



NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

**Agentic Project Management: A Spec-Driven Multi-Agent Framework
for AI-Assisted Software Engineering**

Konstantinos Chaidemenos

Supervisor: Thanassis Avgerinos, Assistant Professor

ATHENS

OCTOBER 2026



ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ**

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

**Agentic Project Management: Ένα Πολυπρακτορικό Πλαίσιο
Οδηγούμενο από Προδιαγραφές για Ανάπτυξη Λογισμικού με
Υποστήριξη Τεχνητής Νοημοσύνης**

Κωνσταντίνος Χαϊδεμένος

Επιβλέπων: Θανάσης Αυγερινός, Επίκουρος Καθηγητής

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2026

BSc THESIS

Agentic Project Management: A Spec-Driven Multi-Agent Framework for AI-Assisted Software Engineering

Konstantinos Chaidemenos

S.N.: 1115202200262

SUPERVISOR: **Thanassis Avgerinos**, Assistant Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Agentic Project Management: Ένα Πολυπρακτορικό Πλαίσιο Οδηγούμενο από Προδιαγραφές για Ανάπτυξη
Λογισμικού με Υποστήριξη Τεχνητής Νοημοσύνης

Κωνσταντίνος Χαϊδεμένος

A.M.: 1115202200262

ΕΠΙΒΛΕΠΩΝ: Θανάσης Αυγερινός, Επίκουρος Καθηγητής

ABSTRACT

Coding agents now carry out substantial software tasks with growing autonomy. As a task grows, the agent must hold the project's intent while accumulating context competes for the model's finite attention. Recent solutions address this by recording intent in persistent documents and by dividing work among multiple agents.

We present Agentic Project Management (APM), an open-source framework that operates on top of interactive coding agents. APM combines persistent planning documents with role-specific division of work, and makes its coordination user-mediated: its agents communicate through the developer, allowing them to review the work behind each message and request corrections before passing it on. How the work is divided and assigned sets how often such opportunities arise. The design aims to keep authority and decision-making over development with the user.

To evaluate it, we propose a reproducible methodology for comparing frameworks on top of coding agents and similar human-in-the-loop workflows. We implement it as a within-participant case study of APM against GitHub's Spec-kit, where twelve developers each completed the same demanding task across two sessions, once with each framework, on Claude Code with Claude Opus 4.7 or Claude Sonnet 4.6. The comparison covers economic cost, each session's temporal and interaction shape, functional test scores, and developer perception. APM cost more in nine of the twelve pairs, adding a median of \$6.10 across the cohort, with the higher cost concentrated under Opus, and took longer in all twelve pairs, adding a median of 36 minutes to session duration. The frameworks carried no firm difference in functional outcome across the cohort, although APM showed a more favourable pattern under Sonnet. Even so, eleven of the twelve participants rated APM higher on effectiveness and nine on confidence. Our study's comparison documents the added time and cost of APM's user-mediated coordination alongside its opportunities for developer control and review.

SUBJECT AREA: Software Engineering

KEYWORDS: agentic project management, coding agents, multi-agent systems, spec-driven development, AI-assisted software engineering, case study

ΠΕΡΙΛΗΨΗ

Οι πράκτορες προγραμματισμού αναλαμβάνουν πλέον σύνθετες εργασίες λογισμικού με περισσότερη αυτονομία. Όσο μια εργασία μεγαλώνει, ο πράκτορας καλείται να διατηρήσει την πρόθεση του έργου ενώ η συσσώρευση πληροφοριών ανταγωνίζεται για την περιορισμένη προσοχή του μοντέλου. Πρόσφατες λύσεις αντιμετωπίζουν αυτό το ζήτημα καταγράφοντας την πρόθεση σε μόνιμα έγγραφα και μοιράζοντας την εργασία σε πολλαπλούς πράκτορες.

Παρουσιάζουμε το Agentic Project Management (APM), ένα πλαίσιο ανοικτού κώδικα που λειτουργεί πάνω από διαδραστικούς πράκτορες προγραμματισμού. Το APM συνδυάζει μόνιμα έγγραφα σχεδιασμού με κατανόηση εργασίας σε διακριτούς ρόλους, και καθιστά τον συντονισμό διαμεσολαβούμενο από τον χρήστη: οι πράκτορές του επικοινωνούν μέσω του προγραμματιστή, επιτρέποντάς του να εξετάζει την εργασία πίσω από κάθε μήνυμα και να ζητά διορθώσεις πριν το προωθήσει. Ο τρόπος που η εργασία διαιρείται και ανατίθεται καθορίζει πόσο συχνά προκύπτουν τέτοιες ευκαιρίες. Στόχος του σχεδιασμού είναι να διατηρηθεί ο έλεγχος και η λήψη αποφάσεων στα χέρια του χρήστη.

Για να το αξιολογήσουμε, προτείνουμε μια αναπαραγώγιμη μεθοδολογία για τη σύγκριση πλαισίων που λειτουργούν πάνω από πράκτορες προγραμματισμού, καθώς και παρόμοιων ροών εργασίας που έχουν ενεργό συμμετοχή του χρήστη. Την υλοποιούμε σε μελέτη περίπτωσης με σχεδιασμό εντός συμμετεχόντων, συγκρίνοντας το APM με το Spec-kit της GitHub. Δώδεκα προγραμματιστές ολοκλήρωσαν την ίδια απαιτητική εργασία σε δύο συνεδρίες, μία με κάθε εργαλείο, στο Claude Code με τα μοντέλα Claude Opus 4.7 ή Claude Sonnet 4.6. Η σύγκριση καλύπτει το οικονομικό κόστος, τη χρονική εξέλιξη και το μοτίβο αλληλεπίδρασης κάθε συνεδρίας, τη βαθμολογία από λειτουργικούς ελέγχους και την αντίληψη των συμμετεχόντων. Το APM κόστισε περισσότερο σε εννέα από τα δώδεκα ζεύγη, με διάμεση επιπλέον δαπάνη 6,10 δολαρίων στο σύνολο του δείγματος και το υψηλότερο κόστος να συγκεντρώνεται στο Opus, και διήρκεσε περισσότερο και στα δώδεκα ζεύγη, με διάμεση επιπλέον διάρκεια 36 λεπτών ανά συνεδρία. Στην λειτουργική βαθμολόγηση δεν προέκυψε σαφής διαφορά στο σύνολο του δείγματος, αν και το APM παρουσίασε ευνοϊκότερη εικόνα με το Sonnet. Παρ' όλα αυτά, έντεκα από τους δώδεκα συμμετέχοντες βαθμολόγησαν το APM υψηλότερα ως προς την αποτελεσματικότητα και εννέα ως προς την αυτοπεποίθησή τους. Η σύγκριση της μελέτης μας καταγράφει τον πρόσθετο χρόνο και το κόστος του διαμεσολαβούμενου από τον χρήστη συντονισμού του APM, μαζί με τις ευκαιρίες που προσφέρει στον προγραμματιστή για έλεγχο και αναθεώρηση της εργασίας.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Τεχνολογία Λογισμικού

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: agentic project management, πράκτορες προγραμματισμού, πολυπρακτορικά συστήματα, ανάπτυξη οδηγούμενη από προδιαγραφές, τεχνητή νοημοσύνη στην ανάπτυξη λογισμικού, μελέτη περίπτωσης

ACKNOWLEDGEMENTS

To my parents, my loved ones, and all the people who stood by me during this challenging effort.

I also thank my supervisor for the guidance that shaped this work, and the twelve participants who gave their time to the study.

CONTENTS

1. INTRODUCTION	14
2. BACKGROUND	15
2.1 AI Assistants and Agents	15
2.2 Context Windows and Context Decay	16
2.3 Spec-Driven Development	16
2.4 Multi-Agent Systems	17
2.5 Frameworks on Top of a Coding Agent	18
2.6 Agentic Project Management and the Research Gap	18
3. APM DESIGN	20
3.1 Design Goals	20
3.2 Workflow Overview	20
3.3 Agent Roles	22
3.4 Agent Coordination	22
3.5 State Management	23
3.6 Project Management	25
3.6.1 Planning Phase	25
3.6.2 Implementation Phase	26
3.7 Distribution and Customisation	28
4. EVALUATION METHODOLOGY	29
4.1 Evaluation Scope and Research Questions	29
4.2 Spec-kit, the Comparison Framework	29
4.3 Case-Study Conditions	30
4.4 Case-Study Methodology	30
4.4.1 Task Specification	30
4.4.2 Task Environment	31
4.4.3 Sessions and Data	31
4.5 Metrics	31
4.5.1 Economic	32
4.5.2 Process	32
4.5.3 Functional	33
4.5.4 Perception	33

4.6	Analysis Plan	33
5.	EVALUATION RESULTS	35
5.1	Economic	35
5.2	Process	39
5.2.1	Temporal Shape	39
5.2.2	Interaction Shape	41
5.3	Functional	44
5.4	Perception	47
6.	DISCUSSION	49
6.1	Two Realisations of Spec-Driven Planning	49
6.2	Coordinating the Implementation	50
6.2.1	The Cost of Several Agents	50
6.2.2	The Demands on the User	52
6.3	Behind the Functional Score	53
6.3.1	Grading with a Black Box	53
6.3.2	Cost versus Score	55
6.4	Perceived Versus Measured Behaviour	56
6.5	APM's Value Across the Findings	56
6.6	Learning Effects	57
6.7	Threats to Validity	58
6.7.1	Construct Validity	58
6.7.2	Internal Validity	59
6.7.3	External Validity	60
6.7.4	Reliability	61
7.	CONCLUSION	62
7.1	Future Work	62
	ABBREVIATIONS - ACRONYMS	64
	APPENDICES	64
A.	EVALUATION APPENDIX	65
A.1	Spec-kit Release Audit	65
A.2	Software and Data Availability	65
A.3	Questionnaire Instrument	65
A.4	Rate Card	66
A.5	Scoring Rubric	66

B. RESULTS APPENDIX	68
B.1 Economic Cost Breakdowns	68
B.2 Process Breakdowns	68
B.3 Functional Breakdowns	72
B.4 Perception Breakdowns	73

LIST OF FIGURES

3.1	Planning in an APM project.	20
3.2	Implementation in an APM project.	21
3.3	Message Bus exchange between the Manager and a Worker.	23
3.4	Compression of execution and coordination detail into persistent project history.	24
3.5	Handoff between outgoing and incoming agent conversations.	24
3.6	Progression of a project through its Stages, illustrated with modular service tasks in Stage 1 and compressed representations of the Stages that follow.	26
3.7	One Task through the implementation cycle. The Task Prompt and the Task Report travel through the Message Bus, while the Task Log is written to Memory and reviewed from there.	27
5.1	Per-participant session cost within each model, Opus on the left and Sonnet on the right. Each participant has a filled APM diamond and an open Spec-kit diamond joined by a line, and each panel carries its own scale. Each framework's cheapest and most expensive session carries its value.	36
5.2	Per-participant paired cost difference by scope, APM minus Spec-kit, in US dollars. The bar gives the session difference and the markers the planning and implementation differences within it. The reference line marks no difference. P004 and P006 have the phase-attribution caveats of notes [a] and [b] of Table 5.2.	37
5.3	Per-participant session temporal shape. Each participant has two bars, APM and Spec-kit; the filled lower segment is the turn time, the open upper segment the inter-turn time, and the full height the session duration.	41
5.4	Per-session turn-prompt bucket shares by phase. A dashed outline marks a session that carried no turn prompt in the phase and so has no shares.	43
5.5	Per-participant total score within each model, Opus on the left and Sonnet on the right. Each participant has a filled APM diamond and an open Spec-kit diamond joined by a line.	46
6.1	Peaks of context windows per session, in thousands of tokens, across Opus model group participants.	51
6.2	Peaks of context windows per session, in thousands of tokens, across Sonnet model group participants. Reference lines mark the 171k observed automatic-compaction bound (dotted) and the two-hundred-thousand-token context window limit (dashed).	52
6.3	Each Sonnet session's total score on the grader's standard JSON input, filled, and its diagnostic score on the compact JSON input, open, joined where they differ. The Opus sessions do not differ and are omitted.	54
6.4	Per-participant paired cost difference and total-score difference, APM minus Spec-kit, with Opus filled and Sonnet open. The reference lines mark no difference on each axis.	55

LIST OF TABLES

2.1	Selected systems and how they handle long tasks and coordinate work.	19
3.1	APM roles by phase, concurrent conversations, and context scope.	22
3.2	APM commands by addressed agent and function.	25
5.1	Per-framework session and phase cost, in US dollars, across each framework's six sessions in each model.	35
5.2	Paired cost differences by scope, APM minus Spec-kit, in US dollars.	37
5.3	Cost by agent role within the APM sessions, by model, in US dollars.	38
5.4	Per-framework cost by token category, in US dollars, across each framework's six sessions in each model. The negligible uncached-input and five-minute cache-write categories are omitted.	38
5.5	Paired cost differences by token category, APM minus Spec-kit, in US dollars.	39
5.6	Per-framework session and phase durations, in seconds, across each framework's twelve sessions.	40
5.7	Paired time differences by scope across the cohort, APM minus Spec-kit, in seconds. The turn time is the active span, and the inter-turn time is the residual to the duration.	40
5.8	Per-framework turn-prompt counts at session and phase scope, across each framework's twelve sessions.	42
5.9	Paired turn-prompt count differences, APM minus Spec-kit.	42
5.10	Turn-prompt bucket shares and mean character counts by phase and framework.	43
5.11	Per-role turn-prompt counts across the twelve APM sessions. Descriptive and APM only, since Spec-kit has no agent roles to separate.	44
5.12	Per-framework total score, in points, for the cohort and within each model group.	45
5.13	Paired total-score difference by participant group, APM minus Spec-kit, in points.	45
5.14	Paired per-section score difference by participant group, APM minus Spec-kit, in points.	46
5.15	Paired direction of each questionnaire rating across the twelve participants, APM against Spec-kit. The second column gives which direction is the better outcome for the item.	47
5.16	Distribution of each questionnaire rating across each framework's twelve sessions, the count rating the item at each point of the one-to-five scale, where one is the low anchor and five the high anchor.	48
6.1	Paired planning-phase behaviour differences, APM minus Spec-kit.	49
6.2	Compaction incidence by model and framework, the number of sessions that compacted at least once and the number that compacted more than once.	51
6.3	Paired implementation-phase behaviour differences, APM minus Spec-kit.	53
6.4	Compact-input diagnostic scores for the sessions whose score changed. These diagnostic scores are not part of the framework comparison.	54
6.5	Paired compact-input diagnostic total-score difference within the Sonnet group, APM minus Spec-kit. This diagnostic does not replace the framework comparison.	54
6.6	Change in total score from first session to second, for each participant whose score moved.	58
6.7	Cache reading above the 171k observed automatic-compaction bound in the four Sonnet conversations served past their stated context window limit, in thousands of tokens, with its corresponding session cost.	60
A.1	The post-session questionnaire's rating items, with the wording at each end of the one-to-five scale.	66

A.2	Per-token rates for the study window, in US dollars per million tokens.	66
B.1	Paired cost differences by realised session order, APM minus Spec-kit, in US dollars, at 95.3% coverage for the $n = 7$ row and 93.8% for $n = 5$	68
B.2	Paired time differences within each model group, by scope, APM minus Spec-kit, in seconds. .	68
B.3	Paired session-duration and turn-prompt differences by realised session order, APM minus Spec-kit. Duration is in seconds; the turn-prompt rows are counts. The $n = 7$ rows have 95.3% coverage and the $n = 5$ rows 93.8%.	69
B.4	Turn-time composition of the APM sessions, in seconds, with the concurrent span as a share of the turn time. Descriptive and APM only, since Spec-kit's single conversation has no concurrent span.	69
B.5	Paired turn-prompt count differences by scope within each model group, APM minus Spec-kit. .	69
B.6	Turn-prompt bucket shares within each model group, by phase and framework, each bucket by its median, Hodges-Lehmann estimate, and mean.	70
B.7	Turn-prompt bucket character counts within each model group, by phase and framework, each bucket by its median, Hodges-Lehmann estimate, and mean.	71
B.8	Per-role turn-prompt bucket shares and character counts across the twelve APM sessions, pooled over each role's agents. Descriptive and APM only, since Spec-kit has no agent roles to separate.	71
B.9	Steers per participant and framework, at session scope.	72
B.10	Per-session scores, in points. The total is the strict part plus the partial part, and the Python and C section totals sum to it.	73
B.11	Paired total-score difference by realised session order, APM minus Spec-kit, in points, at 95.3% coverage for the $n = 7$ row and 93.8% for $n = 5$	73
B.12	Responses to the comparison question, by the framework used in the second session. Each row is the participants whose second framework was the named one, and the counts are how that session felt against their first.	74
B.13	Self-reported completion state against the three-hour cap, by framework.	74

1. INTRODUCTION

AI assistants have expanded from completing code to acting as coding agents that autonomously carry out multi-step software tasks, up to end-to-end development [1]. As those tasks grow, the agent must preserve the project's intent while accumulating context competes for the model's finite attention [2].

Recent tools and research offer two responses relevant to this problem. Some record project intent in persistent documents before implementation [3]. Others divide work among agents, each responsible for a bounded part of it [4].

We present Agentic Project Management (APM), an open-source framework that operates on top of interactive coding agents and combines these responses: several role-specific coding agents execute work guided by persistent planning documents [5]. That combination is not unique. APM differs by making coordination user-mediated, aiming to preserve the user's authority over development. Its agents communicate through the user, who can review the work behind each message and request corrections before passing it on. How the work is divided and assigned sets how often such opportunities arise. The effects of this design have not been established in use.

To examine these effects, we propose a reproducible methodology for comparing frameworks on top of coding agents and implement it as a within-participant case study with twelve participants. Each participant completed the same demanding software task in two sessions, once with APM and once with Spec-kit, GitHub's open-source development toolkit for coding agents [6]. Both frameworks operated on Claude Code with Claude Opus 4.7 or Claude Sonnet 4.6. We compare their economic cost, process, functional test scores, and developer perception.

The comparison presents a trade-off and no clear winner. APM's cost varied by model, with a median of about \$35 against Spec-kit's \$24 under Opus and about \$15 against \$14 under Sonnet. It took longer in every pair, with a median session duration of about 119 minutes against Spec-kit's 71. The frameworks carried no firm difference in functional outcome across the cohort, although APM showed a more favourable pattern under Sonnet. Participants nevertheless rated APM more effective and confidence-inspiring, describing greater control over the development process.

In this thesis, we make the following contributions:

1. We present the design and implementation of APM, a development framework that combines persistent planning documents and role-specific division of work with user-mediated coordination.
2. We provide a reproducible case-study methodology to compare frameworks on top of coding agents, benefiting researchers and practitioners evaluating human-in-the-loop AI assistance.
3. We implement this methodology, with its own task, environment, measures, and supporting infrastructure, to evaluate APM alongside GitHub Spec-kit through a within-participant comparison.

The thesis is organised as follows: Chapter 2 introduces the concepts used throughout the thesis and establishes the research gap. Chapter 3 presents the APM framework. Chapters 4, 5, and 6 describe the case study, report its findings, and interpret them. Chapter 7 draws the findings together and outlines directions for future work.

2. BACKGROUND

2.1 AI Assistants and Agents

The tools that assist developers as they write code have changed quickly, from completing a single line to carrying out whole tasks on their own. Integrated development environments have offered identifier and interface completion since the 1990s. By the late 2010s, machine-learning models trained on large code corpora ranked those suggestions. Such systems retrieve and rank symbols that already exist; they do not compose new code. The generative shift came when large language models (LLMs), whose training data includes large quantities of code, began producing novel, multi-line completions in the editor. GitHub Copilot introduced this form in 2021 [7]. A distinct mode of interaction followed in late 2022, when general-purpose conversational assistants based on LLMs reached the public. These were not explicitly coding tools, but developers used them to explain unfamiliar code, stand in for documentation, think through a problem, and draft code snippets to copy and paste. This conversational mode soon entered the editor and then moved from advice to action. AI assistants began acting directly on software projects. Tools of this kind emerged over 2024 and 2025 [8], [9]. The systems under study descend from this conversational line.

We use the term *agent* in a specific sense. In its classical formulation, an agent is anything that perceives its environment through sensors and acts upon that environment through actuators [10]. In an LLM-based agent, the model serves as the central controller within an iterative loop. The surrounding system provides planning and memory and equips the model with perception and action mechanisms through which it can use tools and respond to its environment [1]. Within the loop, the model reasons about the goal, chooses an action, observes the result, and decides what to do next [11]. Accordingly, agent denotes a model-directed system with these capabilities. A *coding agent* applies them to a software project, allowing the model to inspect and edit files, run commands, and revise its work against the results.

The distinction between two forms of LLM-based agent matters to our comparison. A purpose-built application can implement an agent by calling a model through its programming interface, assembling the input it receives, exposing tools, and routing its outputs in code. Alternatively, an interactive coding agent can package that machinery as a general development tool to which the developer assigns work directly. A framework can then operate on top of that coding agent by supplying natural-language instructions, roles, and procedures for it to follow.

Attempts to measure the effect of these tools on developer productivity have reached conflicting conclusions across different generations of the technology. Field experiments with an earlier generation of assistants reported substantial gains. One study pooled three randomised controlled trials at Microsoft, Accenture, and a Fortune-100 company, in which nearly five thousand professional developers using an AI assistant completed around a quarter more tasks, with the clearest gains among the less experienced [12]. A later METR trial of early-2025 tools reached the opposite result among experienced open-source developers working on large repositories to which they regularly contributed. When allowed to use AI tools, primarily the Cursor editor [13] with Claude 3.5 and 3.7 Sonnet, they were roughly 19% slower even as they expected the tools to speed them up [14]. Their models, tools, tasks, and participants differ, so the effect appears to depend on those conditions. A third study varies neither the tool nor the task but the developer. Among a hundred students who built small graphical applications by prompting an agent, without ever seeing the code it wrote, those who scored higher on a computer-science test and those who wrote more clearly both produced better results, and the test scores accounted for the larger share of the difference [15]. These studies vary the tool, the generation, and the developer. None varies the workflow or framework through which a developer uses AI assistance, the dimension we compare.

2.2 Context Windows and Context Decay

Whatever its form, an agent is built on a model that accepts input and produces output in units called tokens. The model itself does not retain the conversation between independent calls. Instead, the surrounding system preserves the session's continuity by assembling prior messages, tool results, and other retained content as the next call's *context*. The *context window* bounds the span of input and output that the model can process in that call. This context can include the system prompt, the agent's instructions, tool calls and their results, and preceding exchanges retained by the system. The new output then becomes part of the history on the following call. At the time of writing, a frontier model's context window spans several hundred thousand to a million tokens, yet a long real-world task can still reach that limit [16], [17].

The growing conversation history creates strain on both capacity and cost, a condition we refer to as *context pressure*. Bringing the session closer to the context-window limit makes each later call process more input tokens than the one before it. Because model use is priced by the input and output tokens processed, the total input cost of a long session can grow faster than its number of turns [18]. Prompt caching can lower the price of a repeated prefix, but it does not remove those tokens from the context window, and cache reads and writes remain billable.

A model does not use this growing context evenly. Controlled experiments on long-context models found a U-shaped pattern. Performance is highest when the relevant information sits at the start or the end of the input and falls sharply when it lands in the middle [2]. Usable capacity is smaller than the raw token count suggests. In a coding session, subsequent exchanges push earlier decisions and instructions deeper into the model's context, where they can become harder to recover. The agent may then lose relevant information or reasoning consistency as this accumulation continues [2], [16]. In this thesis, we refer to the loss of recall and reasoning fidelity over extended context as *context decay*.

One response is *compaction* [16]. As a conversation approaches its limit, the surrounding system replaces earlier history with a shorter summary and continues from it. This reset frees space in the context window and reduces the input carried into the next call, but the omitted detail is no longer directly available to the model. Compaction exchanges fidelity for capacity rather than eliminating the accumulation problem.

Together, the decline in reliability and the rise in cost have made deliberate management of the context window a concern in its own right. Early practice centred on *prompt engineering*, structuring the natural-language input given to a model to elicit a particular response. As work grew into long agentic sessions, the harder question became what the context window should hold at each step, not only how to phrase a single request. This wider practice manages the whole body of context supplied to a model so that the agent's limited attention and tokens remain focused on relevant material. It is known as *context engineering* [19], [20]. The strategies in the rest of this chapter are different forms of it.

2.3 Spec-Driven Development

One way to keep an agent focused is to give it a durable specification of what it should build before implementation begins. In AI-assisted development, this approach is known as *spec-driven development* (SDD). Its underlying principle predates AI. Design by Contract, described in 1992 [21], makes the specification part of the program itself. It attaches to each routine the conditions assumed on entry, the guarantees required on exit, and the invariants preserved throughout. Test-driven development (TDD), codified in 2002, reverses the usual order of coding and testing. It begins with tests that define correct behaviour, so an executable specification of what the code must do precedes the code itself [22]. The name has an earlier history of its own. In 2004, an approach called specification-driven development [23] combined TDD with Design by Contract, treating tests and contracts as complementary kinds of specification. We reserve SDD for the current AI-assisted practice.

Beyond defining the intended software, the specification helps manage the agent's context. When developers improvise instructions turn by turn, their intent remains only in a lengthening conversation and can become difficult to recover. SDD instead fixes that intent in a durable artifact written before the code. The conversation still grows, but the agent can return to the specification and check its work against it. SDD is therefore one context-engineering strategy.

