IT UNIVERSITY OF COPENHAGEN

AI Game Engine

Slop Engine: A Vertically Integrated Game Engine for AI-Assisted Rapid Prototyping

---

Bachelor Project • S26BIBAPRO1PE591

Alexander Rossau <ross@itu.dk>
Bjørn Møgelhøj <bjom@itu.dk>

Supervisors

Martin Mitrovic <marmi@itu.dk>
Sebastian Risi <sebr@itu.dk>

Software Development
Spring 2026

Abstract

This paper presents Slop Engine, an AI-driven game engine, designed to accelerate the prototyping of interactive 3D games. Unlike conventional engines in which AI assistance is integrated through external protocols such as MCP, Slop Engine embeds AI capabilities throughout the engine architecture, granting the assistant direct access to the scene graph, physics simulation, scripting runtime and editor state. The engine uses a hierarchical agent architecture in which a central orchestrator delegates tasks to five specialized subagents responsible for scene construction, script generation, user interface, asset creation and result validation through testing. The system is evaluated through a comparative study of 27 experimental runs across three game prototypes (Breakout, Dodger, and Platformer), comparing Slop Engine against two baselines: a standalone coding agent paired with Babylon.js, and a coding agent connected to a Roblox development server via the Model Context Protocol (MCP). All three configurations used the same underlying large language model to isolate the effect of the tooling layer, rather than the model. The findings indicate that Slop Engine successfully produces functional 3D game prototypes from natural language prompts, but does not outperform the lightweight code-only baseline, which achieved a median quality score of 12 out of 12 compared to Slop Engine's 9 out of 12, while consuming approximately 30 times fewer uncached input tokens and approximately 2.6 times fewer output tokens. The paper discusses the architectural trade-offs that explain this gap and identifies concrete improvements, particularly in feedback-loop design and token efficiency, that could address it.

Table of Contents

1 Introduction

1.1 Motivation

Game development has historically required significant technical expertise across multiple disciplines. A developer building even a simple 3D game must reason about programming, 3D graphics, physics simulation, user interface design and more. Modern game engines such as Unity [1], Unreal Engine [2], and Godot [3] provide comprehensive tooling for these tasks, but professional game development still requires substantial prototyping and pre-production work before full production. Unity's 2026 Game Development Report notes that 67% of polled developers spend three months or less in the prototyping phase, while Unity's production-cycle guidance describes pre-production as including planning, prototyping, pipeline setup, and initial designs. Academic work on game prototyping also argues that games' emergent rule-based behavior calls for short iterations and frequent prototyping [4], [5].

The arrival of capable large language models (LLMs) has prompted the game industry to explore whether AI can reduce this burden. When development of Slop Engine began in February 2026, AI assistance in mainstream engines was largely supplemental: typically limited to code completion plugins or conversational helpers that operated outside the engine runtime, with no direct access to the scene graph, physics state, or asset pipeline. As a result, these tools functioned primarily as programming aids capable of producing code on request, but unable to observe, modify, or reason about the game environment in which that code would execute. Since then, the landscape has begun to shift. As this project neared its end, both Roblox Studio [6] and Unity [7] had released beta integrations using the Model Context Protocol (MCP), granting AI assistants structured access to scene inspection and asset management. These developments signal growing industry recognition that deeper integration is necessary for AI to be genuinely useful in game development workflows.

The emergence of large language models with function calling capabilities has made this kind of integration technically feasible. Models can now be equipped with structured tool interfaces that allow them to invoke internal engine functions directly, provided the engine exposes its functionality in a way the model can act upon. This creates an opportunity to rethink the relationship between AI and game engine architecture from the ground up, designing the engine around AI interaction rather than attempting to fit AI into an existing system. The practical appeal is clear: if an AI system can accept a short natural language description of a game concept and produce a functional prototype, it would substantially lower the barrier to entry for new developers and accelerate the early iteration cycle for experienced ones.

1.2 Problem Statement

Game prototyping is an inherently iterative process. Developers sketch an idea, build a rough version, playtest it, and refine. Each iteration involves coordinating across scene setup, scripting, asset placement, and debugging, tasks that remain time-consuming even with modern tooling. As established in the previous section, AI systems are beginning to take on parts of this coordination, but it remains an open question what kind of environment best supports them. A purpose-built engine that grants the AI direct access to engine internals may reduce the friction of this coordination, but it also introduces requirements: multi-agent orchestration, context management, token consumption, and error propagation across agent boundaries. A simpler approach, an AI agent that writes code directly against a well-documented library, avoids this overhead entirely, at the potential cost of losing the structured feedback and spatial control that engine integration provides. The central question this project investigates is therefore:

> **Research question:** *How effectively can a multi-agent AI system, operating within a purpose-built game engine environment, generate functional game prototypes from short natural language descriptions?*

1.3 Contributions

In this report, we use "vertically integrated" to mean that the AI assistant is not added as a separate external layer, but is integrated across the engine stack: the editor, scene graph, scripting system, asset pipeline, runtime state, and validation workflow. This integration gives the assistant structured ways to inspect and modify the same internal systems that define and execute the game project.

To investigate this question, we designed and built Slop Engine, a browser-based, vertically-integrated game engine with deep integration between the AI assistant and the engine runtime. We then evaluated it against two established AI-assisted game development workflows to isolate the effect of the tooling layer. This work makes the following contributions:

- The design and implementation of Slop Engine, an open-source, browser-based game engine in which the AI assistant, organized into orchestrator and subagent roles, has direct programmatic access to the scene graph, physics simulation, scripting runtime, and editor state.
- A comparative evaluation of prototyping quality and efficiency across three scenarios: Slop Engine, OpenCode paired with Babylon.js, and OpenCode connected to Roblox Studio via the Model Context Protocol. The evaluation uses a custom test harness that automates environment setup, token and timing instrumentation, and result export across 27 controlled runs.
- An analysis of the architectural trade-offs involved in giving AI agents direct access to engine internals, including the token costs introduced by structured tool use and the limitations of single-pass agent architectures.

1.4 Report Structure

The remainder of this report is organized as follows. In Section 2 the relevant background is surveyed: the history and architecture of game engines, the development of large language models and tool-calling agents, AI in game development, and the web technologies that underpin Slop Engine. Section 3 describes the system architecture and the key design decisions made during implementation. Section 4 discusses the work methodology that was used in the creation of Slop Engine. Section 5 presents the evaluation methodology, the test harness, and the results of the 27-run comparative study. Interpreting the findings, Section 6 identifies the limitations of the current system, and considers the ethical implications of automated game development. Section 7 summarizes the contributions and outlines the most promising directions for future work.

2 Background & Related Work

2.1 Game Engines

2.1.1 Brief history of game engines

In the early days of computer games, during the 1970s and 1980s, games such as Breakout [8] had to be programmed directly for specific hardware architectures. Because standardized development tools did not exist, every component of a game had to be built from scratch. Code was also tightly coupled to the machine it ran on, requiring meticulous, hardware-specific performance optimization on a per-game basis, which made development labor intensive and code reusability nearly impossible.

The development paradigm shifted significantly in the mid-1990s, a transition heavily popularized by the technology behind id Software's Doom (1993) and Quake (1996). Developers pioneered a distinct architectural separation between the game's content and the core technology responsible for executing it [9]. Because standardized tools were now available to manipulate content independently of the runtime code, this decoupling facilitated the rise of the modding community. Players could create entirely new experiences by replacing assets and level data while reusing the underlying technological framework [10, 11-13]. While the separation of content and code in titles like Doom and Quake pioneered the concept of game engines, these early frameworks were inherently genre-specific. The shift toward more general-purpose engines began in the late 1990s as developers recognized the commercial viability of licensing their technology as a standalone product. Instead of engines being developed for a specific game, a general-purpose engine can facilitate multiple game genres, while still using the same underlying technology. In 1998 Epic Games created the first Unreal Engine, deliberately architecting the engine to be highly extensible. It included a dedicated editor (UnrealEd), along with an object-oriented scripting language (UnrealScript) that decoupled game logic from the base engine [9]. A scripting language is a programming language designed to control or automate the behavior of a system. This architectural foresight allowed external studios to license the technology and create entirely different genres, such as Ion Storm's immersive simulation Deus Ex (2000), without having to rewrite the core rendering or physics systems [11].

In the early 2000s, the industry saw the rise of genre-agnostic software components like the RenderWare graphics engine or the Havok physics engine [9], [12]. This modular approach proved that engines did not need to make rigid assumptions about the game they were running. Modern general-purpose platforms are the culmination of this evolution. They provide an architecture, where a 3D first-person shooter and a 2D puzzle game can utilize the exact same underlying mathematics, physics simulations, and rendering pipelines, completely removing the genre constraints that defined early engine architecture.

2.1.2 Core Architecture of Game Engines

A modern game engine is a robust collection of integrated middleware and software frameworks designed to facilitate game creation. Rather than building rudimentary systems from the ground up, developers utilize general-purpose engines to handle complex, low-level technical architecture. An important structural concept within these systems is the use of modularity, which refers to the ability for developers to modify engine source-code or swap out specific subsystems. For instance, a studio might replace an engine's default physics system with specialized third-party middleware, such as the Havok physics engine [12], [13]. Similarly, Unreal Engine 5 features proprietary, highly specialized modules like Nanite for advanced micropolygon rendering [14]. It is also important to distinguish between the game engine runtime and the game editor. The engine functions as the core software, often structured as dynamic-link libraries (DLLs), that executes rendering and logic during gameplay. Conversely, the editor is the dedicated graphical user interface (GUI) used by developers to construct the game. Most commercial game engines feature an editor that includes an asset explorer, scene hierarchy, and properties editor [3]. This abstracts the creation process into a visual environment. Instead of specifying spatial coordinates via text, developers can manipulate data structures through an intuitive interface (Figure 1).

1. Scene View
2. Asset Explorer
3. Scene Hierarchy
4. Properties Editor
5. Settings

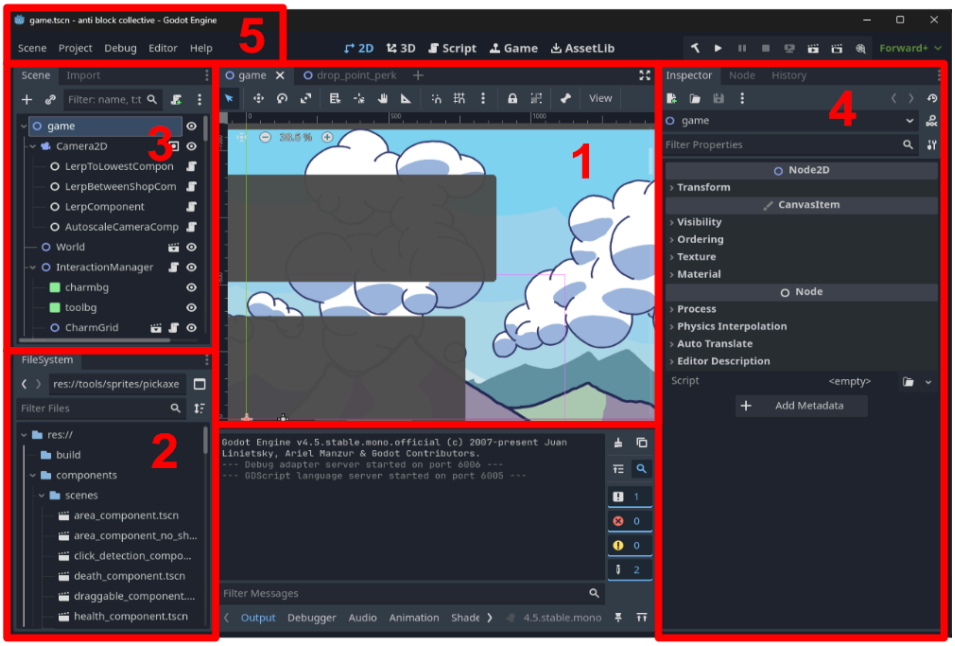

Figure 1: An overview of the main UI elements of the Godot Engine. Screenshot captured by the authors. [3]

2.1.3 Overview of Prominent Game Engines

The modern game engine landscape is dominated by a small number of general-purpose platforms, with Unity and Unreal Engine collectively accounting for nearly 80% of all games released on Steam in 2024 and over half of all units sold (Figure 2). Godot, [3] GameMaker, and Ren'Py account for a further 11% of releases, while the remaining titles are built on proprietary or niche engines. This concentration reflects a broader industry trend: the overhead of maintaining a bespoke engine has become prohibitive for most studios, pushing development toward shared platforms with large ecosystems.

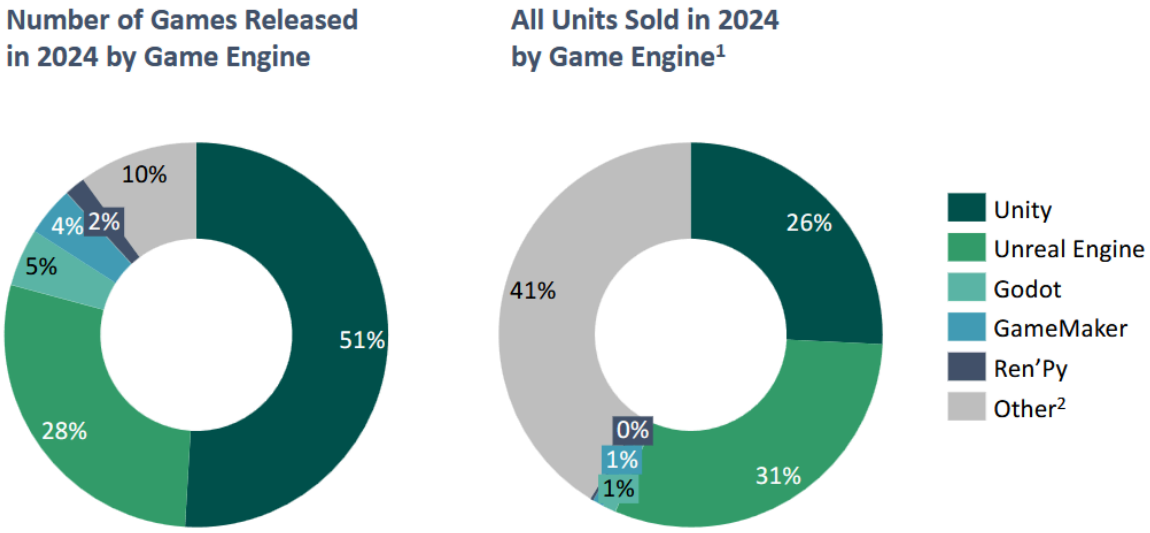

Figure 2: Game engine market share for games released on Steam in 2024, by number of releases (left) and units sold (right). Source: Sensor Tower, 2025 [15].

Despite sharing the same general-purpose positioning, these engines differ substantially in architecture, target audience, and the kinds of games they are best suited for. Table 1 summarizes the most relevant dimensions.

Feature	**Unity**	**Unreal Engine**	**Godot**	**Roblox Studio**
Language	C#	C++ / Blueprints	GDScript / C#	Luau
Rendering	Built-in / URP / HDRP	Nanite / Lumen	Forward / Clustered	Fixed pipeline
Physics	PhysX / custom	Chaos	Godot Physics / Jolt	PGS solver
License	Proprietary	Proprietary	MIT	Proprietary
Target scope	AAA, indie	AAA	Indie, open source	Platform games only
Multiplayer	Netcode package	Built-in	High-Level API	Native, server-hosted
AI tooling (2026)	First Party MCP (Beta)	Limited plugins, Community MCP	Community MCP	First Party MCP (Beta)

Table 1: Comparison of prominent game engines across key dimensions.

Unity (Figure 3) holds 51% of new releases on Steam, making it the most widely adopted engine in the industry. It uses an entity-component-system architecture with C# scripting, and its extensive Asset Store dramatically reduces prototyping time for small teams. Unity targets a broad range of project types, including both AAA and indie, all benefiting from its mature cross-platform export pipeline. As of early 2026, Unity has announced AI-integrated tooling, though no public release is available [1], [7].

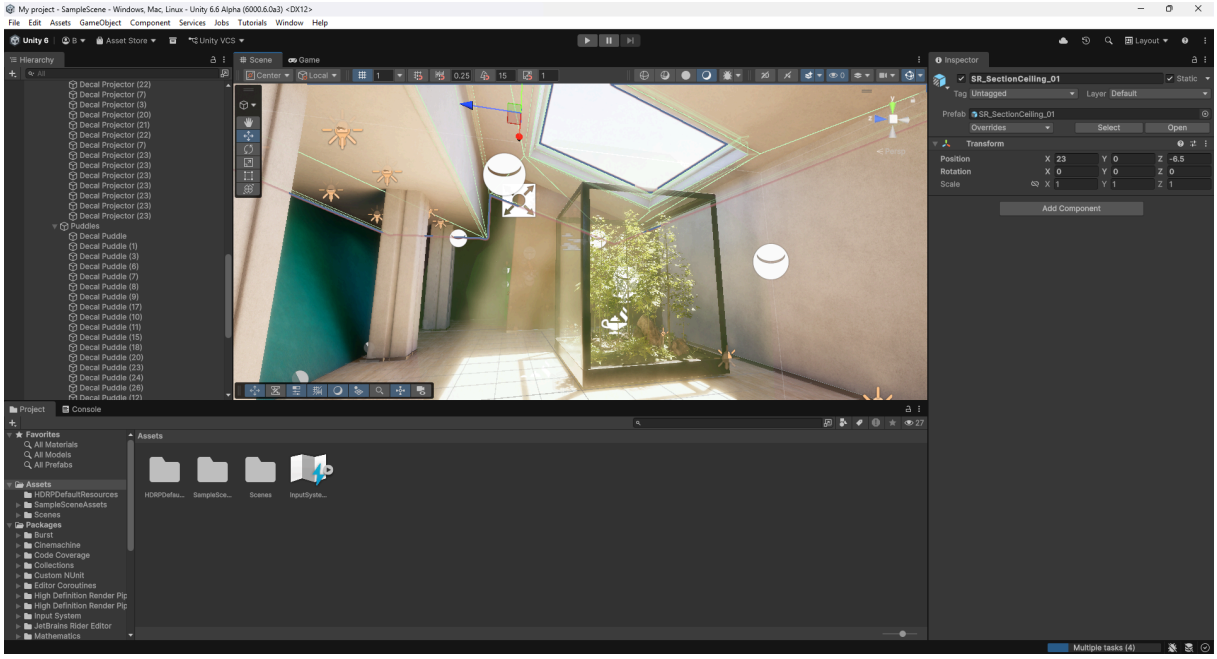

Figure 3: Image of the Unity Editor. Screenshot captured by the authors.

Unreal Engine (Figure 4) is the industry standard for photorealistic 3D rendering [2] and commands 31% of units sold despite accounting for only 28% of releases, reflecting its dominance in high-commercial-value titles. It is predominantly used in AAA development. A defining feature is Blueprints, a node-based visual scripting system that allows users to author game logic without writing C++.

