ISSN: 2473-4901
v12 n6729

Specify First, Code Later: A Matched Comparison of Spec-Driven and Vibe Coding

Margarida Sacouto
msacouto@student.fairfield.edu
Fairfield University
Fairfield, Connecticut, 06824, USA

Luis Sacouto
luis.sacouto@qu.edu
Quinnipiac University
Hamden, Connecticut, 06518, USA

Guido Lang
guido.lang@qu.edu
Quinnipiac University
Hamden, Connecticut, 06518, USA

Abstract

Spec-Driven Development (SDD) reframes AI-assisted coding so that a human authors a specification, optionally with AI assistance, and a model writes the implementation. Proponents claim the specification step improves accessibility, reduces technical debt, and aids reproducibility, but no controlled experiment has evaluated the benefits of SDD benefits. We run a matched comparison on Python benchmark tasks between an LLM-generated SDD approach and a free-prompting baseline that solves the same task from a single complete instruction. Across matched comparisons, specification-driven code raised the weighted test pass rate by 9.2p.p. Because our setup uses a specification that is purely LLM generated, with no additional domain expertise, and does not benefit from the reuse and review that characterize real engineering, this gain should be interpreted as a lower bound on the value of SDD for AI-assisted coding. The method used roughly 28 times the tokens and 15 times as much wall-clock time of the free-prompting baseline. But this cost can be amortized because the specification, plan, and task list are durable artifacts that can be reviewed, versioned, and regenerated against.

Keywords: spec-driven development, SDD, AI-assisted coding, large language models, code generation

Specify First, Code Later: A Matched Comparison of Spec-Driven and Vibe Coding

Margarida Sacouto, Luis Sacouto, Guido Lang

1. INTRODUCTION

Large language models (LLMs) now write code fluently enough that the bottleneck has shifted from producing code to trusting it. The trend is shifting so that instead of writing the implementation, the developer writes or co-writes a specification, and an agent writes the code. This is the premise of spec-driven development (SDD) in its contemporary, AI-assisted form. Frameworks such as GitHub Spec Kit implement the idea through a process with different stages comprising specify, plan, tasks, implement, and validate. The rapid adoption of these toolkits indicates that companies currently believe that specifying first produces better software.

Unconstrained code generation is known to be unreliable and sometimes insecure (Chen et al., 2021). A specification makes intent explicit before generation begins and the SDD process adds checks along the way that should improve the output (Taghavi and Bhavani, 2026). SDD is intended to mitigate known risks with AI-assisted coding (Poesia et al., 2022) such as lack of reproducibility and trust in output, and to support accessibility, reduced technical debt, and reproducibility, since users are able to specify requirements in natural language and iterate on them before producing a single line of code.

With this paper, we try to assess the level of improvement in coding outcomes when comparing SDD against a fair baseline consisting of a short prompt that gives the model a reasonably complete instruction and nothing else. Using a subset of BigCodeBench coding tasks, we run a matched, task-level comparison between a Spec Kit-style process that generates a feature specification, a technical plan, and a task list before solving, and that single-call prompt baseline that solves the same task directly. Both conditions share the same solver instruction, the same models, and the same tasks, so the only thing that varies is whether the specification artifacts were produced. We measure test pass rate, full-pass status, and the token and time cost of each condition, and we account for cost stage by stage so that the price of the method is visible alongside its benefit. Therefore, we offer a controlled observation about the value of the specification step under a fair comparison. Because this design isolates the specification step from the reuse, review, and iteration that characterize real engineering, any improvement it reveals should be interpreted as a lower bound on the method's value. First, we will provide a review of SDD and its benefits. Then we will describe the experiment, report the results, and discuss this study’s limitations.

2. BACKGROUND AND RELATED WORK

SDD is a software-engineering methodology in which a specification, rather than source code, is treated as the primary artifact that guides construction (Ostroff et al., 2004). It is based on an agile approach combining the executable tests of Test-Driven Development with the preconditions, postconditions, and class invariants of Design-by-Contract (Ostroff et al., 2004). The central argument was that both tests and contracts are kinds of specification, complementary rather than opposed, since tests capture concrete, collaborative behavior through specific scenarios, while contracts state general properties that hold across all inputs, and a developer benefits from being able to move between the two (Ostroff et al., 2004).

Contemporary AI-assisted SDD adapts this idea for a new actor in the loop. Where classical SDD asked a human to translate intent into tests and contracts, recent tooling positions the specification as the artifact a human authors so that an LLM, rather than the human, produces the implementation. The specification remains the intermediate artifact sitting between human intent and the final code, but the consumer of that artifact changes. The unit of work is no longer an isolated prompt but an agent-driven development process supported by persistent artifacts, defined roles, and validation mechanisms. Within this process, specifications become central artifacts that guide planning, task decomposition,

implementation, and review (de Macedo, 2026). This relocation of the specification, from something a programmer reads to something an LLM consumes, is what makes the question of whether specification actually helps newly urgent.

