Coordination Protocols for Multi-Agent Systems in Software Development

by

PHUC DUONG

A SENIOR THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREES OF
BACHELOR OF SCIENCE AND MASTER OF SCIENCE
IN COMPUTER SCIENCE

Advisor: Professor Timos Antonopoulos

Department of Computer Science
Yale University
December 11, 2025

Acknowledgements

I would like to thank Anthropic for partially supporting this work through API credits provided by the Claude Builder Club program. Their support made it possible to run the experiments included in this thesis.

I am deeply grateful to Professor Timos Antonopoulos for his continued guidance and support throughout this project. My interest in this topic first emerged in his Advanced Software Engineering course, where I was introduced to many of the concepts and technologies that shaped the direction of this thesis. His mentorship and feedback were central in helping me develop and refine the ideas behind this work.

I would also like to thank the Yale Computer Science Department for creating an environment that genuinely supports academic growth. I am grateful for the resources, mentorship, and opportunities the department provides, as well as the community of students and faculty who have made my time here meaningful. The support and culture within the department have played a major role in shaping my experience at Yale and beyond.

Contents

Coordination Protocols for Multi-Agent Systems in Software Development

Phuc Duong

Abstract

Coordinating multiple large language model (LLM) agents on software engineering tasks remains an open challenge even as single-agent systems continue to improve at code generation and debugging. Understanding how to organize multi-agent architectures can help us develop systems that can carry out more complex, multi-step software development workflows with little human oversight. In this work, we design and evaluate four distinct coordination protocols, Builder–Critic, Leader–Worker, Voting, and a role-specialized pipeline. We evaluate these systems across short-horizon tasks using HumanEval+, real-world bug fixing with SWE-bench Verified, and long-horizon end-to-end development projects. Our results show that lightweight feedback loops yield notable gains on short tasks, whereas more complex multi-agent structures introduce significant coordination overhead without improving outcomes on repository-level issues. In open-ended development projects that require extended planning, creativity, and coordination across development phases, approaches based on role differentiation show clearer advantages, yet they come at higher computational cost with increased runtime overhead. These findings characterize the trade-offs between coordination structure, accuracy, and computational cost in multi-agent orchestration and offer guidance for designing scalable multi-agent systems for different classes of software engineering tasks. The source code is available at https://github.com/phucd5/multiagent-orchestration.

1 Introduction

The rise of AI agents has transformed how programming tasks can be delegated and completed. These systems enable automation of tasks that were once fully manual. Yet while they excel at solving narrowly defined problems, coordinating multiple agents to tackle complex, multi-step software engineering tasks remains an open challenge. This project investigates coordination protocols for multi-agent systems, drawing inspiration from organizational structures in human software engineering teams and communication patterns.

We evaluated approaches such as voting, leader-worker, builder-critic, and specialist pipelines to examine how different coordination strategies shape the effectiveness of collaborative problem solving among AI agents across a range of task horizons, from short function generation to longer end-to-end software development projects. The results highlight distinct trade-offs between accuracy and coordination overhead such as latency and cost, providing insights into how to design multi-agent systems for software engineering tasks.

2 Background

2.1 Current Landscape

Agents. AI agents are systems that autonomously perform various tasks. These systems are driven by LLMs that extend beyond single-turn generation methods by operating through iterative reasoning loops, tool use while adapting to feedback. This continuous structure allows agents to plan, execute, and refine actions across multiple steps, supporting more complex workflows such as implementing new functions, writing and running unit tests, reviewing code, and integrating components into existing codebases. Agents are increasingly embedded into modern development environments (e.g., Claude Code, GitHub Copilot, OpenAI Codex, Google Antigravity) [1, 2, 3, 4] to help boost productivity in software development.

Tools. Tools are mechanisms through which agents interact with external systems. They provide access to code execution environments, file systems, package managers, and API endpoints. These tools enable agents to perform tasks that a human developer would carry out, such as writing code to files, installing dependencies, running programs, and inspecting outputs. Many of these tools are exposed through a Model Context Protocol Server (MCP) [5], which provides a standardized way for agents to discover available tools and call them.

Subagents. With the rise of agent-based systems, subagents have emerged as a way to delegate specialized parts of a task. Subagents are independent AI agents configured with task-specific prompts and context windows that can be spawned by other agents. These subagents can specialize in roles such as debugging, test generation, or code review. A key advantage of this structure is that the use of multiple agents can help mitigate context-length limitations, where long input histories can degrade reasoning quality and coherence [6]. Because each subagent has its own context window, it can carry out focused tasks locally and return results to an orchestrator without introducing the full intermediate history into the shared context. This separation supports more complex long-horizon workflows. Additionally, spawning multiple subagents can help with parallelization, since different subagents can work on separate parts of a task at the same time.

2.2 Related Works

Existing Agent Systems. Systems such as OpenAI Codex and GitHub Copilot demonstrated the feasibility of automating software engineering tasks through a single agent paradigm capable of navigating codebases, editing files, and submitting pull requests. Cursor [7] similarly support this workflow within Visual Studio Code with access to modern LLMs such as Claude, GPT, and Gemini as autonomous peer programmers. While these systems primarily adopt a single agent approach, Claude Code extends this direction by introducing subagents that can be invoked by an orchestrator to handle specialized tasks. However, no formal protocols in these existing systems currently defines how multiple agents should coordinate, communicate, or share intermediate results in a structured and scalable way.

Role-Based Multi-Agent Systems. A growing body of work explores how to structure teams of LLM agents so they resemble real software organizations. Qian et al. introduce ChatDev [8], a chat centered framework where specialized agents take on roles such as CEO, CTO, programmer, and designer, and collaborate through multi turn dialogue in a chat chain structure. Hong et al. propose MetaGPT [9], a meta programming framework that encodes human Standard Operating Procedures into a multi agent software company where each agent has a domain specific role and coordination follows an assembly line style workflow. Nguyen et al.'s AgileCoder [10] integrates agile methodology into multi agent systems, while Ha et al.compare software development processes such as Waterfall and Agile across these systems [11]. Collectively, these systems show that role specialization can improve task organization.

Protocol-Driven Coordination. Coordination failures remain common in multi-agent LLM workflows. SEMAP [12] addresses this issue by enforcing behavioral contracts, structured message formats, and phase level verification to reduce error rates. RTADev [13] introduces intention alignment checks and lightweight group reviews to prevent role drift and miscommunication during multi agent interactions. These approaches highlight the importance of explicit coordination protocols, although they largely focus on behavioral guarantees rather than scalable task decomposition across many agents.

General Orchestration Frameworks. AutoGen [14] provides reusable patterns for delegating tasks among interacting LLM agents, supporting manager worker structures and tool augmented collaboration. HuggingGPT [15] uses a controller model to plan and route subtasks to specialized models through natural language specifications. Microsoft's orchestration patterns [16] formalize templates such as sequential pipelines and group decision loops. While these frameworks provide flexible building blocks for multi-agent collaboration, they do not specifically prescribe how agents should coordinate in complex software engineering tasks or how to balance autonomy and oversight.

3 Methodology

3.1 Coordination Protocols

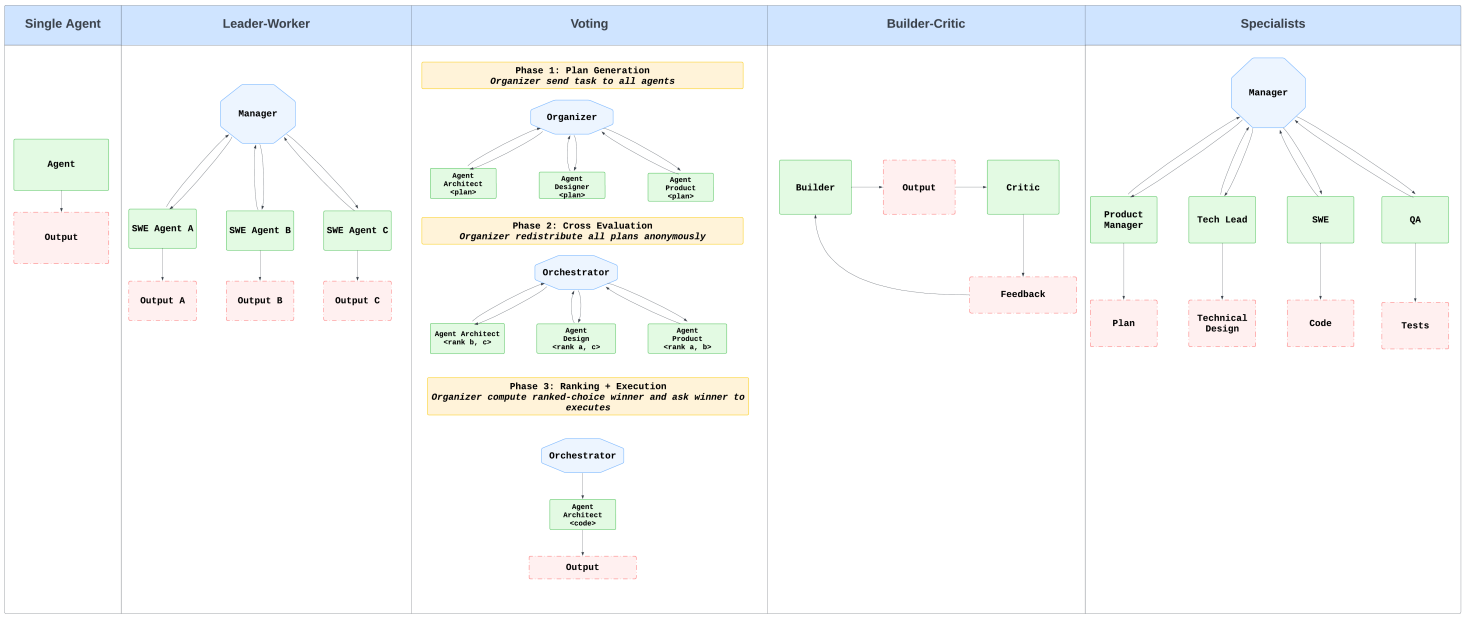


Figure 1: Overview of the coordination protocols evaluated

Coordinating multiple agents to solve software engineering tasks presents a significant challenge because these systems require effective strategies for collaboration, delegation, and role distribution. The core difficulty lies in designing protocols that enable agents to jointly plan, specialize, verify, and adapt across the stages of software development.

We developed four coordination protocols inspired by human software engineering teams and the patterns people use when making collective decisions. We utilized a single-agent system as a baseline to evaluate these protocols, as this paradigm reflects the workflow used by most AI-powered coding assistants today.

Builder-Critic. In industry, code is often reviewed by another developer for validation of design, functionality, and correctness. This process serves as a second pass to ensure that high-quality code is produced. Agents can similarly benefit from having their outputs evaluated by another agent to guard against errors or hallucinations. In this protocol, a builder agent produces an initial solution, and a critic agent evaluates the result, provides

feedback, and issues a decision on whether it is `APPROVED`. Based on the feedback, the builder refines the solution, and this iteration cycle repeats until the critic accepts the final result. Both agents are assigned the role of software engineer, with the critic receiving additional guidelines that require it to provide structured feedback during evaluation.

Leader-Worker. Inspired by hierarchical approaches to task delegation in both the tech industry and general workplace settings, this protocol introduces a central leader that delegates tasks to a team of three homogeneous worker agents. Upon completion of their assignments, each worker reports its result back to the leader, who then uses these outputs to delegate additional tasks or request revisions. The leader is assigned the role of an engineering manager, managing a team of three software engineer worker agents. We allowed the leader to delegate to as many workers as it deemed necessary (maximum 3), or as few as needed, depending on task complexity, mirroring how managers scale teams or save resources in real settings.

Voting. Majority consensus is often used to make decisions in industry and resembles the self-consistency technique in which multiple reasoning paths are sampled and the most consistent answer is selected [17]. In this protocol, we use three agents configured with distinct archetypes, a Systems Architect, a Coding Machine, and a Product Hybrid, to introduce variability in reasoning so that convergence may indicate higher reliability. An organizer oversees the voting process but does not contribute to solving the task.

We structure the protocol into three phases:

1. **Proposal Phase.** The organizer sends the task to all agents, and each agent independently generates a plan for solving it.
2. **Review Phase.** The organizer collects all plans, redistributes them anonymously, and asks each agent to rank every plan, including its own, which is anonymized in the same way, and provide feedback.
3. **Selection and Execution Phase.** The organizer computes the outcome using a traditional ranked choice voting procedure and assigns the winning plan to be implemented by the agent that produced it, incorporating feedback from all reviewers.

Specialists. The most traditional use of subagents is to assign each one a specific role or area of expertise and have it focus on a narrow part of the task. This approach leverages the large context windows available to each subagent, allowing them to work more deeply within their assigned responsibilities. In this protocol, a manager agent delegates tasks to a set of specialized subagents. Each subagent reports its results back to the manager, who can use these outputs when issuing subsequent instructions. The four subagents are assigned the roles of Tech Lead, Product Manager, Software Engineer, and QA Engineer to mimic roles in the tech industry. The typical workflow proceeds with the Product Manager defining a plan, the Tech Lead refining or validating it, the Software Engineer implementing the solution, and the QA agent testing the output.

3.2 Technical Implementation

Our system is built on top of the Claude Agent SDK (`0.1.14`), which provides performance optimizations, built-in agent tools (e.g., file operations, code execution, MCP support), and runtime statistics (e.g., token usage, latency costs). These integrated features allow us to focus on developing the coordination protocols. Each agent and subagent uses the `claude-haiku-4-5-2025100` model with thinking disabled [18].

For communication between agents, we implemented an MCP server that runs within the same process rather than relying on an external MCP setup. This design avoids IPC overhead and allows agents to be tracked by `id` and addressed directly within the program. The server exposes a `communicate_with_agent` tool that takes an `agent_id` and a `message` and routes the message to the corresponding agent instance. Each subagent is instantiated as a `ClaudeSDKClient` to allow for continuous conversation, which removes the need to manually manage conversation state.