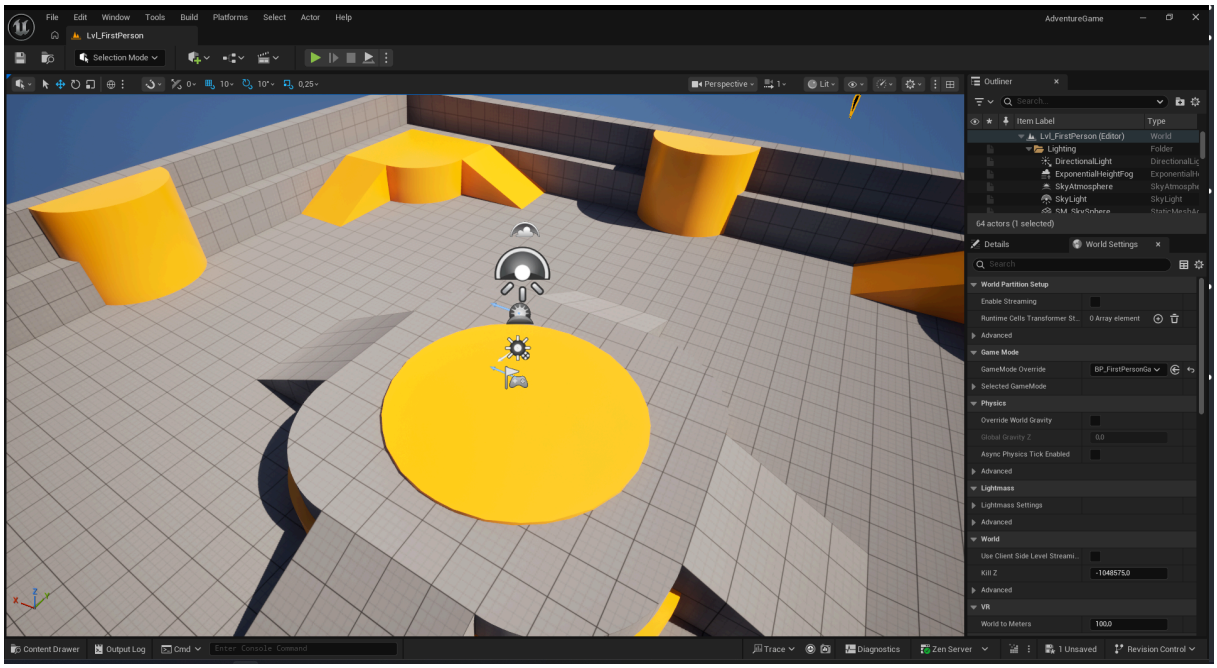

Figure 4: Image of the Unreal Engine Editor Screenshot captured by the authors.

Godot (Figure 5) has grown significantly as a powerful, MIT-licensed alternative to commercial engines [3]. Its scene system is built around reusable, nested nodes rather than entity-component pairs, and GDScript, a Python-like scripting language deeply integrated with the editor, enables

rapid iteration. Godot's open-source governance makes it particularly attractive for independent developers and studios wary of vendor lock-in.

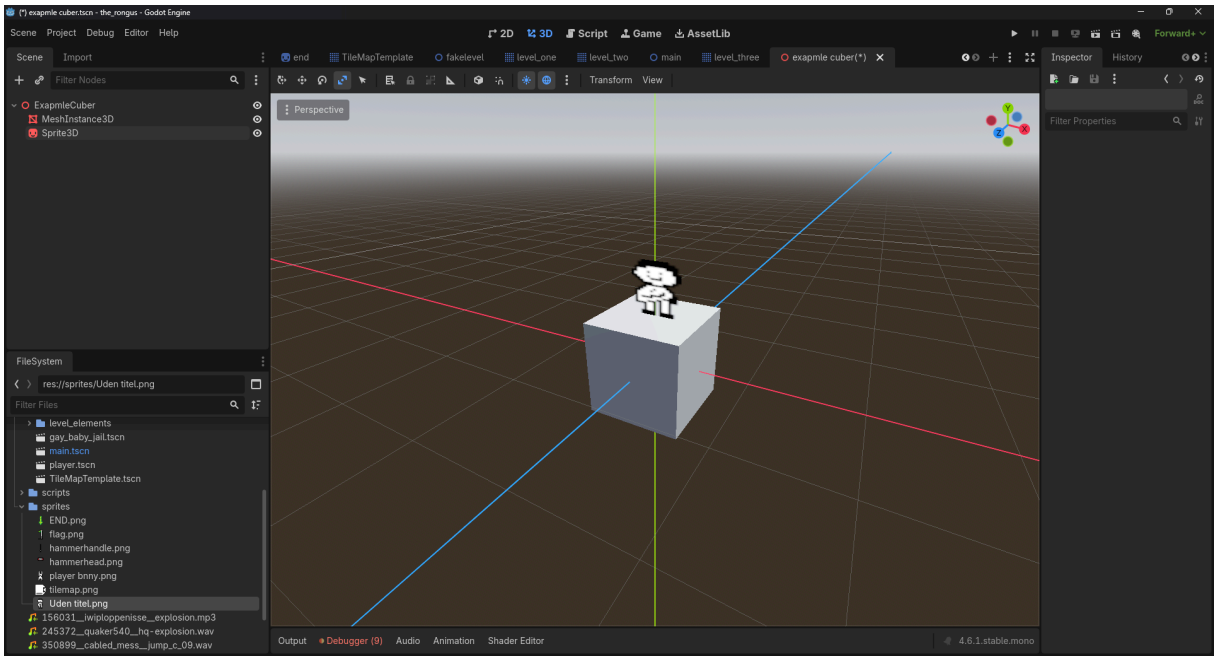

Figure 5: Image of the Godot Editor. Screenshot captured by the authors.

Roblox Studio (Figure 6) is included here not as an architectural peer to the engines above, but because it serves as a direct point of comparison in our evaluation: it has recently introduced AI-driven development tooling via MCP, making it relevant to the research question. As a development environment it is more accurately described as a closed platform for creating experiences within Roblox itself [16]. It abstracts away backend scaling, cross-platform deployment, and server hosting entirely, making it a fast path to distributing multiplayer games within a large existing social ecosystem. Game logic is authored in Luau, a performance-focused dialect of Lua [17].

Figure 6: Image of the Roblox Studio editor. Screenshot captured by the authors.

Browser-based engines represent a distinct category rather than a single product. Frameworks such as Babylon.js [18] and Three.js [19] run entirely within the browser, require no installation, and are accessible from any device with a modern web browser. They are particularly well suited to educational tools, interactive experiences, and cross-platform casual games, as no installation is required on the player's machine. The maturation of WebGL, WebAssembly, and the emerging WebGPU standard (covered in Section 2.4) has made browser-based engines viable even for 3D games with physics simulation at near-native performance. Babylon.js serves as the rendering foundation of Slop Engine and is discussed in detail in Section 3.

A pattern that is seen across aforementioned engines is the growing investment in AI tooling. Unity is developing native AI features, community extensions such as Unreal MCP [20] and Godot MCP [21] expose engine-specific tools to AI agents via the Model Context Protocol, and Roblox has shipped a beta MCP integration. This convergence motivates the present work: rather than adding an AI layer onto an existing engine, Slop Engine asks what an engine designed from the outset for AI-driven development would look like.

2.2 Large Language Models & Tool Use

2.2.1 Transformers and Early LLMs

The transformer architecture, introduced by Vaswani et al. [22], replaced recurrence with a self-attention mechanism that enabled efficient parallelization and scaling to substantially larger models. Early large language models trained on this architecture, such as GPT-2, functioned primarily as text completion systems capable of generating fluent continuations of a given input, but unable to reliably follow instructions or carry out specific tasks. This changed with the introduction of Reinforcement Learning from Human Feedback (RLHF), a fine-tuning approach in which models are trained to align their outputs with human preferences [23]. RLHF allowed models to interpret and execute user instructions rather than merely predicting statistically likely next tokens, a shift most visibly demonstrated by the release of ChatGPT. Subsequent research further improved model reasoning without additional training: Chain-of-Thought prompting, in which the model is instructed to produce intermediate reasoning steps as part of its output, was shown to substantially improve performance on complex tasks [24]. These developments are directly relevant to the present work, as Slop Engine's AI agents depend on both instruction-following and multi-step reasoning to carry out game development tasks.

2.2.2 Tool-calling

Although language models became increasingly capable at instruction-following and reasoning, early systems were still limited by the modality and environment in which they operated. Their inputs and outputs were primarily tokenized text, meaning that they could describe actions but could not directly browse the web, run code, inspect a file system, or interact with external software. This differs from later multimodal systems, which can condition on or generate other modalities such as images, audio, or video. However, even multimodal models still require external interfaces when they need to perform actions in the world or interact with software systems. Tool-use addresses this limitation by allowing a model to emit structured calls to external functions. Instead of only generating natural language, the model can request that an external tool be invoked, such as a search engine, calculator, code interpreter, database query, or file-system operation. The result of that tool call can then be returned to the model and incorporated into the next step of generation. One influential early example is Toolformer, which showed that language models can learn to decide when and how to call external APIs through a self-supervised data annotation procedure [25]. Rather than manually labeling every tool-use example, Toolformer inserted candidate API calls into text and retained those that improved the model's predictions. This demonstrated that tool use could be learned as part of language modeling rather than treated only as a hand-engineered control mechanism.

Modern tool-calling systems typically constrain this process through schemas, where each available tool is described by a name, parameters, and expected argument format. This is important because unconstrained language models may produce invalid, ambiguous, or non-executable calls. Gorilla, for example, showed that accurate API invocation is itself a non-trivial capability and that models benefit from targeted training or structured constraints when generating executable API calls [26]. However, tool-use also introduces new failure modes. A

model must not only understand the user request, but also decide whether a tool is needed, select the correct tool, construct valid arguments, interpret the result, and continue the task without losing context. Recent work on tool learning and structured tool use identifies these as persistent challenges for agentic systems [27], [28], [29]. For Slop Engine, this makes tool use both essential and risky: the agents need tools in order to modify and validate a game project, but every tool call also becomes a point where errors can enter the workflow.

2.2.3 From Reasoning to Agentic Workflows

The combination of instruction-following, reasoning, and tool use led to a more agentic view of LLM applications. In this setting, the model is not only producing a final answer, but repeatedly observing the state of an environment, reasoning about what to do next, taking an action, and incorporating the result into its next decision. A key paper in this direction is ReAct, which combines reasoning and acting in an interleaved loop [30]. Instead of generating a complete plan up front, the model alternates between reasoning traces and task-specific actions. The reasoning traces help the model track goals, update plans, and recover from unexpected observations, while the actions allow it to gather information or change the external environment. This is highly relevant to Slop Engine, where an agent may need to inspect existing project files, generate code, run validation checks, observe errors, and then revise its implementation.

More recently, Model Context Protocol (MCP) has attempted to standardize how LLM-based applications connect to external data sources and tools [31]. Before such protocols, each integration between a model and an external service often required custom interaction code. MCP instead defines a client-server architecture where compliant hosts can discover and invoke tools exposed by compliant servers, reducing the integration overhead for agentic applications and making it easier to connect models to development environments, documentation, databases, and project-specific tools.

These developments have converged in agentic coding tools where systems such as Cursor [32], GitHub Copilot [33], and Claude Code [34] move beyond simple code completion by giving the model access to project context and, in some cases, the ability to edit files, run terminal commands, execute tests, and iterate on its own output. Benchmarks such as SWE-bench capture this shift by evaluating whether models can resolve real software engineering issues from existing repositories rather than only generate isolated snippets [35]. This progression from text prediction to instruction-following, reasoning, tool use, and finally agentic workflows forms the technical background for Slop Engine. The engine assumes that an LLM can operate as a semi-autonomous development agent inside a constrained game-development environment. Its core challenge is therefore not simply whether a model can generate code, but whether it can reliably plan, act, observe, and revise across multiple steps while remaining grounded in the rules and structure of the target game engine. This raises a further architectural question: whether such workflows should be handled by one general-purpose agent, or divided across several specialized agents with narrower responsibilities.

2.2.4 Multi-Agent Architectures

Agentic workflows can be implemented either as a single general-purpose agent or as a system of multiple cooperating agents. A single-agent architecture is simpler, since one model instance

handles planning, context gathering, tool use, implementation, and evaluation. However, this also places many different responsibilities into the same context window. For complex software tasks, this can increase the risk of context overload, especially as longer contexts have been shown to make models less reliable at retrieving relevant information from the middle of the prompt [36].

Multi-agent architectures address this by decomposing a task into specialized roles. Different agents can be assigned responsibilities such as planning, implementation, review, testing, or tool-specific execution. Masterman et al. describe this as part of the broader landscape of emerging agent architectures, where systems combine reasoning, planning, memory, and tool use in different organizational patterns [37]. ChatDev is a concrete example in software development, framing the process as a simulated company where agents with different roles communicating to produce software artifacts [38].

Slop Engine follows this idea of role-based decomposition, but adapts it to game development. Constructing scenes, writing gameplay scripts, generating assets, creating user interface elements, and testing runtime behavior each require different context and different tools. Rather than assigning all of this to one agent, Slop Engine uses a central orchestrator that delegates work to specialized subagents for scripting, scene manipulation, asset generation, user interface construction, and testing. This is closer to orchestrated specialization than to Liu et al.'s cross-reflection pattern, where agents primarily critique each other's plans and reasoning [39]. The goal is therefore not unrestricted autonomy, but to narrow each agent's responsibility and align its prompt and tools with a specific part of the game-development workflow.

2.3 AI in Game Development

The term "AI in game development" can refer to two different uses of artificial intelligence. The first is AI *inside* games, such as pathfinding, enemy behavior, procedural generation, or narrative systems. This form of AI has existed for decades and includes examples ranging from simple finite-state machines like the ghosts in Pac-Man [40], to more recent examples of reinforcement learning like the robot enemies in Arc Raiders [41]. The second is AI *for* game development, where artificial intelligence is used as a production tool to help developers write code, create assets, debug systems, or automate parts of the development workflow.

This paper focuses on the second meaning: the use of Large Language Models (LLMs), tool-calling systems, and agentic workflows as part of the game development process itself.

2.3.1 From Chat Assistants to Engine-Integrated Assistance

Early LLM-based coding assistance was usually mediated through chat interfaces. A developer could ask for code, explanations, or debugging help, but the model only had access to the context that the user manually provided. This made the workflow useful for isolated snippets or conceptual help, but limited for larger software projects where relevant information may be spread across many files. A major step forward came with IDE-integrated tools such as GitHub Copilot [33] and Cursor [32]. These tools place the model directly inside the development environment, allowing it to use local project context such as the currently open file, surrounding code, open files, and broader workspace or codebase references. This reduces the need for

developers to manually assemble context in an external chat interface, and moves AI assistance closer to an IDE-native pair-programming workflow [42], [43].

For game development, however, IDE integration only addresses part of the problem. Game development is not only scripting. It also involves scene composition, asset pipelines, animation, physics, networking, level design, user interface design, and engine-specific workflows. Therefore, an AI system for game development must do more than generate source code. It must also understand how the target engine represents game objects, scenes, resources, and runtime behavior. This creates a mismatch between the way current LLMs operate and the way many game engines are used. LLMs primarily interact through text and structured tool calls, while engines such as Unity [1], Godot [3], and Unreal Engine [2] are centered around graphical editors. Human developers use these editors to manipulate scenes, inspect hierarchies, configure components, author animation graphs, place assets, and test behavior visually. For an LLM-based assistant, these graphical workflows are difficult to access directly as the model cannot naturally click through menus, inspect visual hierarchies, or manipulate objects in the same way a human developer can.

Code-first, browser-native frameworks such as Babylon.js [18] and Three.js [19] avoid much of this mismatch. Both allow developers to construct interactive 3D experiences through JavaScript or TypeScript code. This makes them accessible to language-model agents, because the agent can interact with the project using the same textual interface as a human developer: reading files, modifying source code, and running the project. A different response is to expose editor functionality through tools. Model Context Protocol (MCP) provides one standard mechanism for connecting AI systems to external tools and data sources [31]. In the context of game engines, MCP-style integrations can expose engine-specific operations as callable tools, allowing an agent to perform tasks that would otherwise require GUI interaction. Official implementations such as Unity AI [7] and Roblox Studio MCP [6], as well as community projects such as Unreal MCP [20] and Godot MCP [21] illustrate this direction by exposing parts of the editor workflow as structured actions. For example, an Unreal integration may provide tools for working with Blueprints, while a Godot integration may expose scene or script operations.

These approaches represent two different ways of making game development accessible to AI systems. Code-first frameworks make the project itself textual, while MCP-style integrations translate selected editor operations into structured tool calls. Slop Engine takes the second idea further by designing the development environment around agent access from the beginning. Rather than requiring an AI assistant to operate through a human-oriented graphical interface, Slop Engine exposes scene manipulation, script generation, asset management, and validation as first-class structured operations. This makes the engine a useful case study for evaluating whether deeper integration between an AI agent and a game engine can improve rapid prototyping workflows.

2.3.2 Current Limitations and the Road Ahead

Despite these advances, current AI agents still struggle with several parts of game development, including tasks involving scene composition, visual feedback, shader authoring, animation setup, user interface layout, and interactions between code and engine-specific assets. GameDevBench [44] evaluates this problem directly through 132 game development tasks in the Godot engine, derived from real web and video tutorials. GameDevBench is relevant because it tests agentic

development in a setting where programming is only one part of the task. Agents must also work with sprites, animations, shaders, scene hierarchies, and visual output. The benchmark is therefore more demanding than conventional software engineering benchmarks such as SWE-bench: its tasks require over three times as many lines of code and file changes on average compared to prior software development benchmarks. The results show that current agents are still far from reliable. The best agent solved 54.5 percent of tasks overall, and performance was uneven across task types. Gameplay-oriented tasks had a higher success rate than visually demanding tasks, with success rates dropping from 46.9 percent on gameplay-oriented tasks to 31.6 percent on 2D graphics tasks [44]. This suggests that current agents are better at tasks where correctness can be inferred from code structure, tests, or runtime errors than at tasks where correctness depends on visual judgement. A movement script can often be checked through behavior or test conditions, while a shader, sprite animation, or user interface layout may require the agent to inspect whether the result looks correct. Encouragingly, GameDevBench also shows that performance improves when agents receive multimodal feedback. Giving agents access to screenshots and video captures of the running game, so they can see the visual result of their changes, consistently raised success rates, with the largest reported improvement raising Claude Sonnet 4.5 from 33.3 percent to 47.7 percent [44]. This suggests that as the vision capabilities of frontier models continue to mature, the gap between agent performance on logic tasks and visual tasks is likely to narrow, provided that agents are given access to visual observations of the running game. For AI-assisted game development systems, this makes visual feedback an important design consideration: structured engine state can expose objects, scripts, and errors, but screenshots or video may be necessary for evaluating layout, camera framing, animation, and user interface readability.

