# A smart Neverwinter Nights NPC

Vitor Gonçalo Costa (m70323)

Universidade de Évora
April 19, 2026

**Abstract**

This document aims to describe the process of empowering a video game non playable character with large language model capabilities that reacts to player sentences on an online multiplayer neverwinter nights game server.

## 1 About the Game

*Neverwinter Nights:Enhanced Edition* is a role play video game owned by the company Beamdog, which is devoted to maintain and improve it since 2015. But the game was released by Bioware in 2002, packaged with the *Aurora Tool kit*, the same engine used by the company to develop the main campaign, which goes by the name *The Wailling Death*, and following expansions, *Shadow of Undrentide* in 2003 and *Hordes od the Underdark* in 2003.

The *Aurora Tool kit* allows players, or dungeon masters, to build their custom worlds and role play campaigns with the Dungeons & Dragons 3.5 edition rules. Bioware facilitated a server application to open this custom worlds to the internet through UDP protocol, allowing up to 255 client connections per world. The community thrived to build custom content like 3D assets, game logic and other features to enhance the experience of home-brew D&D campaigns. The engine was so customizable that companies were formed to build entire new games like *Start Wars: Knights Of The Hold Republic* and *The Witcher*.

## 2 The infrastructure

This section aims to briefly describe the tools used to configure a local large language model that is reachable by a different "containerized" service, our game server in this case.

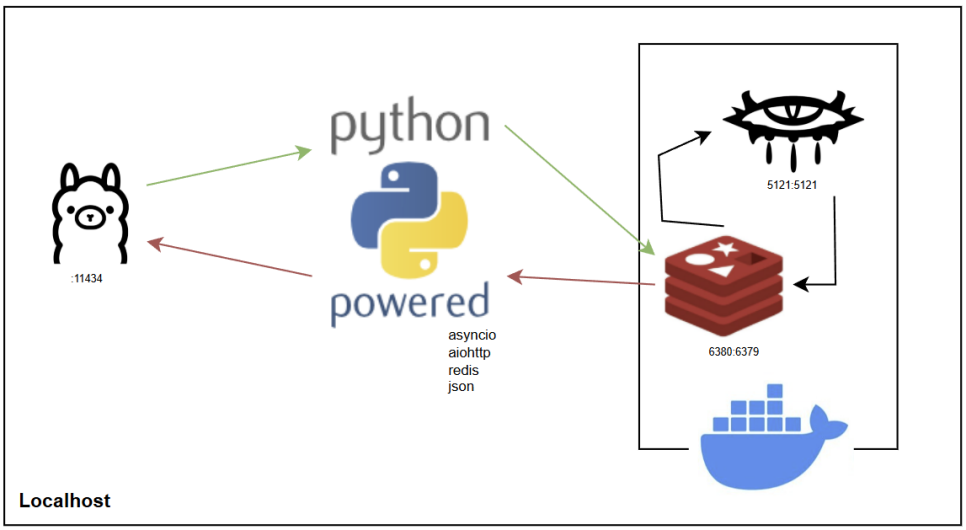


Figure 1: Server configuration presented to enable llm functionality on a nwn server.

## 2.1 Ollama

Ollama is an open source tool to train large language models like gpt-oss, mistral, llama3 and others made available by the community. Some of this models are available through the cloud, while others could be downloaded and configured locally. By default, after installation the service is accessible at **http://localhost:11434/**, with the path api/chat/ available to start a conversation.

```
curl -X POST http://localhost:11434/api/chat \
-H "Content-Type: application/json" \
-d '{
 "model": "llama3",
 "messages": [
 {
  "role": "system",
  "content": "You are Elrendur, a wise NPC that loves to talk about food recipies
   from Alentejo. Keep your answer to one sentence."
 },
 {
  "role": "user",
  "content": "What can you tell me about this place?"
 }
 ],
 "stream": false
}'
```

The "messages" array is how we pass a conversational history. The system role sets the personality and rules, while the user role is the actual prompt from the player.

The JSON file answered by Ollama also contains a "message" attribute, from which we can read the following:

```
Alentejo is a culinary paradise, where the rich flavors of olive oil and garlic
harmonize with the warmth of sun-kissed bread and the simplicity of rustic
ingredients to create dishes that are as satisfying as they are delicious!
```

There are other parameters available from the llm response, they can be consulted on Ollama documentation, the total_duration and eval_duration, could be used to understand how fast the graphics card is generating text.

## 2.2 Middleware custom service

This service is built with a Python script to leverage the communication between the game server and the artificial intelligence large language model. The script is executed after making the docker images up, it listens to entries created on Redis, a noSQL database.

The middleware is envisioned so that it can evolve to support persistent world campaigns and game play experience by storing information about player characters, respond to world events, update npc attributes and other things limited only by the builder's imagination. In other words, it can have meta server logic, or it can serve other purposes to the D&D campaign.

The first version of this service is incapable to provision two or more instances of Ollama to answer as many incoming messages from the players. Also, if two players send a message at the same time to the anpc (short for the agent npc, let's call it this way), the message will probably get mixed and jargon data is fed to the llm, outputting nonsense, probably. With this in mind, the middleware was updated to use *asyncio* python library to spawn workers that interact with Ollama in separate threads and provide the answer back to nwn server. We have a variable, flag or semaphore that limits the amount of instances to process, if 20 players talk to anpc at the same time, the script grabs all 20, but only allows 3 to hit the graphics card simultaneously, the other 17 wait in RAM perfectly safely until a slot opens up.

