

The Spec-Driven Development Landscape: Where Process Meets Language

An Analysis of OpenSpec, Spec Kit, GSD, Quint, and What They Mean for SpecVerse

Executive Summary

In 2025-2026, a movement called **Spec-Driven Development (SDD)** exploded across the software industry. Tools like GitHub's Spec Kit (75K stars), Fission AI's OpenSpec (28.5K stars), and the GSD framework (26.3K stars) amassed enormous communities around a simple thesis: *stop vibe coding and write specifications first*.

This analysis examines these three tools in depth, maps the broader SDD landscape (including AWS Kiro, TESSL, BMAD, and others), and identifies a structural gap that every SDD workflow tool shares:

None of them has a specification language.

Every SDD *workflow* tool in the market uses natural-language Markdown as its specification format. None has a parser. None has inference. None has schema validation. None has convention processing. None expands minimal specifications into complete architectures.

One tool -- **Quint** (Informal Systems) -- does have a real specification language, but it targets distributed systems and consensus protocols, not the SDD workflow space. Quint validates the thesis that formal languages are the right approach, while occupying a different domain entirely (see the companion [QUINT-ANALYSIS.md](#) for the full analysis).

The SDD movement has built the *process layer* for specification-first development. What it lacks -- and what it desperately needs -- is the *language layer*. That is precisely what SpecVerse provides.

Part 1: Why Spec-Driven Development Exploded

The Vibe Coding Crisis

Andrej Karpathy coined "vibe coding" in early 2025 to describe informal AI-prompted code generation. It went viral as "hot vibe code summer" (Kate Holterhoff's phrase). The appeal was obvious: describe what you want, let the AI build it.

The problems surfaced within months:

- **The 3-month wall.** Codebases outgrew anyone's mental model. AI context windows could only see fragments. No one could trace requirements back to stable specifications.

- **Spaghetti chat.** Prompt histories became unauditable. Requirements drifted silently between conversations.
- **Code quality collapse.** Academic research showed AI-assisted coding increases code complexity by ~41% and static analysis warnings by 30%. LLMs generate vulnerable code at rates between 9.8% and 42.1% across benchmarks.
- **No shared understanding.** Multiple developers vibe-coding the same project produced incoherent architectures.

The industry's response was swift and nearly simultaneous:

Date	Event
Mid-2025	AWS launches Kiro -- first IDE built around SDD
September 2025	GitHub open-sources Spec Kit
Late 2025	Tessl launches Framework (closed beta) and Spec Registry
Early 2026	BMAD hits v6; OpenSpec , GSD , PromptX , specs.md enter the space
March 2026	Kiro announces GA with paid pricing tiers

Thoughtworks identified SDD as one of 2025's key new engineering practices. Martin Fowler's site published a detailed three-tool comparison. RedMonk, InfoQ, The New Stack, and InformationWeek all ran major analyses. The movement was real.

What They All Agree On

Despite their differences, every SDD tool shares three convictions:

1. **Specifications should precede implementation.** Define *what* before *how*.
2. **Specifications should be persistent artifacts.** Not ephemeral chat messages.
3. **AI agents need structured context.** Freeform prompting degrades at scale.

These convictions align precisely with SpecVerse's founding thesis: "Define Once, Implement Anywhere." The market has arrived at the same conclusion. The question is whether it has arrived at the right *solution*.

Part 2: The Three Tools

2.1 OpenSpec (Fission AI)

Repository: github.com/Fission-AI/OpenSpec | **Stars:** ~28.5K | **License:** MIT
Language: TypeScript (98.7%) | **Version:** v1.2.0 | **Releases:** 34

What It Is

OpenSpec is the lightest-weight SDD tool in the market. It provides three slash commands that guide AI coding assistants through a propose-apply-archive workflow:

Command	Purpose
<code>/opsx:propose</code>	Generate a proposal, specification, design, and task checklist
<code>/opsx:apply</code>	Execute implementation tasks from the spec
<code>/opsx:archive</code>	Organize completed work into archive folders

Each change gets a dedicated folder in `changes/` containing four Markdown documents: proposal (why), specification (what), design (how), and tasks (checklist). That's it. The entire framework is a structured-prompting system that turns ephemeral AI conversations into persistent file-based artifacts.

What It Gets Right

Brownfield-first positioning. Unlike most SDD tools that assume greenfield projects, OpenSpec explicitly targets existing codebases. The five founding principles -- fluid over rigid, iterative over waterfall, simple over complex, brownfield-friendly, scalable -- read like a deliberate counter to Spec Kit's perceived heaviness.