The urgency comes from a well-documented gap between the fluency of LLMs at producing code and the reliability of that code. Liu et al. (2024) systematically studied 4,066 ChatGPT-generated programs across 2,033 Java and Python tasks and found that roughly two-thirds were functionally correct while the remainder produced wrong outputs or failed to compile, and that 1,930 of the generated snippets exhibited code-style or maintainability problems independent of correctness. They also showed that model output could be improved by more than 20% when guided by iterative feedback from static-analysis tools, reframing code generation as a feedback-driven process rather than a single-shot translation. Similarly, Rahman et al. (2025) reported that LLMs achieving 84–89% correctness on established synthetic benchmarks managed only 25–34% on real-world class-level tasks drawn from open-source repositories, while exhibiting negligible performance differences between pre-training cutoff and post-cutoff codebases.

Within this context, SDD can be positioned as a mechanism for making intent explicit before generation begins, giving the model the information it needs while providing humans with a reviewable contract against which outputs can be evaluated. Two strands of evidence suggest a plausible mechanism for this effect. Zi et al. (2025) introduced a mechanism which augments coding benchmarks with prompts ranging from minimal descriptions to highly detailed specifications, and found that performance generally improved with prompt specificity. Their qualitative analysis showed that higher-performing prompts increasingly incorporated explicit input and output specifications, edge-case handling, and stepwise implementation guidance. Rahman et al. (2025) similarly found that supplying concrete implementation patterns through retrieval improved correctness by 4–7 percentage points (p.p.), while comprehensive docstrings alone produced only modest gains. Together, these findings suggest that specifications are valuable to the extent that they make requirements, constraints, and solution structure explicit.

A growing set of tools now implements SDD as a concrete, staged workflow over a coding agent. The landscape is composed of several frameworks, some of which are attracting more attention, GitHub Spec Kit, OpenSpec, BMAD Method, Get Shit Done, Spec Kitty, and Reversa (de Macedo, 2026). GitHub Spec Kit organizes the development cycle into a sequence of steps whose key ones are specify, plan, tasks, and implement. Each step produces an artifact that feeds the next, and it claims compatibility with many coding agents (GitHub, 2026a, 2026b; de Macedo, 2026). GitHub Spec Kit is well known for strengths such as specification and portability, treating the specification as a source of truth rather than a disposable document (de Macedo, 2026).

Toward the more autonomous end, Taghavi and Bhavani (2026) build on Spec Kit with a multi-agent process that adds repository-grounding and validation hooks, identifying a central failure mode they term context blindness, in which intermediate artifacts are internally coherent yet incompatible with the repository itself, resulting in hallucinated APIs, nonexistent file paths, or architectural mismatches. Their context-grounding hooks improved a composite quality score by 0.15 points on a five-point scale while maintaining 99.7–100% repository-level test compatibility and increasing performance in a specific subset benchmark from 56.5% to 58.2% on the first iteration Kuang et al. (2026) propose a REquirement-driven LLM agent (REAgent), which reconstructs a raw issue description into structured, information-rich requirements and iteratively refines them, improving the number of resolved issues by an average of 17.40% over five baselines across three benchmarks and two LLMs.

The organizational impact of AI-assisted coding remains mixed. While Cui et al. (2025), pooling three randomized trials across 4,867 developers, report a 26.08% increase in completed tasks, Becker et al. (2025) find that experienced contributors to mature repositories were 19% slower with AI assistance despite believing they were 20% faster. This divergence highlights that unstructured AI-supported code generation struggles in complex contexts. In this setting, SDD serves as a mechanism to provide explicit structure, bridging the gap between generative capability and organizational standards.

Collectively, these studies suggest that the limitations of AI-assisted software development stem not only from model capability but also from the quality and structure of the information provided to the

model. Evidence from code-generation, repository-level issue resolution, and prompt-specification research consistently indicates that explicit requirements, implementation constraints, contextual information, and iterative feedback improve performance, particularly on complex real-world tasks. At the same time, emerging frameworks such as Spec Kit, REAgent, and other specification-centered approaches increasingly treat requirements as first-class artifacts that guide planning, implementation, validation, and refinement. Taken together, this body of work provides a plausible theoretical and empirical foundation for SDD, because if software quality depends on making intent, constraints, and success criteria explicit, then specifications may serve not only as documentation but as operational artifacts that improve the effectiveness of AI-assisted development.

