GD-KS: A Spec-Driven Multi-Agent Framework for LLM-Assisted Game Development

Murilo Mazzotti Silvestrini[1], **Leonardo Tórtoro Pereira**[2]

[1] State University of Campinas (UNICAMP)
Institute of Mathematics, Statistics and Scientific Computing
R. Sérgio Buarque de Holanda, 651 – 13083-859 – Campinas – São Paulo – Brazil

m261854@dac.unicamp.br

[2]São Paulo State University (UNESP)
Institute of Geosciences and Exact Sciences
Rua 24 A – 13506-900 – Rio Claro – São Paulo – Brazil

leonardo.t.pereira@unesp.br

Abstract. Introduction: *LLM-based coding assistants operate mainly at the level of code completion and do not model the process by which a software artifact is produced. This gap is acute in game development, whose pre-production (concept, design, narrative, art, audio) generates documentation whose volume and interdependencies rival the implementation effort — a burden falling disproportionately on indie studios and solo developers.* ***Objective:*** *To design a multi-agent framework for indie and solo developers that applies Spec-Driven Development (SDD) to the game development lifecycle and decouples engine-agnostic design from engine-specific implementation.* ***Steps:*** *We designed GD-KS, an open-source four-phase spec-driven pipeline (ideation, design, planning, engine) with up to 32 specialized agents. Its main technical innovation is a plugin-based engine-profile abstraction isolating engine-specific knowledge from design artifacts, currently supporting Unreal Engine 5, Godot 4 and Unity 6, complemented by a single-source-of-truth project state with append-only audit trail. The framework is validated through schema validation, 240 automated tests, and multi-engine installation scenarios.* ***Expected Results:*** *The pipeline enforces documentation discipline that small teams often cannot maintain manually; the engine-profile abstraction lowers the cost of evolving to new engines; the project state supports fragmented sessions typical of solo work. A user study is planned as the next step.*

Keywords *game development, large language models, spec-driven development, AI-assisted software engineering.*

1. Introduction

The integration of Large Language Models (LLMs) into software development has produced tools — GitHub Copilot, Cursor, Claude Code — that assist developers with code completion, refactoring, and conversational pair-programming [GitHub 2024, Anysphere 2024, Anthropic 2024]. These tools are effective during implementation, but lack structural awareness of the *process* by which a project is built.

Game development makes this limitation visible. Producing a game spans several disciplines — narrative, game and level design, art and audio direction, production,

engineering — whose decisions cascade forward in well-documented ways [Schell 2019, Keith 2010] and generate pre-production documentation (GDD, narrative, art and audio bibles) whose volume rivals the code written later. In larger studios this work is distributed across specialized teams; in indie and solo projects, which account for a substantial share of releases on major platforms, the same artifacts fall on a single person or a handful of generalists — the practitioners most in need of process support, and the least served by code-level AI assistants.

Three recurring problems characterize LLM-assisted workflows in this setting. First, *premature phase transitions*: without guardrails, practitioners skip from partial design documentation to implementation, incurring rework avoidable upstream. Second, *memory loss across sessions*: LLM conversations are ephemeral, and decisions made in one session tend to be re-derived inconsistently in the next. Third, *engine coupling of process knowledge*: game design is largely engine-agnostic, yet tools rarely separate engine-specific vocabulary from design artifacts, raising migration costs for small teams already near capacity.

This paper presents GD-KS (Game Development Knowledge System), an open-source framework orchestrating up to 32 specialized LLM agents through a four-phase spec-driven pipeline, with explicit focus on indie and solo developers. Building on Spec-Driven Development (SDD) — the methodology adopted by BMAD-METHOD [BMAD 2024] and recent multi-agent frameworks, in which specifications written by specialist agents drive downstream artifacts and code generation — GD-KS contributes:

- **A domain-specialized SDD pipeline for game development** (Sec. 3): four discipline-aligned phases (ideation, design, planning, engine) with agent teams, deliverable templates, and seven presets that mirror practitioner workflows from solo to studio scale.
- **A plugin-based engine-profile abstraction** (Sec. 3.3): an architecture that isolates engine-specific knowledge, so that engine-agnostic design artifacts coexist with engine-specific implementation specifications. Three engines ship as first-class citizens (Unreal 5, Godot 4, Unity 6).
- **A single-source-of-truth project state** with append-only event log and periodic checkpoints (Sec. 3.4), supporting deterministic recoverability across LLM sessions.