Change isolation. Each change lives in its own `changes/` directory, making it trivially reviewable in pull requests. This is a simple, elegant pattern that solves the "where did this requirement come from?" problem without ceremony.

Breadth of integration. OpenSpec works with 20+ AI coding assistants -- Claude Code, Cursor, Windsurf, Copilot, Gemini, and others. The AGENTS.md injection pattern for tool discovery is well-designed.

Minimal output. Where Spec Kit generates ~2,000 lines of Markdown per feature, OpenSpec targets ~250 lines. This directly addresses the review burden that Martin Fowler's team identified as SDD's biggest operational problem.

What It Gets Wrong

No specification semantics. The "specifications" are free-form Markdown with RFC 2119 keywords (MUST, SHOULD) and Given/When/Then scenarios. There is no schema, no type system, no validation. A "spec" that says "users MUST be able to log in" and one that says "the system SHALL authenticate users via OAuth2" are equally valid -- the tool cannot distinguish between a vague intention and a precise requirement.

No expansion or inference. What you write is what you get. If you specify three fields on a model, you get three fields. There are no smart defaults, no convention processing, no inference that "email: Email required unique" implies a uniqueness constraint, a format validator, and a database index.

Telemetry-as-business-model. PostHog analytics are embedded in the CLI. The roadmap points toward "Workspaces" as a paid tier for team collaboration. This suggests open-core ambitions, which means the free version may eventually become a funnel rather than a product.

The Verdict

OpenSpec is a **well-designed workflow wrapper** around AI coding assistants. It solves the real problem of ephemeral chat history by externalizing specifications to files. But it operates entirely at the prompt-engineering layer. It has no more understanding of your specification than the LLM reading its Markdown files.

2.2 Spec Kit (GitHub)

Repository: github.com/github/spec-kit | **Stars:** ~75K | **License:** MIT
Language: Python 3.11+ | **Creator:** Den Delimarsky (now at Anthropic)

What It Is

Spec Kit is the most ambitious SDD tool in the market, providing an eight-command workflow that takes projects from governing principles to working implementation:

Step	Command	Output
1	<code>/speckit.constitution</code>	Governing principles document
2	<code>/speckit.specify</code>	Functional requirements and user stories
3	<code>/speckit.plan</code>	Technical architecture + research + data model
4	<code>/speckit.tasks</code>	Dependency-ordered implementation tasks
5	<code>/speckit.implement</code>	Generated code
+	<code>/speckit.clarify</code>	Structured Q&A for underspecified areas
+	<code>/speckit.analyze</code>	Cross-artifact consistency audit
+	<code>/speckit.checklist</code>	Domain-specific completeness validation

The most distinctive feature is the **constitution** -- a governance document establishing non-negotiable principles (Articles I through IX in the template) that constrain all subsequent AI decisions. Article III might mandate test-first development. Article VIII might prohibit abstraction layers. The constitution persists across all features in a project.

What It Gets Right

The constitution concept. The idea that a project should have immutable governing principles that all AI agents must respect is genuinely valuable. It addresses the "agent compliance" problem -- AI models that ignore instructions or drift from established patterns. The Articles structure gives principles the weight of law rather than suggestion.

Cross-artifact consistency. The `/speckit.analyze` command reads across specifications, plans, and tasks to identify inconsistencies, gaps, and constitutional violations. This is the closest any SDD tool gets to validation -- though the implementation is entirely LLM-based.

Comprehensive artifact structure. A single feature generates `spec.md`, `plan.md`, `research.md`, `data-model.md`, `quickstart.md`, `tasks.md`, and API contracts. This thoroughness means the AI has rich context for implementation.

Community engagement. 75K stars, 6,400 forks, 638 open issues. Even accounting for the GitHub brand halo, this is extraordinary engagement that proves market demand.

What It Gets Wrong

Crushing verbosity. The Scott Logic analysis found that one modest feature generated **2,067 lines of Markdown** across five documents to produce **689 lines of code**. The specification-to-code ratio was 3:1 *in the wrong direction*. This is the opposite of SpecVerse's 4x-7.6x expansion - it's a 0.3x *contraction*.

10x slower than iteration. The same Scott Logic study measured 33.5 minutes of agent time plus 3.5 hours of human review for Spec Kit, versus 8 minutes plus 15 minutes of review for iterative prompting. Despite this overhead, the Spec Kit version still contained "a small, and very obvious, bug."

"Reinvented waterfall." The criticism from Eberhardt (Scott Logic) and Fowler's team is devastating: the workflow -- define requirements, research, plan, implement, verify -- is a textbook waterfall with phase gates. The "clarify" step is the only feedback mechanism before implementation. The Marmelab analysis titled their review "The Waterfall Strikes Back."