As a named practice in AI-assisted development, SDD took shape over 2025 in tools at different implementation layers. Amazon's Kiro, announced in July as an agentic development environment, built the practice into the tool itself. Its agent carries a developer from natural-language requirements through a technical design to sequenced implementation tasks [24]. GitHub's Spec-kit, released in September, instead supplies an external toolkit that brings a sequence of specification, planning, and implementation to an existing coding agent. It turns specifications into software on the premise that the *spec*, not the code, is the source of truth [3].

In the core workflows introduced by both tools, one agent carries the specification through planning and implementation. The spec-driven discipline itself does not determine how many agents perform the work.

2.4 Multi-Agent Systems

Another context-engineering strategy distributes work across several agents. A *multi-agent system* (MAS) consists of agents whose interactions contribute to a shared task. Their organisation varies: agents may discuss the same problem, take specialised roles, or receive separate parts of a decomposed task [25]. In the decomposed form APM uses, each part can be handled in its own conversation, reducing the accumulating context burden on any one of them and allowing independent work to proceed in parallel. The value of this decomposition depends on how readily the work can be divided and how well the agents coordinate.

Multi-agent systems have a long history in artificial intelligence, where they describe collections of autonomous entities acting within a shared environment [10]. Capable language models prompted a wave of research that applied the idea to LLM-based agents. During 2023, several systems organised these agents into role-specialised teams, including teams applied to software tasks. CAMEL paired agents in a structured dialogue [26]. ChatDev cast them as the staff of a virtual software company [27], while MetaGPT coordinated them through the documents and standard procedures of a development process [4]. Microsoft's open-source AutoGen framework supported collaborative conversations between agents that solved complex problems and wrote code [28]. These research systems belong to the purpose-built form introduced earlier. The code assembles the context and tools for each agent and routes messages between them, while the models choose actions within that scaffold.

The evidence that this approach improves performance compared with a single agent is weak. One study analysed more than sixteen hundred execution traces from seven multi-agent frameworks and found recurrent failures across their workflows, while gains over single-agent systems or simpler baselines were often minimal [25]. A separate empirical comparison found that the advantage of a MAS narrows as the underlying model grows more capable, so much of the benefit seen with weaker models recedes with stronger ones [29]. Together, these results suggest that any improvement from division of labour is neither reliable nor stable as the models improve.

Practitioner accounts identify similar limits for agents used in software development. Anthropic reports a large gain from a multi-agent system over a single agent on open-ended research, where the task allows broad parallel search, and identifies coding as a poor fit. Most coding work, it notes, divides into fewer genuinely independent pieces and depends on context that every agent would need to share [30]. Cognition similarly argues that splitting coding work disperses decisions and prevents agents from sharing enough context, leaving the resulting systems fragile [31]. Although these reports carry the weight of practice rather than controlled measurement, they consistently identify coordination as particularly difficult for the software work we examine. Yet an agent taking on a large task alone still encounters the accumulation described in Section 2.2. Its context

grows, its reliability decays, and the cost of each step rises. The single agent and the coordinated team each face a difficulty the other does not.

2.5 Frameworks on Top of a Coding Agent

Multi-agent coordination nevertheless appeared in everyday development through external frameworks built on the coding agents developers already use, rather than through the purpose-built research systems of Section 2.4. Their natural-language instructions, role descriptions, and methodology guides shape how an existing agent approaches its work. The field has not agreed on a name for this class. We refer to them as *frameworks operating on top of a coding agent*. Some guide one agent through a workflow. Others coordinate several agents and thus implement a MAS.

How such frameworks coordinate several agents has changed as coding agents have become more capable. In earlier forms, the user organised each agent in a separate conversation, assigned its role and task through explicit instructions, and carried work between conversations through shared files. BMAD initially followed this arrangement [32]. It assigned a role to each agent so that they collaborated like members of a software development team.

Later, coding agents gained the ability to delegate work to other agents launched into separate conversations. These delegated agents are known as *subagents*. They receive work from and return their results to the agent that launched them. Frameworks can build around this lower-friction capability. Superpowers, for example, uses reusable natural-language files to teach a coding agent to divide work into tasks and launch subagents to execute and review them [33]. Because the two arrangements are not exclusive, a framework may coordinate several user-facing agents while also using subagents within one or more of them.

2.6 Agentic Project Management and the Research Gap

We present and evaluate Agentic Project Management (APM). The author designed and built APM independently, first releasing it as open source in May 2025 [5]. APM operates on top of an existing coding agent and combines the context-engineering strategies of Section 2.3 and Section 2.4 by having several agents follow a spec-driven workflow. APM's first release predated Kiro in July and Spec-kit in September, two of the tools on which early practitioner reviews of SDD focused [34], [35].

APM has been revised continually as the coding agents beneath it and the underlying models improved. For this thesis, APM v1 was developed as a ground-up redesign whose roles, workflow, artifacts, and file structure differ substantially from earlier releases. In the rest of the thesis, APM refers to this release, and we write APM v1 where it must be distinguished from earlier ones.

One design choice persisted through that redesign: APM's coordination is user-mediated. Several agents work in their own conversations, and every exchange among them passes through the user. These exchanges give the user points at which to review the agents' work, in contrast to prompting an agent ad hoc and accepting its changes unreviewed, a practice known as *vibe coding* [36], [37]. While this arrangement began as the only way to coordinate several coding-agent conversations as a MAS, APM v1 retains it to preserve these opportunities for review and reflection throughout its workflow. File-based protocols govern both communication among APM agents and the spec-driven workflow they follow. Chapter 3 describes these architectural choices in full. Table 2.1 places APM v1 beside selected earlier systems, ordered by their first public appearance.

Table 2.1: Selected systems and how they handle long tasks and coordinate work.

System	How it handles a long task	How work is coordinated
GitHub Copilot (2021) [7]	Suggests code from the current editing context.	The developer accepts or rejects each suggestion.
MetaGPT (2023) [4]	Specialist agents receive information relevant to their roles.	The system routes structured messages through a shared pool.
Early BMAD (Apr 2025) [32]	Documents carry plans and decisions between roles.	The user moves work between role-specific conversations.
Spec-kit (Sep 2025) [3]	Keeps the specification, plan, and tasks in project files.	The developer steers a coding agent through specification, planning, and implementation.
Superpowers (Oct 2025) [33]	Keeps a written plan and gives each task a fresh subagent context.	The coding agent dispatches tasks in sequence and arranges a review after each.
APM v1 (2026)	Divides work across role-specific conversations and preserves plans, state, and history in files.	Agent messages pass through the user.

APM has attracted visible interest in the open-source community,¹ yet it has not been evaluated in use by developers. The studies of Section 2.4 evaluate purpose-built programmatic multi-agent systems on benchmarks. Frameworks operating on top of coding agents remain outside their scope. That omission matters because this framework class has entered everyday development through interactive workflows in which developers assign and steer work. A benchmark also supplies both the task and the measurement, and neither carries over once the developer is in the loop. The closest controlled human-participant study we found compares OpenHands, a purpose-built coding-agent application, with the completion-and-chat form of GitHub Copilot available at the time [38]. The two conditions therefore belong to different tool classes.

In the literature we reviewed, the emerging class of frameworks to which APM belongs, operating on top of coding agents with developers in the loop, remains unevaluated. In this thesis, we describe APM's design, propose a methodology for evaluating frameworks of this class, and apply it with human participants to compare APM against another framework that also applies spec-driven development.

¹On 17 June 2026, APM had roughly 2,300 stars on GitHub and its command-line tool had over 16,000 downloads from npm. By 18 July 2026, the readings were roughly 2,400 stars and over 19,000 downloads.

3. APM DESIGN

3.1 Design Goals

APM is a framework operating on top of a coding agent. It uses several such agents to form a MAS intended for substantial, real-world software development projects. Its design limits context accumulation by dividing a project across separate agent conversations and giving each agent a distinct responsibility: one plans the project, one distributes the work and reviews what comes back, and the rest carry it out.

Partitioning responsibilities across conversations, however, also separates the knowledge and decisions that keep the agents' contributions consistent with what they should collectively produce. To address this separation, APM uses shared files to make that information persistent and available throughout the project. They hold project intent, execution state, and information exchanged among agents, while explicit protocols define what each agent reads and writes.

Beyond coordinating its agents, APM aims to preserve user authority over development and agent interaction. The user holds final authority over project decisions, while workflow review points keep planning, coordination, and execution visible and open to intervention. Unlike standard coding agents that delegate to subagents autonomously within a single conversation, APM places the user directly at the communication boundary between agents, requiring human mediation for their exchanges. Prior work in human–computer and human–AI interaction has explored how deliberately placed pauses can invite reflection [39], [40].

3.2 Workflow Overview

APM realises these goals through a development workflow of two phases: a planning phase and an implementation phase. Planning begins with an agent exploring the project and questioning the user until they share an understanding of what the project requires. The agent records that understanding in documents that define the intended result, break down its implementation into parts, and set the rules for carrying it out.

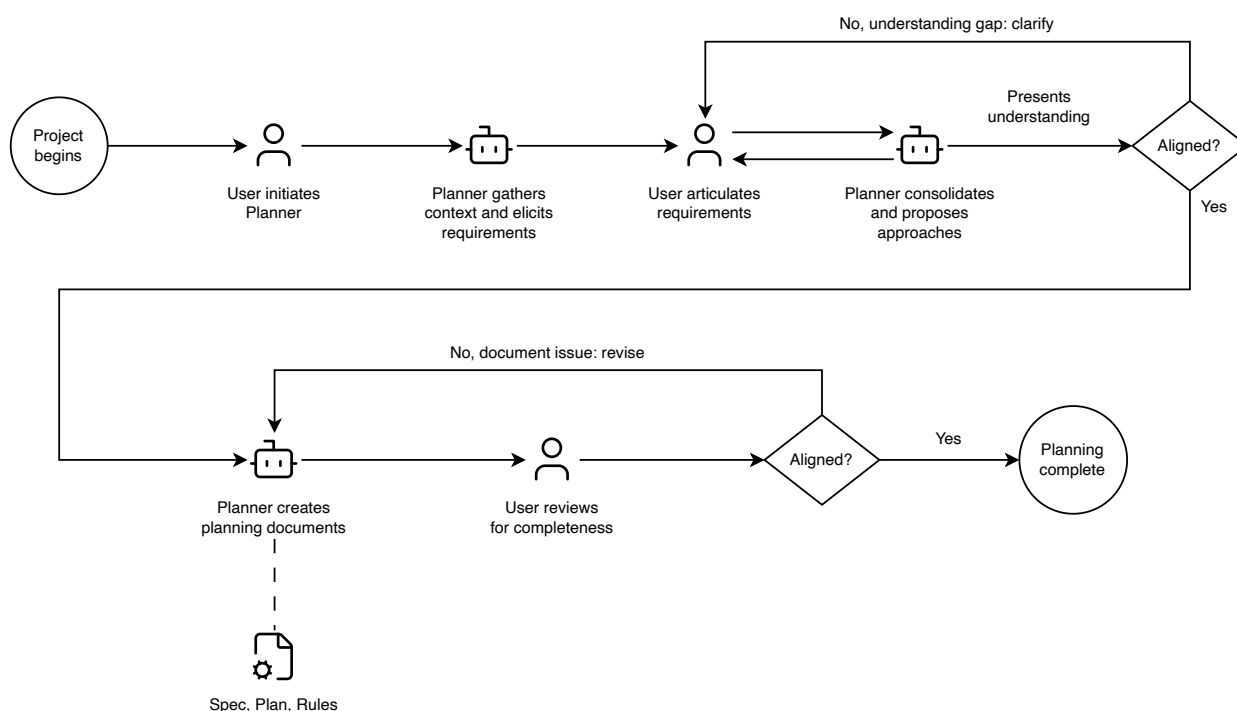


Figure 3.1: Planning in an APM project.

Implementation begins when a coordinating agent, acting much like a job scheduler, reads those documents and prepares work assignments for other agents. Each of those agents executes and validates its work, records the outcome, and reports back for review. Based on its review, the coordinating agent decides the next action and continues until the project is complete. Across this cycle, every exchange between the separate conversations passes through the user, who can review the work behind it and intervene at any point with direction or correction. Shared files hold assignments, reports, project state, and history.

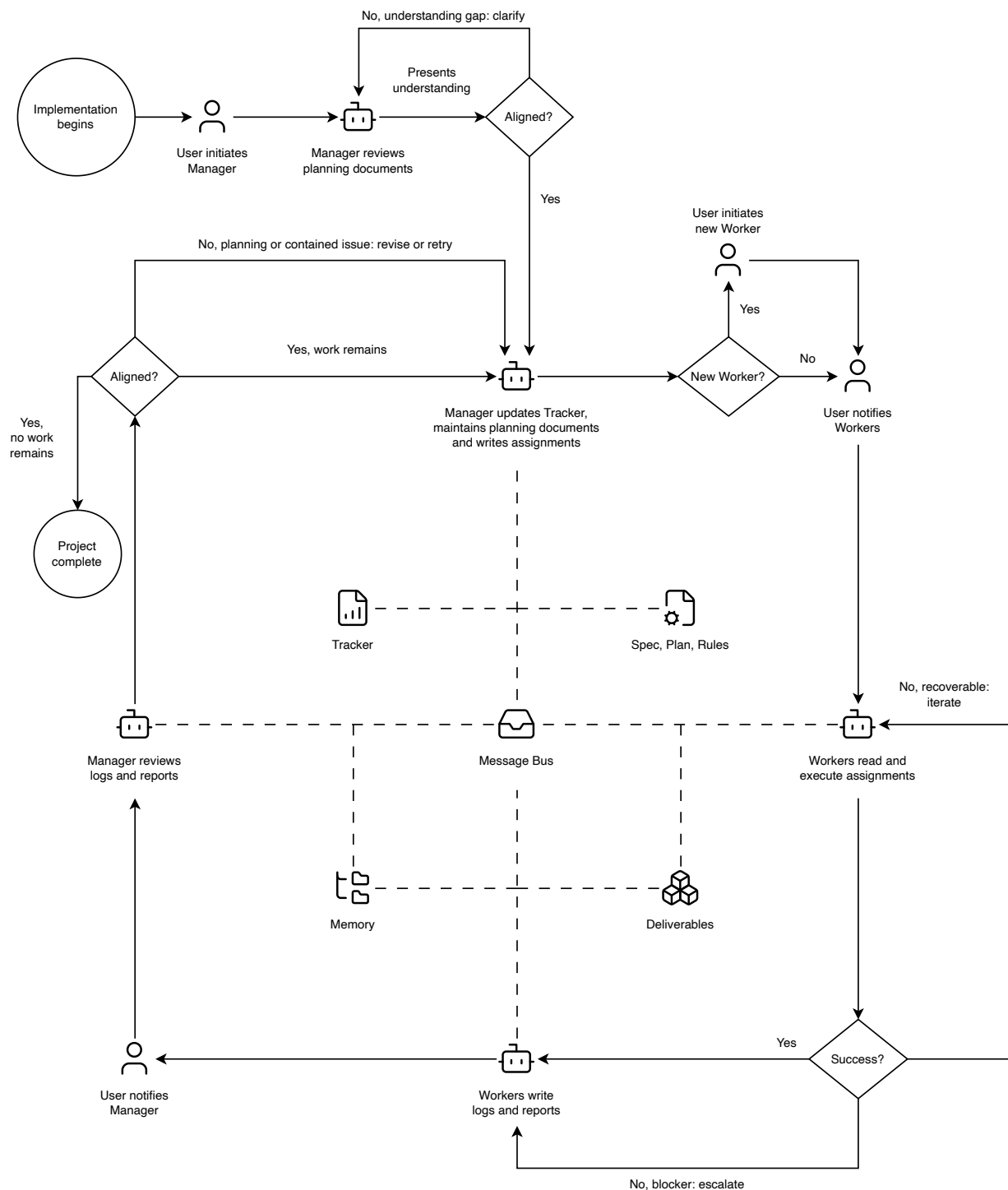


Figure 3.2: Implementation in an APM project.

Sections 3.3 and 3.4 define the roles and describe how user-mediated coordination takes place, while Sections 3.5 and 3.6 explain how APM preserves state and manages the project.

3.3 Agent Roles

APM organises its agents into three roles: the *Planner*, *Manager*, and *Worker*. Figures 3.1 and 3.2 place them in the workflow, while Table 3.1 summarises their properties. Each is a coding agent operating in its own conversation under role-specific instructions. Those instructions define the agent's responsibilities, workflow, and protocols, which determine the project information its conversation needs to hold. Planning involves one Planner, while implementation involves one Manager at a time and one or more Workers. Each retains the coding agent's ordinary tools and access permissions. Their specialisation comes from scoping the context that reaches them.

The Planner does research, resolves requirements and design choices with the user, and records the result in three planning documents. The *Spec* defines what to build. The *Plan* divides implementation into bounded units of work organised by domain. The *Rules* govern how their execution is carried out. The Planner's participation ends when implementation begins, leaving its conversation's full context window available to develop these documents.

The Manager coordinates implementation at project level. It distributes assignments to Workers and supplies the context required to execute them. It reviews their results, which form its context together with the planning documents and project state it maintains. The instructions the Manager receives establish its usual focus, while review or investigation may take it into code or execution detail.

Each Worker receives the assignments of one domain from the Plan, then executes and validates them. Workers follow the Rules. As context accumulates across successive assignments, they build familiarity with their respective part of the project.

When investigation or debugging threatens to consume a conversation's context, the agent launches platform-native subagents. Their findings return to the role's conversation for verification and integration.

Table 3.1: APM roles by phase, concurrent conversations, and context scope.

Role	Phase	Concurrent conversations	Context scope
Planner	Planning	One	Project definition and decomposition
Manager	Implementation	One	Project-level coordination
Worker	Implementation	One or more	Execution within one domain

3.4 Agent Coordination

The Planner produces its three planning documents through iteration with the user. Its accumulated context does not pass into implementation: the Manager receives the finished Spec, Plan, and Rules, whereas Workers receive only the Rules. Withholding the Spec and Plan protects execution context windows from project-wide details that are irrelevant to a single assignment.

A *Task* is a bounded unit of work from the Plan assigned to one Worker. A *Task Prompt* carries the information that the Worker needs to execute it. The Manager draws planned requirements from the Spec and Plan and adds context from related contributions. Familiarity determines the detail: a pointer may recall a Worker's own output, an upstream contribution specifies what was produced and how the Task depends on it, and downstream context indicates how future work will consume the result. After carrying out the work, the Worker returns a *Task Report*, which records the outcome.

The user creates each Worker by opening a new conversation for its first Task, and later Tasks reuse that conversation. As prerequisites complete and unblock downstream work, the Manager dispatches ready Tasks: individually, in sequential batches to a single Worker, or concurrently across separate domains.

These exchanges between conversations are carried through a file-based *Message Bus*. Each Worker has a *Task Bus*, where the Manager writes its Task Prompts, and a *Report Bus*, where the Worker writes its Task Reports. Figure 3.2 places this exchange in the implementation phase loop, while Figure 3.3 maps its two directions. A bus file is empty or holds one message awaiting delivery. Its recipient clears the file after processing that message, before the channel is reused. The user serves as the mediator between the separate conversations: when an agent writes to a bus file, it informs the user, who directs the recipient to read it.

These friction points are deliberate. They let the user exercise authority as decisions and results pass between agents. Before passing a message on, the user can review the work in any way they choose, such as reading the changes or running tests. They can ask its sender to correct the work and update the message, or raise the issue with its recipient. Mistakes or unwanted directions can thus be addressed before they propagate through the project.

The user can also shape how often these points occur. At any time, they can ask for smaller or larger Tasks, and for Tasks to be dispatched one at a time or in batches. Smaller Tasks sent one at a time give more chances to review, each covering less work. Larger Tasks and batches speed up implementation but give fewer, each covering more.

Under this file-based exchange, running turns involve only the Worker and the user, while the Manager remains idle until the user delivers the Task Report. An agent that encounters blocking issues or failing validation reports incomplete or failed progress for the Manager to review. If an agent hangs or strays during execution, the user can interrupt its conversation directly to stop it.

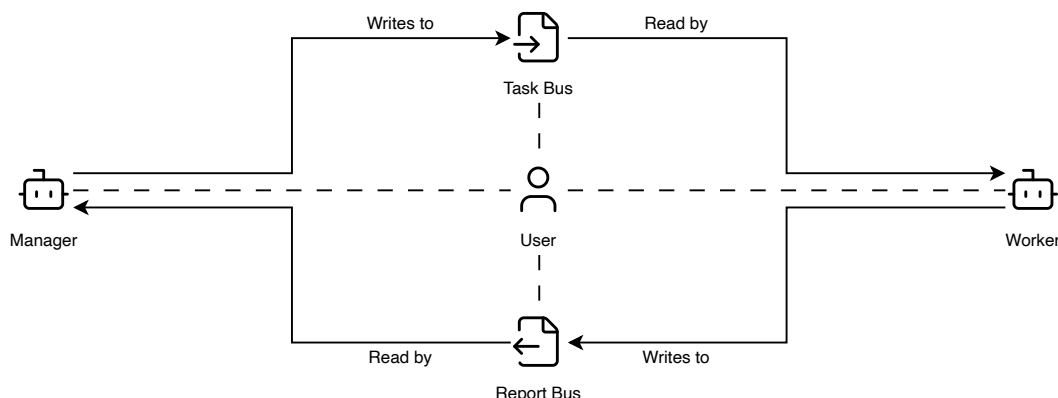


Figure 3.3: Message Bus exchange between the Manager and a Worker.

3.5 State Management

The planning documents define the intended work, but execution may reveal that they need to change. The Manager can revise them within the project's established direction, while Workers may propose changes to the Rules. Changes to project scope, however, require explicit user direction and approval.

During implementation, the Manager also maintains documents for coordination and project history. The *Tracker* is its live record of where the work stands, holding the status of every Task, the Worker on each one, and the version control state and conventions in force. Working notes accumulate in it, carrying pending considerations and findings that may affect later decisions. *Memory* is a hierarchically organised collection of files that preserves project history. Its *Index* provides the overview: summaries of completed work with references to their underlying records, and durable observations distilled from the Tracker's working notes, including established patterns, preferences, and project-level insights. Figure 3.2 shows these documents in the implementation phase loop.

After executing a Task, its Worker writes a *Task Log* that reduces accumulated execution detail to what the

Manager needs for coordination: the outcome, validation results, deliverables, file changes, and findings bearing on the wider project. The Task Report points to this log for review. Condensing execution and coordination detail lets the Manager oversee several Workers, dispatch and review many Tasks, and maintain APM's project state without its own context growing in proportion. Figure 3.4 summarises how both enter Memory.

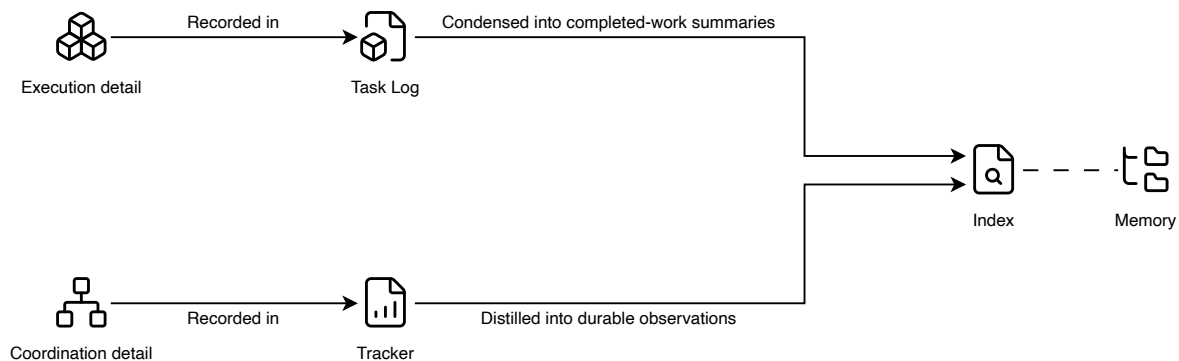


Figure 3.4: Compression of execution and coordination detail into persistent project history.

To address context pressure and decay in implementation-phase agents, the user invokes a *Handoff* on demand, transferring the role from the conversation holding the accumulated context to a fresh one. The outgoing agent follows the framework's instructions to record what must carry over in two documents. The *Handoff Log* records past work, decisions, and working knowledge not held in Task Logs, and joins them in Memory. The *Handoff Prompt* describes the current state and tells the incoming agent what to read and do next. The user carries it through the role's *Handoff Bus*. The incoming agent rereads its role instructions, then uses both documents and relevant project files to resume from where the outgoing agent left off. Figure 3.5 traces the transfer.

