Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Who Broke the Pipeline? Traceability andAccountability in Multi-Agent LLM SoftwareEngineering Pipelines

Jackson Beem
Oakland University
Amine Barrak
aminebarrak@oakland.edu

Oakland University
Emna Ksontini
University of North Carolina Wilmington

Research Article

Keywords: Multi-agent LLM pipelines, Role specialization, Accountability, Traceability, Code generation, Execution-based verification, Pipeline design

Posted Date: April 21st, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9350097/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License. Read Full License

Additional Declarations: No competing interests reported.

Who Broke the Pipeline? Traceability and Accountability in Multi-Agent LLM Software Engineering Pipelines

Jackson Beem[1], Amine Barrak[1*], Emna Ksontini[2]

[1*]Department of Computer Science and Engineering, Oakland University, Rochester, Michigan, USA.
[2]Department of Computer Science, University of North Carolina Wilmington, Wilmington, North Carolina, USA.

*Corresponding author(s). E-mail(s): aminebarrak@oakland.edu;

Abstract

Large language models are increasingly embedded across the DevOps pipeline, from planning and code generation to testing and deployment, yet multi-agent LLM pipelines remain opaque: errors propagate silently across stages with no systematic method to attribute failures to specific agents. We present a two-part study of traceable, accountable multi-agent pipelines with structured handoffs and role-level blame attribution. Part I evaluates a Planner, Executor, and Critic pipeline on three multiple-choice benchmarks using eight configurations of three frontier LLMs. Structured accountability recovers accuracy dramatically over unstructured pipelines (e.g., from 61.42% to 97.64%), and blame attribution reveals complementary role-specific aptitudes that enable data-driven model to role casting. Part II extends the pipeline to execution-grounded code generation on HumanEval, MBPP, and BigCodeBench, introducing a fourth Verifier agent. We find that LLM-based test evaluation is not a reliable substitute for actual test execution: false-fail rates reach 93%, triggering unnecessary re-generation that overwrites correct solutions. Replacing the LLM Critic with an execution-based verification harness eliminates verification harm entirely, achieves the highest accuracy on complex tasks, and reduces token cost by up to 57%. Test execution remains an irreplaceable component of the DevOps pipeline that cannot be delegated to LLM judgment alone. We further find that the Planner adds 1.8 to 3.0 times more tokens to the pipeline without meaningful accuracy gains in code generation, as the Executor produces comparable output with or without the Planner's formal specification.

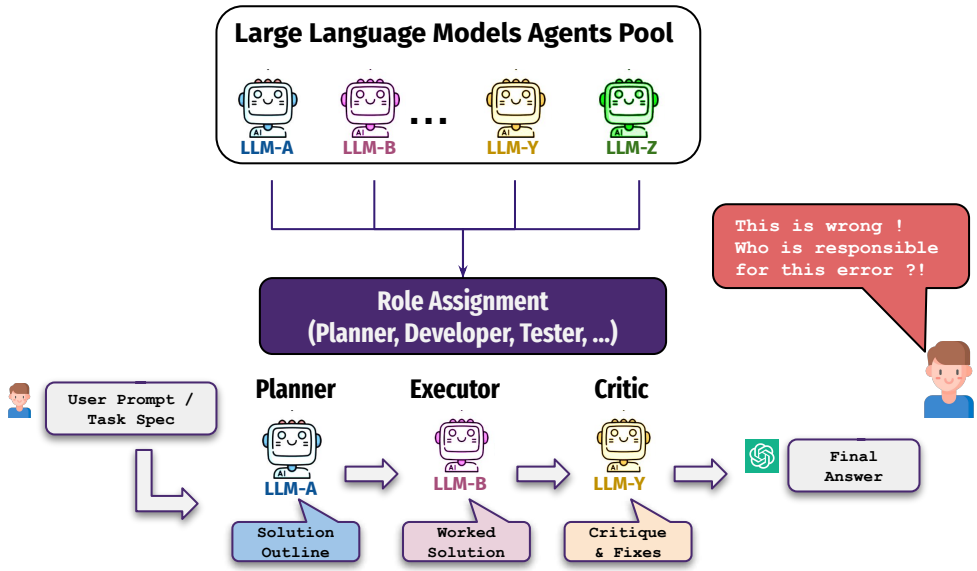


Fig. 1: A sequential multi-agent pipeline. A failure in the final output makes debugging and assigning responsibility a critical challenge.