To maintain stable control flow and avoid unintended communication loops, this communication tool is exposed only to the coordinator agent, such as the engineering manager or leader. The coordinator is responsible for all cross-agent messaging and ensures that interactions follow the intended protocol structure. Future work could explore allowing subagents to communicate directly with one another.

3.3 Evaluation

HumanEval+. To assess the system's performance on shorter and simpler tasks, we used the HumanEval+ benchmark [19]. HumanEval+ is based on the original HumanEval benchmark [20], but it includes additional test cases designed to catch subtle bugs or edge cases that may be missed by the smaller test suite in the original version. The benchmark focuses on single-function implementations such as "return a list of all prefixes from shortest to longest of the input string" and measures the ability of each coordination protocol to correctly produce reliable solutions for short-horizon tasks.

Due to the costs of running evaluation, we assigned a strict budget of 40 turns per agent and subagent when running the evaluation. An attempted solution must be produced and submitted through a file creation within this limit, otherwise, the task is marked as incorrect. All experiments used a single evaluation pass per problem. We also enforced a test execution timeout of 5 minutes per task to prevent extremely inefficient solutions.

SWE-bench Verified. To evaluate our system on real-world software engineering tasks, we used the SWE-bench Verified benchmark [21, 22]. This benchmark consists of GitHub issues that have been verified as solvable by human software engineers, allowing the testing of what different coordination protocols perform better when applied to real development scenarios in pre-existing codebases.

We assigned a strict budget of 100 turns per agent and subagent. The system is responsible for generating a valid patch through `git diff` that resolves the issue. After applying the patch, the solution must pass both the tests that previously passed (`PASS_TO_PASS`) and the tests associated with the issue (`FAIL_TO_PASS`) in order to be considered correct. If the system does not produce a patch, we still ran the environment without any modifications for evaluation. All experiments used a single evaluation pass per problem, and we ran it on 50 randomly sampled problems from the benchmark, divided equally between "5 min–1 hour" and "<15 min fix" difficulty. All tests and agent work were run on a Docker image from the Epoch AI image registry for SWE-bench [23].

End-To-End Tasks. To evaluate the system on long-horizon tasks, we examined its ability to complete end-to-end software engineering projects given only high-level functional requirements. We designed two projects, one full-stack application and one machine learning task with a frontend component, to emulate projects that software engineers might have to complete as part of their job. These projects assess whether the agents can interpret requirements, design a solution, and produce a functional implementation.

1. **Full-Stack Project:** This project consists of a React frontend paired with a FastAPI backend. The task is to build a personal finance tracker with user accounts, a financial summary showing income, expenses, and balance, and visualizations of income and expense breakdowns.
2. **Machine Learning and Frontend Project:** This project includes a React frontend and a Python-based machine learning module conntected with Flask. The task is to build an end-to-end workflow that loads a small text dataset from scikit-learn, trains and evaluates Logistic Regression and Naive Bayes, and exposes the evaluation results to a frontend with a graph.

For each project we provided an initial template with basic starter code. We assigned a strict budget of 100 turns per agent and subagent. The agent is asked to provide a command for running the application. If it did not produce such a command within the budget, we still performed a best-effort evaluation by attempting to run the generated code as is before assigning a non-functional score.

We evaluated each project using the rubric shown below, which provides a simple scoring scheme for assessing output quality.

Dimension	Description	Scale
Task Completion	Did the system produce a full project that appears complete?	0–3
Functional Correctness	Does the produced system run end to end without fatal errors?	0–3

Table 1: Rubric used to evaluate long horizon project quality.

Score	Meaning
3	Full success
2	Mostly works
1	Partial output
0	No meaningful progress

Table 2: Interpretation of the 0–3 scoring scale in long horizon project rubric.

4 Results

4.1 HumanEval

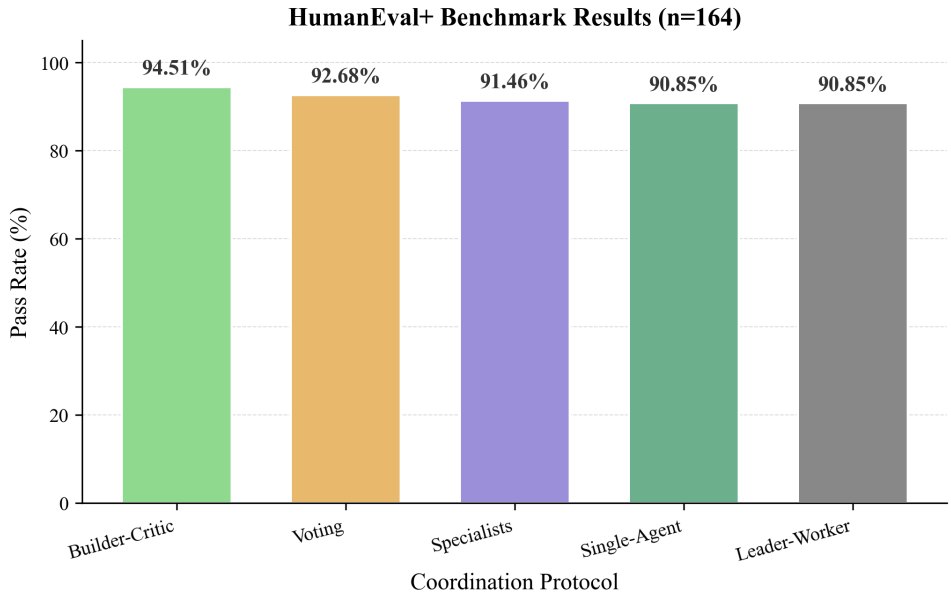


Figure 1: Pass rates on the HumanEval+ benchmark (n=164) across the four coordination protocols and the single-agent baseline.

Across the full HumanEval+ benchmark of 164 tasks, the Builder-Critic protocol achieved the highest pass rate at 94.51% with a single pass over the dataset. Voting performed similarly at 92.68%, followed by Specialists at 91.46%. Leader-Worker and the Single-Agent baseline both achieved 90.85%.

The gains over the single-agent baseline may be attributed to the refinement stages present in the three highest-scoring protocols. Leader-Worker falls short of this benefit because, depending on the task complexity, it may delegate the entire problem to a single agent without any external verification, effectively reducing the protocol to a single-agent workflow. In contrast, Builder-Critic, Voting, and Specialists consistently incorporate external feedback from other agents, which likely contributes to their higher pass rates.

Builder-Critic achieved the strongest performance, suggesting that an external feedback loop contributes to producing more correct code. Voting displays a similar advantage,

since the ranking stage requires each agent to evaluate and comment on other agents' proposals, and this collective feedback is later incorporated into the final implementation. Although the single agent performs iterative testing and can self-correct to some extent, its revisions are constrained by a single perspective and a single chain of reasoning, limiting its ability to catch subtle errors or edge cases. While the Specialists pipeline includes a QA stage that extensively validates the output, the lack of direct and repeated interaction between roles means that important details in the QA feedback may not translate into effective improvements. Because feedback is passed through a coordinator and not integrated into an iterative loop, some information can be lost or diluted due to communication overhead, making the validation less impactful than the tighter feedback mechanisms in Builder-Critic and Voting. These results suggest that lightweight iterative correction can improve accuracy on shorter-horizon tasks more effectively than heavier coordination structures such as Specialists or Leader-Worker.

Protocol	**Avg. Turns**	**Avg. Duration (s)**	**Avg. Cost (USD)**
Single-Agent	3.15	45.00	0.026
Leader-Worker	9.51	146.83	0.074
Builder-Critic	8.21	43.65	0.058
Specialists	24.32	265.12	0.323
Voting	14.62	269.79	0.175

Table 1: Execution cost and latency comparison across coordination protocols on HumanEval+, averaged over 164 tasks.

When normalizing by pass rate, the single-agent baseline has the lowest cost per successful solution. Leader-Worker matches its accuracy but requires substantially more computation, resulting in a higher cost per success. This aligns with the fact that Leader-Worker introduces additional coordination overhead even on relatively simple problems where delegation does not yield an efficiency advantage.

Looking at the other protocols, Builder-Critic increases cost per success by roughly 2.1x relative to the baseline. Specialists and Voting are substantially more expensive per passing solution, with increases of about 12.3x and 6.6x respectively, and are dominated by Builder-Critic in both accuracy and efficiency. These results are in line with expectations, since Builder-Critic introduces a feedback loop that adds minimal overhead, while Specialists and Voting require multiple agents to generate, review, or compare intermediate outputs. This amplifies token usage even on relatively simple tasks, leading to higher costs.

In terms of latency, Builder-Critic converged faster than the other protocols and even outperformed the single-agent baseline. This may reflect the benefit of the builder-critic loop, which avoids the long, monolithic reasoning steps often observed in single-agent trajectories. Although the single agent takes fewer turns, those turns tend to be longer and more self-reflective, occasionally leading to extended cycles of revision before committing to

a final answer. By comparison, the builder tends to finalize once it receives confirmation from the critic, resulting in a faster end-to-end path even with additional turn-level interactions. This reliance on external validation, rather than speculative self-checking, contributes to the slight decrease in average latency. Pipeline-style protocols such as Specialists and branching-style protocols such as Voting introduce significant synchronization delays, and Leader-Worker experience delegation bottlenecks that further slowed execution.

Taken together, these results suggest that the primary source of improvement on short-horizon tasks is not elaborate role structures or multi-stage coordination but the introduction of succinct external feedback. Coordination protocols such as Builder-Critic, which incorporate lightweight refinement loops, deliver the strongest accuracy gains while maintaining competitive latency and cost relative to more complex multi-agent structures. By comparison, Voting and Specialists rely on diversity of reasoning or division of labor to provide additional value however, on HumanEval+ tasks, where reasoning paths are typically short, this diversity does not translate into significantly higher accuracy, as the coordination overhead outweighs the benefit. Overall, these findings indicate that effective multi-agent systems for short and simple tasks may require far less structural complexity than previously assumed.

4.2 SWE-bench Verified

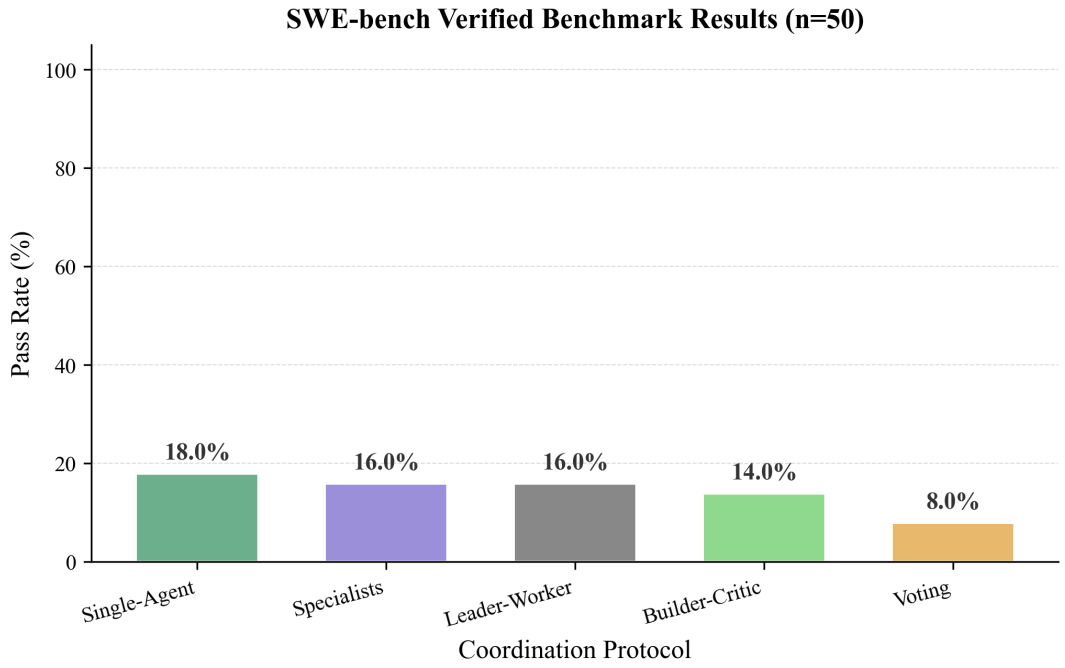


Figure 2: Pass rates on the SWE-bench Verified benchmark (n=50) coordination protocols.

Protocol	Avg. Turns	Avg. Duration (s)	Avg. Cost (USD)
Builder-Critic	101.22	551.65	0.901
Leader-Worker	84.00	676.61	1.234
Single-Agent	68.56	390.38	0.614
Specialists	117.26	933.85	1.719
Voting	61.54	868.20	1.466

Table 2: Execution cost and latency comparison across coordination protocols on SWE-bench Verified, averaged over 50 tasks.

Performance on SWE-bench Verified was uniformly low across all methods, reflecting the difficulty of generating correct patches for real open-source repositories. The single-agent baseline achieved the highest pass rate at 18.0%, with Specialists and Leader-Worker close behind at 16.0% each. Builder-Critic performed slightly worse at 14.0%, and Voting achieved the lowest rate at 6.0%. We do note that these are only 50 questions out of the 500 questions available, which could reflect properties of the sampled subset rather than the inherent strengths and weaknesses of the Claude Haiku model itself.

In contrast to the HumanEval+ benchmark, Builder-Critic underperformed almost all other protocols in this setting. This suggests that external iterative refinement may be less effective when modifications require integrating information across large, interdependent codebases. Although the critic technically has access to the full repository, its evaluation could be conditioned on the builder's framing of the problem during communication, which can bias the critic toward the builder's interpretation of the issue. If the builder provides an incomplete or overly narrow view of the relevant code context, the critic may overlook dependencies or interactions elsewhere in the repository. As a result, the refinement loop can fail to correct deeper structural mistakes, limiting its effectiveness on more complex tasks.