The remainder of the paper is organized as follows. Section 2 reviews related work. Section 3 describes GD-KS. Section 4 reports implementation details and preliminary validation. Section 5 concludes with limitations and future work.

2. Related Work

GD-KS sits at the intersection of three research areas: (i) general-purpose multi-agent LLM frameworks for software engineering; (ii) specification-driven development (SDD) methodologies; and (iii) game-specific LLM applications. This section reviews each strand and concludes with a clear positioning of GD-KS's contributions.

2.1. Multi-Agent LLM Frameworks for Software Engineering

The application of LLMs across the software engineering lifecycle has been surveyed extensively [Hou et al. 2024]. Early LLM-based code assistants (GitHub

Copilot [GitHub 2024], Cursor [Anysphere 2024], Claude Code [Anthropic 2024]) operate at the granularity of in-IDE code completion and refactoring but lack awareness of the development process or the ability to coordinate multiple specialised agents. Subsequent research has introduced more structured orchestration. AutoGen [Wu et al. 2023] provides a conversation-based framework where agents converse, delegate tasks, and use tools, but offers no built-in lifecycle guidance. CrewAI [CrewAI Inc. 2024] focuses on role-based collaboration with deterministic workflows; its domain-agnostic design requires bespoke configuration for any specific discipline. CAMEL [Li et al. 2023] pioneered role-playing between agents with task specification and critique mechanisms, demonstrating emergent cooperation but without embedding a structured development pipeline.

Closer to GD-KS are frameworks targeting the software engineering lifecycle. ChatDev [Qian et al. 2023] adopts a chat-chain paradigm in which managers, coders, reviewers, and testers simulate a software company, producing executable code from a natural language task description. MetaGPT [Hong et al. 2023] encodes standard operating procedures as prompt sequences, achieving structured output (requirement, design, coding) but remaining domain-general. OpenDevin [OpenDevin Contributors 2024] and Devika [StitionAI and Contributors 2024] are open-source AI software engineers that integrate planning, coding, and debugging loops; none, however, offers a pre-defined pipeline for the pre-production and engine-coupling problems of game development.

2.2. Specification-Driven Development and the Rise of SDD Frameworks

Specification-Driven Development treats a formal, machine-readable specification as the source of truth, from which code is generated or verified [Piskala 2026]. Several SDD toolchains have emerged in recent years, including GitHub Spec-Kit [GitHub, Inc. 2025], Amazon Q Developer / Kiro [Amazon Web Services, Inc. 2025], OpenSpec [OpenSpec Contributors 2025], Tessl [Tessl, Inc. 2025], and BMAD METHOD [BMAD 2024], all advocating a spec-first workflow in which natural-language specifications drive implementation.

Of these, BMAD METHOD is the closest precursor to GD-KS. It provides a multi-agent framework with specialised roles (Analyst, Architect, Developer, QA) and a task-based workflow, and uses specification files — product requirements, epics, and stories — to guide agent behaviour. GD-KS adapts this SDD philosophy to game development along three lines. First, the specification pipeline is aligned to the four natural phases of game creation (ideation, design, planning, engine), so that the artifacts produced at each stage match the deliverables practitioners already recognise. Second, a plugin-based engine abstraction cleanly separates engine-agnostic design artifacts from engine-specific implementation details, supporting Unreal Engine 5, Godot 4, and Unity 6 without coupling design to a single target. Third, the framework is positioned for indie and solo developers, combining provider-neutral tooling that incurs no LLM API costs with a state management layer designed for the fragmented sessions characteristic of small-team work.

2.3. Game-Specific LLM Agent Frameworks

A growing body of work applies LLMs to game development, but much of it focuses on either procedural content generation or end-to-end code synthesis without addressing the documentation bottleneck.

UnrealLLM [Tang et al. 2025] presents a multi-agent framework that translates natural language descriptions into executable Procedural Content Generation (PCG) blueprints and assets within Unreal Engine 5, demonstrating impressive 3D scene generation. GameGPT [Chen et al. 2025] proposes a collaborative multi-agent system for automating game development, with specific methods to mitigate hallucination and redundancy during planning, task identification, and implementation. AutoUE [Yin et al. 2026] coordinates multiple agents to generate complete 3D games in Unreal Engine, covering model retrieval, scene generation, gameplay code synthesis, and automated testing.

