UiA University of Agder

Bachelor

Evaluating Spec-Driven Workflows in Agentic Code Generation

A Controlled Comparison of Vibe, Self-Spec, Iterative, and Spec Kit Workflows for Greenfield Backend Applications

ELIAS SKAUGE ERIKSEN,
MIKKEL JANSEN SKRETTEBERG,
WILLIAM HØYVIK EIKESKOG

Supervisor

Arne Wiklund

University of Agder, 2026
Department of Information and Communication Technology
Faculty of Engineering and Science

Mandatory declaration

The individual student is responsible for knowing which aids are permitted, how they may be used, and which rules apply to sources and citations. This declaration is intended to make the author team aware of that responsibility and of the consequences of academic misconduct. Failure to submit the declaration does not remove that responsibility.

1.	I / we hereby declare that this submission is my / our own work and that I / we have not used other sources or received other assistance than what is explicitly stated in the report.	YES
2.	I / we further declare that this submission: • has not been used for another examination at another department, university, or university college in Norway or abroad; • does not refer to the work of others without giving credit; • does not refer to my / our own previous work without giving credit; • has all references listed in the bibliography; • is not a copy, duplicate, or transcription of another person's work or submission.	YES
3.	I / we am / are aware that breach of the above is regarded as academic misconduct and may lead to annulment of the examination and exclusion from universities and university colleges in Norway, cf. the Universities and University Colleges Act §§4-7 and 4-8 and the Examination Regulations §31.	YES
4.	I / we am / are aware that all submitted assignments may be checked for plagiarism.	YES
5.	I / we am / are aware that the University of Agder will handle all cases where there is suspicion of academic misconduct according to the university's procedures for such cases.	YES
6.	I / we have familiarized myself / ourselves with the rules and guidelines for the use of sources and references on the university library's webpages.	YES

Publication agreement

Power of attorney for electronic publication of the thesis. The author(s) hold copyright to the thesis. This includes the exclusive right to make the work available to the public (Copyright Act, §2). All theses that meet the criteria may be registered and published in Brage Aura and on UiA webpages with the author(s)' approval. Theses that are exempt from public disclosure or contain confidential / classified information will not be published.

I / we hereby grant the University of Agder a royalty-free right to make this thesis available for electronic publication:	YES
Is the thesis embargoed (confidential)?	NO
If yes: may the thesis be published when the embargo period ends?	X
Is the thesis exempt from public disclosure?	NO

Preface

This thesis was written as part of the bachelor program in Software Engineering, Information and Communication Technology at the University of Agder.

We would like to thank our supervisor, **Arne Wiklund**, for his guidance, feedback, and support throughout the project. His input helped us refine the scope of the thesis, keep the work focused in the correct direction and give valuable input throughout the whole process of the thesis.

We would also like to thank each other for the collaboration throughout the project period. The work required shared effort across planning, implementation, experimentation, analysis, and writing, and this thesis is the result of that joint contribution.

Throughout the thesis, we have used LLM agents as supporting tools in the work process. We used them for planning, structuring ideas, discussing possible approaches, debugging code, improving wording, and acting as coding and research partners. In addition, we used them directly to generate text and code which we then reviewed, edited, tested, and adapted to fit this thesis. The final content, implementation choices, analysis, and conclusions are our own work and responsibility.

Abstract

AI coding agents can generate backend applications from a Software Requirements Specification (SRS), but it is unclear whether adding more specification scaffolding between the SRS and implementation improves the result. This thesis evaluates whether specification scaffolding improves functional correctness, code organization and file structure, and how it affects token use, cost, and duration[1]. Four workflows[2] are compared; Vibe, Self-Spec, Iterative, and Spec Kit. They represent different amounts of Specification-Driven Development (SDD)-style scaffolding between the SRS and the implementation. With varying use of specification, each workflow is tested on two different backend API applications.

This experiment is designed as a controlled workflow benchmark for both applications. Only the workflow changes between runs. The SRS, API contract, prompt preamble[3] and LLM model stay the same. The SDD-style workflows for this experiment are fully automated. No human reviews or corrects the generated scaffolding before implementation. The results therefore evaluate automated SDD scaffolding, not a human-guided SDD process.

The four workflows produce 240 backend applications in total. All applications are tested using a test suite, which tests all functional requirements (FR) mentioned in the SRS. Token use, estimated model cost and duration is extracted from each separate workflow implementation. The best performing runs from each workflow are manually reviewed on code organization and file structure.

Results show that adding more automated scaffolding before implementation did not consistently improve FR coverage. The two workflows with the least SDD-style scaffolding before implementation had the strongest FR coverage and the lowest costs, but their best runs were weaker in code organization and file structure.

Workflows with more preprocessing used more time and tokens without consistently increasing average FR coverage. However, among the selected runs with the highest FR coverage, the workflows with more automated specification scaffolding showed clearer code organization and file structure. For new backend API applications with a clear SRS and a fixed API contract, more automated SDD-style scaffolding is therefore not automatically better.

[1] Token use refers to model input and output text processed during generation. Duration refers to wall-clock time for a workflow run.

[2] The workflow names are defined for this experiment. They are not established categories in literature.

[3] A shared instruction block given before each workflow prompt to keep general constraints equal across runs.

Contents

List of Figures

List of Tables

Definitions

This chapter gives short working definitions for terms that are used throughout this thesis. The definitions explain how the terms are used in this report. More detailed background and sources are given in the theoretical background chapter.

Large Language Model (LLM) An LLM is a model that generates text from a written prompt. In this thesis, the term is mainly used for models that can also help generate and reason about code.

Agentic code generation Agentic code generation is code generation performed by an AI coding agent: an LLM-based tool that can work inside a software project by inspecting files, editing code, running commands, observing results, and continuing from those results, instead of only returning a code snippet in a chat response.

Software Requirements Specification (SRS) An SRS is a document that describes what a system should do. In this thesis, each SRS is used as the main description of the required backend behavior.

Specification-Driven Development (SDD) SDD is a development approach where specifications guide implementation. In this thesis, SDD-style workflows use generated specifications, plans, or task lists before or during code generation.

Scaffolding Scaffolding means supporting material created before implementation, such as a specification, plan, task list, or feature breakdown. The scaffolding is meant to guide the agent while it writes the backend.

Artifact An artifact is a document, file, or output that is produced during the planning or implementation phase.

Workflow A workflow is the ordered process used to move from the task material to a generated backend. The workflow is the main experimental condition in this thesis.

Vibe Vibe is the direct baseline workflow in this experiment. The agent receives the task material and moves straight to implementation without first creating a plan, specification, or task list.

Self-Spec Self-Spec is the workflow where the agent first creates its own plan and then implements the backend in a later step. The plan is not reviewed or corrected by a human before implementation.

Iterative Iterative is the workflow where the agent first divides the application into smaller feature blocks. The backend is then implemented through several later steps, one feature block at a time.

Spec Kit Spec Kit is the most structured workflow used in the experiment. It separates the process into specification, planning, task creation, and implementation phases.

Functional requirement (FR) A functional requirement is a required behavior that the backend should provide. In the experiment, the SRS for each application is divided into a set of functional requirements that can be evaluated.

FR coverage FR coverage describes how many of the functional requirements were satisfied by a generated backend. It is the main functional result measure in this thesis. Each

functional requirement is classified as *met* when it is fully satisfied, *partial* when it is only partly satisfied, *blocked* when it cannot be tested because another missing or faulty requirement prevents the test from reaching it, and *missed* when it is not satisfied.

Strict coverage Strict coverage counts only functional requirements that were fully met. Requirements that were partial, blocked, or missed do not count toward this score.

Lenient coverage Lenient coverage gives full credit for met requirements and half credit for partial requirements. Blocked and missed requirements do not count toward this score.

Scenario harness A scenario harness is the test program used to evaluate a generated backend. It starts the backend, calls its API, follows scenario flows, and records the result for each functional requirement.

Reasoning effort Reasoning effort is a model setting that controls how much internal reasoning the model may use before it answers or acts.

Token use Token use describes how much text the model processes and generates during a run. It is used as a practical measure of model usage and estimated cost.

Prompt preamble The prompt preamble is the shared instruction block added to workflow prompts in the experiment. It keeps general constraints the same across workflows.

Brownfield Brownfield means working inside an existing codebase rather than starting a new project from an empty repository. In a brownfield setting, new changes must fit the existing architecture, files, conventions, dependencies, and behavior of the project.

Chapter 1

Introduction

1.1 Background and Motivation

AI assisted software development has moved from isolated snippet generation and code completion toward coding agents that can work across a software project [1], [2]. These agents can create and modify files, install dependencies, run commands, inspect failures, and revise their own output. This makes them relevant for generating larger parts of an application, not only assisting with small local edits.

This shift makes the instructions given to the agent more important. When a coding agent is asked to implement a complete backend API, the result depends not only on its ability to write code, but also on how clearly the requirements and constraints are presented before implementation starts.

One possible answer is to add more structure before implementation. SDD represents this idea. Piskala [3] argues that specifications can help AI coding assistants by giving them context they would otherwise have to guess. However, he also notes that more specification is not always useful; for simple or short-lived projects, extra specification may become overhead rather than valuable guidance.

Traditional SDD workflows usually include human review and refinement of specifications, plans, and tasks. This thesis does not add that human step. The workflows start from the same SRS and fixed API contract, and the agent itself transforms this input into specifications, plans, or task breakdowns before implementation. This scaffolding may help the agent work through the requirements more systematically, but it may also repeat misunderstandings or introduce new ones.

More scaffolding also has a cost. Agentic coding can consume substantially more tokens than simpler code interactions. Token use can vary greatly between runs, and higher token use does not necessarily translate into higher accuracy [4]. If automated SDD-style workflows require more prompts and planning steps, their value must therefore be judged against token use, duration, and estimated model cost.

An initial motivation for the study was the expectation that more structured specification work could help coding agents produce better implementations. Since SDD-style scaffolding gives the agent more guidance before implementation, it appeared plausible that this could improve the output drastically. The experiment was therefore designed to test this assumption rather than to assume it as given.

This thesis studies whether automated SDD-style scaffolding helps when AI coding agents generate backend APIs, or whether a more direct workflow is enough. The goal is to compare functional correctness, code structure, token use, cost, and duration.

1.2 Research Problem

The problem is that existing claims about SDD do not clearly show whether automated scaffolding improves agent-generated backend applications in greenfield projects. The shift therefore becomes: should developers spend time and tokens creating extra scaffolding before implementation, or is a direct prompt to implementation enough?

RQ1 - Functional outcome and token use. Which workflow produces the highest functional requirement coverage, and how much token use does it require?

RQ2 - Structural outcome. How do the generated implementations differ in code structure?

RQ1 is whether the workflows differ in what the generated backend can do. The main measure is functional requirement coverage: how many requirements from the SRS are correctly implemented. Token use is reported with the result because a workflow that improves coverage may still be less practical if it requires much more model usage.

RQ2 answers whether the workflows differ in how the generated backend is organized. This is not measured as a numeric score. Instead, selected implementations are reviewed to describe differences in structure, such as how the code is divided into files and components and where responsibilities are placed.

1.3 Report Structure

Chapter 2 (Theory) introduces the concepts used in the thesis, including AI coding agents, agent harnesses, SDD-style scaffolding, fixed API contracts, reasoning effort, and related empirical work.

Chapter 3 (Method) explains how the experiment is designed. It describes the backend tasks, the four workflows, the run procedure, the model settings, the evaluation method, and the analysis procedure.

Chapter 4 (Implementation) describes how the benchmark was implemented in practice. It covers the runner, task material, API contracts, run directories, and scenario test harnesses.

Chapter 5 (Results) presents the results. It reports functional requirement coverage, token use, duration, reasoning-effort effects, and the manual code-structure observations.

Chapter 6 (Discussion) discusses what the results mean and where the study is limited.

Chapter 7 (Societal Perspective) discusses broader implications of AI-generated software.

Chapter 8 (Conclusion) summarizes the main conclusions and suggests future work.

Chapter 2

Theoretical Background

When an agent is requested to build an application from an SRS, it raises an important question; where are the important decisions actually made? Some decisions may be fixed by the requirements or by intermediate artifacts, while others are left for the agent to settle during implementation. This chapter introduces the concepts needed to compare the workflows used in this experiment; direct prompting, SDD, workflow structure, fixed interface contracts, task complexity, and agent harnesses. Together, they explain how requirements are turned into implementation steps and why different workflow structures may lead to different results. These concepts provide the conceptual basis and vocabulary for discussing how the workflows differ and for interpreting the results later in the thesis.

2.1 AI-Assisted Software Development

AI-assisted software development has changed quickly in how developers interact with LLMs. Rather than treating all AI coding tools as one category, this thesis uses three overlapping waves to describe the shift; chat-based interfaces, editor-based code completion, and agentic coding. The waves are not strict historical periods, since all of them still exist, but they help separate different levels of integration.

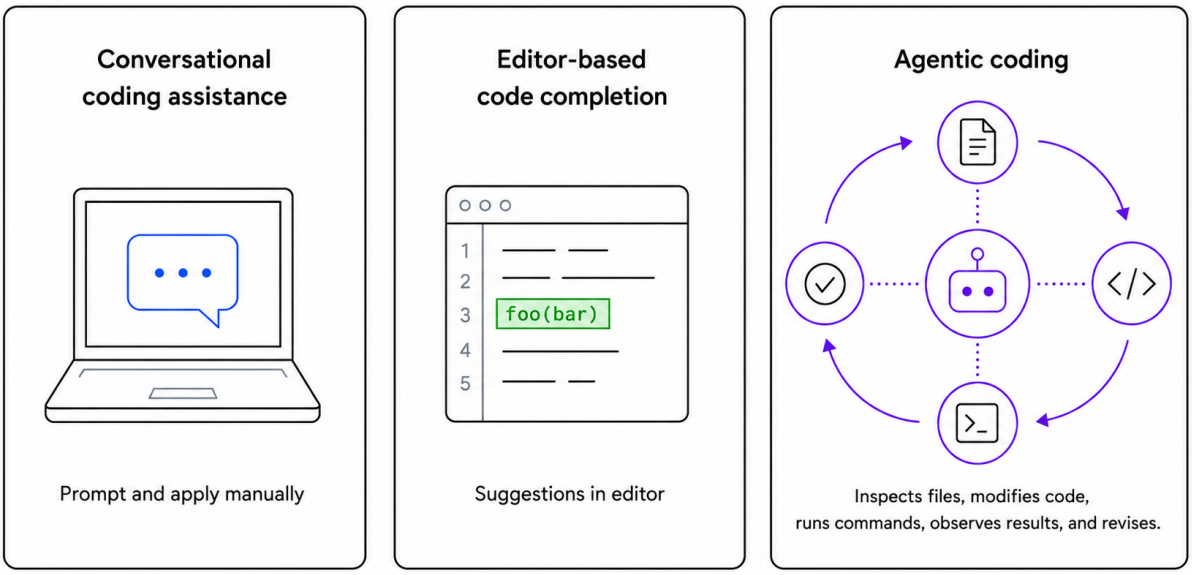

Figure 2.1: The three AI coding waves visualized

Conversational coding assistance describes the use of general-purpose LLMs through chat interfaces [5]. The developer asks questions, pastes code or error messages into the conversation, and transfers changes, from the LLM, back into the project manually. This makes the model useful for explanation, debugging, and small code generation tasks, but the developer remains responsible for carrying context and applying changes.

Editor-based code completion moves the model closer to the development environment. Instead of working in a separate chat window, the assistant suggests code inside the editor while the developer writes. This gives the model more local context and reduces manual copying, but the interaction is still centered on suggestions that the developer accepts, rejects, or edits.

Agentic coding gives the system a larger role in the development loop. An agent can inspect files, modify the repository, run commands, observe failures, and revise its own output [1], [6]. The developer can therefore give a broader implementation goal, while the agent performs several steps toward that goal using the tools available to it.

This thesis focuses only on agentic coding. The experiment does not evaluate chat-based assistance or editor-based completion, because those modes still rely on the developer to carry most of the implementation process.

The improved capability of agentic coding does not remove the need for clear requirements. Software engineering has long treated requirements and specifications as ways to make intended behavior explicit before implementation [7]. Different readers may understand the same requirement in different ways, which can lead to different implementations. That concern becomes clearer when an agent is making system-level decisions about requirements such as persistence, access control, state transitions, error handling, and other domain rules rather than only filling in small coding snippets.

In this broader setting, success is not only about whether the generated system matches the required behavior. It is also about whether the surrounding workflow helps the agent produce code with a clear structure that can be understood, changed, and maintained.

2.2 Vibe Coding or Direct Prompting

Direct prompting is a way of using a coding agent with little or no scaffolding. The developer gives the agent a task description and asks it to implement the system without a separate planning, specification, or task-decomposition step. A direct prompt can still include detailed requirements, constraints, or an API contract; the key point is that the agent moves directly from those inputs to code.

In practice, this style is often associated with the informal term "vibe coding" [8]. The term is not a formal software engineering method, but it has recently become a common label for a fast way of going from requirements to implementation.

The appeal of direct prompting is speed. It reduces the amount of work done before implementation and avoids spending tokens on intermediate artifacts. The risk is that the agent must make more decisions while it is already writing the code, where those decisions are harder to inspect.

Piskala uses this contrast to explain the motivation behind specification-first workflows in AI-assisted development: vague prompts force the model to guess, while clearer specifications can narrow the space of reasonable interpretations [3]. In a backend API, these choices can affect requirements as role separation, state transitions, error behavior, persistence, and the handling of edge cases.

2.3 Specification-Driven Development

SDD starts from the idea that code generation should be guided by a specification, not only by an implementation prompt. In AI-assisted development, this means giving the coding agent a more structured description of what to build before or during implementation [3].