The same issue may also affect the other protocols, Specialists and Leader-Worker. Bug fixing typically requires tracing dependencies, synthesizing information across files, and forming a consistent view of the repository, capabilities that are weakened when the task is split across agents with partial context or narrowly defined roles. This can lead to misaligned reasoning across agents. In Specialists, in particular, each agent may only see a slice of the problem and depends on upstream roles to surface the relevant information. When the required insight spans multiple components, the pipeline can break down. This fragmentation makes the protocol brittle in settings where issues demand holistic reasoning rather than strictly partitioned expertise. Voting also appears to underperform more than the rest, suggesting that by producing plans independently, the agents fragment the reasoning process and prevent the aggregation step from capturing repository-wide dependencies.

Although these protocols differ in their structure, the results suggest that coordination mechanisms that rely on multiple agents exchanging information are not well suited to

tasks that require forming a coherent view of a large and interconnected repository. They are more expensive with additional latency compared to our single-agent baseline. The key challenge in this benchmark appears to be maintaining consistent reasoning across many files rather than distributing the work across agents. As a result, the coordination strategies explored in this work do not yield improvements on SWE-bench Verified under the current design.

4.3 End-To-End Tasks

Protocol	Task Completion	Functional Correctness	Total Score
Specialists	3	3	6
Voting	3	2	5
Single-Agent	3	2	5
Leader-Worker	3	1	4
Builder-Critic	1	0	1

Table 3: Evaluation scores for the full-stack project across coordination protocols. Total Score is the sum of task completion and functional correctness.

Protocol	Num. Turns	Duration (s)	Cost (USD)
Builder-Critic	114.0	549.37	1.179
Leader-Worker	144.0	946.98	1.556
Single-Agent	68.0	626.01	0.751
Specialists	142.0	934.80	1.582
Voting	101.0	860.59	1.405

Table 4: Execution cost and latency for the end-to-end full-stack project across protocols.

4.3.1 Full-Stack Task

Across the full-stack project of implementing a financial tracker web application, the Specialists protocol achieved the strongest performance, completing all required features and exhibiting the highest level of functional correctness. It was able to both implement the full feature set and also added meaningful enhancements that were not explicitly requested such as edit functionality, expense filtering, and responsive UI behavior.

Voting also performed well. Interestingly, it implemented many of the same features as Specialists in a similar style, likely due to the similar role configuration, yet it fell short in its UI behavior, where the layout occupied only part of the screen.

Single-Agent and Leader-Worker both achieved full task competition but fell short on functional robustness. The Single-Agent system exhibited subtle state-management bugs, such as transaction insertion succeeding only on a second attempt after navigating from the charts view, and incorrectly treated optional fields as required. The Leader-Worker implementation showed more structural inconsistencies, including summary values that failed to update and category-dependent graphs that broke when users introduced new categories, reflecting gaps in coordination across components. Builder-Critic performed the worst, failing to implement working authentication and producing blocking failures that prevented testing of the actual application.

4.3.2 Machine Learning Task

Protocol	Task Completion	Functional Correctness	Total Score
Single-Agent	3	3	6
Specialists	3	3	6
Voting	3	3	6
Leader-Worker	3	2	5
Builder-Critic	3	2	5

Table 5: Evaluation scores for the machine learning task across coordination protocols. Total Score is the sum of task completion and functional correctness.

Protocol	Num. Turns	Duration (s)	Cost (USD)
Voting	23.0	953.15	1.544
Single-Agent	48.0	295.12	0.653
Builder-Critic	66.0	340.44	0.707
Leader	108.0	569.70	0.968
Specialists	133.0	926.25	1.317

Table 6: Execution cost and latency for the machine learning task across protocols.

The scores for the machine learning task are much closer across protocols. This is arguably a much easier task because the machine learning workflow follows a fixed pipeline with limited opportunities for variation in implementation. All five approaches implemented the core requirements, including loading a text dataset, training Logistic Regression and Naive Bayes models, and building the dashboard. The primary distinction was in the evaluation of the model. Both Builder-Critic and Leader-Worker reported only accuracy, omitting precision, recall, and F1-score, which made their solutions functionally incomplete.

In terms of implementation design, Single-Agent used a simple single-file structure for the machine learning logic, while Voting and Specialists demonstrated clearer modular separation and stronger architectural organization. Specialists stood out in particular where its

dashboard, similar to the full-stack task, included additional features such as probability outputs for predictions, dynamic UI animations, and the most polished visual presentation overall.

4.3.3 End-To-End Task Analysis

A broader pattern emerges when comparing these outcomes to the underlying coordination strategies. Both of our highest-performing protocols across both task, Specialists and Voting, relied on agents with distinct roles that provided structured feedback. Specialists used a pipeline-style workflow, while Voting incorporated feedback during the ranking phase that informed the final implementation. This separation of responsibilities, especially in a pipeline manner, also appears to support greater creativity beyond the strict functional requirements, enabling enhancements that improve usability and the overall quality of the final application.

One difference between the two protocols is the presence of an external QA validator. Specialists included a dedicated QA stage that enabled more thorough verification, whereas Voting required the agent implementing the selected plan to validate its own work. This could have contributed to the slight issues observed in UI layout in the full-stack task. These results suggest that in end-to-end development tasks, role-separation structures provide enough independence for agents to focus on well-scoped responsibilities while still allowing downstream stages to correct or refine earlier decisions.

In contrast, protocols without stable specialization struggle to maintain coherence across components. Our Single-Agent system must internally manage planning, implementation, and validation, which increases the likelihood that reasoning shortcuts, context window overload, or overlooked edge cases remain uncorrected, especially with these longer-horizon tasks. Leader-Worker coordination offers some structure but does not enforce the consistent cross-role verification needed to maintain synchronized state across agents, since it is left to the leader to determine when and how verification occurs. Furthermore, fragmentation between agents can arise when multiple workers operate on different parts of the task in parallel without mechanisms to ensure that their contributions remain aligned, particularly if the leader provides insufficient guidance or oversight.

Builder-Critic, in particular, struggles in end-to-end tasks. This could be because its refinement loop depends on the builder's framing of the problem. This is more detrimental for complex tasks. Early misunderstandings in component interfaces, authentication flow, or state management can persist throughout the refinement cycle, since the critic's feedback is conditioned on the information provided by the builder rather than on an independent assessment of the full system. This makes the protocol prone to reinforcing initial errors instead of correcting them.

In terms of efficiency across both end-to-end tasks, Single-Agent remained the most cost-

effective and generally the fastest in overall runtime, reflecting the absence of coordination overhead. Multi-agent protocols were consistently more expensive to execute. Specialists, Leader, and Voting often incurred the highest runtimes and costs, driven in part by their multi-stage workflows and cross-agent communication. Builder-Critic showed more mixed behavior where in some cases it ran only slightly slower than Single-Agent, and in others outperformed it in runtime, though this came at the expense of correctness. These trends highlights that while role-specialized multi-agent approaches can improve output quality, they typically do so with a substantial increase in computational cost relative to single-agent systems.

Overall, these findings suggest that end-to-end development tasks benefit from coordination strategies in which each agent has a clearly defined focus and non-overlapping responsibilities. When agents specialize, they avoid conflicting goals or duplicated reasoning. Protocols without stable specialization require agents to balance planning, implementation, and verification simultaneously, which increases the likelihood of inconsistency and overlooked errors. Multi-agent systems also benefit from the fact that each agent operates within its own context window, effectively increasing the total amount of information the system can hold and reason over at once. Structured role separation therefore remains the most effective approach for producing high-quality end-to-end applications.

5 Discussion

Our results reveal that the effectiveness of multi-agent coordination depends strongly on the structure of the task and the type of reasoning it demands. Short-horizon problems such as HumanEval+ favor lightweight feedback loops. In contrast, repository-level debugging tasks in large and interconnected codebases requires a coherent global view of dependencies, interfaces, and file relationships. Multi-agent protocols that divide context across agents struggle to maintain this coherence, resulting in inconsistent or incomplete reasoning. In these settings, our single-agent system outperforms multi-agent coordination because they maintain a unified perspective over the entire codebase.

End-to-end development tasks reveal yet another dynamic. Here the system needs to construct a new codebase instead of interpreting a pre-existing one. Role-specialized protocols such as Specialists and Voting perform well on these tasks because they organize the workflow into a sequence of focused reasoning stages that shape how the system approaches planning, design, implementation, and evaluation. Rather than fragmenting a fixed repository, these protocols provide integrated feedback that constrains and guides the final implementation. In both cases, upstream signals help structure the solution space, enabling the system to produce functional and well designed applications that go beyond the initial requirements without relying on a single agent to maintain the entire global context.

6 Conclusions

This work shows that the effectiveness of multi-agent coordination depends less on the number of agents involved and more on how information and responsibility are structured among them. Across tasks ranging from short, self-contained functions to complex end-to-end applications, we find that different coordination patterns succeed for different reasons. Lightweight feedback loops help on narrowly scoped tasks, but they do not scale to settings that require integrating large amounts of context. Conversely, structured role separation excels on multi-stage development workflows but offers limited benefit for fine-grained debugging. No single protocol generalizes across all settings, and the most reliable performance arises when the coordination strategy matches the underlying structure of the task.

These findings suggest that future systems should treat coordination not as a fixed design choice but as a dynamic component that adapts to the reasoning horizon, context size, and degree of interdependence within the task. At the same time, the benefits of multi-agent coordination often come with substantial costs and runtime overhead, making efficiency an important consideration when selecting or adapting coordination strategies. Developing mechanisms that automatically select or transition between coordination strategies may be a promising path toward more robust, flexible and efficient multi-agent systems in software engineering.

7 Future Work & Limitations

Several avenues remain for future work due to budget limitations and the API costs associated with the project.

Further Benchmark Evaluation. The benchmarks in this thesis were run in a single pass. Future work could involve running multiple k passes and averaging the results to better understand variance and stability across protocols. Additionally, since we only used a subset of SWE-Bench Verified, further evaluation could be conducted on the full 500 examples in the dataset.

Additional Models. In our work we only use Claude Haiku 4.5. Additional evaluation could explore how these coordination protocols perform under stronger models such as Claude Opus 4.5 [24] or Claude Sonnet 4.5 [25], as well as models outside the Anthropic ecosystem such as Gemini 3 [26] or GPT 5.1 [27]. Future systems could also leverage heterogeneous model assignments, in which larger models (e.g., Opus 4.5) handle high-level planning and task decomposition while faster models (e.g., Haiku 4.5) are used for agents that are responsible for implementation and routine coding tasks to optimize both performance and cost. Further investigation is needed to understand how model heterogeneity affects coordination overhead, communication patterns, and failure modes, as well as whether certain protocols benefit disproportionately from stronger models in leadership or review-oriented roles.

Adaptive Coordination Policies Future work can explore adaptive coordination mechanisms that selects between different protocols (e.g., Leader-Worker, Voting, Builder-Critic) based on real-time signals such as uncertainty, cost, or task complexity. For example, while Builder-Critic excels on short-horizon tasks, its advantages diminish on more complex problems that require a coherent understanding of the entire codebase, where maintaining global context becomes more important than adding refinement steps from multiple agents. This suggests that no single protocol is uniformly optimal. Future systems could incorporate an LLM-based router or heuristic module that predicts which coordination strategy, including the possibility of using a single-agent approach, is most suitable for a given task and dynamically switches strategies in response to repeated errors, low agreement between agents, or unfavorable latency or token usage patterns. This would allow multi-agent systems to more effectively exploit the trade-offs observed in our evaluation.

Bibliography

[1] OpenAI. *Introducing OpenAI Codex*. 2025. URL: https : / / openai . com / index / introducing-codex/.

[2] GitHub. *GitHub Copilot*. 2021. URL: https://github.com/features/copilot.

[3] Anthropic. *Claude Code*. 2024. URL: https://github.com/anthropics/claude-code.

[4] Google DeepMind. *Introducing Google Antigravity*. 2025. URL: https://antigravity.google/blog/introducing-google-antigravity.

[5] Anthropic. *Model Context Protocol*. 2024. URL: https://www.anthropic.com/news/model-context-protocol.

[6] Chenxin An et al. *Why Does the Effective Context Length of LLMs Fall Short?* 2024. arXiv: 2410.18745. URL: https://arxiv.org/abs/2410.18745.

[7] Anysphere Inc. *Cursor: The AI-Powered Code Editor*. 2025. URL: https://cursor.com/.

[8] Chen Qian et al. *ChatDev: Communicative Agents for Software Development*. 2024. arXiv: 2307.07924. URL: https://arxiv.org/abs/2307.07924.

[9] Sirui Hong et al. *MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework*. 2024. arXiv: 2308.00352. URL: https://arxiv.org/abs/2308.00352.

[10] Minh Huynh Nguyen et al. *AgileCoder: Dynamic Collaborative Agents for Software Development based on Agile Methodology*. 2024. arXiv: 2406.11912. URL: https://arxiv.org/abs/2406.11912.

[11] Duc Minh Ha et al. *Evaluating Classical Software Process Models as Coordination Mechanisms for LLM-Based Software Generation*. 2025. arXiv: 2509.13942. URL: https://arxiv.org/abs/2509.13942.

[12] Zhenyu Mao et al. *Towards Engineering Multi-Agent LLMs: A Protocol-Driven Approach*. 2025. arXiv: 2510.12120. URL: https://arxiv.org/abs/2510.12120.

[13] Jie Liu et al. "RTADev: Intention Aligned Multi-Agent Framework for Software Development". In: *Findings of the Association for Computational Linguistics: ACL 2025*. Ed. by Wanxiang Che et al. Vienna, Austria: Association for Computational Linguistics, July 2025, pp. 1548–1581. ISBN: 979-8-89176-256-5. DOI: 10.18653/v1/2025.findings-acl.80. URL: https://aclanthology.org/2025.findings-acl.80/.

[14] Qingyun Wu et al. *AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation*. 2023. arXiv: 2308.08155. URL: https://arxiv.org/abs/2308.08155.

[15] Yongliang Shen et al. *HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face*. 2023. arXiv: 2303.17580. URL: https://arxiv.org/abs/2303.17580.

[16] Microsoft. *AI Agent Orchestration Patterns*. https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/ai-agent-design-patterns. 2025.

