive, functional gameplay will be both time-consuming and technically demanding. Ensuring that all assets, scripts, and systems integrate smoothly will require consistent communication and iterative testing.

IV. Ensuring Timely Completion

Project success depends on every team member completing their individual tasks on schedule. Managing dependencies between tasks—where one member's progress directly enables another's work—will require clear communication and efficient coordination. Delays in any area may impact other parts of the workflow, so staying aligned as a team will be essential to meeting milestones.

Module Functionality

The custom module exposes a unified C++ service layer for OpenAI features and game-side randomness, with a simple C# API for use inside Godot scenes and scripts. Its core responsibilities are:

1. **Random Item Generation Service**
   - At game start (or when a new run begins), the module calls the OpenAI API with a prompt describing:
     - Current run parameters (difficulty, floor number, player stats, prior choices).
     - Desired item types (weapons, armor, consumables, relics).
     - Target rarity distribution and thematic constraints (Lovecraftian, fungal, decayed, etc.).
   - The model returns a JSON payload describing a pool of items with:
     - Name, type, rarity, and flavor text.
     - Core stats (damage, defense, cooldowns, etc.).
     - Special effects or affixes (on-hit, on-kill, curses, buffs).
   - The C++ module parses this JSON into strongly typed item structs, caches them, and exposes them to C# via a clean API such as:
     - GetRandomItemByRarity(rarity)
     - GetItemPoolMetadata()

  - All in-run item drops then pull from this pre-generated pool, combining AI creativity with deterministic, reproducible selection for balance and performance.

2. **Centralized OpenAI API Integration**
   - All communication with OpenAI is handled in C++, using:
     - An HTTP client to send POST requests.
     - JSON serialization/deserialization to build prompts and parse responses.
   - <u>The module:</u>
     - Manages API keys (never exposed to game scripts).
     - Queues and throttles requests to respect rate limits.
     - Provides asynchronous callbacks to C# (signals/events) so gameplay never blocks on network latency.
   - Other systems (emotion dialog, lore) call this common service instead of implementing their own HTTP logic.

Technical Architecture and Design Patterns

- **Core Design:**
  - A C++ service registered as a Godot class via GDExtension.
  - A thin C# wrapper class that exposes methods for use from scripts (e.g., NecronomiCoreAI.Instance.GetItem(), RequestLoreSnippet(), etc.).
- **Patterns Used:**
  - <u>Facade:</u> The module hides HTTP, JSON, and token management behind a small, high-level API.
  - <u>Command / Request Objects:</u> Each AI call is represented as a request object (type, prompt, parameters), which is queued and processed in order.
  - <u>Observer / Signals:</u> Responses are delivered to C# via Godot signals/events (e.g., OnItemsGenerated, OnLoreReceived), allowing UI and gameplay systems to react asynchronously.
  - <u>Caching:</u> The module caches item pools and repeated responses (e.g., frequently requested lore templates) to reduce token usage and avoid unnecessary latency.

C# API Surface (High-Level)

Example methods exposed to Godot scripts might include:

- **Item Generation**
  - RequestItemPool(runConfig) – triggers an API call and emits ItemPoolReady(poolId) when complete.
  - GetRandomItemFromPool(poolId, rarity) – returns an item struct for loot drops.
- **Lore and Dialog (Shared Integration)**
  - RequestLoreSnippet(context) – returns a short lore text based on room/level state.
  - RequestEnvironmentalMessage(context) – generates cryptic wall/writing text.

Parameter and return types will be defined in a small set of DTO-style structs (e.g., ItemDefinition, RollResult, LoreResponse) to keep it simple and predictable.

Integration Approach with Godot

- **The C++ module is compiled as a GDExtension and loaded by Godot at startup.**
- **A C# singleton (autoload) wraps the C++ API and is available globally (e.g., AIService).**
- **Game systems interact only with the C# layer:**
  - Loot tables, chests, and enemy death scripts call into the item generator.
  - Narrative systems call lore/environmental text generation.
- **Initialization:**
  - On main menu or run start, the module:
    - Negotiates with the OpenAI API to generate item pools and any run-specific text.
    - Stores results in memory so gameplay calls are instant and do not depend on live network calls mid-combat.