Recent SDD discussions describe a spectrum of use [3]. Piskala distinguishes between spec-first, spec-anchored, and spec-as-source approaches.

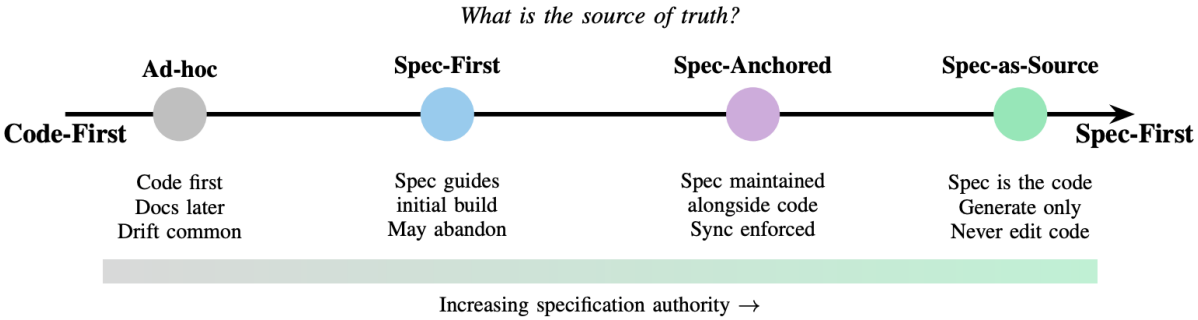

Figure 2.2: Specification spectrum, reproduced from Piskala [3].

At the spec-first end, the specification is written before the code and used to guide the first implementation. The code may later evolve without the specification being kept fully synchronized.

Spec-anchored development keeps the specification alongside the code as a reference for later changes. Spec-as-source goes further: the specification is the main source of truth, and code is generated from it rather than edited directly.

Piskala places tools such as Spec Kit within this broader specification-first tradition, alongside older methods such as BDD, TDD, API-first design, and contract testing [3]. Some sources argue that clearer requirements and staged planning can help AI coding tools stay closer to the intended behavior [9], [10].

This thesis focuses only on spec-first scaffolding. The generated specifications, plans, and task lists guide the initial implementation, but they are not maintained after the code is generated. The experiment therefore does not evaluate spec-anchored or spec-as-source development.

2.3.1 The SDD Loop

Piskala describes SDD as a loop with four main stages: specify, plan, implement, and validate [3]. **The specify stage** defines what should be built. **The plan stage** describes how it should be built. **The implement stage** turns the specification and plan into code. **The validate stage** checks whether the result matches the intended behavior.

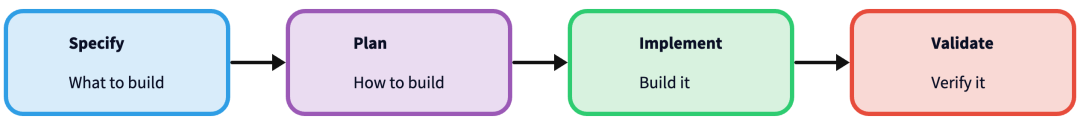

Figure 2.3: Simplified SDD loop based on Piskala [3].

The loop also shows that SDD is not only about writing a specification once. In a full SDD process, the specification can be reviewed, used to guide implementation, checked against the result, and revised when needed. This is what separates a maintained specification process from a one-time planning step.

The loop is useful because it separates specification work into clearer stages: what to build, how to build it, building it, and checking the result. This makes it easier to discuss later workflow designs without treating all scaffolding as the same thing.

2.4 Fixed Interface Contracts

Generated backends are easier to compare when they expose the same API shape. Without a fixed contract, two systems may implement similar behavior but use different endpoints, fields, or response formats. A fixed interface contract gives the tests one stable way to call every system.

OpenAPI is a common way to describe an API in a machine-readable format. It can define which endpoints exist, which methods they support, and what request and response data should look like [11]. API-first and contract-testing approaches use this kind of contract before or during implementation to keep the system aligned with the expected interface [3], [11].

2.5 Task Complexity and Coupling

Scaffolding for some implementations may matter more for some applications than for others. Some applications are difficult because they are wide; they include many roles, entities, and operations that all need to be covered. Other applications are difficult because their rules are coupled[1].

These two kinds of difficulty put different pressure on a coding agent. A wide application mainly tests whether the agent remembers and implements all required behavior. A coupled application mainly tests whether the agent handles order, state, and constraints correctly over time. A missed validation rule may affect one flow, while a wrong state transition can affect several later actions.

More structure may help when many decisions need to stay consistent across the implementation. Specifications, plans, and decompositions can make related requirements easier to keep visible while the agent works [3], [12], [13]. At the same time, extra structure may add overhead when the required behavior is simple or already clear.

This distinction is used later when selecting and interpreting the benchmark applications. The goal is not only to compare workflows on one kind of application, but to see whether their behavior changes across different forms of application complexity.

2.6 Agent Harnesses

In agentic coding, an LLM operates through a harness. The coding harness is the surrounding software that gives the model access to the source code, project files, tools, and execution environment [14]. A simple way to describe this relationship is that the LLM acts as the brain, while the harness acts as the body that lets it observe the environment and perform actions.

The harness provides the model with context, instructions, and available tool definitions. The model can then respond directly or request a tool action, such as reading a file, editing code, or running a command. If the action is allowed, the harness runs it and returns the output to the model as new context. This creates an iterative loop that continues until the task is completed, blocked, or stopped.

This means that the final code depends not only on the LLM, but also on what the harness allows the model to see and do. OpenAI describes Codex CLI as a coding agent that runs from the

[1] Coupling means that one part of the application affects what is correct in another part.

terminal and can read, change, and run code in the selected directory [15]. Different harnesses can therefore affect the result by providing different tools, context, permissions, and feedback.

2.6.1 Tokens, Context, and Cost

LLM systems do not process text as words. Before text reaches the model, it is split into smaller units called tokens. One token can be a whole word, part of a word, or punctuation, depending on the text and model [16]. This matters for agentic coding because the model does not only receive the sentence written. The agent system can also send project instructions, earlier messages, tool results, errors, and file contents to the model. All of this text is counted as tokens.

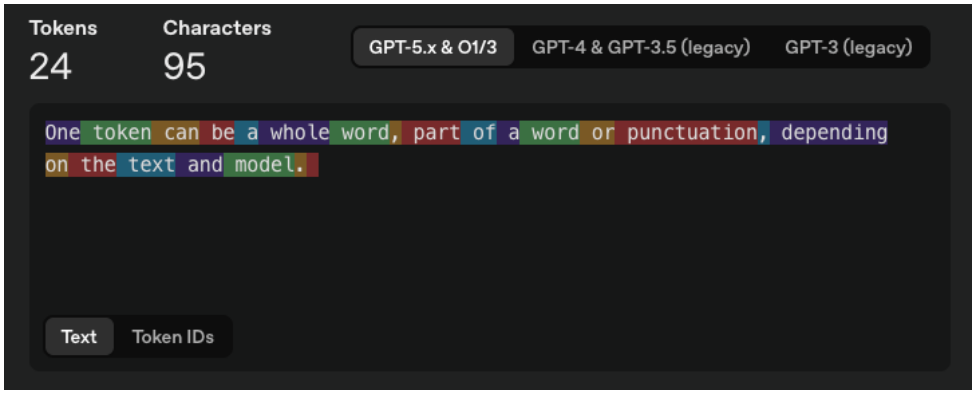

Figure 2.4: OpenAI's tokenizer tool showing how input text is split into tokens [16].

Tokens are counted in two directions. Input tokens are the text sent to the model, and output tokens are the text the model generates. When a generation process has more steps, the model may write more intermediate text and later receive more previous context. Token use therefore shows how much text the model had to process and generate.

Prompt caching changes how token cost is counted. If the same input appears in several model requests, the provider may reuse parts of it from a cache instead of charging it as normal input. OpenAI [17] describes prompt caching as an automatic feature on supported models, where repeated parts of the input, such as earlier messages or tool definitions, can be cached and reported separately in the usage data.

Cached tokens are still part of the context the model can use, but they may be priced differently from other input tokens [18]. Token use is therefore reported in three parts: input tokens, cached input tokens, and output tokens. Input tokens are sent to the model, cached input tokens are the part of the input that is reused from cache, and output tokens are generated by the model.

2.6.2 Reasoning Effort

Reasoning means that the model spends extra time working through a task before it answers or acts. For a coding agent, this can involve planning steps, checking constraints, or adjusting its approach before writing code. The full internal process is not visible to the user, but prior work shows that intermediate reasoning can help on complex tasks [19].

For coding agents, reasoning can matter because software tasks often involve requirements, design choices, dependencies, and failing tests. Some modern LLM systems expose reasoning-effort settings, which control how much thinking the model may use before producing output. Lower reasoning effort is usually faster and cheaper, while higher reasoning effort gives the model more room to work through difficult decisions.

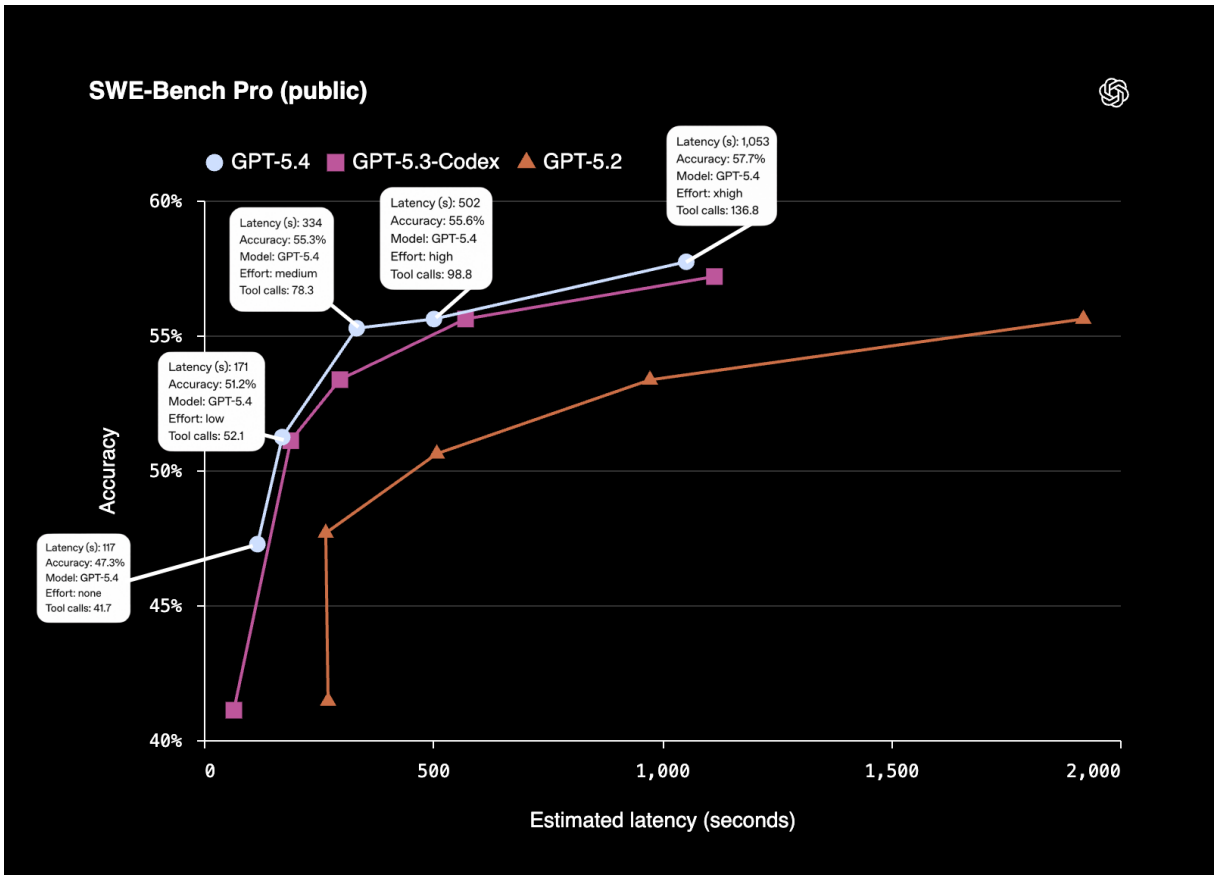

Figure 2.5: OpenAI-reported SWE-Bench Pro accuracy against estimated latency when reasoning-effort settings are swept from none to xhigh. Source: OpenAI [20].

Higher reasoning effort is therefore a tradeoff. SWE-Lancer [21] reports better results with higher reasoning effort on real-world freelance software engineering tasks, but also higher computation use. Other work [22] shows that reasoning-heavy agents can overthink and spend too much time analyzing instead of acting. Reasoning effort is therefore different from workflow structure: it changes how each model call is handled, while a workflow changes the steps and context around those calls.

2.6.3 Planning Mode

Planning before implementation means that the agent first describes how it will approach the task before writing the code. The plan may identify files to change, features to implement, dependencies required, or risks that need attention. In code generation research, this is often connected to decomposition: breaking a larger task into smaller parts before implementation [12], [13].

Planning can make the agent's intended approach easier to inspect. It can also help keep related requirements visible before code is written. However, a plan is not a guarantee of correctness. If the plan misunderstands the task, the later implementation may follow that misunderstanding.

What the agent can plan depends on the context it receives, the tools it can use, and whether it is allowed to inspect files or run commands.

Plan Mode Under the Hood

In Codex, plan mode is exposed as the `/plan` command. The Codex documentation describes it as a way to switch the active conversation into plan mode and ask Codex to propose an execution plan before implementation starts [23].

Plan mode should therefore be understood as a harness-level interaction mode, not as a separate model. It changes what the agent is being requested to produce: a plan rather than direct code changes. What the agent is allowed to do during that turn also depends on the surrounding harness settings, such as approval rules and sandbox configuration [24], [25].

2.6.4 Spec Kit

Spec Kit is relevant because it turns specification-first development into a concrete workflow for AI coding agents. Instead of treating the specification as background documentation, Spec Kit makes it part of the development path: the agent first clarifies what should be built, then derives a plan and tasks before implementation. Piskala describes this kind of approach as closely related to SDD because the specification guides later development steps [3], [9].

This makes Spec Kit a useful reference point for this thesis. It shows how SDD ideas can be adapted to agentic coding, where intermediate artifcats are used to shape the next model call rather than only to communicate between human developers.

Spec Kit Under the Hood

Under the hood, Spec Kit is mainly a prompt-and-file workflow. It provides structured commands for each phase, and each command tells the coding agent what input to read, what decisions to make, and what document to write [26]. The resulting files form a chain: specification informs plan, plan informs task list, and task list informs implementation.

2.7 Prior Empirical Work

Piskala provides the main conceptual starting point for this thesis. He argues that SDD can make AI-assisted development more reliable by giving the coding agent clearer specifications, plans, and contracts to work from [3]. However, the setting he describes is closer to human-guided or brownfield development, where specifications can be reviewed, refined, and connected to an existing system. This thesis studies a narrower case: greenfield backend generation where the agent creates any intermediate scaffolding automatically, without human review between phases.

Empirical work on self-planning supports the idea that intermediate planning can affect code generation. Jiang et al. [12] introduce a self-planning approach where the model first decomposes a programming task before generating code. Their results suggest that planning can help LLMs handle more complex programming intent. Taghavi and Bhavani [27] study a more direct SDD-style agent workflow through Spec Kit Agents. Feng et al. [28] study structured spec-driven engineering for repository-level generation with LLMs. Hill [29] gives a more skeptical result. In a large-scale study of pull requests across open-source repositories, the study finds no support for several common claims that SDD reduces defects, prevents rework, or improves code quality.

Together, these studies show mixed evidence. Much of the field studied is still new, and the exisiting work usually study either local planning techniques, tructured SDD in existing repositories, or broad defect outcomes. There is still limited empirical evidence on when SDD-style scaffolding is worth the extra process in AI-assisted development.

Chapter 3

Method

3.1 Research Design

This study is a controlled comparison of backend-generation workflows. The aim is to compare how different workflow structures affect the generated backend applications while keeping the task material, interface contract, execution setup, and evaluation procedure fixed.

The design follows the research questions introduced in Chapter 1. RQ1 concerns functional-requirement coverage and token use, while RQ2 concerns differences in code structure. The research design therefore compares both the observable behavior of the generated applications and the structure of selected implementations.

The controlled factor in the study is the workflow. The workflows represent different ways of moving from requirements to implementation, ranging from direct prompting to more specification-driven approaches. Since LLM-based generation is non-deterministic, each condition is repeated ten times with the same input [30]. This does not eliminate randomness, but is treated as sufficient for comparing workflow patterns within the scope of this thesis.

3.2 Task Material

The benchmark uses two backend API applications to reduce the risk that the results only reflect one application type. The applications were chosen to differ in complexity. One application is closer to a CRUD-style administrative system, while the other requires more domain logic, and is more complex.

For each application, the agent receives the same material: an SRS, a fixed API contract, and a shared prompt preamble. The SRS describes the expected behavior, while the fixed API contract defines how the generated backend must be called by the evaluator. The preamble contains common instructions about the working directory, local execution and the use of the API contract. These inputs stay fixed across workflows so that differences between runs mainly come from the workflow structure.

The fixed API contract is used to make all generated applications evaluable through the same interface. Without this contract, different agents could choose different endpoint names or payload shapes, which would make comparison harder. The contract does not guarantee correctness by itself. An application can expose the expected endpoint and still fail if the behavior behind it does not satisfy the SRS.