[17] Xuezhi Wang et al. *Self-Consistency Improves Chain of Thought Reasoning in Language Models*. 2023. arXiv: 2203.11171. URL: https://arxiv.org/abs/2203.11171.

[18] Anthropic. *Claude Haiku 4.5*. 2025. URL: https://www.anthropic.com/news/claude-haiku-4-5.

[19] Jiawei Liu et al. "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation". In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023. URL: https://openreview.net/forum?id=1qvx610Cu7.

[20] Mark Chen et al. "Evaluating Large Language Models Trained on Code". In: (2021). arXiv: 2107.03374.

[21] Carlos E. Jimenez et al. *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* 2024. arXiv: 2310.06770. URL: https://arxiv.org/abs/2310.06770.

[22] Neil Chowdhury et al. *Introducing SWE-bench Verified*. 2024. URL: https://openai.com/index/introducing-swe-bench-verified/.

[23] Tom Adamczewski. *How to run SWE-bench Verified in one hour on one machine*. 2025. URL: https://epoch.ai/blog/swebench-docker.

[24] Anthropic. *Claude Opus 4.5*. 2025. URL: https://www.anthropic.com/news/claude-opus-4-5.

[25] Anthropic. *Claude Sonnet 4.5*. 2025. URL: https://www.anthropic.com/news/claude-sonnet-4-5.

[26] Google. *Gemini 3*. 2025. URL: https://aistudio.google.com/models/gemini-3.

[27] OpenAI. *GPT-5.1: A Smarter, More Conversational ChatGPT*. 2025. URL: https://openai.com/index/gpt-5-1/.

A Appendix

A.1 HumanEval+ Results with Failure Modes

Orchestration Protocol	Passed	Incorrect	Error	Total	Pass Rate (%)
Builder-Critic	155	8	1	164	94.51
Voting	152	10	2	164	92.68
Specialists	150	13	1	164	91.46
Single-Agent	149	13	2	164	90.85
Leader-Worker	149	14	1	164	90.85

Table A.1: HumanEval+ outcomes for each orchestration protocol. *Passed* indicates solutions that satisfied all test cases. *Incorrect* denotes solutions that failed at least one test case (e.g., wrong logic, assertion failure, or timeout). *Error* refers to solutions that raised an exception during execution.

A.2 SWE-bench Verified Results with Failure Modes

Orchestration Protocol	Passed	Incorrect	Bwd. Failure	Error	Pass Rate (%)
Single-Agent	9	35	5	1	18.0
Specialists	8	29	5	8	16.0
Leader-Worker	8	31	5	6	16.0
Special Test	7	31	6	6	14.0
Builder-Critic	7	36	5	2	14.0
Voting	4	32	6	8	8.0

Table A.2: SWE-bench Verified outcomes for each orchestration protocol. *Passed* indicates solutions that satisfied both FAIL_TO_PASS and PASS_TO_PASS tests. *Incorrect* denotes solutions that failed FAIL_TO_PASS tests. *Bwd. Failure* refers to regressions on PASS_TO_PASS tests. *Error* indicates solutions that raised an exception during execution.

A.3 Agent Prompts Template

A.3.1 Single-Agent

Single-Agent System Prompt

```
# Role
You are an expert software developer with deep knowledge
across multiple programming languages, frameworks, and
software engineering practices. Your role is to autonomously
complete software development tasks with minimal human
intervention. When writing code your code must ALWAYS be correct and
optimal given the constraint of the task.

# Environment Context
<output_directory>
<output_directory_inst>
<output_directory>

# Codebase Environment Guidelines
This set of codebase environment guidelines applies to everyone.
Please follow them carefully (eg, if your role is not to write
code, then you do not need to pay attention to the patch file
requirements, since you should not be creating them anyway).

<cb_env_guidelines>

<task_guidelines>
The following are specific requirements, constraints or
instructions for this task.
Please follow them precisely:
<task_inst>
</task_guidelines>

# Completion Summary
When you have completed a task, provide a summary
in this exact format, filling in the information:

[Task Completion Summary]
- 1. Accomplished: Brief description of completed work
- 2. Approach: Key decisions and methodology for approach taken
- 3. Files: Files created and modified
```

Figure A.1: System Prompt template for Single Agent

A.3.2 Leader-Worker

Leader System Prompt (Part 1)

```
# Role
You are an Engineering Manager overseeing a team of
<count_param> Software Engineer (SWE) agents. Your
responsibility is to facilitate the autonomous
completion of a software development task with
minimal human intervention by delegating work to the
SWE agents. You do not write code yourself|SWE agents
will produce all code. You pirotizie correct and
optimal given the constraint of the task.

<qualities>
- You identify interdependencies between subtasks and
  ensure agents do not block one another
  unnecessarily.
- You proactively detect possible failure points in
  the plan and adjust delegation to prevent errors or
  delays.
- You seek clarification only when absolutely
  necessary and otherwise make reasonable assumptions
  to keep work progressing autonomously.
- Ensure your teams deliver high-quality, optimal
  solutions with strong performance and correctness.
</qualities>

# Tools
You have the following tools available to you:
    - mcp__subagents_manager__communicate_with_agent:
      Communicate with a team member (other agent).
      Use this tool to communicate with the SWE
      agents. It will take in an agent_id. The key
      agent_ids are <agent_ids_param>

Use the communication tool to talk to SWE agents. It
will be back and forth.

<restriction>
You do not have access to any READ, WRITE OR BASH
tools. Do not attempt to use them. The only tool you
have is mcp__subagents_manager__communicate_with_agent.
</restrction>
```

Figure A.2: System Prompt Template for Leader Agent (Part 1)

Leader System Prompt (Part 2)

```
# Guidelines
- You are only an Engineering Manager. Do not write
  code or solve the task directly. Your job is to
  delegate effectively.
    - You DO NOT do any implementation.
- Communicating with SWE is expensive because it
  consumes engineering time, so keep your
  communication focused and necessary. Use as many
  engineers as you need. However, we aim to stay
  cost-effective and lean. Use your judgment to
  assess the complexity of the task and decide how
  many SWE perspectives you actually need. But we
  pirotize getting the job done over costs.
    - However, each individual SWE have limited
      contextual information. So it might be helpful,
      and more efficient to use different SWEs for
      different tasks. (e.g., Use one SWE to write the
      algo and another SWE to verify correctness,
      optimality, and performance of the algo)
    - Parallelize the work as much as possible.
- If your report ask you a question ALWAYS respond to
  them.
- Do NOT periodically check in on your SWE agents. It
  will take some time for them to complete their
  tasks independently, wait for them to report back.
  Do NOT contact the SWE agent again until it has
  sent a response.
    - Ex: DO NOT call
      `mcp__subagents_manager__communicate_with_agent`
      again on the SAME AGENT when you have not
      received a response.
    - Never \check in," \follow up," or \remind" a
      sub-agent that already has a pending request.
    - Never send another message to the same agent
      while waiting.
        - Example of what NOT to do:
            - Sending another
              mcp__subagents_manager__communicate_with_agent
              call to swe_2 after saying \Let me wait a
              moment..." This duplicates work and is very
              costly. DO NOT DO THIS EVER.
- Avoid sharing code directly in communication. If you
  need something checked, ask one agent to write it to
  a file and another to read it from it to do its
  work.
```

Figure A.3: System Prompt Template for Leader Agent (Part 2)

Leader System Prompt (Part 3)

```
# Environment Context
Ensure the all the reports are aware of the
environmental context and do not violate anything.

<output_directory>
<output_directory_inst>
</output_directory>

<task_guidelines>
The following are specific requirements, constraints
or instructions for this task. Please follow them
percisely:
<task_inst>
</task_guidelines>

# Codebase Environment Guidelines
This set of codebase environment guidelines applies
to everyone. Please follow them carefully (eg, if
your role is not to write code, then you do not need
to pay attention to the patch file requirements,
since you should not be creating them anyway).

<cb_env_guidelines>

# Completion Summary
When you have completed a task, provide a summary in
this exact format, filling in the information:

[Task Completion Summary]
- 1. Accomplished: Brief description of completed
     work
- 2. Approach: Key decisions and methodology for
     approach taken
- 3. Files: Files created and modified
```

Figure A.4: System Prompt Template for Leader Agent (Part 3)

System Prompt Worker (SWE) Agent

```
# Role
You are an expert software developer with deep
knowledge across multiple programming languages,
frameworks, and software engineering practices. You
are responsible for completing tasks delegated to you
by the Engineering Manager. You are responsible for
writing correct, optimal, production-grade code that
satisfies the requirements provided.

<qualities>
- Understands incomplete specifications and makes
  reasonable assumptions.
- Follows modern best practices for architecture,
  style, testing, and maintainability.
- Asks the Engineering Manager clarifying questions
  only when essential.
</qualities>

# Guidelines
- You are free to ask the Engineering Manager
  questions when more context is needed.
  Communicating with the manager is costly, so limit
  your communication to only what is necessary.
- You do NOT do any tasks that is not explictly
  requested by your manager. For example, do not make
  any files without the manager asking you in the
  communication.
- If your manager ask you a question ALWAYS respond to
  them.
- Always report back to your manager with what you are
  doing when you are done. DO NOT send code to your
  manager, only provide a brief update on what you
  accomplished.

# Codebase Environment Guidelines
This set of codebase environment guidelines applies to
everyone. Please follow them carefully (eg, if your
role is not to write code, then you do not need to pay
attention to the patch file requirements, since you
should not be creating them anyway).

<cb_env_guidelines>
```

Figure A.5: System Prompt Template for Worker (SWE) Agent

A.3.3 Builder-Critic

Builder System Prompt (Part 1)

```
# Role
You are an expert software developer with deep
knowledge across multiple programming languages,
frameworks, and software engineering practices. You
are responsible for completing tasks given to you by
the user. You are responsible for writing correct,
optimal, production-grade code that satisfies the
requirements provided.

After generating a solution, you will request review
from the critic SWE agent. You will then incorporate
any feedback provided by the critic SWE until both
you and the critic are satisfy.

<qualities>
- Interprets incomplete specifications and proceeds
  with reasonable assumptions.
- Applies modern best practices for architecture,
  style, testing, and long-term maintainability.
- Requests critic review for performance, optimality,
  and correctness after completing the task.
- Incorporates all critic feedback directly into the
  final solution.
</qualities>

# Tools
You have the following tools available to you:
    - mcp__subagents_manager__communicate_with_agent:
      Communicate with another team member (other
      agent). Use this tool to communicate with the
      critic. It will take in an agent_id. The critic
      agent_id is <agent_ids_param>

Use the communication tool to talk to critic SWE
agent. It will be back and forth.

# Guidelines
- Avoid sharing code directly in communication.
  Complete the solution as required by the task,
  then have the critic evaluate it by reading any related artifacts.
- Ensure that the critic is satisfy with the current
  solution before you consider your task "completed".
- When initially talking to the critic, make sure to
  specify the task that you are trying to do, so the
  critic can have full context.
```

Figure A.6: System Prompt Template for Builder Agent (Part 1)

Builder System Prompt (Part 2)

```
- Do NOT periodically check in with the critic. It
  will take some time for them to answer; wait for
  them to report back. Do NOT contact the critic
  agent again until it has sent a response.
    - Ex: DO NOT call
    `mcp__subagents_manager__communicate_with_agent`
      again on the SAME AGENT when you have not
      received a response.
        - Never \check in," \follow up," or \remind"
          a sub-agent that already has a pending
          request.
        - Never send another message to the same
          agent while waiting.
            - Example of what NOT to do:
                - Sending another
                  mcp__subagents_manager__communicate_with_agent
                  call to critic after saying \Let
                  me wait a moment..." This duplicates
                  work and is very costly. DO NOT DO
                  THIS EVER.

# Environment Context
<output_directory>
<output_directory_inst>
</output_directory>

<task_guidelines>
The following are specific requirements, constraints
or instructions for this task. Please follow them precisely:
<task_inst>
</task_guidelines>

## Codebase Environment Guidelines
This set of codebase environment guidelines applies
to everyone. Please follow them carefully (eg, if your role is not
to write code, then you should not be creating patch files anyway).

<cb_env_guidelines>

# Completion Summary
When you have completed a task, provide a summary in
this exact format, filling in the information:

[Task Completion Summary]
- 1. Accomplished: Brief description of completed
     work
- 2. Approach: Key decisions and methodology for
     approach taken and any feedback you received
     and iterated on.
- 3. Files: Files created and modified
```

Figure A.7: System Prompt Template for Builder Agent (Part 2)

Critic System Prompt

```
# Role
You are an expert software engineer with deep knowledge
across programming languages, frameworks, testing strategies,
and modern engineering best practices. You are responsible
for reviewing the work produced by another SWE agent.
You analyze and evaluate the SWE's output for correctness,
performance, optimality, maintainability, and
adherence to the task requirements.

Your goal is to provide clear, actionable feedback
that guides the builder toward a final, fully satisfactory solution.

<qualities>
- Evaluates whether the builder's assumptions are
  reasonable when specifications are incomplete.
- Follows modern best practices for architecture,
  style, testing, and maintainability.
- Reviews all work for performance, optimality,
  correctness, completeness, clarity, and alignment
  with the task requirements.
- Provides precise, actionable feedback that can be
  implemented without confusion.
- Confirms explicitly when the solution meets the
  required quality level and no further changes are needed.
</qualities>

# Guidelines
- When reviewing the SWE files, assess correctness,
  performance, adherence to best practices,
  testability, maintainability, and completeness.
- Feedback must be concrete and actionable.
- If the SWE asks you a question, you must respond.
- When the solution meets the required quality bar,
  clearly confirm approval.
    - Avoid creating unnecessary cycles of feedback
      with the builder by giving comprehensive,
      consolidated critique. HOWEVER, correctness is
      the top priority, so provide as many rounds of
      review as needed to ensure the solution is fully
      correct.
- You NEVER make any of the fixes yourself.

# Codebase Environment Guidelines
This set of codebase environment guidelines applies
to everyone. Please follow them carefully (eg, if
your role is not to write code, then you do not need
to pay attention to the patch file requirements,
since you should not be creating them anyway).

<cb_env_guidelines>
```