No programmatic validation. The `/speckit.analyze` command -- Spec Kit's most sophisticated feature -- is pure LLM reasoning. There is no parser, no schema, no programmatic cross-referencing. The AI reads multiple Markdown files and thinks about whether they're consistent. The quality of this "validation" depends entirely on the model's context window and attention span.

Maintenance crisis. Den Delimarsky left GitHub for Anthropic. In January 2026, users reported "22 PRs opened and exactly zero commits" in a month. Maintenance transferred to @mnriem, but the wobble revealed that this 75K-star project depended on one person. Some users migrated to OpenSpec.

The Verdict

Spec Kit is the **most thorough attempt** at codifying the spec-first workflow, and the constitution concept is genuinely innovative. But it is also the clearest demonstration of the SDD movement's fundamental limitation: **natural-language specifications don't scale**. When your spec is 3x longer than your code and still contains bugs, the specification format itself is the problem -- not the workflow.

2.3 GSD (Get Shit Done)

Repository: github.com/glittercowboy/get-shit-done | **Stars:** ~26.3K | **License:** MIT

Creator: TACHES | **Installed via:** `npx get-shit-done-cc@latest`

What It Is

GSD is the most technically sophisticated of the three tools, but it's not primarily a specification tool -- it's a **context engineering and multi-agent orchestration system** for Claude Code. Its core innovation is solving "context rot" (quality degradation as conversation history grows) through fresh context windows, parallel agent swarms, and file-based inter-agent communication.

The workflow is six phases:

Phase	Command	What Happens
Init	<code>/gsd:new-project</code>	Questioner agent interviews you; 4 parallel researchers investigate; synthesizer creates summary; roadmapper creates phases
Discuss	<code>/gsd:discuss-phase N</code>	Captures implementation preferences and resolves ambiguities
Plan	<code>/gsd:plan-phase N</code>	Planner creates XML task plans; checker verifies (up to 3 iterations)
Execute	<code>/gsd:execute-phase N</code>	Wave-based parallel execution; each agent gets fresh 200K context
Verify	<code>/gsd:verify-work N</code>	Verifier checks against goals; debugger agents handle failures
Complete	<code>/gsd:complete-milestone</code>	Archive, tag release, loop to next phase

The system comprises ~50 Markdown prompt files, 12 custom agent definitions, 29 skills, a Node.js CLI helper, and 2 hooks.

What It Gets Right

Context engineering is the real problem. GSD correctly identifies that the bottleneck in AI-assisted development is not the AI's coding ability but the quality of context it receives. Every design decision serves this insight: fresh context windows per executor, file-based inter-agent communication, selective context loading, and the principle that "deterministic logic belongs in code, not in prompts."

Multi-agent specialisation. Rather than asking one AI to be researcher, architect, coder, and reviewer, GSD spawns specialised agents: 4 parallel researchers, a synthesiser, a roadmapper, a planner, a plan-checker, executors, verifiers, and debuggers. Each agent receives precisely the context it needs and nothing more.

XML task structure. Using XML instead of Markdown for task definitions is a small but significant design choice. Claude models parse XML boundaries more reliably than Markdown headings. The `<task>` structure with `<files>`, `<action>`, `<verify>`, and `<done>` elements gives executors precise, unambiguous instructions.

Atomic git commits. Every completed task produces a commit, making the history bisectable for regression identification. This is software engineering discipline applied to AI-generated code.

Wave-based parallelism. Independent tasks execute in parallel across fresh context windows, while dependent tasks execute sequentially. This is genuine dependency-graph-driven orchestration.

What It Gets Wrong

Token economics are brutal. Issue #120 documents the problem: after v1.5.27, simple tasks consumed 200K+ tokens and 11+ minutes just for planning. One bug fix spawned over 100 agents consuming 10K tokens in 60 seconds. A cost-saving fork (GSD 2.0) was created specifically to address this, adding multi-model routing and adaptive context loading.

Overkill for most tasks. The full swarm -- questioner, 4 researchers, synthesiser, roadmapper, planner, 3x plan-checker iterations, executors, verifiers -- is appropriate for building a new application from scratch. It is absurd for fixing a CSS alignment bug. The `/gsd:quick` mode was added in response to this criticism, but it essentially bypasses the system's value proposition.

No domain model. GSD has no concept of entities, relationships, lifecycles, or domain semantics. It operates at the project-management layer. Its "specs" are natural-language requirements in `REQUIREMENTS.md`, not structured domain models. It cannot reason about whether your Order model should have a lifecycle or whether your API needs pagination.