3.2.1 Hospital Management Application

The hospital management application was included as the lower-complexity application. The SRS was provided through the university and describes a backend for managing common hospital processes. It includes patient, doctor, and administrator flows, with behavior for registration, appointments, consultations, medical records, billing, payments, reports, and alerts.

The application has broad coverage across several roles and resources, but most of the behavior is administrative. The application mainly needs to create, retrieve, update, and delete records in a consistent way. For this reason, the hospital application is useful as a CRUD-oriented backend application.

3.2.2 Chess Tournament Application

The chess tournament application was chosen as the more complex application. It describes a backend for running a Swiss-system chess tournament from registration to final results. The system must handle player registration, round creation, pairings, byes, result reporting, standings, tiebreaks, crosstables, reports, and Elo rating updates [31], [32].

Compared with the hospital application, the chess application has fewer administrative flows but more rule-dependent behavior. Pairings must follow Swiss-system constraints, each round depends on previous results, and standings must be updated as the tournament progresses. This makes the chess application useful for testing workflows on connected domain rules, not only data management.

3.3 Experimental Conditions

The workflow is the experimental condition. All workflows receive the same task inputs, but they organize the path from requirements to implementation differently.

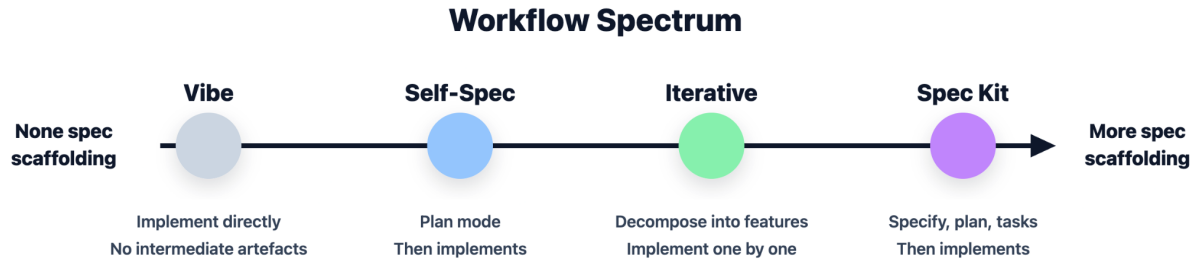

Figure 3.1: Workflow spectrum used in the experiment. The conditions differ by how much specification scaffolding they add before implementation.

The four workflows can be read as a spectrum of increasing specification scaffolding. Vibe uses no intermediate artifacts, while Spec Kit uses separate specification, planning, task, and implementation phases. Self-Spec and Iterative sit between these endpoints.

3.3.1 Workflow 1: Vibe

Vibe is included as the direct baseline. It shows what the agent can produce when it moves straight from the task material to implementation. The agent receives the SRS, the fixed API contract, and the shared preamble in one implementation prompt. It does not first create a plan, specification, task list, or feature decomposition.

In this experiment, Vibe does not mean that the agent receives vague or minimal input. It receives the same written task material as the other workflows. The workflow is called Vibe because it skips intermediate scaffolding and goes directly to implementation. This differs from more informal uses of "vibe coding", as discussed in Section 2.2.

3.3.2 Workflow 2: Self-Spec

Self-Spec adds one planning step before implementation. It is included because plan mode is a common feature in coding agent tools. The workflow adds structure without using a full specification workflow, as discussed in Section 2.6.3.

The workflow has two Codex calls. In the first call, the agent is started in `/plan` mode and asked to create an implementation plan from the task material. This mode is read-only, so the agent can reason about the approach without changing the project files. The second call resumes the same Codex session and asks the agent to implement the plan. The plan from the first call remains in the conversation to guide the implementation.

The workflow is called Self-Spec, instead of plan mode, because the planning step is produced by the agent itself. No human reviews or revises the plan before implementation.

3.3.3 Workflow 3: Iterative

Iterative is included as a middle condition between Self-Spec and Spec Kit. It adds task decomposition, but not a separate specification or planning phase. The purpose is to test whether smaller implementation steps help compared with one larger implementation prompt.

The first Codex call asks the agent to split the SRS into feature-sized blocks and write them to a text file. The runner then resumes the same Codex session once for each generated feature block. In each call, the agent is asked to implement one feature block before moving to the next.

This means that the number of implementation calls depends on the agent's own decomposition for that benchmark run. Each feature call has access to the previous conversation and the code already produced. The workflow therefore tests a sequential implementation process where the same application is built in smaller steps.

3.3.4 Workflow 4: Spec Kit

Spec Kit is included because it gives a practical implementation of specification-first development for AI coding agents [9]. Piskala mentions Spec Kit directly when discussing SDD in AI-assisted development [3]. It is useful for this experiment because it separates the process into specification, planning, task separation, and implementation.

Before generation starts, the runner initializes a local Spec Kit project for Codex. This makes the Spec Kit commands and templates available for Codex. The workflow then uses four Codex calls in the same session:

- `/speckit.specify`
- `/speckit.plan`
- `/speckit.tasks`
- `/speckit.implement`

Each call is asked to complete only its own phase before stopping. The earlier phases create specification, plan, and task documents, which remain in the run directory and are available to later phases.

Spec Kit is fully automated in this experiment. No human reviews or corrects the generated specification, plan, or task list before implementation. The result therefore tests automated specification-first scaffolding, not an ideal human-reviewed SDD process.

3.4 Run Procedure

Each run must be executed in an isolated environment. This means that a run starts from only the fixed task material for that condition and must not have access to files, generated code, or conversation history from earlier runs.

This isolation is necessary for the comparison to be controlled. If one run could reuse information from another, the result might reflect previous generation attempts rather than the workflow being tested. The run procedure therefore treats each generated backend as an independent output from the same type of input.

After a run has completed, the generated backend is evaluated separately through the fixed test procedure. This separates generation from evaluation: the workflow creates the application, while the evaluation procedure measures the observable result afterwards.

3.4.1 Model and Reasoning Effort

The experiment uses GPT-5.4 for all runs. At the time of the experiment, GPT-5.4 represented as one of the current frontier OpenAI models. Keeping the model fixed across all conditions makes the workflow and reasoning-effort settings the relevant variables, rather than differences between model providers or model families.

The model is run through Codex, an agentic coding harness for repository-level tasks such as file editing, command execution, and iterative implementation [33]. The Codex harness is open source, which makes the execution setup inspectable and better suited for a controlled benchmark than a fully closed coding interface [34].

Reasoning effort is included as a separate run parameter, following Section 2.6.2. OpenAI reports public SWE-Bench Pro, Figure 2.5, results across reasoning-effort levels, making it relevant to include in a software-engineering benchmark. The experiment uses:

- `none`,
- `low`,
- `medium`

reasoning effort to compare workflow behavior under different reasoning budgets.

3.5 Evaluation

After generation, each backend is evaluated by an automated scenario harness. The scenario harness starts the generated application and interacts with it through the fixed API contract. The evaluation therefore measures observable API behavior, not whether the code has a specific internal structure.

The tests are behaviour-based. They do not only check whether an endpoint exists or returns a successful status code. A test may create data, perform several API calls, and then check whether the resulting state matches the requirement. This allows the harness to test behavior such as role permissions, validation rules, state changes, calculated results, and error handling.

Each functional requirement is assigned one of four statuses. A requirement is MET when the expected behavior is observed. It is PARTIAL when some relevant behavior works, but the requirement is not fully satisfied. It is BLOCKED when the harness cannot evaluate the requirement because an earlier dependency failed, such as authentication or object creation. It is MISS when the behavior is absent or clearly incorrect.

Two coverage scores are calculated from these statuses. Strict coverage counts only requirements marked as MET:

$$\text{strict coverage} = \frac{\#\text{MET}}{\#\text{functional requirements}}$$

Lenient coverage gives half credit for requirements marked as PARTIAL:

$$\text{lenient coverage} = \frac{\#\text{MET} + 0.5 \cdot \#\text{PARTIAL}}{\#\text{functional requirements}}$$

Requirements marked as BLOCKED or MISS do not count as covered in either score.

The same scenario harness is used for all workflows within the same application. This keeps the evaluation and scoring fixed while the generation workflow changes. The evaluation happens after generation is finished. The agent does not see the scenario harness or test results, and cannot use them to revise the backend during the run.

3.6 Method Overview

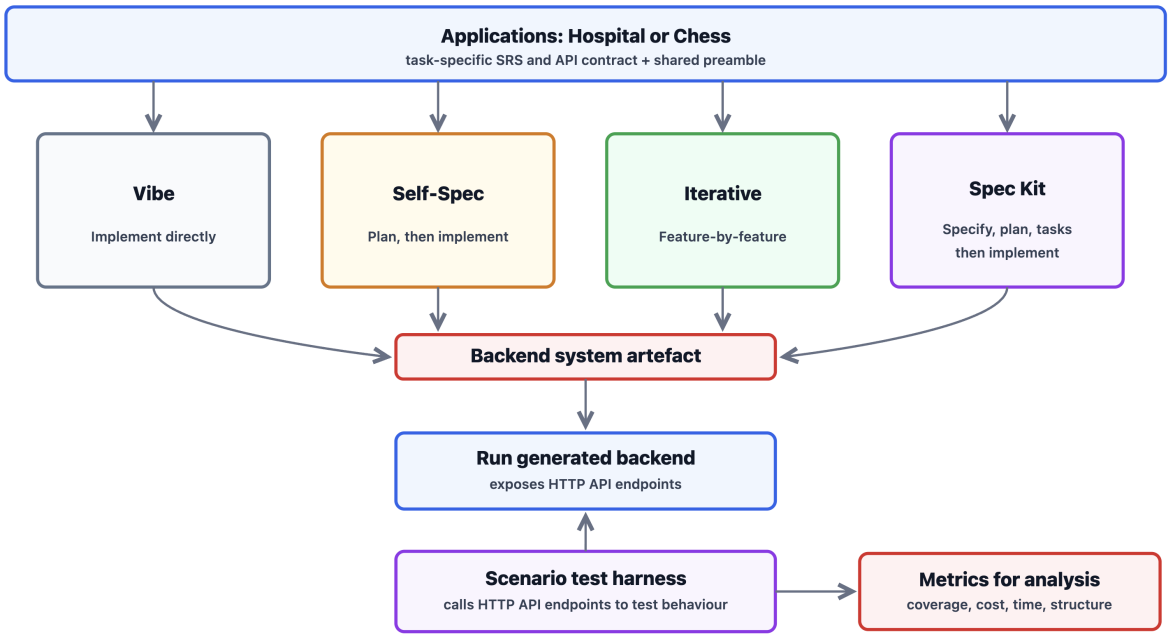

Figure 3.2: Overview of the experimental method from task material, through the four workflows, to generated backend, scenario-harness evaluation, and analysis metrics.

Each run starts from one application task and the same type of task material: an SRS, a fixed API contract, and the shared preamble. A selected workflow then generates one backend implementation. Across repeated runs, this produces multiple implementations for each application and workflow. After generation, the scenario harness starts each backend, calls the fixed API endpoints, and records the metrics used for analysis.

3.7 Analysis Procedure

The analysis is split into two parts. RQ1 uses measurements from all runs, while RQ2 uses a manual review of selected implementations.

3.7.1 Analysis for RQ1

RQ1 focuses on functional coverage, token use, estimated model cost, and run duration. For each run, the analysis uses the strict coverage score, lenient coverage score, token use, estimated model cost, and generation duration.

Estimated model cost is calculated from the recorded token use and OpenAI's public API price table [18]. To keep the calculation consistent across runs, the estimate uses the standard short-context prices for input, cached input, and output tokens. The resulting dollar values are used as comparable estimates rather than exact billing amounts.

Table 3.1: GPT5.4 Token Pricing

Token type	Price per 1M tokens
Input	$2.50
Cached input	$0.25
Output	$15.00

Cost efficiency is reported as lenient FR coverage per estimated dollar. This shows how much functional coverage each condition achieved relative to its estimated model cost.

The results are grouped by application, workflow, and reasoning-effort setting. This makes it possible to compare both average score and variation between runs. Since LLM outputs are non-deterministic, each condition is repeated multiple times. This does not remove randomness, but it reduces the risk that the comparison depends on one unusually good or bad run.

3.7.2 Analysis for RQ2

RQ2 is answered with a manual review of selected implementations. Reviewing all generated application would be too broad for the scope of the thesis, so one implementation is selected for each workflow and application.

The selected implementation comes from the medium reasoning-effort setting. Within that setting, the run with the highest strict coverage is selected. If several runs are tied, lenient coverage is used next, followed by the earliest iteration. This keeps the reasoning-effort setting fixed while selecting functionally strong implementations for review.

The review focuses on code organization, responsibility separation, and overall understandability. It looks at indicators such as the number of source files, the size of the largest class or function, and whether the implementation is structured as a single application file, split service/API files, or a layered layout. The goal is to describe structural differences that the functional test results do not show.

Chapter 4

Implementation

The benchmark was implemented as an automated runner around Codex CLI. The implementation turns the method into a pipeline for preparing application material, executing workflows, testing generated backends, and processing results.

Manual test runs of Codex were used early in development to identify what the runner had to control: the working directory, application material, Codex state, environment variables, generated files, and evaluation steps. These runs showed that manual execution was too slow and error-prone for the full benchmark. The final implementation therefore uses an automated runner and a Docker-based execution environment, where each run receives a fresh work directory mounted as the workspace visible to Codex.

AI assistance was used while implementing the benchmark infrastructure, including application material, API contracts, runner scripts, scenario harnesses, and result-processing utilities. These outputs were not accepted automatically. They were reviewed, edited, tested, and adjusted manually until the benchmark behaved as intended.

4.1 Application Material and Fixed API Contracts

The task material for each application was kept in the runner as one SRS file and one OpenAPI contract. These files were the source files used to prepare each generation run.

Each run also included a fixed preamble and workflow-specific prompts. The preamble defined shared constraints for the backend application, such as the input files, local execution requirements, database configuration, authentication expectations, and assumptions about external services. The workflow-specific prompts then adapted these constraints to each workflow, for example by asking the agent to implement everything directly or to first split the requirements into feature blocks.

The SRS files were written for backend generation. They defined the main roles, resources, assumptions, and functional requirements for each application. They also showed which actions had to happen before others. For example, an appointment needs a patient, a doctor and an available time slot. A chess round needs registered players and after the first round, results from earlier rounds.

Table 4.1: Hospital SRS functional requirements

FR	Requirement
FR1	Patients can register with personal, contact, and insurance details.
FR2	Patients can log in with registered email and password.
FR3	Patients can recover their password.
FR4	Patients can view available doctors, specialties, and availability.
FR5	Patients can schedule appointments based on doctor availability.
FR6	Patients receive appointment confirmations by email and SMS.
FR7	Doctors can access patient medical records, including history, test results, and prescriptions.
FR8	Doctors can update patient medical records after consultations.
FR9	Patients have read-only access to their medical records.
FR10	The system generates bills for consultations, treatments, and services.
FR11	Patients can make online payments through a payment gateway.
FR12	Payment receipts are emailed after successful payment.
FR13	Administrators can manage doctor and staff profiles, contact details, and schedules.
FR14	Administrators can access reports on admissions, consultations, and resources.
FR15	Administrators can view and manage billing and payment information.
FR16	The system sends reminders for upcoming appointments.
FR17	The system alerts administrators about resource shortages or system issues.

The hospital SRS described a management backend for hospital administration, clinical work, and finance. It covered areas such as patients, appointments, medical records, billing, staff, resources, reports, and notifications. Since this SRS was provided by the university, its structure and content were already fixed.

Table 4.2: Chess SRS functional requirements

FR	Requirement
FR1	Arbiters can create a tournament with a name, number of rounds, time control, and start date.
FR2	Arbiters can configure the tiebreak order, with Buchholz, Sonneborn-Berger, and direct encounter as the default order.
FR3	Arbiters can mark a tournament as started, closing further registration.
FR4	Arbiters can mark a tournament as complete, triggering Elo rating updates for all participating players.
FR5	Arbiters can register a player with a name, Elo rating, and optional federation and title.
FR6	Arbiters can withdraw a registered player before the tournament starts.
FR7	Arbiters can withdraw a player mid-tournament, giving the player zero points for remaining rounds and excluding the player from future pairings.
FR8	The system generates Swiss-system pairings for a round only after all previous-round results have been recorded.
FR9	Pairings prioritize players within the same score group, floating one player to an adjacent score group when needed.
FR10	The system prevents the same two players from being paired more than once in the same tournament.
FR11	The system balances colours so that a player is not assigned the same colour three times in a row if an alternative exists.
FR12	The system assigns one bye to the lowest-ranked eligible player when the number of active players is odd.
FR13	Arbiters can record game results as white win, black win, draw, white win by forfeit, or black win by forfeit.
FR14	Arbiters can correct an incorrectly recorded result, causing standings to be recomputed.
FR15	The system blocks later-round pairings until all current-round games have a result or bye.
FR16	The system computes current standings ordered by score and configured tiebreaks.
FR17	The system computes Buchholz as the sum of current scores of all opponents faced.
FR18	The system computes Sonneborn-Berger from defeated opponents' scores and half the scores of drawn opponents.
FR19	The system computes direct encounter between tied players based on their head-to-head result.
FR20	The system computes new Elo ratings after tournament completion using the standard Elo formula.
FR21	The system produces a crosstable showing every tournament pairing and result.
FR22	The system produces a final standings report with final score, rank, and configured tiebreak values.

The chess SRS was created for the benchmark. AI assistance was used to draft an initial version after studying Swiss-system chess tournaments, and the draft was then reviewed and revised manually. The final SRS described a tournament management backend covering setup, players, rounds, pairings, results, standings, reports, and Elo updates. This SRS covers the fundamentals of the tournament rules.