2.4 Web Technologies for 3D

The web was not originally designed as a platform for complex real-time 3D applications. Early websites were primarily document-based, and interactive media required either limited browser APIs or external plugins. As browsers became more capable, developers increasingly wanted to build richer interactive applications, including games, directly on the web. This section outlines the technical developments that made browser-native 3D game development possible and explains why these developments are relevant to Slop Engine.

2.4.1 Early days

Early attempts at 3D on the web included the Virtual Reality Modeling Language (VRML) [45], [46], introduced in the 1990s as a way to describe interactive 3D scenes for the web. However, VRML depended on browser plugins and never became a general-purpose foundation for web games. Flash [47] later became the dominant platform for interactive web content, including many browser games, but it was also plugin-based and proprietary. These plugin-based approaches made interactive content possible, but they sat outside the standard browser platform and created problems around portability, performance, security, and long-term maintainability. WebGL changed this by bringing hardware-accelerated 3D graphics into the browser without requiring plugins. WebGL 1.0 was introduced by the Khronos Group in 2011 and exposed OpenGL ES 2.0-style rendering through the HTML `<canvas>` element [48]. This allowed web applications to render real-time 3D graphics using the browser's native graphics

stack instead of depending on external runtimes and was significant because it made the browser itself a viable target for real-time 3D rendering. Developers could now build interactive 3D applications that ran across platforms using only standard web technologies such as HTML, JavaScript, and GPU-accelerated rendering through WebGL. For Slop Engine, this matters because the system depends on the browser being capable of hosting not only a user interface, but also the runtime and rendering environment of the game itself.

2.4.2 High-level frameworks

WebGL is a low-level graphics API. Developers working directly with WebGL must manage details such as shaders, buffers, draw calls, textures, and rendering state. This gives developers control, but it also makes development verbose and error-prone. As a result, higher-level frameworks emerged to make browser-based 3D development more accessible. Three.js, first released in 2010, became a widely used JavaScript library for creating 3D graphics on the web [19]. It abstracts many low-level WebGL details behind concepts such as scenes, cameras, meshes, materials, and lights. This made it easier to build 3D visualizations, interactive experiences, and browser-based games without writing raw WebGL code. Babylon.js, initially released in 2013, took a more engine-oriented approach [18]. While Three.js is commonly described as a 3D graphics library, Babylon.js is closer to a full browser-native game engine. It includes higher-level systems for scene management, cameras, materials, physics integration, audio, input handling, asset loading, and WebXR support. Its success was significant enough that Microsoft later supported it as one of its open-source projects [18], [49], [50], [51]. This distinction is important for Slop Engine because a 3D graphics library such as Three.js provides the rendering foundation, but a game engine needs additional abstractions for building and managing game worlds. Babylon.js demonstrates that these engine-level abstractions can exist directly inside the browser, using web technologies rather than a separate native editor.

2.4.3 Native and Physics

WebAssembly (WASM) was introduced as a low-level binary instruction format for the web. It allows code written in languages such as C, C++, C#, or Rust to be compiled into a format that can execute efficiently inside the browser [52]. Rather than replacing JavaScript, WebAssembly complements it by allowing performance-critical systems or existing native libraries to be reused in web applications. This was important for browser-based game development because games often depend on computationally demanding systems such as physics, pathfinding, procedural generation, compression, audio processing, and asset pipelines. With WebAssembly, developers can bring mature native libraries into the browser while still integrating them with JavaScript or TypeScript application code.

A clear example is Babylon.js' integration of Havok Physics. Havok has been used as a physics engine in many commercial games since the late 1990s [12]. In Babylon.js 6.0, Havok became available through a WebAssembly-based plugin together with a redesigned physics API [49]. Babylon's documentation describes the Havok engine itself as a WebAssembly module responsible for the physics calculations, while the Babylon plugin integrates it into the engine [13]. This shows how WebAssembly allows browser-native engines to incorporate performance-sensitive systems that would previously have been associated with native game engines. WebAssembly matters because it weakens the boundary between browser-based and native development. A

browser-native engine can still use high-performance compiled modules where needed, while preserving the accessibility and deployability of the web platform.

WebGPU is the next generation graphics and compute API for the web. It was proposed through the W3C GPU for the Web Community Group in 2017 as a way to expose modern GPU capabilities to web applications [53]. Unlike WebGL, which is based on OpenGL ES, WebGPU was designed around concepts from modern native GPU APIs such as Vulkan, Metal, and Direct3D 12 [54]. WebGPU provides a more explicit and modern interface to the GPU than WebGL. It supports both rendering and general-purpose GPU computation, including compute pipelines. This makes it relevant not only for graphics, but also for workloads such as GPU-driven simulation, data processing, and potentially machine-learning inference in the browser. Browser support has matured significantly. Chrome and Edge shipped initial WebGPU support in 2023, and by late 2025 WebGPU was supported across the major browser engines [55], [56]. This is relevant because Babylon.js already supports WebGPU as a rendering backend [18], and Slop Engine uses Babylon.js as part of its browser-native architecture, accelerating the rendering of the scene further.

Together, WebGL, high-level 3D frameworks, WebAssembly, and WebGPU show that the browser has become a credible platform for real-time 3D applications. This is central to Slop Engine's design. Because the engine runs in the browser and is driven primarily through web technologies, an AI agent can interact with the project through textual code, structured tools, runtime logs, and browser-based visual feedback. The web platform therefore does not only make Slop Engine easier to distribute, it also makes the engine more accessible to agentic development workflows.

3 System Design

Before describing the concrete architecture, it is useful to restate the motivation behind Slop Engine's design. The system is not intended to be a conventional game engine with a chat interface added on top, but an experiment in making game-engine functionality directly accessible to an AI system. In Slop Engine, this AI system is explicitly multi-agent: a central orchestrator interprets the user's request and coordinates a set of specialized subagents responsible for scene construction, scripting, user interface, asset generation, and testing. The central idea is that rapid prototyping involves more than writing code. It requires creating scenes, configuring physics, attaching behavior, adding UI, testing runtime behavior, and revising the result. Slop Engine therefore exposes these operations as structured tools, allowing agents to act on the engine state rather than only describe changes or generate standalone code. The multi-agent design reflects the fact that these tasks are heterogeneous: constructing a scene, writing gameplay logic, generating assets, and validating runtime behavior require different context, different tools, and different kinds of feedback.

This approach is motivated by the limitations of external AI assistance. When an agent works outside the engine, it must infer scene state, asset structure, and runtime behavior indirectly. In Slop Engine, the assistant can inspect and modify the project through engine-defined operations, while the orchestrator-subagent architecture divides the work across narrower roles. The design goal is to evaluate whether deeper integration between a multi-agent AI system and a game engine can make natural-language game prototyping more reliable and efficient, while also revealing the trade-offs introduced by tool use, agent orchestration, and context management.

3.1 Architecture Overview

The name “Slop Engine” was chosen as a reference to the nature of AI-authored code work, often referred to as “slop.” [57]

Slop Engine is built as a single-page web application divided into two main layers: the browser editor and the backend API layer. The browser editor handles user interaction, editor state, scene rendering, and the visual interface around the Monaco-based code editor. It is built with TypeScript [58], SolidJS [59], Babylon.js [18], Havok [13], Tailwind CSS [60], and Vite [61]. The backend API layer handles the AI-facing parts of the system, including prompt construction, API calls, and streaming responses back to the frontend. It runs using the Bun runtime and uses ElysiaJS [62] as a backend together with the Vercel AI SDK for LLM requests. Slop Engine uses the SolidJS framework to create the website [59]. This selection was primarily driven by performance gains over React [63], alongside syntactical similarities, which is especially relevant for a game engine context in which it is necessary to reserve as much performance for the engine itself as possible.

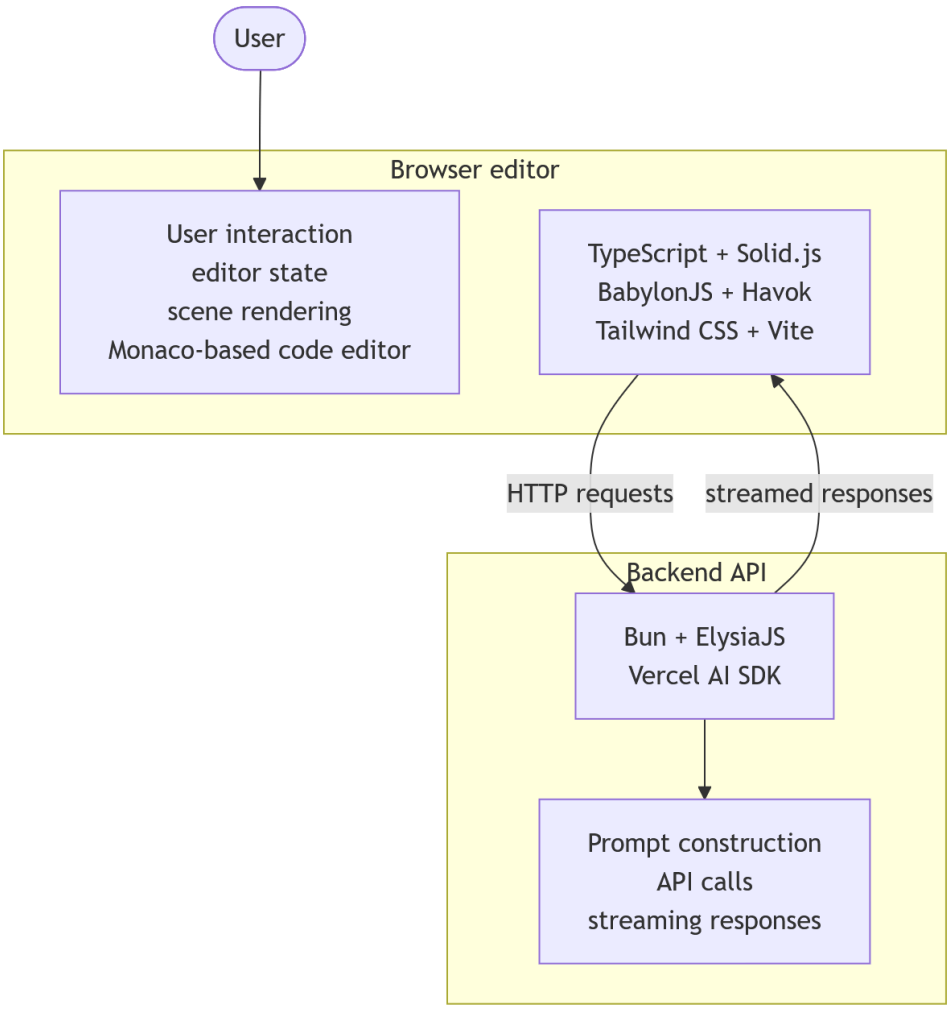

Figure 7: High-level architecture of Slop Engine, showing the browser editor and backend API layer.

3.2 Browser Editor

3.2.1 Interface

The interface of Slop Engine is split into resizable panels, inspired by other graphical game engines like Unity [1], Godot [3] and Roblox Studio [16]. As can be seen in Figure 8 which shows the default layout of the engine, panels are split into 6 sections: Toolbar, AI Assistant, Scene view / Script view, Console / Asset browser, Scene tree, and Properties. Additionally, the user can select a "Vibe" mode, seen in Figure 9 layout where only the AI assistant, a stripped down Toolbar and scene view are displayed.

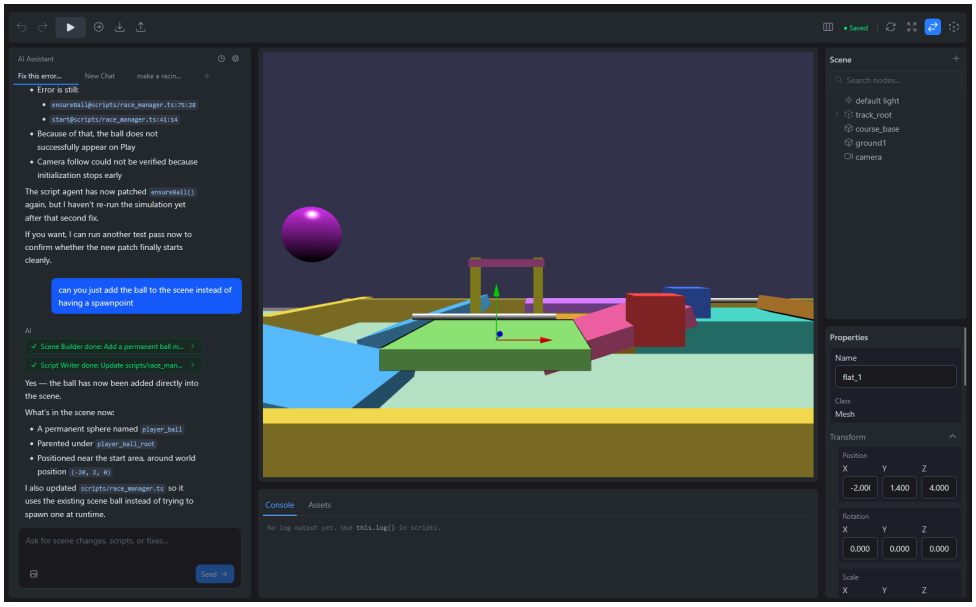

Figure 8: Default interface of Slop Engine.

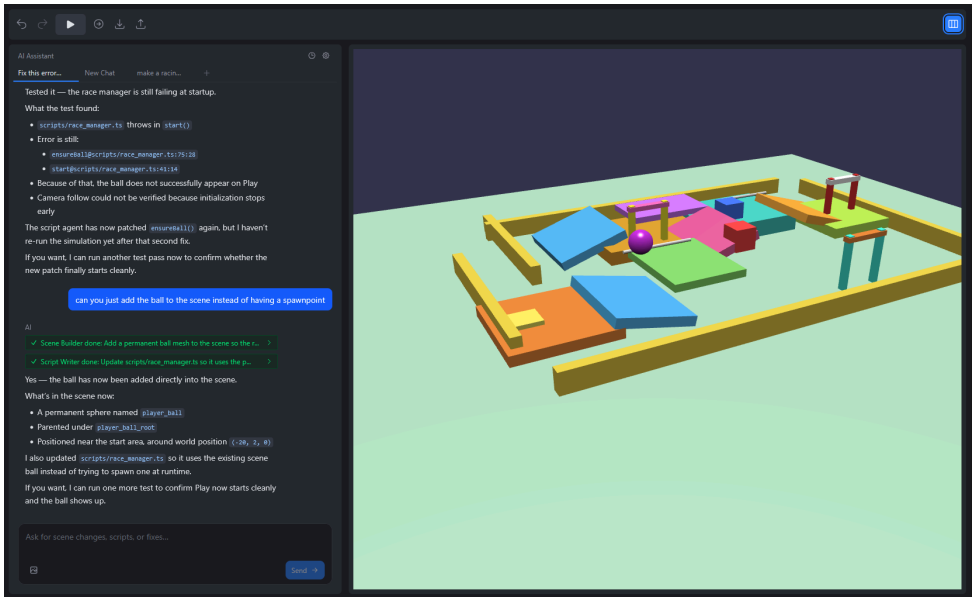

Figure 9: Vibe mode interface of Slop Engine.

The Toolbar is at the top of the interface and provides controls for undoing and redoing actions, playing and pausing the game simulation, resetting the project, importing and exporting scenes, and toggling Vibe mode. In the default interface, the toolbar also shows the autosave status and gizmo selectors[1] for interacting with scene objects. The AI Assistant occupies the left column. It provides a text input field for prompts, and supports image attachments. The chat view streams responses from the AI back to the user, displays the results of tool calls, and allows inspection of what subagents are doing. Subagent responses are streamed live alongside those of the main agent, including detailed transcripts of subagent work. Provided to the main orchestrator agent is a special "planning" mode, that allows the agent to provide follow-up questions to the user, ensuring that it is clear on how the user wants their game to be. The chat interface supports tabs, ensuring that the user can easily swap between multiple chats while preserving history. When a tab is closed, the chat can still be found through the chat history screen. The AI assistant features a settings screen, where the user stores their API credentials, sets their provider and the specific AI model for each agent. Slop Engine supports three providers: Azure OpenAI (Microsoft foundry) [64], OpenRouter [65] and Google AI Studio [66], while being expandable through use of the AI SDK. See Figure 10 for provider configuration interface.

Figure 10: Cutout of the Slop Engine provider settings screen.

[1] A gizmo is an interactive visual control shown in the scene view that lets users manipulate objects directly, for example by moving, rotating, or scaling them along visible axes.

Focusing on the Scene view in the middle column, is a canvas that renders the current Babylon.js scene. It allows the user to fly around the scene, select objects and move objects using gizmos[1]. When the game simulation starts, the scene view becomes the game area and the default editor behavior is disabled. Once the simulation stops, the editing position and context is restored. The Console is located at the bottom of the center column, displaying runtime errors and logs generated by user code. In the case of runtime errors, a button appears which automatically opens a new conversation with the error as a prompt. An alternative tab, Assets, displays the project's files and assets, including scripts, prefabs, images and 3D models. Controls are provided to make new folders, scripts and importing assets. Double clicking images, prefabs or 3D will insert them into the scene, while double clicking a script opens the script view. Located alongside the Scene View is the Script View, which replaces the scene when a script is opened. It is implemented using the Monaco Editor [67], which is the bare-bones script renderer that Visual Studio Code [68] uses. The editor has the language server for TypeScript installed, along with types for the scripting API. Bundling a language server allows the user to edit code with full IntelliSense and error highlighting [69], improving the Developer Experience (DX) of the script viewer significantly. The Scene Tree occupies the upper portion of the right column and displays the hierarchical structure of the current scene, including nodes, their children, and their types. A search field at the top enables for searching of specific nodes, indexed by name, and also contains a button allowing the addition of new nodes of various types to the scene. At the bottom of the right column, the Properties panel displays the editable properties of the currently selected nodes. The properties shown, depend on selection context. For example, a light node does not expose the same properties as a mesh node will. Properties available for the mesh node, which is most common, include: name, transform, scripts, physics, rendering and material.

3.2.2 Editor State and Scene Representation

Slop Engine separates editor state from scene state. The Babylon.js scene graph is the source of truth for the actual game world: meshes, lights, cameras, transforms, materials, physics settings, and parent-child relationships live on Babylon objects. The editor state stores information about how the editor is currently interacting with that scene, such as which nodes are selected, which panels should update, and when changes should be saved or added to the change stack. This distinction is important because the engine combines two different programming models. Babylon.js scene objects are mutated imperatively: changing a mesh position or material means modifying the object directly. SolidJS, however, updates the user interface through reactive state. Slop Engine therefore needs a bridge between imperative scene mutation and reactive editor views. That bridge is implemented through a custom hook called `useEditorEngine`. In SolidJS, a hook is a reusable function that encapsulates stateful logic and exposes it to components [63]. `useEditorEngine` provides shared accessors and actions for the editor, including the active Babylon scene, the currently selected nodes, undo handling, autosave scheduling, and update signals for derived UI panels. It acts as the coordination layer between the 3D scene, the hierarchy tree, the properties panel, the viewport gizmos, and the AI tools. The most important editor-state field is `selectedNodes`, which stores the nodes currently selected by the user or by an editor operation. Selection can originate from several places: clicking an object in the viewport, selecting an item in the hierarchy tree, or executing a scene operation through the AI tools. Regardless of where the selection change originates, it is routed through shared editor

actions such as `setSelectedNode`, so that the hierarchy panel, properties panel, and viewport gizmos remain synchronized. Because Babylon.js objects are mutated directly, not every change automatically creates a new SolidJS state value. Slop Engine uses `nodeTick` as an explicit update signal for this case. When a node is modified in place, the editor increments `nodeTick`, causing dependent views to recompute their derived state. For example, if the properties panel changes a mesh transform directly, `nodeTick` tells the hierarchy tree, property readouts, and gizmos that the underlying scene object may have changed. This keeps the UI reactive without duplicating the entire scene graph into a separate application-state model.