Keywords: Multi-agent LLM pipelines, Role specialization, Accountability, Traceability, Code generation, Execution-based verification, Pipeline design

1 Introduction

The field of software engineering is undergoing a significant transformation, driven by the rise of Large Language Models (LLMs). This evolution is progressing from LLM-powered "co-pilots" that assist developers to fully autonomous, multi-agent systems capable of tackling complex software development tasks with minimal human intervention [1, 2]. A common architectural pattern is the *sequential multi-agent pipeline*, where specialized LLM-based agents collaborate in a predefined order to perform roles such as planning, development, and testing [3]. Systems like ChatDev and MetaGPT have demonstrated the potential of this approach, where task decomposition allows multiple agents to synergistically solve problems that would be intractable for a single monolithic model [4, 5].

However, this shift to sequential agent pipelines introduces a critical software engineering challenge: the loss of transparency and the immense difficulty of debugging. When agents are chained together, an error introduced by an early agent can silently cascade, corrupting the entire workflow and leading to a final output that is incorrect for reasons that are difficult to diagnose [3]. As illustrated in Figure 1, this turns the collaborative pipeline into a new kind of opaque system where failures are observable, but their origins are not. With developers already spending over 50% of their time on debugging traditional software, the introduction of these complex, non-deterministic

agentic systems threatens to exacerbate this challenge significantly [6]. Debugging a sequential pipeline of agents is therefore a crucial, yet largely unsolved, problem standing in the way of their reliable deployment [7, 8].

While the potential of multi-agent systems is well-documented, the engineering principles required to make them trustworthy and maintainable are still nascent [9]. Much of the existing research on systems like ChatDev [4], MetaGPT [5], and other debugging agents [6] focuses on demonstrating the capabilities of agent collaboration. While these studies are foundational, they often treat the pipeline as a black box, evaluating only the final output. Consequently, they do not provide a systematic analysis of the internal system dynamics or the root causes of failure. This is a critical omission, as recent work has shown that multi-agent systems can unexpectedly underperform strong, yet simpler, single-agent baselines due to coordination overhead and these very cascading errors [10]. The software engineering community has long recognized traceability as a cornerstone of building robust systems [11], yet this principle has not been systematically applied to the internal workings of agentic pipelines. This gap has recently been framed as the challenge of "automated failure attribution," that is, identifying which agent is responsible for a failure, a task that is critical for debugging but remains a labor-intensive and underexplored problem [12, 13].

To address this gap, in our prior work [14] we presented an empirical study of an **accountable sequential pipeline** of LLM-based agents. Rather than proposing a new framework, our contribution was a rigorous analysis of the internal dynamics of a `Planner` → `Executor` → `Critic` pipeline. We employed a blame attribution methodology to monitor the correctness of a solution as it passes through each stage, allowing us to quantify novel, role-specific behaviors such as **repair** (when an agent corrects an error from a preceding stage) and **harm** (when an agent introduces an error to a previously correct state). Through a large-scale study involving three state-of-the-art LLMs in eight distinct pipeline configurations across three diverse multiple-choice benchmarks, we investigated three research questions:

- **RQ1:** How do sequential multi-agent pipelines compare to the performance of monolithic LLMs?
- **RQ2:** How do individual LLMs perform in specialized roles, and what are the dynamics of error propagation and correction within the pipeline?
- **RQ3:** What are the trade-offs between accuracy, cost, and latency when designing accountable sequential agent pipelines?

The prior study was deliberately conducted in a controlled setting: multiple-choice benchmarks with known correct answers at every pipeline stage, enabling blame attribution through simple label matching. While this design allowed systematic exploration of model-role configurations and established the core methodology, quantifying role-specific behaviors through novel repair and harm rate metrics and producing a data-driven model-selection rationale, it remained a proxy setting. Multiple-choice benchmarks, while enabling precise measurement, do not reflect the generative, open-ended nature of real software engineering. In practice, when a multi-agent pipeline is deployed to generate code, refactor a module, or resolve a bug, there are no gold labels

available at intermediate stages. The correctness of a plan or a generated function can only be determined by executing the code against test suites, not by matching against a predefined answer key. This fundamental difference between the controlled setting and real-world deployment leaves several important questions unanswered and creates gaps that must be addressed before the methodology can be considered practically applicable.