Additionally, organizations may benefit from SDD in AI-assisted coding in multiple ways. A potential benefit is the reduction of technical debt. The argument is that in its free-prompting form, AI accelerates code production without accelerating alignment, so unconstrained generation produces plausible but fragile code that drifts from intent and accrues debt. By making design decisions and constraints explicit before code is written, SDD is claimed to reveal assumptions early. Lucchetti et al. (2025) found that beginner programmers' success with code-generation systems was primarily predicted by the information content of their prompts rather than their mastery of technical vocabulary. Additionally, security and traceability become more easily enforceable. Because the specification separates the stable what from the flexible how, it is argued to support clarity, auditability, and the ability to regenerate or re-target an implementation from the same source of truth, a property that traces directly back to the original SDD goal of keeping specifications consistent with code (Ostroff et al., 2004). In a banking microservices case study comparing a constitutional workflow against unconstrained AI-assisted development using the same model and developer, Marri (2026) reported a 73% reduction in identified security defects, a 56% reduction in time to first secure build, and a fourfold increase in compliance-documentation coverage. The intervention required the model to operate under an explicit set of security and compliance constraints, preventing categories of vulnerabilities that appeared in unconstrained generations. The study provides a concrete example of how making constraints explicit may improve traceability and prevent classes of defects that AI systems can introduce when optimizing primarily for functional correctness.

However, there are reasons to question several commonly asserted benefits of SDD. At the framework level, de Macedo (2026) identifies specification–implementation drift as a recurring risk across SDD-oriented tooling, warns that community-maintained templates and command kits introduce software-supply-chain risks, and notes the absence of standardized process-level benchmarks capable of validating many of the productivity claims made by framework authors. As a result, much of the current evidence consists of demonstrations and case studies rather than controlled evaluations.

There remains the question of how any claimed benefits of SDD should be evaluated. Such evaluation requires benchmarks that exercise realistic, instruction-rich programming tasks rather than short algorithmic exercises. Zhuo et al. (2025) introduced BigCodeBench, a benchmark comprising 1,140 programming tasks spanning 139 libraries and seven domains, requiring models to combine multiple function calls while following complex instructions. Despite rapid advances in code generation, the strongest evaluated models achieved only around 60% task success, compared with a 97% human baseline. The benchmark also includes paired variants that differ in the amount and structure of instructional detail, making it particularly suitable for studying how specification richness influences coding performance.

Although SDD shows promise, several conditions may keep organizations from adopting SDD even where the method is effective. Kemell et al. (2025), in a case study of seven European software organizations, find that generative AI remains largely a personal assistant rather than an organizational practice, constrained by data privacy, a fast-moving tool market, difficulty measuring impact, and change resistance. SDD requires the transition these firms have not made, since the specification is an organizational artifact that must be authored, reviewed, and owned. The managerial implications should therefore not be limited to whether specification improves correctness, but whether and how an organization can systematically implement SDD before the technical debt it prevents has already accumulated.

Taken together, prior literature suggests: First, unconstrained AI-generated code remains unreliable, insecure, and prone to divergence from user intent. Second, increasing the quality and specificity of requirements can improve outcomes, although the effect appears bounded and dependent on particular informational aspects such as explicit constraints, edge cases, and implementation guidance. Third, workflows that externalize intent into structured artifacts frequently report improvements in quality, traceability, and issue resolution, but these gains are typically accompanied by additional mechanisms such as retrieval, validation, grounding, role specialization, or iterative refinement. Finally, independent process-level evaluation remains scarce, and many proposed benefits of SDD have yet to be isolated from the broader workflows in which they are embedded.

This paper expands on the existing literature by using BigCodeBench's tasks differing in instructional detail, to isolate the actual effects of specification.

3. EXPERIMENTAL SETUP

We evaluated the performance of free-prompting versus specification-driven LLM generated code on Python programming tasks from BigCodeBench. The experiment used the BigCodeBench-Hard split for hard tasks and constructed a smaller normal-difficulty set by loading the full BigCodeBench split and removing task identifiers that appear in BigCodeBench-Hard. All tasks were drawn from the v0.1.0_hf split. For each selected task, the evaluation harness retrieved the task identifier, difficulty level, natural-language instruction prompt, code template, structured documentation fields, and benchmark test code.

3.1 Conditions

The study compared two generation conditions, the first of which, the 'free-prompt' (instruct) baseline, presented the programmer model with the benchmark-provided instruction prompt directly so that it returned a single complete Python solution. In the specification-driven condition, a separate LLM-driven specification generation process first converted the task description into intermediate development artifacts, following a workflow modeled on GitHub Spec Kit, independent of the person behind the validation. The programmer model then solved the task from those artifacts and the original code template. Both conditions used the same solver system instruction requiring the model to return only a complete Python solution as a single code block, with no explanation.

3.2 Models

The experiment included Anthropic and OpenAI models comprising Haiku, Sonnet, GPT-5.4, and GPT-5.4-mini. Model calls were executed with temperature set to 0.0 due to deterministic nature of coding tasks, and a maximum generation budget of 4096 output tokens. In the specification-driven condition, the specification model was recorded separately from the programmer model so that the analysis could distinguish the cost of producing artifacts from the cost of generating the final code.