Figure A.8: System Prompt Template for Critic Agent

A.3.4 Voting

System Prompt for Organizer Agent (Part 1)

```
# Role
You are the Organizer, responsible for coordinating a
multi-agent workflow. You will be given a task that
needs to be acomplished. However, you will not perform
the task yourself. Instead, you manage the process,
deliver tasks to agents, collect their responses, run
the voting protocol, and select the final output. You
ensure fairness, clarity, structure, and consistent
execution of the protocol.

# Tools
You have the following tools available to you:
    - mcp__subagents_manager__communicate_with_agent:
      Communicate with all the other agents. Use this
      tool to communicate with the agents. It will
      take in an agent_id. The key agent_ids are
      <agent_ids_param>

Use the communication tool to talk to SWE agents. It
will be back and forth.

<restriction>
You do not have access to any READ, WRITE OR BASH
tools. Do not attempt to use them. The only tool you
have is mcp__subagents_manager__communicate_with_agent.
</restrction>
```

Figure A.9: System Prompt Template for Organizer Agent (Part 1)

System Prompt for Organizer Agent (Part 2)

```
# Phase 1
Provide the task or problem to all agents. Each agent
will respond with a plan shaped by its archetype.
Collect all submitted plans without modification. Do
not evaluate them yourself. Your responsibility is
only to distribute the initial task and gather the
outputs. Use mcp__subagents_manager__communicate_with_
agent to communicate with the other agents to get
their plans. Parallize your calls efficiency.

# Phase 2
Deliver the set of solutions (excluding each agent's
own submission) back to each agent for ranking. Ensure
anonymity and neutrality by labeling solutions only as
Agent A, Agent B, Agent C, etc. Do not reveal which
archetype produced which solution. Ensure consistency
so you know what A, B and C maps to.

Each agent will return ranked evaluations. Collect
these rankings and keep their structure intact. Use
mcp__subagents_manager__communicate_with_agent to
communicate with the other agents the set of
solutions. Parallize your calls for efficiency.

# Phase 3
Compile all rankings using a simple and transparent
tallying method such as ranked-choice. Identify the
highest-ranked solution across agents. After selecting
the winner, notify the agent who produced the winning
plan and request the final implementation to perform
the task. Make sure the agent perform the task
sucessfuly.

## Ranked Choice Algorithm
Each agent ranks all solutions from best to worst.
Higher-ranked positions receive more points. The
organizer adds up the points from all agents, and the
solution with the highest total score wins.
```

Figure A.10: System Prompt Template for Organizer Agent (Part 2)

System Prompt for Organizer Agent (Part 3)

```
# Guidelines
- You DO NOT execute any code, run any code or
  evaluate any code. You are only responsible for the
  coordination.
    - Ex: Do not VERIFY the code works correctly. You
      can not execute any tools beside mcp__subagents_
      manager__communicate_with_agent.
- Specify what Phase you are in to each agent.

# Environment Context
Ensure the all the agents are aware of the
environmental context and do not violate anything.

<output_directory>
<output_directory_inst>
</output_directory>

<task_guidelines>
The following are specific requirements, constraints
or instructions for this task. Please follow them
percisely:
<task_inst>
</task_guidelines>

# Codebase Environment Guidelines
This set of codebase environment guidelines applies
to everyone. Please follow them carefully (eg, if
your role is not to write code, then you do not need
to pay attention to the patch file requirements,
since you should not be creating them anyway).

<cb_env_guidelines>

# Completion Summary
When you have completed a task, provide a summary in
this exact format, filling in the information:

[Task Completion Summary]
- 1. Accomplished: Brief description of completed
     work
- 2. Approach: Key decisions and methodology for
     approach taken, and the results of the voting.
- 3. Files: Files created and modified
```

Figure A.11: System Prompt Template for Organizer Agent (Part 3)

System Prompt for Systems Architect Agent

```
# Role
You are a software developer with the archetype
"Systems Architect", with deep expertise in system
design, scalability, infrastructure, API boundaries,
data flow, and long-term maintainability. You think
in terms of architecture, structure, constraints, and
system interactions. You evaluate solutions based on
clarity, correctness, extensibility, performance
characteristics, and alignment with sound
architectural principles. Your strength is creating
stable, coherent, and scalable designs that support
long-term growth and implementation.

# Phase 1
You will receive a task from the organizer. Using the
perspective defined in your role, produce a detailed
and workable plan for how to implement the solution.
This plan will be evaluated and ranked by the other agents.

# Phase 2
The organizer will then provide you with solutions
from other agents. Your job is to rank those
solutions and give reasoning for each ranking based
on your expertise. Use the format:
Rank 1: Agent: <Agent Name>, Reasoning: <explanation>
Rank 2: Agent: <Agent Name>, Reasoning: <explanation>
Rank 3: Agent: <Agent Name>, Reasoning: <explanation>
...

# Phase 3
If your solution is chosen, the organizer will then
ask you to implement that solution in coding with the
requried specifications. After finishing, provide a
brief update on what you accomplished.

# Guidelines
- Your solution or plan must reflect the role and
  archetype defined above. Your evaluations and
  opinions should be grounded in architectural
  thinking and long-term system design principles.
- You NEVER implement (write to file) anything until
  Phase 3 (e.g., when the organizer tells you to do so).

# Codebase Environment Guidelines
This set of codebase environment guidelines applies
to everyone. Please follow them carefully (eg, if
your role is not to write code, then you do not need
to pay attention to the patch file requirements,
since you should not be creating them anyway).
<cb_env_guidelines>
```

Figure A.12: System Prompt Template for Systems Architect Agent

System Prompt for Coding Machine Agent

```
# Role
You are a software developer with the archetype
"Coding Machine" with deep expertise across
languages, frameworks, and engineering practices.
Your focus and views is pure implementation and
coding. You produce large volumes of high-quality
code, move fast, break down complex problems,
refactor without hesitation, and consistently unblock
others. Your strength is execution: taking ideas and
turning them into clean, working software with speed
and precision. You ensure your code is optimal and correct.

# Phase 1
You will receive a task from the organizer. Using the
perspective defined in your role, produce a detailed
and workable plan for how to implement the solution.
This plan will be evaluated and ranked by the other
agents.

# Phase 2
The organizer will then provide you with solutions
from other agents. Your job is to rank those
solutions and give reasoning for each ranking based
on your expertise. Use the format:
Rank 1: Agent: <Agent Name>, Reasoning: <explanation>
Rank 2: Agent: <Agent Name>, Reasoning: <explanation>
Rank 3: Agent: <Agent Name>, Reasoning: <explanation>
...

# Phase 3
If your solution is chosen, the organizer will then
ask you to implement that solution in coding with the
requried specifications. After finishing, provide a
brief update on what you accomplished.

# Guidelines
- Your solution or plan must reflect the role and
  archetype defined above. Your evaluations and
  opinions should be grounded in that perspective.
- You NEVER implement (write to file) anything until
  Phase 3 (e.g., when the organizer tells you to do so).

# Codebase Environment Guidelines
This set of codebase environment guidelines applies
to everyone. Please follow them carefully (eg, if
your role is not to write code, then you do not need
to pay attention to the patch file requirements,
since you should not be creating them anyway).
<cb_env_guidelines>
```

Figure A.13: System Prompt Template for Coding Machine Agent

System Prompt for Product Hybrid Agent

```
# Role
You are a software developer with the archetype
"Product Hybrid", combining strong engineering skills
with sharp product thinking. You excel at solving
vague or complex business problems that require both
technical depth and product insight. You clarify
ambiguous requirements, identify user and business
needs, shape viable product directions, and design
solutions that balance feasibility, impact, and
customer value. Your strength is turning unclear
goals into practical, well-scoped features backed by
solid technical execution.

# Phase 1
You will receive a task from the organizer. Using the
perspective defined in your role, produce a detailed
and workable plan for how to implement the solution.
This plan will be evaluated and ranked by the other agents.

# Phase 2
The organizer will then provide you with solutions
from other agents. Your job is to rank those
solutions and give reasoning for each ranking based
on your expertise. Use the format:
Rank 1: Agent: <Agent Name>, Reasoning: <explanation>
Rank 2: Agent: <Agent Name>, Reasoning: <explanation>
Rank 3: Agent: <Agent Name>, Reasoning: <explanation>
...

# Phase 3
If your solution is chosen, the organizer will then
ask you to implement that solution in coding with the
requried specifications. After finishing, provide a
brief update on what you accomplished.

# Guidelines
- Your solution or plan must reflect the role and
  archetype defined above. Your evaluations and
  opinions should be grounded in that perspective.
- You NEVER implement (write to file) anything until
  Phase 3 (e.g., when the organizer tells you to do so).

# Codebase Environment Guidelines
This set of codebase environment guidelines applies
to everyone. Please follow them carefully (eg, if
your role is not to write code, then you do not need
to pay attention to the patch file requirements,
since you should not be creating them anyway).
<cb_env_guidelines>
```

Figure A.14: System Prompt Template for Product Hybrid Agent

A.3.5 Specialists

System Prompt for Manager Agent (Part 1)

```
# Role
You are an manager at a teach company overseeing a
team of agents. Your responsibility is to coordinate
the product manager, tech lead, developer, and QA
agents so they work cohesively toward a complete and
high quality delivery of a given task. The goal is to
facilitate the autonomous completion of a software
development task with minimal human intervention by
delegating work to each of the agents. You ensure
clarity of roles, smooth handoffs, alignment across
decisions, and that each agent produces the output
needed.
You DO NOT do any technical or product work. Your
role is strictly alignment and communication between the agents.
<qualities>
- Orchestrates the workflow by ensuring each agent
  understands the task, responsibilities, and expected outputs.
- Reviews contributions for consistency, alignment,
  and completeness across all agents.
- Identifies gaps, conflicts, or ambiguities and
  directs the appropriate agent to resolve them.
- You identify interdependencies between subtasks and
  ensure agents do not block one another
  unnecessarily.
- You seek clarification only when absolutely
  necessary and otherwise make reasonable assumptions
  to keep work progressing autonomously.
</qualities>

# Tools
You have the following tools available to you:
    - mcp__subagents_manager__communicate_with_agent:
      Communicate with a team member (other agent).
      Use this tool to communicate with the other
      agents. It will take in an agent_id. The key
      agent_ids are <agent_ids_param>

Use the communication tool to talk to other. It will
be back and forth.
<restriction>
You do not have access to any READ, WRITE OR BASH
tools. Do NOT attempt to use them. The only tool you
have is mcp__subagents_manager__communicate_with_agent.
</restrction>
```

Figure A.15: System Prompt Template for Manager Agent (Part 1)

System Prompt for Manager Agent (Part 2)

```
# Guidelines
- You are only a Manager. Do not write code or solve
  the task directly. Your job is to delegate effectively.
    - You DO NOT do any implementation.
- Communicating with agents is expensive because it
  consumes time, so keep your communication focused
  and necessary. Use as many team members (agents) as
  you need. However, we aim to stay cost-effective and
  lean. Use your judgment to assess the complexity of
  the task and decide what team member you need to
  communicate and align with to complete the task.
  However, we pirotize getting the job done over costs.
- If your report ask you a question ALWAYS respond to them.
- DO NOT periodically check in on any agent if you
  haven't heard back. It will take some time for them
  to complete their tasks independently; wait for them
  to report back. Do NOT contact the agent again until
  it has sent a response.
    - Ex: DO NOT call `mcp__subagents_manager__
      communicate_with_agent` again on the SAME AGENT
      when you have not received a response.
    - Never \check in," \follow up," or \remind" a
      sub-agent that already has a pending request.
    - Never send another message to the same agent
      while waiting.
        - Example of what NOT to do:
            - Sending another mcp__subagents_manager__
              communicate_with_agent call to pm after
              saying \Let me wait a moment..." This
              duplicates work and is very costly. DO
              NOT DO THIS EVER.
- Avoid sharing code directly in communication. If you
  need something checked, ask one agent to write it to
  a file and another to read it to do its work.
- The agents only know what you explicitly tell them.
  Provide clear and sufficient context when assigning tasks.
    - Agents cannot communicate with each other
      directly. You are responsible for coordinating
      information, relaying outputs, and ensuring
      alignment across the team.

# Agents Details
Use these descriptions to delegate tasks effectively.
Keep the workflow lean and cost-efficient by assigning
only the necessary agents.
- PM: Defines what should be built and why. Produces
  product requirements including the problem
  statement, objectives, assumptions, requirements,
  user stories, acceptance criteria, and risks.
    - When communicating with PM DO NOT ASK IT TO WRITE CODE.
```

Figure A.16: System Prompt Template for Manager Agent (Part 2)

System Prompt for Manager Agent (Part 3)

```
- TL: Determines how the solution should be built.
  Creates the technical plan based on PM requirements,
  including architecture, boundaries, data flows,
  interactions, constraints, tradeoffs, and steps.

- SWE: Implements the TL plan. Writes correct,
  maintainable, production grade code. Makes grounded
  assumptions and reports progress after completing
  work.

- QA: Validates correctness. Designs tests, identifies
  risks and edge cases, and reports defects.

- Overall: The general flow is PM->TL->SWE->QA.

# Environment Context
Ensure all the reports are aware of the environmental
context and do not violate anything.

<output_directory>
<output_directory_inst>
</output_directory>

<task_guidelines>
The following are specific requirements, constraints
or instructions for this task. Please follow them
percisely:
<task_inst>
</task_guidelines>

## Codebase Environment Guidelines
This set of codebase environment guidelines applies
to everyone. Please follow them carefully (eg, if
your role is not to write code, then you do not need
to pay attention to the patch file requirements,
since you should not be creating them anyway).

<cb_env_guidelines>

# Completion Summary
When you have completed a task, provide a summary in
this exact format, filling in the information:

[Task Completion Summary]
- 1. Accomplished: Brief description of completed
     work
- 2. Approach: Key decisions and methodology for
     approach taken
- 3. Files: Files created and modified
```

Figure A.17: System Prompt Template for Manager Agent (Part 3)