Both API contracts were created after the SRSs were finished. AI assistance was used to draft the OpenAPI files, and the drafts were checked manually against the requirements. The review focused on whether each functional area in the SRS could be reached through the contract, and whether the request bodies, response bodies, identifiers, and status codes gave the scenario harness enough information to test behavior. When an endpoint or response shape made a requirement hard to test, the contract was adjusted.

The two contracts also reflected differences between the applications. The hospital application included login behavior, so the contract defined authentication endpoints, session tokens, and fixed evaluation credentials for doctor and administrator accounts. The chess application did not evaluate authentication, so arbiter access was represented through an `X-Role` header. This kept the chess benchmark focused on tournament logic rather than account management.

During run preparation, the runner copied the selected SRS into the run workspace as `srs.md`. It also copied the matching OpenAPI contract as `api_contract.yaml`. The workflow prompts referred to these fixed file names. This kept prompt construction simple and ensured that all workflows for the same task received the same requirements and the same public API.

4.2 Runner and Benchmark Pipeline

The benchmark pipeline was built around two runner scripts. The first script, `run.py`, handled one workflow run at a time. It took the workflow, application, and iteration number as command-line arguments. It then found the right task files, created a fresh workspace, merged the necessary prompts, ran Codex, extracted metadata, and saved the finished run folder.

```
python3 run.py run vibe hospital --iter 1 --timeout 1800
```

Because `run.py` only executed one run at a time, `run_parallel.py` was added to run the experiment faster. This script expanded a batch request into multiple independent `run.py` jobs and limited how many jobs could run in parallel. It did not define a separate benchmark procedure; it only started several normal runner jobs at the same time. Each individual run followed the same sequence: prepare the workspace, copy in the task material, run the selected workflow, start the generated backend, run the scenario harness, and store the results. The Codex event stream was saved as `run.jsonl`, while run metadata was saved as `metadata.json`.

The runner also handled fragile setup steps. It chose a free local port, checked for stale servers, and waited until the generated backend responded. This reduced the risk that evaluation started against the wrong process or before the backend application was ready. This runner structure made the benchmark less dependent on manual execution. The workflow, application, iteration, model setting, reasoning-effort setting, Codex output, generated files, and evaluation result were all stored as part of the run. Later scripts could then process the preserved run folders without needing to reconstruct what happened during generation.

4.3 Codex Execution Environment

After the runner was working, the main challenge was how Codex should be run. The runner could create a fresh folder for each run, but the surrounding Codex setup still depended on the local machine.

The first version used temporary work directories under `/tmp/bachelor-runs`. For each run, the runner created a new folder, copied in the task material, wrote the prompt files, and pointed Codex at that folder. This worked for early testing and made the run output easier to keep separate from the benchmark code.

The temporary-folder setup improved separation, but it was still only folder isolation. Codex still ran on the host machine, with the host toolchain and local Codex setup around it. To make each run more self-contained, the benchmark was moved to a Docker-based execution environment.

The final setup therefore ran Codex inside a Docker-based runner environment. The runner built a fixed Docker image and mounted the run workspace into the container as `/work`. Codex was then executed with `/work` as its working directory. Before each generation run, the runner performed a preflight [1] check inside the container. This checked that the workspace was readable and writeable, that required tools such as Python, Node, npm, and that Codex were available.

Codex still needed access to authentication, but the full local `.codex` folder was not mounted into the container. Instead, the runner created a temporary Codex folder for the run and copied only the authentication file into it. The runner also set the model and reasoning-effort value, and stored both values in `metadata.json`.

Some workflows needed more than one Codex call. The runner kept these calls connected by saving the Codex thread identifier[2] from the first call and resuming that thread in later calls. The resumed thread kept the conversation context, and the shared workspace kept the files from earlier phases.

Codex was executed in JSON output mode[3], and the event stream was saved as `run.jsonl`. For workflows with several Codex calls, new events were appended to the same file. The runner then read this file to extract token use, command executions, file changes, and agent messages. These values were stored in `metadata.json` together with timing and configuration data.

4.4 Scenario Harness Design

The scenario harnesses were built from the functional requirements in the SRSs. For each application, the first version was created by going through the SRS one FR at a time and drafting a test for that behavior. AI assistance was used for these drafts, but each test was checked manually against both the SRS and the OpenAPI contract before it was kept.

The tests were then organized into scenario flows. Many requirements could not be tested as isolated API calls, because they depended on earlier calls. In the hospital application, for example, a patient had to be registered before login could be tested, and login was needed before appointments, records, bills, and payments could be tested. In the chess application, a tournament had to be created before players could be registered and players had to exist before pairings could be generated.

The harnesses therefore behaved like short user flows. They made one API call, saved useful values from the response, and used those values in later calls. For example, a login response could provide a session token, while a created tournament could provide the tournament ID needed for player registration, pairings, and results. The checks used both status codes and response data to see whether the expected state change had happened.

Development was iterative, and FR coverage was double checked during harness development. Results was checked against SRS, the API contract, and the actual responses. If the test made a wrong assumption or was too narrow, it was changed.

After each backend was generated, the runner started the application and passed its local base

[1] Preflight check verifies workspace could be read and written to and that it had the required tools.

[2] A Codex thread identifier is the ID Codex uses for one conversation session. Resuming the thread lets a later Codex call continue with the earlier conversation context.

[3] JSON output mode means that Codex prints structured event records instead of only human-readable terminal text.

URL to the matching scenario harness. The same harness was used for every iteration of the same application. Each run produced a `test_results.json` file in the run folder, together with backend logs that could be inspected when a result needed explanation.

4.5 Result Storage and Processing

After a run finished, the runner moved the workspace into `runs/results`. The folder name used the timestamp, workflow, application, reasoning-effort setting, and iteration number. This made the folder name unique and easy to recognize.

Each run folder kept the generated backend and the files needed to understand the run. This included the copied application files, prompt files, Codex event stream, metadata, backend logs, and scenario-harness results.

`compare_results.py` read the saved run folders and collected the values needed for analysis. It took requirement results from `test_results.json` and token, timing, and configuration data from `metadata.json`. It then grouped the runs by application, workflow, reasoning-effort setting, and iteration, for showcasing the results in different graphs.

The script also created the tables and figures used in the results. These covered coverage scores, token use, estimated cost, duration, and requirements that were not fully met.

The saved run folders were also used for the manual code review. Each selected implementation could be reviewed together with its prompts, metadata, and test results. This made the results easy to trace back to their run folders.

Chapter 5

Results

The result set contains 240 runs. Each workflow was run 30 times for each backend application: 10 runs with reasoning effort set to `none`, 10 runs with reasoning effort set to `low`, and 10 runs with reasoning effort set to `medium`. This gives 120 runs for the hospital application and 120 runs for the chess application.

FR coverage is reported as both strict and lenient coverage. The strict score counts only requirements marked MET, while the lenient score also gives half credit for requirements marked PARTIAL. Requirements marked MISS or BLOCKED receive no credit. In figures that show which requirements failed most often, NOT FULLY MET means requirements marked PARTIAL, BLOCKED, or MISS. The main comparisons are made between workflows and between the two backend applications. Reasoning effort is reported as a secondary comparison where it helps explain differences between workflows.

Run duration measures the time from workflow start to completed generation. Duration is treated as an approximate measure. It helps show broad workflow patterns, but small timing differences may come from network conditions, provider load, or local machine load.

5.1 Overall Results

Across the result set, the lighter workflows gave the strongest coverage-cost trade-off. Vibe and Self-Spec reached high mean FR coverage on both applications while using less time and fewer tokens. Spec Kit was more expensive and had lower mean coverage. Iterative was the only workflow that produced perfect chess runs, but its mean coverage was not clearly higher than the lighter workflows. Across both applications, the highest mean lenient coverage was Self-Spec with reasoning effort set to `medium`, at 97.35%. Vibe with reasoning effort set to `none` had the highest coverage per dollar and was the cheapest combination on both backend applications.

Figure 5.1: Overall coverage vs cost by backend applications, workflow, and reasoning-effort setting. Higher x-axis values indicate higher lenient FR coverage, while lower y-axis values indicate lower estimated price. Vibe and Self-Spec are closest to the high-coverage, low-cost area in both applications; Spec Kit and Iterative generally move upward in cost without a matching increase in mean coverage.

5.2 Hospital Results

Vibe, Self-Spec, and Iterative each had a median strict score of 17 out of 17 on the hospital application. In other words, at least half of the runs for each of these workflows satisfied every hospital functional requirement. Their mean strict and lenient scores were also close, with less than one requirement separating the three workflows. Self-Spec had the highest mean result, followed closely by Vibe and Iterative.

Figure 5.2: Hospital FR coverage vs cost trade-off by workflow and reasoning-effort setting.

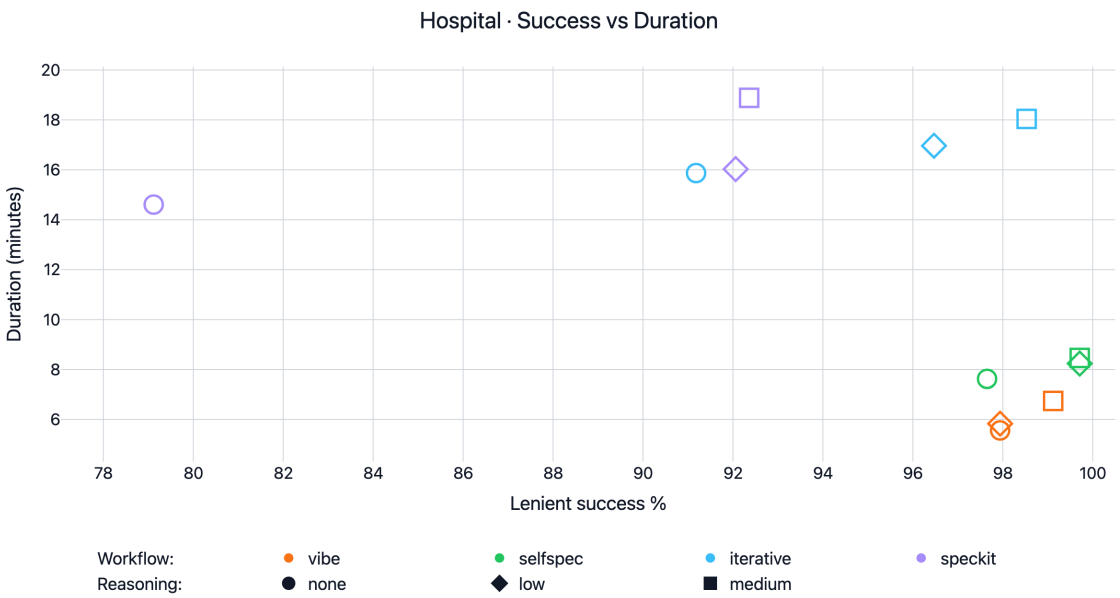

Figure 5.3: Hospital FR coverage vs duration trade-off by workflow and reasoning-effort setting.

Figure 5.2 and Figure 5.3 show the same broad pattern. Self-Spec and Vibe satisfied almost all hospital requirements while using less estimated model cost and shorter execution time than Spec Kit and Iterative. Cost and runtime differed more than requirement coverage. This made Vibe and Self-Spec the strongest hospital trade-off.

Spec Kit had lower aggregate scores, but this result was strongly affected by the no-reasoning setting. Across all hospital runs, Spec Kit reached 13.23 strict points and 14.93 lenient points out of 17. The best Spec Kit hospital run still reached full strict coverage. This indicates that the lower aggregate score was partly driven by the no-reasoning runs.

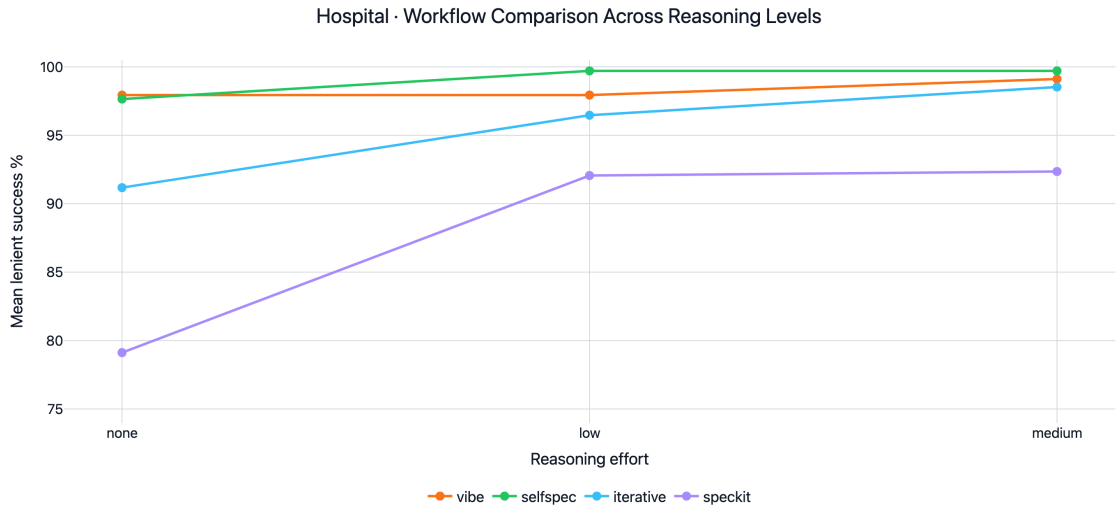

Figure 5.4: Hospital mean lenient FR coverage by workflow and reasoning-effort setting.

Table 5.1: Hospital mean lenient FR coverage by workflow and reasoning-effort setting. The change column shows the percentage difference from `none` to `medium`.

Workflow	None	Low	Medium	Change
Vibe	97.94%	97.94%	99.12%	+1.18 pp
Self-Spec	97.65%	99.71%	99.71%	+2.06 pp
Iterative	91.18%	96.47%	98.53%	+7.35 pp
Spec Kit	79.12%	92.06%	92.35%	+13.23 pp

Figure 5.4 and Table 5.1 show the reasoning-effort difference more directly. Spec Kit changed the most across reasoning settings. Iterative also improved, while Vibe and Self-Spec were already close to full lenient coverage with no reasoning effort and changed less.

The remaining hospital failures were concentrated in a small set of functional areas.

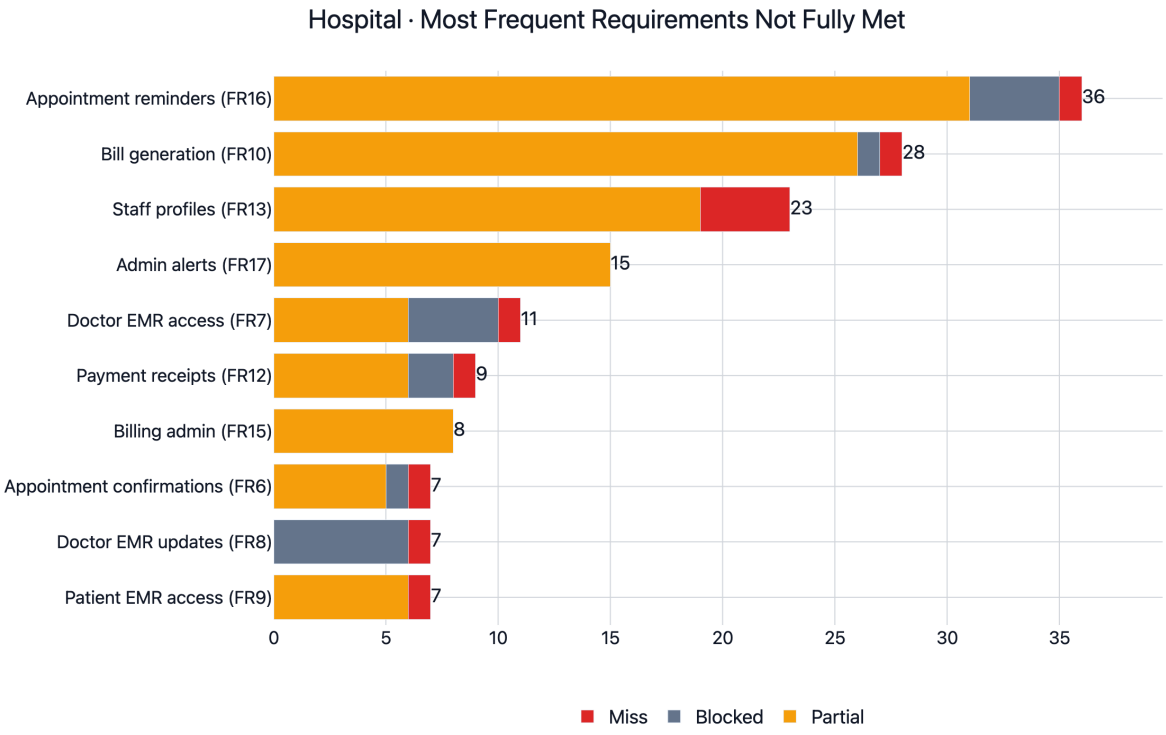

Figure 5.5: Most frequent not fully met hospital requirements across 120 benchmark runs, split by status.

Most of these NOT FULLY MET results were partial rather than missed, especially for appointment reminders and bill generation. This means the generated applications often implemented part of the required behavior, but still failed at least one scenario check for the requirement.

Most hospital runs still implemented most requirements of the API successfully: creating resources, updating state, retrieving records, and enforcing the main role-based behaviors. The NOT FULLY MET results therefore describe the remaining problem areas, not a general failure to build the hospital API.

5.3 Chess Results

The chess application had lower aggregate requirement coverage than the hospital application. The main NOT FULLY MET results were concentrated around tournament completion, Swiss-pairing constraints, bye assignment, and rating updates.

