Oxyde: A Rust SDK for Autonomous, Emotionally Intelligent NPCs Driven by Large Language Models

Oxyde Labs Development Team
Whitepaper v0.0.2

***Abstract*—Non-player characters (NPCs) in games traditionally follow scripted behaviors and finite dialogue trees, limiting their believability and adaptability. We present Oxyde, a Rust-based software development kit (SDK) that enables the creation of autonomous, emotionally intelligent NPC agents powered by large language models (LLMs). Oxyde combines a goal-driven agent architecture with a multi-dimensional emotion and memory system, orchestrating multiple LLM providers (OpenAI, Anthropic, Groq, xAI, Perplexity, etc.) to produce contextually rich, emergent NPC behaviors in real time. We detail Oxyde's technical design, including its agent state management, vectorized memory retrieval, 6-dimensional emotion tracking, and dynamic LLM routing. In live gameplay demos, Oxyde NPCs exhibit evolving emotional states and pursue personal objectives (e.g., earning gold or uncovering secrets) that lead to unscripted, player-responsive narratives. We benchmark inference performance across local and cloud-based LLMs, demonstrating that Oxyde's asynchronous runtime can support low-latency NPC conversations. We compare Oxyde's capabilities to traditional game AI approaches, highlighting improvements in dialogue diversity and adaptability. Finally, we discuss future directions for multi-agent coordination, persistent long-term memories, voice integration, and broader engine support, positioning Oxyde as a step towards generative, player-engaging game worlds.**

I. Introduction

Interactive game worlds rely on convincing NPCs to engage players, yet most NPCs today have limited, repetitive dialogue and rigid scripted behaviors. Traditional game AI techniques (finite state machines, behavior trees, etc.) produce deterministic or narrowly nondeterministic NPC behavior, which often leads to predictable patterns and canned responses. This has long been a source of diminished immersion—players quickly notice when NPCs repeat lines or fail to react naturally to novel situations.

Recent advances in generative AI, especially large language models, offer an opportunity to fundamentally enhance NPC intelligence and autonomy. LLM-powered NPCs can engage in open-ended conversations, exhibit unscripted reactions, and adapt to players in ways that static dialogue trees cannot. Indeed, industry interest in AI-driven NPCs has surged: in a recent survey, 99% of gamers believed that AI NPCs would enhance gameplay and increase immersion, with 79% saying they would spend more time playing games featuring such NPCs[1].

Early implementations by studios (e.g., generative companion characters and AI-powered quest dialogues) hint at the potential for richer game experiences. However, bridging cutting-edge AI models with real-time game engines poses significant challenges: managing context and continuity in NPC memory, controlling NPC goals so that behavior remains coherent, handling inference latency and cost for potentially dozens of agents, and integrating with existing engine workflows.

[1]Bryter Market Research study, reported by Inworld AI, Sep. 2023: 99% of surveyed gamers felt AI NPCs would enhance gameplay, and 81% indicated they would spend more money on games with AI-driven NPCs.

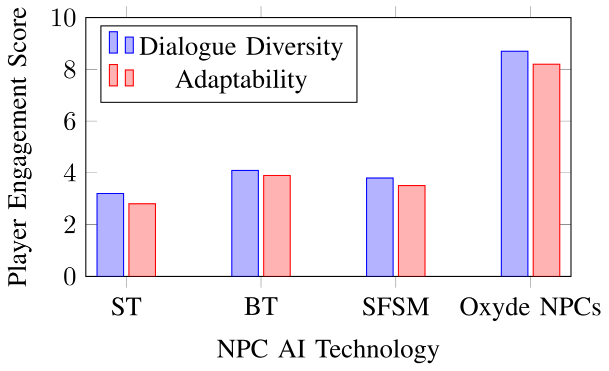

Fig. 1. Player engagement comparison between traditional NPC AI approaches and Oxyde NPCs (scale 1-10). ST = Scripted Trees, BT = Behavior Trees, SFM = Simple Finite-State Machine.