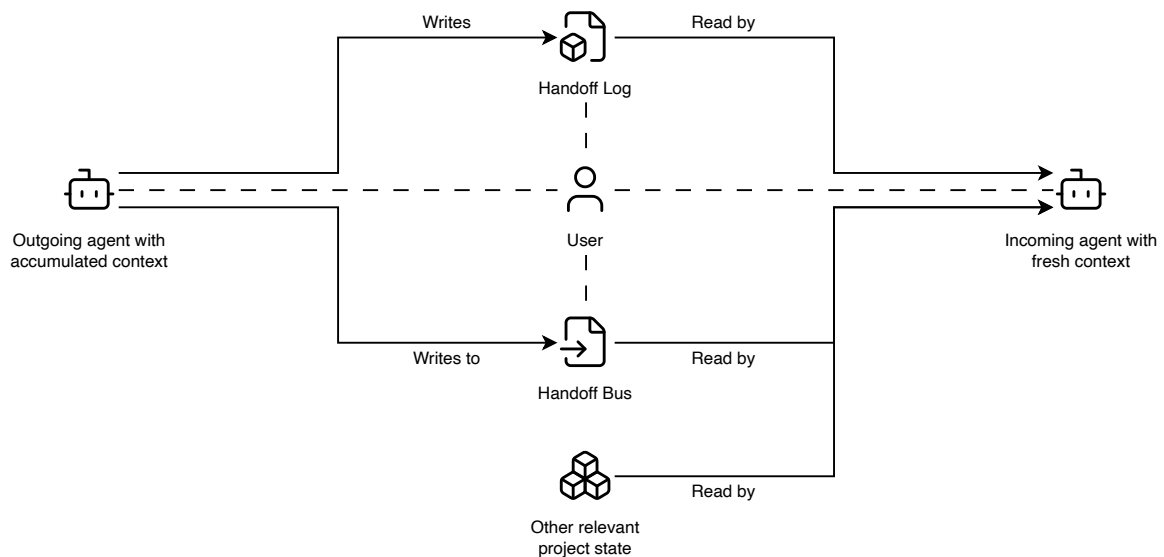


Figure 3.5: Handoff between outgoing and incoming agent conversations.

The context window limit is the usual occasion for a Handoff to occur, but the user can invoke one earlier to reduce cost or try to restore performance. The system hosting the agent can also reset accumulated context through compaction, described in Section 2.2, but the resulting summary is ephemeral to the conversation. If the conversation fills again, the next compaction condenses it together with the context accumulated since, further reducing the detail retained from earlier work. A Handoff instead records the transfer in persistent files, so successive Handoffs retain a durable history of the role's work.

When an agent loses its working state without having performed a Handoff, the user can invoke APM's recovery procedure. The agent then rereads its role instructions and relevant project files to restore the framework's procedures and establish where its work stands.

3.6 Project Management

APM operates through structured documents and commands. Its artifacts use Markdown, often paired with YAML fields that identify state and determine procedural branches. APM packages each command as a pre-written prompt and adapts its invocation syntax to each coding-agent platform it supports. Table 3.2 lists the commands in the slash-prefixed form common to those platforms. The user invokes each command in the appropriate conversation.

Table 3.2: APM commands by addressed agent and function.

Command	Addressed agent	Function
/apm-1-initiate-planner	Planner	Initiate a Planner conversation
/apm-2-initiate-manager	Manager	Initiate a Manager conversation
/apm-3-initiate-worker	Worker	Initiate a Worker conversation
/apm-4-check-tasks	Worker	Deliver a Task Prompt to the Worker
/apm-5-check-reports	Manager	Deliver Task Reports to the Manager
/apm-6-handoff-manager	Manager	Initiate a Manager Handoff
/apm-7-handoff-worker	Worker	Initiate a Worker Handoff
/apm-8-summarize-session	Standalone agent	Summarise the project and optionally archive its state
/apm-9-recover	Manager or Worker	Reconstruct the role's working context

Each command establishes a role and points the agent to the procedures its next step requires. Detailed procedures live in separate guides, each addressed to a single role, which the agent reads when it reaches the step that needs one. Instructions shared across roles, such as the Message Bus protocols, are packaged as Agent Skills [41].

3.6.1 Planning Phase

Planning begins with the Planner building an understanding of the project. It reads the materials the user identifies as authoritative and explores any codebase the project involves before questioning the user, so its questions address what remains unknown. The Planner may delegate substantial codebase research to subagents, then verifies their findings before using them in its questions or planning documents. Successive rounds go deeper, covering the project's purpose and deliverables, then its technical requirements and dependencies, then the constraints, standards, and conventions under which the work should be carried out. Where research or the user's answers open more than one viable approach, the Planner presents the alternatives with their trade-offs and the user chooses.

The questioning ends with a consolidated account of the Planner's understanding, and planning advances only once the user approves it. Corrections send the Planner back to the material: it applies what the user raised, asks whatever the correction opens up, and presents the revised account again.

The Planner then produces the Spec, the Plan, and the Rules, one at a time. It presents its reasoning for each in the conversation before writing it, and each document passes through that same review cycle before the next begins. Written against each other in turn, the three hold one account of the project.

The Spec records the decisions that shape what is built: the choice made, the alternatives it displaced, and the constraints that bound it. Its structure follows the project's concerns, and where the user's documents already define a requirement it references them instead of restating it. Decisions are kept separable so the Manager can extract the ones a given Task needs.

The Plan organises the work. It partitions the project into cohesive domains, separating areas requiring distinct expertise or mental models, and combining those sharing tight context and dependencies. Each domain gets its own Worker. For every Task, it defines the deliverables, the validation criteria, and the Tasks it depends on. A *Stage* is a group of related Tasks that delivers a coherent part of the project and begins only after the preceding Stage completes. Within a Stage, a Task waits for the ones it depends on, while those whose dependencies are complete can proceed together. Parallel work therefore happens only within a Stage, as Figure 3.6 shows.

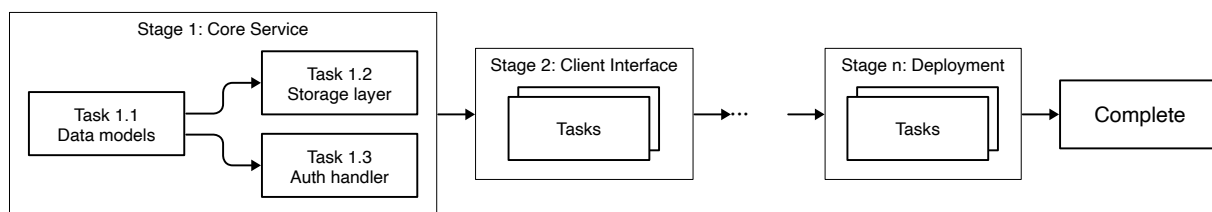


Figure 3.6: Progression of a project through its Stages, illustrated with modular service tasks in Stage 1 and compressed representations of the Stages that follow.

The Rules hold the execution patterns that apply across all or most Tasks, such as coding conventions and quality standards. Every Worker receives the same Rules, yet a requirement is not a Rule merely because it reaches every Worker: an output format or interface contract belongs to the Spec.

3.6.2 Implementation Phase

Implementation begins with the Manager reading the planning documents and presenting its understanding of the project and the conventions it will work under. It dispatches the first Task once the user approves.

A Task is ready when the Tasks it depends on are complete. Its Worker works from the Task Prompt, the Rules, whatever it needs from the project's files, and the context it has accumulated over earlier Tasks. It executes the work and validates it against the Task's criteria, iterating where validation fails and pausing for the user where a criterion needs the user's judgement or an action outside the development environment. A difficult failure may be delegated to a debugging subagent, whose findings the Worker verifies before applying them. It then reports the outcome it reached: the Task complete, partial progress where iteration stalled, or a failure.

The Manager reviews the Task Report and the Task Log. An outcome short of complete, or a Worker's note that execution touched something beyond its Task, prompts investigation before anything is accepted. The Manager handles contained checks directly and may delegate context-intensive investigation to a subagent. The findings of this review decide among three outcomes. The work stands as it is, the Worker retries under a refined Task Prompt, or the planning documents need updating, as Section 3.5 describes. In each case the Manager records the result in the Tracker and reassesses which Tasks have become ready. Figure 3.7 follows one Task through this cycle.

The Manager uses version control to keep each dispatched Task's work separate. At the start it establishes with the user which repositories the work targets and what branch and commit conventions each follows. Work

for a dispatched Task goes on its own branch off the agreed base of the repository it touches, and concurrent Workers each get an isolated working copy of it through a git worktree. Workers commit their own work, while merging belongs to the Manager alone. This way work lands on the base branch only after passing the user who mediated its report and the Manager who reviewed and accepted it.

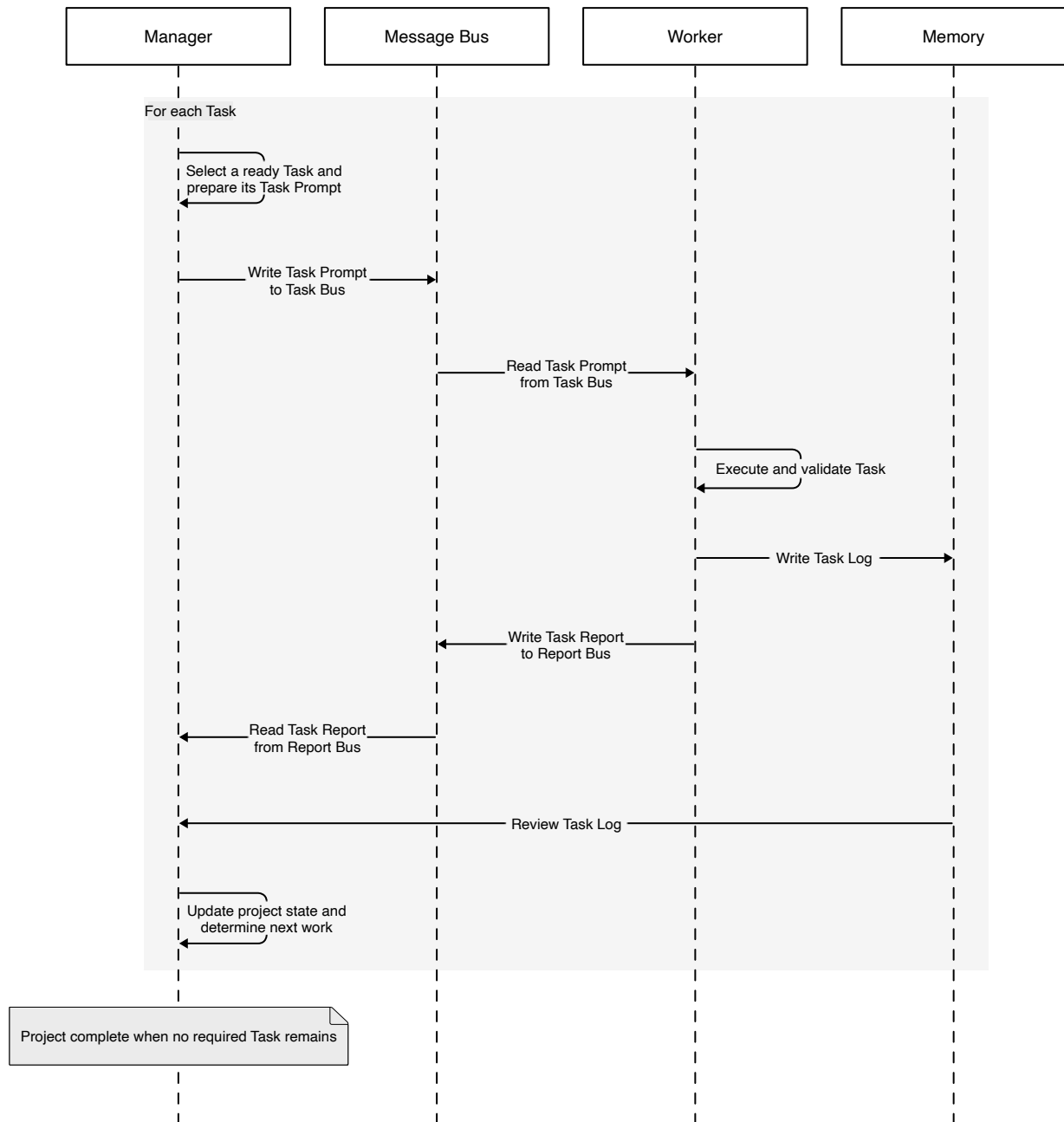


Figure 3.7: One Task through the implementation cycle. The Task Prompt and the Task Report travel through the Message Bus, while the Task Log is written to Memory and reviewed from there.

When every Task of a Stage is complete, the Manager closes it by writing a summary into Memory's Index and distilling the working notes the Stage accumulated. Those with lasting value join the Index as durable observations. Notes needed by the next Stage stay in the Tracker, and the rest are cleared. The Manager then opens the next Stage. When no Stage remains, the Manager records the project as complete and presents an account of what was built.

At project completion or any point requested by the user, APM can archive all files that hold project state and

optionally add a document summarising their scope, outcomes, decisions, deliverables, and issues. A later Planner can use APM's custom archive-research subagent to inspect a relevant archive and inform project research and planning.

3.7 Distribution and Customisation

APM keeps a single platform-neutral source of commands, guides, skills, subagent definitions, and project artifacts as templates in its GitHub repository. Coding-agent platforms differ in where and how they accept these documents. During a release build, the build system inserts each platform's unique paths, formats, and syntax into the templates, and the resulting packages are published in a GitHub Release with a manifest identifying which package belongs to each supported platform [5].

A command-line tool, `agentic-pm`, provides the user interface to this distribution. It is released separately through npm [42]. The user selects a platform; the tool fetches a compatible release, resolves the corresponding package from its manifest, and scaffolds it into the project's workspace. It records the source and release for later updates. The tool and templates follow independent semantic versioning, with only the major version shared for compatibility.

The same architecture also makes the workflow customisable. Within their own copy or fork of the repository, users can extend or adapt any part of the framework's workflow, protocols, and procedures. The build system turns that custom source into platform packages and a release manifest that users can publish as their own GitHub Release. Through its custom-repository interface, the command-line tool fetches and scaffolds these releases in the same way as the official distribution.

Two official variants demonstrate this extension surface. APM Auto [43] replaces the original workflow's separate, user-mediated Worker conversations with Worker subagents launched directly by the Manager. APM Semi [44] retains the original workflow but gives the user more control by letting them claim any Task to complete themselves, with the Manager or assigned Worker standing by for assistance and validation.

4. EVALUATION METHODOLOGY

4.1 Evaluation Scope and Research Questions

We examine the research gap established in Section 2.6 through a reproducible case-study methodology, implemented here as a within-participant case study of APM and Spec-kit. To the best of our knowledge, this is among the first methodologies for evaluating frameworks of this class in use by developers.

Both frameworks belong to the class introduced in Section 2.5, and each has its own realisation of SDD described in Section 2.3.

The comparison treats each framework as a complete workflow, including how it carries specification, planning, and implementation and how it coordinates the agents involved. Four Research Questions (RQs) organise the primary measured comparison:

- **RQ1 (Economic):** How do APM and Spec-kit differ in the dollar cost of completing a demanding task, and is that difference consistent across models of differing capability?
- **RQ2 (Process):** How do APM and Spec-kit differ in the time the task takes to complete and in the form and frequency of the user's recorded interactions with the frameworks' agents?
- **RQ3 (Functional):** How do APM and Spec-kit differ in functional outcome, including how much of the task each completes and how they fail when they fall short?
- **RQ4 (Perception):** What are the participants' self-reported experiences with each framework and how do they relate to the behaviour measured in their sessions?

4.2 Spec-kit, the Comparison Framework

Spec-kit is GitHub's open-source spec-driven development toolkit [6]. Evaluating APM alongside Spec-kit compares two realisations of spec-driven development, contrasting single-agent and multi-agent coordination. In Spec-kit, one agent applies the discipline from end to end, producing a specification and a plan and then implementing them, all within one continuous conversation. That agent may still launch subagents through the underlying coding agent, just as APM's agents can.

Spec-kit's workflow is driven by a set of commands that the user triggers or the agent invokes on its own. Each is a pre-written prompt whose instructions walk the agent through one step of the workflow. The first is `/speckit-constitution`, which fixes the project's governing principles in a file; these come either from the user or from the agent's reading of the existing project, and the agent is to uphold them throughout. The core workflow then produces three successive artifacts:

1. `/speckit-specify`: directs the agent to produce a structured specification of requirements, user stories, and acceptance criteria from a feature description.
2. `/speckit-plan`: has it turn that specification into a technical implementation plan with architecture decisions, data models, and API contracts.
3. `/speckit-tasks`: instructs it to derive an executable task list from the plan, annotated with dependencies and sequencing constraints.

Finally, `/speckit-implement` has the agent work through the task list in sequence, while further commands support iterative refinement, cross-artifact consistency checking, and custom quality checklists along the way.

Across these steps the agent works largely on its own. It fills gaps in a brief with documented assumptions and puts a question to the user only where a choice is consequential or genuinely open, offering each as a short table of options with a recommendation.

4.3 Case-Study Conditions

The study was conducted from 5 May to 5 June 2026. Both frameworks operated through the Claude Code command-line interface, and participants used the Claude Opus 4.7 or Claude Sonnet 4.6 models. Pricing, model, and framework descriptions refer to this study window.

The study used version 1.0.1 of both the APM templates and the `agentic-pm` command-line tool [42], [45]. It used Spec-kit versions from v0.8.5, tagged on 4 May 2026, through v0.8.13, tagged on 21 May 2026 [46]. Because a fair comparison needs a fixed target, we audited that span release by release over the instructions that reach the agent and compose the framework’s workflow. The change across the eight release steps is minimal, and we therefore read the comparison as operating against one fixed version. Appendix A.1 reports the audit, and Section 6.7 treats the risk it leaves.

4.4 Case-Study Methodology

The case study employs a within-participant comparison across twelve university students and working professionals. Each participant attempts the same task twice, once per framework, holding individual skill, prior experience, and working style constant across both attempts. Each attempt has a three-hour limit.

Participants receive materials bundled as an open-source task package: a task brief, a portable testing environment, and a self-contained guide that walks them through setup and their task attempts. An optional agent helper skill accompanies the guide to automate or assist with setup; agents evaluated in the study sessions do not have this skill. The two subsections that follow detail the brief and the environment. Appendix A.2 lists all package materials.

4.4.1 Task Specification

The task is delivered to each participant as a product requirements document (PRD) that defines its objectives, requirements, and acceptance criteria. It builds on `eclass-mcp-server` [47], an existing open-source Python Model Context Protocol (MCP) [48] server that connects to Open eClass [49], the learning-management platform used across Greek academic institutions. The existing server exposes four MCP tools for authentication and course retrieval. Participants are required to add two tools that retrieve course and platform announcements. This requires investigating the Open eClass PHP codebase to determine how the platform exposes those announcements, then implementing the new retrieval behaviour in the Python server.

Participants are also required to build a standalone native C replica that exposes the four existing and the two new tools and matches the Python server’s input schemas, responses, and error behaviour. The Python server receives its MCP and JSON-RPC protocol machinery from MCP’s Python SDK; the C replica must implement that layer itself. Additional constraints from the PRD prohibit the use of external JSON-parsing libraries and require the C program to leak no memory across the supported tool-session surface under Valgrind. Together, the two deliverables give both frameworks a demanding end-to-end problem, spanning codebase investigation, application behaviour, and manual protocol implementation across the Python server and its native C replica.

4.4.2 Task Environment

Each participant works in a freshly created, dedicated Linux virtual machine. The participant's workspace contains the PRD and an opening prompt alongside two pinned codebases: `eclass-mcp-server`, which they are required to modify, and Open eClass, which they can investigate as a reference. Installed in its own directory inside the same virtual machine, the portable testing environment runs a containerised instance of the same Open eClass release together with a mock single-sign-on service. The installed instance holds a deterministic mock dataset and is reached over localhost, letting participants develop in their workspace and validate their work against a live platform on the same virtual machine.

4.4.3 Sessions and Data

We call one attempt to complete the task with the assigned framework a *study session*. It begins with the first prompt sent to the agent and ends when the participant stops task work or reaches a three-hour cap. Across the cohort, this gives twenty-four study sessions. Between each pair the environment is reset to its initial state, so that no code, workspace modification, or assistant context from the first session carries into the second.

Participants were assigned one of the two models across both study sessions, forming two six-participant model groups. Framework order was assigned separately, placing APM in the first or the second session. Counterbalancing the framework order lets us investigate whether improvement from growing task familiarity can be distinguished from the framework difference. Together, the two assignments formed four model-and-order groups of three participants each.¹

A post-session procedure has the participants package their solution patch written against the pinned Python codebase and their Claude Code transcripts for every agent conversation they opened. They then submit that package and answer a short questionnaire.

4.5 Metrics

This section defines the study's metrics, which combine automated computation with participant self-reporting, and the terminology used with them. Automated tooling built for the study extracts the objective outcomes deterministically: a transcript parser processes the participants' Claude Code conversation logs to compute the economic and process metrics, while a test-based grader scores the participant's code patch against the functional assertions. A post-session questionnaire supplies the perception metrics. The parser and grader belong to the open-source infrastructure of Appendix A.2, and Appendix A.3 details the questionnaire items.

Every transcript-derived measurement covers the three-hour attempt window. Records timestamped more than three hours after a session's first user message are excluded before any measurement is computed. Any excluded tail is flagged for review before the session's measurements are accepted.

Each transcript-derived measurement is computed at one or more scopes. A measurement scope is the level of the recorded workflow over which a value is aggregated. The session scope covers the whole study session. The phase scope gives separate values for planning and implementation. The role scope gives APM-only values for each agent role, since Spec-kit has no roles to separate. A measurement breakdown further separates a value at one scope into named components.

¹One participant departed from their assigned order in practice: a tooling failure ended their first Spec-kit attempt before any task work was completed, they ran APM the following day, and then repeated Spec-kit. Analyses by order therefore use realised framework order. The realised APM-first and APM-second groups contain seven and five participants. Within them, the two realised Opus model-and-order groups contain three participants each, while the two realised Sonnet model-and-order groups contain four and two participants. All tables grouped by realised framework order in Appendix B.1, Appendix B.2, Appendix B.3, and Appendix B.4 use this classification.

The phase and role scopes follow the frameworks' own workflow as recorded in the transcript. Spec-kit's single conversation turns to implementation when its workflow reaches `/speckit-implement`, whether the participant types it or the agent invokes it on its own. APM divides by role, with its Planner conversation forming planning and its Manager and Worker conversations forming implementation.

Within a phase, each measurement is attributed as the work happens: cost, tokens, and model-request counts follow the individual record, and a turn that crosses from planning into implementation divides its time at the crossing. The counts of user messages instead follow the turn, attributed to the phase in which the participant opened it.

4.5.1 Economic

The economic measurement is the *dollar-equivalent cost*, computed by pricing the recorded token usage of each billed model request against Anthropic's published per-token rate card in effect during the study window, recorded in Appendix A.4. Cost is measured at session and phase scope, and for APM at role scope. Its session-scope value is also broken down by token category to locate where the spending accrues.

4.5.2 Process

The process measures describe the session's temporal shape and the user's interaction with the agent.

The temporal shape of the session is captured by the session duration and the two measures that decompose it, the turn time and the inter-turn time, all reported at session and phase scope only:

- *Session duration* is the wall-clock span from the prompt that opens the first turn to the latest retained timestamped transcript record within the three-hour attempt window.
- *Turn time* is the wall-clock union of the turn intervals, each spanning from its opening prompt to its last retained assistant record. Its breakdown separates the time covered by exactly one interval (*solo*) from the time covered by two or more intervals (*concurrent*). For APM this shows whether its agents' turns overlap or proceed serially.
- *Inter-turn time* is the part of the session duration covered by no turn interval. It is not read as a measure of participant activity, since a gap may be spent reading, thinking, or away from the task.

The interaction shape is captured through two forms of user input:

- A *turn prompt* is a message submitted while the agent is idle, opening a new turn that lasts until the agent's idle return.
- A *steer* is an interjected message submitted while the agent is still working. In Claude Code these are enqueued until the return of the agent's next tool call, which delivers the message to the agent mid-turn unless the participant has removed it from the queue first.

Turn prompts are counted, and each falls into one of three content buckets:

- *Protocol* for a prompt that invokes an APM or Spec-kit framework command.
- *Short* for a non-command prompt below 50 characters.
- *Standard* for a non-command prompt of 50 characters or more.

The turn-prompt breakdown is described by the *bucket-share distribution*, which gives each bucket's share of the prompts, and the *magnitude per bucket*, their mean character count within each bucket. Together with the *turn-prompt count*, these measures describe how often the user opened turns, the form those prompts took, and the amount of text recorded in each form.

Unlike the temporal measures, the turn-prompt measures are also reported at role scope for APM, attributing each prompt to the agent the participant was addressing.

Steers are reported at session scope only, as two counts: those *delivered* to the agent and those *not delivered*.

4.5.3 Functional

The functional dimension is measured by ninety atomic functional tests applied to each submission, each testing one expected behaviour and carrying one or two points towards a hundred-point total. The grader tests the resulting behaviour against the MCP surface required by the task. It observes only the running Python server's and the compiled C binary's responses, never the participant's source, and is therefore indifferent to how the solution is built. It runs each submission against an environment equivalent to the one participants used in their sessions, with its own seed data for finer-grained checks. Appendix A.5 specifies the rubric's weighting and its partial-credit rule.

Each check returns one of three verdicts. A *passed* check matches an accepted form and earns its full points. A *partial* check matches no accepted form exactly but comes close to one, measured by Levenshtein edit distance [50] and computed with the Wagner-Fischer algorithm [51]; it earns points scaled by that similarity and is flagged for review. A *failed* check earns zero.

The *total score* sums the points earned across the rubric. Its breakdown separates a strict part, the full points earned by passed checks, from a partial part, the fractional credit earned by partial checks. A task-section breakdown divides the 100 points between the Python extension, worth 30, and the heavier C replica, worth 70. The resulting *per-section scores* locate where each framework's work lands.