Table 5.2: Chess FR coverage summary by workflow.

Workflow	Mean strict	Mean lenient	Median strict	Best strict
Vibe	19.67/22	20.63/22	20/22	21/22
Self-Spec	19.47/22	20.50/22	20/22	21/22
Iterative	19.70/22	20.33/22	20/22	22/22
Spec Kit	18.33/22	19.87/22	19/22	21/22

Table 5.2 shows that Iterative had the highest mean strict score and was the only workflow to reach 22 out of 22 strict requirements. Vibe had the highest mean lenient score, while Spec Kit had the lowest mean chess coverage across the board.

Figure 5.6 compares chess requirement coverage with estimated model cost. Each point shows one workflow and reasoning-effort setting, averaged across its runs, with coverage compared against estimated model cost.

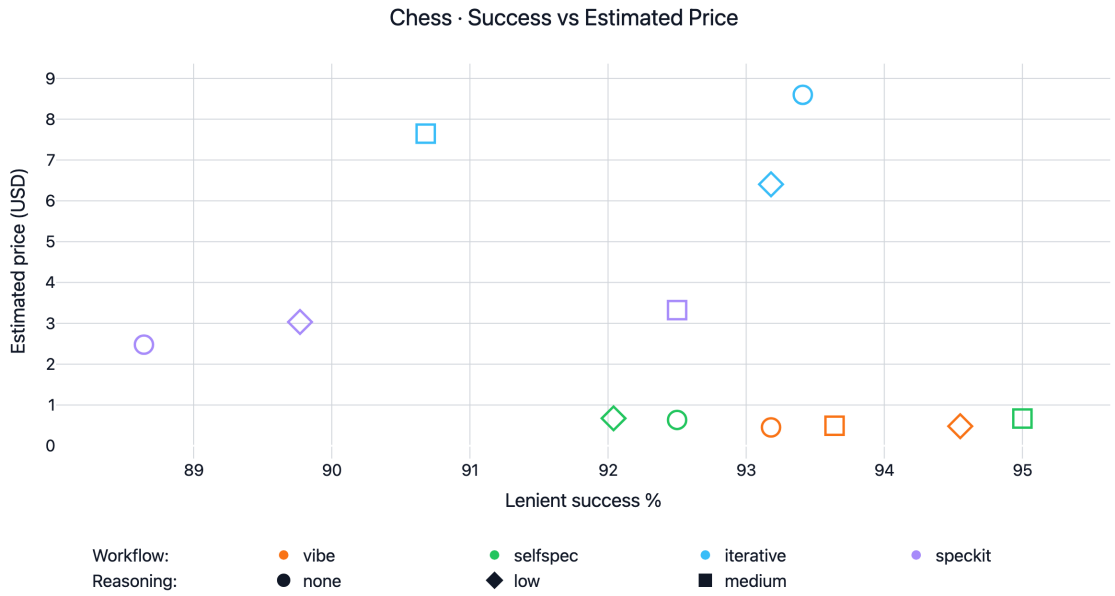

Figure 5.6: Chess mean FR coverage vs cost trade-off by workflow and reasoning-effort setting.

The Vibe and Self-Spec configurations reached the strongest lenient chess coverage while using less estimated model cost than Spec Kit and Iterative. Iterative had the strongest best-run strict score, but its higher model cost meant that it did not have the best observed coverage-cost trade-off on the chess application.

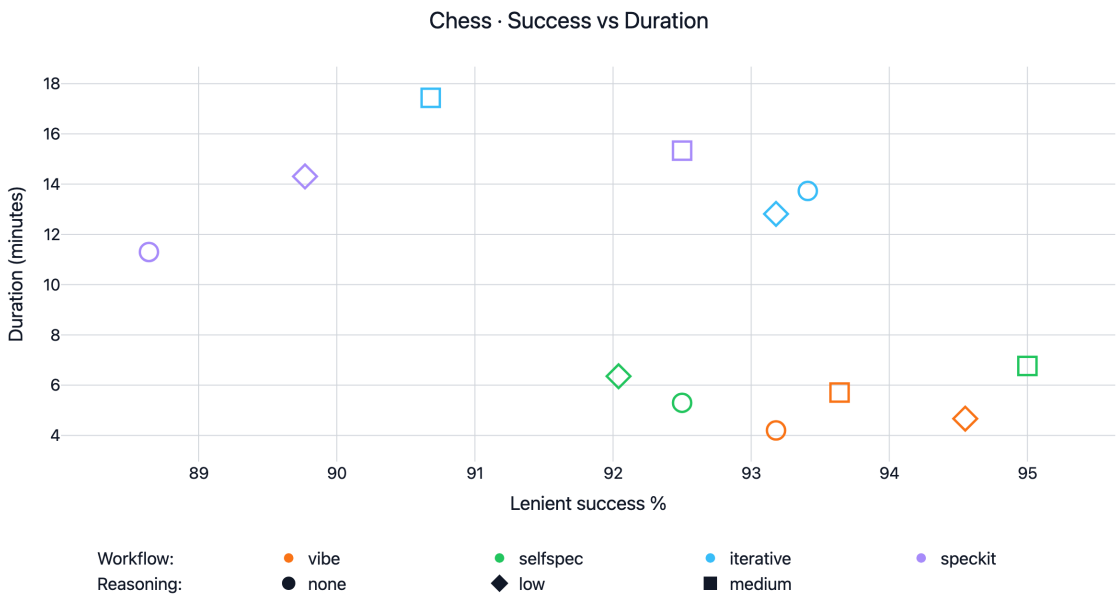

Figure 5.7: Chess mean FR coverage vs duration trade-off by workflow and reasoning-effort setting.

The Vibe and Self-Spec configurations reached similar or higher chess coverage with shorter duration than Spec Kit and Iterative. The same broad grouping is therefore visible in both estimated model cost and duration.

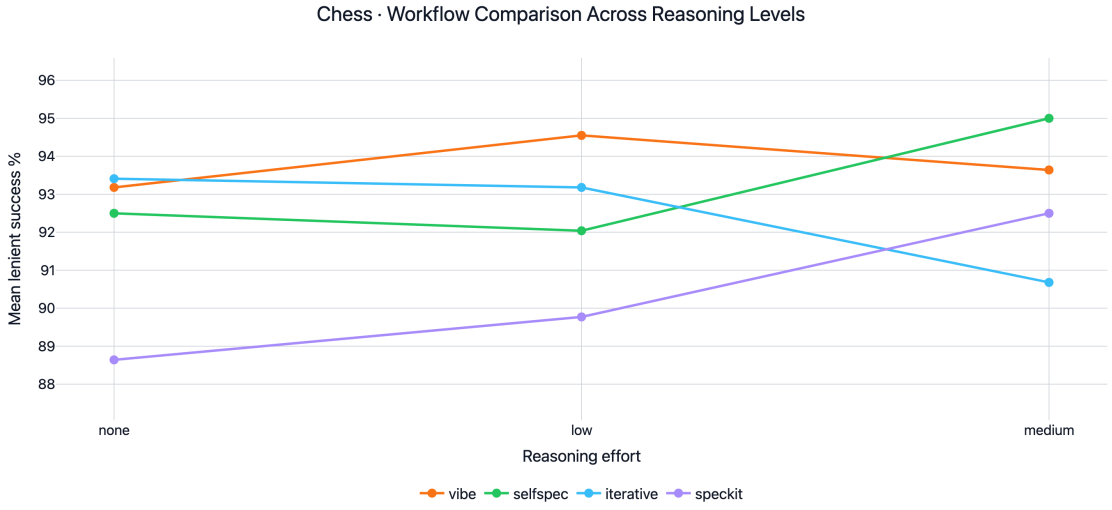

Figure 5.8: Chess mean lenient FR coverage by workflow and reasoning-effort setting.

Table 5.3: Chess mean lenient FR coverage by workflow and reasoning-effort setting. The change column shows difference from `none` to `medium` in percentage.

Workflow	None	Low	Medium	Change
Vibe	93.18%	94.55%	93.64%	+0.46 pp
Self-Spec	92.50%	92.05%	95.00%	+2.50 pp
Iterative	93.41%	93.18%	90.68%	-2.73 pp
Spec Kit	88.64%	89.77%	92.50%	+3.86 pp

Figure 5.8 and Table 5.3 show that the reasoning-effort changes were smaller than the main differences between workflows. The chess results did not show that higher reasoning effort consistently improved coverage across all workflows.

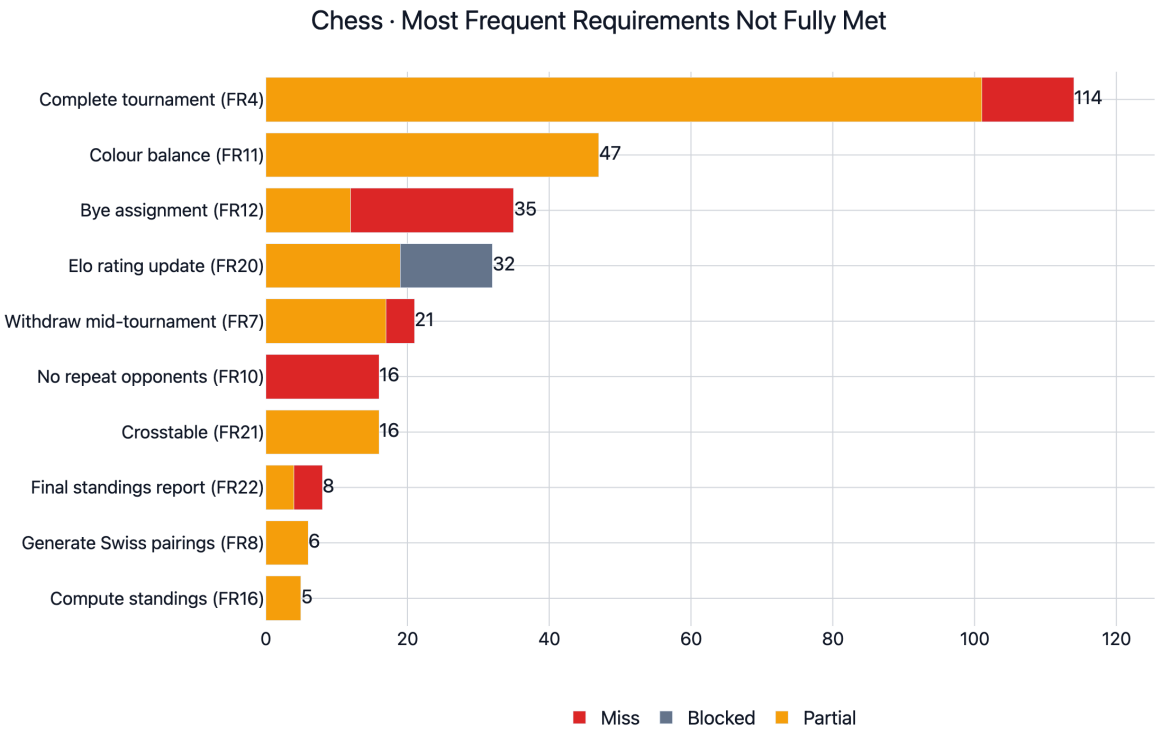

Figure 5.9: Most frequent not fully met chess requirements across 120 benchmark runs, split by status.

The remaining chess failures were more concentrated than the hospital failures. Figure 5.9 shows the ten most frequent not fully met chess requirements across the 120 chess benchmark runs, split into PARTIAL, BLOCKED, and MISSED results.

The lower-frequency chess failures still followed the same broad pattern. They mostly involved chess-application requirements that depend on earlier tournament state, such as withdrawals, repeat-opponent prevention, crosstables, final standings, and score-group pairing.

5.4 Cost Results

Cost was measured in two main ways. Estimated model cost was calculated from token usage. Duration time measured as practical execution time for a run.

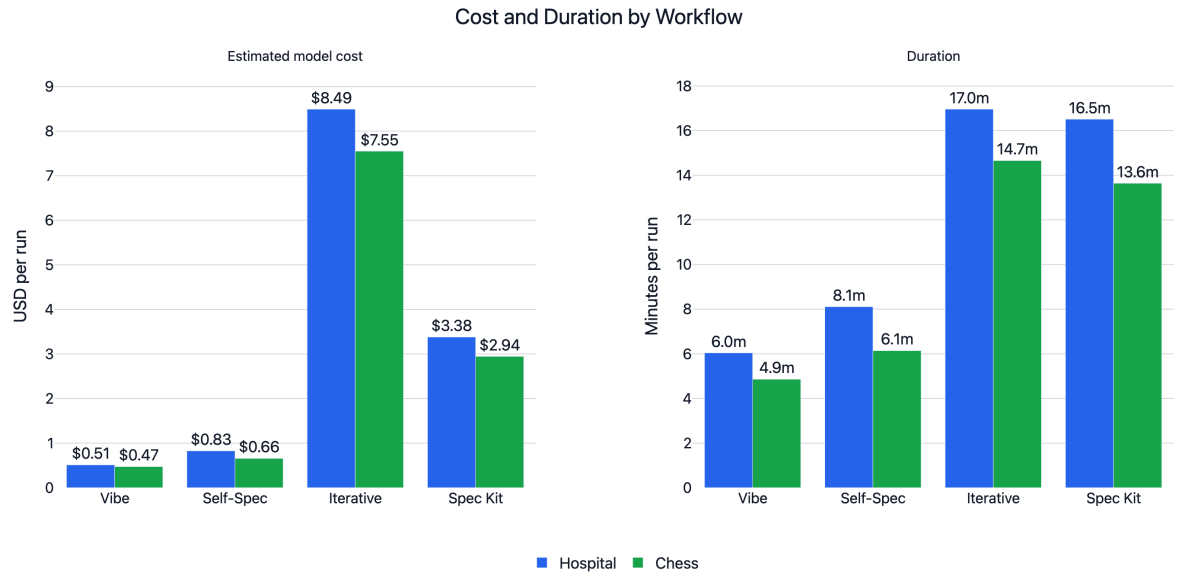

Figure 5.10: Mean estimated model cost and duration time by application and workflow.

Figure 5.10 show that the cost difference between workflows was much larger than the FR coverage difference reported earlier. Vibe was the cheapest workflow on both applications, followed by Self-Spec. Spec Kit and Iterative used substantially more tokens and duration time.

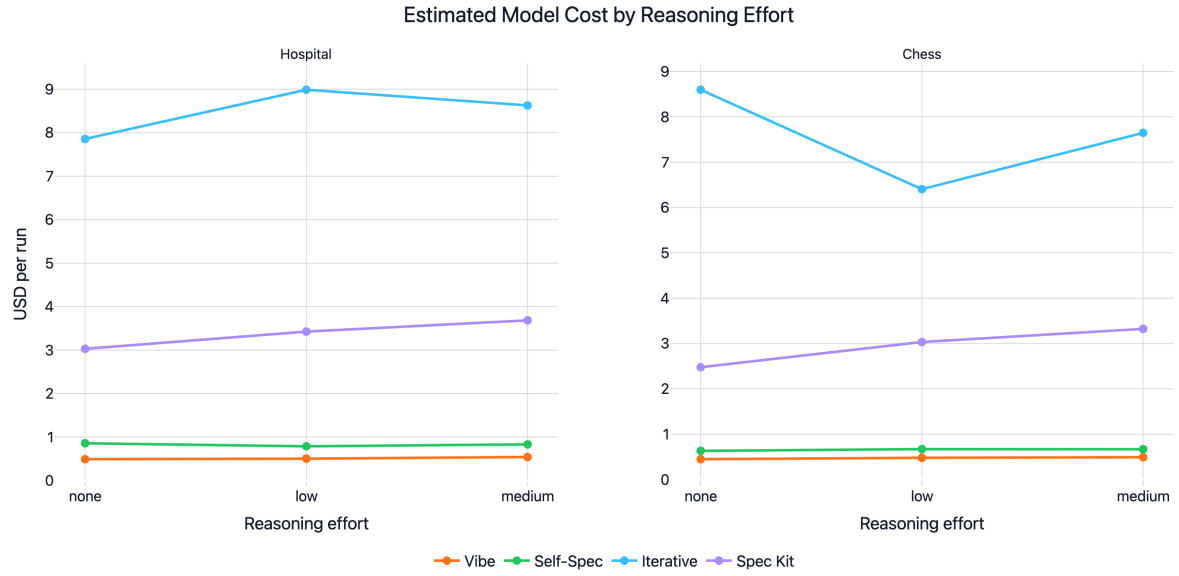

Figure 5.11: Mean estimated model cost by application, workflow, and reasoning-effort setting.

Table 5.4: Estimated model cost range across reasoning-effort settings.

Workflow	Hospital range	Chess range
Vibe	$0.49–$0.54	$0.45–$0.49
Self-Spec	$0.79–$0.86	$0.63–$0.67
Iterative	$7.86–$8.99	$6.40–$8.60
Spec Kit	$3.03–$3.68	$2.48–$3.32

Figure 5.11 and Table 5.4 show that the lighter workflows stayed low across both applications and all three reasoning settings. The heavier workflows were more expensive across the full reasoning range, with Iterative remaining the highest-cost workflow overall.

The cost results therefore follow the same broad pattern as the coverage-cost figures in the hospital and chess sections. The lighter workflows gave the best observed coverage-cost trade-off, while the heavier workflows used more time and model budget without consistently producing higher mean FR coverage.

5.5 Observed Code Structure

This section reports selected code-structure observations for RQ2. For each of the four workflows, one hospital implementation and one chess implementation were reviewed, giving eight implementations in total. The reviewed implementations were chosen by first selecting the run with the highest lenient FR coverage, then the highest strict FR coverage, and finally the earliest iteration if there was still a tie. In these cases, the selected runs were all medium-reasoning runs.