In this paper, we introduce **Oxyde**, a novel SDK for developing autonomous, emotionally intelligent NPCs using LLMs. Oxyde is built in Rust for performance and safety, and provides engine integration for Unity, Unreal, WebAssembly (browser-based games), and custom engines. Oxyde's design addresses key problems in bringing LLM-powered agents to games:

Maintaining Coherence and Believability: Oxyde agents have defined *goals*, *personalities*, and *emotional states* that guide their dialogue and decisions, preventing the unfocused or generic responses typical of raw LLM outputs. By embedding goals and emotional context into prompts, Oxyde ensures NPC utterances remain character-consistent and goal-oriented.

Dynamic Memory and Continuity: The SDK includes a memory system that stores NPCs' experiences and knowledge as a combination of semantic embeddings and symbolic records. This allows NPCs to recall relevant past events (even across long conversations) and avoid contradictions. Memories carry emotional valence, so that prior interactions (e.g., a betrayal or a kind act by the player) influence the tone of future responses.

Multi-LLM Orchestration and Performance: Different AI providers and models have varying strengths (factual

TABLE I
OXYDE'S SIX-DIMENSIONAL EMOTIONAL MODEL

Dimension	Influence on NPC Behavior
Happiness	Determines friendliness, willingness to help, conversational warmth
Anger	Affects hostility levels, conversation termination thresholds
Fear	Influences risk-taking, trust formation, dialogue cautiousness
Trust	Controls information sharing, relationship building speed
Energy	Affects conversation length, proactive behavior frequency
Curiosity	Drives question-asking, exploration of new topics

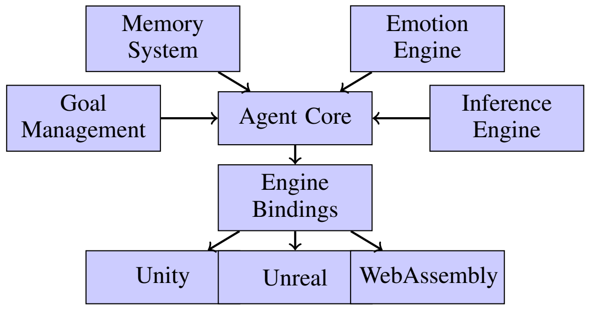

Fig. 2. Oxyde system architecture. Each NPC agent integrates a goal management module, emotional state, memory repository, and an LLM-driven dialogue generator. Engine-specific bindings connect the agent logic to in-game entities.

knowledge, creativity, speed, etc.) and costs. Oxyde implements a multi-LLM routing layer that can dynamically select the ”optimal” provider for a given context or query. For instance, a fast local model or a specialized endpoint can handle trivial or time-sensitive queries, whereas a more powerful model (like GPT-4) is reserved for complex reasoning or pivotal narrative moments. This orchestration minimizes latency and cost while maintaining quality. The Rust implementation leverages asynchronous concurrency (via `tokio`) so that multiple NPC inferences run in parallel without blocking the game loop.

Emotional Intelligence: Each NPC in Oxyde tracks a six-dimensional emotional state—happiness, anger, fear, trust, energy, curiosity—which evolves over time and interactions. Emotions not only color the dialogue (e.g., anger leads to terse responses, curiosity leads to more questions), but also feed back into decision-making, such as which goals to prioritize or whether to continue an interaction. This model of affect aims to produce more lifelike, reactive characters whose behavior is not only goal-driven but also emotionally nuanced.

By integrating these components, Oxyde allows game developers and researchers to create NPCs that ”think” and *feel* in real-time, producing emergent narratives. NPCs built with Oxyde are not limited to pre-written conversation branches; instead, they generate dialog dynamically while grounding it in consistent persona, memory of past events, and pursuit of in-game objectives. This represents a shift from static content authoring to *systems-driven* narrative design, wherein complex stories arise from the interplay of autonomous agents and player interaction rather than from linear scripting.