4.5.4 Perception

The perception measures come from the post-session questionnaire, the self-reported counterpart to the measured behaviour. Its core is eight *rating items* on a one-to-five ordinal scale, each answered once per session and paired by framework. The items cover the participant's prior familiarity with the framework, confidence in the submitted solution, the framework's perceived effectiveness, the session's mental demand and frustration, the overall effort invested, and the effort spent checking and steering the agent. The items differ in polarity, a higher rating being the better outcome for some, such as confidence and effectiveness, and the worse for others, such as mental demand, frustration, and the three effort items.

Beyond the rating items, the questionnaire asks a *comparison question* after the second session: whether the task felt easier or harder than with the first framework. It also records the participant's own *completion state* against the three-hour cap. It closes with two free-text prompts, read as the participants' own accounts, on what stood out and any closing remarks.

4.6 Analysis Plan

Our findings lead with the measured values for APM and Spec-kit. Where a measure supports a within-participant comparison, it is followed by each participant's APM-minus-Spec-kit difference and summaries of its sign, typical size, and uncertainty. Model-group breakdowns present the respective group's paired differences separately and compare them descriptively.

Each participant i contributes the paired difference $d_i = x_i^{\text{APM}} - x_i^{\text{SK}}$, where SK denotes Spec-kit. The cohort analysis uses these twelve differences, while each model-group breakdown uses the group's six. Since the Opus and Sonnet differences come from different participants, a contrast between the groups carries their differences in skill and working style alongside the assigned model.

The number of positive against negative differences gives the sign count, following the convention of the paired sign test [52]. It reports the directional consistency of the difference across participants without assuming symmetry or a specific distribution shape, with zero-difference pairs contributing to neither count.

The typical difference is reported through the median, the Hodges-Lehmann (HL) estimate [53], and the mean. The Hodges-Lehmann estimate is the median of the averages of every pair of differences,

$$\hat{\theta} = \text{median} \left\{ \frac{d_i + d_j}{2} : 1 \leq i \leq j \leq n \right\}.$$

It reflects more of the observed differences than the median while being less affected by a few large differences than the mean. Read together, the three show the centre of the paired differences and whether asymmetry changes that reading.

A confidence interval (CI) based on the exact Wilcoxon signed-rank procedure [52], [54], [55] captures the uncertainty surrounding the typical difference. Rather than approximating the distribution, the procedure computes it directly, fitting a small sample where normality cannot be assumed. The test statistic ranks the absolute paired differences and sums the ranks of the positive differences,

$$W^+ = \sum_{i: d_i > 0} \text{rank}(|d_i|).$$

Under the null hypothesis, each rank is equally likely to be positive or negative. Comparing W^+ against this distribution determines which values of the typical difference are compatible with the data; those values, expressed in the original units of the paired differences, form the confidence interval. Its exact coverage is discrete: 95.8% across the twelve cohort pairs and 96.9% across each model group's six pairs. At six pairs, its endpoints are the smallest and largest observed differences, so it excludes zero only when all six differences share the same sign.

The dollar-equivalent cost, temporal process measures, turn-prompt count, and total score are each presented first through the two frameworks' measured values and then through their within-participant paired difference.

Measurements at phase or role scope, and their component breakdowns, are compared within participant when they give one directly comparable value for each framework. This applies to phase-scope cost, temporal measures, and turn-prompt count, as well as cost by token category and score by task section. Role-scope values and turn-prompt bucket distributions are presented descriptively through their median, Hodges-Lehmann estimate, and mean, without a paired difference or confidence interval. The solo and concurrent breakdown of turn time is shown per participant because concurrent time applies only to APM.

When an irregularity in one participant's data materially shifts a summary, the affected table row identifies it. Alongside these summaries, participant-level values are shown when the individual pattern is needed to read the result or when the events are too sparse for an aggregate summary.

The perception ratings are ordinal, so each item is summarised by order-based statistics. These are its sign count, read with the item's polarity, and each framework's full distribution across the five points. Responses to the comparison question and the completion state are presented descriptively as categorical distributions.

The interpretation in Chapter 6 draws further on supplementary descriptive counts where they support a specific reading. These counts are supporting evidence and not additional measures of the framework comparison, and their presentation follows the question they help interpret.

These statistics are computed, and the figures and table data generated, deterministically by the analysis scripts listed in Appendix A.2.

5. EVALUATION RESULTS

5.1 Economic

An APM session cost a median of about \$35 under Opus and \$15 under Sonnet, against Spec-kit's \$24 and \$14, as Table 5.1 reports. APM was the more expensive of the two under both models, by a wide margin under Opus and only slightly under Sonnet. Within a model each framework's median, Hodges-Lehmann estimate, and mean lay close together. The two model groups are read apart, however, since the Opus sessions cost roughly twice the Sonnet ones at that model's higher rate. The two frameworks placed that cost differently across the task's phases. APM's fell mostly in implementation, at a median close to \$28 under Opus and \$12 under Sonnet, while its planning cost was the smaller of the two and stayed below Spec-kit's.

Table 5.1: Per-framework session and phase cost, in US dollars, across each framework's six sessions in each model.

Group	Scope	Framework	Median	HL	Mean
Opus	Session	APM ^a	35.20	35.57	36.67
		Spec-kit	23.94	23.94	24.08
	Planning	APM	5.50	5.85	6.53
		Spec-kit	6.20	6.84	7.04
	Implementation	APM	27.74	27.74	29.89
		Spec-kit	17.73	16.69	16.98
Sonnet	Session	APM	14.58	14.58	14.54
		Spec-kit	13.54	13.54	12.86
	Planning	APM	2.67	2.67	2.66
		Spec-kit	3.77	3.81	4.28
	Implementation	APM	12.22	11.71	11.83
		Spec-kit	9.76	9.69	8.58

^a P009's assigned-Opus APM session records both Opus and Sonnet tokens in their APM agent transcripts. The row prices calls as recorded, so it is not a pure Opus session. P009 has the largest paired difference within the Opus group.

Figure 5.1 plots each participant's two session costs. Under Opus, APM's sessions ran from \$25.38 to \$50.76 against Spec-kit's \$18.75 to \$29.11, while under Sonnet the wider spread was Spec-kit's, from \$7.67 to \$19.46 against APM's \$12.89 to \$16.74.

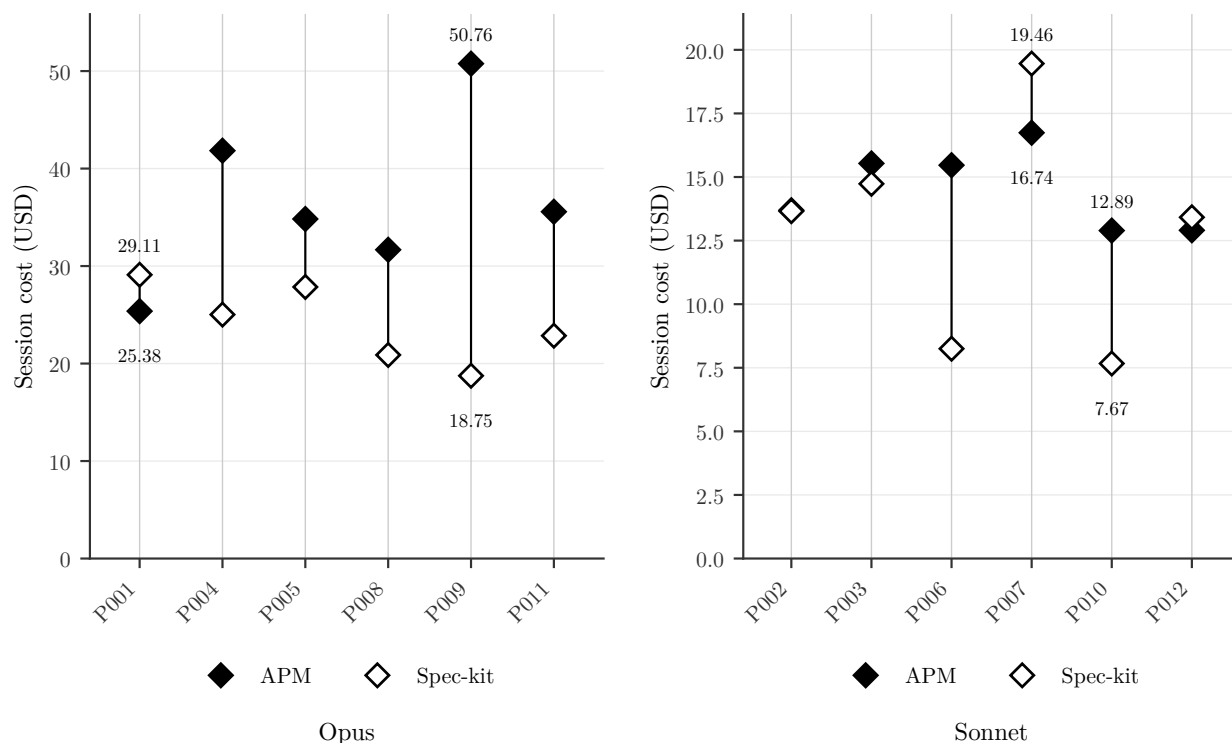


Figure 5.1: Per-participant session cost within each model, Opus on the left and Sonnet on the right. Each participant has a filled APM diamond and an open Spec-kit diamond joined by a line, and each panel carries its own scale. Each framework's cheapest and most expensive session carries its value.

Paired within each participant, the APM session incurred the higher dollar-equivalent cost for nine of the twelve, and all three summaries of the difference are positive. The median and the Hodges-Lehmann estimate fall close together and the mean sits somewhat higher, a sign of a few large values pulling the average up. The Wilcoxon interval excludes zero, though its width leaves the size of the difference imprecise.

The difference behaved differently in the two model groups. Among the participants who worked with Opus, five of the six APM sessions cost more. The difference was large on all three summaries, with the median and the Hodges-Lehmann estimate close together and the mean a little above. Those differences run from \$3.73 less to \$32.01 more. Among the participants who worked with Sonnet, the difference was small on all three summaries and mixed in direction, running from \$2.72 less to \$7.22 more. Those large Opus values pulled the cohort mean up, so the cohort-level cost difference comes mostly from the Opus group.

Only the implementation phase carries a cohort difference of its own.¹ APM's implementation cost was the higher in nine of the twelve pairs, with an interval excluding zero, and it set the direction of the overall difference. Its planning cost was the lower in eight of the twelve, but its interval spans zero, so the phase carries no firm difference. The implementation excess concentrated within the Opus group, where APM cost more in all six pairs, against three of the six under Sonnet. Planning stayed mixed in both, at three of the six pairs under Opus and one of the six under Sonnet.

Table 5.2 gives these differences at session scope and in each phase, for the cohort and within each model group.

¹Session activity outside either framework's workflow phases accounts for under one percent of cohort cost, and its interval spans zero, so it is omitted.

Table 5.2: Paired cost differences by scope, APM minus Spec-kit, in US dollars.

Group	Scope	<i>n</i>	Sign (+/−)	Median	HL	Mean	CI
Cohort	Session	12	9 / 3	+6.10	+6.10	+7.14	[+0.42, +14.14]
	Planning ^a	12	4 / 8	−0.84	−1.27	−1.07	[−3.53, +0.61]
	Implementation ^{b,c}	12	9 / 3	+6.52	+6.52	+8.08	[+2.08, +12.65]
Opus	Session	6	5 / 1	+11.75	+11.89	+12.60	[−3.73, +32.01]
	Planning ^a	6	3 / 3	−1.20	−1.20	−0.51	[−6.64, +9.44]
	Implementation ^c	6	6 / 0	+10.66	+10.66	+12.91	[+2.08, +34.96]
Sonnet	Session	6	4 / 2	+0.42	+1.25	+1.68	[−2.72, +7.22]
	Planning	6	1 / 5	−0.84	−1.10	−1.62	[−6.05, +0.28]
	Implementation ^b	6	3 / 3	+1.37	+2.80	+3.24	[−2.15, +13.26]

^a P004's APM Planner and Manager ran in one conversation, so the Manager's work is recorded under the planning phase, and under the Planner role wherever a measure separates the roles. This makes the cohort planning difference less negative. Without P004 the median, Hodges-Lehmann estimate, and mean are −1.10, −1.77, and −2.02, rather than −0.84, −1.27, and −1.07.

^b P006's Spec-kit implementation work is recorded under planning; see note [c] of Table 5.6. That session's implementation phase therefore shows no cost. This raises the cohort implementation difference. Without P006 the median, Hodges-Lehmann estimate, and mean are +5.99, +5.86, and +7.61. The cohort values are +6.52, +6.52, and +8.08.

^c P009 is the mixed-model cost case described in Table 5.1. It shifts all three cohort implementation row summaries, most visibly the mean. Without P009 the median, Hodges-Lehmann estimate, and mean are +5.99, +5.69, and +5.63. The cohort values are +6.52, +6.52, and +8.08.

Figure 5.2 plots each participant's session difference beside its two phase parts. The implementation difference sits above the planning difference in eight of the twelve pairs.

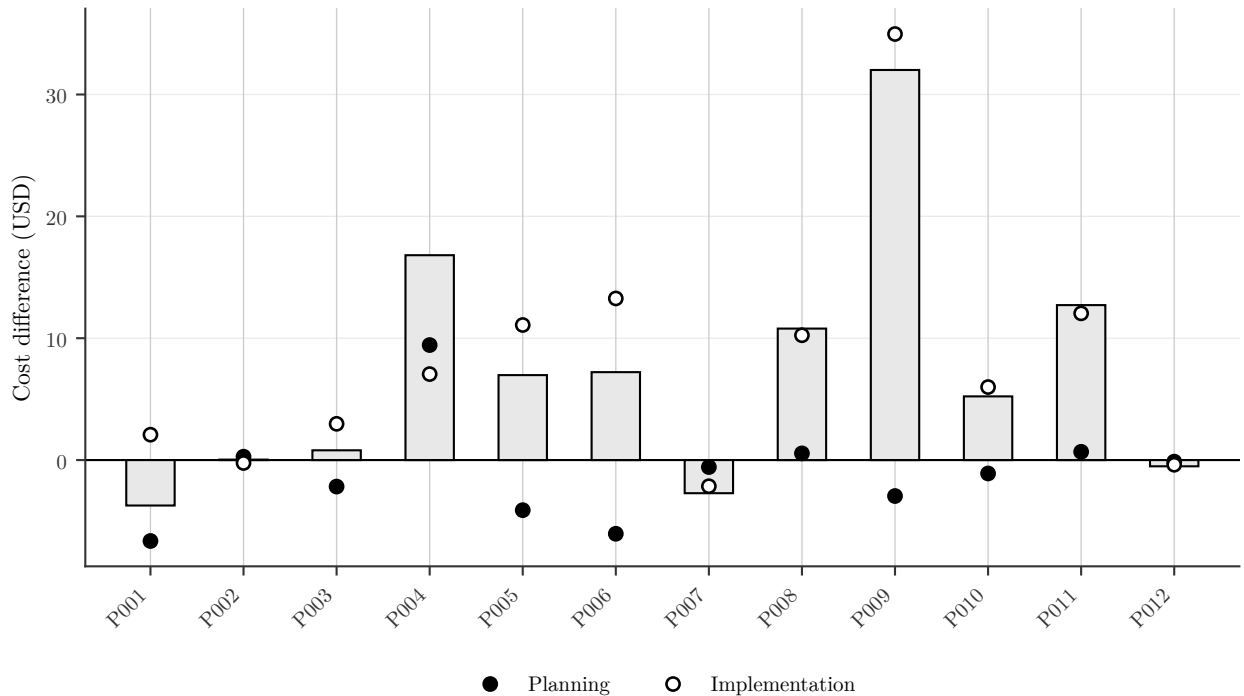


Figure 5.2: Per-participant paired cost difference by scope, APM minus Spec-kit, in US dollars. The bar gives the session difference and the markers the planning and implementation differences within it. The reference line marks no difference.

P004 and P006 have the phase-attribution caveats of notes [a] and [b] of Table 5.2.

Within an APM session, cost divided among the three agent roles in similar proportions under both models, as Table 5.3 details. The Planner and Manager cost comparable amounts, each a fraction of the Worker cost.

Every role cost more under Opus than under Sonnet.

Table 5.3: Cost by agent role within the APM sessions, by model, in US dollars.

Group	Role	Median	HL	Mean
Opus	Planner ^a	5.50	5.85	6.53
Opus	Manager ^a	7.35	7.31	6.97
Opus	Worker	21.76	21.80	22.91
Sonnet	Planner	2.67	2.67	2.66
Sonnet	Manager	2.68	2.68	2.62
Sonnet	Worker	9.54	9.25	9.21

^a P004's merged Planner/Manager conversation, explained in note [a] of Table 5.2, shifts the Opus Planner and Manager row summaries in opposite directions. Without P004 the Opus Planner median, Hodges-Lehmann estimate, and mean are 4.76, 4.76, and 4.84; the cohort values are 5.50, 5.85, and 6.53. The Opus Manager values are 7.86, 7.86, and 8.37 without P004 and 7.35, 7.31, and 6.97 for the cohort.

For both frameworks, cache-read accounted for most cost, followed by output and one-hour writes. Uncached input cost a few cents either way, and five-minute writes were negligible. APM's output and one-hour-write costs exceeded Spec-kit's under both models, while its cache-read cost was higher under Opus and slightly lower under Sonnet, as Table 5.4 reports.

Table 5.4: Per-framework cost by token category, in US dollars, across each framework's six sessions in each model. The negligible uncached-input and five-minute cache-write categories are omitted.

Group	Framework	Category	Median	HL	Mean
Opus	APM	Output	7.99	7.99	8.32
		Cache read	19.80	19.80	19.71
		Cache write (1-hour)	7.84	8.07	8.42
	Spec-kit	Output	4.79	5.00	5.22
		Cache read	14.96	14.96	14.98
		Cache write (1-hour)	3.90	3.71	3.75
Sonnet	APM	Output	4.30	4.30	4.28
		Cache read	6.72	6.72	6.70
		Cache write (1-hour)	3.55	3.44	3.45
	Spec-kit	Output	2.82	2.72	2.72
		Cache read	7.56	7.56	7.78
		Cache write (1-hour)	2.54	2.34	2.27

Output and one-hour cache writes each carried a cohort difference of its own, while cache reading changed direction between the model groups. Table 5.5 gives the paired difference across all five categories, for the cohort and within each model group. APM produced more output than Spec-kit in all but one session and wrote more to the one-hour cache in every session. Each difference excludes zero across the cohort and is larger under Opus than under Sonnet. The cache-read difference was mixed, with its interval including zero. The five-minute cache writes carried a small positive difference.

Table 5.5: Paired cost differences by token category, APM minus Spec-kit, in US dollars.

Group	Category	<i>n</i>	Sign (+/−)	Median	HL	Mean	CI
Cohort	Input	12	7 / 5	+0.00	+0.00	+0.01	[−0.00, +0.01]
	Output	12	11 / 1	+2.24	+2.20	+2.33	[+1.15, +3.40]
	Cache read	12	7 / 5	+1.16	+1.20	+1.82	[−2.33, +6.03]
	Cache write (5-minute)	12	10 / 2	+0.07	+0.07	+0.05	[−0.00, +0.13]
	Cache write (1-hour)	12	12 / 0	+2.67	+2.50	+2.92	[+1.28, +4.65]
Opus	Input	6	5 / 1	+0.00	+0.00	+0.01	[−0.01, +0.06]
	Output	6	5 / 1	+3.28	+3.20	+3.10	[−0.07, +6.12]
	Cache read	6	5 / 1	+3.51	+4.91	+4.73	[−7.30, +17.12]
	Cache write (5-minute)	6	6 / 0	+0.07	+0.08	+0.09	[+0.00, +0.24]
	Cache write (1-hour)	6	6 / 0	+3.72	+4.51	+4.67	[+2.92, +8.53]
Sonnet	Input	6	2 / 4	−0.00	+0.00	+0.00	[−0.00, +0.01]
	Output	6	6 / 0	+1.30	+1.58	+1.56	[+0.66, +2.80]
	Cache read	6	2 / 4	−1.34	−1.34	−1.08	[−5.74, +1.94]
	Cache write (5-minute)	6	4 / 2	+0.07	+0.05	+0.02	[−0.29, +0.14]
	Cache write (1-hour)	6	6 / 0	+1.12	+1.12	+1.18	[+0.20, +2.41]

By session order, all three summaries of the session cost difference were positive in both groups, with four of the seven APM-first pairs and all five APM-second pairs positive, as Table B.1 of Appendix B.1 reports. They were larger and more consistent when participants met APM second. At seven and five participants the intervals are wide and the two order groups cannot be told apart.

Chapter 6 reads these differences further. Section 6.1 traces the planning-phase divergence to how the two frameworks carry out spec-driven planning. Section 6.2.1 reads the implementation cost, showing how working across several agents shapes where the cost falls and why the two model groups diverge.

5.2 Process

The process comparison separates the time each task took from the form and frequency of the user’s recorded interaction with each framework.

5.2.1 Temporal Shape

An APM session lasted a median of about 119 minutes against Spec-kit’s 71, as Table 5.6 reports. APM’s Hodges-Lehmann estimate and mean sat close to its median, while Spec-kit’s mean and Hodges-Lehmann estimate sat above its median, pulled up by a few long sessions. The two phases divided that time differently. APM’s time fell mostly in implementation, at a median of about 77 minutes against Spec-kit’s 37, while the two spent almost the same time planning, APM at about 36 minutes against Spec-kit’s 35.

Table 5.6: Per-framework session and phase durations, in seconds, across each framework's twelve sessions.

Scope	Framework	Median	HL	Mean
Session	APM ^a	7136.1	7140.5	7267.2
	Spec-kit	4274.6	4684.6	4810.1
Planning	APM ^b	2153.4	2179.3	2298.9
	Spec-kit	2076.5	2250.2	2356.8
Implementation	APM	4641.5	4790.6	4865.1
	Spec-kit ^c	2196.2	2228.5	2439.1

^a P010's APM session went about ten minutes past the three-hour cap and is measured to it, so that tail lies outside every summary in this row. The excluded records carry verification runs, version-control work, and framework bookkeeping, and no part of the submitted solution was written in them.

^b P004's merged APM Planner/Manager conversation, explained in note [a] of Table 5.2, raises the APM planning-duration summary. Without P004 the APM planning median, Hodges-Lehmann estimate, and mean are 1850.6, 2129.2, and 2181.7 seconds, rather than 2153.4, 2179.3, and 2298.9.

^c P006's Spec-kit agent carried out the implementation without reaching `/speckit-implement`, which marks the workflow's implementation phase for this study, so all its measurements were allocated to planning. The implementation phase therefore has no duration for that session. This lowers the Spec-kit implementation-duration summary. Without P006 the Spec-kit implementation median, Hodges-Lehmann estimate, and mean are 2210.7, 2283.4, and 2660.8 seconds. The cohort values are 2196.2, 2228.5, and 2439.1 seconds.

APM was the longer session for every participant, at a median difference of about 36 minutes, close to its Hodges-Lehmann estimate, with a confidence interval excluding zero. The session duration divides into turn time, when at least one agent was working, and inter-turn time. Both were higher under APM in eleven of the twelve pairs.

The implementation phase was longer under APM in every pair, with most of the extra duration spent in turn time, and it set the direction of the whole-session difference. The planning duration difference was mixed across participants, and its interval spans zero. Its turn time and inter-turn time divided the same way, both mixed in sign, with intervals that include zero.

Table 5.7 gives the cohort's paired time differences at session and phase scope.

Table 5.7: Paired time differences by scope across the cohort, APM minus Spec-kit, in seconds. The turn time is the active span, and the inter-turn time is the residual to the duration.

Scope	Measure	<i>n</i>	Sign (+/−)	Median	HL	Mean	CI
Session	Duration	12	12 / 0	+2160.7	+2343.6	+2457.0	[+1394.2, +3594.2]
	Turn time	12	11 / 1	+1834.7	+1868.5	+1860.4	[+935.0, +2845.0]
	Inter-turn idle	12	11 / 1	+674.4	+650.1	+596.7	[+93.5, +1201.8]
Planning	Duration	12	5 / 7	−258.7	−155.0	−57.9	[−635.7, +467.9]
	Turn time	12	5 / 7	−163.6	−165.1	−141.9	[−668.4, +347.0]
	Inter-turn idle	12	7 / 5	+98.6	+118.9	+84.0	[−116.0, +387.0]
Implementation	Duration	12	12 / 0	+1820.6	+2153.2	+2426.0	[+1097.1, +3765.3]
	Turn time	12	11 / 1	+1498.6	+1712.1	+1970.2	[+929.7, +3127.1]
	Inter-turn idle	12	9 / 3	+247.9	+337.5	+455.9	[−39.8, +949.2]

The same session and phase patterns held within both model groups; Table B.2 of Appendix B.2 gives the model-specific estimates.

APM was the longer session in all seven pairs where participants met it first and all five where they met it second, as the duration rows of Table B.3 in Appendix B.2 report. The three summaries are comparable

across the two groups; at seven and five participants in each order the intervals are wide and the two orders cannot be told apart. The duration pattern leaves session order an unlikely account of APM's longer sessions.

Within the turn time APM could have two or more of its agents working at once, but participants used concurrent work sparingly. In five sessions it made up less than 2% of turn time; across the cohort it remained below a sixth of turn time in all but one session, where it accounted for about a quarter. Table B.4 in Appendix B.2 gives the per-session split.

Figure 5.3 shows each participant's two sessions as paired bars, with the filled lower part of each the turn time and the open upper part the inter-turn time. APM's bars stand taller across the cohort, its filled turn-time segment larger than its Spec-kit pair's for nearly every participant.