Comparison With Existing Solutions

3. **A typical Godot approach for AI integration uses:**

   - Pure GDScript or C# HTTPRequest nodes,
   - Manually written JSON handling, and
   - Per-system, ad-hoc API calls.
4. **Our module improves on this by:**
   - Centralizing all OpenAI interactions in one well-tested C++ service.
   - Providing a typed, reusable API for C# scripts instead of repeating boilerplate.
   - Handling performance and token usage proactively (pre-generation and caching).
5. **Compared to external helper scripts or tools, this module:**
   - Feels “native” to Godot (registered classes, autoloads, signals).
   - Is reusable across multiple projects as a drop-in extension for AI-driven item, roll, and text generation.

Game Design Document (Noah)

This game is a 2D top-down dungeon crawler roguelike, drawing inspiration from titles like *Enter the Gungeon* and *The Binding of Isaac*. The core gameplay centers around fluid movement and dodging as players navigate through dangerous rooms filled with enemies and environmental hazards. Combat emphasizes positioning, pattern recognition, and quick reflexes. Enemies are introduced gradually across the game’s levels, creating a steady difficulty curve. The first level contains only a single enemy type to introduce mechanics clearly, while later levels combine all three unique enemy types alongside more complex layouts, culminating in a final level featuring all enemies together and a likely boss encounter. Each level is pre-generated, but will require a robust set of assets such as wall tiles, environmental obstacles, and fully animated sprites for all enemy types to maintain visual quality and immersion. Custom modules will support randomized gameplay elements, ensuring each run feels unique. This may include luck-based events, environmental interactions, and variable modifiers that alter enemy behavior or room dynamics. These systems aim to enhance replayability—an essential trait for any roguelike experience. Visually, the game embraces a Lovecraftian aesthetic fused with an overgrown dungeon theme. Dark, damp stone chambers blend with invasive plant life, creating an atmosphere that feels abandoned, unsettling, and ancient. Characters and enemies take inspiration from real-world mushrooms, using their natural properties, appearances, and behaviors as references for enemy design. This reinforces the game's swampy, eldritch tone and helps build a cohesive world. The game is developed for PC, targeting players who enjoy indie roguelikes, atmospheric dungeon crawlers, and games with high replay value.

Technical Implementation Plan

Development Timeline and Milestones

The implementation will be broken into small, clear milestones so that it can be integrated with level design, art, and dialog work:

- **Weeks 1–2 – Design and Scaffolding**
  Define the exact responsibilities of the random item generation module and its C# API. Set up the C++ GDExtension project, create a minimal Godot test scene, and agree on JSON schemas for item definitions *(name, rarity, stats, flavor text)* and item pools.

- **Weeks 3–4 – Core Item and API Logic *(Offline)***
  Implement the internal C++ data structures for item pools and expose them through a simple C# wrapper *(e.g., functions to create a pool and pull a random item by rarity)*. During this phase, the module will use hard-coded or local JSON data instead of the OpenAI API, so we can test drops, inventory UI, and loot flow without network dependencies.

- **Weeks 5–6 – OpenAI Integration and Pre-Generation Flow**
  Connect the module to the shared OpenAI service, build prompts that describe the desired item pool for a run, and parse the JSON response into the existing item structures. Implement run start logic so that item pools are generated once when a new game begins and then reused by loot systems *(chests, enemies, rewards)* during gameplay.

- **Weeks 7–8 – <u>Balancing, Optimization, and Final Polish</u>**
  Playtest generated items for power level and theme consistency, adjust prompt constraints and clamping rules, and add fallbacks for when the API is unavailable *(e.g., a small static item set)*. Final tasks include cleaning up code, documenting the public API for the rest of the team, and verifying that item generation works reliably with Ali's NPC dialog module and Noah's level layouts.