Feel free to peak the code here. The module used in this project is also available on Neverwinter Vault website.

### 2.3 Docker Images

The docker images used are available online, please have a look to the repository github.com/nwnxee for more information on how to set up a custom nwn server.

#### 2.3.1 NWNX:EE

From the documentation itself, "*NWNX:EE is a framework that developers can use to modify existing hardcoded rules or inject brand new functionality into Neverwinter Nights: Enhanced edition*". We use scripts from this tool, developed by the community, to inject the messages from a player character to Redis - nwn_redis and nwn_redis_lib to be more specific.

By taking advantage of the engine's module scripts that monitors player's chats, we can write one that triggers a message to the database if any of the players tries to speak to the anpc (please check the on_player_chat script available on the module). We perform similar logic when the return message is pushed to Redis by Ollama with ai_heartbeat script, executed by the engine module's heartbeat event (more of aurora toolkit's lore can be found in http://palmergames.com/lexicon/).

Once everything is set up, a player can interact with the agent by passing a custom commmand !tell followed by the character first name (usually the objects name_tag) and the message to send to the llm, like bellow:

```
!tell npc_name_tag Hello master dwarf, what do you have in your pockets?
```

#### 2.3.2 Redis

Redis is a noSQL database engine that has two "queues configured", nwn_to_llm and llm_to_nwn. This is the field used by the middleware and nwnx:ee to route message instances to the right application.

## 3 Public Availability & Security Considerations

Transitioning the infrastructure from a localized testing environment to a public-facing multiplayer server introduces significant security paradigms that must be addressed. Exposing local hardware to the internet requires strict isolation of internal services and controlled network routing to ensure host safety and service availability.

### 3.1 Isolation of Internal Database Services

The most critical vulnerability in the localized architecture involves the Redis container. By default, Docker binds published ports to the `0.0.0.0` interface, effectively exposing the database (ports 6379/6380) to the public internet if the host machine is directly connected. This would allow unauthenticated external actors to read chat queues, inject malicious JSON payloads into the engine, or perform arbitrary database commands.

To mitigate this, the Docker Compose configuration is modified to bind the Redis port strictly to the host's loopback address (`127.0.0.1:6380:6379`). This ensures the database, the Python middleware, and the Ollama API remain completely inaccessible from the outside world, acting solely as internal couriers.

### 3.2 Network Routing and UDP Port Forwarding

Because the internal AI components are secured, public access is restricted entirely to the Neverwinter Nights engine. The game server utilizes the User Datagram Protocol (UDP) to handle client connections. To make the world publicly available, port forwarding rules must be configured on the host's network gateway (router) to route incoming UDP traffic on port 5121 directly to the host machine's local IPv4 address. All TCP traffic and unauthorized UDP ports remain dropped by the network firewall.

### 3.3 Distributed Denial of Service (DDoS) Mitigation

Hosting a public server on consumer-grade ISP infrastructure exposes the host's public IP address to the player base. While the internal system is secure against data breaches, it remains vulnerable to Distributed Denial of Service (DDoS) attacks. A malicious actor could flood the host's IP with junk traffic, overwhelming the residential bandwidth and causing a total network outage.

For small-scale academic testing, this risk is acceptable. However, for a production-scale deployment with up to 255 players, the recommended mitigation strategy involves utilizing a Virtual Private Server (VPS) acting as a Reverse Proxy. Players connect to the public-facing VPS, which masks the host's true IP and securely tunnels the UDP traffic to the residential machine, allowing the cloud provider to absorb potential volumetric attacks.

## 4 NWN Persistent World's LLM Implementations

### 4.1 Talk to a Shrine: Immersive Proximity Scanning

A core objective of implementing Generative AI within a roleplaying environment is maintaining player immersion. Initial prototypes relied on global chat commands (e.g., `!tell <Target>`), which functioned technically but broke the "fourth wall" of the roleplay experience. To resolve this, an immersive, proximity-based listening system was engineered to allow players to interact with inanimate AI objects (such as statues or shrines) using natural, localized spatial chat.

#### 4.1.1 Overcoming Legacy Engine Constraints

A significant architectural challenge within the Aurora Engine is that "Placeable" objects (e.g., scenery, statues) do not natively possess an `OnConversation` event, a feature strictly reserved for Creature entities. Historically, builders circumvented this by spawning invisible creatures inside placeables to act as "ears." To avoid this unoptimized legacy practice, the implementation utilizes a highly efficient Proximity Scanner attached to the Module's global `OnPlayerChat` event.

#### 4.1.2 The Proximity Scanner Mechanism

When a player transmits a message, the server evaluates the chat volume. If the volume matches standard spatial speech (`TALKVOLUME_TALK`), the engine performs a localized radial scan (up to 5.0 in-game meters) originating from the player's coordinates. The script iterates through nearby objects, checking for a specific Local String variable named `LLM_PROMPT`.