First, because real software engineering tasks lack gold labels at intermediate pipeline stages, blame attribution must be grounded in execution outcomes rather than label matching. Determining whether an error originated in the Planner's specification, the Executor's implementation, or the Critic's review requires running the generated code and observing where correctness breaks down. Extending the methodology to *execution-grounded* evaluation is therefore essential for practical applicability. Second, the three-role pipeline lacked a structured recovery path: when the Critic detected a failure, the pipeline had no mechanism to act on this signal. In a real deployment, simply flagging an error without a recovery mechanism wastes the diagnostic effort. Recent work on iterative refinement and test-time compute has shown that verification and retry loops can substantially boost code correctness [15–17]. Systems such as AlphaCodium [15] and CodeChain [16] demonstrate that structured verification stages, where generated code is tested and iteratively revised, are critical for closing the gap between initial generation and correct solutions. Similarly, research on multi-agent verification has shown that scaling test-time compute through multiple verifiers can outperform simply scaling model parameters [17, 18]. These findings motivate the introduction of a fourth agent, a `Verifier`, to enable structured recovery from detected failures. However, the Verifier's effectiveness depends on the reliability of the gating signal that triggers it. If the Critic that decides whether to invoke the Verifier is miscalibrated, the recovery mechanism may do more harm than good, raising new questions about when verification helps or harms overall accuracy. Third, the Planner's value was assessed on multiple-choice reasoning tasks, but code generation presents a fundamentally different challenge. When the task is already expressed as a well-defined function signature with docstrings and test cases, a strong Executor may not benefit from an additional natural-language specification. Whether the Planner adds meaningful signal or merely introduces overhead in such settings remained unknown. Finally, in the prior study the Critic's LLM-based evaluation was the only available correctness signal. In code generation, an execution oracle provides ground-truth correctness independently of any LLM judgment. Recent studies on LLM-as-a-judge approaches have highlighted the limitations of using LLMs as evaluators, including positional bias and inconsistency [19]. Whether the LLM Critic adds value beyond what the test harness already provides, or whether it introduces noise that degrades the pipeline, is a question that only arises in the generative setting where objective verification through execution is possible.

This paper is a substantial extension of the prior work. We address the gaps identified above by transitioning from multiple-choice benchmarks to execution-grounded code generation, extending the pipeline with a Verifier agent, and introducing execution-based gating as an alternative to LLM-based critique. We evaluate the

extended pipeline across three code generation benchmarks spanning a difficulty spectrum from near-saturated tasks (HumanEval [20]) to complex, library-heavy problems (BigCodeBench [21]), and investigate three new research questions:

- **RQ4:** When does the Verifier improve or degrade pipeline accuracy under Critic miscalibration?
- **RQ5:** How much does the Planner contribute to performance in code generation pipelines?
- **RQ6:** Does replacing the Critic with execution-based gating improve verification reliability and overall performance?

Together, the six research questions across the two studies form a coherent research programme: the prior study established the methodology and quantified agent-role dynamics in a controlled symbolic setting, while this extension validates and extends those findings in the generative, execution-grounded setting that reflects real software engineering practice. The contrast between these two settings, and the degree to which findings transfer across them, constitutes a central contribution of this paper.

This paper makes five contributions: (1) a traceable, accountable multi-agent pipeline with blame attribution grounded in execution oracles, extending the label-matching methodology of the prior work [14] to generative settings; (2) the design and evaluation of a Verifier agent under two gating strategies (LLM Critic and direct execution gating), demonstrating that execution gating eliminates Verifier harm and achieves the highest accuracy on complex tasks; (3) empirical answers to RQ4 through RQ6 across three code generation benchmarks spanning a difficulty spectrum; (4) a cross-study generalizability analysis that compares role-pipeline dynamics in symbolic versus generative settings, identifying which findings transfer unconditionally, which are task-type conditional, and which are setting-specific; and (5) public release of datasets, prompts, logs, and evaluation scripts for both studies to support replication and extension.[1]

2 Related Work

Our work sits at the intersection of multi-agent LLM systems for software development, traceability and accountability in agentic workflows, and iterative refinement with verification mechanisms. This section surveys each of these areas and positions our contributions relative to the existing literature.