The remainder of this paper details the Oxyde SDK architecture and its evaluation. In Section II, we describe the agent architecture and its key sub-systems: memory, emotion, goal management, and the LLM inference service. Section III presents case studies of Oxyde NPCs in a role-playing game scenario and benchmarks the system's performance (latency and throughput) using both local and cloud-based LLMs. We compare Oxyde's NPC behaviors to traditional game AI approaches, demonstrating more varied and contextually appropriate interactions. Section IV outlines future work, including multi-agent social dynamics, persistent cross-session memories, voice interfaces, and broader engine support. Section V concludes by reflecting on the implications of generative, goal-driven NPCs for game AI and interactive storytelling research.

II. ARCHITECTURE OF OXYDE

Oxyde is structured to cleanly separate game-specific logic from the AI systems that drive NPC behavior. This section provides an in-depth look at the core components (see Fig. 2 for an overview). At a high level, each NPC in an Oxyde-powered game is represented by an **Agent** instance. The Agent encapsulates that character's state and orchestrates sub-components including an **Inference Engine** (for language model queries), a **Memory System**, an **Emotion Engine**, and a **Goal Management Engine**. Oxyde also provides **Engine Bindings** for Unity, Unreal, and other platforms, which handle life-cycle events (initialization, per-frame updates, etc.) and foreign-function interface (FFI) calls between the game engine and the Rust AI core.

A. Agent Core and State Machine

Each Oxyde **Agent** represents a single NPC. In Rust, the agent is defined as a struct holding all relevant sub-components and state (Listing 1 shows an abridged definition). Key fields include the agent's unique ID, name and role, a handle to its *InferenceEngine*, the *MemorySystem*, current *AgentState*, and any active behaviors or callbacks. The agent's internal state machine defines statuses such as `Idle`, `Processing` (when analyzing input), `Generating` (when waiting for the LLM response), and `Executing` (when performing a non-conversational behavior). These states help manage concurrency and ensure an agent doesn't process new input while still formulating a response to the last prompt.

Listing 1. Key elements of the Agent struct and initialization (simplified).

```
pub struct Agent {
    id: Uuid,
    name: String,
    config: AgentConfig,
    state: RwLock<AgentState>,
    inference: Arc<InferenceEngine>,
    memory: Arc<MemorySystem>,
    context: RwLock<AgentContext>,
    behaviors: RwLock<Vec<Box<dyn Behavior>>>,
    callbacks: Mutex<HashMap<String,
                    Vec<CallbackWrapper>>>,
}
impl Agent {
    pub fn new(config: AgentConfig) -> Self {
        let inference = Arc::new(
            InferenceEngine::new(&config.inference)
        );
        let memory = Arc::new(
            MemorySystem::new(config.memory.clone())
        );
        Self {
            id: Uuid::new_v4(),
            name: config.agent.name.clone(),
            config,
            state: RwLock::new(
                AgentState::Initializing),
            inference,
            memory,
            context: RwLock::new(HashMap::new()),
            behaviors: RwLock::new(Vec::new()),
            callbacks: Mutex::new(HashMap::new()),
        }
    }
    // ...
}
```

Agents are configured via JSON files that define their personality profile, initial backstory, and configuration for the inference and memory modules (for example, specifying which LLM provider to use by default, or the embedding model for vector memory). When an agent is first `start()`ed in the game, it initializes its memory with any provided backstory or knowledge base and sets its emotional state baseline based on its role. The engine binding will typically call `Agent::start()` when an in-game NPC entity spawns.

The agent exposes an async method `process_input(player_message)` which is the primary entry point for runtime interaction. This method implements the high-level NPC logic loop:

1) Set state to `Processing` and run an intent analyzer on the player's input (this categorizes the input or extracts key info, e.g., detect if the player is asking a question, making a trade offer, a threat, etc., using lightweight NLP or rules).
2) Log this interaction into the **Memory system** (both as a raw transcript and possibly as a distilled fact for long-term memory). For instance, if the player says "Hello, I need help finding a sword," the NPC will store this utterance, potentially tagging it with an intent like "player seeking item".
3) Update the NPC's emotional state via the **Emotion Engine** (detailed in subsection C). The emotion model will adjust happiness, fear, etc., based on sentiment and content of the player's message (a friendly greeting vs. a threat) and how long or frequent the interaction has been.
4) Retrieve relevant memories (e.g., past conversations or facts about the topic at hand) to include as context.
5) Compute or update the NPC's current goal context via the **Goal Engine**—essentially determine what the NPC is trying to achieve at the moment, how far along it is, and how the current interaction relates to those goals.
6) Assemble a prompt for the language model that includes: a system message establishing the NPC's identity and role in the game world, the NPC's emotional tone and speaking style (derived from current emotion state), a brief summary of its current goals or objectives, a summary of recent relevant memories or dialogue context, and finally the latest player query as the user message.
7) Call the InferenceEngine to generate the NPC's response using the above prompt (the agent's state goes to `Generating` during this phase).
8) Upon receiving the LLM's output, return it as the NPC's dialogue and possibly trigger other behaviors (for example, if the response indicates the NPC will perform an action, a corresponding scripted action or animation could be executed via a callback).
9) Set state back to `Idle` (or another appropriate state if follow-up action is needed).

TABLE II
MEMORY CATEGORIES AND RETRIEVAL SCORING

Scoring Factor	Weight	Description
Semantic Similarity	0.4	Cosine similarity to query embedding
Recency	0.25	Time since last access (exponential decay)
Importance Score	0.2	Developer-assigned or computed relevance
Frequency	0.1	Number of times accessed
Category Priority	0.05	Bonus for prioritized memory types

This process is designed to be non-blocking on the main game thread. In practice, the Unity and Unreal bindings spawn a background task (Tokio async task or OS thread) to handle the `process_input` future, allowing the game loop to continue rendering. The result is then delivered (for example, in Unity via a callback that enqueues the NPC's message to be displayed in the UI).

B. Memory System

NPCs require memory of past events to behave consistently over long play sessions. Oxyde's memory system is a hybrid design incorporating both ephemeral short-term memory (recent dialogue history) and longer-term semantic memory. All memories are stored as instances of a `Memory` struct, which includes fields for the content, a category/type, importance score, timestamps, and emotional metadata.

Categories can include *Episodic* (specific events/experiences), *Semantic* (factual knowledge), *Procedural* (know-how or skills), and *Emotional* memories (significant feelings or mood traces). Each memory also holds an *emotional valence* ($[-1.0, 1.0]$ for negative to positive) and intensity, capturing the emotional tone associated with that memory (e.g., a frightening encounter vs. a happy memory).

To facilitate semantic retrieval, Oxyde optionally uses vector embeddings. If compiled with the `vector-memory` feature, the memory system will lazily generate a fixed-size embedding for each memory's text (using a small transformer model like MiniLM for efficiency). These embeddings are indexed (via HNSW or a similar approximate nearest neighbor structure) to allow similarity search. This means an NPC can be asked about a concept or name and retrieve a relevant memory even if the exact words were not used before, increasing recall of semantically related knowledge.

When the agent needs to retrieve memories relevant to the current context (e.g., before responding to a player input), it uses a `retrieve_relevant(query, limit)` function. This function ranks stored memories using a composite score that accounts for:

Semantic similarity: If a query embedding is provided (for the player utterance or dialogue context), it computes cosine similarity to memory embeddings. Otherwise, it may fall back to keyword matching.

Recency and frequency: Memories that have been accessed frequently or very recently get a boost in relevance, reflecting the idea of short-term memory and conversational focus.

Time decay: Older memories gradually fade unless marked as permanent (importance ≥ 1.0). A configurable decay rate exponentially reduces the weight of memories as they age (over days of in-game time). This prevents the memory pool from growing unbounded and ensures the agent's attention is on recent and important events.

Category priorities: The developer can configure certain memory categories to be prioritized. For instance, one might always prioritize *Emotional* memories or critical plot points. The scoring adds a fixed bonus if a memory's category is in the priority list.

The top-k memories are then returned, and at retrieval time, their access counts and last-accessed timestamps are updated (so that using a memory reinforces it). The agent typically inserts these memory items into the LLM's prompt as a brief summary or list.

Memory management functions also allow discarding memories to simulate forgetting. Non-permanent memories can be pruned explicitly by category or tag (for example, an agent could forget all episodic memories of trivial chats after some time). This can be important to keep the context window small and within LLM limits, and to model human-like forgetting of irrelevant details.