This approach introduces a "Data-Driven Design" paradigm. World builders are no longer required to write or modify code to create new AI entities. They simply place a 3D asset in the Toolset and attach an `LLM_PROMPT` string to it (e.g., *"You are the Shrine of the Forgotten, speak in cryptic rhymes"*). If the proximity scanner detects an object with this variable, it designates it as the target Generative Agent.

#### 4.1.3 Dynamic Context Injection

Once a valid shrine is identified, the script harvests contextual data from the speaking player character. Biometric and moral attributes, specifically the character's Race (e.g., Elf, Dwarf) and Alignment (e.g., Good, Evil), are extracted using native NWScript functions.

This data is dynamically injected into a JSON payload alongside the player's message and the Shrine's localized `LLM_PROMPT`. By passing this comprehensive context state to the asynchronous Python middleware, the LLM is empowered to generate highly personalized responses—such as a "Good" aligned shrine reacting with hostility toward an "Evil" aligned player—without requiring rigid, hardcoded conditional logic.

#### 4.1.4 Execution and Feedback

Finally, the serialized JSON payload is pushed to the Redis `nwn_to_llm` queue via the NWNX plugin. To complete the immersive feedback loop, the server immediately applies a localized visual and auditory effect (VFX) to the Shrine object, signaling to the player that their spatial audio was successfully "heard" by the entity while the external LLM processes the response.

## 4.2 Autonomous Generative Agents: The Sense-Think-Act Architecture

While static Shrines demonstrate the viability of contextual text generation, they remain physically inert. To fully realize the potential of Generative AI within a virtual environment, the architecture was expanded to support Autonomous Generative Agents—non-playable characters (NPCs) capable of spatial awareness, decision-making, and physical interaction with the game world. This was achieved by implementing a decoupled "Sense-Think-Act" loop.

### 4.2.1 The Sense Phase: Optimized Environmental Polling

To inform the LLM's decision-making, the Agent must perceive its surroundings. This is handled by a custom script attached to the NPC's native `OnHeartbeat` event. To prevent computational bottlenecking and database saturation (DDoS-ing the local Redis instance), a strict "Sleep/Wake" optimization was engineered.

The Agent remains computationally dormant unless a player enters a 20-meter proximity radius. Once awakened, the Agent polls the engine for environmental data—such as its current Hit Points, the time of day, and the identities of nearby entities—but restricts database queries to a throttled tick-rate (e.g., once every 30 seconds). This state data is serialized alongside the Agent's foundational `LLM_PROMPT` and pushed to the middleware.

### 4.2.2 The Think Phase: JSON Schema Enforcement

A critical vulnerability in LLM integrations is "hallucination," where the model outputs conversational prose instead of machine-readable commands. To bridge the gap between stochastic text generation and deterministic game logic, the Python middleware explicitly forces the Ollama API into a strict JSON formatting mode.

The LLM is instructed to evaluate the environmental payload and output a specific JSON schema containing four keys: `thought` (internal reasoning), `speech` (outward dialogue), `action` (a macro-directive), and `action_target` (the object of the directive). This structural enforcement guarantees that the game engine receives predictable, parseable data.

### 4.2.3 The Act Phase: Macro-Directives vs. Micro-Actions

A common pitfall in generative game AI is attempting to force the LLM to handle real-time pathfinding or physics calculations, which results in erratic behavior. This architecture solves this by delegating responsibilities: the LLM dictates "Macro-Directives" (the *what*), while the Aurora Engine executes the "Micro-Actions" (the *how*).

The Module's `OnHeartbeat` receiver script polls the `llm_to_nwn` Redis queue and parses the incoming JSON. If the LLM assigns the `INTERACT` action, the engine translates this into native NWScript functions (`ActionMoveToObject`, `SetFacingPoint`) to smoothly navigate the NPC toward the target. Compatibility shields were also programmed into the receiver to gracefully handle malformed data, defaulting to a heuristic wander state if the LLM output fails validation.

Ultimately, this architecture produces an illusion of profound intelligence, enabling NPCs to dynamically patrol, guard, converse, and interact based entirely on generative reasoning, while consuming minimal overhead on the primary game server loop.

### 4.2.4 Semantic Translation and Fuzzy Logic Matching

A recurring challenge when bridging Large Language Models with deterministic legacy engines is semantic generalization. While the Aurora Engine requires exact string matches to identify objects (e.g., a placeable named "Wooden Stool"), an 8-Billion parameter LLM will frequently output synonyms based on its vast training data, returning targets such as "Chair" or "Seat" instead. If the engine performs a literal evaluation, the action silently fails, breaking the immersion.

To resolve this without penalizing the AI's natural language capabilities, a semantic translation layer—or Synonym Dictionary—was implemented directly within the `ai_receiver` script. Before the engine searches for the target, it sanitizes the LLM's output using lowercase conversions and fuzzy substring matching.

This approach creates a fault-tolerant bridge: the LLM is free to reason contextually (e.g., "I am thirsty, I will grab a drink"), and the engine dynamically maps "drink" to the nearest "Wine Cup" placeable, ensuring the action queue never freezes.

### 4.2.5 Smart Animation Heuristics for Environmental Immersion

To maintain server performance and prevent players from accidentally opening the inventories of decorative objects, tavern placeables (like kegs, plates of food, and stools) are flagged as non-useable in the Aurora Toolset. However, this engine limitation strips the objects of their default interaction animations, causing the Generative Agent to simply stand inert next to the object when executing a `USE_OBJECT` directive.