Several works focus on Game Design Document (GDD) processing. Hassan [Hassan 2025] presents an end-to-end system that parses GDDs, extracts structured specifications, and synthesises Unity-compatible C# code. Serra et al. [Serra et al. 2025] describe PromptingGameCraft (PGC), an SBGames 2025 publication that takes a GDD as input and automatically generates structured JSON game design files, class diagrams, and game code — a directly relevant but functionally complementary effort, since PGC is a GDD-to-code pipeline whereas GD-KS is a process-oriented SDD framework.

Surveys such as Hu et al. [Hu et al. 2025] and Gallotta et al. [Gallotta et al. 2024] provide taxonomies of LLM-based game agents and a roadmap for LLMs in games, respectively. These surveys confirm the absence of a process-oriented SDD framework tailored to game development's pre-production and engine abstraction needs.

2.4. Game Documentation and Process Support for Indie and Solo Developers

The practitioner literature [Schell 2019, Keith 2010] underlines the centrality of the Game Design Document and the cascade from concept to implementation. Yet the tooling landscape for indie and solo developers remains fragmented. General documentation platforms (Notion, Nuclino [Nuclino, Inc. 2016]) and project management tools (Trello, Jira) are often adapted for game development, but they lack built-in phase gating and deliverable templates. Dedicated game-project tools such as Codecks [Codecks, Inc. 2015], HacknPlan [HacknPlan Ltd. 2017], and Grapken [Grapken Contributors 2024] provide kanban boards, GDD editors, and backlog management; however, none integrates with LLM agents, nor imposes a structured specification pipeline tied to code generation. SoloMode [SoloMode, Inc. 2026] is a recent all-in-one tool for solo game developers, offering a GDD editor, milestone tracker, and Steam market research, again without AI agent orchestration.

2.5. Positioning of GD-KS

To the best of our knowledge, no prior publicly available framework integrates a domain-specialised SDD pipeline for game development, a plugin-based engine abstraction, a single-source-of-truth project state with deterministic recoverability across sessions, and a provider-neutral design that never invokes LLMs directly and therefore incurs no API

costs. Table 1 summarises this positioning against representative frameworks. GD-KS therefore complements rather than replaces existing game-development AI tools: it supplies the process architecture and engine abstraction layer that sit above both LLM agents and game engines.

Table 1. Comparison of selected multi-agent and game-development AI frameworks.

Framework	Domain	Pipeline	Eng. abs.	Audience	API cost	Open
AutoGen [Wu et al. 2023]	Gen.	Flexible conv.	–	Researchers	Yes	✓
CrewAI [CrewAI Inc. 2024]	Gen.	Role-based wf.	–	General devs	Yes	✓
ChatDev [Qian et al. 2023]	Gen.	Chain-of-managers	–	General devs	Yes	✓
BMAD [BMAD 2024]	Gen.	PRD → code	–	AI-assisted	No	✓
UnrealLLM [Tang et al. 2025]	Game	Multi-agent 3D	Unreal	Game devs	Yes	✓
GameGPT [Chen et al. 2025]	Game	Collaborative	–	Game devs	Yes	–
PGC [Serra et al. 2025]	Game	GDD → code	–	Game devs	Yes	–
GD-KS	**Game**	**Spec-driven 4-phase**	**Plugin (3)**	**Indie/solo**	**No**	✓

Gen. = general purpose; Eng. abs. = engine abstraction; API cost = the framework issues LLM API calls on the user's behalf; "–" = not applicable or not reported.

3. The GD-KS Framework

3.1. Architecture overview

GD-KS is a Node.js (ESM) package distributed via npm, organized into an engine-agnostic core (state manager, transition validator, preset manager, template renderer), a modules layer with discipline teams and engine plugins, and an installer that validates every YAML artifact against JSON Schema (Draft 2020-12) [Pezoa et al. 2016]. The framework is provider-neutral: agent definitions authored in YAML are compiled to Markdown and placed in the user's project, where any AI-capable IDE (Cursor, Windsurf, VSCode with Copilot, Claude Code) loads them as context. GD-KS does not invoke LLMs directly, which keeps indie and solo developers free to choose their own LLM provider and avoid recurring API costs they would not otherwise incur.