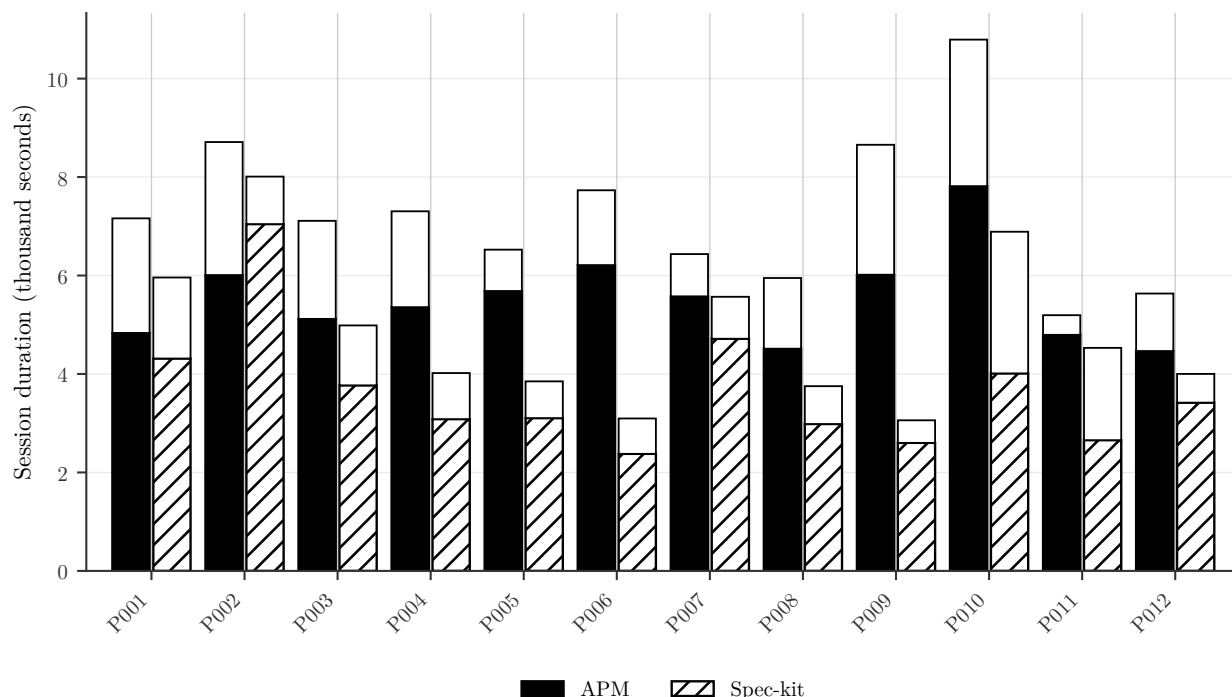


Figure 5.3: Per-participant session temporal shape. Each participant has two bars, APM and Spec-kit; the filled lower segment is the turn time, the open upper segment the inter-turn time, and the full height the session duration.

5.2.2 Interaction Shape

An APM session had a median of about 22 turn prompts against Spec-kit's 13, as Table 5.8 reports, with the Hodges-Lehmann estimate and mean close to each median. That total was not shared evenly between the two phases. In planning, the median counts were about 10 under both frameworks, while in implementation APM had about 11 against Spec-kit's 3.

Table 5.8: Per-framework turn-prompt counts at session and phase scope, across each framework's twelve sessions.

Scope	Framework	Median	HL	Mean
Session	APM	21.5	22.0	22.2
	Spec-kit	13.0	12.8	13.2
Planning	APM ^a	9.5	9.5	9.7
	Spec-kit	10.0	9.5	9.8
Implementation	APM	10.5	11.0	12.2
	Spec-kit ^{b,c}	3.0	3.5	3.3

^a P004's merged Planner/Manager conversation, explained in note [a] of Table 5.2, raises the APM planning turn-prompt count. Without P004 the APM planning turn-prompt count median, Hodges-Lehmann estimate, and mean are 9.0, 9.0, and 8.7 turn prompts. The cohort values are 9.5, 9.5, and 9.7.

^b P006's Spec-kit session opens no implementation turn prompt, for the reason given in Table 5.6 note [c]. This lowers the Spec-kit implementation turn-prompt count summary. Without P006 the Spec-kit implementation median, Hodges-Lehmann estimate, and mean are 4.0, 3.5, and 3.6 turn prompts. The cohort values are 3.0, 3.5, and 3.3.

^c P007 and P009 started implementation within a planning turn of their Spec-kit sessions, so that turn remains planning and each had one subsequent implementation turn.

Paired within each participant, the added turns under APM fell almost entirely in implementation, as Table 5.9 reports. The session difference was a median of eight more turns, positive in eleven of the twelve pairs, with its interval excluding zero. Planning was level, with five pairs each way and an interval spanning zero. Implementation added a median of eight turns for APM, with a positive difference in eleven of the twelve pairs, setting the direction of the whole-session difference.

Table 5.9: Paired turn-prompt count differences, APM minus Spec-kit.

Scope	<i>n</i>	Sign (+/-/=)	Median	HL	Mean	CI
Session	12	11 / 0 / 1	+8.0	+8.5	+8.9	[+4.5, +14.0]
Planning	12	5 / 5 / 2	+0.0	-0.2	-0.2	[-3.5, +2.0]
Implementation	12	11 / 0 / 1	+8.0	+8.2	+8.8	[+5.0, +13.0]

The same pattern appeared under both models, with the per-model split across the scopes given in Table B.5 of Appendix B.2.

In planning, APM's turn prompts were mostly short or standard text, at mean character counts of 19 and 146 respectively. Spec-kit's planning instead centred on the framework's commands, with few standard prompts from the user. That division held steady across APM's participants, while Spec-kit's varied widely from one session to the next.

In implementation the pattern reversed for APM, and the protocol commands that drive its workflow made up most of its turn prompts. Spec-kit's implementation carried few turn prompts of any kind, and one session opened none. Both its division and its character counts there rest on those few prompts, too sparse to read as typical.

Table 5.10 gives each bucket's share and mean character count, and Figure 5.4 plots each session's shares.

Table 5.10: Turn-prompt bucket shares and mean character counts by phase and framework.

Phase	Framework	Bucket	Bucket share			Magnitude (chars)		
			Median	HL	Mean	Median	HL	Mean
Planning	APM	Protocol	0.11	0.12	0.18	150	150	147
		Short	0.50	0.50	0.47	20	20	19
		Standard	0.38	0.37	0.35	156	139	146
	Spec-kit	Protocol	0.56	0.56	0.54	106	109	114
		Short	0.26	0.32	0.34	20	20	20
		Standard	0.10	0.08	0.12	81	82	85
Implementation	APM	Protocol	0.73	0.72	0.71	124	125	125
		Short	0.13	0.14	0.14	18	20	20
		Standard ^b	0.10	0.11	0.15	79	102	184
	Spec-kit	Protocol	0.33	0.25	0.32	100	101	125
		Short	0.20	0.32	0.34	32	33	33
		Standard	0.07	0.17	0.26	74	76	86

^a Each bucket is summarised independently across the sessions, so neither the three medians nor the three Hodges-Lehmann estimates need sum to one. The three mean shares sum to one unless a session has no turn prompt in the phase and contributes three zeros.

^b P002 has the largest APM implementation standard-prompt character count in the cohort by a substantial margin, from a single prompt in which the participant manually pasted handoff content. This raises all three row summaries. Without P002 the median, Hodges-Lehmann estimate, and mean are 69, 83.5, and 96 characters. The cohort values are 79, 102, and 184.

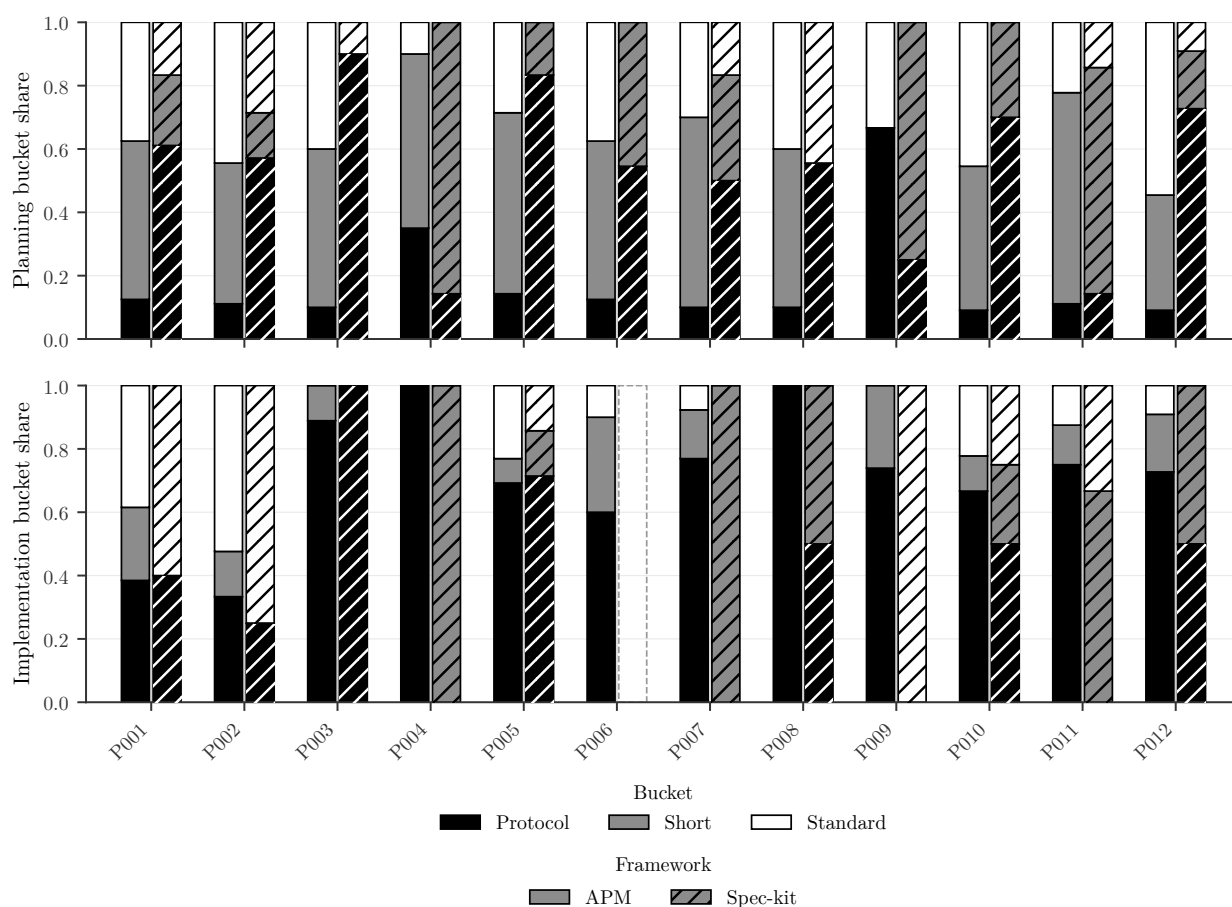


Figure 5.4: Per-session turn-prompt bucket shares by phase. A dashed outline marks a session that carried no turn prompt in the phase and so has no shares.

The prompt composition held under both models. The visible difference was in Spec-kit’s planning, where the protocol share was lower under Opus than under Sonnet. The character-count summaries were otherwise aligned across models, apart from APM’s Sonnet implementation standard bucket, whose mean was higher. Tables B.6 and B.7 of Appendix B.2 give the shares and character counts by model.

Within APM the turn prompts also split by agent role, as Table 5.11 reports. The Planner’s turns were those of the planning phase, while the Manager and the Workers divided the implementation phase between them, at medians of 6.5 and 5.0 turn prompts. The composition of those turns by role is given in Table B.8 of Appendix B.2. The Workers were addressed almost entirely through protocol commands, while the Manager’s prompts, mostly protocol as well, also included some short and standard text.

Table 5.11: Per-role turn-prompt counts across the twelve APM sessions. Descriptive and APM only, since Spec-kit has no agent roles to separate.

Role	Median	HL	Mean
Planner	9.5	9.5	9.7
Manager	6.5	6.5	6.2
Worker	5.0	5.0	6.0

Steering was sparse under both frameworks and confined to a minority of sessions. Seven of the twelve participants steered at all under APM and four under Spec-kit, most of them once or twice. No session carried more than four steers. Across the cohort that came to thirteen steers under APM, of which three were delivered to the working agent, and six under Spec-kit, of which none were delivered. Table B.9 of Appendix B.2 gives each session’s counts.

Session order did not reverse the result. The higher turn-prompt count under APM held whether participants met it first or second, as the turn-prompt rows of Table B.3 in Appendix B.2 report.

Chapter 6 interprets the added time and recorded interaction in light of the frameworks’ workflows. Section 6.1 examines planning behaviour, and Section 6.2.2 examines the demands placed on the user during implementation.

5.3 Functional

Most sessions scored near the rubric’s hundred-point ceiling. The cohort medians were close, at about 96 points under APM and 97 under Spec-kit. Lower scores pulled the mean below the median for both frameworks, more so for Spec-kit, whose three summaries lay further apart.

The model split changes that reading. Under Opus, all three summaries for both frameworks remained near the ceiling. Under Sonnet, APM’s median sat well above its Hodges-Lehmann estimate and mean because two sessions scored 34.0 points while the other four scored from 91.0 to 98.0. Spec-kit split into three scores from 29.6 to 40.0 and three from 84.2 to 97.0, placing its median between those clusters and much closer to its other two summaries. Table 5.12 gives the cohort and model-group summaries, and Table B.10 of Appendix B.3 lists the individual sessions.

Table 5.12: Per-framework total score, in points, for the cohort and within each model group.

Group	Framework	Median	HL	Mean
Cohort	APM	95.7	95.5	86.2
	Spec-kit	96.5	84.3	78.8
Opus	APM	99.0	97.5	97.9
	Spec-kit	98.5	98.5	94.2
Sonnet	APM	92.0	66.0	74.4
	Spec-kit	62.1	63.3	63.5

Paired within each participant, the scores were close. APM scored higher than Spec-kit for six participants and Spec-kit for only two, with four pairs tied. The cohort median difference was about half a point, its Hodges-Lehmann estimate about 2 points, and its mean about 7 points. These summaries point to a small positive cohort difference, but the cohort interval includes zero.

The model-group split locates the cohort's positive score pattern under Sonnet. Under Opus, both frameworks' summary values remained near the ceiling, and three paired comparisons tied; the paired median and Hodges-Lehmann estimate were zero, while one +23.4-point APM difference kept the mean positive. Under Sonnet, APM's values were higher on all three summaries, and four pairs favoured APM, one favoured Spec-kit, and one was tied.

Table 5.13 gives the cohort and model-group differences.

Table 5.13: Paired total-score difference by participant group, APM minus Spec-kit, in points.

Group	<i>n</i>	Sign (+/-/=)	Median	HL	Mean	CI
Cohort	12	6 / 2 / 4	+0.5	+2.2	+7.3	[-0.8, +17.9]
Opus	6	2 / 1 / 3	+0.0	+0.0	+3.7	[-2.0, +23.4]
Sonnet	6	4 / 1 / 1	+3.2	+4.4	+10.9	[-6.0, +53.0]

Figure 5.5 plots this contrast. Apart from one wide gap, the Opus pairs cluster at the same score, whereas the Sonnet pairs have a broader range of gaps, mostly favouring APM.

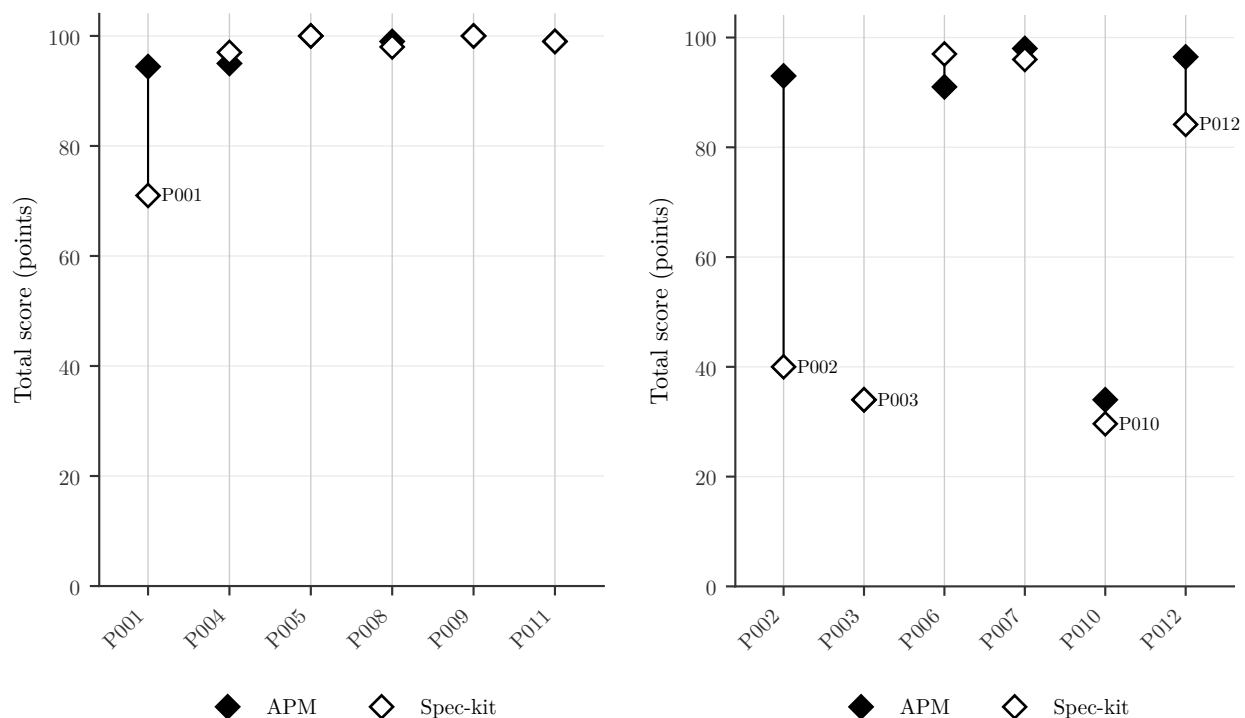


Figure 5.5: Per-participant total score within each model, Opus on the left and Sonnet on the right. Each participant has a filled APM diamond and an open Spec-kit diamond joined by a line.

Across the cohort, both task sections had a zero median difference but positive Hodges-Lehmann estimates and means. This pattern also concentrated under Sonnet, where both sections had positive medians, Hodges-Lehmann estimates, and means. Under Opus, both section medians were zero, with a zero Hodges-Lehmann estimate for the Python extension and only +0.5 points for the C replica. Table 5.14 gives the paired section differences for the cohort and each model group.

Table 5.14: Paired per-section score difference by participant group, APM minus Spec-kit, in points.

Group	Section	<i>n</i>	Sign (+/−/=)	Median	HL	Mean	CI
Cohort	Python extension	12	4 / 2 / 6	+0.0	+2.0	+3.4	[−0.7, +8.7]
	C replica	12	5 / 2 / 5	+0.0	+1.0	+3.9	[−2.2, +11.5]
Opus	Python extension	6	1 / 1 / 4	+0.0	+0.0	+1.1	[−2.0, +8.7]
	C replica	6	2 / 0 / 4	+0.0	+0.5	+2.6	[+0.0, +14.7]
Sonnet	Python extension	6	3 / 1 / 2	+2.0	+6.3	+5.7	[−1.5, +18.0]
	C replica	6	3 / 2 / 1	+1.0	+1.9	+5.2	[−9.6, +35.0]

Partial credit was identical in eight pairs and differed by no more than about six points in the other four. Table B.10 of Appendix B.3 gives the partial part of every session.

The median, Hodges-Lehmann estimate, and mean were non-negative in both order groups, as Table B.11 of Appendix B.3 reports. At seven and five participants the intervals are wide and the groups cannot be told apart. The small positive difference reads as following the framework rather than the session order.

Chapter 6 takes up these score outcomes. Section 6.1 sets the size and kind of the sessions that fell short against how each framework planned. Section 6.3.1 reads the low scores under a diagnostic re-grade, Section 6.3.2 weighs the score difference against what each framework cost, and Section 6.6 reads the same scores for a practice effect.

5.4 Perception

Some of the eight rating items divided the cohort sharply and others not at all. Table 5.15 reports, for each item, how many of the twelve participants rated it higher, lower, or the same under APM, alongside the direction that is the better outcome for that item.

Table 5.15: Paired direction of each questionnaire rating across the twelve participants, APM against Spec-kit. The second column gives which direction is the better outcome for the item.

Rating	Better when	Sign (+/-/=)
Familiarity	context	3 / 2 / 7
Confidence	higher	9 / 0 / 3
Effectiveness	higher	11 / 0 / 1
Mental demand	lower	6 / 2 / 4
Frustration	lower	2 / 8 / 2
Effort	lower	7 / 4 / 1
Verification effort	lower	4 / 3 / 5
Steering effort	lower	3 / 4 / 5

Prior familiarity was low under both frameworks, with the ratings massed at the bottom of the scale for each.

Participants rated APM the more confidence-inspiring framework in nine of the twelve pairs and the more effective in eleven, with no participant rating Spec-kit higher on either item. Under APM the effectiveness ratings clustered at the top of the scale, with every session at four or five and eight of the twelve at the maximum. The Spec-kit ratings formed two clusters, at two and four, with only two at three. The confidence ratings separated almost as far, with ten of APM's twelve at four or five against nine of Spec-kit's at two or three. Table 5.16 gives the per-framework distribution behind these readings for every rating.

Table 5.16: Distribution of each questionnaire rating across each framework's twelve sessions, the count rating the item at each point of the one-to-five scale, where one is the low anchor and five the high anchor.

Rating	Framework	1	2	3	4	5
Familiarity	APM	7	1	2	2	0
	Spec-kit	8	2	2	0	0
Confidence	APM	0	0	2	7	3
	Spec-kit	1	5	4	1	1
Effectiveness	APM	0	0	0	4	8
	Spec-kit	0	5	2	5	0
Mental demand	APM	2	4	4	2	0
	Spec-kit	2	4	5	1	0
Frustration	APM	7	3	1	1	0
	Spec-kit	1	3	4	3	1
Effort	APM	1	4	6	1	0
	Spec-kit	6	2	2	1	1
Verification effort	APM	2	5	4	0	1
	Spec-kit	2	6	2	1	1
Steering effort	APM	1	5	4	2	0
	Spec-kit	1	6	3	2	0

Mental demand leaned towards Spec-kit, with six pairs favouring it, two favouring APM, and four tied. APM was the less frustrating framework in eight of the twelve pairs, with its frustration ratings low and Spec-kit's spread upward across the scale. Set against this, participants reported investing more effort under APM in seven of the twelve pairs and more under Spec-kit in four. Spec-kit's own effort ratings massed low, with half its sessions at the minimum. The effort of checking what the agent produced and the effort of instructing and steering it showed no consistent direction.

The comparison question asked after the second session, grouped by the framework that session used, gave a less decided verdict, as Table B.12 of Appendix B.4 reports. Among the five whose second framework was APM the verdicts split evenly between easier and harder, with one rating it the same.² Among the six whose second framework was Spec-kit four leaned harder and the other two were divided.

Nineteen of the twenty-four sessions finished comfortably within the three-hour cap and five finished near it, nine against three under APM and ten against two under Spec-kit. No participant reported hitting the cap without completing or stopping early, leaving the other two states in Table B.13 of Appendix B.4 empty.

Section 6.4 reads these ratings against the measured behaviour behind them, and Section 6.5 weighs them alongside the study's other findings.

²P012 is excluded because their response cannot be assigned to a realised-order group; see footnote 1 in Section 4.4.3.

6. DISCUSSION

6.1 Two Realisations of Spec-Driven Planning

Both frameworks plan and create specifications before they build, and both explore the codebase and launch subagents for the heavier research. They differ in who makes the planning decisions. Spec-kit has the agent resolve the open questions itself, making informed guesses based on the model's training data. It fills a fixed set of template documents from what it finds and knows, and it turns to the user only where its own checks leave a question open, offering recommended choices to confirm. Its planning is therefore sensitive to the model's capability and to how the autonomous exploration plays out. APM keeps the user as the authority. Its Planner goes through a series of question rounds with the user and brings back what it understood for approval before any document is written. It is directed to ask rather than assume, and it produces a few short planning documents that generalise across project types. The task brief the participants worked from was already detailed, yet under it APM still asked many targeted questions grounded in its research, while Spec-kit, with the same brief, asked few.

Both gate their planning and ask the user for approval. APM's gates are points where the user approves each planning document, while Spec-kit's are the agent's own validation checks and a holistic presentation to the user at the end.

The planning turns differed in kind accordingly, as the planning buckets of Table 5.10 report. Spec-kit, planning mostly on its own, turned to the user only to confirm its recommended choices. APM brought the user's decisions into the planning, requiring more substantive interactions.

That difference in authority does not show in the size of the planning phase, which carries no firm difference in cost, duration, or turn-prompt count, as the planning rows of Table 5.2, Table 5.7, and Table 5.9 report. What differs is where the planning work took place, which Table 6.1 details. Under Spec-kit the main agent did that work itself, issuing more requests to the model in eleven of the twelve pairs and making more tool calls in the same eleven. Under APM part of it was offloaded to subagents, which made more tool calls than Spec-kit's in ten of the twelve pairs.