3.2.3 Runtime & Physics Simulation

Slop Engine internally keeps one Babylon scene alive at all times, meaning that the scene the user sees while editing, is the same as the one during the simulation. This was done for simplicity of implementation, as well as ensuring minimal latency for turning on and off the simulation. In order to separate the editor from the simulation, flags for enabling physics and scripts are stored as metadata on nodes.

The simulation life-cycle starts when the user first presses play, or it is triggered through an agent tool-call. First a snapshot of the scene is captured, to be able to restore the state of the scene after play is terminated. Afterwards, a scan is done for the meshes in the scene to check for a `physicsEnabled` flag. If true, a `PhysicsAggregate` with a convex hull shape is added to the mesh. Finally, a runtime camera is configured. The camera used while editing is a special Babylon type that allows for flying around using `WASD` controls. During runtime, more control is needed over the camera's position and behavior. Therefore the editor's camera is removed and a new blank camera, controllable by scripts, is created. Lastly, a new `ScriptRuntime` instance is created and started. The stop path does the reverse: stop scripts, dispose all runtime physics aggregates, wait one animation frame, restore the captured snapshot, switch back to the editor camera, and clear the playing flag. Restoring is what returns the scene to the exact pre-simulation stored state.

The `ScriptRuntime` is what allows for objects to have behaviors. On start it listens to inputs from the canvas, creates a `RuntimeWorld`, collects every node with `metadata.scripts`, compiles each script using `sucrase` from IndexedDB blobs, validates any `nodeType` constraint, instantiates the script, wires in scene/input/camera/world access, and calls start on every instance. After that it registers collision callbacks and subscribes to the Havok collision observable through the `RuntimeWorld`, then installs an `onBeforeRender` observer that ticks every script once per frame. The tick path updates `deltaTime` and time, then calls update on each live script. Objects created at runtime are managed by `RuntimeWorld`. Scripts can spawn primitives, clone nodes, spawn prefabs, and optionally add physics bodies. Additional tracking is done here, in order to restore the pre-simulation state perfectly. `RuntimeWorld` also listens to Havok collisions and dispatches start/end events back to registered script callbacks. When play ends, `disposeAll` removes the collision observer, disposes runtime physics aggregates and every runtime-created node, and tears down the GUI manager.

3.2.4 Scripting API

In Slop Engine, a custom Scripting API is provided to developers and the scripting agents. Slop Engine scripts are written in TypeScript, but target a small engine-specific API rather than the full Babylon.js API directly. This API is declared in `api.d.ts` and exposes globals such as `Script`, `Input`, `SceneNode`, `Scene`, `Camera`, and `GUI`. A script exports a class extending `Script`, and may implement lifecycle methods such as `start`, `update`, and `destroy`. Scripts are attached directly to nodes, mimicking the scripting implementation of Unity [1], and Godot [3].

```
export default class extends Script {
  speed = 2

  start() {
    this.log('Hello from', this.node.name)
  }

  update() {
    this.node.rotation.y += this.speed * this.deltaTime
  }

  destroy() {
    this.log('Goodbye')
  }
}
```

Listing 1: Generic script example showing the base methods all Scripts can implement

As can be seen in Listing 1 a script can implement three methods: `start`, `update` and `destroy`. The start method is run when the simulation starts, update is run every engine tick which happens once per frame, and the destroy is called on simulation end. We have assessed that this is the minimal interface necessary in order to represent the life-cycle of an object.

There are a couple of reasons for creating a custom API for Slop Engine. The first is that Babylon.js does not provide a Unity-style scripting system where arbitrary script files are attached to scene nodes and automatically executed through a standardized lifecycle. Scripting is done through lower-level extension mechanisms such as Behaviors, Observables, ActionManager, and node metadata. Behaviors are the closest built-in equivalent, since they can be attached to nodes, but they do not by themselves provide the lifecycle management and editor integration that Slop Engine requires. The other reason is to give both users and AI agents a narrower and more predictable interface for gameplay behavior. Instead of requiring the agent to reason about the complete Babylon.js API surface, scripts operate through a smaller set of concepts that match common game-engine patterns: nodes, lifecycle hooks, input, scene access, prefabs, raycasts, and physics helpers. The trade-off is that this custom API is not part of the model's pretraining data, so agents need explicit documentation and type feedback to use it reliably.

3.3 Backend API Layer

The backend API layer is responsible for connecting the browser editor to external AI providers. While the editor owns the active scene, project files, runtime state, and user interface, the backend coordinates model calls, builds prompts, selects providers, streams responses, and exposes server-side endpoints for operations that cannot run entirely in the browser. This separation is important because Slop Engine is browser-native: the game engine itself runs client-side, while the backend acts as the coordination layer between the editor and remote model providers.

3.3.1 Server Architecture and Streaming

Slop Engine's backend is implemented as a Bun server using ElysiaJS. It serves the built frontend from `dist` and exposes an `/api` route group for AI chat, subagent execution, script typechecking, scripting API lookup, asset generation, and benchmark instrumentation. The main user-facing endpoint is `/api/chat`. When a prompt is submitted, the frontend sends the chat history, selected node context, and model settings to the backend. The server converts the messages into the provider format, selects the configured model, builds the orchestrator system prompt, attaches the orchestrator tools, and calls `streamText` through the Vercel AI SDK. The response is streamed back to the browser so the user can see assistant output and tool progress while the request is still running. Subagents are handled through `/api/subagent`. This endpoint runs a complete subagent call with `generateText`, using a role-specific system prompt and a narrowed tool set. The result is returned to the orchestrator as text, tool calls, and a finish reason. This keeps the orchestrator as the user-facing coordinator while allowing implementation work to be delegated to scene, script, UI, asset, and testing agents. Supporting endpoints provide functionality used by the tools themselves. `/api/typecheck` validates scripts against the Slop Engine scripting API, `/api/lookup-scripting-api` retrieves relevant documentation from `api.d.ts`, and the asset-generation endpoints call the external image and 3D model providers. This keeps credential-dependent and long-running operations on the backend while preserving the browser editor as the owner of the live project state.

3.3.2 Provider Abstraction

Slop Engine currently supports three AI providers: Microsoft Foundry (formerly Azure OpenAI) [64], OpenRouter [65], and Google AI Studio [66]. These providers expose the same core feature-set within Slop Engine, including tool-calling, agent spawning, and image upload support. In the AI assistant settings, users can select their preferred provider, enter the required API credentials, and configure which models should be used for each type of agent. The provider abstraction was implemented using the Vercel AI SDK [70], which made it possible to integrate multiple LLM providers through a unified interface. This reduced provider-specific implementation complexity and made it easier to switch between providers during development. Microsoft Foundry was used during development because it was available through a free student subscription and reduced experimentation cost.

Although LLM providers are abstracted, asset generation providers are not. This decision was primarily driven by practical limitations and cost considerations. The Azure student subscription used during development did not provide access to the required models for image generation

or 3D model generation. As a result, Slop Engine uses NanoBanana through nanobananaapi.ai [71] for image generation and Tripo AI [72] for 3D model generation. These providers were selected because they offered accessible and cost-effective options for Slop Engine. Tripo AI provides a free tier with five model generation requests per month [72], while NanoBananaAPI offered image generation at a relatively low cost [71]. This made it possible to include asset generation functionality without significantly increasing development expenses.

In the original design of Slop Engine, asset generation was intended to be a major feature that would help make generated games feel more complete. However, during development, the scope of this feature was reduced. Slop Engine's direction shifted toward generating playable prototypes rather than complete games, which made advanced asset generation less central to the overall system. As a result, we prioritized reliable and affordable asset generation over a fully abstracted provider architecture for this part of the system.

3.3.3 Tool Calling Pipeline

For the LLM to interact with Slop Engine, it must be able to invoke engine functionality through structured interfaces rather than purely free-form text. This is handled through the tool-calling pipeline. As discussed in Section 2.2.2, tool calling allows a model to request that external functions be executed with structured arguments. In Slop Engine, these functions expose editor and engine capabilities such as reading the scene, creating meshes, writing scripts, attaching scripts to nodes, generating assets, running the simulation, and inspecting console output. The pipeline is split across three parts: the model provider, the backend API layer, and the browser editor. The model provider decides when to call a tool and with which arguments, while the backend defines which tools are available to each agent and coordinates model calls through the Vercel AI SDK. The browser editor owns the project state and runs the selected tool. Tools are defined using a JSON schema that describes its name, purpose, input fields, and required arguments. Listing 2 shows a simplified example of the `create_script` tool. This tool allows an agent to create or update a TypeScript script file in the project asset store, with the schema telling the model that the tool expects two things: a path to the script location, and the full source contents.

```
import { jsonSchema } from 'ai'

export const createScriptTool = {
    description:
        'Create or update a TypeScript script file in the project asset store.
The script should export a default class extending Script. Use forward-slash
paths like "scripts/rotate.ts".',
    inputSchema: jsonSchema<{ path: string; content: string }>({
        type: 'object',
        properties: {
            path: {
                type: 'string',
                description:
                    'File path for the script (e.g. "scripts/rotate.ts"). Use
forward slashes.',
            },
            content: {
                type: 'string',
                description:
                    'Full TypeScript source code for the script. Must export a
default class extending Script.',
            },
        },
        required: ['path', 'content'],
    }),
}
```

Listing 2: A tool definition from Slop Engine's tool definitions script

The tool is included in the model call through the Vercel AI SDK, where the tool description helps the model decide when the tool is appropriate to use, while the input schema makes the call executable by the surrounding system. This improves reliability as we can give the model specific recommendations on how, when and why to use a tool instead of AI having to infer usage directly from the input schema.

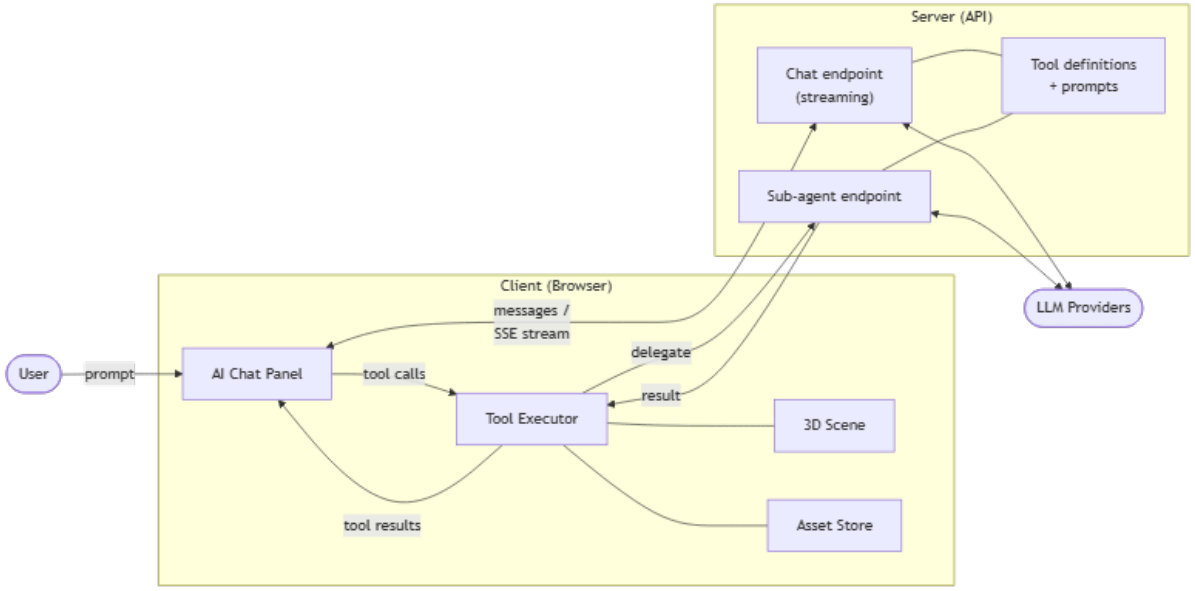

Figure 11: End-to-end architecture of the Slop Engine tool-calling pipeline.

Figure 11 shows the end-to-end flow. A user message begins in the browser editor and is sent to the backend through `/api/chat`. The backend builds the orchestrator prompt, attaches the orchestrator tools, and streams the model response back to the frontend. If the orchestrator calls `spawn_agent`, the frontend triggers a subagent request through `/api/subagent`. The backend then runs the selected subagent with its own prompt and tool set. When the subagent calls tools that affect the project, those calls are applied to the editor state, asset store, scene graph, or simulation through the frontend-side engine integration. The subagent result is returned to the orchestrator, which decides whether to continue delegating work or answer the user. See Section 8.4 for a sequence diagram of the tool-calling pipeline.

Therefore, the pipeline acts as the bridge between language-model reasoning and concrete engine operations. The backend does not itself contain the entire game engine, and the model does not directly mutate arbitrary browser state. Instead, the model produces validated tool calls, the backend coordinates agent execution, and the editor applies the resulting operations to the live project. This preserves the editor as the source of truth while giving the AI system structured access to the engine.

3.4 AI Integration

In this section we will discuss the architecture and implementation of the AI in Slop Engine. It describes how the AI agents interact with the engine internals such as scene manipulation, script generation, asset generation, and simulation control.

3.4.1 Orchestrator and Subagents

Slop Engine uses an orchestrator-subagent architecture in which a central agent interprets the user's request, forms an implementation plan, and delegates concrete development tasks to specialized agents. The system consists of one Orchestrator and five subagents: the Scripting Agent, UI Agent, Scene Agent, Asset Agent, and Tester Agent. This design follows the general

idea of role-based decomposition discussed in Section 2.2.4, but adapts it to the structure of a game engine rather than to a conventional software repository.

The motivation for this architecture is that game development tasks are heterogeneous. A request such as "make a simple breakout game" is not only a programming task. It also requires scene construction, object placement, physics configuration, camera setup, player input, gameplay logic, user interface elements, and runtime validation. Assigning all of these responsibilities to a single agent would require one prompt to contain every available tool, instruction, and implementation concern. This would increase context size and make the agent more likely to confuse responsibilities, use tools from the wrong domain, or spend tokens reasoning about irrelevant parts of the engine. The subagent design attempts to reduce this problem by narrowing each agent's scope and exposing only the tools relevant to its role.

The Orchestrator is the only agent that communicates directly with the user during a normal request. Its purpose is not to implement the game directly, but to coordinate the workflow. When a user submits a prompt, the Orchestrator interprets the requested game or edit, inspects the current project state when necessary, and decides whether the task is specific enough to execute. If the request is underspecified, it can ask follow-up questions through `askClarificationTool` and present a structured plan through `presentPlanTool`. This planning phase was added because natural-language game prompts often omit important design choices, such as visual style, difficulty, camera perspective, or control scheme. Rather than forcing subagents to resolve these details independently, the Orchestrator resolves them once before implementation begins.

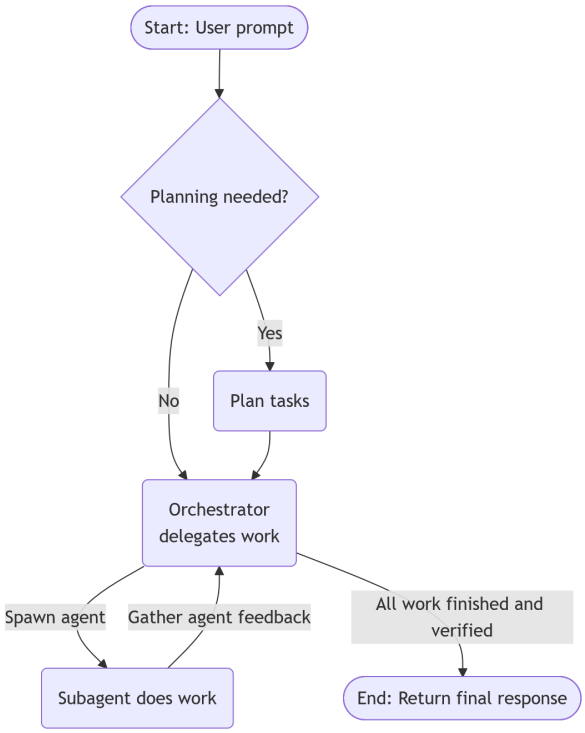

Figure 12: General workflow of the AI assistant.

The workflow is shown in Figure 12. After receiving a user prompt, the Orchestrator determines whether planning is required. It then decomposes the request into implementation tasks, delegates those tasks to one or more subagents, gathers their summaries, and decides whether further

work is needed. Once the implementation has been completed and validated, the Orchestrator returns a final response to the user.

The five subagent roles divide the game-development workflow into smaller responsibilities. The Scene Agent modifies the scene graph by creating nodes, placing meshes and prefabs, configuring transforms, and setting up lights. The Scripting Agent writes and edits gameplay scripts using the custom Slop Engine scripting API, including movement, collisions, scoring, spawning, camera behavior, and win or lose conditions. The UI Agent focuses on runtime interface elements such as score displays, timers, instructions, and win or lose screens. Scripting and UI agent have access to the same tools. Where they differ is their system prompts that help them understand their respective domains. The Asset Agent handles asset-oriented tasks, including generating images or 3D models, applying textures, and updating materials. Finally, the Tester Agent validates the result by starting the simulation, inspecting console logs, and running autonomous tests that apply timed inputs and observe scene snapshots. This division is primarily a prompt-design and tooling decision: each role receives a narrower prompt and a smaller set of tools matched to one part of the engine.

The agents are not peers in a free-form conversation. Instead, the architecture is hierarchical: the Orchestrator controls when each subagent is invoked and what task it receives. A typical request proceeds through a sequence such as scene construction, script generation, UI creation, optional asset generation, and testing. Each subagent returns a summary of its work to the Orchestrator, which then decides whether to invoke another subagent, request further changes, or report completion to the user. The main practical advantage of this hierarchy is context reduction at the level of each individual agent call. The Orchestrator maintains the broader conversation and user intent, while subagents receive narrower task descriptions, specialized system prompts, and only the tools relevant to their domain. A scripting task, for example, does not need to include asset-generation instructions, scene-editing rules, and the full testing procedure at the same time. This makes each model call more targeted and reduces the amount of irrelevant context inside each subagent prompt. The main cost is orchestration overhead. The Orchestrator is intentionally prevented from performing most implementation actions directly, so concrete edits are routed through subagents. Each subagent call requires its own prompt, context, tool definitions, and model response. Although specialization reduces the context required by an individual subagent, the combined workflow can consume more total tokens than a single monolithic agent. The design therefore depends on whether the benefits of specialization outweigh the additional coordination costs, which is evaluated later in the report (see Section 5).