3.2. Four-phase pipeline and agent teams

GD-KS organizes the lifecycle into four sequential phases, each owned by a specialized team. An orchestrator agent (`gdks-master`) is always installed and owns cross-cutting workflows such as initialization, status, collaboration, and tutorial. Figure 1 gives a consolidated view of the pipeline: the four phases in order, the phase transitions that gate each advance, and the three engine profiles that can be slotted into Phase 4. Throughout this section we refer to *Cosmic Explorer*, the reference project bundled with the framework and described in Section 4.2, to make the artifacts of each phase concrete.

Each specialist agent is declared in a YAML file with persona, principles, critical actions, and user-facing commands prefixed with $\star$. At install time, a compiler translates each YAML file into a Markdown file with XML-like semantic blocks that IDEs and LLMs parse reliably. Phase 4 is engine-specific: depending on the engine selected, either the UE5 team (5 agents), the Godot 4 team (4 agents), or the Unity 6 team (4 agents) is installed. Engine teams do not generate code themselves; they produce architectural

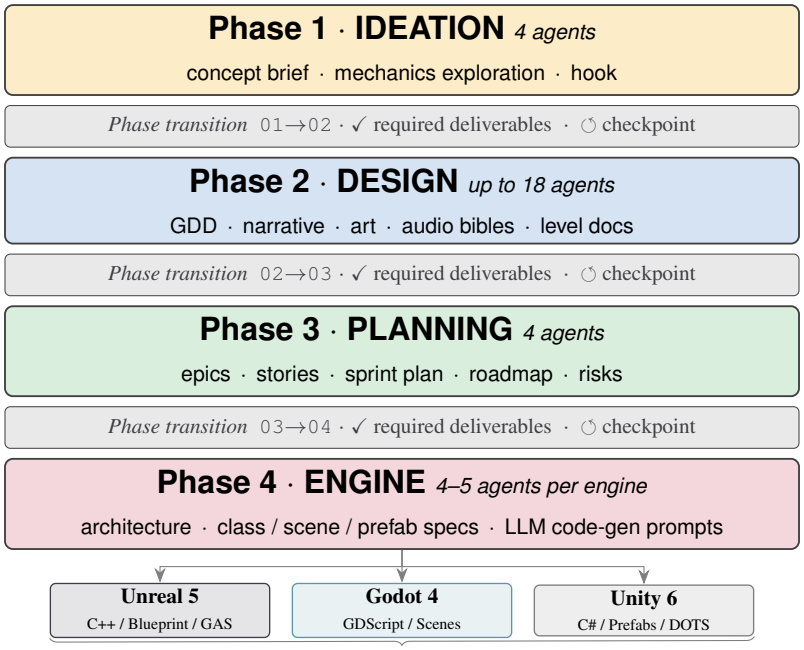

Figure 1. GD-KS pipeline. Four sequential phases are separated by phase transitions; each transition checks the presence of required deliverables and saves a checkpoint before advancing state.

specifications and LLM-ready prompts that the user pastes into a chosen LLM for code synthesis. Seven presets package common agent subsets, ranging from *minimal* (roughly 8 agents, aimed at game jams and hobby projects) through *solo-indie* (roughly 16 agents, the default, aimed at one- or two-person teams working over three to eighteen months) up to *full-studio*, in which all 32 agents are active.

3.3. Plugin-based engine profiles

Each engine module ships four artifacts: `engine-profile.yaml` (capability declaration), `module.yaml`, an `agents/` directory, and a `workflows/` directory. The profile declares paradigms (languages, ECS, visual scripting, rendering), capabilities (platforms, multiplayer, physics, audio), `agents_provided`, and `planning_template_hints` consumed by the Planning phase. Table 2 summarizes the three shipped profiles.

Table 2. Engine profile highlights for the three shipped engines.