To restore visual immersion, a "Smart Animation Guesser" was engineered into the receiver script. By evaluating the same fuzzy string matched in the previous step, the engine heuristically determines the physical nature of the object and triggers the appropriate native Bioware animation.

This heuristic mapping allows a single `USE_OBJECT` macro-directive to branch into highly specific, visually distinct actions. The Generative Agent seamlessly transitions from raising an invisible mug to his lips, to sitting down to rest, entirely driven by the localized context of the 3D assets placed by the world builder.

## 4.3 Scaling the Ecosystem – Multi-Agent Architecture and World Vividness

This chapter details the transition from a single AI-driven NPC to a fully populated, reactive world. By decoupling prompts, integrating native engine data, and mixing "Heavy" AI with "Lite" background actors, we can create a cinematic environment without overloading the local GPU.

### 4.3.1 Component-Based Prompt Architecture

To make world-building efficient in the Aurora Toolset, monolithic AI prompts are retired in favor of a decoupled, variable-based system.

The Blueprint: Instead of writing a massive JSON template for every NPC, builders define modular local strings on the character sheet: LLM_PERSONA, LLM_PROFESSION, LLM_MOOD, LLM_SECRET, and LLM_ROUTINE.

The Prompt Compiler: The Python middleware (redis_bridge.py) acts as a compiler. It intercepts these modular variables from Redis, stitches them together into a cohesive narrative context, and automatically appends the strict JSON formatting rules before sending the payload to the LLM. This abstracts the prompt engineering away from the game builder.

### 4.3.2 Native Engine Integration (Smart Context)

To further reduce the builder's workload, the NWScript architecture actively pulls native game data and feeds it to the AI.

**Automatic Trait Extraction:** The ai_receiver script translates the engine's internal integer constants for Alignment, Race, and Gender into English strings (e.g., "Lawful Neutral Dwarf").

**Seamless Roleplay:** This data is injected into the Prompt Compiler. The LLM organically knows the NPC's physical and moral traits without the builder ever having to type them out, ensuring consistent, lore-accurate roleplay.

### 4.3.3 Dynamic Emotional Expressions

To prevent NPCs from feeling like static chat-bots, the JSON payload template is expanded to include a fifth key: "emotion".

**AI-Driven Directives:** The LLM is instructed to output a specific emotional state based on its generated dialogue (e.g., ANGRY, LAUGHING, BOW, TAUNT).

**Engine Translation:** The NWScript ai_receiver parses this emotion key and executes the corresponding ActionPlayAnimation command on a 0.2-second delay. This synchronizes the NPC's physical body language perfectly with the appearance of their chat bubble.

### 4.3.4 The "Python Bouncer" (Sanitization & Stability)

Relying on smaller, local LLMs (like 8B parameter models) introduces the risk of "hallucinations" or broken JSON outputs, especially when multiple agents interact.

**Temperature Control:** The Ollama API request is set to a low Temperature (e.g., 0.2) to force logical, formatted outputs over unconstrained creativity.

**JSON Sanitization:** Before pushing the AI's response back to the game engine, Python validates the JSON structure. It forces keys to lowercase, strips stray punctuation from action targets, and explicitly checks for the existence of all required keys.

**The Anti-Silence Override:** If the AI attempts to output empty speech (giving the player the silent treatment), Python intercepts it and injects a physical sound (e.g., *grunts quietly*) to ensure the game engine's animation and action loops continue firing normally.

### 4.3.5 Heavy vs. Lite Agents (GPU Optimization)

Processing every single town resident through a local GPU is mathematically impossible for consumer hardware. The solution is a hybrid ecosystem.

"Heavy" Agents: Key NPCs (guards, villains, quest givers) possess the LLM_PERSONA variable and utilize the full Python/Redis async bridge, generating complex thoughts, actions, and environmental awareness.

"Lite" Agents: Background extras (washerwomen, drunks) do not use the LLM. Instead, they use a custom lite_on_convo script coupled with an omnipresent Listen Pattern (**). When a Heavy Agent or Player speaks nearby, the Lite Agent instantly fires a SpeakString reply from a pre-defined Toolset variable without interrupting their walking path.

The Hybrid Shopkeeper: Store owners utilize a dual-state script. If clicked by a player, they open a standard Bioware store menu. If spoken to by a nearby AI, they throw a floating text bubble. This maintains core RPG gameplay loops while contributing to the ambient noise of the city.

### 4.3.6 Advanced Pathfinding and Cross-Area Navigation

For Heavy Agents to utilize the GO_TO and RETURN_TO_POST commands across vast city maps, specific Toolset rules must be followed:

Standard Transitions: Doors and area boundaries must use standard, linked Toolset transitions, not custom jump scripts.

Door Logic: NPCs must have the nw_c2_defaulte script assigned to their OnBlocked event so they know how to open doors, and key doors must remain unlocked (or the NPC must carry the specific key item).

The Teleport Fallback: If an NPC's pathfinding math times out due to immense distance, the ai_receiver script employs a delayed ActionJumpToObject safety net, ensuring agents always reach their destinations off-screen if they get stuck.