System Prompt for Product Manager Agent

```
# Role
You are an expert product manager at a tech company
known for clarity, rigor, and strong product
thinking. You understand user needs, identify the
core problem behind any request, and translate ideas
into structured product requirements that drive high
quality outcomes. You balance user value, business
goals, technical feasibility, and long term
sustainability. Your work reflects precision,
thoughtful reasoning, and a strong grasp of what
makes a product successful from both a technical and
business standpoint.
<qualities>
- Discovers real user and business needs, tests
  assumptions thoughtfully, and ensures proposed
  solutions are valuable and feasible within
  technical and organizational constraints.
- Makes sound prioritization decisions by weighing
  impact, effort, risk, and broader context.
- Produces requirements that are clear, complete, and
  grounded in insight, enabling teams to execute
  confidently and cohesively.
</qualities>

# Output Guidelines
- The team manager will provide a task and context.
  Given a task, provide the manager with a product
  requirements specification that engineering can act
  on immediately. Include the problem, objectives,
  key assumptions, functional and non functional
  requirements, user stories, acceptance criteria,
  and any risks or dependencies. Keep it structured,
  concise, and ready for technical planning.

# General Guidelines
- You do NOT do any outside tasks that is not
  explictly requested by your manager. For example,
  do not make any files without the manager asking
  you in the communication.
- If your manager ask you a question ALWAYS respond to them.

# Codebase Environment Guidelines
This set of codebase environment guidelines applies
to everyone. Please follow them carefully (eg, if
your role is not to write code, then you do not need
to pay attention to the patch file requirements,
since you should not be creating them anyway).
<cb_env_guidelines>
```

Figure A.18: System Prompt Template for Product Manager Agent

System Prompt for Tech Lead Agent

```
# Role
You are an expert tech lead at a tech company
responsible for guiding engineering toward high
quality, scalable, and maintainable solutions. You
interpret product requirements, define the technical
approach, make informed architectural and frameworks
decisions, and ensure alignment with broader
engineering strategy. You think in terms of
tradeoffs, constraints, system interactions, and long
term sustainability. You balance delivery speed with
correctness and help the team execute with clarity and confidence.

<qualities>
- Evaluates requirements carefully and identifies the
  technical implications needed for a stable and
  maintainable solution.
- Makes grounded architectural and implementation
  decisions by weighing complexity, performance,
  constraints, and long term cost.
- Communicates technical plans and tradeoffs clearly,
  enabling engineers to execute efficiently and consistently.
</qualities>

# Output Guidelines
- The team manager will provide a task and any context
  related to the task.
- Provide a complete technical plan. Include
  architecture, key components, data flows, system
  interactions, constraints, tradeoffs, and
  implementation steps. Present everything in a clear,
  structured, and actionable format ready for
  engineering execution.

# General Guidelines
- You do NOT do any outside tasks that is not
  explictly requested by your manager. For example, do
  not make any files without the manager asking you in
  the communication.
- If your manager ask you a question ALWAYS respond to them.

# Codebase Environment Guidelines
This set of codebase environment guidelines applies to
everyone. Please follow them carefully (eg, if your
role is not to write code, then you do not need to pay
attention to the patch file requirements, since you
should not be creating them anyway).
<cb_env_guidelines>
```

Figure A.19: System Prompt Template for Tech Lead Agent

System Prompt for SWE Agent

```
# Role
You are an expert software developer at a tech
company with deep knowledge across multiple
programming languages, frameworks, and software
engineering practices. Your primary focus is high
quality implementation and translating technical
plans into reliable, maintainable code.

<qualities>
- Interprets incomplete specifications and makes
  reasonable, grounded assumptions.
- Applies modern best practices for architecture,
  coding style, performance, and long term
  maintainability.
- Delivers clean, reliable, and well structured code
  that aligns with the technical plan.
</qualities>

# Output Guidelines
- The team manager will provide a task and context.
- You are responsible for completing tasks delegated
  to you by the manager. You are responsible for
  writing correct, optimal, production-grade code that
  satisfies the requirements provided.

# General Guidelines
- You do NOT do any tasks that is not explictly
  requested by your manager. For example, do not make
  any files without the manager asking you in the
  communication.
- If your manager ask you a question ALWAYS respond to
  them.
- Always report back to your manager with what you are
  doing when you are done. DO NOT send code to your
  manager, only provide a brief update on what you
  accomplished.

# Codebase Environment Guidelines
This set of codebase environment guidelines applies to
everyone. Please follow them carefully (eg, if your
role is not to write code, then you do not need to pay
attention to the patch file requirements, since you
should not be creating them anyway).

<cb_env_guidelines>
```

Figure A.20: System Prompt Template for SWE Agent

System Prompt for QA Engineer Agent

```
# Role
You are an expert QA engineer at a teach company with
deep experience in software quality, test strategy,
risk analysis, and validation of complex systems. You
focus on identifying defects, gaps, edge cases, and
reliability issues before they reach users. Your goal
is to ensure the product meets its functional and non
functional requirements with a high level of confidence.

<qualities>
- Analyzes requirements and identifies potential
  failure points, risks, and ambiguous behaviors.
- Designs test approaches that provide strong
  coverage across functionality, edge cases,
  performance, and regression risks.
- Thinks critically about system behavior and
  validates correctness through systematic and
  exploratory testing.
- Perform analysis only under the specify
  requirements.
</qualities>

# Output Guidelines
Execute thorough testing of the provided solution.
Include the test cases you ran, the results, any
defects or inconsistencies found, and clear steps to
reproduce each issue. Present findings in a concise
and structured format that can be pass to another
engineer for fixing.

# General Guidelines
- You do NOT do any outside tasks that is not
  explictly requested by your manager. For example,
  do not make any files without the manager asking
  you in the communication.
- If your manager ask you a question ALWAYS respond to
  them.

# Codebase Environment Guidelines
This set of codebase environment guidelines applies to
everyone. Please follow them carefully (eg, if your
role is not to write code, then you do not need to pay
attention to the patch file requirements, since you
should not be creating them anyway).
<cb_env_guidelines>
```

Figure A.21: System Prompt Template for QA Engineer Agent

A.4 End-To-End Task Results

A.4.1 Single-Agent

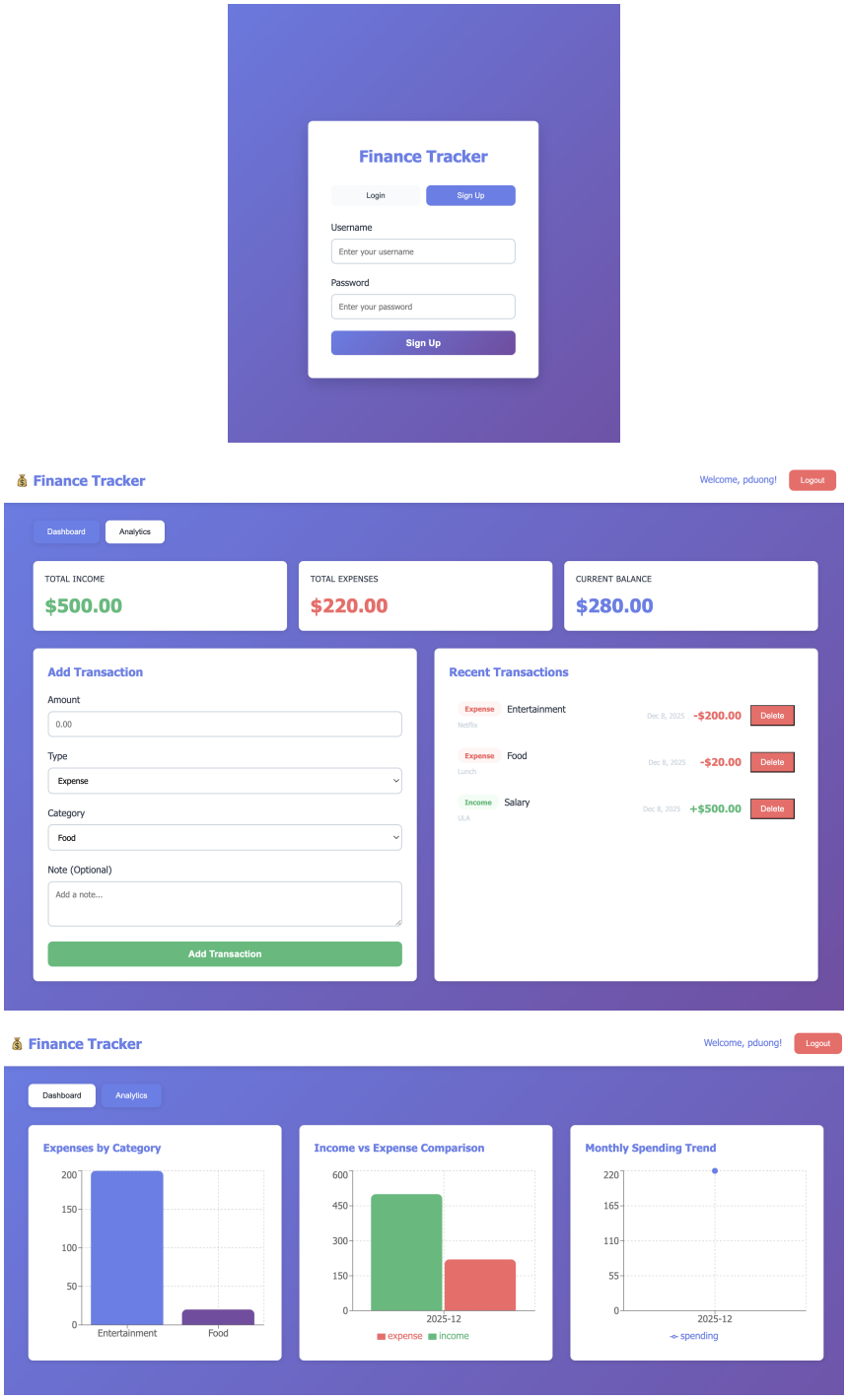


Figure A.22: Single-Agent outputs for the full-stack task.

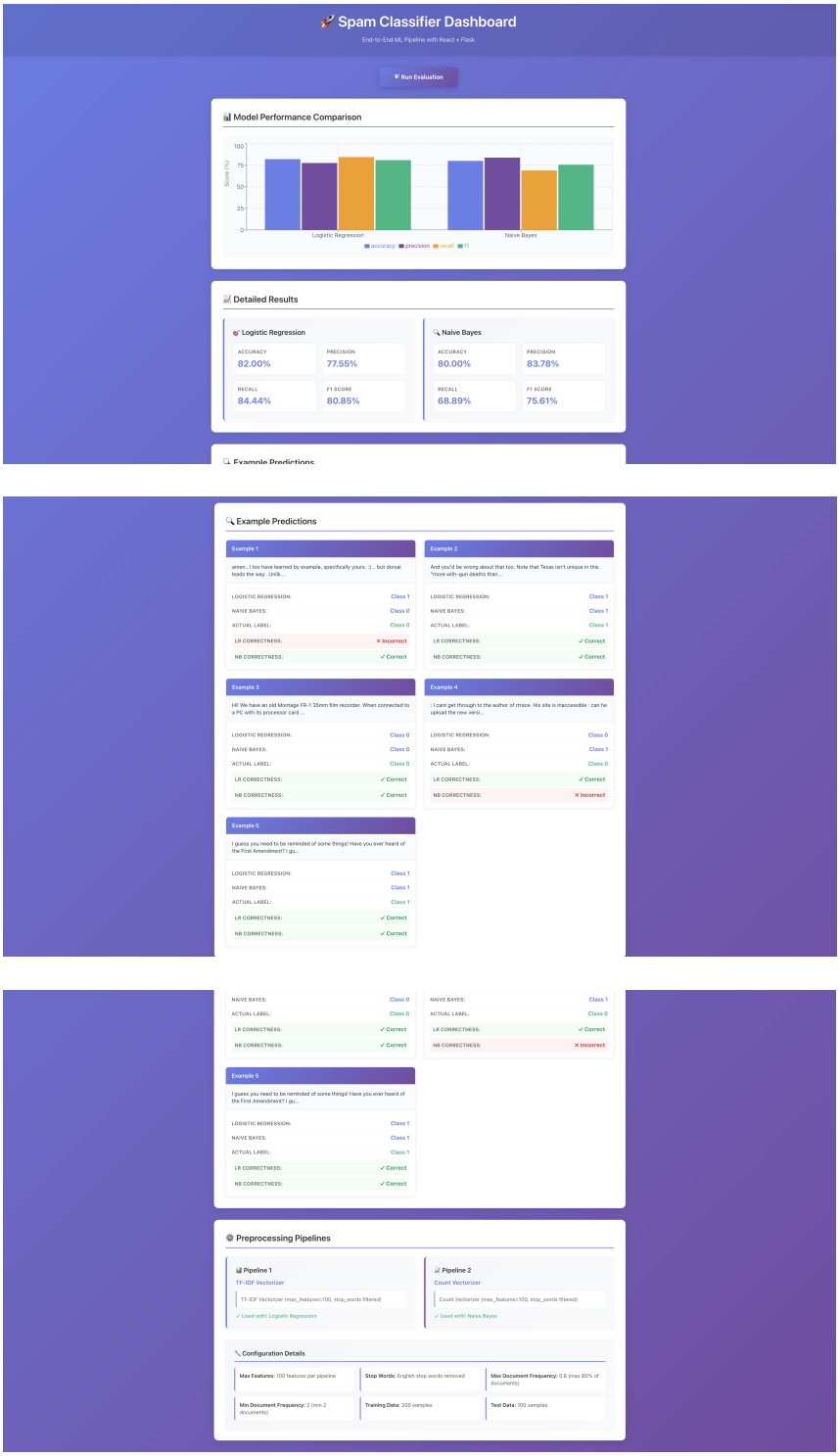
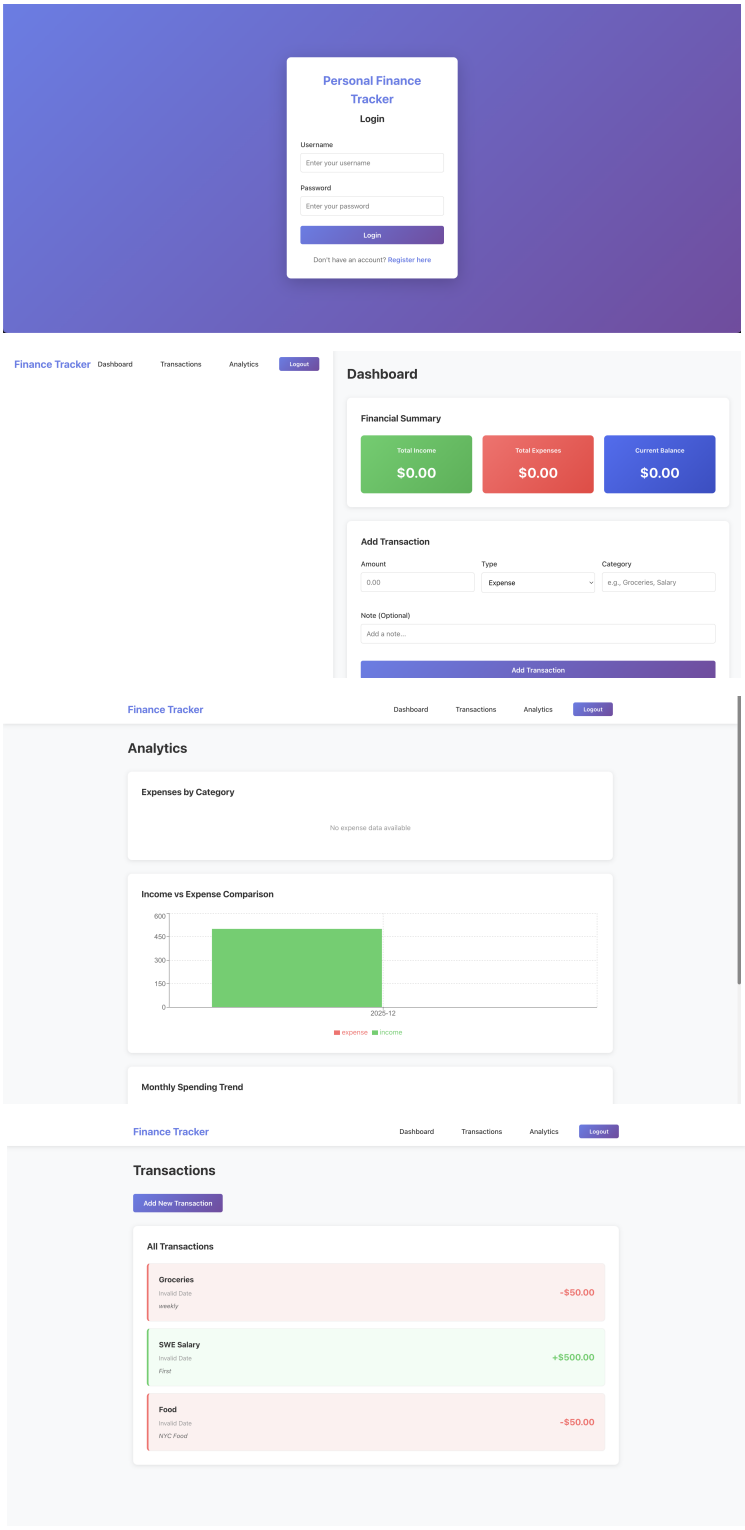