	Unreal 5	Godot 4	Unity 6
Languages	C++, BP	GDScript, C#	C#
Rendering	Nanite+Lumen	Forward+	URP/HDRP
ECS	–	–	DOTS
GAS	✓	–	–
Engine agents	5	4	4

A template conditional renderer resolves `{{#engine <id>}}...{{/engine}}` blocks and `{{engine.id}}` substitution. The default

story template uses these constructs to emit engine-specific implementation notes from a single source; Figure 2 shows the same template excerpt as rendered for two different target engines. Design and planning artifacts are, by construction, engine-agnostic, so only Phase 4 outputs are engine-specific. Adding a new engine such as Bevy, O3DE, or GameMaker requires no framework modification. This separation matters in particular for indie and solo developers, who often switch engines between projects depending on scope, licensing, or publisher constraints.

```
## Implementation Notes
### Unreal Engine 5 Implementation
- Core logic as a C++ UObject subclass
- Tunables via UPROPERTY(EditAnywhere,
  BlueprintReadWrite)
- Blueprint child class in Content/ for
  content-side configuration
- Profile with Unreal Insights
- Test in PIE, then in a packaged build
Suggested class location:
Source/<Project>/Gameplay/
```

```
## Implementation Notes
### Godot 4 Implementation
- Core logic in GDScript, statically
  typed (var speed: float = 100.0)
- Tunables via @export annotations
- signals for loose coupling between
  nodes
- Autoload for global state (EventBus)
- Profile with the built-in profiler
Suggested locations:
scenes/gameplay/, scripts/gameplay/
```

Figure 2. Excerpt of the same planning story template rendered under two engine profiles.

3.4. Project state, audit trail, and interaction loop

Every installation produces `_gdks/_state/project-state.yaml`, the single source of truth for the project: metadata, active preset and target engine, per-phase progress and deliverable verification, a decision log, and an open-question log with states `pending`, `answered`, `dropped`, or `blocker`. The file is schema-validated before every write. Two complementary artifacts accompany it: an append-only event log (`events.ndjson`) tolerant to corrupt lines, and timestamped checkpoints taken before every handoff and every rollback.

Figure 3 shows how state, agents and the user interlock in practice. At agent invocation, GD-KS injects a `<project_state_context>` block into the compiled agent Markdown, so that every agent, across any LLM session, sees up-to-date decisions and deliverable status without the user manually re-supplying context — an important property for solo developers, who often work on a single project across dozens of fragmented sessions over weeks or months.

3.5. Installation

A developer moves from `npx gd-ks install` to a fully configured project in under a minute. A wizard collects the project name, the preset, the teams to install, and, when an engine team is selected, the target engine; a confirmation prompt summarizes the plan before any file is written. Validation is front-loaded: the JSON Schemas mentioned in Section 3 are applied to every artifact *before* disk writes begin, so an invalid configuration aborts cleanly without leaving partial state.

4. Implementation and Preliminary Validation

4.1. Implementation

GD-KS v0.4.1-beta.2 is released under the MIT license and distributed via npm as the `gd-ks` package [Silvestrini 2026b, Silvestrini 2026a]. Quantitative metrics of the current