Technology Stack and Dependencies

The item module will be implemented as a **C++ GDExtension** loaded by Godot at startup and wrapped by a **C# singleton** for easy use in scripts. It will rely on:

- Godot's GDExtension C++ bindings.
- A lightweight C++ JSON library to serialize requests and parse responses.
- An HTTP/HTTPS client for communication with the OpenAI API *(shared configuration with the other custom modules)*.
- The existing Godot project structure, scenes, and prefabs for items, chests, and enemies already maintained by the team.

This keeps the integration consistent with the rest of the project while isolating the ‘hard' work *(API, JSON, caching)* in one reusable component.

Testing Strategy and Success Metrics

Testing will happen in three layers:

- **Local / Offline Testing:**
  Use canned JSON files instead of live API calls to verify that pools are created correctly, that items can always be drawn at each rarity tier, and that the C# API behaves as expected when called from loot and inventory scripts.
- **Integration Testing in Godot:**
  Run the game end-to-end with the module active to confirm that:
    - A new run correctly pre-generates an item pool.

    - Chests, enemies, and other reward sources always receive valid items.
    - No noticeable stutter or blocking occurs during gameplay.
- **Playtesting and Tuning:**
  Have the team run multiple sessions to evaluate variety and balance. Success is defined as:
    - No crashes or softlocks caused by missing or malformed item data.
    - Item stats staying within agreed ranges for each rarity.
    - Each run feeling different and thematically fitting the ‘Elder Bloom’ fungus/Lovecraft aesthetic.

Risk Assessment and Mitigation Strategies

- **Risk: OpenAI API latency or downtime**
  Mitigation: Pre-generate item pools at run start instead of in the middle of combat, and provide a small static fallback pool that can be used if the API cannot be reached.
- **Risk: Invalid or overpowered items from the model**
  Mitigation: Define hard min/max ranges for stats and clamp values in C++ after parsing. Discard malformed entries and regenerate or fall back to static items when necessary. Use playtest feedback to revise prompts and balance rules.
- **Risk: Integration conflicts with other modules**
  Mitigation: Keep the public API small and well-documented so Ali and Noah can call it without touching internal logic. Use a shared test scene and weekly check-ins to detect integration issues early.
- **Risk: Time constraints near the end of the term**
  Mitigation: Prioritize getting the **offline** *(non-API)* item system working first. OpenAI integration is layered on top of a working local system, so the game remains playable even if AI-driven generation needs to be simplified.

Team Coordination & Workflow

- **Primary Focus Areas**
  - *Landon:* API integration and ensuring stable communication between the game and external AI systems.
  - *Alexandra:* Visual asset production, including sprite creation, animation, and tileset design, as well as development of the AI-Powered Emotion Dialogue module.
  - *Noah:* Level design, environment layout, and collision implementation, along with development of the Random Roll Generation module.
- **Meeting Schedule & Communication**
  - The team will meet once per week on Wednesdays at 7:30 PM to review progress, plan upcoming tasks, and resolve any major issues. Ongoing communication will be handled through Discord, which will be used for both quick updates and larger discussions when needed. GitHub will serve as the primary workspace for version control, project tracking, and collaborative development.
    - *Repository:* https://github.com/acc668/IMG420-Final
- **Role Distribution**
  - Responsibilities are assigned according to each member's strengths and preferred areas of expertise to ensure efficient development and timely completion of each component. Team members are expected to complete their tasks on-schedule, and coordinate with others when systems overlap or dependencies arise.
- **Communication Protocols**
  - If a team member encounters a significant problem or technical block, they should send a summary email detailing the issue. If the problem cannot be resolved through asynchronous communication, it will be escalated for discussion during the next weekly meeting or addressed in a dedicated session on Discord.
- **Conflict Resolution**