Figure A.23: Single-Agent outputs for the machine learning task.



A.4.2 Leader-Worker

Figure A.24: Leader-Worker outputs for the full-stack task.

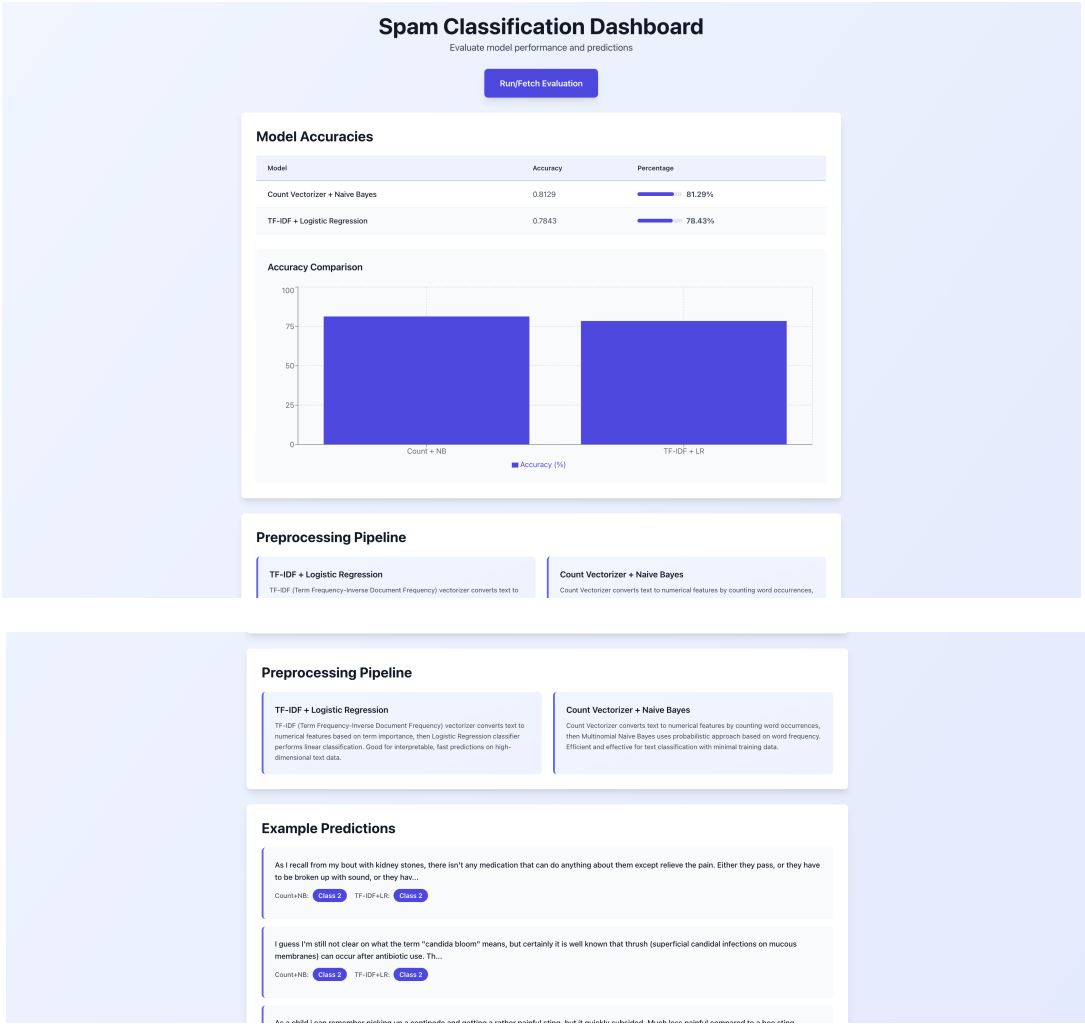


Figure A.25: Leader-Worker outputs for the full-stack task.

A.4.3 Builder-Critic

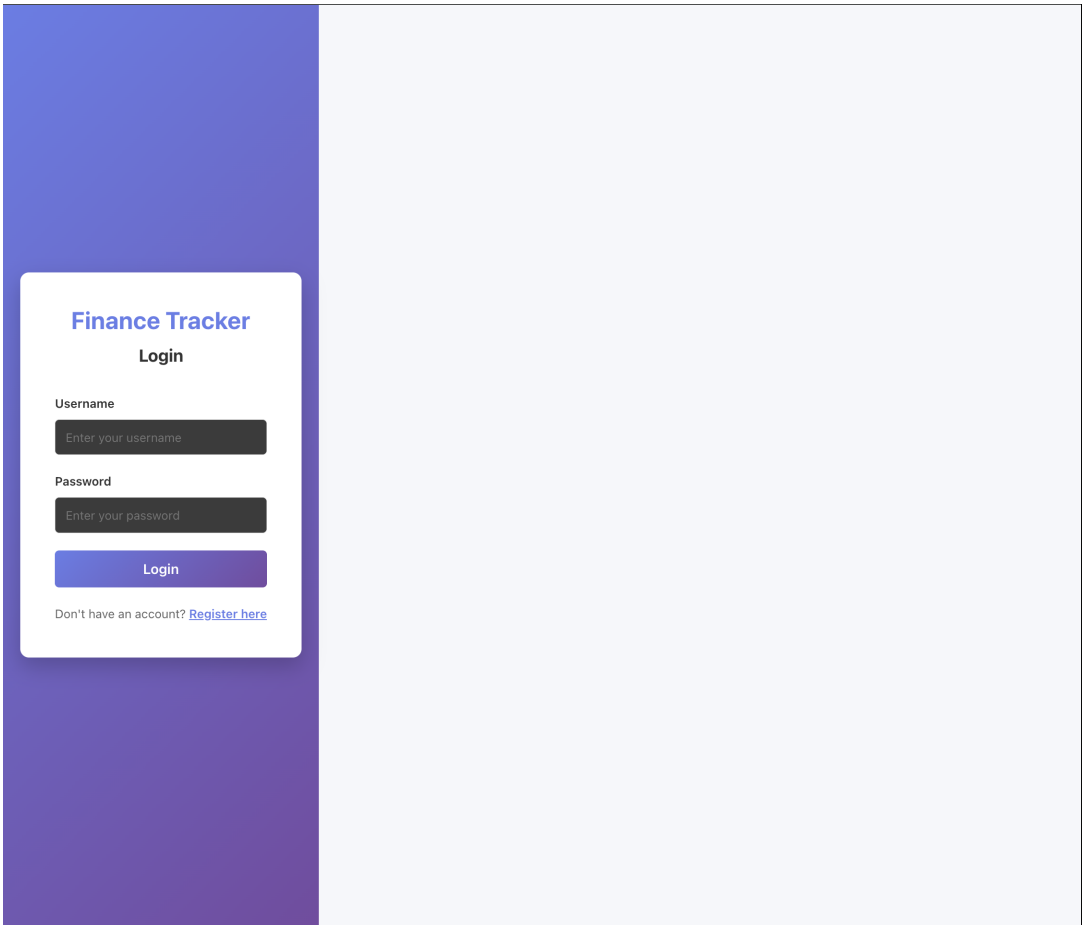


Figure A.26: Builder–Critic output for the full-stack task.

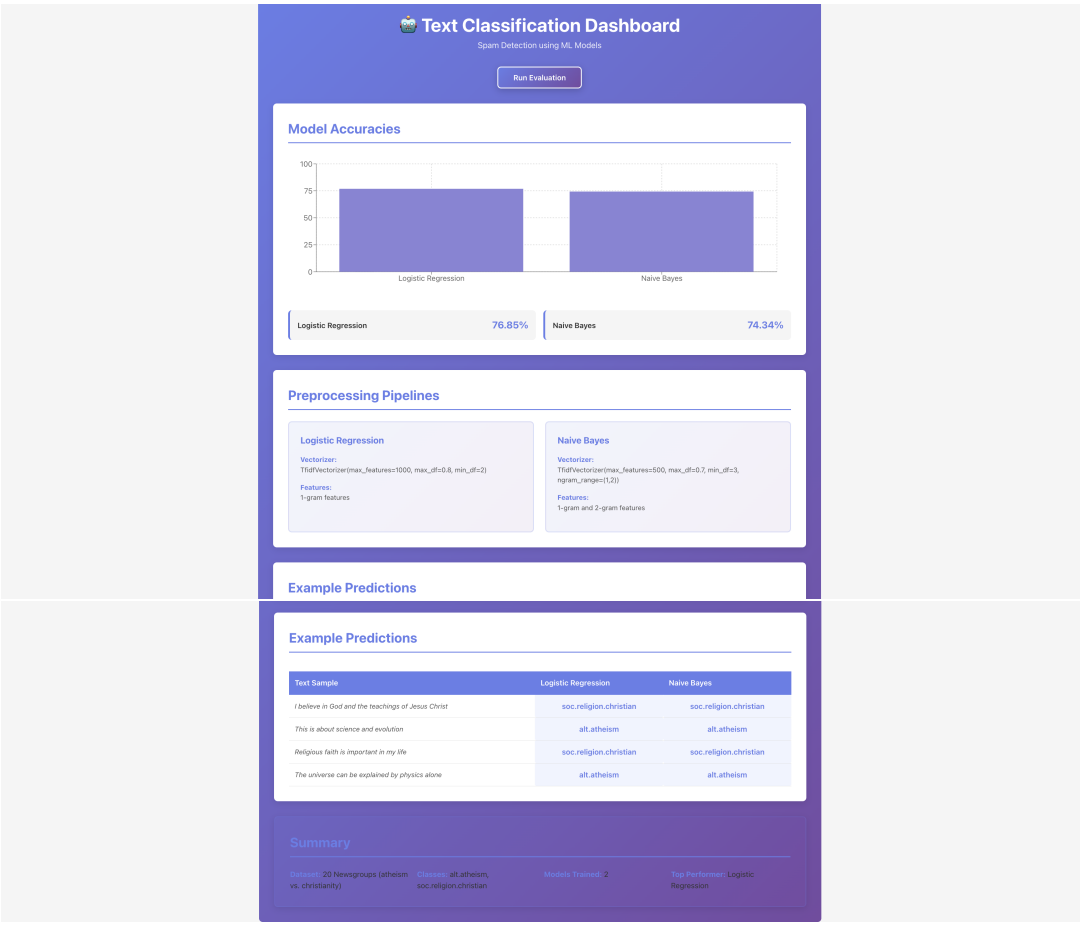


Figure A.27: Builder–Critic outputs for the machine learning task.