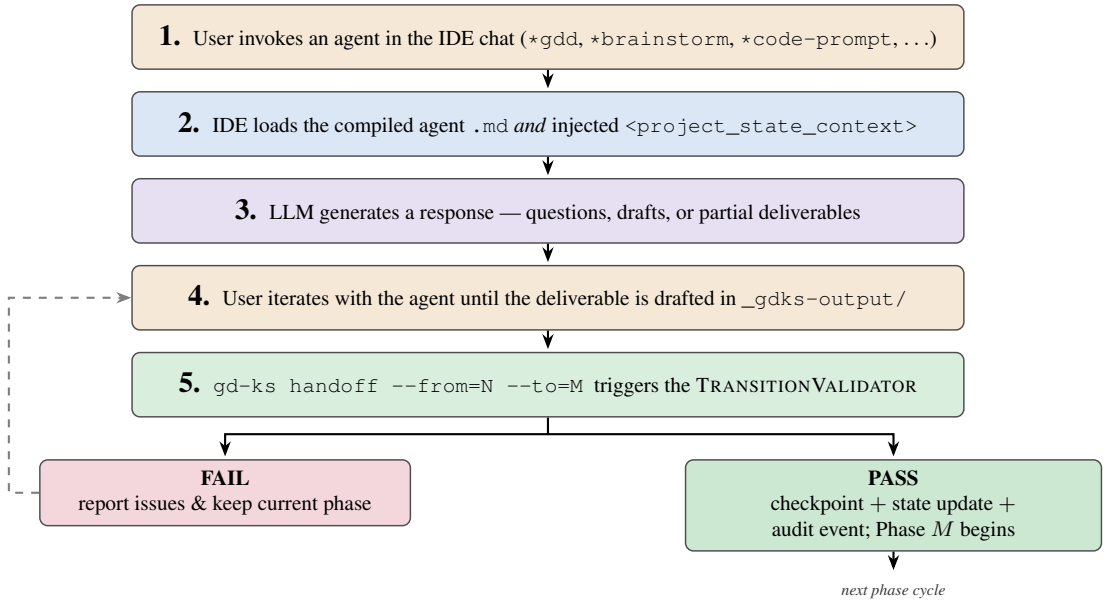

Figure 3. User interaction loop. The compiled agent Markdown is loaded together with live project state, so that every session inherits earlier decisions.

distribution are summarized in Table 3. Continuous integration runs on every push: lint (`eslint --max-warnings=0`), schema validation across all YAML artifacts, and the full test suite under Node's built-in `node:test` runner. Coverage is measured with `c8`, and core modules exhibit 93–98% line coverage.

Table 3. Key metrics of GD-KS v0.4.1-beta.2.

Metric	Value
Agents (shipped / max active per project)	40 / 32
Workflows (`workflow.yaml`)	59
JSON Schemas	7
Presets	7
Engines supported	3
Automated tests (unit + integration)	240
Global test coverage	93%

4.2. Preliminary validation and the reference project

We defer a user study to future work and, for this stage, validate GD-KS internally along three axes reflecting the framework's correctness criteria. Phase transition correctness is exercised by 28 tests covering the pre-transition checks (required deliverables, completion thresholds, per-preset conditionals), the atomic state commit (write-to-temp, `fsync`, rename), and the `--dry-run` and `--force` flags. Preset integrity is verified by a parameterized test installing each of the seven presets and checking that the resulting agent directory matches the active set, together with a five-case multi-engine suite confirming that Godot or Unity installations produce the expected team without leaking UE5-specific agents. End-to-end coverage is provided by an 8-case test simulating the full lifecycle: installation, deliverable registration, phase transition, checkpoint verification, and rollback.

The framework ships a reference project, *Cosmic Explorer*, a low-gravity puzzle-platformer built around three mechanics — Drift, Anchor and Resonance Pulse. It consists of six hand-authored Markdown documents that instantiate the deliverables of all four phases: a concept brief (Phase 1); a GDD and a design-pillars document (Phase 2); epics and a roadmap (Phase 3); and an architecture specification (Phase 4). These documents are exemplars rather than generated output: they show what a well-formed deliverable looks like at each stage, and are deliberately authored by hand so that the target is stable across LLM providers. The command `gd-ks tutorial` copies them into a sandbox folder with a separate `tutorial-state.yaml`, leaving any real project untouched, and the `gdks-master` agent then narrates a nine-step, fifteen-minute walkthrough that references each document as a concrete example. *Cosmic Explorer* also served as the common premise for the pilot evaluation reported next, so that every IDE and LLM combination was exercised against an identical design target.

4.3. Internal pilot evaluation across IDEs and LLMs

To complement the test suite, we conducted a structured internal pilot evaluation across four IDEs — Cursor, Antigravity, Claude Code, and VS Code with GitHub Copilot — and four LLMs: Claude Opus 4.x, Sonnet 4.x, GPT-5.4, and Gemini 2.x. Each combination ran the same canonical workflows: initialization, ideation kickoff, drafting of GDD sections, and a complete phase transition from design to planning.

Four pre-declared criteria scored each combination: (1) *context injection fidelity*, whether the IDE surfaces the `<project_state_context>` block; (2) *command adherence*, whether the LLM respects $\star$-prefixed commands across turns; (3) *artifact completeness*, whether deliverables satisfy the template's section structure; and (4) *session continuity*, whether decisions are inherited across sessions. Combinations were rated best fit (●), acceptable (○), or partial (–). Results appear in Table 4.

Table 4. Pilot evaluation across IDEs and LLMs.