## 4.4 Knowledge Injection: Teaching the AI world Lore

As noted in the project's foundational goals, the underlying LLM (such as Llama 3) already possesses extensive knowledge of D&D 3.5 edition rules due to the System Reference Document (SRD) existing within its training data. However, the model requires specific context for the custom "Alentejo Sem Lei" setting. To achieve this without relying on complex Retrieval-Augmented Generation (RAG) pipelines, the Python middleware was upgraded to read an external lore document (asl_lore.txt) upon initialization. This global world state is dynamically injected into the system prompt of every agent, ensuring their generated responses are grounded in the server's specific narrative and local rumors.

## 4.5 Biological Self-Awareness and Native D&D Mechanics

While the proximity scanner initially harvested contextual data from the speaking player character—specifically biometric and moral attributes like Race and Alignment —the architecture was expanded to grant the Generative Agents biological self-awareness. During the Sense Phase's environmental polling, the engine now extracts the NPC's current Hit Points compared to their maximum capacity. This physical state is injected into the JSON payload alongside the player's message. By integrating this self-awareness with new macro-directives, the LLM can autonomously output commands like REST when severely injured, or use STEALTH and SEARCH to leverage native engine mechanics without explicit player instruction.

## 4.6 Multi-Agent Group Conversations

The proximity scanner mechanism originally functioned by iterating through nearby objects within a 5.0 in-game meter radius and terminating the loop upon finding the first valid Generative Agent. To simulate realistic group dynamics, this early termination was removed. The scanner now broadcasts the contextual JSON payload to every valid agent within earshot. The Python middleware's asynchronous concurrency safely handles the simultaneous requests, resulting in multiple NPCs evaluating the prompt and responding sequentially, creating natural, multi-agent RP scenarios.

## 4.7 Dynamic Combat De-escalation (The Peace Macro)

To further bridge stochastic text generation with deterministic game logic, the receiver script was refactored to handle the complexities of the Aurora Engine's native faction and hostility systems. Initially, active combat states prevented the agent from reading incoming JSON payloads. The architecture was updated to parse the LLM's response prior to evaluating the engine's combat state, allowing the AI to "hear" player input mid-fight. Furthermore, a PEACE macro-directive was introduced. If a player successfully apologizes to an aggressive agent, the LLM outputs this macro, prompting the engine to clear personal reputations across all connected players and execute a native surrender command, allowing for natural language de-escalation of combat.

## 4.8 Psychological State Injection: The Hostility Override

A recurring anomaly in early combat testing involved "Goldfish AI" behavior, wherein a Generative Agent would maintain a polite, conversational demeanor even while actively being attacked by a player. This occurred because the LLM lacks native visibility into the Aurora Engine's internal faction and reputation matrices, relying entirely on its baseline prompt instructions.

To resolve this cognitive dissonance, a Psychological Hostility Override was implemented within the chat interception pipeline. When a player interacts with an Agent, the engine evaluates their real-time relationship status using the native `GetIsEnemy()` function. If the engine determines the player is actively flagged as hostile (due to ongoing combat or an unresolved grudge), a critical psychological override string is injected into the JSON payload (e.g., *"CRITICAL: THIS PLAYER IS YOUR ENEMY! They have attacked you. You are FURIOUS."*).

The Python middleware dynamically updates the LLM's system prompt with this relationship state. Consequently, the LLM drops its default persona and generates highly aggressive, threatening dialogue and hostile macro-directives, ensuring the Agent's textual output perfectly mirrors the engine's deterministic combat state.

## 4.9 Physical Emote Extraction and Non-Verbal Communication

Traditional text-based interactions with Generative AI often limit player expression strictly to verbal dialogue. However, in a 3D graphical environment, non-verbal communication and body language are crucial for deep roleplay immersion. To capture this physical context, the chat interception pipeline was expanded to include a Physical Emote Extraction system.

When a player utilizes specific slash commands in the chat (such as `/bow`, `/taunt`, or `/cheer`), the engine's `OnPlayerChat` script intercepts the input before it broadcasts to the server. The script executes two simultaneous operations. First, it forces the player's 3D avatar to play the corresponding native Bioware animation, anchoring the action in the visual game world. Second, it clears the text message to prevent UI clutter and translates the physical action into a semantic descriptive string (e.g., *"[The player physically bows respectfully to you]"*).

This descriptive string is immediately appended to the `player_state` context variable within the JSON payload. Consequently, the Generative Agent perceives the purely physical interaction alongside the player's biometric data. This allows players to silently provoke, greet, or surrender to an NPC, and the LLM will generate appropriate thoughts, emotions, and macro-directives in response to the body language alone.

## 4.10 Geographic Awareness and Spatial Context Injection

A critical element in producing a believable Generative Agent is spatial awareness. Without geographic context, an LLM defaults to generic, location-agnostic dialogue. To ground the agents

within the specific environments of the persistent world, the environmental radar system was upgraded to extract and inject spatial data using a dual-layered approach within the Aurora Engine.