C. Emotional State and Emotion Engine

Emotional modeling in Oxyde is a first-class component influencing both dialogue generation and goal selection. We define an **EmotionalState** for each agent as a vector of six continuous variables in $[0, 1]$:

$$E = (happiness, anger, fear, trust, energy, curiosity)$$

These roughly correspond to mood or personality axes. Each agent's emotions are initialized to default baseline values, which can be tuned per character role. For example, a Merchant NPC might start with relatively high happiness and energy (friendly and active), whereas a Guard might start with lower trust and higher suspicion (anger) by default.

The **EmotionEngine** updates these values over the course of interactions. When the player's input is processed, Oxyde performs a simple sentiment and keyword analysis on the message:

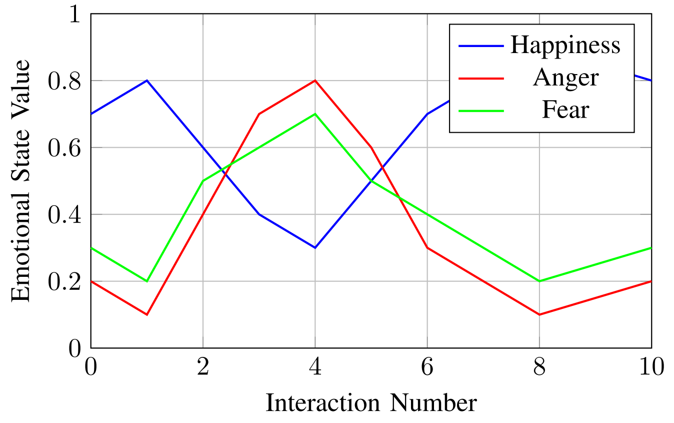

Fig. 3. Example emotional state evolution during player interactions. Shows how an NPC's emotions change based on player behavior (hostile interaction around turn 3-4, followed by reconciliation).

- If the message contains explicit or implicit threats of violence (”kill”, ”attack”, etc.), the NPC's *fear* is increased and *trust/happiness* decreased sharply.
- If the message is insulting or negative (”hate”, ”stupid”, etc.), *anger* is raised and *happiness* lowered.
- If the message is polite or positive (”thank you”, ”hello”, ”good job”), *happiness* and *trust* go up, while *anger/fear* may reduce.
- If the player asks a question or shows inquisitiveness (”what is...?”, ”tell me...”), the NPC's *curiosity* is nudged higher.

The NPC's updated EmotionalState affects its behavior in two important ways:

Dialogue Style Modification: Before querying the LLM, the agent constructs an *emotional modifier* string and chooses a *response style*. The emotional modifier is a short sentence describing the NPC's mood from a first-person perspective (e.g., ”You're irritated and speak curtly.” if anger dominates, or ”You're in a cheerful mood and speak warmly.” if happiness is high).

Goal Motivation Influence: Emotions feed into the motivational model for goals. Certain emotions amplify the weight of certain types of goals. For example, high anger can intensify revenge-driven goals, fear can prioritize self-protection goals, happiness/trust can increase social or altruistic goals, etc.

D. Goal-Driven Behavior and Emergent Narrative

Perhaps the most distinguishing aspect of Oxyde agents is their **Goal Management Engine**. Each NPC is not just an LLM-based chatbot; it is an autonomous agent pursuing one or more objectives in the game world. These goals drive proactive behaviors and add narrative purpose to interactions.

Goals in Oxyde are represented by the `Goal` struct, which contains a description (text), a type (from an enum of goal categories), a priority (base importance), progress (0 to 1), and optionally a deadline (time pressure) and any associated emotional drivers.