	(1)	**(2)**	**(3)**	**(4)**
IDE				
Cursor	●	●	●	●
Claude Code	●	●	●	●
Antigravity	●	○	●	○
VS Code + Copilot	○	○	–	–
LLM				
Claude Opus 4.x	●	●	●	●
GPT-5.4	●	●	●	●
Claude Sonnet 4.x	●	○	●	○
Gemini 2.x	○	–	○	○

(1) context injection fidelity; (2) command adherence; (3) artifact completeness; (4) session continuity.
Symbols: ● best fit; ○ acceptable; – partial.

On the IDE axis, Cursor and Claude Code matched all four criteria, since their native rule-loading conventions align directly with GD-KS's compiled output. Antigravity performed acceptably, with occasional drift attributable to its evolving rule-priority semantics. VS Code with Copilot diverged the most: its single instruction-file entry point has a tight effective context budget that the full project state plus active agent definition

routinely exceeds. In our runs, the material dropped first was consistently the tail of the injected state block, so that agents produced structurally valid deliverables while silently losing earlier design decisions. On the LLM axis, Claude Opus 4.x and GPT-5.4 reached best fit across all four criteria, attributable to large effective context windows (200K and 1M tokens respectively) and strong adherence to the XML-tagged structures used in GD-KS's compiled agents. Sonnet 4.x preserved structural fidelity but showed occasional command drift in extended sessions. Typically the agent began answering in free-form prose instead of following its declared $\star$ command menu after roughly a dozen turns, requiring an explicit re-invocation of the agent to recover. Gemini 2.x performed partially, which we attribute to instruction-following idioms of that generation rather than to context capacity.

These findings are limited by three factors: the evaluation was conducted by the authors rather than by external participants; only the bundled *Cosmic Explorer* premise was used across all combinations; and the rating scheme is qualitative without inter-rater agreement procedures. We report these results as feasibility evidence motivating the controlled user study planned in Section 5.

5. Conclusion and Future Work

We have shown how a multi-agent LLM pipeline for game development can be organized around Spec-Driven Development, a plugin-based engine-profile abstraction, and a single-source-of-truth project state with append-only audit trail. GD-KS targets indie studios and solo developers, supports three engines as first-class citizens, ships seven presets ranging from an eight-agent minimal configuration for game jams to a thirty-two-agent full-studio configuration, and passes 240 automated tests at 93% coverage.

Limitations remain. GD-KS does not invoke LLMs directly, a deliberate choice preserving provider neutrality at the cost of some user friction. Agent skills are implicit in principles and menus rather than first-class. Engine switching mid-project is not supported. No user study has yet been conducted, and the pilot reported above was run by the authors on a single reference premise. Future work is led by a controlled study with indie and solo developers, in which participants take a game concept of their own from ideation through the design-to-planning transition, with and without the framework, and deliverable completeness against template structure, phase-transition discipline, and time-on-task are recorded as dependent measures; a first-class skills taxonomy follows.

GD-KS v0.4.1-beta.2 is available under the MIT License at `https://github.com/muriloms/gd-ks` and on npm as `gd-ks` (`https://www.npmjs.com/package/gd-ks`); the framework installs with `npx gd-ks install`.

Acknowledgments

This work was developed within RE-G3X, a research group dedicated to the scientific study of digital games.

References

Amazon Web Services, Inc. (2025). Amazon Q Developer (Kiro). `https://aws.amazon.com/q/developer/`. Accessed: 2026-07-26.

Anthropic (2024). Claude Code: Agentic Coding in the Terminal. https://docs.claude.com/en/docs/claude-code/overview. Accessed: 2026-07-26.

Anysphere (2024). Cursor — The AI Code Editor. https://cursor.com. Product documentation. Accessed: 2026-07-26.

BMAD (2024). BMAD-METHOD: Breakthrough method of agile AI-driven development. https://github.com/bmad-code-org/BMAD-METHOD. Accessed: 2026-07-26.

Chen, D., Zhang, H., Wang, H., Huo, Y., Li, Y., e Wang, J. (2025). GameGPT: Multi-agent collaborative framework for game development.

Codecks, Inc. (2015). Codecks: Project management for game developers. https://www.codecks.io/. Accessed: 2026-07-26.

CrewAI Inc. (2024). CrewAI: Framework for orchestrating role-playing, autonomous AI agents. https://github.com/crewAIInc/crewAI. Accessed: 2026-07-26.