Table 5.5: All selected hospital implementations reached full strict coverage. In chess, Iterative was the only selected implementation with full strict coverage.

Task	Workflow	Selected run	Strict score
Hospital	Vibe	`i1`	17/17
Hospital	Self-Spec	`i1`	17/17
Hospital	Spec Kit	`i4`	17/17
Hospital	Iterative	`i1`	17/17
Chess	Vibe	`i7`	21/22
Chess	Self-Spec	`i1`	21/22
Chess	Spec Kit	`i10`	21/22
Chess	Iterative	`i1`	22/22

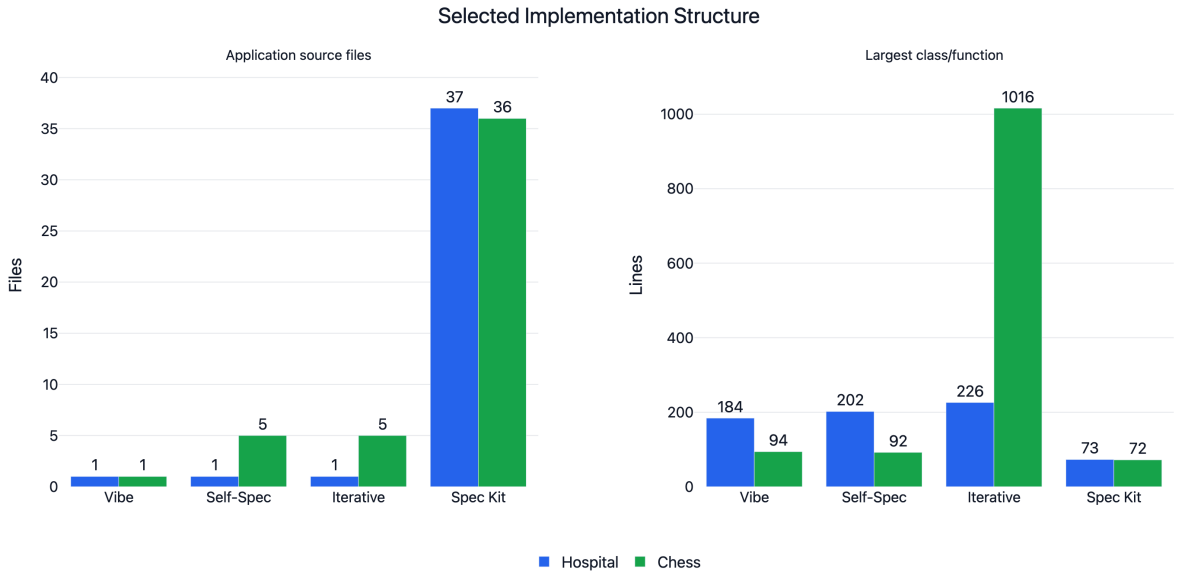

Figure 5.12: Application source-file count and largest class or function in the selected medium-reasoning implementations.

Figure 5.12 shows two simple structure measures: how many generated Python application files each implementation used, and the size of its largest class or function. Tests, dependencies, caches, prompts, run metadata, and workflow support files are not counted.

Table 5.6: Selected code-structure indicators.

Task	Workflow	Source files	Largest unit	Main structural pattern
Hospital	Vibe	1	184 lines	Single application file
Hospital	Self-Spec	1	202 lines	Single application file
Hospital	Spec Kit	37	73 lines	Layered file layout
Hospital	Iterative	1	226 lines	Single application file
Chess	Vibe	1	94 lines	Single application file
Chess	Self-Spec	5	92 lines	Split service/API files
Chess	Spec Kit	36	72 lines	Layered file layout
Chess	Iterative	5	1,016 lines	Large repository class

Spec Kit showed the clearest structural difference. The selected Spec Kit implementations used more source files, but their largest classes and functions were smaller. The code was split into areas such as routes, services, repositories, models, and schemas. Among the inspected runs, this gave Spec Kit the most consistent separation of responsibilities.

Figure 5.13 shows the file structure of the code. It counts source lines and groups them by role, such as routes, services, repositories, models, schemas, database code, and support files. Single-file implementations are categorised as app/other because their code was not separated into architectural layers.

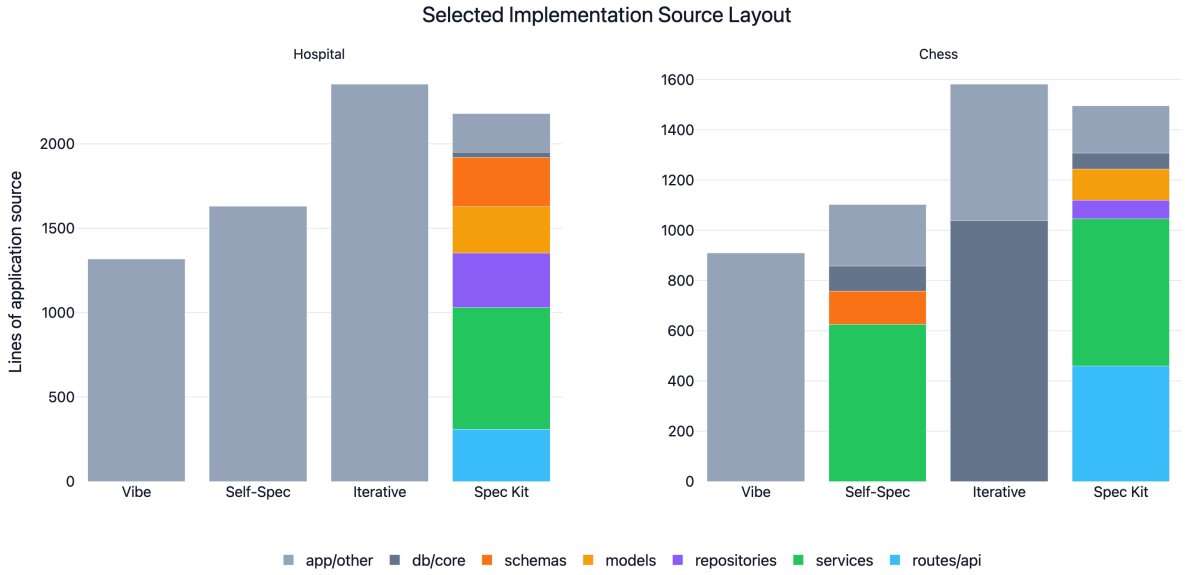

Figure 5.13: Application source layout by task and workflow in the selected medium-reasoning implementations.

The layout view shows the same pattern in a more direct form. Vibe was a single-file implementation on both applications. Spec Kit spread code across route, service, repository, model, schema, and database on both applications. Self-Spec and Iterative changed more between applications.

In hospital, both selected implementations were single-file applications. In chess, both split the code across five application source files.

Vibe remained compact. The selected Vibe implementations were both single-file applications. Both Vibe implementations were monoliths, but the chess version was easier to follow because its single file had clearer routes, request models, and helper functions.

Self-Spec also changed by application. Hospital stayed as one file, with authentication, scheduling, billing, notifications, and reporting together. Chess was split into five files, but one service module still contained most of the domain logic.

Iterative followed mostly the same route. The hospital implementation was one large file that handled API routes, validation, business rules, and database work together. Most of the domain and database logic was concentrated in one large repository class. It did not have a clear internal structure.

Spec Kit had clearer module boundaries, but the parts were still not fully separate. In the hospital run, service functions also handled database access, response data, and notifications. In the chess run, the service layer was cleaner, but some domain logic still depended on database and API details.

Across the inspected runs, functional coverage and internal structure did not move together consistently. Spec Kit produced the clearest and most consistent module layout, but this did not correspond to the highest mean FR coverage in the benchmark. Vibe, Self-Spec, and Iterative could still reach high FR scores even when much of the logic was kept in one file or a small number of files.

Chapter 6

Discussion

Results can make a comparison look more final than it really is. A score shows what happened under the benchmark conditions, but it does not explain why it happened or how far the pattern should be trusted.

The benchmark was designed to make the workflows comparable. Each run used fixed application inputs, a fixed API contract, and the same scenario harness for its application. This control was necessary, but it also shaped what the experiment could show. The results say most about greenfield backend generation from clear requirements and a fixed interface.

The main tension is that FR coverage and code structure did not always point in the same direction. Vibe and Self-Spec often gave strong FR coverage with low cost, while Spec Kit produced clearer structure in the selected implementations. This makes the results less about finding one best workflow and more about understanding which trade-offs each workflow creates.

6.1 Main Findings

The results show that more workflow scaffolding did not automatically lead to better functional coverage. Vibe and Self-Spec gave the strongest overall trade-off between FR coverage, token use, estimated cost, and duration. Spec Kit and Iterative used more phases[1] and more tokens, but did not achieve clearly higher mean FR coverage.

The results also show that functional behavior and code structure should be discussed separately. Spec Kit produced clearer structure in the selected implementations, but this did not translate into the best mean FR coverage. Some compact implementations reached high coverage even though their internal structure was less maintainable.

The applications affected the results in different ways. The hospital application reached high coverage across several workflows, while the chess application exposed more problems in connected domain logic. Reasoning effort also helped unevenly, which suggests that a larger reasoning budget is not a simple quality switch.

The overall conclusion is therefore not that one workflow is always best. The lighter workflows were more efficient for clear greenfield backend applications, while Spec Kit showed more value for code organization. The rest of the discussion explains why those trade-offs appeared and what limits the conclusions.

6.2 Clear Requirements and Fixed Interfaces

The benchmark design gave all workflows a clear starting point. Each run began with a prepared SRS, a fixed API contract, and a shared preamble. This meant that the agent already had the main task structure before implementation began. It did not have to design the API, choose request and response formats, or decide the main parts of the application.

[1] Phases in this context means a message to the LLM, in the same session.

This made the runs easier to compare. Since all implementations had to expose the same endpoints and response formats, the scenario harness could evaluate them in a consistent way. Without this constraint, a failed test could reflect a different design choice rather than an incorrect implementation. The fixed interface therefore reduced noise and kept the benchmark focused on functional behavior.

At the same time, this design limited what the SDD-style workflows could contribute. SDD is often useful before implementation, where it can help reduce ambiguity, identify missing requirements, and make unclear behavior more explicit [3]. In this benchmark, much of that work had already been done before the workflow started. Spec Kit therefore had little room to improve the API, reshape the requirements, or explore alternative interpretations of the task.

This means the results mainly show how SDD-style workflows performs when the task and interface are already clear. They are less suited to showing the value of SDD in more open-ended situations, where the problem still needs to be clarified before implementation can begin.

6.3 Code Structure and Functional Behavior

The difference between FR coverage and code structure is important for how the benchmark should be read. The scenario harness tested the backend from the outside. It showed whether the generated application responded correctly through the API, but it did not show how easy the code would be to change later.

This matters because generated code is often judged by whether it works first. For a small greenfield backend, that may be enough in the short term. However, if the application is extended, debugged, or handed to another developer, internal structure becomes more important. A compact implementation can pass the tests and still be difficult to maintain.

Spec Kit's structure advantage is therefore still meaningful, even though it did not lead to the best mean FR coverage. Clearer separation between routes, services, data access, and models can make an application easier to inspect and modify. That kind of value is not fully captured by the scenario harness.

High FR coverage should not be read as proof of good design. The benchmark only shows that the tested behavior worked. It does not show that responsibilities were well separated, that domain logic was easy to reuse, or that future changes would be simple.

This also raises a design question for the experiment. The lighter workflows were not asked to produce a specific architecture. If Vibe had been told to use a layered file structure, its code structure might have improved.

A later benchmark could make structure a more explicit target. For example, it could compare Vibe with and without architecture instructions, or review code structure across all runs instead of only selected implementations.

6.4 Task Complexity and Reasoning Effort

The two applications created different kinds of difficulty. The hospital application was broad, with several roles and resources, but much of the behavior followed familiar administrative patterns. The chess application had fewer user roles, but more connected domain logic. Later actions depended on earlier tournament state.

This helps explain why the hospital results were higher. Many hospital runs reached full or near-full FR coverage. When several workflows already satisfy almost all requirements, there is

little room for extra scaffolding or reasoning effort to show a large improvement. This makes the hospital results useful, but it also means that workflow differences can be harder to see.

The chess application added more depth to the benchmark. Requirements such as pairings, byes, color balance, standings, tiebreaks, and Elo updates depend on the tournament history. A mistake early in the tournament flow can affect later requirements. This made the chess application better at exposing weaknesses in connected domain logic.

Reasoning effort helped unevenly. On the hospital application, the effect was more visible for workflows that had room to improve. Spec Kit and Iterative improved more across reasoning settings, while Vibe and Self-Spec were already close to full coverage. In that case, higher reasoning effort could help some runs, but it could not improve much on results that were already near the top.

The chess results were less consistent. More reasoning effort gave the model more room to work, but it did not guarantee that it would choose the right tournament rule. If the model misunderstood byes, color balance, or when a tournament should finish, extra reasoning did not always fix the mistake. In some workflows, an early wrong assumption could also be carried into later phases.

The benchmark used two applications to cover different types of backend work. The hospital application was a broad resource-management API, while the chess application was more rule-driven. This gave the experiment some variation while keeping the projects at a similar scale. Larger systems or very different domains may behave differently, but similar backend tasks are expected to show similar patterns.

6.5 Cost, Duration, and Practical Trade-offs

Cost and duration matter because the functional gains between workflows were often small. A more expensive workflow is easier to justify when it clearly improves coverage, structure, or reliability. When coverage is similar, the extra cost needs to provide some other value.

This makes the results different from a simple ranking by FR coverage. A workflow that scores slightly higher is not always the better practical choice if it requires more time and tokens. In this benchmark, Vibe and Self-Spec were often stronger from a practical perspective because they reached high coverage with lower cost and shorter duration.

The results also show that more tokens and more phases did not automatically produce better behavior. This was especially visible in the chess application, where longer workflows still struggled with some domain rules. Extra reasoning time can help if the model uses it to correct or refine its understanding, but it can also preserve an early wrong assumption through more steps.

6.6 Automated SDD and Intermediate Artifacts

The Spec Kit condition should be read as automated SDD-style scaffolding, not as a full human-guided SDD process. In the benchmark, the agent generated the specification, plan, and task list. Later phases then used those documents without human correction. A human-guided process would not have to work the same way.

Human review can change the role of artifacts. A developer can stop after the specification phase and check whether the generated specification describes the right application. The same can happen after planning and task generation. Missing edge cases, wrong assumptions, or unclear priorities can be corrected before implementation starts.

This connects to Piskala's argument that AI-assisted development shifts more work toward describing intent clearly [3]. The benchmark supports this idea in one sense: clear application input mattered. However, it also shows that producing more documents is not the same as improving intent. The documents need to describe the right behavior.

Artifacts are only useful if they are good enough to guide the next step. They can make assumptions clearer, but they can also carry wrong assumptions forward. If a plan misunderstands a requirement, the implementation may follow that mistake. More phases can then repeat the problem instead of fixing it.

Hill gives a similar warning from a different setting. A specification artifact does not reduce defects or rework just by existing [29]. The same applies here: an artifact helps only if it improves the implementation.

This helps explain why Spec Kit produced clearer organization but not the best mean FR coverage. The artifacts may have shaped the codebase and made the implementation more organized. They did not guarantee that every domain rule was implemented correctly. Human intervention may have helped with this.

The practical point is that generated specifications, plans, and task lists should be checked before they are trusted. If SDD-style tools are used, the artifacts should be reviewed and corrected before they guide implementation. For clear greenfield API backend applications, adding more phases alone may not improve functional behavior.

6.7 Methodological Choices and Limitations

Several choices shaped what the benchmark could show. These choices were not only practical implementation details. They affected which results could be measured, how repeatable the experiment became, and what kinds of conclusions could be drawn. This section discusses the most important choices and the trade-offs they introduced.

6.7.1 Backend APIs Instead of Frontend Applications

The benchmark was limited to backend API applications. This made the applications easier to evaluate in a repeatable way. A backend can be started locally, called through HTTP, and checked against expected responses and state changes. This fit the scenario-harness approach used in the evaluation.

Frontend applications would have made the benchmark harder to control. A frontend can be evaluated by checking whether buttons, forms, and pages work, but the quality of a user interface also depends on layout, visual design, responsiveness, accessibility, and how the application feels to use. These parts are harder to score automatically and would have added more subjective judgment to the evaluation.

Choosing backend APIs therefore made the experiment more measurable. It also made it possible to automate many runs without manually inspecting every generated application. The cost of this choice is that the results do not say much about AI-assisted frontend development or full-stack product work. They mainly apply to backend generation where behavior can be tested through a fixed API.

6.7.2 Automation Instead of Manual Runs

Manual Codex runs were useful early in the project. They showed which parts of the process had to be controlled, such as folders, prompts, environment variables, Codex state, generated

files, and evaluation steps. For the full benchmark, manual execution would have introduced too much variation.

Automation made the comparison cleaner. The runner prepared each workspace, started the selected workflow, ran the generated backend, executed the scenario harness, and stored the results in the same way each time. This made it possible to run many iterations without relying on manual setup for every run.

The main advantage was repeatability. If a workflow performed better or worse, the result was less likely to come from a small manual difference in how the run was started or evaluated. Automation also made the result folders easier to trace, because the prompts, metadata, logs, and test results were stored together.

The trade-off is that this is less like normal AI-assisted development. In real use, a developer often reads the output, gives feedback, runs tests, and asks the agent to fix problems. The benchmark did not include that kind of ongoing human interaction. It measured first-pass workflow behavior under controlled conditions.