Table 6.1: Paired planning-phase behaviour differences, APM minus Spec-kit.

Measure	<i>n</i>	Sign (+/−)	Median	HL	Mean	CI
Requests ^a	12	1 / 11	−33.0	−31.8	−29.7	[−44.5, −18.5]
Main tool calls ^a	12	1 / 11	−29.5	−30.2	−32.6	[−54.5, −13.0]
Subagent requests ^b	11	8 / 3	+26.0	+25.0	+25.8	[+5.0, +48.0]
Subagent tool calls	12	10 / 2	+33.0	+31.5	+30.8	[+7.0, +51.5]

^a P004's merged Planner/Manager conversation, explained in note [a] of Table 5.2, puts the Manager's requests in the planning phase. P004 is the single pair in which APM's main agent was the busier of the two. Without P004 the request median, Hodges-Lehmann estimate, and mean are −35.0, −34.0, and −37.6. The cohort values are −33.0, −31.8, and −29.7. The main tool-call values are −30.0, −34.2, and −39.8 without P004 and −29.5, −30.2, and −32.6 for the cohort.

^b P001's APM session launched subagents whose transcripts the submission did not include, so the pair is excluded from this row.

Five sessions across the cohort fell well short. Two scored below fifty under APM and three under Spec-kit, as Table B.10 details, so the shortfall was not one framework's alone. They did not all fail in the same way. The re-grade of Section 6.3.1 lifts a fault that occurred under both frameworks, leaving two Spec-kit sessions below fifty.

In Spec-kit's largely autonomous planning, a step that settles an open question without the user can do so wrongly, and the build then rests on that decision. Its instructions set a ceiling on how many questions it may raise and have the agent reach a documented assumption wherever a reasonable default exists. The sparse

questioning observed is what the framework instructs. APM puts its open questions to the user in the Planner’s question rounds and holds each planning document at a review gate. A wrong assumption can therefore meet the user before the build inherits it. The two remaining low-score sessions align with that difference in planning authority, but they do not establish that it produced them.

6.2 Coordinating the Implementation

Both frameworks build against their planning documents and revise them as findings require. Spec-kit carries the implementation out in the same agent that planned it. In APM, a Manager and its Workers carry it out across several agents and coordinate through the user. Those several agents shape what the implementation costs, and their coordination shapes what it demands of the user.

6.2.1 The Cost of Several Agents

Working across several agents changes where the context accumulation described in Section 2.2 occurs. Each APM conversation accumulates context just like Spec-kit’s, but handles only part of the work. At session scope, however, their costs add together: each builds its own cache and produces its own output. APM exchanges concentrated accumulation in one conversation for the aggregate cost of several.

The observed balance differed with each model’s context window limit and the resets that interrupted accumulation. Claude Code is expected to compact automatically at an auto-compaction threshold below that limit, and the user may compact manually. APM also supports Handoff, which opens a new conversation and context window, as Section 3.5 describes. We refer to all three transitions as resets. After each reset, context begins from a smaller starting point, accumulates, and enters the cache again.

Opus 4.7 had a one-million-token context window limit, large enough to hold the task from start to finish. Sonnet 4.6 had a two-hundred-thousand-token limit in Claude Code’s standard configuration [56], and conversations often approached its auto-compaction threshold.

Figures 6.1 and 6.2 plot the peak of each context window. For example, a conversation that compacts twice uses three consecutive context windows and contributes three peaks. The highest markers are Spec-kit conversations that grew long without a reset. Most APM peaks form a low cluster, but a few APM markers reach the high ranges, one even passing the Spec-kit conversation it is paired with. These are all APM Workers. They accumulate the most context of any agent role in an APM session, and account for the largest part of what the session cost, as Table 5.3 details.

No conversation reset under Opus. APM’s several conversations together produced more output and cache writes than Spec-kit’s one, while its cache-read summaries were also higher despite one pair in the other direction, as the Opus rows of Table 5.5 report. Figure 6.1 shows the pattern behind those differences: Spec-kit has a few lone high markers, while APM has a wider spread reaching towards them. Without any reset to interrupt accumulation, distributing the work did not offset the aggregate cost of APM’s several conversations.

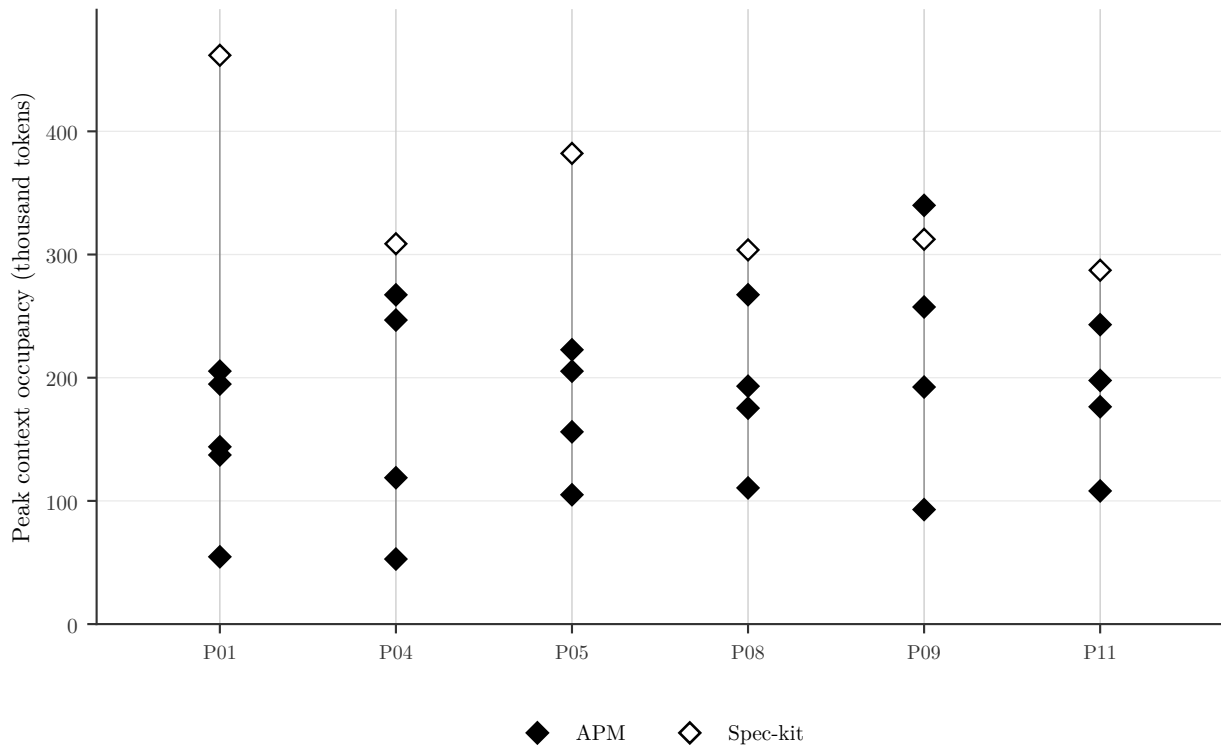


Figure 6.1: Peaks of context windows per session, in thousands of tokens, across Opus model group participants.

Under Sonnet, five of the six Spec-kit conversations compacted, three more than once, as Table 6.2 reports. Four APM sessions compacted automatically. The other two used Handoff instead, so every APM session reset once. Each APM reset occurred in the Worker assigned the C part of the task, the heaviest work in the build.

Table 6.2: Compaction incidence by model and framework, the number of sessions that compacted at least once and the number that compacted more than once.

Model	Framework	Sessions	Compacted	More than once
Opus	APM	6	0	0
	Spec-kit	6	0	0
Sonnet	APM	6	4	0
	Spec-kit	6	5	3

Spec-kit's single conversation moved through consecutive context windows as it compacted. Before each reset, it grew close to the auto-compaction threshold, so later calls reread a large accumulated cache. Its cache-read cost exceeded APM's, reversing the Opus group's direction and offsetting APM's output and one-hour cache-write excess, as the Sonnet rows of Table 5.5 report.

APM instead kept most of its conversations' context windows smaller, so their calls reread less accumulated context. Figure 6.2 shows the difference: Spec-kit contributes repeated peaks near the auto-compaction threshold within individual sessions, while most APM context windows peak well below it.

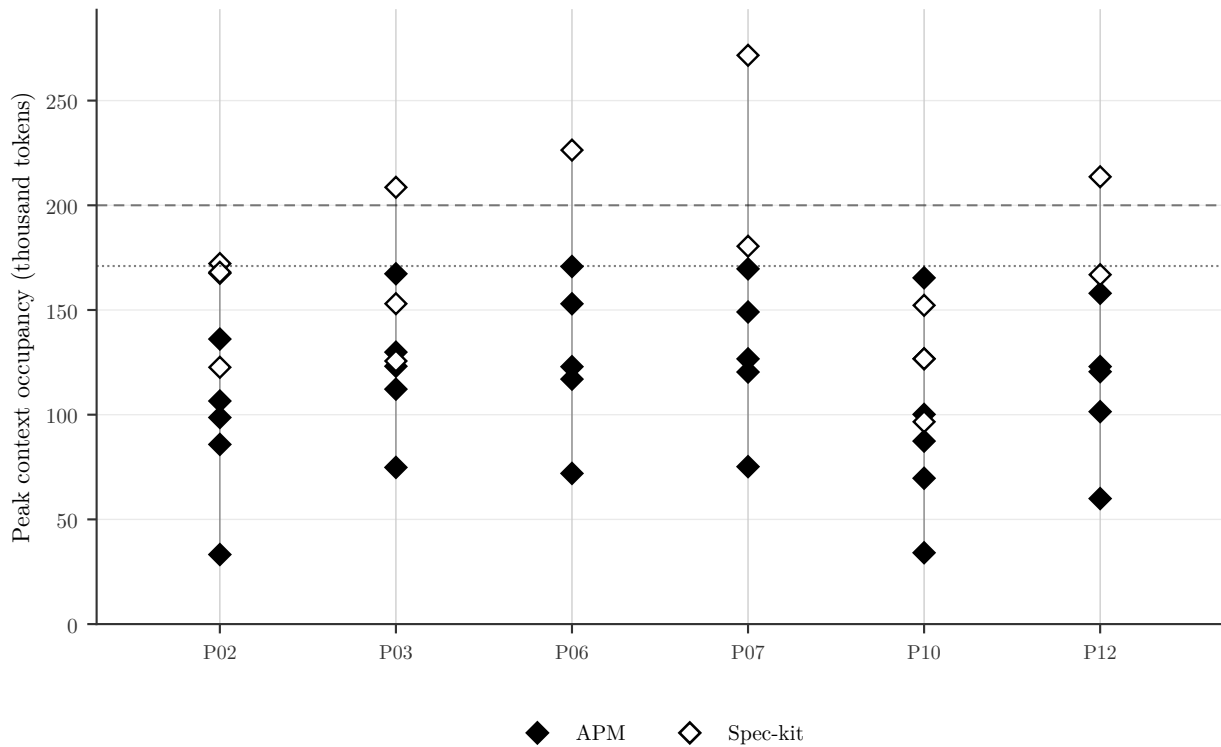


Figure 6.2: Peaks of context windows per session, in thousands of tokens, across Sonnet model group participants. Reference lines mark the 171k observed automatic-compaction bound (dotted) and the two-hundred-thousand-token context window limit (dashed).

A platform issue complicates the Sonnet reading. In four Spec-kit sessions, the conversation continued beyond the model's context window limit without automatic compaction, and Claude Code's context window consumption indicator passed full. These sessions form the highest points of Figure 6.2 and appear to have been served by the model's one-million-token variant, which the platform offered separately but did not identify. The additional accumulation therefore plausibly lifted Spec-kit's cache reads.

Section 6.7 prices that excess and treats the anomaly as a threat to validity. Removing it leaves every Sonnet pair's direction and the group's sign count unchanged, as note [c] of Table 6.7 reports.

The cost result therefore appears to depend on how much of a task fits within the model's context window limit. Under Opus, the whole task fit and the aggregate cost of APM's several conversations dominated. Under Sonnet, the same task repeatedly pressed Spec-kit's conversation towards the auto-compaction threshold, bringing the two frameworks' costs close. A larger task that similarly pushes an Opus conversation towards its auto-compaction threshold could plausibly narrow APM's cost premium. On such tasks, however, users might also try to limit context accumulation and its cost through manual compaction, making participant-controlled reset timing another influence on cost.

6.2.2 The Demands on the User

APM routes planning and implementation through the user, while Spec-kit leaves more of both to its agent. Spec-kit's planning autonomy extends into implementation as it works through the task list largely on its own.

That difference appears in the user's input. Table 5.8 reports far more turn prompts under APM than under Spec-kit. In implementation, Table 5.10 reports that most of APM's prompts are protocol. These include the exchanges between Manager and Worker conversations that APM places in the user's hands, one for each dispatched Task or batch.

Standard prompts are the user's own non-command messages, which may carry instructions or other content directing the agent. They are few under both frameworks. Table B.8 in Appendix B.2 reports that APM's few standard prompts reach the Manager rather than the Workers. Together, the turn-prompt and role data show that participants passed through APM's coordination points, though prompt counts alone cannot tell how much control they exercised through them.

Spec-kit asks much less. Its few turns are the protocol command that launches the implementation and an occasional short or standard follow-up, too few to read as a settled composition. The agent proceeds largely on its own, and the user has little to do until the build is done. This concentrates review into a single accumulated diff.

Each turn prompt opens an interval of agent work. In implementation, APM added a median of eight prompts, and its turn time was higher in eleven of the twelve pairs. That turn time made up most of the phase's duration difference. Tables 5.9 and 5.7 report the additional interaction and time falling in the phase where APM's user-mediated coordination operates.

APM's several-agent implementation plausibly contributes to its larger model request count. APM issued more requests than Spec-kit in every one of the twelve pairs, and its main agents made more tool calls in all twelve as well, as Table 6.3 reports. Both directions reverse the planning phase, where Spec-kit's main agent led on the two measures. Each agent also produces its own output, which is part of why APM's implementation cost the more of the two, as Section 6.2.1 explains.

Table 6.3: Paired implementation-phase behaviour differences, APM minus Spec-kit.

Measure	<i>n</i>	Sign (+/−)	Median	HL	Mean	CI
Requests	12	12 / 0	+97.0	+113.0	+121.0	[+67.0, +185.0]
Main tool calls	12	12 / 0	+135.5	+151.2	+156.6	[+103.5, +212.5]

APM's implementation involved more activity on both sides of the conversation: its agents made more requests and tool calls, and its users interacted with them more often.

6.3 Behind the Functional Score

The total score is the study's one measure of what each session produced. A grader that judges only behaviour can miss sound work, and the score read alone says nothing about what the work cost.

6.3.1 Grading with a Black Box

The grader judges a submission only by the task-required behaviour of the running program, never its source, as Section 4.5.3 describes. A single fault that blocks the program's responses can therefore be recorded as near-total failure, even where the rest of the implementation may be sound.

The C part of the task required every participant to implement a JSON parser and route each MCP request to its requested operation. MCP carries requests as JSON-RPC objects [48], and the JSON standard treats whitespace between fields as insignificant [57]. Three Sonnet implementations shared one defect. Their parsers accepted a compact request such as `{"id": 1}`, but not the ordinary spaced form `{"id": 1}`. A conformant parser must accept both forms; failing to handle whitespace is an incorrect implementation of the task.

Each scored 34.0 points, with the Python part complete and almost all of the C part lost. The grader's ordinary JSON requests elicited no response, so it could not exercise the affected C programs' behaviour at all. When

re-graded with compact requests, the three sessions received diagnostic scores from 93.9 to 100.0. Figure 6.3 plots the full Sonnet group, and Table 6.4 gives the three changed scores.

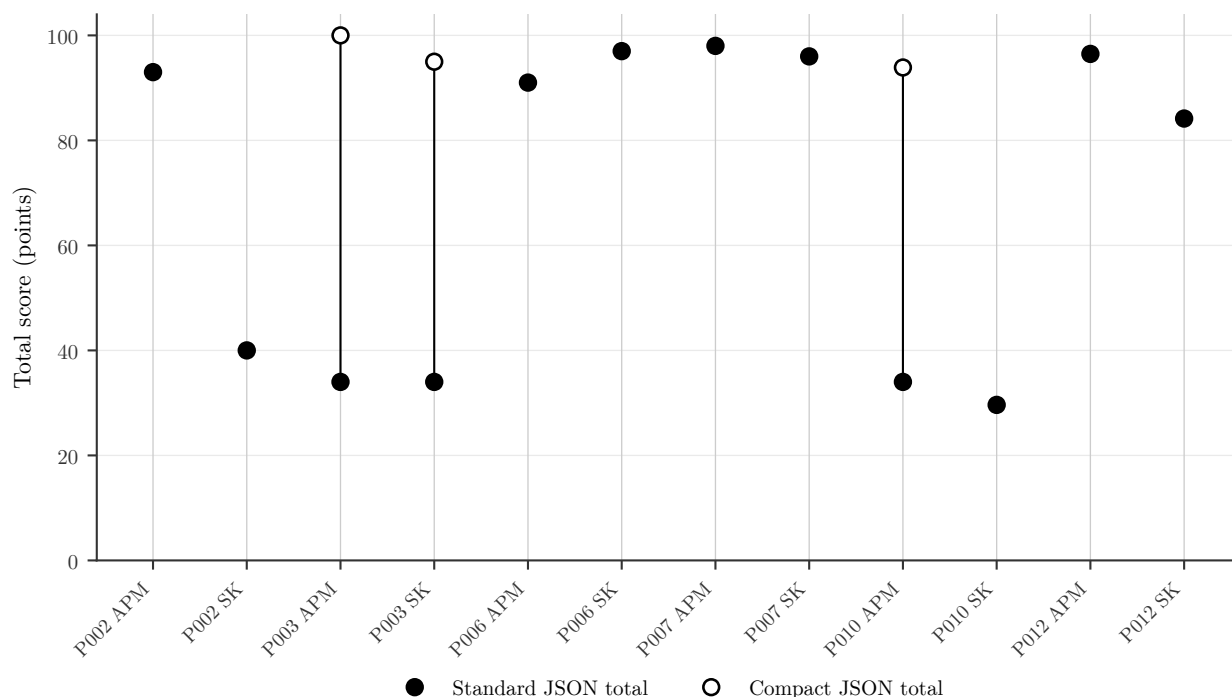


Figure 6.3: Each Sonnet session’s total score on the grader’s standard JSON input, filled, and its diagnostic score on the compact JSON input, open, joined where they differ. The Opus sessions do not differ and are omitted.

Table 6.4: Compact-input diagnostic scores for the sessions whose score changed. These diagnostic scores are not part of the framework comparison.

Session	Standard total score	Compact-input diagnostic score
P003 S1 (APM)	34.0	100.0
P003 S2 (Spec-kit)	34.0	95.0
P010 S2 (APM)	34.0	93.9

The re-grade recovers three of the five sessions below fifty, both APM sessions and one Spec-kit session. The two scores it does not lift remain indeterminate. They may conceal sound work beyond the re-grade’s reach, or they may be genuine shortfalls. The three recovered sessions belong to two participant pairs. In one, both framework sessions carried the defect, so compact input moves the APM-minus-Spec-kit difference from 0.0 to +5.0 points. In the other, only the APM session was affected, moving the difference from +4.4 to +64.3 points. Across all six Sonnet pairs, the compact-input diagnostic favours APM in five. Its median difference is about +8.7 points, while the Hodges-Lehmann estimate and mean are +23.5 and +21.8, raised by the two large gaps, as Table 6.5 reports.

Table 6.5: Paired compact-input diagnostic total-score difference within the Sonnet group, APM minus Spec-kit. This diagnostic does not replace the framework comparison.

Group	n	Sign (+/–)	Median	HL	Mean	CI
Sonnet	6	5 / 1	+8.7	+23.5	+21.8	[−6.0, +64.3]

This diagnostic re-grade helps interpret some low scores as hiding otherwise sound implementations and gives APM the more favourable Sonnet pattern. The standard total scores remain the basis of the framework comparison because the re-grade tests only this fault class.

6.3.2 Cost versus Score

Across the cohort, APM cost more, while its total-score median, Hodges-Lehmann estimate, and mean were positive. The score interval includes zero, so the paired comparison carries no firm difference, as Sections 5.1 and 5.3 report. Figure 6.4 plots the paired values together. Most participants had an APM cost premium with a score gap at or near zero. Under Opus, the larger premium came with both sessions close to the score ceiling. Under Sonnet, costs were close and APM's paired score summaries were more positive.

The same plot shows three participant pairs that depart from that pattern. In each, APM scored clearly higher while costing about the same or less. These are individual observations and do not change the cohort reading.

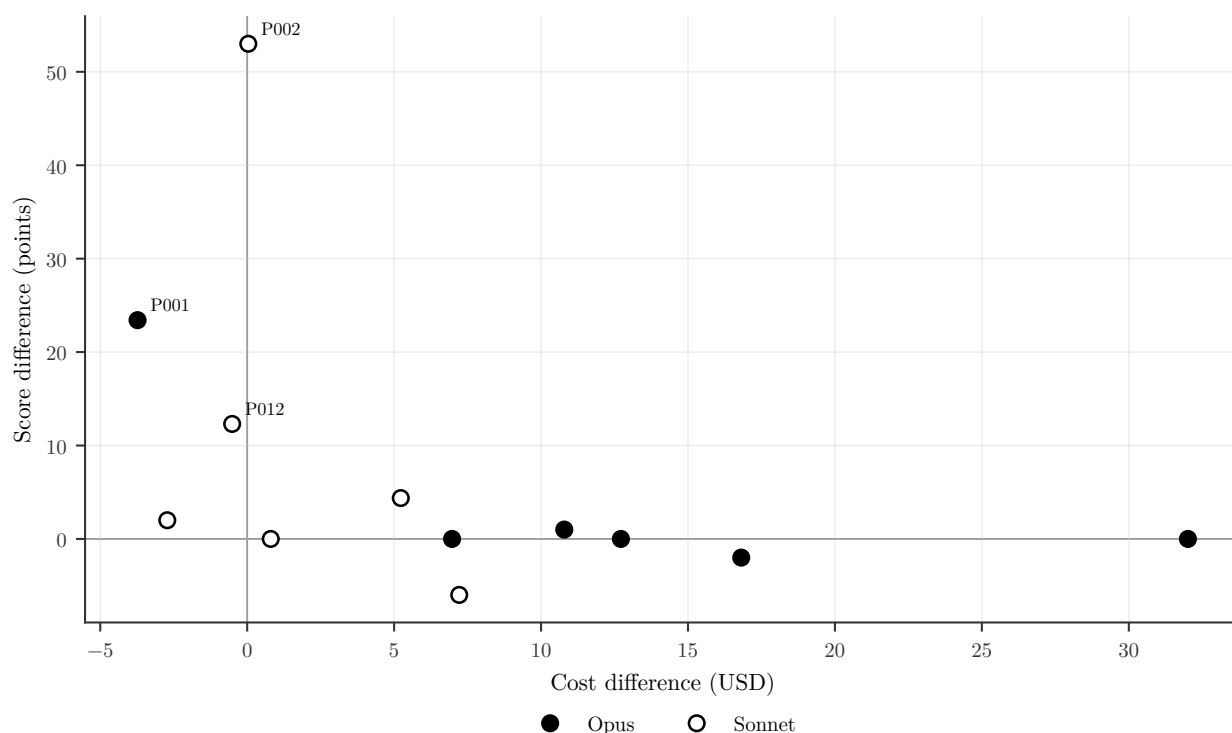


Figure 6.4: Per-participant paired cost difference and total-score difference, APM minus Spec-kit, with Opus filled and Sonnet open. The reference lines mark no difference on each axis.

The comparison prices software production. APM cost more for a functionally comparable outcome, but software also incurs corrective and maintenance costs after delivery. A more expensive session could still yield a lower lifecycle cost if it leaves software easier to understand, change, and repair. Those qualities depend on the maintainer, later changes, and other conditions that vary across projects and over time. This study therefore confines the comparison to production cost and functional behaviour.

APM's added cost accompanied a more structured workflow, which the total score does not read directly. In this study, the task proved bounded and self-contained. With either model and without either framework guiding it, Claude Code could plausibly work through the task largely on its own while launching subagents where needed. The study did not test that counterfactual, so it leaves open whether the underlying coding agent would achieve comparable total scores, or how the grader would read its work. APM's heavier structure therefore had little total-score gap to close, while whether that structure earns its cost lies beyond what the total score can show.

6.4 Perceived Versus Measured Behaviour

The ratings reach what the instruments do not: a participant's confidence in what was built, and the control they kept over building it, leave little trace in a transcript or a score. We read each rating of Section 5.4 against the behaviour recorded beneath it, for agreement and for divergence.

Effectiveness and confidence favour APM far more strongly than the total-score comparison does. No participant rated Spec-kit higher on either, as Table 5.15 reports. Yet the paired total-score comparison carries no firm difference across the cohort, so higher scores alone do not explain it. The accounts that follow point instead to participants' role in the process and the control they retained over the build.

One participant, who used APM in their second session, wrote that they were “much more confident in APMs output” despite “not having manually checked either of the code output”. They named the framework being “strict as to enforcing the PRD when speckit was more lenient” as the reason. Their confidence reads as trust in a specification the participant helped settle during planning. It is not a judgement of code they verified, and it echoes the reading Section 6.1 already builds from the scores. The Spec-kit accounts name the corresponding lack of control directly. One participant wrote that they “felt lack of control when it came to the deliverables and how they were produced”. Another found it odd that the framework did not operate interactively, letting the agent take its own decisions with the user left out entirely.