2.1 Multi-Agent Systems for Automated Software Engineering

The inherent limitations of single, monolithic LLMs in handling complex, multi-step tasks are well-established. A prominent issue is the "lost in the middle" problem, where models exhibit degraded performance in utilizing information positioned in the center of long context windows [22]. This has catalyzed a shift towards multi-agent systems, where a complex problem is decomposed and delegated to a society of specialized, collaborative agents [1–3].

[1] https://sites.google.com/view/mas-gain2025/home

Within software engineering, this paradigm has been successfully instantiated by frameworks that simulate entire development organizations. ChatDev [4] constructs a virtual software company where LLM agents embody roles like CEO, programmer, and tester, following a structured waterfall process to develop software. MetaGPT [5] employs agents as product managers and architects, mandating the generation of standardized artifacts like requirement documents and system designs before implementation begins. AutoGen [23] provides a general-purpose toolkit for orchestrating complex multi-agent conversations, while SWE-agent [24] focuses on repository-level bug fixing through autonomous tool use. Chain of Agents [25] addresses long-context tasks through sequential agent collaboration, and multi-agent LLMs have also been applied to complex engineering problem-solving beyond software [26].

The choice of LLM for each specialized agent is a critical design decision with economic and performance implications. Stronger models deliver higher accuracy and reasoning depth but incur greater cost and latency, while lighter models are faster and cheaper but less effective for planning or critical analysis [3, 27]. To automate this allocation, routing frameworks have emerged. CARGO [28] introduces a confidence-aware routing system that dynamically assigns prompts to the most suitable model, managing performance-cost trade-offs without manual intervention. Recent work has further shown that increasing inference-time compute through self-consistency sampling or structured deliberation can improve reasoning performance at the cost of additional tokens and latency [17, 29], reframing LLM inference as a cost-accuracy trade-off problem.

These systems convincingly demonstrate the feasibility of role-based decomposition for software engineering tasks. However, they evaluate only the final output, treating the pipeline as a black box. None performs role-specific blame attribution or quantifies the repair and harm rates of individual agents at each stage. Our work opens this black box by instrumenting every stage of the pipeline with traceable, accountable handoffs and measuring precisely where errors originate, propagate, and are corrected. Moreover, our prior study [14] demonstrated that heterogeneous pipelines, in which different models fill different roles based on empirically established strengths, consistently dominate Pareto frontiers in the accuracy-cost-latency space. This extension tests whether this result holds when the pipeline expands to four roles and operates in a generative code setting with substantially higher per-task costs and latency, where the compounding effect of re-generation loops amplifies the cost differential between configurations.

2.2 Traceability and Accountability in Agentic Pipelines

The software engineering community has long recognized traceability as a cornerstone of building dependable systems [11], yet this principle has not been systematically applied to the internal workings of agentic pipelines [9]. A primary challenge in current agentic systems is the lack of observability: when a pipeline produces an incorrect final output, it is often difficult to perform root cause analysis and identify which stage or agent introduced the error [3, 7]. This "black box" nature hinders debugging and iterative improvement. Observable and diagnosable pipelines are also central to the

emerging field of agent operations, which advocates for monitoring and tracing agentic workflows in production [30].

The concept of "automated failure attribution," identifying which agent is responsible for a pipeline failure, has been framed as a critical challenge by Zhang et al. [12] and by Triantafyllou et al. [13] in the context of multi-agent sequential decision making. Recent work has also shown that multi-agent systems can unexpectedly underperform simpler single-agent baselines due to coordination overhead and cascading errors [10], underscoring the need for systematic debugging methods. Research on LLM-as-a-judge has further investigated model-based evaluation, revealing both effectiveness and systematic biases such as positional bias and inconsistency in automated judging systems [19], raising questions about the reliability of LLM-based critique stages in pipelines.