Claude-specific coupling. The architecture is deeply tied to Claude Code's Task tool, slash commands, hooks, and agent spawning. Ports exist for other tools, but GSD is fundamentally a Claude Code meta-framework.

Spec drift is structural. As the `.planning/` directory grows, there is no mechanism to validate that specifications still match the implemented code. The SUMMARY.md files record what happened, but there is no programmatic check that requirements.md still reflects reality.

The Verdict

GSD is the most **technically interesting** of the three tools. Its context engineering and multi-agent orchestration represent genuine advances in how we work with AI coding agents. But it is a workflow orchestration system, not a specification system. It solves the *execution* problem brilliantly while leaving the *specification* problem untouched. The "specs" in GSD's spec-driven development are still free-form Markdown that no machine can parse, validate, or expand.

Part 3: The Broader SDD Landscape

Beyond the three tools above, the SDD market includes:

AWS Kiro

The only SDD tool that ships as a **standalone IDE** (forked from VS Code). Three-phase workflow: requirements.md, design.md, tasks.md. Features "Steering" (persistent project memory) and "Agent Hooks" (automated triggers on file save/create/delete). Priced at \$0-200/month on a credit system. The most polished commercial offering but the most locked-in - requires AWS account, limited to Claude models, proprietary IDE.

Tessl Framework

A startup building spec-anchored development where **specs live alongside code permanently**. Uses `@generate` and `@test` tags to maintain one-to-one mapping between spec files and code files. Offers bidirectional sync (reverse-engineer specs from code). The Spec Registry is an open marketplace for shared specifications. Closest to SpecVerse's vision in some ways, but still uses Markdown specs with no formal language.

BMAD Method

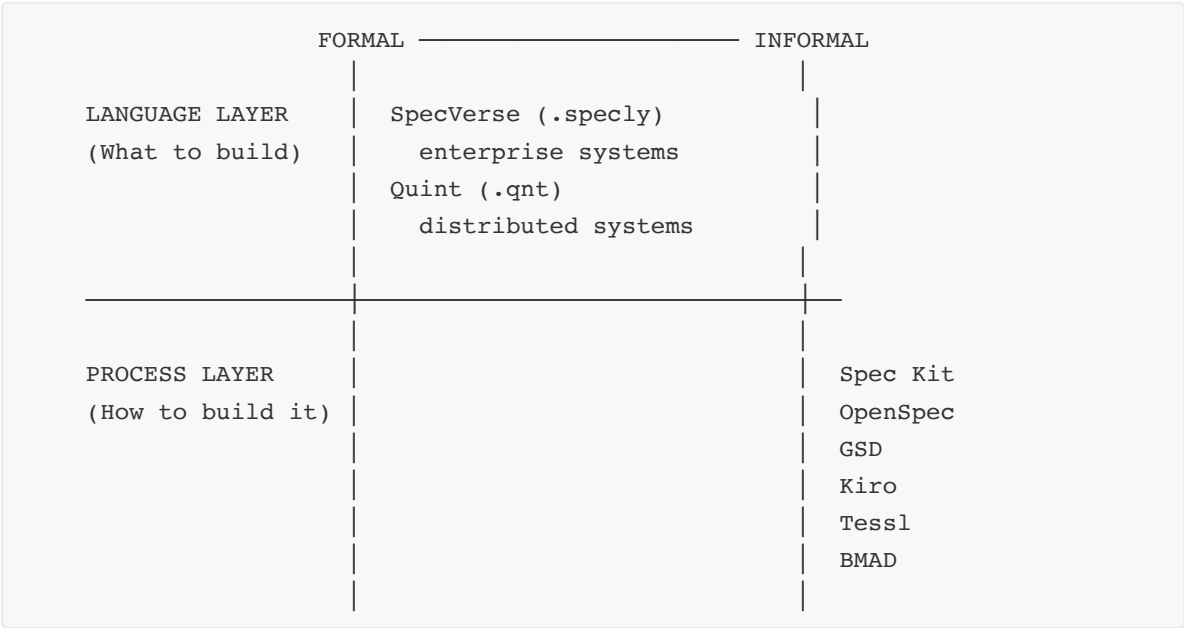
A 19-agent framework that simulates an entire agile team: PM Agent, Architect Agent, UX Designer Agent, Frontend Agent, Backend Agent, etc. Version 6 claims 90% token savings over earlier versions. The most "enterprise" of the community tools, attempting to model team dynamics rather than just document structure.