Effort is the one item where APM is behind. It was the more effortful framework for a slight majority of participants. That effort does not align with where APM adds turns. Section 6.2.2 reads APM's extra implementation turns as the intended user-mediated coordination. Section 6.1 reads its planning turns differently: the count there sits level between the frameworks, but APM's prompts are the more substantive ones, since its Planner brings real open questions to the user while gathering context. One participant described planning as the “most demanding thing” because it involved “questions about the whole plan in advance”. They then wrote that the Workers executed the task “easily without any assistance from me”, apart from authorising terminal commands and checking that it aligned with the task.

Frustration points the other way from effort. APM was still rated the less frustrating framework, despite the added time and turns, so the added burden did not compound into felt friction.

Mental demand and the effort spent checking the agent's output have no measured counterpart. That absence is itself a finding. Some of the cognitive work that AI-assisted development demands leaves almost no trace in the instrument set. Self-reporting attempts to fill that gap. One participant's account of a Spec-kit session is the clearest instance. They wrote, “I didn't have a way to validate the results of the session despite taking the agents word which run tests with the mock instance”. They added, “I didn't do my due diligence in reviewing the code output”. That session's total score is the lowest in the cohort, so here the participant's own doubt and the measured outcome point the same way. The account reflects a common failure in AI-assisted software development. When developers stop reading generated code, unreviewed diffs accumulate, leaving them to trust tests that were themselves generated by the agent and never reviewed.

6.5 APM's Value Across the Findings

Across the cohort, APM achieved a comparable total score with longer sessions in every pair and a higher cost. Each score summary favoured APM, although the interval includes zero, as Chapter 5 reports. With six pairs per model group, the split locates that pattern under Sonnet without establishing a model-specific effect.

The cost trade-off depended on how much of the task fit within the model's context window limit, as Section 6.2.1 explains. Under Opus, APM's several conversations cost more while both frameworks scored near the ceiling. Under Sonnet, APM's score summaries were more positive and the frameworks cost about the same. The study did not vary task size, so it cannot establish when distributing work across conversations offsets their aggregate cost.

The process and perception findings give APM's structure value beyond the total score. Its additional interaction and time fall in implementation, where user-mediated coordination operates, as Section 6.2.2 reports. Participants remained involved in planning and implementation, favoured APM in effectiveness and confidence ratings, and described greater control. For users who want that role, APM's additional time and interactions may represent *productive friction*: opportunities to inspect decisions and preserve understanding of the resulting system. Removing those points of interaction allows faster autonomous execution, but leaves fewer occasions to engage with software for which the developer remains responsible.

This value depends on the work and on how the user runs APM. On small or straightforward work, where the user can review a single agent's output directly, the same friction would likely add cost and time for little benefit. How often the user meets these points also depends on how they size and dispatch Tasks, which participants chose freely. The study examined neither that choice nor whether participants used these points to review code, prevent defects, or reduce technical debt.

6.6 Learning Effects

The within-participant comparison has each participant carry out the same task twice, with the environment reset between the two sessions so that no code, workspace state, or assistant context passes from the first into the second. Anything a participant took from their first attempt could then lift their second, and left unaccounted it would be indistinguishable from an effect of the framework used. Reading the cohort for such an effect holds to the same reproducible standard as the rest of the chapter, and so sets qualitative transcript reading aside.

That standard fixes what evidence the question can draw on. The chapter measures each session along four dimensions, and the score is not the only measure that could in principle bear on learning. The interaction measures of Section 5.2.2 might shift as the task grew familiar. The obstacle is that the comparison is within participant, between two sessions that used different frameworks. The two demand such different interaction that the difference between a participant's sessions is already large from the framework change alone. A learning-driven change could merge into those framework differences in the interaction measures, session cost, and session duration. The score is the task's graded outcome on one rubric, directly comparable across a participant's two sessions and the measure practice should raise. We therefore read the score alone for learning effects.

One limit bounds what the score can show of a learning effect: the rubric's hundred-point ceiling. Most sessions scored near it, as Table 5.12 reports, and Table B.10 reports nine of the twelve participants above ninety in their first session. A practised second session had little room to register a gain, and only three participants entered their second with substantial ground to recover.

The individual changes have no common direction. Four participants scored higher in their second session and four scored lower, as Table 6.6 lists, while the other four scored the same. Nine changes lay between a two-point loss and a six-point gain. Of the three larger movements, P002 gained 53.0 points while P001 and P012 lost 23.4 and 12.3 points. These movements all favour APM regardless of where it fell in the order, so they cannot separate a practice effect from the framework or from session-to-session variation.

Table 6.6: Change in total score from first session to second, for each participant whose score moved.

Participant	Model	Second framework	Score change ^a
P001	Opus	Spec-kit	−23.4
P002	Sonnet	APM	+53.0
P004	Opus	APM	−2.0
P006	Sonnet	Spec-kit	+6.0
P007	Sonnet	Spec-kit	−2.0
P008	Opus	APM	+1.0
P010	Sonnet	APM	+4.4
P012	Sonnet	Spec-kit	−12.3

^a The change is the second-session score minus the first-session score. Positive values indicate a higher score in the second session.

P010 reads differently under the diagnostic re-grade of Section 6.3.1. Their first-session diagnostic score remains 29.6 while the second rises to 93.9, widening the difference from 4.4 to 64.3 points. The re-grade recovers a strong second implementation but reaches only one fault class; the unchanged first-session score cannot establish that the first implementation was unsound. The diagnostic difference still cannot separate learning from the framework change.

The order split supports the same cautious reading. Table B.11 of Appendix B.3 separates the paired total-score difference by which framework each participant met first, and both confidence intervals span zero. Counterbalancing distributed first exposure across the frameworks, although the realised seven-to-five allocation was not perfectly even.

The total scores have no cohort-wide learning direction. Mixed changes, the score ceiling, and the uneven order allocation prevent them from identifying a learning effect.

6.7 Threats to Validity

A measured comparison is exposed at four points: what its instruments capture, what else could have produced its differences, how far its findings reach, and who built what was measured. Each threat identifies the inference at risk, the study's response, and the remaining limitation.

6.7.1 Construct Validity

The turn-prompt buckets were constructed for this study to describe the form the user's interaction with the agent took. The three buckets of Section 4.5.2 separate a framework command from a short message and a longer one at a fifty-character threshold. The study sets that boundary. As a description of interaction form the buckets measure what they were built to measure. As a stand-in for the effort behind the interaction they are weaker. A framework command is a few keystrokes for the user and can set a substantial amount of agent work in motion, while a short reply may follow a long stretch of reading. The comparison they support between the frameworks is therefore one of interaction form, and any reading of user effort taken from them carries that gap.

Reported session costs standardise token consumption across sessions. For each conversation, every request's tokens are priced against the published per-token rates of Appendix A.4. A developer's total cost would also include the participant's time and the cost of correcting and maintaining software after delivery. Neither is measured here, and both can outweigh the token cost of the session that produced them. The paired differences are untouched by that, since both sides of every pair are priced identically.

The cost of a session is a floor rather than a total, for two reasons. When a conversation compacts, the platform makes a further call to summarise the context it is about to discard. That call is not written to the transcripts participants supply, so its tokens are absent from every figure here. Nine of the twenty-four sessions compacted at least once, with twelve compactations between them. A session is also measured only to the three-hour cap of Section 4.4.3, so anything after that boundary lies outside every value derived from the transcripts.

The remaining dimensions' instruments each reach a defined part of the session too, and what falls outside differs between them. The process measures locate activity on the transcript's timeline, so a participant reading the codebase, testing a build, or thinking a decision through registers only as an interval between turns. The functional measures test the running program's behaviour against the behaviour the task asks for, in the manner of assertions, so they say nothing about code quality, maintainability, security posture, or the debt a solution leaves behind. A single blocking fault also reads there as near-total failure, which Section 6.3.1 shows is not hypothetical. The perception measures are single self-reported ratings standing in for constructs such as mental demand and effort, which dedicated instruments measure through several items each and which one rating cannot separate. Each dimension's headline measure carries its own claim, and none of them stands in for another.

We chose a contained replica of Open eClass over a live instance. The shared replica of Section 4.4.2 pins the platform to a fixed upstream baseline and seeds deterministic fixture data. Every participant develops against identical state and the grading runs against an equivalent one, which is what makes the cohort comparable and the scoring reproducible. The choice gives up any property of a live deployment the replica does not reproduce, since the task never exercises it. Such properties include timezone handling of locale-formatted dates, session-cookie behaviour under load, cross-origin interactions with the mock single-sign-on service, and upstream behaviour that has shifted since the pinned baseline.

6.7.2 Internal Validity

Each participant carried out the same task twice, so familiarity gained in the first session could lift the second and be taken for an effect of the framework. Counterbalancing the framework order exists to separate the two, as Section 4.4.3 describes, and the realised allocation left the four assignment cells uneven. The order splits of Appendix B.1 and their counterparts show no order effect on any headline measure. Section 6.6 reads the total score further, participant by participant, and finds no cohort-wide direction in it. The evidence, however, reaches an absence in our measures and does not establish that no learning occurred.

The framework comparison is paired within each participant, and the model comparison is not. Each participant kept one model across both sessions, so a difference between the Opus and the Sonnet group is a difference between six participants and six others, and it carries their skill and working style along with the model. Section 4.6 therefore reads every statement between the model groups descriptively.

Neither Claude Code nor the comparison framework was version-locked across the study window. Spec-kit sessions used a range of releases, which we audited release by release over the instructions installed into the project. One release changed what the agent is told, and that by a single line, as Section 4.3 records and Appendix A.1 reports. What the audit establishes is that the cohort met one workflow throughout, while whether that single line shifted the sessions meeting it lies beyond its reach.

Claude Code itself was updated repeatedly over the same period, so some participants' two sessions ran on different versions of it. One instance of that drift left a mark on the measures. In four Sonnet Spec-kit sessions, the conversation continued past its reported context window limit without automatic compaction. Claude Code continued to identify the model as Sonnet 4.6 after its context window consumption indicator passed full. The sessions appear to have been served by the model's one-million-token variant, which the platform offered separately but did not identify. Table 6.7 gives the cost of the resulting cache-read excess. Earlier compaction could have altered the agent's subsequent behaviour, making any counterfactual session cost hypothetical.

Table 6.7: Cache reading above the 171k observed automatic-compaction bound in the four Sonnet conversations served past their stated context window limit, in thousands of tokens, with its corresponding session cost.

Participant	Cache read ^a	Above bound ^b	Cost	Session cost
P003	30755.9	985.4	0.30	14.73
P006	16654.5	1487.1	0.45	8.25
P007	45132.7	6042.5	1.81	19.46
P012	26723.5	944.0	0.28	13.42

^a All cache-read tokens, including those accumulated below the 171k observed automatic-compaction bound.

^b Summed over every call whose context stood above the bound, the part standing above it, priced at that call's own cache-read rate and capped at the reads the call made.

^c Removing each session's excess from its Spec-kit side leaves every Sonnet pair's direction and the group's sign count unchanged. The Sonnet session-scope median, Hodges-Lehmann estimate, and mean become +0.57, +2.16, and +2.15, against the +0.42, +1.25, and +1.68 of Table 5.2.

Resetting a conversation was partly a participant decision. The platform is expected to compact automatically as a conversation nears the auto-compaction threshold, but the anomaly above shows that a reset was not guaranteed. Participants could compact manually at any point. Spec-kit conversations under Sonnet reset both automatically and manually, as Table 6.2 reports. A reset changes what every later call is charged for, so where the participant chose the timing, subsequent cost reflects their judgement alongside the agent's and framework's behaviour. The two cannot be separated: the same accumulated context is both a product of work under the framework and a reason for the participant to reset it.

Participants knew which framework they used in each session, because blinding was not possible: the two workflows differ visibly in what they ask of the user and how the agents behave. They also knew the two were being compared. The ratings of Section 5.4 are therefore open to recall bias and to a pull towards the answer a participant expects to be wanted, and no instrument-side measurement corrects either. Section 6.4 reads each rating against the behaviour measured beneath it and reports the divergences as findings. That reading does not isolate the bias, since a participant answering as they believe is wanted may land either with the measures or against them, so neither agreement nor divergence identifies it.

6.7.3 External Validity

The study is conducted on one coding agent, against one task, in a setting between laboratory and field conditions. The task asks for work in two languages and research into an unfamiliar legacy codebase, as Section 4.4.1 sets out, exercising a demanding end-to-end workload. It is nonetheless one task rather than a benchmark of many, and it runs against the contained environment the construct-validity discussion above describes, which fixes what the cohort meets at the price of a live deployment's variety. The findings hold within that scope. They do not reach other coding agents, projects where developers hold deep domain ownership, work simpler or less knowledge-intensive than this task, work substantially larger than it, or purely laboratory or natural settings. This boundary bears directly on the cost interpretation, which Section 6.2.1 relates to how much of the task fits within the model's context window limit.

Both the wider setting of artificial intelligence and the two frameworks move quickly, and they move for different reasons. Token prices, model availability, and the pace at which agentic frameworks release belong to AI-assisted software development at large. The command sets, agent instructions, and workflows examined here belong to APM and Spec-kit, and change at each project's own pace. Every statement of either kind belongs to the study window of Section 4.3. The cost values of Chapter 5 are dollar equivalents priced from the rate card of Appendix A.4. They are defensible against that snapshot and would need re-estimating under a materially different one. The descriptive claims about how each framework behaves are tied to the releases tested.

Our measures were built for this comparison and have been applied once. Some take their shape from the frameworks under it: the protocol bucket of Section 4.5.2 is an example, as it measures interaction only through commands issued by the two frameworks used in this comparison. The parser also reads transcripts produced by Claude Code, a closed product whose transcript format has changed since the study window. How far the methodology reaches beyond this comparison is untested.

6.7.4 Reliability

Before the case study, the author independently created APM and the codebase that the task extends. As part of this thesis, we developed APM v1, then designed the study and its case-study methodology, wrote the task and participant brief, and built the task environment. We also recruited the cohort, which comprises twelve university students and working professionals. The within-participant comparison supports a claim within this cohort, not an inference about a wider population.

Additionally, we built the parser and the grader the study uses. The grader applies the same fixed assertions to each of a participant's submissions, so automated scoring leaves no interpretive discretion at that step. It does not, however, make the task, rubric, or measurement choices independent. Nothing in the study's design offsets the resulting author bias. AI assistance was used throughout this work, in its engineering and its writing. The infrastructure and analysis scripts are published openly. The dataset is published encrypted to protect the participants, and its key is available to reviewers on request. Together they allow our measurements and scoring to be independently reproduced.

The chapters draw several views from the same twenty-four sessions. Each dimension has one session-scope headline measure. Phase and role scopes, measurement breakdowns, and participant-group breakdowns explain that result rather than carrying separate primary claims. The smaller group breakdowns have wider uncertainty and include differences between participants. The analysis scripts produce the statistics, tables, and figures.

7. CONCLUSION

This thesis presented APM v1, a ground-up redesign of the open-source APM framework. It operates on top of coding agents, combining persistent planning documents and role-specific division of work with user-mediated coordination. It also proposed a reproducible case-study methodology for comparing frameworks of this class, and implemented it to compare APM v1 against Spec-kit through a within-participant case study with twelve participants, covering economic, process, functional, and perception outcomes.

Across the cohort, the frameworks carried no firm difference in functional outcome. That comparability is what a grader reading only the running program's responses can see. Section 6.3 bounds what it reaches.

APM cost more, and its several conversations account for that. Each accumulates its own context, builds its own cache, and produces its own output. At session scope their costs add together. Under Opus the whole task fit within one context window limit and no conversation reset, leaving nothing to offset that aggregate against Spec-kit's single conversation. Under Sonnet the same task repeatedly pushed Spec-kit towards Claude Code's auto-compaction threshold, and it reread a large accumulated cache before each reset. APM instead kept its conversations' context smaller and compacted less, bringing the two frameworks' costs close, as Section 6.2.1 explains.

The APM session lasted longer in every pair and had more turn prompts in eleven of the twelve, with its additional interaction and time falling together in implementation. APM coordinates several agent conversations through the user by design, and both follow from it. Each Task Prompt and Task Report passes through the user and opens a turn in the conversation receiving it. More turn prompts open more intervals of agent work, and turn time made up most of that phase's longer duration, as Section 6.2.2 reports.

Against those additional process and economic demands, participants rated APM higher on effectiveness and confidence, and described greater control over the development process. Our interaction measurements show the user-mediated coordination operating. They leave open whether participants used it to review code, prevent defects, or improve the resulting software. The findings remain bounded by the factors Section 6.7 discusses. The methodology is meant to apply to other frameworks of this class, with a task, environment, and tooling suited to each comparison.

7.1 Future Work

This thesis concentrated on designing and building APM v1, and on designing the evaluation methodology and building its infrastructure. Several directions that follow from that work remain open, and we outline them for researchers who wish to pursue them.

Task size and work decomposition. It remains open how task size and demand shift the value of multi-agent systems and their decomposition of work. Future evaluations of frameworks on top of coding agents could vary them, comparing single-agent continuous workflows with multi-agent decomposition across models of differing capability and context limits. That comparison would show how the fit between task demand and context limits shifts the trade-off between an accumulating conversation and distributed agent contexts. Such evaluations would also need an unguided agent baseline. A less capable model may require additional guidance to succeed, whereas a capable model may manage alone, rendering framework structure pure overhead. Rotating these factors against one another would isolate where such decomposition provides genuine economic and functional value. Monitoring how users size and dispatch APM Tasks would also show how often exchanges pass through the user, how often users take these opportunities to review the work, and what that costs.

Maintainability of produced software. Software is read, changed, and repaired long after it is delivered. Deterministic tests verify immediate correctness, but leave architectural degradation and downstream maintainability unmeasured. Rapid code generation risks producing code bloat and superficial solutions that satisfy

immediate checks yet complicate later maintenance. A longitudinal evaluation following software produced by APM alongside other frameworks of this class would track defect correction, code churn, and lifecycle cost. Two properties of APM's design bear directly on this: focused Workers operating on scoped contexts to resist context decay, and the user-mediated coordination mechanism. Such an evaluation would test whether these mechanisms protect against software degradation, repaying APM's upfront coordination overhead with lower maintenance costs later.

Developer comprehension. Beyond coordination mechanics, user mediation raises questions about developer comprehension and authority. The turn-prompt buckets record the form of user input but not its content; classifying that content across the session, and for each APM role, would show if, and how, participants exercised their authority at each friction point. Our case study measured self-reported confidence and perceived control, but left technical understanding unmeasured. That gap leaves open whether APM fosters genuine program comprehension or an illusion of control. Such an illusion could add to the cognitive debt that builds as developers offload mental effort to AI assistants [58]. Future studies could adapt empirical comprehension methods, using structured mental representation tasks [59] to test developers objectively on program structure, data flow, and state transitions depending on the task at hand. Evaluating comprehension accuracy alongside subjective ratings would determine whether the framework's design preserves usable developer authority or merely inflates perceived confidence.

User mediation against autonomous coordination. Finally, it remains open what user mediation itself contributes. APM's user-mediated coordination could be compared directly with an autonomous variant whose agents communicate without the user, such as the APM Auto design described in Section 3.7. A controlled comparison would isolate the contribution of human oversight from the coordination of its agents, measuring whether removing the mechanism accelerates delivery at the expense of autonomous drift, undetected error propagation, and downstream maintenance debt across complex dependency graphs.

ABBREVIATIONS - ACRONYMS

Abbreviation	Meaning
AI	Artificial Intelligence
API	Application Programming Interface
APM	Agentic Project Management
CI	Confidence Interval
HL	Hodges-Lehmann estimate
JSON	JavaScript Object Notation
JSON-RPC	JSON Remote Procedure Call
LLM	Large Language Model
MAS	Multi-Agent System
MCP	Model Context Protocol
PHP	PHP: Hypertext Preprocessor
PRD	Product Requirements Document
RQ	Research Question
SDD	Spec-Driven Development
SDK	Software Development Kit
SK	Spec-kit
TDD	Test-Driven Development
YAML	YAML Ain't Markup Language

APPENDIX A. EVALUATION APPENDIX

A.1 Spec-kit Release Audit

The audit installs each release of the span as a project and diffs each against the next, since Spec-kit renders its templates as it installs them. The compared files are the commands of Section 4.2 together with file templates, scripts, and a context file they instruct the agent to use, each compared byte for byte.

What a change amounts to is a matter of interpretation, since one line can carry a great deal of instruction to an agent. The audit therefore ships a unified diff per release step, alongside a manifest hashing every installed file at every release that ties each diff to the release it comes from.

Six of the eight steps leave the files identical. Of the other two, v0.8.9 cleans up whitespace in three templates, leaving v0.8.6 as the one release to alter what the agent is told, adding a line that directs the implementing agent to reread the project constitution where one exists. Appendix A.2 locates the script and its output, where each change can be read in full.

A.2 Software and Data Availability

The study's software is openly available, and its dataset is published encrypted, in the following repositories:

- The task and participant-materials repository, holding the task brief, the participant guide, and the portable testing environment, at <https://github.com/sdi2200262/bsc-apm-case-study-task>;
- The evaluation infrastructure, holding the transcript parser, the grader with its scoring rubric, and the grading environment, at <https://github.com/sdi2200262/bsc-apm-case-study-infra>;
- The dataset of participant submissions, from which our measurements are computed, as one encrypted archive whose key is available to reviewers on request, at <https://github.com/sdi2200262/bsc-apm-case-study-data>;
- The analysis scripts that compute the reported statistics and produce the figures and table data from the parser and grader outputs, in the source repository under `analysis/`, at <https://github.com/sdi2200262/bsc-apm-thesis>;
- The Spec-kit release audit of Appendix A.1, holding its script together with the diffs and per-file hashes it produced, in the same repository under `spec-kit-audit/`.

A.3 Questionnaire Instrument

The second-session question, "Compared to the framework you used in your first session, completing the task with this framework felt...", offers five levels:

- much harder
- harder
- about the same
- easier
- much easier

The completion-state question, "Completion state", offers four options:

- Completed the task comfortably within the three-hour cap
- Completed the task near the three-hour cap

- Hit the three-hour cap without completing
- Stopped early without completing

The two free-text prompts are:

- “What stood out about working with this framework on this task?”
- “Anything else you want to add about this session?”

The eight rating items and the wording at each end of their scale are given in Table A.1.

Table A.1: The post-session questionnaire’s rating items, with the wording at each end of the one-to-five scale.

Question	1	5
How familiar were you with this framework before this study?	Never used it	Use it regularly
How confident are you in your submitted solution?	Very low	Very high
How effective was the framework in helping you complete the task?	Not effective	Very effective
How mentally demanding was the session?	Very low	Very high
How frustrated did you feel during the session?	Not at all	Extremely
How much effort did you have to invest?	Very low	Very high
How much effort went into checking what the AI in this framework produced?	Very low	Very high
How much effort went into instructing and steering the AI in this framework?	Very low	Very high

A.4 Rate Card

The dollar-equivalent cost prices each request’s tokens at the rates in Table A.2. The rates cover five token categories: fresh input and output, reads from the prompt cache, and writes to the cache at its five-minute and one-hour retention periods.

Table A.2: Per-token rates for the study window, in US dollars per million tokens.

Model	Input	Output	Cache read	Cache write (5 min)	Cache write (1 h)
Opus 4.7	5.00	25.00	0.50	6.25	10.00
Sonnet 4.6	3.00	15.00	0.30	3.75	6.00
Haiku 4.5	1.00	5.00	0.10	1.25	2.00

Subagents launched within study sessions and some background operations were at times routed to Haiku. Some sessions therefore include a small share of Haiku tokens alongside their assigned model. The evaluation infrastructure listed in Appendix A.2 holds the rates in `rates.yaml`.

A.5 Scoring Rubric

The rubric’s ninety checks divide into twenty-seven over the Python server, worth thirty points, and sixty-three over the C replica, worth seventy. Ten checks carry two points and the remaining eighty carry one.

A string-matched check compares the observed response against the responses the rubric accepts. An exact match earns the check’s full points. Otherwise the grader measures how near the response came: it takes the

edit distance to the nearest accepted response, divides it by the length of the longer of the two, and subtracts the result from one. At a ratio of 0.45 or above the check earns its points scaled by that ratio; below it the check earns zero. Every partial check is flagged for review before the score is accepted.

Structural checks and memory checks carry no similarity. They pass or fail outright. Twenty-six of the C memory checks are each paired with a functional check and are awarded only where that check passes. A clean memory run of a non-functional program earns no points.

APPENDIX B. RESULTS APPENDIX

B.1 Economic Cost Breakdowns

Table B.1 gives the per-order detail behind the Economic results of Section 5.1: the cost difference by realised session order, with each order group's own summaries and interval.

Table B.1: Paired cost differences by realised session order, APM minus Spec-kit, in US dollars, at 95.3% coverage for the $n = 7$ row and 93.8% for $n = 5$.

Order	n	Sign (+/−)	Median	HL	Mean	CI
APM first	7	4 / 3	+0.80	+2.74	+5.72	[−2.72, +19.49]
APM second	5	5 / 0	+10.79	+8.97	+9.12	[+0.04, +16.81]

B.2 Process Breakdowns

Table B.2 gives the model-specific paired time differences behind the Process results of Section 5.2, at session and phase scope. At session scope, APM was longer in all six pairs under each model, with a wider duration gap under Opus. Its turn-time and inter-turn-time components were higher in at least five of the six pairs in each group. At phase scope, the implementation phase was longer under APM in all six pairs of each model, with close median differences. For implementation duration, the Sonnet group had the higher Hodges-Lehmann estimate and mean. Planning remained mixed in sign under both models.

