Capstone Milestone 02

Core Gameplay Loop & Hybrid Architecture

Arad Fadaei, Mahboobeh Yasini, Johnpaul Tamburro

Version 2.0, February 12, 2026

Table of Contents

1. Milestone Goal . . . 1
2. Core Technology Stack . . . 2
    2.1. Frontend: Svelte . . . 2
    2.2. Backend: FastAPI . . . 2
    2.3. LLM Provider: Ollama . . . 2
3. Scope & Architecture Adjustments . . . 3
    3.1. Hybrid Command Architecture . . . 3
    3.2. Hybrid Generation Strategy . . . 3
4. In Scope: The Next 3 Weeks . . . 4
    4.1. Front End Terminal Commands (System Level) . . . 4
    4.2. User Defined Themes & Context . . . 4
    4.3. Enhanced Tool Calling Reliability . . . 4
    4.4. Game Generation Refactor . . . 4
    4.5. Voice Model Integration (TTS) . . . 4
    4.6. State Synchronization Protocol . . . 5
5. Out of Scope (For This Milestone) . . . 6

Chapter 1. Milestone Goal

The Goal: This milestone focuses on bridging the gap between basic functionality and a playable, robust game loop. We will enhance the user interaction model by introducing a hybrid command system (system-level terminal commands vs. narrative natural language) and refining the LLM's role to focus on narrative enrichment rather than raw state management.

The Problem (Current State): Currently, the system relies heavily on the LLM for all interpretations, which can be slow and inconsistent for game-mechanic operations. Additionally, the game generation is monolithic, leading to heavy token usage and slower load times. Use of voice alone can be limiting for system administrative tasks.

Chapter 2. Core Technology Stack

The project leverages a modern, high-performance stack designed for low-latency AI interaction and reactive UI updates.

2.1. Frontend: Svelte

We utilize **Svelte (via SvelteKit)** for the user interface. Svelte's compile-time reactivity allows us to build a highly responsive "Terminal" interface without the runtime overhead of Virtual DOM frameworks.

- **Role:** Currently handles command parsing and reactive state visualization. It is architected to handle the upcoming audio capture (Web Audio API) and Text-to-Speech integration.
- **Justification:** Instant state updates are crucial for the "typing" effect of text generation and will be essential for real-time feedback during future voice interactions.

2.2. Backend: FastAPI

The backend is built with **FastAPI (Python)**.

- **Role:** Orchestrates game logic, manages the procedural generation algorithms, and serves as the middleware between the client and the LLM.
- **Justification:** Python is the native language of AI/ML. FastAPI provides native asynchronous support (`async`/`await`), which is critical when handling long-running LLM inference requests without blocking the server.

2.3. LLM Provider: Ollama

We use **Ollama** for local Large Language Model inference.

- **Role:** Hosting and serving open-source models (specifically **Qwen 3**) locally during development.
- **Justification:**
- **Cost:** Eliminates per-token context costs during the heavy iteration phase of prompt engineering.
- **Privacy & Speed:** Local inference reduces network latency and keeps user data ensuring privacy.
- **Flexibility:** Allows us to hot-swap models (e.g., testing a quantized 7B model vs. a larger model) to benchmark performance vs. accuracy.

Chapter 3. Scope & Architecture Adjustments

3.1. Hybrid Command Architecture

We are adopting a **Hybrid Command System** to handle user inputs:

1. **System Commands (Slash Commands):** Deterministic, rule-based commands for game management (e.g., `/save`, `/theme`). These will bypass the LLM for instant feedback and reliability.
2. **Narrative Actions:** Natural language inputs (e.g., "I want to explore the cave") that are processed by the LLM for intent classification and narrative generation.

3.2. Hybrid Generation Strategy

To solve the latency and coherence issues in game generation:

- **Programmatic Structure:** Room layouts, connectivity, and basic stats will be generated via procedural code (Python), ensuring valid game logic and fast execution.
- **Narrative Overlay:** The LLM will be prompted to "flesh out" this skeleton with descriptions and lore based on the user's selected theme.

Chapter 4. In Scope: The Next 3 Weeks

4.1. Front End Terminal Commands (System Level)

Implementation of a client-side command parser to handle non-gameplay interactions.

- **Positional Arguments:** Support for structured syntax (e.g., `/load slot1`).
- **Slash Command Support:** Implementation of core commands:
- `/help`: Lists available commands.
- `/save <slot>` & `/load <slot>` (Interface only, persistence later).
- `/theme <name>`: Debug command to switch contexts.
- **Syntax Feedback:** Immediate client-side validation without API calls.

4.2. User Defined Themes & Context

Moving beyond a static fantasy setting to dynamic, user-driven world generation.

- **Theme Interface:** A UI flow at game start allowing users to select or type a genre (e.g., Sci-Fi, Horror, High Fantasy).
- **Context Injection:** Passing this theme data into the LLM context methods so all generated descriptions align with the chosen setting.

4.3. Enhanced Tool Calling Reliability

Improving the "Smart" part of the game loop to be more robust against hallucinations.

- **Structured Output Validation:** Enforcing strict schema adherence for LLM responses.
- **Retry Logic:** Automatically retrying LLM calls if the returned JSON is malformed.
- **Schema Refinement:** Providing clearer tool definitions to the specific model (Ollama/Local) to improve classification accuracy.

4.4. Game Generation Refactor

Splitting the dungeon generation responsibility.

- **Code-First Dungeon Generation:** Writing Python logic to create graph-based dungeon maps (Nodes/Edges) deterministically.
- **Narrative Decorators:** Using the LLM only to generate room names and descriptions based on the structural skeleton, reducing token costs and generation time.

4.5. Voice Model Integration (TTS)

- **Text-to-Speech:** Integrating a TTS engine to read the narrator’s response aloud, fulfilling the

"Voice First" vision of the charter.

4.6. State Synchronization Protocol

To support the "Game State Management" goal from Milestone 01, the data layer must be unified.

- **API Expansion:** Extending `GameTurnResponse` to include a snapshot of the `PlayerState` (HP, Inventory) and `RoomState` (Exits, Name) alongside the narrative.
- **Frontend Stores:** Implementing Svelte 5 reactive stores (`$state`) to track these values essentially, enabling the "invisible UI" to still react to game events (e.g., screen flashing red on low health).

Chapter 5. Out of Scope (For This Milestone)

- **Complex Persistence:** While command interfaces for Save/Load will be built, the backend database implementation for long-term storage is deferred.
- **Advanced Combat:** Turn-based mechanics and stat tracking are future features; combat will remain narrative-driven for now.
- **Multi-turn Memory:** We are focusing on immediate intent; long-term NPC memory/history is out of scope for Milestone 02.