3.3 Specification-Generation Pipeline

Our process is based on the GitHub Spec Kit workflow, following the same structure through three artifact-generation stages comprising feature specification, technical plan, and task list. The content of each artifact is based on Spec Kit's published templates. We selected Spec Kit for its production of high-quality, reusable specifications that serve as persistent project artifacts (de Macedo, 2026). Additionally, grounding the artifact definitions in an externally defined and publicly documented framework reduces researcher discretion over the content and structure of the generated specifications.

For the specification-driven condition, the process formatted each task's structured documentation into a compact function specification containing the available description, notes, parameters, returns, requirements, raised errors, and examples. This formatted description was passed through three artifact-generation stages before solution generation.

1. **Feature specification:** The model produces a Spec Kit-style feature specification covering user scenarios, functional requirements, interface contract, success criteria, and assumptions, while preserving explicit constraints and avoiding code.
2. **Technical plan:** The model produces an implementation plan from the original task description and generated feature specification, including technical context, implementation approach, data flow, error handling, validation strategy, and remaining assumptions.

3. **Task list:** The model produces ordered, checkable implementation tasks with setup, core implementation, validation, dependencies, and acceptance mapping.

The final solver prompt supplies the generated feature specification, technical plan, task list, and benchmark code template, and requires the programmer model's output to begin from there.

3.4 Matched-Comparison Design

Comparisons were matched at the task level. Each comparison unit was defined by the same task ID, difficulty level, and programmer model across the two conditions. This design controls for variation in task content, benchmark difficulty label, and code generator while isolating the prompt-condition difference between direct instruction and specification-driven artifact generation.

3.5 Evaluation Metrics

The primary metric was test pass rate, computed from the benchmark tests as the number of passed pytest cases divided by the total number of passed and failed cases. Full-pass status and coverage were treated as secondary metrics, where full-pass status indicates whether all tests for a task passed, and coverage summarizes the share of planned matched comparison conditions that produced evaluable test-count records. The weighted test pass rate aggregates the per-task test pass rate across tasks by pooling the underlying test cases, dividing the total number of passed cases across all tasks by the total number of passed and failed cases, so that each task contributes in proportion to its number of tests.

Commented [A1]: I inferred that this is how you calculated weighted test pass rate, but I might be wrong.

3.6 Token and Time Accounting

We use token count as a proxy for monetary cost, since provider pricing is billed per input and output token, and wall-clock seconds as a proxy for latency. Reporting tokens rather than dollars keeps the comparison stable across providers and pricing changes. For each model call, we recorded input tokens, output tokens, and wall-clock seconds. In the instruct condition, these quantities came from the single solver call. In the specification-driven condition, accounting summed the specification, plan, task-list, and solve stages, while also preserving per-step usage for later audit. The same fields were written for each task-level record, comprising run identifier, dataset index, task_id, level, condition, programmer model, specification model, test counts, token counts, wall time, generated code, and the raw pytest transcript.

3.7 Exclusions

We exclude tasks 417 and 418 from the analysis. Both are HARD-difficulty tasks where the evaluation harness returned zero parsed tests, for both Claude Haiku and Claude Sonnet, in both the INSTRUCT and SPEC-DRIVEN arms. This gives 4 excluded task-model pairs, 8 rows total. These were excluded before matched comparisons because the test harness reported total = 0, so they do not provide a valid denominator for test pass rate. A total of zero can arise when pytest does not yield a parsed passed or failed count, including timeout or collection/execution failures that prevent the summary parser from recovering a test-count denominator. Excluding these records keeps the primary metric defined consistently across conditions.

4. RESULTS

4.1 Test Pass Rate

This purely LLM generated SDD version improved outcomes across every condition. Across 296 matched comparisons, spec-driven prompting raised the weighted test pass rate from 78.2% to 87.4%, a gain of 9.2% (95% bootstrap confidence interval (CI) from +5.5% to +13.0%), and exceeded the baseline in every condition, which is shown in Figure 1.

ISSN: 2473-4901
v12 n6729

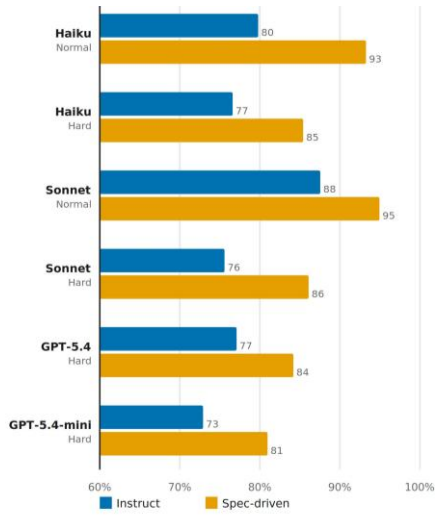

Figure 1. Weighted test pass rate by model and difficulty.