Table B.2: Paired time differences within each model group, by scope, APM minus Spec-kit, in seconds.

Group	Scope	Measure	n	Sign (+/−)	Median	HL	Mean	CI
Opus	Session	Duration	6	6 / 0	+2436.4	+2436.4	+2603.4	[+664.1, +5596.1]
		Turn time	6	6 / 0	+2206.9	+2139.4	+2076.9	[+519.7, +3415.5]
		Inter-turn idle	6	5 / 1	+674.4	+667.1	+526.5	[−1475.3, +2180.6]
	Planning	Duration	6	3 / 3	−46.2	−46.2	+205.5	[−885.5, +2347.0]
		Turn time	6	4 / 2	+108.6	+139.3	+168.5	[−1185.8, +1497.0]
		Inter-turn idle	6	3 / 3	+147.7	+132.4	+37.0	[−1188.3, +849.9]
	Implementation	Duration	6	6 / 0	+1820.6	+1862.6	+2259.3	[+331.7, +5752.0]
		Turn time	6	6 / 0	+1498.6	+1612.5	+1873.5	[+782.2, +3566.5]
		Inter-turn idle	6	4 / 2	+247.9	+247.9	+385.8	[−450.4, +2185.5]
Sonnet	Session	Duration	6	6 / 0	+1878.6	+2302.2	+2310.7	[+701.8, +4635.4]
		Turn time	6	5 / 1	+1199.3	+1384.4	+1643.8	[−1034.8, +3833.9]
		Inter-turn idle	6	6 / 0	+679.3	+679.3	+666.9	[+5.8, +1736.6]
	Planning	Duration	6	2 / 4	−366.6	−361.4	−321.3	[−1411.2, +821.2]
		Turn time	6	1 / 5	−344.1	−472.6	−452.3	[−1354.9, +308.5]
		Inter-turn idle	6	4 / 2	+98.6	+111.2	+130.9	[−156.0, +512.7]
	Implementation	Duration	6	6 / 0	+1862.5	+2486.9	+2592.8	[+589.7, +6046.7]
		Turn time	6	5 / 1	+1450.1	+1822.9	+2066.9	[−858.7, +5188.8]
		Inter-turn idle	6	5 / 1	+412.4	+464.9	+526.0	[−47.1, +1448.4]

Table B.3 lists the session-duration and turn-prompt differences within each realised session order. Neither order reverses either result; the central differences are comparable across the two groups.

Table B.3: Paired session-duration and turn-prompt differences by realised session order, APM minus Spec-kit. Duration is in seconds; the turn-prompt rows are counts. The $n = 7$ rows have 95.3% coverage and the $n = 5$ rows 93.8%.

Measure	Order	n	Sign (+/-/=)	Median	HL	Mean	CI
Duration	APM first	7	7 / 0	+2124.3	+2537.8	+2676.2	[+1201.5, +4635.4]
	APM second	5	5 / 0	+2197.1	+2197.1	+2150.3	[+664.1, +3902.6]
Turn prompts	APM first	7	6 / 0 / 1	+7.0	+7.5	+7.9	[+1.0, +15.0]
	APM second	5	5 / 0 / 0	+9.0	+10.0	+10.4	[+4.0, +19.0]

Table B.4 details each APM session's turn-time composition, including its solo and concurrent spans, behind the concurrency reading in Section 5.2.1.

Table B.4: Turn-time composition of the APM sessions, in seconds, with the concurrent span as a share of the turn time. Descriptive and APM only, since Spec-kit's single conversation has no concurrent span.

Participant	Solo	Concurrent	Concurrent share
P001	4834	0	0.0%
P002	5993	16	0.3%
P003	4458	661	12.9%
P004	5300	58	1.1%
P005	5686	0	0.0%
P006	6203	9	0.1%
P007	4765	812	14.6%
P008	3934	580	12.8%
P009	4476	1541	25.6%
P010	6556	1259	16.1%
P011	4218	577	12.0%
P012	4037	429	9.6%

Table B.5 details the turn-prompt difference by scope within each model group. The implementation difference set the session difference under both models and planning stayed level, with the gap clearer under Sonnet, where all six pairs favoured APM at the session and implementation scopes.

Table B.5: Paired turn-prompt count differences by scope within each model group, APM minus Spec-kit.

Group	Scope	n	Sign (+/-/=)	Median	HL	Mean	CI
Opus	Session	6	5 / 0 / 1	+6.5	+7.0	+8.0	[+0.0, +21.0]
	Planning	6	3 / 3 / 0	+0.0	+0.0	+0.0	[-10.0, +13.0]
	Implementation	6	5 / 0 / 1	+7.0	+7.0	+7.7	[+0.0, +22.0]
Sonnet	Session	6	6 / 0 / 0	+8.0	+8.5	+9.8	[+7.0, +19.0]
	Planning	6	2 / 2 / 2	+0.0	+0.0	-0.3	[-3.0, +2.0]
	Implementation	6	6 / 0 / 0	+9.5	+9.5	+10.0	[+5.0, +17.0]

Table B.6 gives the turn-prompt bucket shares within each model group, each bucket by its median, Hodges-Lehmann estimate, and mean.

Table B.6: Turn-prompt bucket shares within each model group, by phase and framework, each bucket by its median, Hodges-Lehmann estimate, and mean.

Group	Phase	Framework	Bucket	Median	HL	Mean
Opus	Planning	APM	Protocol	0.13	0.22	0.25
			Short	0.53	0.53	0.46
			Standard	0.31	0.30	0.29
		Spec-kit	Protocol	0.40	0.40	0.42
			Short	0.47	0.46	0.45
			Standard	0.07	0.08	0.13
	Implementation	APM	Protocol	0.74	0.74	0.76
			Short	0.10	0.12	0.12
			Standard	0.06	0.12	0.12
		Spec-kit	Protocol	0.20	0.25	0.27
			Short	0.32	0.33	0.38
			Standard	0.24	0.30	0.35
Sonnet	Planning	APM	Protocol	0.10	0.10	0.10
			Short	0.48	0.48	0.48
			Standard	0.42	0.42	0.42
		Spec-kit	Protocol	0.64	0.64	0.66
			Short	0.24	0.24	0.24
			Standard	0.10	0.10	0.11
	Implementation	APM	Protocol	0.70	0.68	0.66
			Short	0.15	0.15	0.17
			Standard	0.10	0.11	0.17
		Spec-kit	Protocol	0.38	0.38	0.38
			Short	0.12	0.25	0.29
			Standard	0.00	0.12	0.17

^a The three buckets are summarised independently, so their summaries need not sum to one; see note [a] of Table 5.10.

Table B.7 lists the corresponding turn-prompt character counts within each model group.

Table B.7: Turn-prompt bucket character counts within each model group, by phase and framework, each bucket by its median, Hodges-Lehmann estimate, and mean.

Group	Phase	Framework	Bucket	Median	HL	Mean
Opus	Planning	APM	Protocol	150	144	144
			Short	20	21	21
			Standard	85	139	136
		Spec-kit	Protocol	104	104	110
			Short	20	14	15
			Standard	81	81	81
	Implementation	APM	Protocol	117	121	122
			Short	26	23	23
			Standard	88	93	95
		Spec-kit	Protocol	100	100	100
			Short	29	29	29
			Standard	74	74	71
Sonnet	Planning	APM	Protocol	150	150	150
			Short	19	18	17
			Standard	167	167	155
		Spec-kit	Protocol	113	115	117
			Short	25	25	25
			Standard	82	88	88
	Implementation	APM	Protocol	130	129	128
			Short	15	17	18
			Standard ^a	69	134	237
		Spec-kit	Protocol	100	101	144
			Short	38	38	37
			Standard	117	117	117

^a P002's high APM implementation standard-prompt character count is explained in note [b] of Table 5.10.

Table B.8 details the turn-prompt bucket shares and character counts for each APM agent role, pooled over the role's agents across the sessions.

Table B.8: Per-role turn-prompt bucket shares and character counts across the twelve APM sessions, pooled over each role's agents. Descriptive and APM only, since Spec-kit has no agent roles to separate.

Role	Bucket	Bucket share			Magnitude (chars)		
		Median	HL	Mean	Median	HL	Mean
Planner	Protocol	0.11	0.12	0.18	150	150	147
	Short	0.50	0.50	0.47	20	20	19
	Standard	0.38	0.37	0.35	156	139	146
Manager	Protocol	0.59	0.58	0.56	130	127	125
	Short	0.16	0.17	0.18	21	21	22
	Standard	0.15	0.19	0.18	79	88	99
Worker	Protocol	0.92	0.85	0.81	128	128	128
	Short	0.00	0.08	0.10	29	22	23
	Standard ^a	0.00	0.00	0.08	96	326	403

^a P002's high standard-prompt character count is explained in note [b] of Table 5.10.

^b The three buckets are summarised independently, so their summaries need not sum to one; see note [a] of Table 5.10.

Table B.9 gives each session's steers, separating those delivered to the working agent from those that were not.

Table B.9: Steers per participant and framework, at session scope.

Participant	APM		Spec-kit	
	Delivered	Not delivered	Delivered	Not delivered
P001	0	0	0	0
P002	0	0	0	2
P003	0	2	0	1
P004	0	0	0	0
P005	0	0	0	0
P006	1	0	0	1
P007	0	1	0	0
P008	0	0	0	0
P009	0	1	0	0
P010	0	1	0	0
P011	2	2	0	0
P012	0	3	0	2
Cohort	3	10	0	6

B.3 Functional Breakdowns

Table B.10 gives every session's strict and partial score parts, total score, and Python and C section totals, behind the Functional results of Section 5.3.

Table B.10: Per-session scores, in points. The total is the strict part plus the partial part, and the Python and C section totals sum to it.

Participant	Framework	Strict	Partial	Total	Python	C
P001	APM	91.0	3.4	94.4	28.7	65.7
	Spec-kit	71.0	0.0	71.0	20.0	51.0
P002	APM	93.0	0.0	93.0	28.0	65.0
	Spec-kit	40.0	0.0	40.0	10.0	30.0
P003	APM	34.0	0.0	34.0	30.0	4.0
	Spec-kit	34.0	0.0	34.0	30.0	4.0
P004	APM	95.0	0.0	95.0	28.0	67.0
	Spec-kit	97.0	0.0	97.0	30.0	67.0
P005	APM	100.0	0.0	100.0	30.0	70.0
	Spec-kit	100.0	0.0	100.0	30.0	70.0
P006	APM	88.0	3.0	91.0	27.5	63.5
	Spec-kit	97.0	0.0	97.0	29.0	68.0
P007	APM	98.0	0.0	98.0	29.0	69.0
	Spec-kit	96.0	0.0	96.0	29.0	67.0
P008	APM	99.0	0.0	99.0	30.0	69.0
	Spec-kit	98.0	0.0	98.0	30.0	68.0
P009	APM	100.0	0.0	100.0	30.0	70.0
	Spec-kit	100.0	0.0	100.0	30.0	70.0
P010	APM	34.0	0.0	34.0	30.0	4.0
	Spec-kit	28.0	1.6	29.6	16.0	13.6
P011	APM	99.0	0.0	99.0	30.0	69.0
	Spec-kit	99.0	0.0	99.0	30.0	69.0
P012	APM	96.0	0.5	96.5	30.0	66.5
	Spec-kit	78.0	6.2	84.2	26.1	58.1

Table B.11 gives the total-score difference by realised session order.

Table B.11: Paired total-score difference by realised session order, APM minus Spec-kit, in points, at 95.3% coverage for the $n = 7$ row and 93.8% for $n = 5$.

Order	n	Sign (+/-/=)	Median	HL	Mean	CI
APM first	7	3 / 1 / 3	+0.0	+1.5	+4.5	$[-3.0, +12.7]$
APM second	5	3 / 1 / 1	+1.0	+2.2	+11.3	$[-2.0, +53.0]$

B.4 Perception Breakdowns

Table B.12 gives responses to the comparison question of Section 5.4, grouped by the framework used in the second session.

Table B.12: Responses to the comparison question, by the framework used in the second session. Each row is the participants whose second framework was the named one, and the counts are how that session felt against their first.

Second framework	Much harder	Harder	About the same	Easier	Much easier
APM ^a	0	2	1	1	1
Spec-kit	2	2	1	0	1

^a P012 is excluded from this row; their comparative response is tied to the session they labelled second, which does not match the session they actually completed second, so it cannot be assigned to either framework's realised-order group. Their other perception ratings are reported per session and retain their classification.

Table B.13 gives each framework's sessions across the four completion states the questionnaire offered.

Table B.13: Self-reported completion state against the three-hour cap, by framework.

Framework	Comfortably within	Near the cap	Hit the cap	Stopped early
APM	9	3	0	0
Spec-kit	10	2	0	0

BIBLIOGRAPHY

- [1] J. Liu, K. Wang, Y. Chen et al., “Large language model-based agents for software engineering: A survey”, *ACM Transactions on Software Engineering and Methodology*, Mar. 2026. DOI: 10.1145/3796507
- [2] N. F. Liu, K. Lin, J. Hewitt et al., “Lost in the middle: How language models use long contexts”, *Transactions of the Association for Computational Linguistics*, vol. 12, 2024. DOI: 10.1162/tac1_a_00638
- [3] D. Delimarsky. “Spec-driven development with AI: Get started with a new open source toolkit”, GitHub, Accessed: 12th Oct. 2026. [Online]. Available: <https://github.blog/ai-and-ml/generative-ai/spec-driven-development-with-ai-get-started-with-a-new-open-source-toolkit/>
- [4] S. Hong, M. Zhuge, J. Chen et al., “MetaGPT: Meta programming for a multi-agent collaborative framework”, in *International Conference on Learning Representations (ICLR 2024)*, May 2024. arXiv: 2308.00352 [cs . AI] .
- [5] K. Chaidemenos, *Agentic Project Management (APM)*, 2026. Accessed: 12th Oct. 2026. [Online]. Available: <https://github.com/sdi2200262/agentic-project-management>
- [6] GitHub. “GitHub Spec Kit”, Accessed: 12th Oct. 2026. [Online]. Available: <https://github.github.io/spec-kit/>
- [7] N. Friedman. “Introducing GitHub Copilot: Your AI pair programmer”, GitHub, Accessed: 12th Oct. 2026. [Online]. Available: <https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/>
- [8] S. Wu. “Introducing Devin, the first AI software engineer”, Cognition, Accessed: 12th Oct. 2026. [Online]. Available: <https://cognition.com/blog/introducing-devin>
- [9] I. Nikolic. “Introducing GitHub Copilot agent mode (preview)”, Visual Studio Code, Accessed: 12th Oct. 2026. [Online]. Available: <https://code.visualstudio.com/blogs/2025/02/24/introducing-copilot-agent-mode>
- [10] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021, ISBN: 978-1-292-40113-3.
- [11] S. Yao, J. Zhao, D. Yu et al., “ReAct: Synergizing reasoning and acting in language models”, in *International Conference on Learning Representations (ICLR 2023)*, May 2023. arXiv: 2210.03629 [cs . CL] .
- [12] K. Z. Cui, M. Demirel, S. Jaffe, L. Musolff, S. Peng and T. Salz, “The effects of generative AI on high-skilled work: Evidence from three field experiments with software developers”, *Management Science*, Feb. 2026. DOI: 10.1287/mnsc.2025.00535
- [13] Anysphere, Inc., *Cursor*, 2026. Accessed: 12th Oct. 2026. [Online]. Available: <https://cursor.com>
- [14] J. Becker, N. Rush, E. Barnes and D. Rein. “Measuring the impact of early-2025 AI on experienced open-source developer productivity”. arXiv: 2507.09089 [cs . AI] , pre-published.
- [15] S. Thorgeirsson, T. B. Weidmann and Z. Su, “Computer science achievement and writing skills predict vibe coding proficiency”, in *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*, ACM, Apr. 2026. DOI: 10.1145/3772318.3791666
- [16] Anthropic. “Context windows”, Accessed: 12th Oct. 2026. [Online]. Available: <https://platform.claude.com/docs/en/build-with-claude/context-windows>
- [17] Anthropic. “How Claude Code works”, Accessed: 12th Oct. 2026. [Online]. Available: <https://code.claude.com/docs/en/how-claude-code-works>
- [18] P. Gao and C. Peng. “More with less: An empirical study of turn-control strategies for efficient coding agents”. arXiv: 2510.16786 [cs . SE] , pre-published.

- [19] P. Rajasekaran, E. Dixon, C. Ryan and J. Hadfield. "Effective context engineering for AI agents", Anthropic, Accessed: 12th Oct. 2026. [Online]. Available: <https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents>
- [20] B. Böckeler. "Context engineering for coding agents", martinfowler.com, Accessed: 12th Oct. 2026. [Online]. Available: <https://martinfowler.com/articles/exploring-gen-ai/context-engineering-coding-agents.html>
- [21] B. Meyer, "Applying 'Design by Contract'", *Computer*, vol. 25, no. 10, Oct. 1992. DOI: 10.1109/2.161279
- [22] K. Beck, *Test-Driven Development: By Example*. Boston, MA: Addison-Wesley, 2002, ISBN: 978-0-321-14653-3.
- [23] J. S. Ostroff, D. Makalsky and R. F. Paige, "Agile specification-driven development", in *Extreme Programming and Agile Processes in Software Engineering (XP 2004)*, ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2004. DOI: 10.1007/978-3-540-24853-8_12
- [24] N. Swaminathan and D. Singh. "Introducing Kiro", Kiro, Accessed: 12th Oct. 2026. [Online]. Available: <https://kiro.dev/blog/introducing-kiro/>
- [25] M. Cemri, M. Z. Pan, S. Yang et al., "Why do multi-agent LLM systems fail?", in *Advances in Neural Information Processing Systems 38 (NeurIPS 2025), Datasets and Benchmarks Track*, Dec. 2025. arXiv: 2503.13657 [cs . AI] .
- [26] G. Li, H. A. A. K. Hammoud, H. Itani, D. Khizbullin and B. Ghanem, "CAMEL: Communicative agents for 'Mind' exploration of large language model society", in *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*, Dec. 2023. DOI: 10.52202/075280-2264
- [27] C. Qian, W. Liu, H. Liu et al., "ChatDev: Communicative agents for software development", in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL 2024)*, Association for Computational Linguistics, Aug. 2024. DOI: 10.18653/v1/2024.acl-long.810
- [28] Q. Wu, G. Bansal, J. Zhang et al., "AutoGen: Enabling next-gen LLM applications via multi-agent conversation", in *Conference on Language Modeling (COLM 2024)*, Oct. 2024. arXiv: 2308.08155 [cs . AI] .
- [29] M. Gao, Y. Li, B. Liu et al. "Single-agent or multi-agent systems? why not both?" arXiv: 2505.18286 [cs . MA] , pre-published.
- [30] J. Hadfield, B. Zhang, K. Lien, F. Scholz, J. Fox and D. Ford. "How we built our multi-agent research system", Anthropic, Accessed: 12th Oct. 2026. [Online]. Available: <https://www.anthropic.com/engineering/multi-agent-research-system>
- [31] W. Yan. "Don't build multi-agents", Cognition, Accessed: 12th Oct. 2026. [Online]. Available: <https://cognition.com/blog/dont-build-multi-agents>
- [32] BMad Code, LLC, *BMAD-METHOD: Breakthrough method for agile Ai driven development*, 2025. Accessed: 12th Oct. 2026. [Online]. Available: <https://github.com/bmad-code-org/BMAD-METHOD>
- [33] J. Vincent, *Superpowers*, 2025. Accessed: 12th Oct. 2026. [Online]. Available: <https://github.com/obra/superpowers>
- [34] B. Böckeler. "Understanding spec-driven-development: Kiro, spec-kit, and Tessel", martinfowler.com, Accessed: 12th Oct. 2026. [Online]. Available: <https://martinfowler.com/articles/exploring-gen-ai/sdd-3-tools.html>
- [35] Thoughtworks. "Spec-driven development". Technology Radar Vol. 33, Accessed: 12th Oct. 2026. [Online]. Available: <https://www.thoughtworks.com/radar/techniques/spec-driven-development>
- [36] A. Karpathy. "There's a new kind of coding I call 'vibe coding'", X, Accessed: 12th Oct. 2026. [Online]. Available: <https://x.com/karpathy/status/1886192184808149383>

- [37] S. Willison. “Not all AI-assisted programming is vibe coding (but vibe coding rocks)”, Simon Willison’s Weblog, Accessed: 12th Oct. 2026. [Online]. Available: <https://simonwillison.net/2025/Mar/19/vibe-coding/>
- [38] V. Chen, A. Talwalkar, R. Brennan and G. Neubig, “Code with me or for me? how increasing AI automation transforms developer workflows”, in *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*, ACM, Apr. 2026. DOI: 10.1145/3772318.3790850
- [39] L. Hallnäs and J. Redström, “Slow technology – designing for reflection”, *Personal and Ubiquitous Computing*, vol. 5, no. 3, Aug. 2001. DOI: 10.1007/PL000000019
- [40] Z. Chen and R. Schmidt, “Exploring a behavioral model of “positive friction” in human–AI interaction”, in *Design, User Experience, and Usability (HCII 2024)*, ser. Lecture Notes in Computer Science, Springer, Jun. 2024. DOI: 10.1007/978-3-031-61353-1_1
- [41] Agent Skills. “Agent skills overview”, Accessed: 12th Oct. 2026. [Online]. Available: <https://agentskills.io/home>
- [42] K. Chaidemenos, *agentic-pm*, version 1.0.1, Apr. 2026. Accessed: 12th Oct. 2026. [Online]. Available: <https://www.npmjs.com/package/agentic-pm/v/1.0.1>
- [43] K. Chaidemenos, *APM Auto*, 2026. Accessed: 12th Oct. 2026. [Online]. Available: <https://github.com/sdi2200262/apm-auto>
- [44] K. Chaidemenos, *APM Semi*, 2026. Accessed: 12th Oct. 2026. [Online]. Available: <https://github.com/sdi2200262/apm-semi>
- [45] K. Chaidemenos, *Agentic Project Management (APM)*, version 1.0.1, Apr. 2026. Accessed: 12th Oct. 2026. [Online]. Available: <https://github.com/sdi2200262/agentic-project-management/releases/tag/v1.0.1>
- [46] GitHub, *Spec Kit*, version 0.8.5–0.8.13, 2026. Accessed: 12th Oct. 2026. [Online]. Available: <https://github.com/github/spec-kit/releases>
- [47] K. Chaidemenos, *eClass-mcp-server: A Model Context Protocol (MCP) server for the Open eClass platform*, version dbd2d16, 2026. Accessed: 12th Oct. 2026. [Online]. Available: <https://github.com/sdi2200262/eClass-mcp-server>
- [48] Model Context Protocol. “Model Context Protocol specification, revision 2025-06-18”, Accessed: 12th Oct. 2026. [Online]. Available: <https://modelcontextprotocol.io/specification/2025-06-18>
- [49] GUnet, *Open eClass*, version 4.3.3, 2026. Accessed: 12th Oct. 2026. [Online]. Available: https://github.com/gunet/openeclass/releases/tag/Release_4.3.3
- [50] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals”, *Soviet Physics Doklady*, vol. 10, no. 8, 1966, Translated from Doklady Akademii Nauk SSSR, 163(4):845–848, 1965.
- [51] R. A. Wagner and M. J. Fischer, “The string-to-string correction problem”, *Journal of the ACM*, vol. 21, no. 1, Jan. 1974. DOI: 10.1145/321796.321811
- [52] G. W. Corder and D. I. Foreman, *Nonparametric Statistics: A Step-by-Step Approach*, 2nd ed. Hoboken, NJ: John Wiley & Sons, 2014, ISBN: 978-1-118-84031-3.
- [53] J. L. Hodges and E. L. Lehmann, “Estimates of location based on rank tests”, *The Annals of Mathematical Statistics*, vol. 34, no. 2, Jun. 1963. DOI: 10.1214/aoms/1177704172
- [54] F. Wilcoxon, “Individual comparisons by ranking methods”, *Biometrics Bulletin*, vol. 1, no. 6, Dec. 1945. DOI: 10.2307/3001968
- [55] M. Hollander, D. A. Wolfe and E. Chicken, *Nonparametric Statistical Methods*, 3rd ed. Hoboken, NJ: John Wiley & Sons, 2013, ISBN: 978-0-470-38737-5.

- [56] Anthropic. "Model configuration", Accessed: 12th Oct. 2026. [Online]. Available: <https://code.claude.com/docs/en/model-config>
- [57] T. Bray. "The JavaScript object notation (JSON) data interchange format", Internet Engineering Task Force, Accessed: 12th Oct. 2026. [Online]. Available: <https://www.rfc-editor.org/info/rfc8259>
- [58] M. Borg, D. Hewett, N. Hagatulah et al., "Echoes of AI: Investigating the downstream effects of AI assistants on software maintainability", *Empirical Software Engineering*, vol. 31, no. 6, Nov. 2026, Art. no. 161. DOI: 10.1007/s10664-026-10889-1
- [59] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs", *Cognitive Psychology*, vol. 19, no. 3, Jul. 1987. DOI: 10.1016/0010-0285(87)90007-7