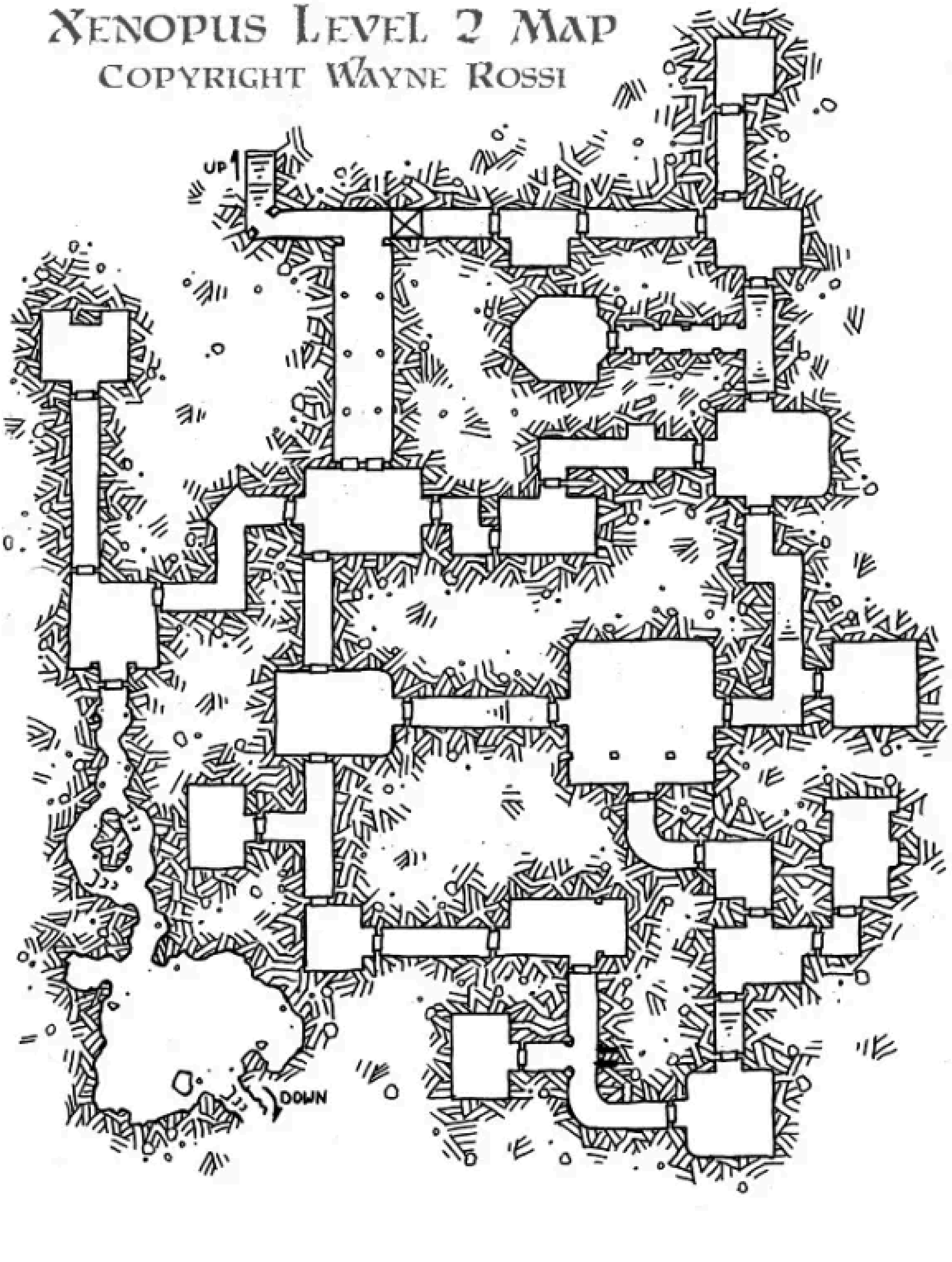
- In the event of disagreements or workflow conflicts, the team will first attempt to resolve the issue through open discussion in meetings or Discord. If needed, the team will collectively determine a compromise that supports project progress, maintains fairness, and keeps development on track.

References and Resources

Using old godot projects from assignments #4 and #1 from Noah Valdez as a basic starting point, and the code for some of the assets.

Tutorials and documentation consulted.

Inspiration and prior art.


Xenopus Level 2 Map
Copyright Wayne Rossi
UP
DOWN