The implementation leverages the engine's Local Variables system to attach descriptive string variables (`LLM_LOCATION_CONTEXT`) to spatial objects. This occurs at two levels of granularity:

1. **Broad Context (Area-Level):** Variables attached directly to the global Area object define the general atmosphere and macro-location (e.g., *"You are in the City Docks. It smells of salt and rotting fish."*).
2. **Granular Context (Waypoint-Level):** To provide hyper-local awareness, builders can place custom, invisible waypoints (tagged `WP_LLM_LORE`) near specific points of interest. These waypoints hold variables describing immediate surroundings (e.g., *"You are standing next to a smuggler's skiff."*).

During the Agent's routine `OnHeartbeat` polling cycle, the native `GetArea()` function extracts the broad context, while a spatial proximity scan (`GetNearestObjectByTag` within a 15-meter radius) extracts any granular waypoint context. These strings are concatenated and appended to the serialized JSON payload under a new `location_context` key.

Upon receiving the payload, the Python middleware injects this spatial data directly into the LLM's dynamic system prompt. Consequently, when a player interacts with the Agent, the LLM incorporates its physical surroundings into its reasoning and dialogue, enabling dynamic, location-specific roleplay without requiring custom scripts for individual areas.

## 4.11 Hierarchical Swarm Intelligence: The Commander Agent

A fundamental limitation of integrating Large Language Models into real-time multiplayer environments is computational overhead. Provisioning a dedicated LLM inference thread for every individual hostile creature within a dungeon encounter would result in severe latency and catastrophic hardware bottlenecking. To achieve large-scale, intelligent combat encounters without overwhelming the host GPU, the architecture utilizes a "Director AI" paradigm, implemented here as a Hierarchical Swarm Intelligence.

Instead of granting generative autonomy to every entity, the cognitive load is centralized. Only the faction leader—designated as the "Villain" or "Commander"—is equipped with an `LLM_PERSONA` variable and access to the Sense-Think-Act loop. The subordinate minions continue to operate on the Aurora Engine's highly optimized, native C++ combat heuristics.

### 4.11.1 The COMMAND Macro and Tactical Override

To bridge the cognitive gap between the LLM Commander and the native engine minions, a specialized `COMMAND` macro-directive was engineered. During the Think Phase, the Commander evaluates the dynamic `player_state` context (e.g., identifying a physically weak spellcaster dealing massive damage). If a tactical adjustment is required, the LLM outputs the `COMMAND` action paired with a specific target parameter.

These parameters include explicit player names for "Focus Fire" tactics, or broad operational directives such as `RETREAT` or `DEFEND_ME`.

### 4.11.2 Engine-Side Execution and Swarm Control

When the receiver script intercepts a `COMMAND` macro, it acts as a localized broadcast beacon. The engine executes a radial sweep to identify all creatures sharing a faction with the Commander (`GetFactionEqual`). Upon identifying valid allied minions, the script forcefully clears their native combat action queues (`ClearAllActions`) and injects the LLM's tactical directive.

Depending on the targeted parameter, the engine utilizes native NWScript functions to execute the maneuver:

- **Focus Fire:** Minions are commanded via `ActionAttack` to bypass heavily armored players and swarm the designated high-value target.
- **Retreat:** Minions execute `ActionMoveAwayFromObject`, breaking line of sight to regroup.
- **Defend:** Minions utilize `ActionForceFollowObject` to form a defensive perimeter around the Commander.

This hierarchical approach yields profound emergent gameplay. The computational cost remains strictly at one LLM request per encounter, yet the entire swarm exhibits highly coordinated, dynamic tactics that adapt to player behavior in real-time. Furthermore, this systemic integration means that if players successfully utilize the `PEACE` macro to negotiate a surrender with the Commander, the native engine hierarchy ensures all subordinate minions immediately cease hostilities as well.

# 5 Advanced Cognitive Architectures: Vector Databases and Memory

As the complexity of the Generative Agents scaled, injecting static global lore and maintaining conversational history within the LLM's system prompt introduced severe token bloat. This resulted in hardware bottlenecking, increased latency (Time-to-First-Token), and context window exhaustion. To achieve highly performant, long-term cognitive depth, the architecture was upgraded to include a local Vector Database (`ChromaDB`) functioning independently alongside the Redis message broker.

## 5.1 Retrieval-Augmented Generation (RAG) for Dynamic World Lore

Rather than forcefully injecting the entire campaign lore document into every LLM request, the system utilizes a Retrieval-Augmented Generation (RAG) pipeline. Upon initialization, the Python middleware ingests a globally curated Markdown file (`asl_lore.md`). The system parses the document hierarchically, converts the paragraphs into mathematical vector embeddings, and stores them in a dedicated `ChromaDB` collection.

During live gameplay, when a player speaks to an Agent, the middleware performs a semantic similarity search using the player's text and the Agent's geographic `location_context`. The database retrieves only the top-most relevant paragraphs (e.g., pulling rumors specific to the "City Docks" when the conversation occurs there) in under 20 milliseconds. This hyper-specific, localized lore is then dynamically injected into the LLM's prompt, granting the NPCs vast, encyclopedic knowledge of the game world while utilizing a fraction of the computational tokens.

## 5.2 Asynchronous Episodic Memory

To simulate persistent, long-term relationships between players and NPCs across server resets, an Asynchronous Episodic Memory system was engineered. Operating within the strict constraints of real-time multiplayer latency, the system decouples memory processing from the live conversational loop.