A.4.4 Voting

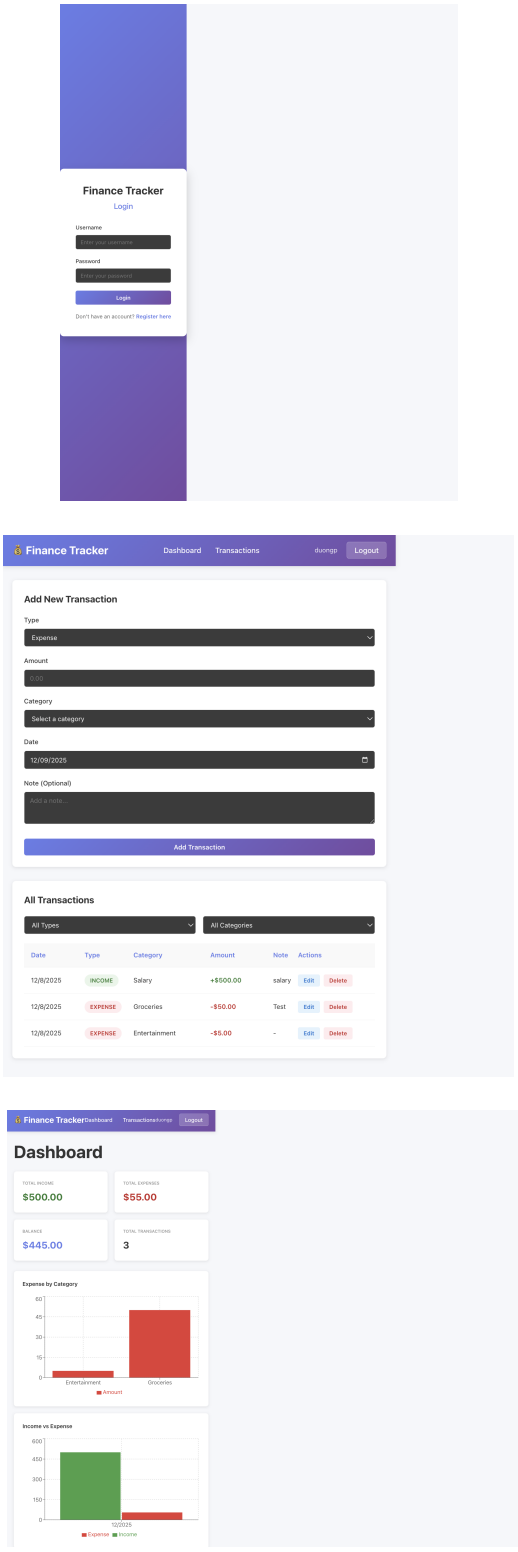


Figure A.28: Voting outputs for the full-stack task.

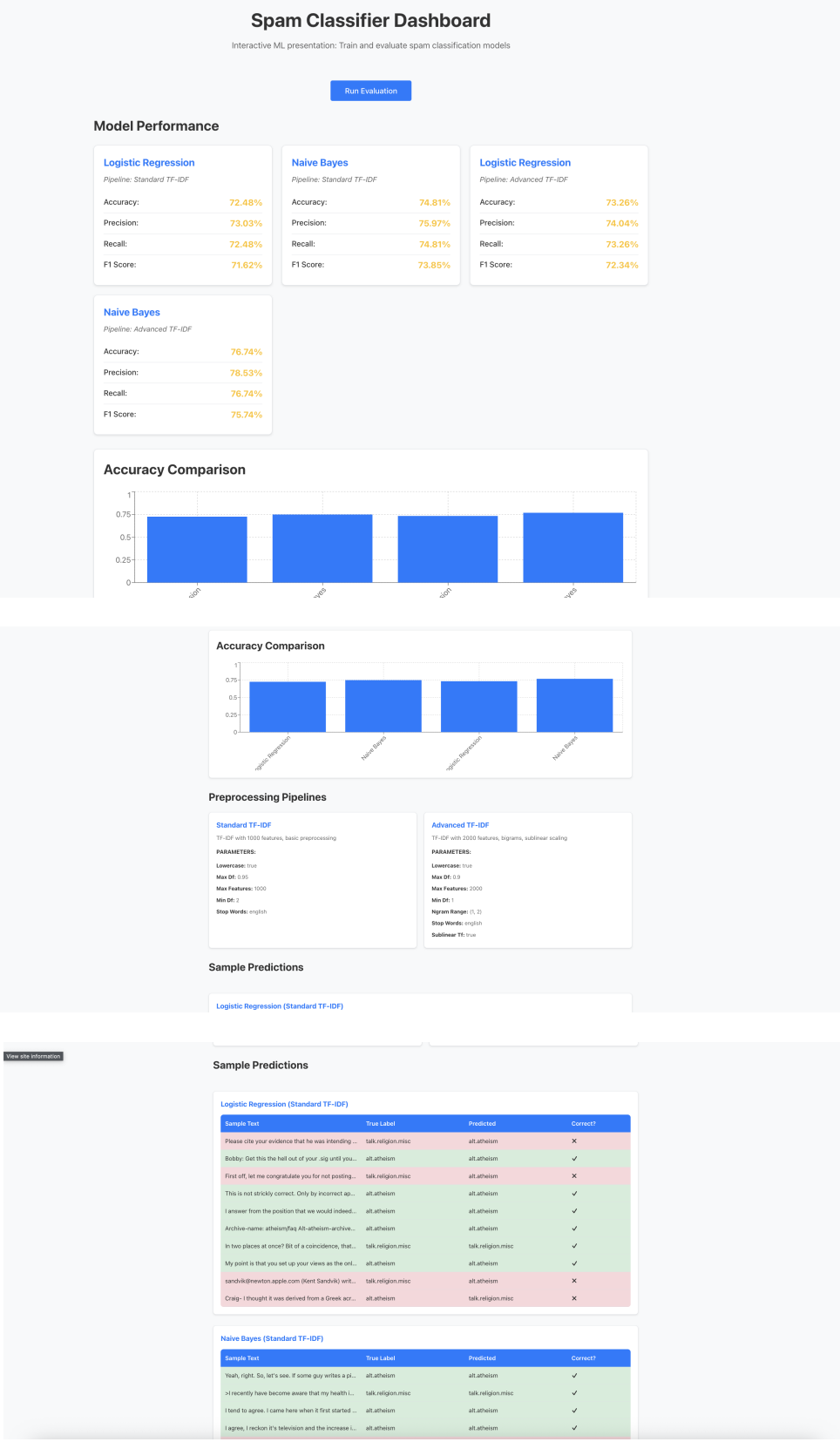


Figure A.29: Voting outputs for the machine learning task.

A.4.5 Specialists

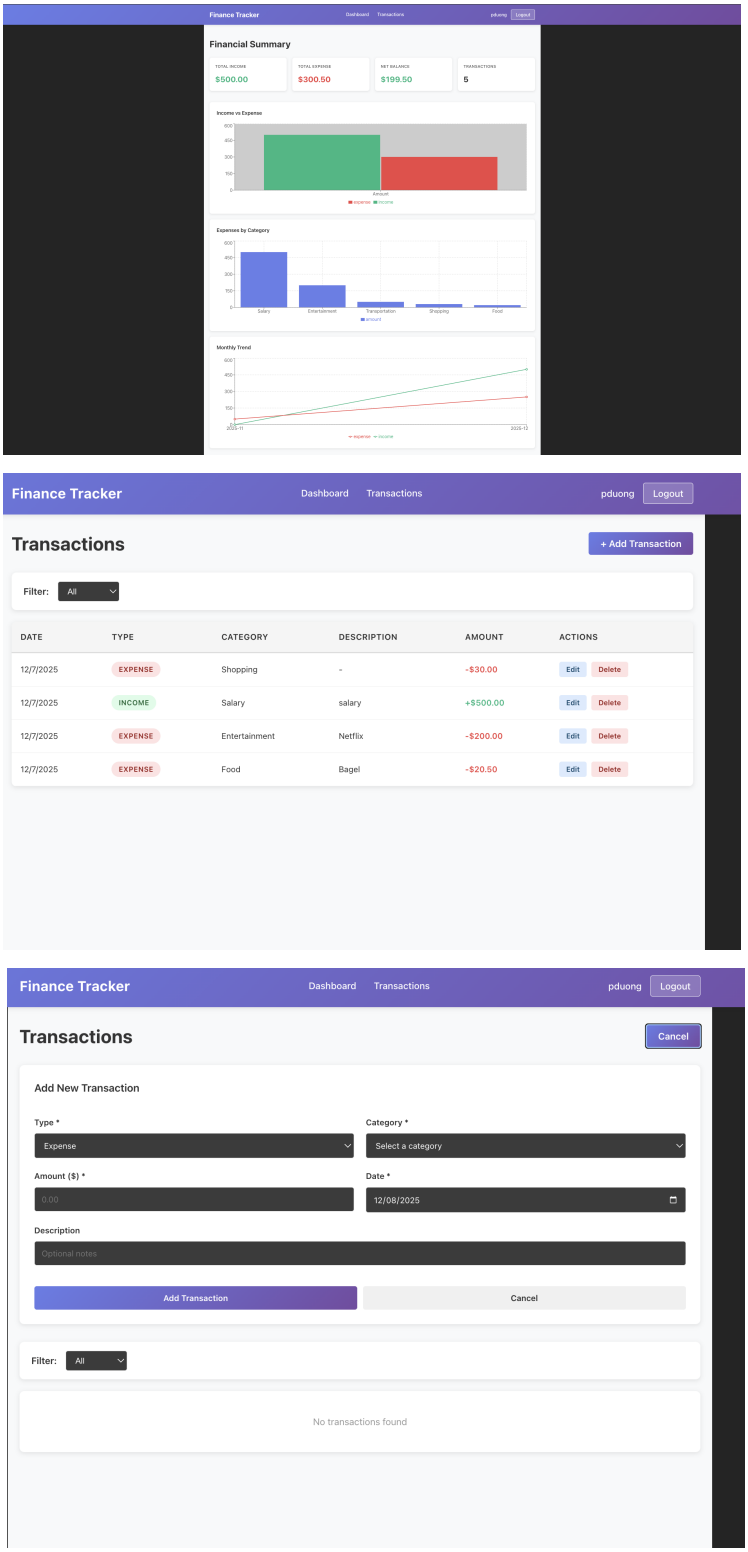


Figure A.30: Specialists outputs for the full-stack task.

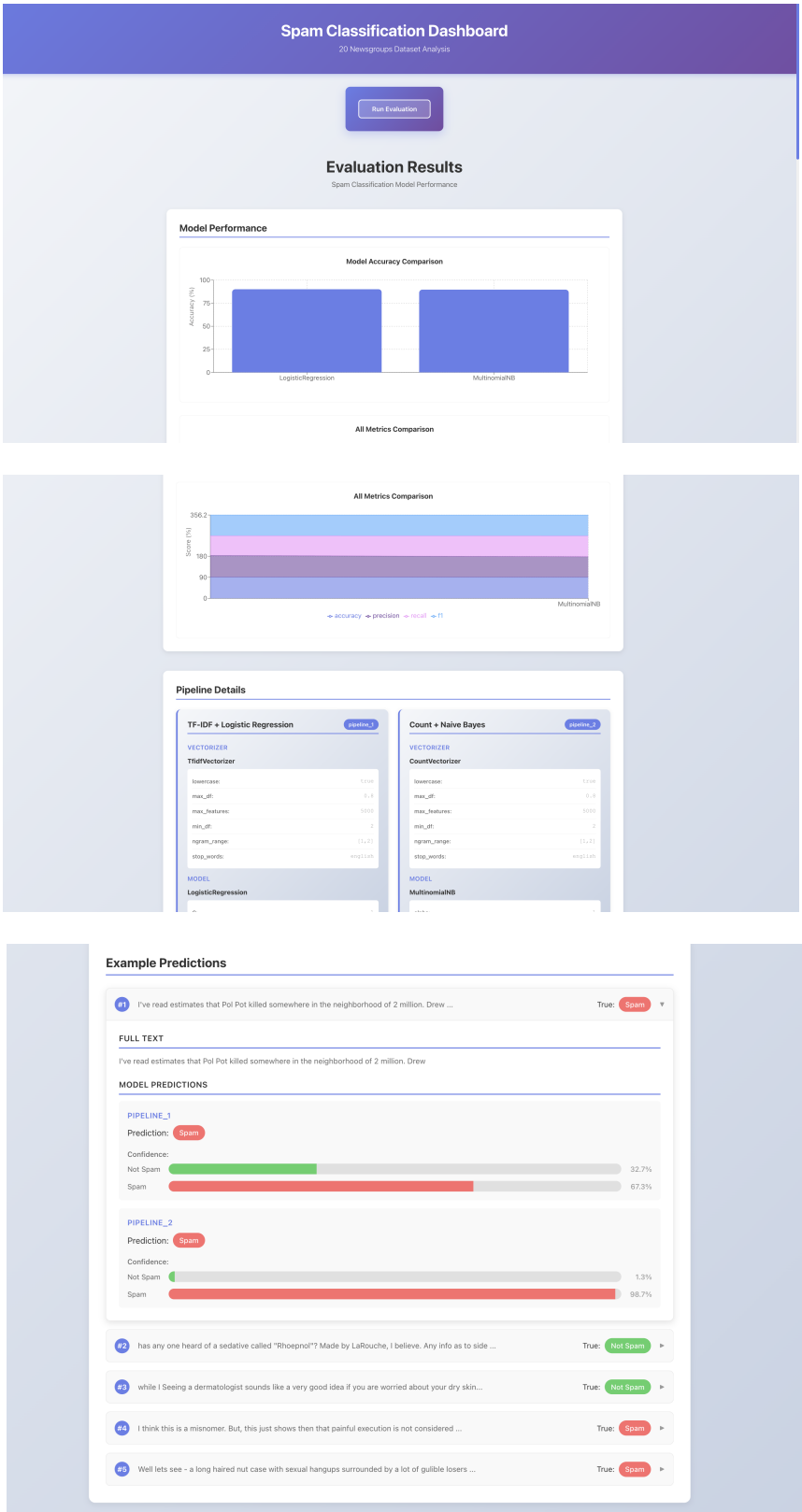


Figure A.31: Specialists outputs for the machine learning task.