The aggregate improvement was not driven by a small number of high-test-count tasks; the mean paired task-level improvement was 8.0%, and a sign test over non-tied paired deltas confirmed the significance of the effect ($p = 0.0059$). The performance gain was more pronounced in normal-difficulty tasks (+10.4%) compared to hard tasks (+8.6%). At the model level, the largest improvement occurred in the Claude Haiku / normal condition, where the pass rate rose by 13.5%. Overall, spec-driven prompting performed better in 80 matched tasks, worse in 48, and tied in 168 (see Table 1).

Scope	Pairs	Extra tests	Weighted Δ	Mean task Δ	Better	Worse	Ties	Full-pass Δ	Tokens Multiple	Time Multiple
All matched conditions	296	166	+9.2%	+8.0%	80	48	168	+5.7%	28.0x	14.9x
Normal tasks	100	62	+10.4%	+10.4%	23	11	66	+7.0%	35.3x	16.3x
Hard tasks	196	104	+8.6%	+6.8%	57	37	102	+5.1%	25.0x	13.9x
Haiku / NORMAL	50	40	+13.5%	+13.3%	13	6	31	+6.0%	30.0x	18.9x
Haiku / HARD	48	26	+8.8%	+7.9%	17	7	24	+14.6%	25.0x	15.9x
Sonnet / NORMAL	50	22	+7.4%	+7.5%	10	5	35	+8.0%	45.1x	13.6x
Sonnet / HARD	48	31	+10.5%	+8.6%	14	9	25	+14.6%	28.1x	17.9x
GPT-5.4 / HARD	50	22	+7.1%	+4.9%	14	10	26	+0.0%	27.7x	13.6x
GPT-5.4-mini / HARD	50	25	+8.1%	+5.7%	12	11	27	-8.0%	21.3x	8.9x

Table 1. Matched within-task comparison (spec-driven minus instruct).

The distribution of these improvements is shown in Figure 2.

ISSN: 2473-4901
v12 n6729

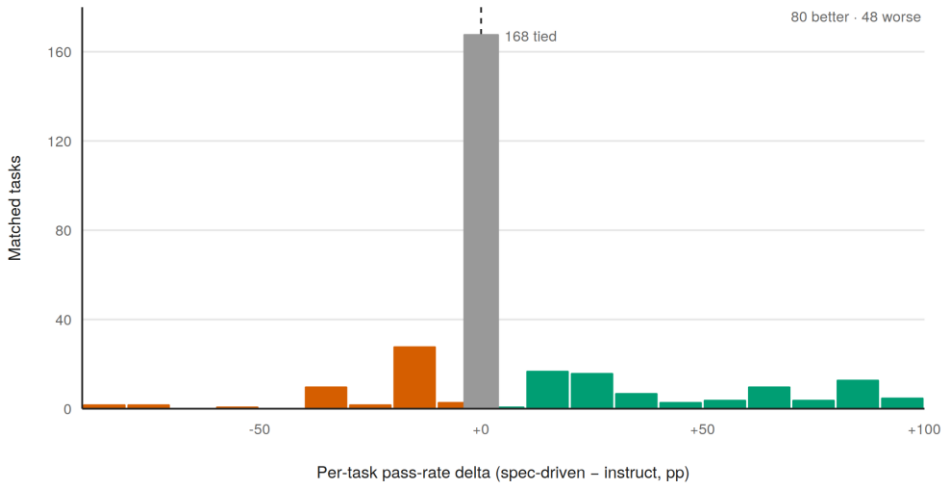

Figure 2. Distribution of matched per-task pass-rate deltas.

The failure-mode analysis in Figure 3 further clarifies where the specification had the most impact:

- Type errors decreased significantly from 15 to 5.
- Full-pass outcomes (tasks where all tests passed) rose from 156 to 173.
- Assertion failures remained unchanged at 91 instances in both conditions.

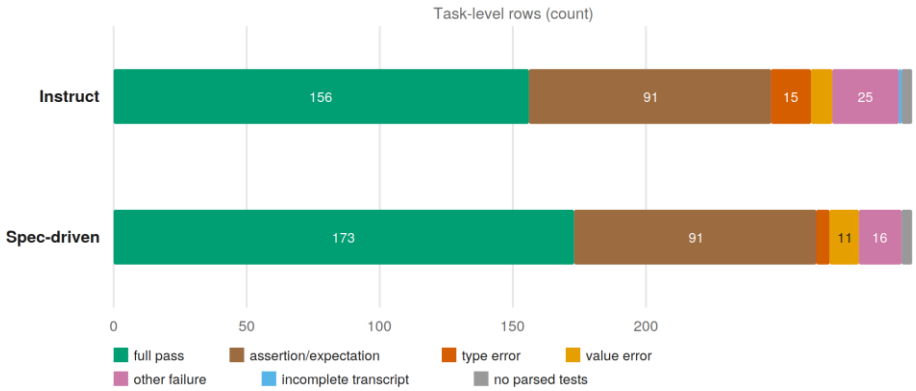

Figure 3. Outcome modes by condition

The results also indicate where the specification contributes most, a question relevant to organizations deciding how far to delegate. While the weighted test pass rate improved by 9.2%, the full-pass rate increased by only 5.7%. This suggests that while an automated specification helps the model better organize the information already present in the task documentation, it cannot provide the missing intent or hidden test criteria required for a perfect score. This is particularly evident in the GPT-5.4-mini results