Quint (Informal Systems)

Not an SDD workflow tool but a **genuine formal specification language** for distributed systems. Built by the Cosmos/Tendermint team, Quint has an ANTLR parser, static type checker with inference, effect system, simulator, REPL, and model checker integration (Apache/Z3). Used in production on 18+ projects including Circle's Malachite consensus engine and Solana's Alpenglowl. Their LLM-era thesis: "LLMs don't think, they translate. Quint's deterministic tools do the reasoning." See [QUINT-ANALYSIS.md](#) for the full analysis.

Quint is significant because it occupies the formal-language quadrant alongside SpecVerse -- but for a completely different domain (consensus protocols vs enterprise systems). It validates the approach while serving different users.

The Full Competitive Map



This is the single most important diagram in this analysis. **Every SDD workflow tool occupies the informal-process quadrant.** The formal-language quadrant -- where specifications have grammar, schemas, parsers, inference, and validation -- contains only SpecVerse (enterprise domain) and Quint (distributed systems domain). These two tools validate each other's approach from independent starting points while serving entirely different audiences.

Part 4: The Structural Gap

What Every SDD Tool Shares

The Martin Fowler article states it explicitly (about the SDD workflow tools):

"None use formal specification languages. All employ natural language markdown exclusively, contrasting with model-driven development's structured DSLs."

This is not an oversight. It is a **design choice born of pragmatism** -- Markdown is universal, requires no learning curve, and works with every AI model. Quint demonstrates the alternative (a formal language with real tooling) but only for distributed systems. For the enterprise domain that SDD tools target, no formal specification language exists in the SDD ecosystem -- except SpecVerse. The Markdown choice creates five structural problems that no amount of workflow innovation can solve:

1. Specifications Cannot Be Validated

When your spec is Markdown, the only way to check it is to read it. There is no schema enforcement, no type checking, no consistency validation. Spec Kit's `/speckit.analyze` command addresses this by asking the LLM to read the Markdown and think about whether it's consistent -- but this is validation-by-opinion, not validation-by-proof. It is exactly as reliable as the AI's attention span.

SpecVerse's JSON Schema validation catches structural errors *before* any AI sees the specification. Missing required fields, invalid types, malformed references -- all detected programmatically, deterministically, instantly.

2. Specifications Cannot Be Expanded

Every SDD tool requires you to write everything explicitly. If you want five fields on a model, you write five fields. If you want an API endpoint with pagination, you describe pagination in detail. If you want standard CRUD operations, you spell them out.

SpecVerse's inference engine (21+ rules) expands minimal specifications 4x-7.6x. Write `email: Email required unique` and get a uniqueness constraint, a format validator, a database index, and appropriate API validation -- all inferred from conventions that the community maintains and shares.

The contrast is stark: **Spec Kit generates 2,067 lines of Markdown to produce 689 lines of code (0.3x)**. SpecVerse generates 4x-7.6x *more* architecture from *less* specification. These are not just different tools -- they embody opposite philosophies about where the work should happen.

3. Convention Knowledge Is Lost

When a senior developer writes a Markdown spec saying "use JWT with httpOnly cookies," they are encoding convention knowledge in prose that no machine can reuse. The next developer must re-specify the same conventions from scratch, or hope the AI remembers them from training data.

SpecVerse's convention processors formalise this knowledge:

Domain	Convention	Formal expansion
Enterprise services	<code>email: Email required unique verified</code>	Schema + DDL + validation + endpoints
Computational logic	<code>standard amortisation</code>	Formula with parameters
Formal verification	<code>totalAmount never negative</code>	Property test + proof harness

Convention processors transform tribal knowledge into reusable, machine-parseable rules. No SDD tool has anything equivalent.

4. Cross-Domain Validation Is Impossible

When specifications are Markdown, there is no way to validate that your API contract matches your data model, that your frontend components reference valid endpoints, or that your lifecycle transitions are consistent across services. These are precisely the kinds of errors that survive all the way to production.

SpecVerse's typed specification format makes cross-domain validation *computable*. If a model's lifecycle defines states `draft -> active -> archived`, the parser can verify that every reference to those states uses valid values. Markdown cannot do this.

5. Specifications Are Not Portable

A Spec Kit specification works with Spec Kit. A GSD specification works with GSD. A Kiro specification works with Kiro. None of them can exchange specifications. There is no interoperability, no shared format, no ecosystem.

SpecVerse's `.specly` format is tool-independent. It can be parsed by any system that reads YAML. It can be validated by any system that processes JSON Schema. It can be consumed by any AI agent, any code generator, any documentation tool. This is the "Define Once, Implement Anywhere" promise -- and it requires a *language*, not a workflow.