This was a good fit for comparing workflows, but it narrows the conclusion. The results show how the workflows behaved when the process was automated from generation to evaluation. They do not show how the same workflows would perform with active human steering during each run.

6.7.3 Codex and the Execution Environment

Codex was chosen primarily because Codex CLI is an open-source coding harness from OpenAI, the same provider as the GPT-5.4 model used in the experiment. This made the harness inspectable while still using one of the frontier models selected for the benchmark. Codex could also be run from the command line, started by the runner, and resumed across workflow phases. This mattered for workflows such as Self-Spec, Iterative, and Spec Kit, where later phases had to continue from earlier Codex calls.

Codex also made the runs easier to measure. Its structured event output made it possible to collect token use, command executions, file changes, and timing data in the same way for every run. This was important because token use and duration were part of the comparison, not only functional coverage.

The execution environment was controlled for the same reason. The model output was still non-deterministic, but the goal was to make the setup around the model as stable as possible. Running Codex in a Docker-based environment helped keep the visible workspace, tools, and configuration more consistent across runs.

This choice made the benchmark cleaner, but also narrower. The results describe how these workflows behaved with Codex in this controlled environment. Other agent tools, models, or more interactive development environments could behave differently. For this study, the controlled Codex setup was a reasonable trade-off because the goal was to compare workflows, not agent platforms.

6.7.4 Workflow and Reasoning-Effort Selection

The four workflows were chosen to cover a spectrum of scaffolding rather than every possible way to use an AI coding agent. Vibe gave a direct baseline, Self-Spec added one planning step, Iterative added feature-by-feature decomposition, and Spec Kit represented the most specification-heavy workflow. Together, they made it possible to compare direct implementation with gradually more structured approaches.

This selection kept the benchmark manageable. Adding more workflows would have made the run count much larger, especially because each condition was repeated across two applications and three reasoning-effort settings. The four selected workflows gave enough variation to compare different styles without making the experiment too large to execute and analyze.

Reasoning effort was included because it changes how much internal reasoning the model can use. It is not the same as SDD, because it does not create visible specifications, plans, or task lists. However, it can still affect how carefully the model works through the task before producing code. For that reason, it was useful to test alongside the workflow conditions.

6.7.5 Scenario Harnesses Instead of Gherkin

Gherkin[2] was considered because its Given-When-Then format[3] fits behavior-based testing. It could have made the scenario tests easier to read and closer to the language of the functional requirements.

The final scenario harnesses needed more control than readable scenario text alone could provide. They had to send HTTP requests, save identifiers and tokens from responses, reuse state across later steps, and record evidence for each FR result. Direct scenario harness code made this easier to implement.

The main advantage of direct harnesses was precision. They could handle setup steps, dependencies between requirements, PARTIAL results, and BLOCKED results in a consistent way. This made them easier to connect to the runner and the result files.

The trade-off was readability. Direct test code is more precise, but less accessible than Gherkin scenarios. For this benchmark, repeatable execution was more important than having the tests written in a more readable scenario format. Future work could still use Gherkin as a readable layer above executable tests.

6.7.6 Expectations and Researcher Bias

The study did not start from the assumption that all workflows would perform the same. SDD-style workflows seemed like strong candidates because they promise more control over AI-generated code. The original expectation was that Spec Kit would help solve some of the problems that appear when agents generate a lot of code quickly, but the quality and structure of that code are uncertain.

This expectation was reasonable. If unclear intent is a major problem in AI-assisted development, then a workflow that makes intent clear before implementation should help. This connects to the theory chapter, where SDD was presented as a way to move more work into requirements, plans, and tasks before code is written.

Because of this expectation, it was important that the scoring did not depend on manual preference after the runs were generated. The benchmark used fixed application input, fixed API contracts, automated scenario harnesses, repeated runs, and stored result files. These choices helped reduce the risk that one workflow was favored during evaluation.

6.8 Future Work

The controlled setup was necessary for comparing the workflows, but it also narrowed what the benchmark could show. A future study could test workflows with less fixed input, where the

[2]Gherkin is a simple text format for writing test scenarios.

[3]Given-When-Then means: given some starting situation, when something happens, then this result should follow.

agent has to help shape the API or clarify the requirements before implementation. This would test the part of SDD where it may have more value.

A useful next step would be human-reviewed Spec Kit. The agent could generate the specification, plan, and task list, but a developer would review and correct each document before implementation. This would separate the value of the generated artifacts from the value of human review.

Another useful condition would be Vibe with clearer guidance about code organization. The current Vibe workflow did not ask for a layered file structure. Adding that guidance would show whether Vibe can produce clearer structure without using the full Spec Kit workflow.

Future work should also include brownfield applications. The benchmark used fresh backend applications, but real software work often happens inside existing codebases. Hill's study is closer to this setting because it examines pull requests in existing open-source repositories [29]. However, it does not compare the same agent workflows or controlled task inputs. A useful next step would therefore be a controlled workflow benchmark inside existing codebases, where the agent has to fit into code that already exists.

The benchmark also used a one-shot generation setup. In practice, developers often run tests, inspect failures, and ask the agent to repair the code. A future benchmark could give every workflow the same repair budget, such as one or two feedback rounds after evaluation. This would test whether heavier workflows help more when the agent can revise its first implementation.

The task set could also be expanded. More applications, more backend domains, more models, and more agent harnesses would show whether the observed pattern is specific to this setup. These changes would not replace the current benchmark, they would instead test the parts that this benchmark intentionally held fixed.

6.9 Answering the Research Questions

RQ1 asked which workflow produced the highest FR coverage, and how much token use this required. The results indicate that Vibe and Self-Spec provided the strongest overall trade-off between FR coverage and model usage. Both workflows achieved high coverage with fewer tokens, lower estimated cost, and shorter duration than the more scaffolded workflows. Spec Kit and Iterative used more phases and more tokens, but this did not result in clearly higher mean FR coverage. In this controlled greenfield setting, Vibe and Self-Spec therefore appeared to be the most practical workflows for functional backend generation, as they produced comparable or better functional outcomes with less model usage.

RQ2 asked how the generated implementations differed in code structure. The selected implementations showed that Spec Kit produced clearer organization than the lighter workflows. Its use of artifacts appeared to encourage a clearer division of responsibilities, even though this did not translate into the best mean FR coverage. This suggests that SDD-style scaffolding may provide value for maintainability, reviewability, and code organization, rather than only for functional behavior.

This result should also be read in relation to Hill's critisicm for Spec Kit [29]. The benchmark does not fully confirm that criticism. Rather, it shows that the value of Spec Kit depends on what is being measured and in which development setting it is used. Hill's argument is more aligned with settings where structure, review, and work inside existing codebases are important. This benchmark tested a different setting: greenfield backend generation from clear requirements and fixed API contracts. In that setting, Spec Kit's structural advantages were visible, but they did not produce the strongest functional coverage and token-use trade-off.

The combined answer to the research questions is therefore that the study did produce meaningful results, but not as a universal ranking of workflows. The results show that workflow choice affects both what the generated backend can do and how the backend is organized. However, these effects do not always point in the same direction. A workflow can be efficient and functionally strong without producing the clearest structure, while a more structured workflow can produce more maintainable and reviewable code without achieving the highest coverage.

Chapter 7

Societal Perspective

LLM-based coding agents change who can produce software. A person no longer needs to understand every framework, database pattern, or HTTP detail before creating a small backend, internal tool, or prototype. They can describe what they want, ask the model to implement it, run the result, and keep asking for fixes. This makes software creation more accessible for people who previously depended on professional developers for even simple applications.

The societal impact is not that everyone automatically becomes a software engineer. A simpler way to put it is that more people can now make software, but the tool does not guarantee that they understand what they build. LLMs can explain code, suggest tests, describe security risks, and act as patient tutors. However, they usually do this only when the user asks for it. A user can stay in output mode, repeatedly asking the model to add features and fix errors, without asking why the solution works, what assumptions it makes, or where it may fail.

This chapter discusses that wider shift through the idea of software abundance. The hospital and chess applications in this thesis are small examples of a larger pattern. A hospital-like CRUD application[1] points toward hidden responsibility around access, privacy, and maintenance. A chess tournament backend points toward hidden domain logic behind plausible API responses. The broader issue is that systems can now look finished before people have the judgment needed to trust them.

7.1 From Software Scarcity to Software Abundance

Software has traditionally been scarce because it required specialized labor. Even a small internal system often needed someone who could translate a problem into requirements, choose a technical design, write the implementation, test the result, and maintain it over time. Software engineering treats these activities as part of a wider engineering discipline, not only as programming [35].

LLM coding agents reduce part of that scarcity. Empirical work on GitHub Copilot shows that AI assistance can increase implementation speed in controlled tasks [36]. In everyday practice, the effect can be broader than speed alone. A non-expert can ask for a database schema, an API endpoint, an authentication flow, or a test file and receive something runnable. For small organizations, students, hobbyists, and domain experts, this lowers the cost of experimentation and makes it easier to turn local needs into software.

The consequence is more software produced outside traditional engineering processes. Some of it will be useful and low risk. Some of it may become important before anyone has checked who can access it, how it handles data, how it fails, or who must maintain it.

This is where specification-driven development becomes socially relevant, not only technically relevant. SDD shifts attention from code alone toward explicit intent, constraints, and acceptable behavior [3]. In a world with more AI-generated software, that structure helps make generated systems inspectable by people other than the person who prompted them.

[1] CRUD means create, read, update, and delete. It describes applications where much of the work is storing, retrieving, changing, and removing records.

7.2 The Tutor That Only Teaches When Asked

The same tools that make software easier to produce can also make it easier to understand. A model can explain unfamiliar code, summarize architecture, describe why a test fails, compare implementation alternatives, or point out likely security concerns. For a motivated learner, this is a powerful form of support.

The limitation is that this learning is optional. A model can generate a working endpoint without explaining the access-control assumptions behind it. It can fix a failing test without explaining why the original design was weak. It can produce a Swiss pairing function without proving that the algorithm handles byes, rematches, score groups, and color balance correctly. The user must ask the right questions, and must have enough judgment to notice when the answer is incomplete.

This creates a new competence divide. In the past, the main divide was between people who could write software and people who could not. With LLMs, more people can cross the production barrier. The divide then moves toward how the tool is used: as a code generator, or as a tutor, review partner, and source of explanations that still need verification.

A "vibe coding" workflow makes this tension visible. The user can keep asking for the next visible problem to be fixed: add login, make the endpoint work, make the test pass, fix the error. That can be productive, but it does not necessarily build understanding. The slower questions are often the more important ones: What assumptions did you make? Which users can access this data? What edge cases are not tested? Why is this algorithm correct? What happens if this dependency or API changes later?

7.3 AI-Assisted Drift, Context Drift, and Validation

AI-assisted drift describes how a generated system can move away from its original intent through a series of small plausible changes. Each change may solve the latest visible problem, but the builder may gradually lose track of what has changed, which assumptions are now integrated in the codebase, and which behaviors only work for the narrow case just tested.

Context drift is the model-side version of the same problem. As a session[2] grows, the model may lose grip on earlier requirements, constraints, or design decisions. The information may still exist in the SRS, the plan, a task list, or an earlier file, but the model may not use it consistently. This is especially relevant in multi-phase workflows where several artifacts must remain aligned with the SRS, the fixed API contract, and the generated implementation. Long-context research shows that models do not always use information equally well across a long context [37].

The societal point is that drift can happen while the system still appears to be progressing. The user sees more files, more endpoints, fewer errors, and a model that explains its work confidently. That can create trust before there is strong reason for trust. Work on AI code assistants warns that programmers can become over-reliant on generated code and accept undetected errors when validation practices are weak [38].

Validation is the counterweight to drift. Generated systems need checks outside the conversation that created them. This means real scenario tests, security and maintainability review, and requirements that state important assumptions clearly. The fixed API contract and scenario harnesses in this thesis are examples of such mechanisms. They are not important because every organization will use the same harness, but because they show the kind of structure needed when plausible output is no longer difficult to obtain.

[2]Here, session means one ongoing conversation with the LLM.

7.4 Hidden Complexity Behind Working Software

The hospital and chess applications illustrate why visible completeness can be misleading. The hospital application resembles broad administrative software: users, authentication, appointments, records, billing, reports, and notifications. Many real organizations depend on systems with this kind of CRUD-oriented structure. An LLM can help produce such a system quickly, but a real hospital-like system would also need careful access control, secure handling of patient-related information, auditability, error handling, and long-term maintenance routines.[3]

A non-expert builder may not know which of those properties need to be specified or tested. They may ask the model to "add patient records" without asking who can read those records, how access is logged, what happens when roles conflict, or how sensitive data should be protected. The software can therefore look ordinary while carrying responsibilities that are not ordinary.

The chess task shows a different form of hidden complexity. A tournament API can look complete because it exposes endpoints for tournaments, players, pairings, results, standings, and reports. Yet its value depends on whether the generated logic handles state over several rounds. A weak pairing algorithm may work for the first simple case and fail later when score groups, byes, rematches, withdrawals, color balance, and tiebreaks interact.

This matters beyond the two applications. Many systems produce outputs that users tend to trust because they come from software, such as schedules, rankings, recommendations, decisions, reports, and summaries. The danger is not only that a generated system may contain bugs. It is that it can look complete before its access rules or domain logic are clear enough to question.

The same shift also affects professional software work, because the tasks most easily generated are often the tasks that once helped new developers learn.

7.5 Junior Developers and the Labor Shift

LLMs are especially strong at tasks that have traditionally been common entry points for junior developers: boilerplate, simple endpoints, CRUD screens, standard data models, repetitive tests, and small bug fixes. Studies of generative AI and work suggest that many tasks may be affected, but not always through full job replacement. In many cases, AI is expected to change or support parts of the work instead. [39].

The risk is not simply that junior developers disappear. The more realistic risk is that the learning ladder changes. If mechanical tasks are automated, the remaining work may move toward specification, review, domain understanding, security judgment, and validation. Those are often senior-like skills. A junior developer may be expected to assess generated code before they have built the experience that normally makes such assessment reliable.

LLMs can also make junior developers better. Used actively, they can explain unfamiliar code, suggest tests, compare designs, and shorten feedback loops. The problem is whether organizations design work so that juniors learn from the tool rather than only consume its output. A useful junior task in this setting may be less about writing all code by hand and more about explaining generated code, tracing requirements to tests, identifying missing edge cases, and documenting why a generated solution is acceptable.

This connects to wider expectations that technological change will transform jobs and skills by 2030 [40]. The important societal issue is not only how many developers are employed, but how people enter the field, how competence is built, and who is trusted to approve AI-generated software.

[3] Real hospital systems also need legal, security, and operational controls, such as GDPR compliance, audit logging, backups, and access control.

Security makes the same competence shift more visible, because lack of awareness can turn a working implementation into a real liability.

7.6 Security Becomes a Race

AI does not automatically create vulnerable software. The more precise risk is that it can make software production faster than security awareness. A careful user can ask for threat modeling, access-control checks, tests, and safer alternatives. An unaware user may instead accept generated code because it runs, without noticing weak authentication, missing authorization checks, unsafe input handling, or insecure defaults. Earlier work on Copilot shows that assistants can produce insecure code in security-sensitive tasks. This is best read as a warning about unreviewed generation, not as proof that all AI-generated software is insecure [41].

This matters because LLMs change both sides of security work. They can increase the amount of code that needs review, but they can also help reviewers inspect that code. A model with broad knowledge of programming languages, frameworks, APIs, vulnerability classes, exploit patterns, and patching techniques can act as a force multiplier for a skilled security researcher. The researcher contributes context, judgment, priorities, and an understanding of which findings matter in a real system. The model contributes breadth, patience, and the ability to scan many technical possibilities quickly. In combination, those strengths can make AI-assisted security work more effective than either the model or the researcher working alone.

Project Glasswing and Claude Mythos are useful examples of this shift[4]. Anthropic presents Glasswing as a controlled defensive program where selected organizations use Mythos Preview to analyze critical software infrastructure [42]. Public reporting describes the same direction: major vendors and researchers are experimenting with AI systems that can help find vulnerabilities across large and complex codebases [43], [44]. The exact public numbers should be treated cautiously; for example, reporting around Firefox findings and Project Glasswing-linked CVEs shows that headline counts do not always map cleanly to confirmed high-severity vulnerabilities [44], [45]. The important point is not the precise count. The important point is that AI-assisted vulnerability discovery is becoming practical enough to affect how security researchers and software vendors organize their work.

The societal issue is therefore a race between generation and validation. If AI helps more people create software while review practices remain weak, society gets more attack surface. If the same capabilities are paired with skilled review, responsible disclosure, patching capacity, and stronger validation habits, they can also improve defense. The risk is not that AI necessarily creates vulnerable software. The risk is that unaware users may create and deploy software without recognizing where the security questions are, while more capable users and researchers can use the same technology to find both mistakes and protections faster.

7.7 Trust in an AI-Generated Software Society

LLM-based coding is not a future scenario waiting to be accepted or rejected. It is already becoming part of how software is produced. The societal question is therefore how trust moves when code can be generated by people who may not fully understand the systems they are creating.

The scarce skill shifts from typing code toward asking the right questions and demanding the right evidence. A runnable system is not enough if no one has checked the assumptions behind

[4]Project Glasswing and Claude Mythos are examples of AI agents being used for larger software-related tasks, not only small code snippets.

it. Access rules, edge cases, misuse cases, maintenance responsibility, and reviewability become part of whether generated software can be relied on. The examples in this thesis are small, but they keep the same red line visible: generated systems can appear complete while important responsibilities remain hidden.