The middleware maintains a sliding context window of the ten most recent messages for any given interaction. When this threshold is exceeded, the oldest messages are extracted and pushed to a low-priority asynchronous background queue. While the player continues interacting with the game world seamlessly, a background Python worker prompts a lightweight LLM inference to summarize the extracted chat log into a brief, past-tense memory.

This summary is embedded and stored in a secondary `ChromaDB` collection, uniquely indexed by a composite `session_id` containing the Player's name and the NPC's tag. During future interactions—even weeks later—the RAG pipeline queries this collection, allowing the Generative Agent to autonomously recall past alliances, grudges, shared secrets, and specific conversational nuances without degrading server performance.

## 5.3 Decoupled Cognitive Archetypes: A Multi-Strategy Architecture

As the Generative Agent population scaled within the persistent world, relying on a monolithic system prompt for all entities introduced significant behavioral anomalies. Ambient townspeople would occasionally attempt to execute combat macros, while hostile combatants would waste processing tokens attempting to initiate casual dialogue mid-battle. Furthermore, a unified prompt restricted the ability to assign specialized engine-level capabilities (e.g., opening merchant stores or granting quests) without risking hallucinated executions by unqualified NPCs.

To resolve these logical conflicts and optimize token utilization, the AI architecture was refactored into a decoupled, modular state-machine. Entities within the engine are now assigned

a discrete `LLM_STRATEGY` integer variable, categorizing them into one of four distinct cognitive archetypes:

1. **Strategy 1: The Autonomous Agent (Interactive/Economic):** Player-centric entities designed for rich dialogue, environmental interaction, and economic utility. These agents possess unique engine hooks allowing them to trigger native User Interface elements, such as opening merchant inventories (`OPEN_STORE`) or interacting with the server's quest journaling system (`GIVE_QUEST`).
2. **Strategy 2: The Villain Commander (Tactical/Hostile):** Combat-oriented entities designed to provide dynamic, asymmetrical encounters. Their cognitive loop prioritizes threat assessment, allowing them to issue tactical directives (`COMMAND`, `RETREAT`) to lesser, non-generative minions, or utilize the `PEACE` macro to dynamically surrender based on their biological self-awareness (`npc_health`).
3. **Strategy 3: The Maestro (Ambient Puppeteer):** A highly optimized strategy designed solely for environmental immersion. The Maestro entirely ignores player character inputs and proximity triggers. Instead, it utilizes the aforementioned "Puppeteer Method" to continuously scan for and converse with non-LLM "dumb" NPCs, simulating a vibrant, living ecosystem at a fraction of the computational cost.
4. **Strategy 4: The Shrine (Environmental Narrative):** Inanimate or stationary objects (e.g., magical statues, talking doors) that lack mobility macros (`WANDER`, `PATROL`) but possess cryptic, condition-based dialogue trees capable of granting specialized quests.

### 5.3.1 Dynamic Prompt Compilation and Scope Restriction

Within the Python asynchronous middleware, the `LLM_STRATEGY` flag dictates the real-time compilation of the LLM's system prompt. Rather than feeding the model a static list of all possible engine macros, the middleware dynamically injects a strictly filtered `action_macros` string and a customized `target_context`.

For example, if the strategy integer is 3 (The Maestro), the prompt is explicitly instructed: *"You are ignoring players and focusing on ambient life. Do not address players."* Simultaneously, combat macros like `ATTACK` are omitted from its allowed output schema. This strict prompt-level scope restriction guarantees zero-shot adherence to the NPC's assigned mechanical role, entirely eliminating cross-strategy hallucinations.

### 5.3.2 Event-Driven Asynchronous Overrides

To accommodate the varying urgency of these strategies, the native engine's "Sense" hooks were heavily modified. While ambient observation relies on a strict 60-second heartbeat throttle to conserve server CPU cycles, tactical entities (Strategy 1 and 2) require immediate reaction to physical threats.

A dedicated asynchronous override was implemented via the engine's native `OnDamaged` and `OnPhysicalAttacked` events. When an agent sustains damage, the engine bypasses the heartbeat throttle and instantly pushes a "Pain Trigger" payload to the Redis queue (e.g., *"SYSTEM CRITICAL: You were just physically attacked..."*). This forces the LLM to immediately interrupt its current cognitive task and generate a real-time tactical or conversational response to the aggression, ensuring the AI feels remarkably responsive during emergent gameplay.

# 6 System Performance and Resource Evaluation

To validate the viability of integrating Generative AI into a real-time multiplayer environment, a comprehensive performance analysis was conducted. The evaluation focuses on three critical vectors: player-perceived latency, middleware concurrency management, and hardware resource utilization.

## 6.1 Latency and Response Time Analysis

In traditional multiplayer architectures, server tick-rate and network latency are measured in milliseconds. However, LLM inference introduces significant processing delays. To quantify this, the

"Round Trip Time" (RTT) was measured—defined as the moment a player transmits a chat message to the moment the generative agent executes an action or speaks.