3.4.2 Engine Tool Surface

The tool-calling pipeline described in Section 3.3.3 explains how tool calls are routed through the backend and frontend. This section instead describes what Slop Engine exposes through that pipeline. The tool surface is the set of engine capabilities made available to the orchestrator and subagents. It defines what the agents can observe, modify, generate, and validate inside a project. Tools are grouped by responsibility rather than exposed as one flat API, which is important because Slop Engine uses specialized subagents, each of which should only receive the tools relevant to its role. A scene agent needs operations for manipulating objects and hierarchy, while a scripting agent needs access to script files, typechecking, and the scripting API reference. Similarly, a testing agent needs simulation and console tools, while an asset

agent needs generation and material tools. Keeping these groups separate reduces prompt size, limits accidental misuse, and makes each agent's behavior easier to reason about.

The orchestrator has the smallest tool surface. It can inspect the scene through `get_scene`, delegate work through `spawn_agent`, ask follow-up questions through `ask_clarification`, and present implementation plans through `present_plan`. This is a deliberate constraint: the orchestrator coordinates the task, but most concrete implementation work is routed through specialist agents. The scene tool group exposes operations for editing the scene graph. These tools allow an agent to add meshes and lights, update existing nodes, delete nodes, create groups, change parent-child relationships, import assets, save prefabs, and apply multiple scene edits through `bulk_scene`. This makes scene construction possible without requiring the model to write Babylon.js code directly. Instead, it works through higher-level operations that match the concepts visible in the editor: nodes, transforms, hierarchy, meshes, lights, materials, and prefabs. The scripting tool group exposes operations for creating and maintaining gameplay scripts. These include listing scripts, reading existing script files, creating new scripts, editing or deleting scripts, and attaching or detaching scripts from scene nodes. The scripting and UI agents also receive `lookup_scripting_api`, which lets them retrieve relevant parts of the local scripting API reference instead of receiving the entire `api.d.ts` file in every prompt. It is important that they can retrieve this because Slop Engine's scripting API is project-specific and therefore not something the model can be expected to know from training data. The runtime and validation tool group lets agents test the current project. These tools include starting and stopping the simulation, waiting for a fixed amount of time, reading console logs, and running autonomous tests that apply timed inputs and collect observations. These tools are mainly used by the testing agent, but selected runtime tools are also available to scripting and UI agents so they can check whether their own changes compile and behave correctly. This gives the assistant a limited feedback loop: it can make a change, run the game, observe errors or state changes, and revise its implementation. The asset tool group exposes image, model, texture, and material operations. The asset agent can generate images, generate 3D models, list available assets, list image assets, apply or remove textures, update material properties, enable billboard rendering, delete assets, and create asset folders. These tools support visual prototyping by allowing the agent to create and apply simple art assets without leaving the editor workflow.

The complete tool surface is summarized in the appendix Section 8.5 in Table 10. The table shows that tools are not distributed evenly across agents. Instead, each agent receives a role-specific slice of the full engine API.

3.4.3 Context Management and Validation

Slop Engine intentionally uses a project-specific tool layer and scripting API (see Section 3.4.2 Section 3.2.4). These interfaces were created for the engine and should therefore not be assumed to be known by the language model in advance. This makes the system a useful test case for agentic development with unfamiliar APIs: the model cannot rely only on general programming knowledge, but must instead be grounded in the context that Slop Engine provides. To support this, each agent is given a dedicated system prompt that defines its role, scope, constraints, and available tools. A system prompt is the high-priority instruction set that frames the model's behavior before the user's request is added. In Slop Engine, the prompt explains what the agent is responsible for, which operations it may perform, which tools it can call, and what

assumptions it should avoid. The user's request is then interpreted relative to this agent-specific context. Prompt composition and tool-call formatting are handled through Vercel's AI SDK [70].

Each agent prompt follows a consistent structure: a role definition, a concise responsibility statement, a set of critical rules, a description of the tools available to the agent, and a short section of task-specific guidance. This structure improves predictability and preserves a clear separation between scene editing, scripting, user interface work, asset management, and simulation testing. The prompts for specialist subagents are kept static in order to reduce context churn and improve prompt caching efficiency. The orchestrator agent differs slightly in that it uses a partially dynamic prompt generated by `buildCoordinatorSystemPrompt(selectedNode)`. The `selectedNode` parameter refers to the currently selected node in the scene, enabling the user to refer to specific objects without repeatedly naming them. Context acquisition is handled through a small set of structured tools rather than through open-ended retrieval. The `getSceneTool` provides a JSON snapshot of the scene hierarchy and simulation state, giving the agents a current model of the environment. The scripting agents additionally rely on `lookupScriptingApiTool`, which retrieves the relevant section of the local scripting reference in `api.d.ts` through topic-based lookup. This provides the model a way to look up the specifics of API usage, without needing more complex retrieval methods. Not every tool is available to every agent, which is intentional as described in Section 3.4.2. Giving all agents access to all tools would increase prompt size, reduce specialization, and make it easier for an agent to perform actions outside its intended responsibility. Instead, tool access is scoped according to the role of each agent. This mirrors the broader design of Slop Engine: the model should have enough context and capability to complete its task, but not so much that unrelated responsibilities are mixed together in the same prompt. The available tools are summarized in the appendix Section 8.5 in Table 10.

This scoped tool design is also a form of validation. Each agent can only act through predefined operations, and those operations can be checked before being applied to the project. For instance, script changes can be grounded in the actual contents of existing files, scene changes can be grounded in the current scene hierarchy, and debugging steps can be grounded in console output rather than guessed from the user's description alone. In addition to contextual validation, Slop Engine supports runtime validation. The testing agent can start the simulation, inspect console output, and execute input-driven tests. This is important because game behavior is often difficult to verify through static code inspection alone. A script may compile but still fail to move an object correctly, trigger an event at the wrong time, or behave differently once physics and frame updates are involved. By allowing the agent to observe runtime behavior, Slop Engine moves closer to a closed-loop workflow where the agent can generate a change, run the project, inspect the result, and revise its output. This validation loop is especially important for agentic game development because the environment contains both code and state. Unlike ordinary code generation, where the output is often a single file or function, a game-development task may involve scripts, assets, scene hierarchy, simulation state, and visual output at the same time. Slop Engine therefore treats context management as part of the system architecture rather than as a prompt-engineering detail. The model is not expected to remember or infer the entire engine; instead, the engine provides targeted context and controlled tools at the moment they are needed.

3.5 Challenges & Trade-offs

The central trade-off in Slop Engine is context management. An agent can only act reliably if it understands the current project state, the available tools, and the engine-specific scripting API. However, giving the model too much context at once can also reduce performance. Long prompts increase latency and cost, and they can make it harder for the model to identify which information is relevant to the current task. This is related to the "lost in the middle" problem, where language models become less effective at using information placed inside long contexts [36] and creates a design tension. If the agent receives too little context, it may have to guess about the capabilities of Slop Engine, hallucinate objects, or use API methods that are not available. If it receives too much context, the prompt becomes noisy and the model may focus on irrelevant details. Instead, it uses a hybrid approach: the agent is given stable role instructions up front, while project-specific information is retrieved through tools when needed. Scene-editing tasks depend on knowing which objects exist, how they are nested, and which node is currently selected. For this reason, agents usually begin by calling `getSceneTool` before modifying the scene. This increases token usage, but it prevents a more serious failure mode, which would be acting on an outdated or imagined version of the scene. In a visual editor, such mistakes are especially costly because a wrong change can affect hierarchy, transforms, physics settings, or attached scripts at the same time.

The scripting API created another context-management problem. Slop Engine uses its own scripting layer rather than exposing the full Babylon.js API directly. This was done to make scripts safer, smaller, and more predictable for both the runtime and the agent. A conventional game engine API is often designed for experienced human developers who can search documentation, inspect types, and understand implicit engine conventions. An LLM, however, may combine partially remembered APIs from different frameworks or infer methods that sound plausible but do not exist in the local environment. Early versions of the Slop Engine scripting API followed more conventional game-engine patterns, including broader lifecycle hooks and more direct access to the underlying engine objects. This made the API flexible for humans, but unreliable for agents. The model frequently misused lifecycle methods, attached scripts to the wrong node types, or referenced properties from Babylon.js or general TypeScript examples that were not actually exposed for use in Slop Engine. These errors showed that a broad API increases the surface area for hallucination. Therefore, the API was narrowed iteratively. Ambiguous entry points were removed, type definitions in `api.d.ts` were made more explicit, and the scripting interface was shaped around common agent tasks. A possible alternative would have been to expose the full scripting reference in every agent prompt. This would give the model access to all available API details, but it would also increase prompt length for every task, including tasks that do not require scripting. Slop Engine instead provides a small snapshot of the API rather than the entire scripting API interface (which is several hundred lines of code), and includes a tool found `lookupScriptingApiTool`, which allows scripting and UI agents to retrieve only the relevant part of the local API reference when needed (See Section 3.4.3 for more). This architecture reflects a broader principle in Slop Engine, that context should be available but not always present. The agent should be able to inspect the scene, read scripts, query documentation, and check runtime logs, but these sources should be pulled into the context only when they are relevant to the current task. In practice, this makes the system more robust than a purely prompt-stuffed design while avoiding the opposite failure mode of leaving the agent under-informed.

4 Work Methodology

Development of Slop Engine followed an iterative, feature-driven workflow. We used a Kanban board in Linear [73] and divided the project into six phases at the beginning of the semester. This gave each phase a clear focus and helped structure the available development time. Phase 1 focused on the technical foundation, including project setup and basic scene rendering with Babylon.js. This phase was prioritized during weeks 1-2. Phase 2, during weeks 3-4, focused on the editor interface, including the scene view, script view, file explorer, properties panel, and related UI components. At this stage, the focus was on layout and editor structure rather than AI integration. Phase 3, during weeks 5-6, introduced the AI infrastructure. This included system prompts, LLM integration through the Vercel AI SDK, basic AI actions, and improvements to the chat interface. Phase 4, during weeks 7-8, focused on the more advanced AI features: creating and modifying game objects, writing and editing scripts, starting and stopping the simulation, and generating assets. This was the most technically demanding phase of the project. Phase 5, during weeks 9-10, focused on testing, bug fixing, and preparing the system for evaluation. Phase 6 focused on completing the evaluation, analyzing results, and writing the report. Although some tasks changed during development, the phase structure helped keep the project aligned with the available time, and all major phases were completed within the planned period. Programming was performed using a parallel pair-programming workflow. Both developers worked simultaneously on the same codebase using the Live Share extension for Visual Studio Code, rather than following a traditional driver-navigator pair-programming model. This made it possible to work on separate parts of the system while still maintaining shared context. AI coding tools were also used during implementation, including Claude Code, Cursor, and GitHub Copilot through Visual Studio Code. Much of the code in Slop Engine was written with AI assistance, but the generated code was manually reviewed, architected, tested, and in many cases substantially revised. The AI tools were therefore used as implementation assistants rather than as autonomous authors of the system. This methodology is consistent with the theme of the thesis. Slop Engine investigates how AI can assist in creative technical work, and the project itself used AI assistance as part of its development process. Given the scope of implementing a game engine, editor, AI assistant, tool system, scripting runtime, and evaluation harness within a bachelor-project timeframe, AI-assisted implementation was also a practical necessity. The final system should therefore be understood as both an object of study and a product of the AI-assisted workflows discussed in the report.

5 Evaluation

5.1 Methodology

This evaluation compares Slop Engine against two alternative AI-assisted game development workflows in order to measure how different tooling environments affect task completion, iteration count, runtime correctness, and development speed. The comparison should therefore be read as a workflow-level comparison rather than an ablation study isolating the effect of multi-agent decomposition alone.

5.1.1 Metrics

The evaluation collects both quantitative and qualitative metrics.

Quantitative metrics include:

- Agent response time: wall-clock time from prompt submission to completed agent response.
- Total workflow time: cumulative agent response time across all iterations, excluding manual evaluator play-test time.
- Token usage: input tokens, output tokens, total tokens, and cached tokens.
- Iterations: the number of user-agent interaction cycles required to reach the final result.
- Correction iterations: the number of evaluator follow-ups required after observed failures.
- Tool calls: the number of external actions performed by the agent, treated as a measure of interaction overhead.
- Runtime errors: whether runtime errors remain after the agent's final attempt.

Qualitative evaluation is performed through manual play-testing after each run. Each result is evaluated using six rubric criteria, each scored on a 0–2 scale:

- Movement / core mechanic works: whether the main gameplay loop works.
- Win condition exists & triggers: whether the game can be won.
- Lose condition exists & triggers: whether the game can be lost.
- UI exists and is visible: whether required interface elements are visible and functional.
- No crashes, game is playable: whether the game runs without crashing.
- Camera faces the scene: whether the game is visible.

5.1.2 Automated Benchmarks

In order to evaluate how our agent performs in relation to other AI-powered game development workflows, we perform controlled experiments across three scenarios:

1. **Slop Engine** - The user interacts with our built-in AI assistant inside the Slop editor. The assistant has direct access to the Babylon.js scene graph via tool calls (creating meshes, configuring physics, setting materials). Additionally, it can assign programming tasks to a scripting agent.

2. **OpenCode + Babylon.js** - The open-source coding agent OpenCode is given a minimal workspace containing an `index.html` (which loads Babylon.js 8 from a Content Delivery Network) and an empty `game.js` stub. The agent writes the entire game as vanilla JavaScript in `game.js`. It has no scene-graph tools, instead it reasons about and generates raw Babylon.js API calls in the JavaScript code.
3. **OpenCode + Roblox MCP** - OpenCode is connected to Roblox Studio via a local MCP (Model Context Protocol) server. The agent uses MCP tools to create instances, insert scripts, and manipulate the Studio scene tree. Game logic is placed in `ServerScriptService` or `StarterPlayerScripts`.

All three scenarios use the same underlying LLM, specifically `GPT-5.4 mini` [74], to control for model capability. The comparison therefore focuses on the combined effect of development environment, tool interface, and engine abstraction. It does not isolate the game engine itself from the tooling layer, which is treated as part of each workflow.

We evaluate three game types: Breakout, Dodger, and Platformer, across the three scenarios. Each game-scenario combination is repeated three times, producing $3 \times 3 \times 3 = 27$ experimental runs. This gives every scenario the same number of attempts on every task, while keeping the evaluation feasible within time and token constraints. Although a larger study with more repetitions, games, and scenarios would provide stronger statistical evidence, 27 runs was chosen as a practical minimum for comparing the workflows across multiple game genres and repeated trials. The repeated runs are not intended to support strong statistical inference, but to reveal whether observed behavior is stable across multiple attempts rather than the result of a single lucky or unlucky run.

The Breakout game is based on the classic game from 1976, developed by Atari, Inc. for arcades, in which you hit a ball with a paddle, which bounces to hit blocks that break on impact [8]. Choosing Breakout as a benchmark game allows for testing how well each scenario performs on constructing a well-known and established game, which is likely represented in public examples and tutorials included in the model's training data. Breakout is included as a familiar reference task, and because the game is well known it tests whether each workflow can reproduce a conventional arcade game with clearly defined mechanics rather than requiring the agent to invent an unfamiliar design. The Dodger game is a game where the player has to dodge blocks that fall from the top of the arena. Player loses if they are hit by a block, and wins if they survive for long enough. Dodger is included as a simple but less canonical task. Unlike Breakout, it is described primarily through the prompt rather than relying on a single widely recognized reference implementation. Platformer is a popular genre in gaming [75], and is included because it requires spatial level construction, player movement, jumping, collision handling, and a goal state. It is therefore a broader test of scene layout and gameplay logic than Breakout or Dodger, but its prompt must specify the required mechanics clearly to reduce ambiguity.

For each game, all scenarios receive the same base prompt describing the game mechanics, controls, win/lose conditions, and visual requirements. Scenario-specific prefixes are prepended to guide the agent toward the correct tooling. For example, the OpenCode + Babylon.js prefix instructs the agent to use `BABYLON.MeshBuilder` and a perspective camera, and forbids editing `index.html`. The Slop Engine assistant receives its normal system prompt and tool definitions,

which are part of the evaluated system. No additional game-specific prefix is added. The exact prompts can be found in Section 8.2.

5.1.2.1 Iteration and error-correction protocol

After the agent completes its initial attempt, the evaluator[2] play-tests the result and checks each qualitative criterion. If the game is not in a correct and complete state, the evaluator provides a fixed follow-up prompt based on the type of failure observed. Each failure-driven follow-up counts as one correction iteration. A run is terminated after three correction iterations and is scored as-is. The fixed prompts are:

- **Planning failure**: *"You have misunderstood the task. Read it again carefully."*
- **Tool failure**: *"You have not used the correct tools for the task. Try again."*
- **Scope failure**: *"Continue to implement the features of the game."*
- **Core mechanics failure**: *"The core mechanics do not work. Please fix them."*
- **Win/lose condition missing**: *"There is no win / lose condition. Please implement them."*
- **Runtime errors**: *"There are runtime errors. Please fix them."*
- **Camera failure**: *"I can't see the game. Please fix the camera."*
- **UI failure**: *"I can't see the UI / the UI is broken, please fix it."*

When multiple failures are present, the corresponding prompts are concatenated. Fixed follow-up prompts are used to reduce evaluator variability and provide the minimum correction needed to identify the failure type without giving scenario-specific implementation guidance.

Planning turns are counted separately from correction iterations. They are included in total iteration count and total time, because they require user interaction, but they are not treated as failures. Planning mode is part of the normal Slop Engine workflow. The orchestrator agent is instructed in its system prompt to determine if planning is needed, and it always has access to planning mode tools. It is therefore evaluated as part of the system rather than disabled.

5.2 Test Harness

To collect consistent measurements across the 27 experimental runs, we implemented a custom evaluation harness that automates environment setup, event logging, metric aggregation, qualitative grading, and CSV export. The harness runs as a server-side plugin in Slop Engine's existing Elysia web server, with a companion dashboard at `/harness` for starting runs, monitoring progress, submitting follow-up prompts, and recording manual rubric scores. The model used across all runs is pinned to `GPT-5.4 mini` [74], served through Microsoft Foundry, with model version `2026-03-17` and the `Microsoft.DefaultV2` guardrail. No custom temperature, top-p, or seed configuration was applied. This means the evaluation controls for model and provider configuration, but does not enforce deterministic decoding.

5.2.1 Architecture overview

The harness consists of four layers:

1. **Scenario runners** - For the two OpenCode scenarios, the harness spawns an `opencode` subprocess in JSON-output mode, using an isolated artifact directory as the workspace. For

[2] "The evaluator" refers to the person evaluating the game, not the custom evaluation harness.

the Slop Engine scenario, no external subprocess is needed. Instead, the harness starts a run through the editor's existing `/api/chat` endpoint and opens the corresponding editor session in a browser tab.