Chapter 8

Conclusion

This thesis compared four AI-assisted backend-generation workflows: Vibe, Self-Spec, Iterative, and Spec Kit. The workflows were tested on two backend API applications with fixed SRSs, fixed API contracts, and automated scenario harnesses. Across 240 generated implementations, the study measured functional requirement coverage, token use, estimated cost, duration, and selected code-structure differences.

For functional outcomes, the clearest result is that more automated scaffolding did not consistently improve functional requirement coverage. Vibe and Self-Spec gave the strongest practical trade-off. They reached high FR coverage with fewer tokens, lower estimated cost, and shorter duration than the more scaffolded workflows. Spec Kit and Iterative used more steps and more model budget, but their average FR coverage was not clearly higher.

The code-structure review showed a different pattern. Spec Kit produced the clearest organization among the inspected implementations. Its generated backends were split into more focused files and had clearer separation between routes, services, repositories, models, and schemas. The lighter workflows could still reach high functional scores, but they more often produced compact or concentrated implementations. This shows that functional behavior and code structure should not be treated as the same outcome.

The main conclusion is therefore not that one workflow is always best. In this benchmark, lighter workflows were more efficient for clear greenfield backend API applications, while Spec Kit showed more value for code organization. More workflow scaffolding was not automatically better. Specifications, plans, and task lists can structure the work, but they only help if they lead the implementation toward the right behavior.

This distinction matters because the Spec Kit condition was fully automated. The agent generated the specification, plan, and task list, and later phases used those documents without human correction. A human-guided SDD process could behave differently. A developer could review the artifacts, correct wrong assumptions, and clarify missing behavior before implementation. The results should therefore be read as a indication about automated SDD-style scaffolding, not as a complete judgment of SDD as a development practice.

The scope of the result is limited by the benchmark design. The study used greenfield backend API applications with clear requirements and fixed API contracts. This made the workflows comparable and made automated evaluation possible, but it also reduced the amount of ambiguity the workflows had to resolve. The results do not directly show how the same workflows would perform on frontend applications, full-stack, brownfield codebases, or development with several prompts.

Future work should test less controlled settings. This could include human-reviewed Spec Kit, applications where the agent helps shape the API or clarify requirements, Vibe with clearer guidance about code organization and brownfield applications inside existing codebases. More applications, models, and agent harnesses would also show whether the same pattern appears outside this benchmark.

The overall lesson is that generated software should be judged by its results, not by how complex

the workflow is. A simple workflow can be efficient when the task is clear, but good structure still matters when the code must be inspected, extended, or maintained. The strongest workflow therefore depends on what the developer needs most: fast functional output, low model cost, or a codebase that is easier to understand after it works.

Appendix A

Experiment Materials

A.1 Software Requirements Specifications

The final experiment uses two SRS documents:

- Hospital Management Application, with 17 functional requirements.
- Chess Tournament Application, with 22 functional requirements.

The experiment-ready SRS files used by the runner are stored in `srs/hospital.md` and `srs/chess.md`. They are copied into each run directory as `srs.md` before generation starts. The SRS remains the source of truth for required behavior.

A.2 Fixed API Contracts

For the hospital and chess applications, the runner also copies a fixed HTTP API contract into each run directory as `api_contract.yaml`. These contracts are stored as `contracts/hospital_api_contract.yaml` and `contracts/chess_api_contract.yaml`. They define the paths, methods, request fields, response fields, and OpenAPI-shaped operation metadata expected by the scenario harness.

The fixed contract is not a behavioral oracle. It standardizes the API surface so that the comparison is not dominated by endpoint naming or payload-shape choices. FR credit still depends on the scenario tests observing the required behavior and state changes from the SRS.

A.3 Prompt Templates

The prompt material used by the runner is stored in `prompts/`. It contains:

- the shared preamble;
- the Vibe workflow prompt;
- the Self-Spec `/plan` planning prompt;
- the Self-Spec implementation prompt;
- the Iterative decomposition prompt;
- the Iterative implementation prompt; and
- the Spec Kit `/speckit.specify`, `/speckit.plan`, `/speckit.tasks`, and `/speckit.implement` prompts.

The shared preamble defines local execution constraints, mocked external services, real-auth expectations, documented local evaluation access for protected roles, and contract-first interface handling. The workflow templates insert this preamble where needed. The prompts instruct the generated backend to use `srs.md` as the required behavior and `api_contract.yaml` as the

HTTP interface contract. These prompt controls support local execution and testability without replacing either the SRS or the fixed API contract.

Appendix B

Benchmark Implementation and Configuration

B.1 Runner and Workflow Code

The runner files and experiment artifacts referenced in this appendix are available in the open-source repository at `https://github.com/sailedev/bachelor-thesis-2026`.

The main runner entry point is `run.py`. It creates a fresh run directory, copies the selected task inputs, executes the chosen workflow, writes metadata, and starts evaluation after generation. Parallel execution for the main result set is handled by `run_parallel.py`.

Workflow-specific orchestration is implemented in `runner_core/workflows.py`. This file defines the Vibe, Self-Spec, Iterative, and Spec Kit turn structures. Codex execution, Docker commands, preflight checks, JSONL logging, and token extraction are implemented in `runner_core/codex_runner.py`. Run-directory naming and file collection are handled by `runner_core/run_storage.py`.

B.2 Scenario Harnesses

The scenario harnesses are stored under `tests/`. The hospital harness is implemented in `tests/hospital/`, and the chess harness is implemented in `tests/chess/`. Shared helpers for HTTP calls, contract handling, FR result records, and OpenAPI helpers are stored in `tests/common/`.

The harnesses are external to the generated applications. They start the saved backend, call it through the fixed contract, and write FR-level results.

B.3 Measurement Rubric

The measurement rubric is described in the evaluation method and encoded in the scenario-harness result records. The shared result code in `tests/common/fr_results.py` records the MET, PARTIAL, BLOCKED, and MISS statuses, calculates the strict and lenient coverage scores, and records endpoint-source audit fields. The comparison utility `compare_results.py` then reads these records alongside `metadata.json` to aggregate coverage, token use, timing, and estimated cost.

Endpoint selection is contract-first when a fixed contract exists. Optional live OpenAPI drift helpers are kept separately in `tests/common/openapi.py` and `tests/common/contract_drift.py`.

B.4 Run Outputs and Traceability

Completed runs are stored under `runs/results/`. Each run folder preserves the fixed inputs, prompts, generated files, Codex event stream, metadata, server logs, and test results for that

run. The most important files are:

- `srs.md`: the SRS copied into the run directory
- `api_contract.yaml`: the fixed contract, when available
- `prompt*.txt`: the prompt text for the workflow phase or turn
- `run.jsonl`: the Codex event stream used for token accounting
- `metadata.json`: workflow, model, timing, token, and file metadata
- `test_results.json`: FR-level evaluation results
- `server.log`: backend output captured during scenario-harness evaluation

Token counts are measured from `run.jsonl`. Estimated dollar costs in the results are calculated from those token counts using public API prices. Because the runs used Codex through an OAuth subscription rather than direct API billing, the dollar values are comparable estimates rather than actual charged amounts.

B.5 Model and Harness Configuration

The benchmark was executed with the following fixed generation configuration and per-run logging setup:

- Codex CLI version `0.121.0`.
- Model `gpt-5.4`.
- Reasoning efforts included in the reported result set: `none`, `low`, and `medium`.
- Code-structure review reasoning effort: `medium`.
- Docker image `bachelor-runner-codex:0.121.0`.
- Spec Kit commit `13d88d22a64b77aa6ee481d079c35d74fa10bacb`.
- Per-run Docker preflight written to `docker-preflight.json`.
- Per-run event logging to `run.jsonl`.
- Final agent message written to `_last.txt`.
- Fixed API contract copied as `api_contract.yaml` when available.

The Docker image is defined in `docker/Dockerfile`. It is based on `node:20-bookworm` and installs Python, Node, SQLite, Git, `jq`, `ripgrep`, `curl`, the Codex CLI, `uv`, and the pinned Spec Kit CLI commit listed above.

Each Codex turn is executed inside Docker with only the run directory mounted as `/work` and a temporary Codex authentication/state directory mounted at `/root/.codex`. Initial turns use `-cd /work`, `-skip-git-repo-check`, and `-json`. Single-turn workflows use `-ephemeral`. Multi-phase workflows keep the temporary Codex state mount for the duration of the run and resume later turns by explicit `thread_id`. Codex's inner sandbox is disabled inside the container because nested Linux namespace sandboxing is not available there; Docker is the isolation boundary.

B.6 Isolation Verification

An early pre-Docker check showed that the Codex `workspace-write` sandbox restricted writes but did not reliably prevent reads elsewhere on the host file system. The runner was therefore changed to use Docker isolation for generation.

The final isolation policy is based on mount control rather than host sandbox read restrictions. The generation container receives only the current run directory and the temporary Codex auth/state mount. Previous runs and project source directories are not mounted into the container. A Docker preflight check verifies the mounted work directory, required tools, and package registry connectivity before generation proceeds.

Bibliography

[1] H. Jin, L. Huang, H. Cai, J. Yan, B. Li, and H. Chen, *From LLMs to LLM-based agents for software engineering: A survey of current, challenges and future*, 2024. DOI: 10.48550/arXiv.2408.02479. arXiv: 2408.02479 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2408.02479.

[2] J. Liu et al., "Large language model-based agents for software engineering: A survey," *ACM Transactions on Software Engineering and Methodology*, 2026. DOI: 10.1145/3796507.

[3] D. B. Piskala, *Spec-driven development: From code to contract in the age of ai coding assistants*, 2026. arXiv: 2602.00180 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2602.00180.

[4] L. Bai et al., *How do AI agents spend your money? analyzing and predicting token consumption in agentic coding tasks*, 2026. DOI: 10.48550/arXiv.2604.22750. arXiv: 2604.22750 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2604.22750.

[5] M. Akhoroz and C. Yildirim, "Conversational ai as a coding assistant: Understanding programmers' interactions with and expectations from large language models for coding," *arXiv preprint arXiv:2503.16508*, 2025. DOI: 10.48550/arXiv.2503.16508. [Online]. Available: https://arxiv.org/abs/2503.16508.

[6] J. Yang et al., "Swe-agent: Agent-computer interfaces enable automated software engineering," *arXiv preprint arXiv:2405.15793*, 2024. DOI: 10.48550/arXiv.2405.15793. [Online]. Available: https://arxiv.org/abs/2405.15793.

[7] H. F. Hofmann and F. Lehner, "Requirements engineering as a success factor in software projects," *IEEE Software*, vol. 18, no. 4, pp. 58–66, 2001. DOI: 10.1109/MS.2001.936219. [Online]. Available: http://www.ics.uci.edu/~wscacchi/SA/Readings/Req-Engr-SuccessFactors-Software-July01.pdf.

[8] A. Karpathy, *Vibe coding (x post)*, Originating use of the term "vibe coding", Feb. 2025. [Online]. Available: https://x.com/karpathy/status/1886192184808149383.

[9] GitHub, *Spec-kit: A toolkit for spec-driven development with ai*, Accessed: 2026-03-13, 2025. [Online]. Available: https://github.com/github/spec-kit.

[10] L. Griffin and R. Carroll, *Spec driven development: When architecture becomes executable*, InfoQ, Jan. 2026. [Online]. Available: https://www.infoq.com/articles/spec-driven-development/.

[11] OpenAPI Initiative, *Openapi specification*, Accessed 2026-05-02, 2025. [Online]. Available: https://spec.openapis.org/oas/latest.html.

[12] X. Jiang et al., "Self-planning code generation with large language models," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–30, 2024. DOI: 10.1145/3672456.

[13] H. Ding et al., "Reasoning and planning with large language models in code development," in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2024, pp. 6480–6490. DOI: 10.1145/3637528.3671452.

[14] N. Pathak, *What is an agent harness? the infrastructure that makes ai agents actually work*, https://www.firecrawl.dev/blog/what-is-an-agent-harness, Firecrawl blog. Accessed: 2026-05-14, Apr. 2026.

[15] OpenAI, *Codex cli*, Version 0.121.0; accessed 2026-04-22, 2026. [Online]. Available: https://github.com/openai/codex.

[16] OpenAI, *Tokenizer*, Accessed: 2026-05-13, 2026. [Online]. Available: https://platform.openai.com/tokenizer.

[17] OpenAI, *Prompt caching*, OpenAI API documentation; accessed 2026-05-10, 2026. [Online]. Available: https://platform.openai.com/docs/guides/prompt-caching.

[18] OpenAI, *Api pricing*, Accessed: 2026-05-13, 2026. [Online]. Available: https://openai.com/api/pricing/.

[19] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, Available at: https://proceedings.neurips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html, 2022.

[20] OpenAI, *Introducing gpt-5.4*, https://openai.com/index/introducing-gpt-5-4/, Accessed: 2026-05-13, Mar. 2026.

[21] S. Miserendino, M. Wang, T. Patwardhan, and J. Heidecke, "Swe-lancer: Can frontier llms earn $1 million from real-world freelance software engineering?" *arXiv preprint arXiv:2502.12115*, 2025. DOI: 10.48550/arXiv.2502.12115. [Online]. Available: https://arxiv.org/abs/2502.12115.

[22] A. Cuadron et al., "The danger of overthinking: Examining the reasoning-action dilemma in agentic tasks," *arXiv preprint arXiv:2502.08235*, 2025. DOI: 10.48550/arXiv.2502.08235. [Online]. Available: https://arxiv.org/abs/2502.08235.

[23] OpenAI, *Slash commands in codex cli*, https://developers.openai.com/codex/cli/slash-commands, Accessed 2026-05-13, 2026.

[24] OpenAI, *Config basics: Learn the basics of configuring your local codex client*, https://developers.openai.com/codex/config-basic, Accessed 2026-05-13, 2026.

[25] OpenAI, *Agent approvals & security*, Codex documentation; accessed 2026-04-22, 2026. [Online]. Available: https://developers.openai.com/codex/agent-approvals-security.

[26] GitHub, *Specification-driven development*, https://github.com/github/spec-kit/blob/main/spec-driven.md, Accessed: 2026-05-15, 2026.

[27] P. Taghavi and S. Bhavani, "Spec kit agents: Context-grounded agentic workflows," *arXiv preprint arXiv:2604.05278*, 2026. DOI: 10.48550/arXiv.2604.05278. [Online]. Available: https://arxiv.org/abs/2604.05278.

[28] S. Feng, B. Chen, B. H. Meyer, and G. Mussbacher, "Llm-assisted repository-level generation with structured spec-driven engineering," *arXiv preprint arXiv:2605.02455*, 2026. [Online]. Available: https://arxiv.org/abs/2605.02455.

[29] B. Hill, *Does spec-driven development reduce defects? an empirical test of industry claims across 119 open-source repositories*, SSRN, 2026. [Online]. Available: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6515898.

[30] OpenAI, *Text generation*, https://platform.openai.com/docs/guides/text-generation, Accessed: 2026-05-13, 2026.

[31] Wikipedia contributors, *Swiss-system tournament*, Accessed 2026-04-22, 2026. [Online]. Available: https://en.wikipedia.org/wiki/Swiss-system_tournament.

[32] International Chess Federation (FIDE), *Fide handbook, c.04 general handling rules for swiss tournaments*, Accessed 2026-04-22, 2024. [Online]. Available: https://handbook.fide.com/chapter/C0402.

[33] OpenAI, *Codex cli*, https://developers.openai.com/codex/cli, Accessed: 2026-05-13, 2026.

[34] OpenAI, *Codex*, https://github.com/openai/codex, Accessed: 2026-05-13, 2026.

[35] Michigan Technological University, *What is software engineering?* https://www.mtu.edu/cs/undergraduate/software/what/, Accessed: 2026-05-14, 2026.

[36] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, *The impact of ai on developer productivity: Evidence from github copilot*, 2023. DOI: 10.48550/arXiv.2302.06590. arXiv: 2302.06590 [cs.SE].

[37] N. F. Liu et al., "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024. DOI: 10.1162/tacl_a_00638.

[38] G. O'Brien, A. Parker, N. Eisty, and J. Carver, *A survey of generative ai adoption and perceived productivity among scientists who program*, 2026. DOI: 10.48550/arXiv.2512.19644. arXiv: 2512.19644 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2512.19644.

[39] T. Eloundou, S. Manning, P. Mishkin, and D. Rock, *Gpts are gpts: An early look at the labor market impact potential of large language models*, 2023. DOI: 10.48550/arXiv.2303.10130. arXiv: 2303.10130 [econ.GN].

[40] World Economic Forum, *The future of jobs report 2025*, Published: 7 January 2025, 2025. [Online]. Available: https://www.weforum.org/publications/the-future-of-jobs-report-2025/.

[41] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," *Communications of the ACM*, 2025. DOI: 10.1145/3610721.

[42] Anthropic, *Project glasswing*, Accessed: 2026-05-02, 2026. [Online]. Available: https://www.anthropic.com/project/glasswing.

[43] L. H. Newman, *Anthropic teams up with its rivals to keep ai from hacking everything*, WIRED, Published: 7 April 2026; accessed: 2026-05-02, Apr. 2026. [Online]. Available: https://www.wired.com/story/anthropic-mythos-preview-project-glasswing/.

[44] E. Kovacs, *Claude mythos finds 271 firefox vulnerabilities*, SecurityWeek, Published: 22 April 2026; accessed: 2026-05-02, Apr. 2026. [Online]. Available: https://www.securityweek.com/claude-mythos-finds-271-firefox-vulnerabilities/.

[45] J. Lyons, *Nobody knows how many cves anthropic's project glasswing has actually found*, The Register, Published: 15 April 2026; accessed: 2026-05-02, Apr. 2026. [Online]. Available: https://www.theregister.com/2026/04/15/project_glasswing_cves/.