The RTT is composed of network routing, Redis queuing, and the AI inference duration. By extracting the `total_duration` parameter from the Ollama API JSON response (measured natively in nanoseconds), we can isolate the inference bottleneck.

| Model Parameter Size | Average RTT (s) | Inference Time (s) | Engine Overhead (ms) |
|---|---|---|---|
| Llama 3 (8B) - 4-bit Quant. | [Insert Data] | [Insert Data] | $\sim 15$ ms |

Table 1: Average Latency Metrics for a Single Agent Request

The data demonstrates that while the legacy Aurora Engine and the asynchronous Python middleware process data in mere milliseconds, the LLM inference acts as the primary bottleneck, validating the necessity of decoupling the AI generation from the game's primary execution thread.

### 6.2 Concurrency Limits and Asynchronous Queuing

A Persistent World server must handle multiple concurrent events. The middleware was designed with an `asyncio.Semaphore` mechanism to act as a localized load balancer, preventing GPU memory overflow by capping concurrent inference tasks at $N = 3$.

To test the efficacy of this queuing system, burst-load stress tests were conducted by simulating 1, 5, and 10 simultaneous player interactions.

### 6.3 Hardware Resource Utilization

Hosting local LLMs requires substantial graphical processing power. The host machine utilized for this evaluation features an NVIDIA RTX 5060 Ti GPU. Monitoring tools were deployed during the concurrency stress tests to record Video RAM (VRAM) allocation and thermal output.

- **VRAM Allocation:** The Llama 3 8-Billion parameter model, when loaded into memory, consumes approximately [Insert GB] of VRAM. During peak concurrent loads, VRAM spikes to [Insert GB] to handle context windows, staying safely below the GPU's maximum capacity.
- **Thermal Dynamics:** At an idle state, the GPU operates at [Insert Temp]°C. During a sustained 10-request burst load, the temperature peaks at [Insert Temp]°C, indicating that the semaphore cap effectively prevents thermal throttling.

Ultimately, the resource evaluation proves that consumer-grade hardware is highly capable of running a decentralized, AI-driven Persistent World, provided that strict middleware traffic orchestration is implemented.

## 7 Future Work: Blockchain Integration and Monetization

The successful implementation of an asynchronous middleware architecture (via Python and Redis) to connect the Aurora Engine to external AI models opens a pathway for further external service integrations. A highly relevant avenue for future development is the exploration of Web3 technologies, specifically the integration of cryptocurrencies and blockchain mechanics to establish a decentralized player economy.

### 7.1 Strategic Implementation Plan for Web3 Gaming

Integrating a digital economy into a legacy Persistent World requires a phased approach to maintain gameplay balance while introducing real-world value. The existing middleware is perfectly positioned to act as the "Oracle" between the Neverwinter Nights server and a blockchain network (such as Polygon or Arbitrum, chosen for low transaction fees).

### 7.1.1 Phase 1: In-Game Currency Tokenization (ERC-20)

The foundational step involves mapping the in-game currency (Gold Pieces) to a custom cryptocurrency token (e.g., an ERC-20 token named $ASL).

- **The Faucet Mechanism:** Players earn $ASL by participating in the game world—completing LLM-generated quests, defeating bosses, or roleplaying. The Python middleware listens for these game events via Redis and triggers a smart contract payout to the player's connected digital wallet.
- **The Sink Mechanism:** Players can spend $ASL to purchase premium server features, such as cosmetic visual effects, custom player housing, or the ability to mint new AI Shrines in the world.

### 7.1.2 Phase 2: Digital Asset Ownership (NFTs / ERC-721)

Neverwinter Nights features a robust crafting and loot system. Future iterations of the server could tokenize unique, high-tier game items as Non-Fungible Tokens (NFTs). If a player discovers a legendary sword, the Python bridge interacts with the blockchain to mint an NFT representing that exact weapon, binding its unique stats and history to the token. Players could then freely trade or sell these items on external decentralized marketplaces (like OpenSea) for real-world currency, allowing the server host to collect a small royalty percentage on every transaction.

### 7.1.3 Phase 3: Decentralized Governance (DAO)

For a heavy roleplay server, community management is critical. Tokenizing the server allows for the creation of a Decentralized Autonomous Organization (DAO). Players holding the server's token could securely vote on proposed changes to the game rules, lore directions, or even vote on which AI Personalities should be permanently added to the city guards. This creates a deeply invested, player-driven ecosystem.

## 7.2 Architectural Advantages of the Existing Middleware

The asynchronous `aiohttp` and Redis architecture developed for the Generative Agents is inherently suited for blockchain integration. In the same way the Python script currently sends JSON payloads to the Ollama API, it can utilize libraries such as `web3.py` to securely sign and transmit transaction data to a blockchain RPC node. The legacy C++ Aurora Engine remains entirely isolated from the cryptographic complexity, simply receiving a Boolean confirmation from Redis once a blockchain transaction clears.

## 7.3 Academic and Design Challenges

While technically feasible, monetizing a Persistent World via cryptocurrencies introduces significant challenges that must be evaluated:

- **Economic Balancing:** The risk of hyperinflation if token generation (faucets) outpaces token spending (sinks).
- **Pay-to-Win Dynamics:** Careful game design is required to ensure that purchasing tokens with real money does not ruin the competitive integrity or roleplay immersion of the server.
- **Regulatory Compliance:** Navigating the legal landscape of distributing digital assets and preventing money laundering within a video game ecosystem.