2. **Event pipeline** - All activity across scenarios flows through a unified event system. Nine event types capture the full lifecycle of a run: `run_started`, `iteration_started`, `llm_call`, `tool_call`, `text_chunk`, `awaiting_input`, `iteration_ended`, `runtime_error`, and `run_stopped`. Each event carries a millisecond timestamp.
3. **Aggregation and storage** - An in-memory aggregator accumulates per-iteration and per-run totals (tokens, tool calls, duration). When a run stops, the final summary is persisted to a SQLite database [76] alongside all iteration-level detail. Persisting summaries and iteration details to SQLite ensures that completed measurements survive server restarts or crashes.
4. **Dashboard and export** - A web dashboard streams events in real time via Server-Sent Events (SSE), lets the evaluator send follow-up prompts (nudges), and provides a grading form for qualitative rubric scores. Two CSV export endpoints produce run-level and iteration-level data for analysis.

5.2.2 Environment isolation

Each run receives its own artifact directory under `~/.slop-harness/runs/<run-id>/`. This directory is intentionally located outside the Slop Engine repository to prevent OpenCode from discovering or modifying the engine source code during workspace detection. Inside each artifact directory, the harness creates a minimal `.git` directory that acts as a project-root boundary. OpenCode therefore treats the artifact directory as the repository root instead of walking upward into a parent repository. This isolation was added after early evaluation runs showed that OpenCode sometimes attempted to inspect or modify Slop Engine's own source code rather than the files in the benchmark workspace. Without this boundary, the OpenCode baselines would have access to implementation details unavailable to the Slop Engine assistant and could also corrupt the evaluation environment. For the OpenCode + Babylon.js scenario, the artifact directory is seeded with a template containing `index.html` (loading Babylon.js from CDN) and an empty `game.js` stub. For the Roblox scenario, only a `README.md` is placed, since the agent operates on Roblox Studio directly via MCP tools. In both cases, a generated `opencode.json` configures the LLM provider credentials and, for the Roblox scenario, the local MCP server connection.

5.2.3 Data collection

Quantitative metrics are captured automatically, with no manual intervention:

- **Token counts** - For OpenCode scenarios, each `step_finish` event from the subprocess reports input tokens, output tokens, and cache-read tokens. For the Slop scenario, the harness instruments the Vercel AI SDK's response object after each LLM call, extracting the same fields from `usage.inputTokens`, `usage.outputTokens`, and `usage.cachedInputTokens`.
- **Tool calls** - Each tool invocation is logged with the tool name, a truncated preview of its input and output (capped at 200 characters), and any error message. The total count is aggregated per iteration and per run.
- **Duration** - The harness measures *active* duration only: the sum of per-iteration time spans from `iteration_started` to `iteration_ended`. Time spent by the evaluator between itera-

tions (play-testing, deciding on follow-up prompts) is excluded. For iterations that end with an `awaiting_input` event rather than an explicit `iteration_ended` (common for planning/clarification turns), the harness falls back to the timestamp of the last LLM or tool-call activity within that iteration.

- **Iterations** - Each prompt-response cycle is recorded as a separate iteration with its kind (`initial`, `nudge`, `clarification`, or `plan_approval`), the prompt text, and all associated metrics.
- **Runtime errors** - Errors reported while the LLM is idle (i.e., after an iteration ends but before the next begins) are counted toward the run's error total. Runtime errors are counted only when they persist while the agent is idle, because these represent errors left for the evaluator or player to encounter. Errors emitted during an active iteration are still logged, but they are not included in the final error count unless they remain after the agent finishes that iteration.

All events are simultaneously written to an append-only NDJSON [77] file (`events.ndjson`) for post-mortem analysis and streamed to connected dashboard clients via Server-Sent Events (SSE).

5.2.4 Qualitative grading

After each agent response, the evaluator manually launched and interacted with the game to determine whether a correction prompt was needed. Final rubric scores were submitted only after the run completed or reached the correction limit. Each qualitative dimension discussed in Section 5.1.1 is scored on a 0–2 scale:

Score	Meaning
0	No - the criterion is not met
1	Partially - the criterion is partly met or has issues
2	Yes - the criterion is fully met

Table 2: Rubric scoring scale

Additionally, if the game failed to reach a complete state, the evaluator records the primary failure mode (planning, tool, or scope). The evaluator may also write an additional note, but this is optional and does not factor into the final score.

5.2.5 CSV export

The harness provides two export endpoints that produce downloadable CSV files:

- **Run-level export** - One row per run, containing: run ID, game, scenario, run number, status, timestamps, all aggregated quantitative metrics (duration, tokens, iterations, tool calls, runtime errors), all rubric scores, failure mode, evaluator notes, and grading timestamp.
- **Iteration-level export** - One row per iteration across all runs, containing: run ID, game, scenario, run number, iteration index, iteration kind, prompt text, duration, and per-iteration token and tool-call counts.

These exports provide the raw data for all figures and statistical analysis in the Results section. The run-level export is used for aggregate comparisons between scenarios, while the iteration-level export is used to analyze how costs and failures accumulate over repeated corrections.

Because qualitative grading is performed manually by a single evaluator, the rubric provides consistency within the experiment but does not measure inter-rater reliability.

5.3 Results

This section presents the results from the 27 evaluation runs. The raw run-level data is included in appendix Table 9. The results are grouped by quality, token usage, runtime duration, and interaction overhead. The composite quality score is computed from the six gameplay-functionality dimensions (movement, win condition, lose condition, no crash, UI, camera), each scored 0/1/2, for a maximum of 12.

As seen in Figure 13, across the evaluated tasks, OpenCode + Babylon.js achieved the highest overall quality scores. It reached the highest mean score on Breakout and Dodger and remained close to the best result on Platformer. Slop Engine produced more mixed results: it performed moderately on both Platformer and Breakout, but poorly on Dodger. OpenCode + Roblox MCP performed worst on Breakout and Dodger, but was competitive on Platformer.

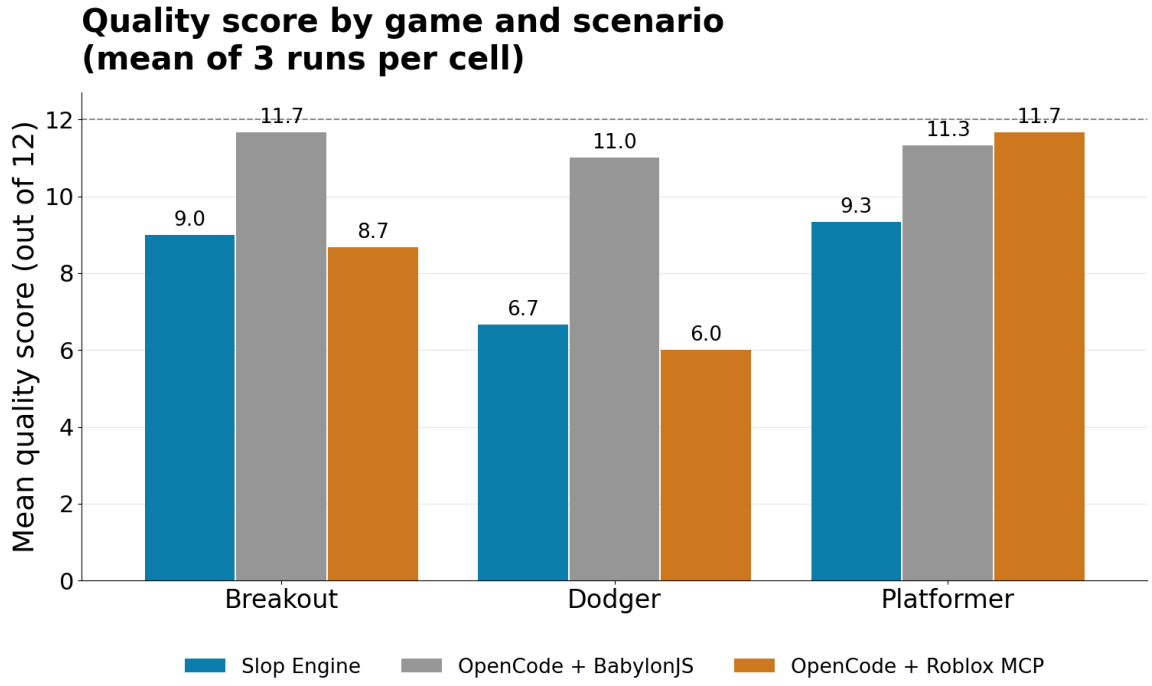

Figure 13: Mean quality score per scenario-game cell. Each cell is the mean of three runs.

In Figure 14 the per-dimension rubric scores show that Slop Engine performed best relative to the baselines on visual presentation dimensions, especially camera setup. However, it performed worse on gameplay-related dimensions such as movement and lose-condition handling. OpenCode + Babylon.js had consistently high scores across most rubric dimensions, while OpenCode + Roblox MCP performed particularly well on UI and camera but less consistently on movement and win-condition handling.

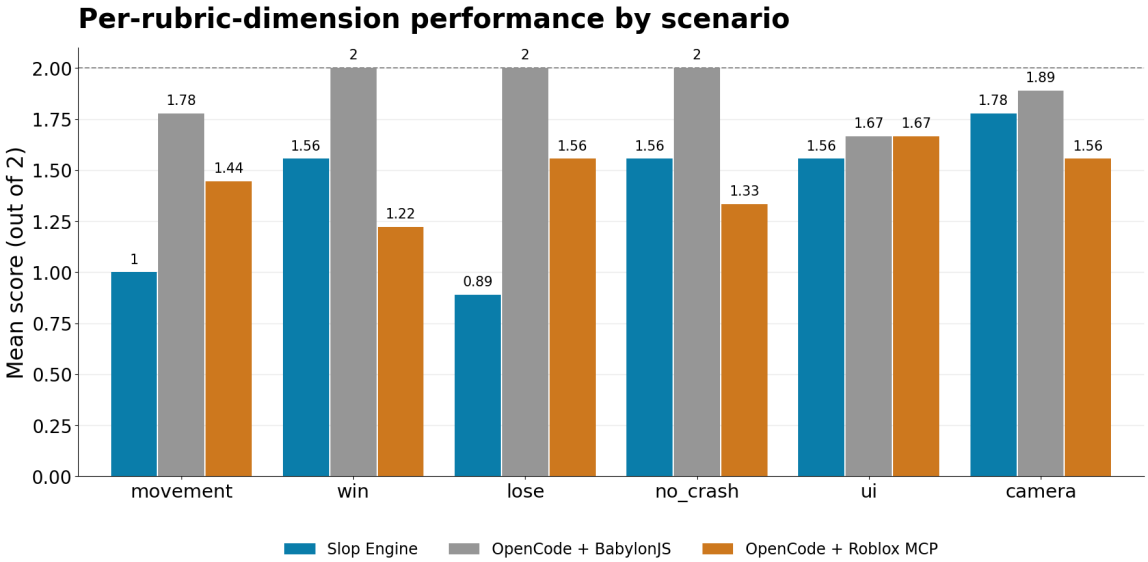

Figure 14: Per-rubric-dimension averages. Each dimension is scored from 0 to 2.

Token usage varied substantially between scenarios, as seen in Figure 15. Slop Engine consumed the most uncached input tokens, with a median of 1,172,949 uncached input tokens per run. This is considerably higher than OpenCode + Babylon.js, which had a median of 38,622 uncached input tokens per run. OpenCode + Roblox MCP had the highest cached input token count, with a median of 3,008,000 cached input tokens per run, while its output token usage remained similar to the other scenarios. Inspection of the Slop Engine runs showed that a large share of this token usage came from the testing workflow, especially the autonomous test tool, which returned full scene snapshots across multiple time steps.

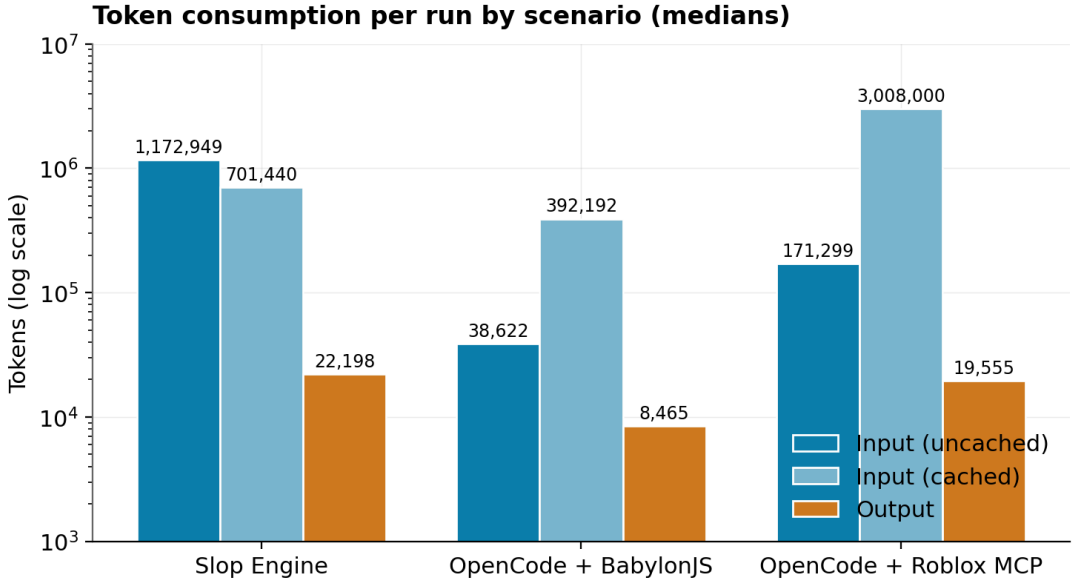

Figure 15: Median token consumption per run by scenario. The y-axis uses a logarithmic scale.

Figure 16 shows that run duration also differed across scenarios. Slop Engine had relatively stable run times, while OpenCode + Babylon.js had the lowest typical duration. OpenCode + Roblox MCP showed the widest duration spread, including one run that lasted approximately 47 minutes after the agent became stuck in a loop.

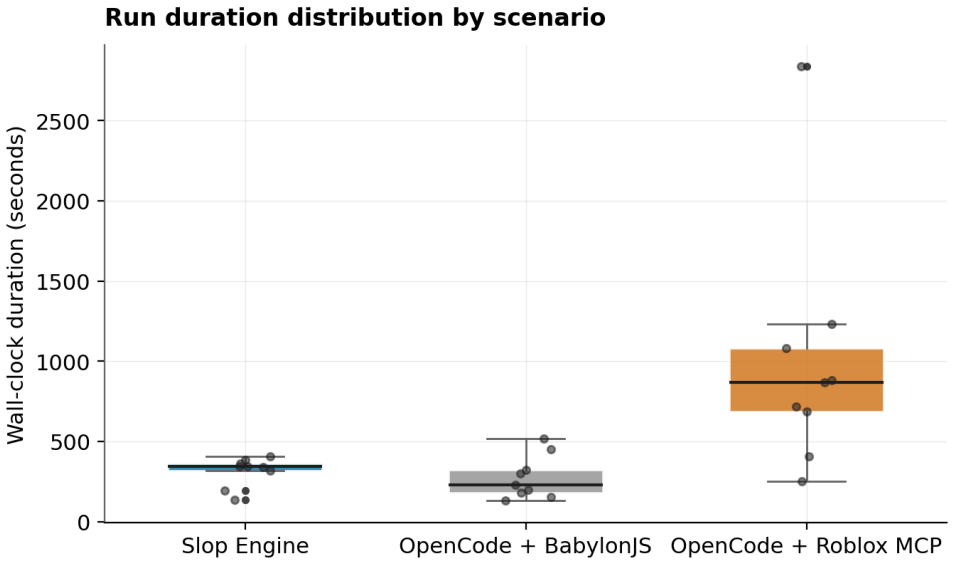

Figure 16: Wall-clock run duration distribution by scenario.

The relationship between iteration count and tool calls, seen in Figure 17, further shows differences between workflows. Slop Engine runs generally required more iterations, partly because its planning mode introduces additional interaction steps before implementation. OpenCode + Babylon.js clustered at low iteration and low tool-call counts. OpenCode + Roblox MCP showed greater variance, including runs with high tool-call counts.

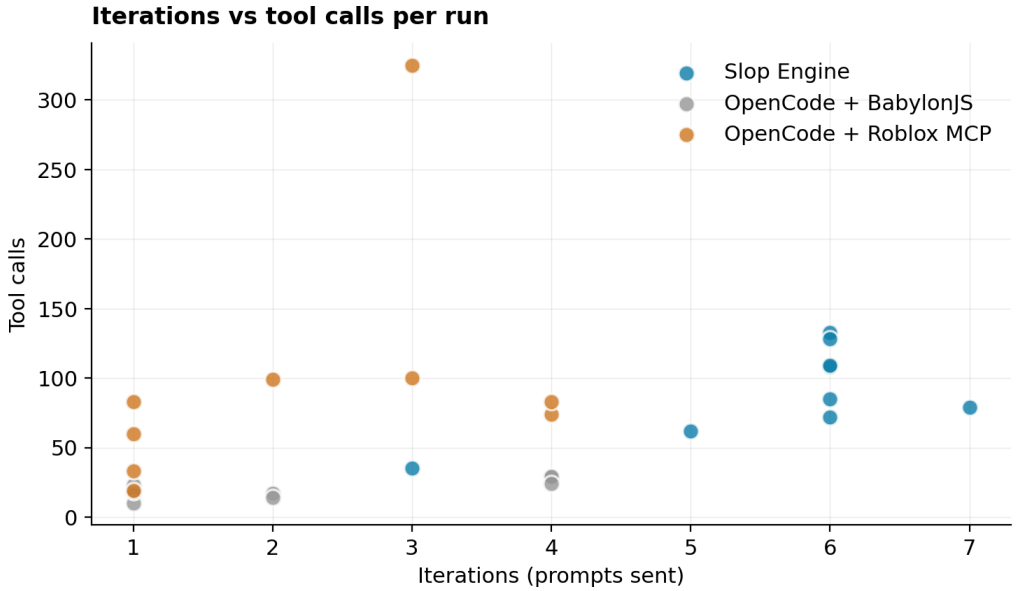

Figure 17: Iteration count plotted against tool calls per run.

5.4 Threats to Validity

Several factors limit the strength of the conclusions that can be drawn from this evaluation. First, the sample size is small. Each game-scenario combination was repeated three times, resulting in 27 total runs. This is enough to identify broad patterns, but not enough to make strong statistical claims about performance differences between workflows. Second, the scenarios are not identical environments. Although all scenarios use the same underlying model, they differ in engine, tool interface, available abstractions, and prompting. The results should therefore be interpreted as a comparison of complete AI-assisted development workflows rather than an isolated measurement of Slop Engine against a single controlled variable. Third, the prompts differ between scenarios. The OpenCode scenarios include harness-specific instructions describing the workspace and tooling, while the Slop Engine scenario relies on its built-in system prompt and tool definitions. These differences are necessary for each agent to operate in its environment, but they may influence the results. Fourth, Slop Engine's planning mode affects the iteration metric. Slop Engine always requires at least three iterations because the assistant presents and confirms a plan before implementation. These planning turns are part of the normal workflow and are therefore included, but they inflate iteration counts compared to workflows that begin implementation immediately. Fifth, one OpenCode + Roblox MCP run lasted over 40 minutes because the agent became stuck in a loop. The run was kept in the dataset because the looping behavior is a real failure mode of that workflow. However, it increases the variance and affects aggregate duration metrics for the Roblox MCP scenario. Finally, qualitative grading was performed manually by a single evaluator. The rubric provides consistency within the experiment, but no inter-rater reliability was measured. The qualitative scores should therefore be interpreted as structured observations rather than fully objective measurements.