ISSN: 2473-4901
v12 n6729

(Table 2), where the model achieved an 8.1% gain in weighted pass rate but saw an 8.0% decline in full-pass status, suggesting that more complex specifications overwhelm smaller models.

Model	Level	Condition	N	Tests	Test pass	Mean task	Full-pass	Mean tokens	Mean time (sec.)
Haiku	NORMAL	Instruct	50	297	79.8%	80.7%	70.0%	559	2.4
Haiku	NORMAL	Spec-driven	50	297	93.3%	93.9%	76.0%	16,734	43.8
Haiku	HARD	Instruct	48	295	76.6%	77.4%	39.6%	802	3.5
Haiku	HARD	Spec-driven	48	295	85.4%	85.4%	54.2%	19,245	53.6
Sonnet	NORMAL	Instruct	50	297	87.5%	88.5%	76.0%	408	3.6
Sonnet	NORMAL	Spec-driven	50	297	94.9%	96.0%	84.0%	16,313	44.2
Sonnet	HARD	Instruct	48	295	75.6%	77.2%	43.8%	619	5.1
Sonnet	HARD	Spec-driven	48	295	86.1%	85.8%	58.3%	16,629	88.5
GPT-5.4	HARD	Instruct	50	310	77.1%	78.1%	42.0%	518	4.1
GPT-5.4	HARD	Spec-driven	50	310	84.2%	83.1%	42.0%	13,316	48.0
GPT-5.4-mini	HARD	Instruct	50	310	72.9%	73.9%	44.0%	503	2.3
GPT-5.4-mini	HARD	Spec-driven	50	310	81.0%	79.6%	36.0%	10,100	19.3

Table 2. Per-condition outcomes by model, difficulty, and prompting condition.

Marri (2026) observed a parallel effect, in which supplying the full constitution dropped compliance to 78% while a trimmed set raised it to 96%. More specification is not monotonically better, and a smaller model can be overwhelmed by it. The deployment implication would be to pair richer specifications with models capable of consuming them, and tailor the artifact to the request.

Additionally, our tasks were single-function Python problems, where a specification has little structure to capture. Repository-level and multi-file work is where the context-blindness failures that Taghavi and Bhavani (2026) describe grow larger, and where a specification has the most structure to impose. The leverage may be largest where this benchmark is smallest, which is where most organizational software resides.

4.2 Token Usage

The gain carried substantial overhead, with the median task using 28.0 times as many total tokens and 14.9 times as much wall-clock time under the spec-driven condition.

ISSN: 2473-4901
v12 n6729

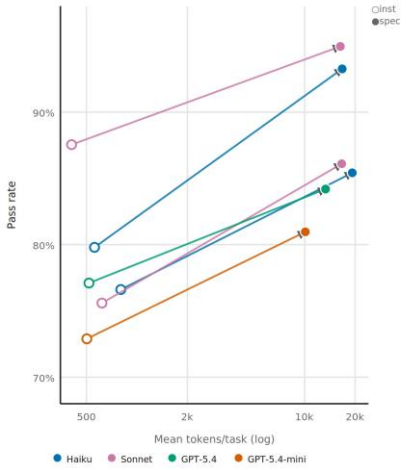

Figure 5. Accuracy versus cost.

However, an organization can amortize such costs because the expensive artifacts are produced once and are themselves reusable, reviewable assets that can be approved, versioned, regenerated against a new target, or handed to a different agent. The trade is difficult to justify only when each generation is one-shot and discarded. It becomes more defensible wherever each correct test is highly valued and the artifacts persist, which is the normal condition of production engineering rather than benchmark solving.

Model	Stage	Mean tokens	Mean time (sec.)	Token share	Time share
Haiku	Feature spec	2,242	11.7	12.5%	24.1%
Haiku	Technical plan	4,134	18.1	23.0%	37.3%
Haiku	Task list	5,701	14.2	31.7%	29.2%
Haiku	Solve	5,886	4.5	32.8%	9.3%
Sonnet	Feature spec	2,202	18.0	13.4%	27.3%
Sonnet	Technical plan	3,839	24.4	23.3%	37.0%
Sonnet	Task list	5,264	18.3	32.0%	27.8%
Sonnet	Solve	5,162	5.2	31.3%	7.9%
GPT-5.4	Feature spec	2,174	17.5	16.3%	36.4%
GPT-5.4	Technical plan	3,293	17.5	24.7%	36.4%
GPT-5.4	Task list	3,971	9.5	29.8%	19.8%
GPT-5.4	Solve	3,878	3.6	29.1%	7.5%
GPT-5.4-mini	Feature spec	1,779	6.8	17.6%	35.2%
GPT-5.4-mini	Technical plan	2,501	6.3	24.8%	32.9%
GPT-5.4-mini	Task list	2,957	3.8	29.3%	19.4%
GPT-5.4-mini	Solve	2,862	2.4	28.3%	12.5%

Table 3. Spec-driven cost composition by pipeline stage.