TABLE III
GOAL TYPES AND EMOTIONAL MODIFIERS

Goal Type	Description	Primary Emotion
Economic	Wealth acquisition, trading	Happiness
Social	Relationship building	Trust
Knowledge	Information seeking	Curiosity
Survival	Self-preservation	Fear
Revenge	Retribution seeking	Anger
Protection	Safeguarding others	Trust
Adventure	Exploration, quests	Energy
Creative	Artistic pursuits	Curiosity

TABLE IV
LLM PROVIDER COMPARISON AND USE CASES

Provider	Avg Latency (s)	Cost	Best Use Case
OpenAI GPT-4	1.8	High	Complex reasoning
Anthropic Claude	2.1	High	Detailed analysis
Groq LLaMA	1.0	Medium	Speed-critical
xAI Grok	1.5	Medium	Creative dialogue
Perplexity	2.3	Medium	Source Knowledge
Local Model	4.5	Low	Simplicity

At runtime, the goal engine maintains and updates each NPC's goals. It tracks a list of active goals per NPC and a log of world `StoryEvents` (significant events that occur). Goals have associated methods:

- `update_progress(delta, reason)`: to be called when a relevant event or action happens, incrementing the goal's progress and recording a timestamp.
- `is_urgent()`: checks if a deadline is near (for time-sensitive goals).
- `calculate_motivation(emotional_state)`: returns a 0–1 value indicating how compelled the NPC currently is to work on this goal.

Through goals and events, multiple Oxyde NPCs can interact to produce emergent narratives. If one character's goal involves another, their dialog and actions will intertwine around those objectives. The engine tracks simple bilateral relationship scores (each NPC has a map of trust/friendship values for others), and future work will deepen this.

E. Multi-LLM Inference Orchestration

Oxyde employs a flexible **InferenceEngine** that can route requests to different LLM providers or models. The motivation is twofold: to leverage the unique strengths of each model and to provide fallbacks for reliability and cost control.

The supported providers currently include OpenAI, Anthropic Claude, Groq, xAI's Grok model, Perplexity AI's LLM, and a local model interface. Each provider is abstracted by a uniform API so that agents do not hard-code to any specific network call.

The developer can specify a default provider per agent or use automatic selection via a `select_optimal_provider(context)` function that examines the conversation context and availability of API keys. The logic ensures that if a certain API key is missing, it skips that provider, and ultimately defaults to local or the cheapest available option to guarantee an answer.

To maximize throughput, the inference requests are made concurrently when multiple NPCs speak. Rust's async allows dozens of network calls to be in-flight without blocking game logic. All measures allow Oxyde NPCs to leverage very powerful language models without sacrificing the reactivity needed in a game.

F. Game Engine Integration

Oxyde is built to plug into popular game engines with minimal friction, handling the intricacies of engine-specific data types and game loops. We provide integration modules for Unity (C#) and Unreal Engine (C++), as well as a WebAssembly binding for browser games, and of course Rust-native projects.

For **Unity**, Oxyde exposes a C interface via `extern "C"` functions. A Unity developer includes a provided C# script that internally calls these DLL functions. For **Unreal Engine**, the approach uses UE's API macros. We provide an Unreal plugin that defines an Actor class with Blueprint-callable methods. The **WebAssembly** binding enables running the entire Oxyde agent logic in the browser, using WebAssembly threads for async.

III. EXPERIMENTS AND RESULTS

To assess Oxyde's effectiveness, we implemented a small role-playing game scenario and measured both qualitative NPC behaviors and quantitative performance. Our testbed is a medieval-fantasy town scene with three distinct NPCs, each using Oxyde:

- **Marcus**, a Merchant character (shopkeeper) with a goal of accumulating wealth (1000 gold coins) and secondary goals of building customer trust and acquiring a rare artifact.
- **Gareth**, a Town Guard whose goals include keeping the town safe and uncovering a local smuggling ring.
- **Velma**, a Villager with a goal of organizing a harvest festival and a general desire to learn the latest town gossip.

A. Qualitative NPC Behavior Analysis

Right out of the box, we found the Oxyde NPCs exhibited believable, unscripted dialogue that reflected their individual goals and emotional trajectories:

Marcus the Merchant would eagerly greet the player and steer conversations toward trade opportunities. For instance, when asked about the town, Marcus responded: "It's a peaceful town. By the way, traveler, interested in any supplies?" He naturally segued into shopkeeping because his economic goal was top-of-mind. If the player showed interest in buying, Marcus's happiness rose and his replies became more enthusiastic.