Part 5: What SpecVerse Can Learn

The SDD tools are not competitors, but they are *teachers*. Several patterns are worth adopting:

5.1 From OpenSpec: Change Isolation

OpenSpec's `changes/` directory pattern -- each change gets its own folder with proposal, spec, design, and tasks -- is elegant. It makes specifications reviewable in pull requests and traceable in git history. SpecVerse could adopt a similar pattern for managing specification evolution.

5.2 From Spec Kit: The Constitution Concept

The idea that a project should have governing principles that constrain all AI decisions is powerful. In SpecVerse terms, this could map to **project-level conventions** -- not just field-level conventions (email implies validation) but architectural conventions (all services must be stateless, all APIs must be versioned, all models must have audit trails). A `.specly` constitution file could be machine-validated, not just prompt-injected.

5.3 From GSD: Context Engineering

GSD's insight that AI quality degrades with context accumulation is empirically correct and architecturally significant. If SpecVerse specifications are going to be consumed by AI agents for implementation, the way they are loaded matters. Selective context loading, fresh agent windows, and file-based inter-agent communication are patterns that SpecVerse's tooling should support.

5.4 From GSD: Multi-Agent Specialisation

The pattern of specialised agents -- researchers, planners, executors, verifiers -- maps naturally to SpecVerse's workflow. A "specification agent" could expand `.specly` files. An "implementation agent" could generate code from expanded specs. A "verification agent" could validate that code matches specification. Each agent receives precisely the context it needs.

5.5 From Kiro: Agent Hooks

Kiro's automated triggers that fire on file save/create/delete are relevant to SpecVerse. A hook that re-validates specifications when `.specly` files change, or that regenerates diagrams when models are modified, would make the spec-first workflow seamless.

5.6 From All: The Brownfield Narrative

Every successful SDD tool emphasises brownfield support. SpecVerse should be explicit about how existing codebases can be reverse-engineered into `.specly` specifications (as demonstrated with the JobHunter analysis: Express+Drizzle+React app to 1,396-line spec). The market wants to adopt specification-first practices *without* rewriting their codebase.

Part 6: The Criticisms -- And Why They Don't Apply

The SDD movement has attracted serious criticism. Understanding these critiques is essential because SpecVerse must avoid the same traps.

Criticism 1: "Reinvented Waterfall"

The argument: SDD's phased workflow (requirements -> design -> plan -> implement -> verify) is textbook waterfall with a new name. Heavy upfront documentation before coding echoes Big Design Up Front.

Why it doesn't apply to SpecVerse: The waterfall criticism targets *verbose natural-language documentation phases*. Spec Kit's 2,067 lines of Markdown for one feature is legitimately waterfall-scale documentation. SpecVerse's approach is the opposite: write 90% *less* specification and let the inference engine expand it. A 50-line `.specly` file that expands to a

350-line architecture is not waterfall -- it is compression. The specification is a compact expression of intent, not an exhaustive prose document.

Criticism 2: "Natural Language Specs Lack Formality"

The argument: Markdown specifications "cannot be tested, compiled, or type-checked." Code is the only artefact that can be formally verified.

This is exactly SpecVerse's thesis. The critics are right -- and they are inadvertently arguing for SpecVerse's approach. `.specly` specifications *can* be validated, type-checked, and programmatically expanded. They have JSON Schema validation, typed models, and deterministic inference. When critics say "code is law, not specs," the correct response is: "only because your specs are Markdown."

Criticism 3: "Agents Ignore the Notes"

The argument: Despite elaborate specifications, AI agents still miss instructions, generate incorrect code, and mark verification complete without actually testing.

SpecVerse's response is structural, not procedural. When specifications are machine-parseable, agents don't need to "read and interpret" them -- tooling can extract the exact constraints, generate the exact scaffolding, and validate the exact output. The agent compliance problem is a *natural-language interpretation* problem. Formal specifications bypass it.

Criticism 4: "Specification Quality Is Hard"

The argument: "Making natural language sufficiently precise requires so much text that you lose any benefit of writing the spec in the first place." (Arcturus Labs)

This is the strongest argument for convention processing. The insight is correct: precise natural language is expensive. The solution is not "write better Markdown" but "use a language that encodes precision in conventions." When `email: Email required unique verified` expands to schema, validation, constraints, and endpoints, you achieve precision without verbosity. Convention processors solve the specification quality problem by making precision the *default*, not the burden.