ISSN: 2473-4901
v12 n6729

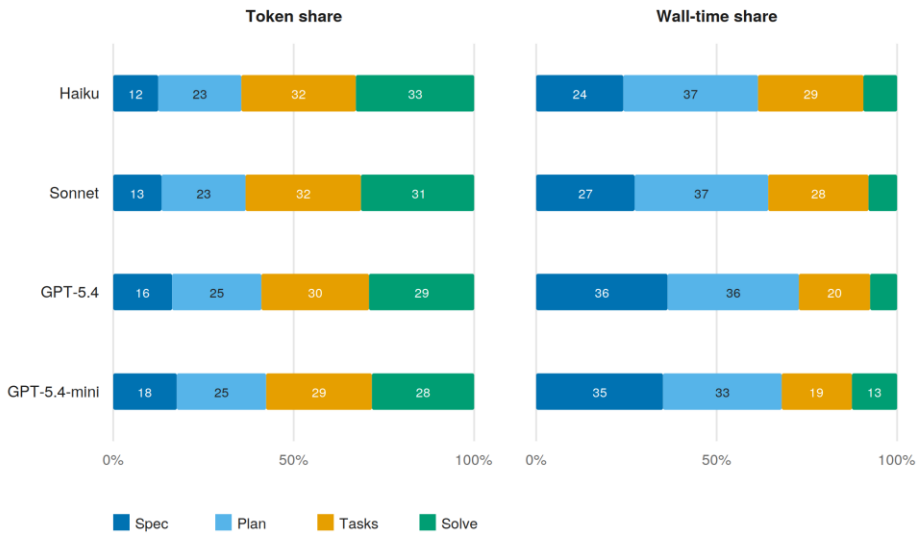

Figure 6. Per-stage token and wall-time share of the spec-driven pipeline across models.

4.3 Overall assessment

These findings establish a statistically significant "floor" for the value of SDD in AI-assisted coding. Even without human intervention or specialized infrastructure, the transition from a single prompt to a structured specification process yielded a 9.2% gain in functional correctness.

This aligns with prior research by Zi et al. (2025) and Rahman et al. (2025), which found that performance gains in LLM coding are driven by explicit input/output contracts and structural decomposition rather than simple verbosity. Our results confirm that making these elements explicit through a staged pipeline provides a measurable advantage, particularly as task complexity increases.

5. LIMITATIONS

Establishing a fair floor for SDD that is relevant for modern organizations requires frontier models. Running them across every model, difficulty, and condition combination is expensive. That cost set the boundaries of what we could test, in three ways. First, on model coverage, GPT-5.4 and GPT-5.4-mini were evaluated only on the hard split, while Haiku and Sonnet were evaluated on both normal and hard tasks, so the normal-task results rest on two models rather than four. Second, on task and SDD framework scope, we ran one benchmark, Python only, single-function tasks, and one Spec Kit-style pipeline, so the results do not extend to repository-level or multi-file work, where the context-blindness failures that Taghavi and Bhavani (2026) describe would likely be larger and where a specification has more structure to capture. Third, on the question itself, a complete evaluation of SDD would require human-authored specifications, but a human in the loop is exactly the expensive component a fixed budget cannot scale. We therefore measured the floor an automated specification reaches, not the gain a human-authored contract would deliver, which is the most likely ceiling on the full-pass gain and the key difference from the human-in-the-loop SDD that proponents describe. The metrics do not cover everything SDD promises, since test pass rate over hidden benchmark tests captures functional correctness only. It does not measure the maintainability, security, or technical-debt reductions that motivate SDD in the first place. Two tasks were crossed out because the test harness reported a total of zero parsed tests for them in the Haiku and Sonnet hard conditions, leaving no valid denominator for test pass rate.

6. CONCLUSION

An automated specification, generated by a model from the benchmark's own documentation, improved performance on these coding tasks. This effect represents a lower bound on what SDD can offer, and within our setting it was positive and consistent. That finding reframes the practical question facing organizations, since the debate is not whether specification helps, but how far above this floor a given deployment can reach. The gap between partial and full success shows that the remaining value lies in information a self-generated specification cannot supply, comprising the intent, constraints, and success criteria that a human author contributes and that never appeared in the task documentation. The cost analysis shows that the price of the method, though large, produces reusable, reviewable artifacts that can be amortized across the engineering organization. And the model and task-scope findings suggest that leverage grows with the capability of the consuming model and the structural complexity of the work, conditions that hold in real codebases and are largely absent in the constrained setting of single-function benchmark tasks.