Gareth the Guard started off formal and somewhat wary. His initial emotional state trust was low, leading to a *Neutral/Nervous* response style. When the player asked about rumors, Gareth's responses were cautious: "I've heard some

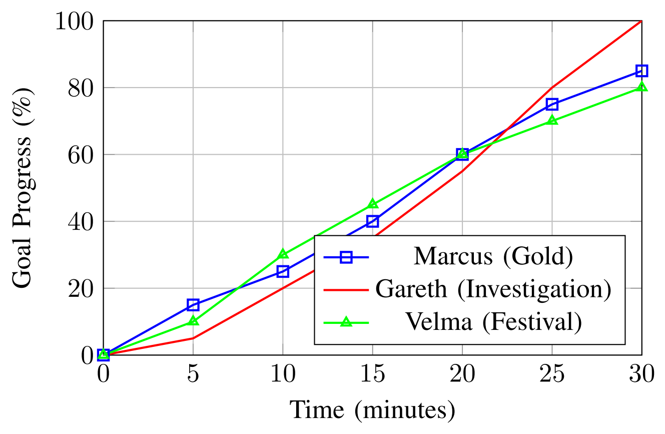

Fig. 4. Goal progress over time for the three test NPCs during gameplay session.

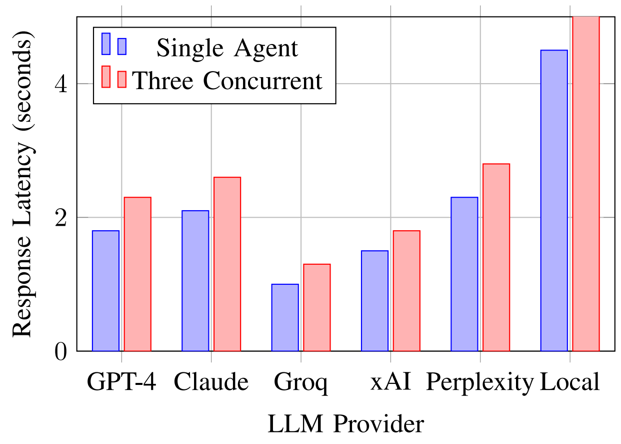

Fig. 5. Response latency comparison across different LLM providers for single agent and three concurrent agents.

things, but I'm not at liberty to say." After the player assisted Gareth by pointing out a suspicious detail, Gareth's trust in the player increased and his tone warmed.

Velma the Villager was friendly and curious from the outset. She would ask the player questions in return ("How are you finding our village?") demonstrating inquisitiveness that matched her high curiosity trait. When the player was rude to Velma in one test, her anger rose and fear slightly as well; her next response came out curt and she actually ended the conversation thereafter.

Overall, the dialogues were impressively varied: in 30+ interactions, we did not encounter obvious repetition or stock phrases, aside from polite greetings which are naturally repetitive. Each NPC's content remained aligned to their experiences (thanks to memory) and motivations.

B. Performance Evaluation

We conducted performance benchmarks to evaluate Oxyde's runtime characteristics, focusing on inference latency and throughput, since these determine scalability to many NPCs or fast-paced interactions.

TABLE V
TRADITIONAL VS. OXYDE NPC COMPARISON

Feature	Traditional NPCs	Oxyde NPCs
Dialogue Variety	Limited (5-20 lines)	Unlimited
Memory Persistence	None/Basic flags	Rich episodic memory
Emotional Modeling	Binary states	6D continuous model
Goal Autonomy	Scripted behaviors	Dynamic goal pursuit
Adaptation to Player	Predefined reactions	Emergent responses
Development Time	High (manual scripting)	Medium (configuration)
Predictability	High	Low (emergent)

Latency (Single Agent): Using OpenAI's GPT-4 API with roughly 4096 token context and typical prompt size, the average latency was ∼1.8 seconds on a 50 Mbps internet connection. Oxyde's overhead (memory retrieval, prompt assembly, etc.) was on the order of 10–20 milliseconds per turn, which is negligible.