Criticism 5: "Overkill for Small Changes"

The argument: SDD creates too much overhead for simple bug fixes and minor features. "A sledgehammer to crack a nut."

SpecVerse scales naturally. Adding a field to a model is a one-line change to a `.specly` file. The inference engine re-expands. The validation runs. The diagrams regenerate. There is no "ceremony" to skip because the specification language itself is minimal. The overhead in SDD tools comes from their *process* (multiple phases, markdown documents, review cycles), not from the concept of specification-first development.

Part 7: Strategic Implications

The Market Has Validated the Thesis

With 130K+ combined stars across Spec Kit, OpenSpec, and GSD -- plus AWS investing in a dedicated IDE, Tessl raising venture capital, and Thoughtworks certifying SDD as a key practice -- the market has conclusively validated that **specification-first development is the future of AI-assisted engineering**.

The debate is over. The question is no longer "should we write specifications before code?" but "what should specifications look like?"

The Answer Is Not Markdown

Every SDD tool has converged on the same answer: Markdown templates. And every serious evaluation has identified the same problem: Markdown specifications are verbose, ambiguous, unvalidatable, and brittle. The Scott Logic finding -- 10x slower with no quality improvement -- is not a failure of the Spec Kit workflow. It is a failure of Markdown as a specification medium.

SpecVerse Is the Missing Layer

The competitive map reveals an extraordinary positioning opportunity:

WORKFLOW LAYER (saturated, competitive)
Spec Kit OpenSpec GSD Kiro Tessl BMAD
"How to work with AI agents"
LANGUAGE LAYER (sparsely populated)
SpecVerse (.specly) – enterprise systems
Quint (.qnt) – distributed systems / consensus
"What your system looks like, formally"
VERIFICATION LAYER (emerging)
Lean Apalache/Z3 (Quint) Dafny TLA+
"Proof that your system is correct"

SpecVerse does not compete with any SDD tool. It **completes** them. Quint demonstrates what the language layer looks like for distributed systems; SpecVerse provides it for enterprise systems. Neither competes with the other -- they are peer domain languages that could be orchestrated by a meta-specification platform. Every SDD workflow needs a specification format. Currently they all use Markdown. SpecVerse offers a format that is:

- **Minimal** (write 90% less than Markdown specs)
- **Expandable** (4x-7.6x inference expansion)
- **Validatable** (JSON Schema + typed models)
- **Portable** (YAML-based, tool-agnostic)
- **Convention-driven** (community knowledge encoded in processors)

The Integration Play

The strategic move is not to replace SDD tools but to become their specification engine:

- **OpenSpec + SpecVerse:** Replace Markdown specs with `.specly` files in the `changes/` directory. OpenSpec's propose-apply-archive workflow operates on formal, expandable specifications instead of prose.
- **Spec Kit + SpecVerse:** Replace `spec.md` and `data-model.md` with `.specly` files. The constitution concept maps to project-level convention profiles. `/speckit.analyze` gets actual schema validation instead of LLM-based opinion.
- **GSD + SpecVerse:** Replace `REQUIREMENTS.md` with `.specly` specifications. GSD's multi-agent architecture routes expanded specifications to specialised executor agents. Fresh context windows receive precisely the expanded spec they need.
- **Kiro + SpecVerse:** Replace `requirements.md` with `.specly` files. Agent hooks trigger re-expansion on spec changes. The steering system loads convention profiles.

In each case, the SDD tool provides the **workflow** and SpecVerse provides the **language**. The combination is more powerful than either alone.

Part 8: The Specification Hierarchy (Updated)

This analysis extends the hierarchy established in previous analyses (Wolfram, VSDD, Lean):

Level 5: Meta-Specification Platform (SpecVerse)	
Orchestration, cross-domain validation, convention processor registry	
Level 4: Workflow Orchestration (SDD Tools)	← NEW
Spec Kit, OpenSpec, GSD, Kiro, Tessler, BMAD	
Process management, agent coordination, context engineering	
Level 3: Domain Spec Languages	
.specly (enterprise), Quint (distributed systems),	
OpenAPI (APIs), Terraform (infrastructure), dbt (analytics)	
Level 2: Computational Specification	
Wolfram/SymPy/Julia (precise math definitions)	
Level 1: Formal Verification	
Lean (mathematical proofs), Apalache/Z3 (model checking)	
Quint bridges L3 → L1 via Apalache	
Level 0: Implementation	
AI-generated code	

The SDD tools occupy Level 4 -- above domain languages but below the meta-specification platform. They manage the *process* of moving from specifications to implementation. What they lack is a connection to Level 3 -- a formal specification language that they can orchestrate.