6 Discussion

6.1 Interpretation of Results

The research question asks how effectively a multi-agent AI system, operating within a purpose-built game environment, can autonomously generate functional game prototypes. The evaluation results show that Slop Engine produces functional prototypes, given that all nine runs created games that launched and rendered 3D scenes, but it does not do so more efficiently or at a higher quality than the baseline. The lightweight OpenCode + Babylon.js setup achieved a median quality score of 12/12 compared to 9/12 for Slop Engine, while completing runs in fewer iterations, fewer tool calls, roughly 30 times fewer uncached input tokens and roughly 2.6x fewer output tokens.

This outcome was not obvious from the design motivation. The engine was designed with the hypothesis that embedding the AI in the game development environment, providing the AI structured access to the game runtime, would reduce errors and improve output quality. In practice, this strategy introduced its own costs that offset the expected gains. Most significantly, Slop Engine uses many more input tokens than any of the other scenarios, with the median being at approximately 1.17 million tokens.

The per-game breakdown reveals that Slop Engine's weaknesses are not uniform. On the platformer task, Slop Engine scored 9.3/12 which is below the baseline's but within a competitive range, with all three runs producing navigable 3D levels. On breakout, it achieved one perfect run alongside two weaker attempts, suggesting the engine *can* produce correct results, but does so inconsistently. Dodger, however, seemed to reveal a systematic failure: none of the three runs scored more than 1 on movement, and none implemented a functioning lose condition. This points to a specific gap in how the engine or agents handle collision-triggered logic rather than a general inability to produce functional games. Dodger is particularly sensitive to runtime event handling: falling objects must spawn repeatedly, move over time, collide with the player, trigger a lose state, and coexist with a survival timer. These requirements depend less on static scene construction and more on reliable per-frame logic and collision callbacks, which were among Slop Engine's weakest areas.

The uneven distribution of failures across rubric dimensions also suggest that the architecture has genre-specific strengths that our benchmark games do not reward. Slop Engine did well on the UI and camera scores while falling behind on movement and lose-condition logic. This makes sense in the context of the design: scene-manipulation tools give the agent direct spatial control of the scene, but the single-pass architecture and the snapshot-based test agent make tight gameplay-logic loops expensive. The three games we chose to benchmark with: Breakout, Dodger and Platformer are all arcade games where logic is a lot more important than the spatial layout, playing to the code-only strengths of OpenCode. Genres where spatial structure, scene composition or visual layout carry more of the design weight than per-frame logic may be where vertical engine integration actually pays off. Potential candidates include walking simulators, open-world games or turn-based games. Future evaluations should include at least one genre

in this category to test whether Slop Engine's architectural overhead can be worth it when the agent's spatial tooling is doing more of the work.

6.2 Issues and Limitations

The most significant factor behind the token difference is architectural: Slop Engine's test agent. After the other subagents have finished creating the scene and scripts, the testing subagent is invoked to validate the result. The `run_autonomous_test` tool that allows it to test the game E2E (End-To-End), can return millions of tokens in response. This is because it returns all the properties of each node, at each requested snapshot time, so that the test agent can inspect how it changes over time and if it matches the assumed values. The tool is valuable in principle, since automated runtime validation is desirable, but its current implementation has a poor cost-benefit ratio. In the current implementation there is no mechanism to filter out irrelevant objects from the results, which means that for larger scenes, a single scene-retrieval can fill a significant portion of the testing subagent's context window by itself. The large amount of tokens most likely means that for scenes with many nodes, the relevant information gets "lost in the middle", as it has been shown that large context leads to decreased output quality [36]. The test agent cannot visually inspect the scene, meaning it cannot detect whether UI elements are visible. For seeing if game objects are spatially aligned or the camera framing is correct, it needs to infer the positions based on our returned snapshot values.

Another constraint of our architecture is that the main agent operates in a single tool-calling pass. Once it emits a text response, it will not invoke additional tools until the next iteration. This means the engine cannot self-correct within a single turn the way OpenCode does, unless it does so before sending any text to the user. In our observations, OpenCode frequently read its own output, detected issues, and revised the implementation, sometimes making dozens of tool-calls in a single iteration. Slop Engine's one-pass design forces corrections into follow-up iterations.

Slop Engine averaged 3-4 iterations higher than baseline, because it uses a forced Planning Mode. This planning mode allows the orchestrator to ask the user questions before proceeding, and allows it to finalize an implementation plan. Theoretically, this should allow it to create a better output as it would require fewer iterations to create what the user wants. However, as observed, it performed worse than the baseline which did not make use of planning mode.

The model that was used in these experiments was consistent, as all scenarios used GPT-5.4-mini [74], through Microsoft Foundry. The model choice greatly affects the capability of the agent. Since this was a smaller model that does not have SOTA (state-of-the-art) spatial reasoning, coding abilities and knowledge, it means that all our experiments perform that much worse than what using a SOTA model (like Opus 4.7 or GPT-5.5, at the time of writing) would [78]. Smaller models are also notoriously bad at complex, multi-step tool calls, and maintaining attention over massive context windows. Conversely, they are relatively good at one-shot code generation, which gives baseline OpenCode even more of an advantage against Roblox MCP and Slop Engine.

A limitation of our testing harness that became apparent during grading, is the absence of a spatial correctness dimension. Multiple runs across all three scenarios produced games where geometry was technically present but misaligned. Platforms placed too high to reach, paddles

sized incorrectly and play areas oriented along the wrong axis. These issues were not captured by any rubric category. Movement was graded as functional if the player could move at all, regardless of whether the level was completable. We could not prompt the agents for follow-up iterations to fix spatial issues, because our fixed-prompt methodology had no category for them. A future evaluation should include a "spatial correctness" or "level completability" dimension to capture this type of error, which we suspect affects AI-generated 3D games much more than 2D ones.

Overall, the evaluation suggests that Slop Engine's current architecture succeeds at making game state accessible to an AI agent, but does not yet convert that access into better benchmark performance. The main challenge is not whether the agent can manipulate a scene, but whether it can do so efficiently while maintaining enough runtime and visual feedback to correct its own mistakes. This makes future work less about adding more tools and more about improving observation quality, filtering context, and enabling tighter self-correction loops.

6.3 Ethical Considerations

Slop Engine automates the process of building scenes, creating meshes and textures, and writing game code. These are tasks traditionally performed by human developers. The most immediate concern is attribution and ownership of generated output: code and assets produced by the engine is authored by a large language model (LLM), not by the user who prompted it. Current legal frameworks around AI-generated works remain mostly unsettled [79]. A production version of the engine should make this explicit, both to the user and in any artifacts it produces, so that downstream consumers of the output can make informed decisions.

A related issue is over-reliance. The engine is designed to minimize the amount of programming knowledge required to produce a game prototype. But if adopted as a primary development tool, it risks creating a dependency where users cannot debug, extend or maintain the code the engine generates for them. This is not unique to our system, it applies broadly to all AI generation tools. However, the risk is higher in our case because the engine abstracts away not just the code syntax but the entire development environment. A user of Copilot or Cursor still works inside an editor, reads compiler errors and builds a mental model of their codebase. A user of Slop Engine may never see any code at all.

7 Conclusion

This project set out to answer whether a multi-agent AI system, operating within a purpose-built game engine, can autonomously generate functional game prototypes. We designed and implemented Slop Engine, a browser-based, vertically integrated game engine with vertical integration between the AI assistant and the engine's scene graph, physics simulation, scripting runtime, and editor state, and evaluated it against two baselines: OpenCode + Babylon.js (a lightweight, code-only workflow) and OpenCode + Roblox MCP (a tool-augmented workflow targeting an existing engine).

The answer to our research question is qualified: the system *can* produce functional prototypes, but it does not yet do so more effectively than simpler alternatives. All nine Slop Engine runs produced games that launched, rendered 3D scenes, and responded to at least some user input. However, the lightweight OpenCode + Babylon.js baseline achieved a median quality score of 12/12 compared to Slop Engine's 9/12, while consuming roughly 30 times fewer uncached input tokens, 2.6 times fewer output tokens, and completing runs in fewer iterations. The Roblox MCP scenario fell between the two, with higher variance and its own set of tool-integration challenges. Our results suggest that giving an AI agent structured access to engine internals is not, by itself, sufficient to outperform a more traditional agent that simply writes code. The structured environment introduced overhead, a multi-agent orchestration pipeline, a planning phase, and an automated test agent, that consumed tokens and iterations without proportionally improving output quality. The code-only baseline benefited from a tighter feedback loop: it could read, revise, and re-execute its own output within a single turn, whereas Slop Engine's single-pass architecture forced corrections into subsequent iterations. That said, the evaluation also revealed that Slop Engine's design has distinct strengths. It produced the most consistent run durations across all scenarios, and it tied or led on the visual dimensions of the rubric (UI and camera), likely because the scene-manipulation tools gave the agent more direct control over spatial layout than raw code generation. These properties suggest that the architecture has merit, but that the current implementation has not yet realized its potential.

The core contribution of this work is not a claim of superiority, but a concrete, open-source reference implementation and evaluation framework for AI-assisted game engines. We believe the comparative harness, the orchestrator–subagent architecture, and the lessons learned from the evaluation particularly around token efficiency, single-pass limitations, and the cost of automated testing provide a useful foundation for future work in this space.

7.1 Future Work

Several directions could improve on the results presented here. First, improving subagents' abilities to perform multiple tool-calling passes before returning control to the orchestrator would enable within-turn self-correction, closing the feedback-loop gap with code-only agents. Second, the test agent's snapshot mechanism should be redesigned to return compacted results, being able to query specific nodes and properties, rather than full object property dumps, which would drastically reduce token consumption. Integrating visual feedback is a natural extension that gives the test agent access to screenshots of the running game, as GameDevBench

[44] demonstrated improves agent performance on visual tasks. Third, the evaluation rubric should be expanded to include a spatial correctness dimension, capturing the misalignment issues that affected runs across all three scenarios but were not scored by the current criteria. Finally, repeating the evaluation with a frontier-tier model (rather than GPT-5.4 mini) would help distinguish the limitations of the architecture from the limitations of the model's spatial reasoning and coding capabilities.

Bibliography

[1] Unity Technologies, "Unity Engine." 2026. [Online]. Available: https://unity.com/products/unity-engine

[2] Epic Games, "Unreal Engine." 2024. [Online]. Available: https://www.unrealengine.com/features

[3] Godot Community, "Godot Engine Features: Design." 2026. [Online]. Available: https://godotengine.org/features/#design

[4] Unity Technologies, "2026 Unity Game Development Report: Trends, tools, and insights." Accessed: May 14, 2026. [Online]. Available: https://unity.com/blog/2026-unity-game-development-report-trends

[5] J. Manker and M. Arvola, "Prototyping in Game Design: Externalization and Internalization of Game Ideas," in *Proceedings of HCI 2011*, 2011. Accessed: May 14, 2026. [Online]. Available: https://www.researchgate.net/publication/265974873_Prototyping_in_Game_Design_Externalization_and_Internalization_of_Game_Ideas

[6] Roblox, "MCP Servers." Accessed: May 13, 2026. [Online]. Available: https://create.roblox.com/docs/studio/mcp

[7] Unity Technologies, "Unity AI: AI Game Development Tools & RT3D Software." Accessed: May 13, 2026. [Online]. Available: https://unity.com/features/ai

[8] Wikipedia, "Breakout (video game)." Accessed: May 09, 2026. [Online]. Available: https://en.wikipedia.org/wiki/Breakout_(video_game)

[9] P. Paul, S. Goon, and A. Bhattacharya, "History and comparative study of modern game engines," *International Journal of Advanced Computer and Mathematical Sciences*, vol. 3, pp. 2230–9624, 2012.

[10] J. Gregory, *Game Engine Architecture*, 3rd ed. Boca Raton: CRC Press, Taylor & Francis, 2018.

[11] W. Spector, "Postmortem: Ion Storm's Deus Ex," *Game Developer Magazine*, Dec. 2000, [Online]. Available: https://www.gamedeveloper.com/design/postmortem-ion-storm-s-i-deus-ex-i-

[12] Microsoft, "Havok Physics for Web." Accessed: Mar. 23, 2026. [Online]. Available: https://www.havok.com/

[13] Babylon.js Documentation, "Using Havok and the Havok Plugin." [Online]. Available: https://doc.babylonjs.com/features/featuresDeepDive/physics/havokPlugin

[14] Epic Games, "Nanite Virtualized Geometry." [Online]. Available: https://dev.epicgames.com/documentation/unreal-engine/nanite-in-unreal-engine

[15] K. Kontus and Sensor Tower, "The Big Game Engines Report of 2025: Unity vs Unreal Trends," technical report, 2025. [Online]. Available: https://app.sensortower.com/vgi/assets/reports/The_Big_Game_Engines_Report_of_2025.pdf

[16] Roblox Corporation, "Roblox Creator Documentation." [Online]. Available: https://create.roblox.com/docs

[17] Lua.org, "The Programming Language Lua." Accessed: May 13, 2026. [Online]. Available: https://www.lua.org/

[18] Babylon.js Contributors, "Babylon.js — Powerful, Beautiful, Simple, Open." Accessed: Mar. 23, 2026. [Online]. Available: https://www.babylonjs.com/

[19] R. Cabello and Three.js Contributors, "Three.js – JavaScript 3D Library." [Online]. Available: https://threejs.org/

[20] D. Chong, "unreal-mcp: Control Unreal Engine through natural language using MCP." [Online]. Available: https://github.com/chongdashu/unreal-mcp

[21] Coding-Solo, "godot-mcp: Model Context Protocol (MCP) implementation for Godot." [Online]. Available: https://github.com/Coding-Solo/godot-mcp

[22] A. Vaswani *et al.*, "Attention Is All You Need," *Advances in Neural Information Processing Systems*, vol. 30, 2017, [Online]. Available: https://arxiv.org/abs/1706.03762

[23] L. Ouyang *et al.*, "Training Language Models to Follow Instructions with Human Feedback," *Advances in Neural Information Processing Systems*, vol. 35, 2022, [Online]. Available: https://arxiv.org/abs/2203.02155

[24] J. Wei *et al.*, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," *Advances in Neural Information Processing Systems*, vol. 35, 2022, [Online]. Available: https://arxiv.org/abs/2201.11903

[25] T. Schick *et al.*, "Toolformer: Language Models Can Teach Themselves to Use Tools," *Advances in Neural Information Processing Systems*, vol. 36, 2023, [Online]. Available: https://arxiv.org/abs/2302.04761

[26] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, "Gorilla: Large Language Model Connected with Massive APIs," *arXiv preprint arXiv:2305.15334*, 2023, [Online]. Available: https://arxiv.org/abs/2305.15334

[27] C. Qu *et al.*, "Tool Learning with Large Language Models: A Survey," *Frontiers of Computer Science*, vol. 19, no. 8, p. 198343, 2025, [Online]. Available: https://arxiv.org/abs/2405.17935

[28] Z. R. Tam, C.-K. Wu, Y.-L. Tsai, C.-Y. Lin, H.-y. Lee, and Y.-N. Chen, "Let Me Speak Freely? A Study on the Impact of Format Restrictions on Performance of Large Language Models," *arXiv preprint arXiv:2408.02442*, 2024, [Online]. Available: https://arxiv.org/abs/2408.02442

[29] A. Gupta and others, "Task Interference in Tool-Use Language Models," 2024.

[30] S. Yao *et al.*, "ReAct: Synergizing Reasoning and Acting in Language Models," in *International Conference on Learning Representations (ICLR)*, 2023. [Online]. Available: https://arxiv.org/abs/2210.03629

[31] Anthropic, "Introducing the Model Context Protocol." Accessed: Apr. 08, 2026. [Online]. Available: https://www.anthropic.com/news/model-context-protocol

[32] Anysphere, "Cursor: The AI Code Editor." [Online]. Available: https://www.cursor.com/

[33] GitHub, "GitHub Copilot Documentation." [Online]. Available: https://docs.github.com/en/copilot

[34] Anthropic, "Claude Code." [Online]. Available: https://docs.anthropic.com/en/docs/claude-code

[35] C. E. Jimenez *et al.*, "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024. [Online]. Available: https://www.swebench.com/

[36] N. F. Liu *et al.*, "Lost in the Middle: How Language Models Use Long Contexts," *Transactions of the Association for Computational Linguistics*, 2024, [Online]. Available: https://arxiv.org/abs/2307.03172

[37] T. Masterman, S. Besen, M. Sawtell, and A. Chao, "The Landscape of Emerging AI Agent Architectures for Reasoning, Planning, and Tool Calling: A Survey," *arXiv preprint arXiv:2404.11584*, 2024, [Online]. Available: https://arxiv.org/abs/2404.11584

[38] C. Qian *et al.*, "Communicative Agents for Software Development," *arXiv preprint arXiv:2308.08155*, 2023, doi: 10.48550/arXiv.2308.08155.

[39] Y. Liu *et al.*, "Agent design pattern catalogue: A collection of architectural patterns for foundation model based agents," *The Journal of Systems and Software*, vol. 220, p. 112278, 2025, doi: 10.1016/j.jss.2024.112278.

[40] A. R. Manikanta, B. Ameer Shaik, Y. Kancharla, R. Gurram, and G. Ramasamy, "AI-Driven Game Mechanics: Implementing Finite State Machines in Pac-Man," in *2025 IEEE International Conference on Interdisciplinary Approaches in Technology and Management for Social Innovation (IATMSI)*, 2025, pp. 1–6. doi: 10.1109/IATMSI64286.2025.10985319.