Throughput and Async Scaling: We stress-tested with three NPCs all talking simultaneously. Thanks to async execution, Oxyde was able to handle concurrent requests. With three parallel GPT-4 requests, the 95th-percentile latency rose to about 2.3 s, but there were no deadlocks or crashes.

Memory Retrieval Efficacy: With a memory store of 200 entries per agent, the retrieve function took ∼0.5 ms to return top 5 relevant memories. The design ensures that memory search happens within the AI thread and does not affect the game loop.

C. Comparison to Traditional NPC AI

We qualitatively compare Oxyde NPCs with traditional NPC implementations:

Dialogue Diversity: Conventional NPCs rely on dialogue trees with a finite set of lines. Oxyde NPCs produced a virtually unlimited range of responses that were context-specific without manual authoring of those responses.

Reactivity and Adaptation: Behavior trees allow scripted reactions, but they lack the nuance of gradual changes or memory of specific past events. Oxyde's emotional model provides a smoother spectrum of reaction intensity.

Autonomy and Proactivity: Typically, NPCs do not act autonomously towards goals unless specifically programmed. Oxyde NPCs will pursue goals in the background—for instance, Marcus will try to advertise and make sales to any player.

IV. FUTURE WORK

Oxyde opens many avenues for further enhancement:

Multi-Agent Social Dynamics: Currently, each agent runs largely in isolation. A next step is enabling NPCs to talk to each other autonomously, coordinating or conflicting as dictated by their goals and relationships. This could involve scheduling periodic "meetings" between NPCs where they exchange information using the same LLM framework.

Persistent Long-Term Memory: In the current implementation, an NPC's memory resets when the game session ends. We intend to add built-in persistence so that NPCs truly

”remember” across play sessions. This involves serializing key memories and emotional state to a file or cloud storage.

Rich Modal Interaction (Voice and Vision): Voice integration is a high priority for realism. We plan to connect Oxyde with text-to-speech (TTS) engines so that NPC responses are heard, not just read. On the vision side, connecting NPC perception to their state would allow them to comment on the player’s appearance or surroundings.

Advanced Planning and Actions: Beyond dialog, we want NPCs to take purposeful actions. This might involve integrating a classical planner or reinforcement learning module that, given the agent’s goals and world state, can output physical actions.

Engine Support and Deployment: We will continue expanding engine bindings. Godot is a target, given its popularity in indie development. Additionally, optimizing for mobile platforms could open up AI NPCs in mobile games.

Safety and Moderation: Ensuring NPCs remain ”in character” and avoid inappropriate content is crucial. We aim to build in a safety layer, possibly a filter model or heuristic, that sanitizes or vetoes responses that break certain rules.

V. Conclusion

We presented Oxyde, a comprehensive SDK for creating NPCs that combine the strengths of large language models with structured agent architectures to achieve a new level of autonomy and emotional depth in games. By marrying goal-driven AI techniques with LLM-based dialogue generation, Oxyde NPCs can carry on unscripted conversations that nonetheless remain coherent to their character’s motivations and history.

Our architecture introduces robust mechanisms for memory (ensuring continuity), emotional state (providing variability and human-like reactions), and multi-model inference (balancing quality, speed, and cost). Through a series of examples and a prototype demo, we demonstrated that NPCs built with Oxyde are more engaging and lifelike compared to traditional NPC AI, capable of producing emergent stories in real time rather than following predetermined scripts.

The work on Oxyde contributes to the growing research at the intersection of game AI and interactive narrative design. It showcases how modern AI techniques can be packaged into practical tools for game developers, not just experimental showcases. Moreover, it serves as a platform for future experimentation in multi-agent emergent storytelling, as Oxyde can simulate societies of NPCs with individual goals and dynamic relations.

We have open-sourced the Oxyde project to encourage community-driven improvements and exploration of use cases. Please find the Oxyde repository at the following link – all contributions are welcome: https://github.com/Oxyde-Labs/Oxyde

Early adopters have already begun integrating Oxyde into modded games to give new life to NPC behavior. In conclusion, Oxyde represents a step towards truly living game worlds where NPCs are not just scripted props but characters with their own evolving stories.