The value of SDD is lowest when used for short, self-specified, single-function problems solved once and discarded. It rises along every dimension that distinguishes production software from a benchmark, including human-authored intent, capable models, persistent artifacts, and multi-file systems. Therefore, we established a conservative floor for the added value of SDD. An organization adopting SDD would be relying not on the specific gain observed in this work, but on a method whose returns may scale with the conditions of its own work.

For practitioners, our findings suggest treating the specification as a durable asset rather than a transient prompt, to pair richer specifications with models capable of consuming them, and to apply the method where intent is hard to infer and code is meant to persist. For research, future work may wish to measure the method where SDD's leverage is greatest, with human authors supplying the missing intent, on repository-scale tasks, and against the maintainability, security, and traceability outcomes that motivate SDD but that functional test passage cannot capture. The specification as contract offers one interface for delegating parts of the coding process, though its broader value remains to be established.

7. REFERENCES

Becker, J., Rush, N., Barnes, B., & Rein, D. (2025). Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. *arXiv:2507.09089 (METR).*

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

de Macedo, S. O. (2026). *From prompt to process: A process taxonomy and comparative assessment of frameworks supporting AI software development agents* [Preprint]. arXiv. https://arxiv.org/abs/2606.04967

GitHub. (2026a). *github/spec-kit: Toolkit to help you get started with spec-driven development* [Computer software]. GitHub. https://github.com/github/spec-kit

GitHub. (2026b). *Spec Kit: A specification-driven development toolkit*. https://github.github.io/spec-kit/

Kemell, K.-K., Saarikallio, M., Nguyen-Duc, A., & Abrahamsson, P. (2025). Still just personal assistants? A multiple case study of generative AI adoption in software organizations. *Information and Software Technology*, 186, 107805. DOI: 10.1016/j.infsof.2025.107805

Kuang, S., Tian, Z., Lin, K., Tao, C., Wang, S., Bai, H., Shang, L., & Chen, J. (2026). *REAgent: Requirement-driven LLM agents for software issue resolution* [Preprint]. arXiv. https://arxiv.org/abs/2604.06861

Liu, Y., Le-Cong, T., Widyasari, R., Tantithamthavorn, C., Li, L., Le, X.-B. D., & Lo, D. (2024). Refining ChatGPT-generated code: Characterizing and mitigating code quality issues. *ACM Transactions on Software Engineering and Methodology, 33*(5), Article 116. https://doi.org/10.1145/3643674

Lucchetti, F., Wu, Z., Guha, A., Feldman, M. Q., & Anderson, C. J. (2025, April). Substance beats style: Why beginning students fail to code with LLMs. In Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers) (pp. 8541-8610).

Marri, S. R. (2026). *Constitutional spec-driven development: Enforcing security by construction in AI-assisted code generation* [Preprint]. arXiv. https://arxiv.org/abs/2602.02584

Ostroff, J. S., Makalsky, D., & Paige, R. F. (2004). Agile specification-driven development. In J. Eckstein & H. Baumeister (Eds.), *Extreme programming and agile processes in software engineering (XP 2004)* (Lecture Notes in Computer Science, Vol. 3092, pp. 104–112). Springer. https://doi.org/10.1007/978-3-540-24853-8_12

Poesia, G., Polozov, A., Le, V., Tiwari, A., Soares, G., Meek, C., & Gulwani, S. (2022). Synchromesh: Reliable code generation from pre-trained language models [Paper presentation]. International Conference on Learning Representations (ICLR 2022).

Rahman, M., Khatoonabadi, S., & Shihab, E. (2025). *Beyond synthetic benchmarks: Evaluating LLM performance on real-world class-level code generation* [Preprint]. arXiv. https://arxiv.org/abs/2510.26130

Taghavi, P., & Bhavani, S. (2026). *Spec Kit Agents: Context-grounded agentic workflows* [Preprint]. arXiv. https://arxiv.org/abs/2604.05278

Zhuo, T. Y., Vu, M. C., Chim, J., Hu, H., Yu, W., Widyasari, R., Yusuf, I. N. B., Zhan, H., He, J., Paul, I., Brunner, S., Gong, C., Hoang, T., Zebaze, A., Hong, X., Li, W.-D., Kaddour, J., Xu, M., Zhang, Z., . . . von Werra, L. (2025). BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Zi, Y., Menon, H., & Guha, A. (2025). More than a score: Probing the impact of prompt specificity on LLM code generation. In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics* (pp. 2380–2402). The Asian Federation of Natural Language Processing and the Association for Computational Linguistics

ISSN: 2473-4901
v12 n6729