[41] ARC Raiders Team, "The Evolution of ARC Raiders EP3: Building ARC Machines." [Online]. Available: https://arcraiders.com/news/evolution-of-arc-raiders-episode-3

[42] GitHub, "GitHub Copilot — Your AI accelerator for every workflow." Accessed: May 14, 2026. [Online]. Available: https://github.com/features/copilot

[43] Microsoft, "Manage context for AI in Visual Studio Code." Accessed: May 14, 2026. [Online]. Available: https://code.visualstudio.com/docs/copilot/chat/copilot-chat-context

[44] W. Chi *et al.*, "GameDevBench: Evaluating Agentic Capabilities Through Game Development." [Online]. Available: https://arxiv.org/abs/2602.11103

[45] W. 3D Consortium, "X3D & VRML, The Most Widely Used 3D Formats." Accessed: Apr. 11, 2026. [Online]. Available: https://www.web3d.org/x3d-vrml-most-widely-used-3d-formats

[46] I. A. P. I. S. M. P. V. L. M. Gavin Bell Silicon Graphics, "The Virtual Reality Modeling Language." [Online]. Available: https://www.martinreddy.net/gfx/3d/VRML.spec

[47] Google, "Saying goodbye to Flash in Chrome." [Online]. Available: https://web.archive.org/web/20170726213033/https://www.blog.google/products/chrome/saying-goodbye-flash-chrome/

[48] Khronos Group, "WebGL 1.0 Specification." [Online]. Available: https://registry.khronos.org/webgl/specs/1.0.0/

[49] Babylon.js Team, "Announcing Babylon.js 6.0." [Online]. Available: https://babylonjs.medium.com/announcing-babylon-js-6-0-dcb5f1662e3a

[50] Wikipedia, "Babylon.js Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Babylon.js

[51] Microsoft Open Source, "Microsoft Open Source Success Story: Babylon." [Online]. Available: https://opensource.microsoft.com/blog/2021/02/22/microsoft-open-source-success-story-babylon/

[52] World Wide Web Consortium, "WebAssembly Core Specification." [Online]. Available: https://www.w3.org/TR/wasm-core-1

[53] Apple WebKit Team, "Next-generation 3D Graphics on the Web." [Online]. Available: https://webkit.org/blog/7380/next-generation-3d-graphics-on-the-web/

[54] W3C GPU for the Web Working Group, "WebGPU." [Online]. Available: https://www.w3.org/TR/webgpu/

[55] Chrome for Developers, "Chrome Ships WebGPU." [Online]. Available: https://developer.chrome.com/blog/webgpu-release/

[56] web.dev, "WebGPU Is Now Supported in Major Browsers." [Online]. Available: https://web.dev/blog/webgpu-supported-major-browsers

[57] Adam Nemeroff, "What is AI slop? A technologist explains this new and largely unwelcome form of online content." [Online]. Available: https://theconversation.com/what-is-ai-slop-a-technologist-explains-this-new-and-largely-unwelcome-form-of-online-content-256554

[58] Microsoft, "TypeScript: JavaScript With Syntax For Types." Accessed: May 13, 2026. [Online]. Available: https://www.typescriptlang.org/

[59] SolidJS Contributors, "SolidJS — Reactive JavaScript Library." Accessed: Mar. 23, 2026. [Online]. Available: https://solidjs.com/

[60] Tailwind Labs, "Tailwind CSS." Accessed: May 13, 2026. [Online]. Available: https://tailwindcss.com/

[61] VoidZero Inc., "Vite." Accessed: May 13, 2026. [Online]. Available: https://vite.dev/

[62] ElysiaJS, "ElysiaJS." Accessed: May 13, 2026. [Online]. Available: https://elysiajs.com/

[63] SolidJS, "Fine-grained reactivity." Accessed: May 13, 2026. [Online]. Available: https://docs.solidjs.com/advanced-concepts/fine-grained-reactivity

[64] Microsoft, "Azure AI Foundry." [Online]. Available: https://azure.microsoft.com/en-us/products/ai-foundry

[65] OpenRouter, "OpenRouter." [Online]. Available: https://openrouter.ai/

[66] Google, "Google AI Studio." [Online]. Available: https://aistudio.google.com/

[67] Microsoft, "Monaco Editor." Accessed: Mar. 23, 2026. [Online]. Available: https://microsoft.github.io/monaco-editor/

[68] Microsoft, "Visual Studio Code." Accessed: May 11, 2026. [Online]. Available: https://code.visualstudio.com/

[69] Microsoft, "IntelliSense." [Online]. Available: https://code.visualstudio.com/docs/editing/intellisense

[70] Vercel, "Vercel AI SDK." Accessed: Mar. 23, 2026. [Online]. Available: https://sdk.vercel.ai/

[71] NanoBanana API, "NanoBanana API." Accessed: May 11, 2026. [Online]. Available: https://nanobananaapi.ai/

[72] Tripo, "Tripo." Accessed: May 11, 2026. [Online]. Available: https://www.tripo3d.ai/

[73] Linear Orbit, Inc., "Linear: A Better Way to Build Product." [Online]. Available: https://linear.app/

[74] OpenAI, "Introducing GPT-5.4 mini and nano." Accessed: May 12, 2026. [Online]. Available: https://openai.com/index/introducing-gpt-5-4-mini-and-nano/

[75] Debadatta Patel, "Platformer Games Market Research Report." Accessed: May 09, 2026. [Online]. Available: https://marketintelo.com/report/platformer-games-market

[76] SQLite, "SQLite." Accessed: May 13, 2026. [Online]. Available: https://sqlite.org/

[77] JSONL Tools, "What is NDJSON?." Accessed: May 13, 2026. [Online]. Available: https://jsonltools.com/what-is-ndjson

[78] Arena Intelligence, "Text Arena." [Online]. Available: https://arena.ai/leaderboard/text

[79] Preeti Sharma, "AI And Copyright Law: Who Owns AI-Generated Content?." [Online]. Available: https://www.mondaq.com/india/copyright/1737464/ai-and-copyright-law-who-owns-ai-generated-content

8 Appendices

8.1 Appendix A: Code and Deployment

- Deployment (available as of 2026): slopengine.studio
- GitHub Repository: github.com/alexop1000/slop-engine

NOTE: The custom evaluation harness discussed in Section 5.2 is only available in the `slop-evaluation` branch of the git repository.

Component	Technology
Language	TypeScript
UI Framework	Solid.js
3D Renderer	Babylon.js
Physics	Havok (WASM)
Build Tool	Vite
Runtime	Bun
Styling	Tailwind CSS 4
Code Editor	Monaco Editor
AI Integration	Vercel AI SDK
Backend Framework	ElysiaJS

Table 3: Core technology stack.

8.2 Appendix B: Prompts Used For Evaluation

Breakout: "Build a game called 'Breakout'. The player controls a paddle that moves left and right along the bottom of a play area using the A and D keys. A ball bounces between the paddle, the side and top walls, and a grid of bricks at the top. Each brick disappears when hit by the ball. The player wins by clearing all bricks. The player loses if the ball falls past the paddle. Display a score and a win/lose screen when the game ends."

Dodger: "Build a game called 'Dodger'. The player controls a block that moves left and right along the bottom of a play area using the A and D keys. Blocks fall from the top of the area at a steady rate, gradually speeding up as the game progresses. The player must avoid being hit by falling blocks. The player wins by surviving for 30 seconds. The player loses if they are hit by a falling block. Display a timer visible to the player and a win/lose screen when the game ends."

Platformer: “Build a simple platformer. The player controls a character that moves left and right with the A and D keys and jumps with the Space key. The level contains at least three platforms at varying heights and a goal flag on one of them. The player wins by reaching the goal flag. The player loses if they fall off the bottom of the level. Display a win/lose screen when the game ends.”

Every scenario has been given an additional part to the prompt in order to standardize results.

- Slop Engine: “” - (the Engine System Prompt is the addition itself)
- Opencode Plain: “Using Babylon.js 8 (already loaded from CDN in index.html), write the entire game in game.js. Do not edit index.html.”
- Opencode Roblox: “Using Roblox Studio via the provided MCP tools, build the game in a blank baseplate place. Place all code in ServerScriptService or StarterPlayerScripts as appropriate.”

8.3 Appendix C: Full Benchmark Results

Scenario	**n**	**Quality (med / mean)**	**Duration (s, med)**	**Input tok (med)**	**Output tok (med)**	**Iterations (med)**	**Tool calls (med)**
Slop Engine	9	9.0 / 8.3	343	1172949	22198	6	85
OpenCode + Babylon.js	9	12.0 / 11.3	228	38622	8465	2	18
OpenCode + Roblox MCP	9	11.0 / 8.8	869	171299	19555	2	83

Table 4: Headline comparison across scenarios. Quality is on a 0-12 scale, summing all six rubric dimensions: movement, win, lose, no crash, UI, and camera, each scored 0/1/2. Quantitative values are medians of n runs except quality, which reports median and mean.

Scenario	**Breakout**	**Dodger**	**Platformer**
Slop Engine	9.0	6.7	9.3
OpenCode + Babylon.js	11.7	11.0	11.3
OpenCode + Roblox MCP	8.7	6.0	11.7

Table 5: Mean quality score per scenario-game cell, out of 12. Each cell is the average of three runs.

Scenario	**Movement**	**Win**	**Lose**	**No crash**	**UI**	**Camera**
Slop Engine	1.00	1.56	0.89	1.56	1.56	1.78
OpenCode + Babylon.js	1.78	2.00	2.00	2.00	1.67	1.89

Scenario	Movement	Win	Lose	No crash	UI	Camera
OpenCode + Roblox MCP	1.44	1.22	1.56	1.56	1.67	1.56

Table 6: Mean score per rubric dimension. Each dimension is scored 0/1/2, so 2.00 indicates a perfect average across all nine runs in the scenario.

Scenario	None	Planning	Tool	Total runs
Slop Engine	7	0	2	9
OpenCode + Babylon.js	6	2	1	9
OpenCode + Roblox MCP	6	2	1	9

Table 7: Distribution of explicit failure modes per scenario. "None" means the run completed without falling into one of the rubric's named failure categories; it does not imply a perfect quality score.

Game	Scenario	Movement	Win	Lose	No crash	UI	Camera
Breakout	Slop Engine	1.00	1.33	2.00	2.00	1.33	1.33
Breakout	OpenCode + Babylon.js	2.00	2.00	2.00	2.00	1.67	2.00
Breakout	OpenCode + Roblox MCP	1.00	1.33	1.33	1.33	1.67	2.00
Dodger	Slop Engine	0.67	2.00	0.00	0.67	1.33	2.00
Dodger	OpenCode + Babylon.js	2.00	2.00	2.00	2.00	1.33	1.67
Dodger	OpenCode + Roblox MCP	1.33	0.67	1.33	0.67	1.33	0.67
Platformer	Slop Engine	1.33	1.33	0.67	2.00	2.00	2.00
Platformer	OpenCode + Babylon.js	1.33	2.00	2.00	2.00	2.00	2.00
Platformer	OpenCode + Roblox MCP	2.00	1.67	2.00	2.00	2.00	2.00

Table 8: Mean score per rubric dimension for each game-scenario cell. Each dimension is scored 0/1/2. Each row averages across the three runs for that game and scenario.

Scenario	Game	Run	Dur (s)	In tok	Out tok	Iter	Tools	Quality (/12)
OpenCode + Babylon.js	breakout	1	183	32878	7566	1	18	12
OpenCode + Babylon.js	breakout	2	228	38622	8465	1	10	12
OpenCode + Babylon.js	breakout	3	450	56546	15277	4	29	11
OpenCode + Babylon.js	dodger	1	131	26047	6816	1	16	12
OpenCode + Babylon.js	dodger	2	154	28093	8117	1	23	12
OpenCode + Babylon.js	dodger	3	321	78410	12825	4	29	9
OpenCode + Babylon.js	platformer	1	197	33905	8458	2	14	12
OpenCode + Babylon.js	platformer	2	300	44145	8599	2	17	12
OpenCode + Babylon.js	platformer	3	516	82847	12246	4	24	10
OpenCode + Roblox MCP	breakout	1	1081	247815	55408	3	100	4
OpenCode + Roblox MCP	breakout	2	1233	292740	44685	2	99	10
OpenCode + Roblox MCP	breakout	3	688	131717	13516	1	60	12
OpenCode + Roblox MCP	dodger	1	251	45165	8167	1	19	0
OpenCode + Roblox MCP	dodger	2	717	171299	19555	1	83	12
OpenCode + Roblox MCP	dodger	3	881	263538	24306	4	74	6
OpenCode + Roblox MCP	platformer	1	407	115593	6805	1	33	12

Scenario	Game	Run	Dur (s)	In tok	Out tok	Iter	Tools	Quality (/12)
OpenCode + Roblox MCP	platformer	2	2836	1023161	66891	3	325	12
OpenCode + Roblox MCP	platformer	3	869	155975	18994	4	83	11
Slop Engine	breakout	1	363	1417552	28805	6	108	10
Slop Engine	breakout	2	136	402589	8792	3	35	12
Slop Engine	breakout	3	316	1253740	20179	7	79	5
Slop Engine	dodger	1	193	407340	10754	5	62	8
Slop Engine	dodger	2	405	947267	22198	6	85	7
Slop Engine	dodger	3	385	1322294	18784	6	72	5
Slop Engine	platformer	1	343	1172949	26501	6	128	9
Slop Engine	platformer	2	343	1207013	24729	6	133	10
Slop Engine	platformer	3	339	886829	22230	6	109	9

Table 9: Full per-run quantitative results across all 27 runs. Quality is reported on a 0-12 scale.

8.4 Appendix D: Tool-calling Sequence Diagram

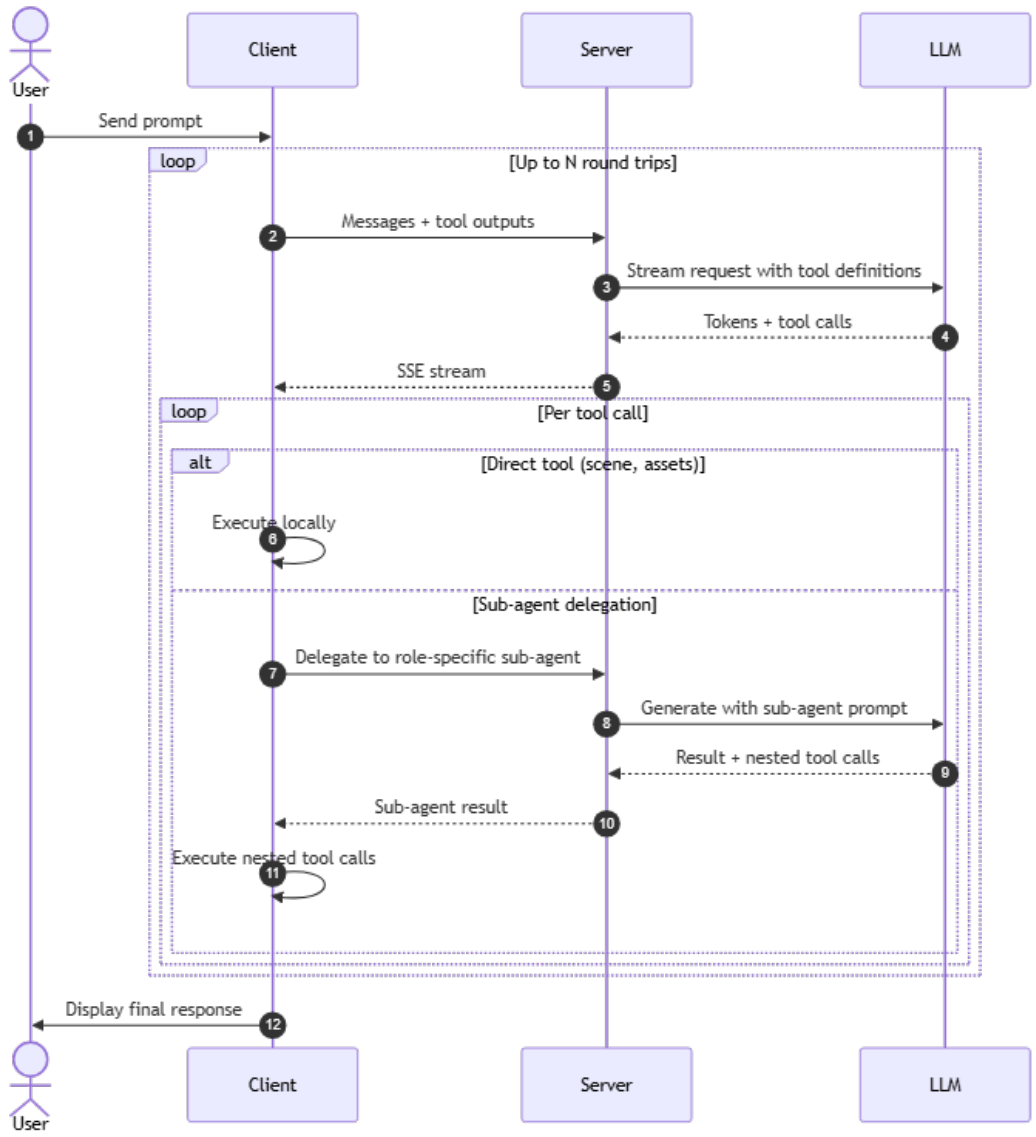

Figure 18: Sequence of one user turn, including the orchestrator and subagent loop.

Figure 18 shows a single user turn in more detail. The orchestrator receives the prompt, may inspect the scene, may ask for clarification or present a plan, and can then delegate implementation work to one or more subagents. Each subagent performs its own model call and may execute multiple tools before returning a summary. These tool calls can modify scripts, create scene objects, import assets, run the simulation, or inspect errors. The final result is streamed back to the user through the assistant interface.

8.5 Appendix E: Summary of Available Tools

Tool group / tool	Available to	Purpose
`get_scene`	All agents	Returns a simplified representation of the current scene hierarchy and simulation state.
`spawn_agent`	Orchestrator	Invokes a specialist subagent with a task description and role-specific tool set.
`ask_clarification`	Orchestrator	Requests structured clarification from the user during planning.
`present_plan`	Orchestrator	Presents a proposed implementation plan before execution.
`add_mesh`, `add_light`, `update_node`, `delete_node`	Scene agent	Creates, modifies, and removes scene nodes.
`create_group`, `set_parent`, `bulk_scene`	Scene agent	Organizes hierarchy and applies batched scene edits.
`list_assets`, `import_asset`, `save_prefab`	Scene agent, Asset agent	Finds available assets, imports them into the scene, and saves reusable prefabs.
`list_scripts`, `read_script`, `create_script`, `edit_script`, `delete_script`	Scripting agent, UI agent	Lists, reads, creates, modifies, and deletes project scripts.
`attach_script`, `detach_script`	Scripting agent, UI agent	Connects or removes scripts from scene nodes.
`lookup_scripting_api`	Scripting agent, UI agent	Retrieves relevant documentation from the local scripting API reference.
`play_simulation`, `stop_simulation`, `sleep`	Scripting agent, UI agent, Testing agent	Controls the runtime simulation and waits between observations.
`get_console_logs`	Scripting agent, UI agent, Testing agent	Returns editor console output, including script logs and runtime errors.
`run_autonomous_test`	Scripting agent, UI agent, Testing agent	Runs timed input tests and records observations from the running project.
`generate_image`, `generate_tripo_mesh`	Asset agent	Generates image and 3D model assets through external providers.

Tool group / tool	Available to	Purpose
`list_image_assets`, `apply_texture`, `remove_texture`	Asset agent	Finds image assets and applies or removes them as textures.
`update_material_properties`, `set_billboard_mode`	Asset agent	Updates visual material settings and billboard rendering behavior.
`delete_asset`, `create_asset_folder`	Asset agent	Maintains the asset store structure.

Table 10: Engine tool surface and agent availability