Gallotta, R., Todd, G., Zammit, M., Earle, S., Liapis, A., Togelius, J., e Yannakakis, G. N. (2024). Large language models and games: A survey and roadmap. *IEEE Transactions on Games*, pages 1–18.

GitHub (2024). GitHub Copilot — Your AI Pair Programmer. https://github.com/features/copilot. Accessed: 2026-07-26.

GitHub, Inc. (2025). GitHub Spec-Kit. https://github.com/github/spec-kit. Accessed: 2026-07-26.

Grapken Contributors (2024). Grapken: Visual node-based editor for indie game design docs. https://grapken.com/. Accessed: 2026-07-26.

HacknPlan Ltd. (2017). HacknPlan. https://hacknplan.com/. Accessed: 2026-07-26.

Hassan, A. (2025). Automated Unity game template generation from GDDs via NLP and multi-modal LLMs.

Hong, S., Zheng, X., Chen, J., Cheng, Y., Wang, J., Zhang, C., Wang, Z., Yau, S. K. S., Lin, Z., Zhou, L., et al. (2023). MetaGPT: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., e Wang, H. (2024). Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79.

Hu, S., Huang, T., Liu, G., Kompella, R. R., Ilhan, F., Tekin, S. F., Xu, Y., Yahn, Z., e Liu, L. (2025). A survey on large language model-based game agents.

Keith, C. (2010). *Agile Game Development with Scrum*. Addison-Wesley Professional.

Li, G., Hammoud, H., Itani, H., Khizbullin, D., e Ghanem, B. (2023). CAMEL: Communicative agents for "mind" exploration of large language model society. *arXiv preprint arXiv:2303.17760*.

Nuclino, Inc. (2016). Nuclino: Unified workspace for game design documentation. https://www.nuclino.com/solutions/game-planner. Accessed: 2026-07-26.

OpenDevin Contributors (2024). OpenDevin. https://github.com/OpenDevin/OpenDevin. Accessed: 2026-07-26.

OpenSpec Contributors (2025). OpenSpec. https://github.com/openspec/openspec. Accessed: 2026-07-26.

Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., e Vrgoč, D. (2016). Foundations of JSON schema. In *Proceedings of the 25th International Conference on World Wide Web (WWW)*, pages 263–273. ACM.

Piskala, D. B. (2026). Spec-driven development: From code to contract in the age of AI coding assistants.

Qian, C., Cong, X., Yang, C., Chen, W., Su, Y., Xu, J., Liu, Z., e Sun, M. (2023). ChatDev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*.

Schell, J. (2019). *The Art of Game Design: A Book of Lenses*. CRC Press, 3rd edition.

Serra, C. B., Serra, G. M. B., e de Classe, T. M. (2025). Digital game development using large language models (LLMs): An exploratory study. In *Anais do XXIV Simpósio Brasileiro de Jogos e Entretenimento Digital*, pages 538–549, Porto Alegre, RS, Brasil. SBC.

Silvestrini, M. M. (2026a). GD-KS: Game development knowledge system. https://github.com/muriloms/gd-ks. Open-source repository, version v0.4.1-beta.2. Accessed: 2026-07-26.

Silvestrini, M. M. (2026b). gd-ks npm package. https://www.npmjs.com/package/gd-ks. Version 0.4.1-beta.2. Accessed: 2026-07-26.

SoloMode, Inc. (2026). SoloMode: All-in-one tool for solo game devs. https://www.producthunt.com/products/solomode. Accessed: 2026-07-26.

StitionAI and Contributors (2024). Devika. https://github.com/stitionai/devika. Accessed: 2026-07-26.

Tang, S., Zhao, K., Wang, L., Li, Y., Liu, X., Zou, J., Wang, Q., e Chu, X. (2025). UnrealLLM: Towards highly controllable and interactable 3D scene generation by LLM-powered procedural content generation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 19417–19435, Vienna, Austria. Association for Computational Linguistics.

Tessl, Inc. (2025). Tessl. https://tessl.io/. Accessed: 2026-07-26.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., Awadallah, A. H., White, R. W., Burger, D., e Wang, C. (2023). AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*.

Yin, L., Cheng, W., Qin, Z., Huang, T., Li, Y., e Ding, G. (2026). AutoUE: Automated generation of 3D games in unreal engine via multi-agent systems. In *Findings of the Association for Computational Linguistics: ACL 2026*. Association for Computational Linguistics.