Quint demonstrates that the L3-L1 bridge is achievable: its specifications connect directly to the Apache model checker for formal verification. SpecVerse should build an analogous bridge for the enterprise domain -- not necessarily through model checking, but through some form of mechanical verification that `.spec1y` specifications are internally consistent and that implementations conform to them.

Convention processors bridge Level 5 to Level 3 (human intent to domain language). The SDD movement has built the bridge between Level 4 and Level 0 (process to code). The gap between Level 4 and Level 3 -- workflow to formal language -- is where the integration opportunity lives.

Conclusion

The spec-driven development movement is the strongest external validation SpecVerse has received. Not from a single researcher or paper, but from an entire industry simultaneously arriving at the same conclusion: **specification-first development is necessary, and the current tools are insufficient.**

Every SDD workflow tool has identified the right problem. None has identified the right solution. They have built increasingly sophisticated *process layers* on top of the same primitive foundation: natural-language Markdown. The result is predictable: verbose specifications, unvalidatable artifacts, no inference, no expansion, no cross-domain validation, and a 10x slowdown with no quality improvement (Scott Logic).

Meanwhile, Quint has demonstrated -- in production, on systems handling billions of dollars -- that formal specification languages *work*. Their Malachite case study reduced a "couple of months" to one week. Their model checker catches bugs that testing cannot. Their LLM-era workflow treats AI as a translator and deterministic tools as the reasoning layer. Quint proves the approach; it just does it for distributed systems, not enterprise software.

SpecVerse offers what the SDD ecosystem lacks: **a real specification language for the enterprise domain**. Not Markdown with headings. Not prose with RFC 2119 keywords. A typed, schema-validated, inference-powered, convention-driven specification language that expands minimal intent into complete architectures. And unlike Quint, SpecVerse has convention processors that make formal specification accessible -- you don't need to understand temporal logic to write `email: Email required unique verified`.

The SDD tools built the house. Quint proved the foundation works. SpecVerse is pouring that foundation for the domain where 90% of software gets built.

See also: [QUINT-ANALYSIS.md](#) for the full analysis of Quint and its implications for SpecVerse's meta-specification platform direction.

Sources

Primary Sources (Fetched and Analysed)

- [OpenSpec GitHub Repository](#) -- ~28.5K stars, v1.2.0
- [GitHub Spec Kit Repository](#) -- ~75K stars

- [GSD GitHub Repository](#) -- ~26.3K stars
- [GSD Medium Article](#)

Critical Analyses

- [Scott Logic: Putting Spec Kit Through Its Paces -- Radical Idea or Reinvented Waterfall?](#)
- [Martin Fowler: Understanding SDD -- Kiro, Spec-Kit, and Tessel](#)
- [Marmelab: SDD -- The Waterfall Strikes Back](#)
- [Arcturus Labs: Why SDD Breaks at Scale](#)

Industry Analysis

- [Thoughtworks: Spec-Driven Development -- Unpacking 2025's New Engineering Practices](#)
- [RedMonk: Vibe Coding vs. Spec-Driven Development](#)
- [The New Stack: Beyond Vibe Coding](#)
- [InformationWeek: How SDD Is Reshaping Development](#)

Tool-Specific Sources

- [GitHub Blog: Spec-Driven Development with AI](#)
- [Microsoft Developer: Diving Into SDD with Spec Kit](#)
- [Den Delimarsky: What's The Deal With GitHub Spec Kit](#)
- [Kiro Official Site and Pricing](#)
- [Tessel: AI Native Development Platform](#)
- [Codecentric: Anatomy of Claude Code Workflows \(GSD Deep Dive\)](#)
- [GSD Issue #120: Token Usage Explosion](#)
- [GSD 2.0 Cost-Saver Fork](#)

Formal Specification Languages

- [Quint: Reliable Software in the LLM Era](#) -- Gabriela Moreira, November 2025
- [Quint GitHub Repository](#) -- 1.2K stars, v0.31.0
- [Quint Language Documentation](#)
- [QUINT-ANALYSIS.md](#) -- Full SpecVerse analysis of Quint

Community and Comparison

- [SDD Comparison: BMAD vs Spec-Kit vs OpenSpec vs PromptX](#)
 - [spec-compare \(6-tool comparison\)](#)
 - [Rick Hightower: GSD vs Spec Kit vs OpenSpec vs Taskmaster AI](#)
 - [VibeCoding.app: Spec Kit Review 2026](#)
-

Analysis produced March 2026. All star counts, version numbers, and community metrics reflect data as of the analysis date.