Our prior work [14] introduced a blame attribution methodology using label matching in a controlled multiple-choice setting with three benchmarks: PythonIO [31], LogiQA [32], and AGIEval [33]. The present paper extends that methodology to execution-grounded attribution, replacing gold-label comparison with runtime test-suite results. This transition is enabled by code generation benchmarks that provide execution oracles. HumanEval [20] provides 164 hand-crafted Python function synthesis problems with deterministic test cases, representing clean, well-specified tasks. MBPP [34] offers crowd-sourced programming problems at moderate complexity. BigCodeBench [21] extends evaluation to library-heavy, multi-step synthesis tasks that more closely approximate real-world software engineering. The progression from HumanEval through MBPP to BigCodeBench represents an increasing difficulty spectrum, which our study uses to investigate how pipeline dynamics change with task complexity. Critically, execution-based evaluation removes the requirement for intermediate ground-truth answers and enables blame attribution in generative settings where such labels do not exist, providing a runtime oracle that resolves a key limitation of the prior study.

2.3 Iterative Refinement and Verification in Code Generation

A growing number of studies have examined how LLMs can improve their outputs step by step by using feedback loops. The Self-Refine framework [35] demonstrated that a single LLM can improve its output by generating self-feedback and then acting upon it. Self-Debug [36] adapted this principle for code, prompting an agent to analyze and explain a bug before attempting a fix. Automated code review agents [37] further explore the potential of model-based critique in development workflows. Moving beyond single-agent refinement, AlphaCodium [15] introduces a multi-stage code generation pipeline with iterative testing and refinement, demonstrating that flow engineering with structured verification stages substantially outperforms direct prompting. CodeChain [16] uses a chain of self-revisions guided by representative sub-modules. Reflexion [38] employs structured verbal reflection to correct prior reasoning errors across episodes. Execution-guided program repair uses unit test failures to iteratively refine generated programs [39]. AgentCoder [40] adopts a multi-agent design with separate agents for code generation, testing, and iterative optimization. Research on

iterative pipeline refinement has shown that modifying and evaluating one component at a time leads to more stable improvements [41], and the use of multiple diverse verifiers to evaluate an output has been shown to improve reliability [18].

Several of these systems use execution feedback directly to drive repair loops, bypassing LLM-based critique entirely. Our study provides controlled evidence for this design choice: we compare a Verifier driven by an LLM Critic's verdict against a Verifier driven by raw execution results, demonstrating that the latter eliminates Verifier-induced harm while achieving higher accuracy on complex tasks. Importantly, our Verifier is deliberately non-generative in both configurations: it gates, classifies, and routes, but does not produce code or patches itself. This design is grounded in verification-asymmetry arguments, which observe that verifying correctness is often easier than generating solutions and can therefore be delegated to specialized evaluators [42]. This separation keeps blame attribution clean, as any code modification occurs only through the Executor.

3 Study Design

This paper presents a two-part empirical investigation of traceable, accountable multi-agent LLM pipelines, as illustrated in Figure 2. Part I (RQ1–RQ3) evaluates blame attribution in a controlled multiple-choice setting, where gold labels enable precise label-matching at every pipeline stage. Part II (RQ4–RQ6) extends the methodology to execution-grounded code generation, introduces a fourth Verifier agent, and replaces label matching with runtime test-suite results. The remainder of this section presents each part with its own benchmarks, models, configurations, and blame attribution methodology, followed by shared evaluation metrics.

3.1 Part I: Multiple-Choice Benchmarks (RQ1–RQ3)

3.1.1 Benchmarks

We curated multiple-choice datasets, retaining only prompts with valid ground-truth answers to ensure consistent evaluation of accuracy across role configurations and blame attributions. All prompts were aligned to the standard answer space ($\{A, B, C, D, E\}$). We evaluate on three complementary benchmarks:

- **PythonIO** (127 prompts): programming and algorithmic reasoning problems expressed in natural language, stressing systematic reasoning and code-like logic [31].
- **LogiQA** (283 prompts): formal logic and deductive reasoning tasks; originally released by Liu et al. (2020) and publicly available via GitHub [32].
- **AGIEval** (263 prompts): standardized test-style questions (math, reading, commonsense); derived from public exam data (e.g. Gaokao, SAT) and repurposed in AGIEval benchmark [33].

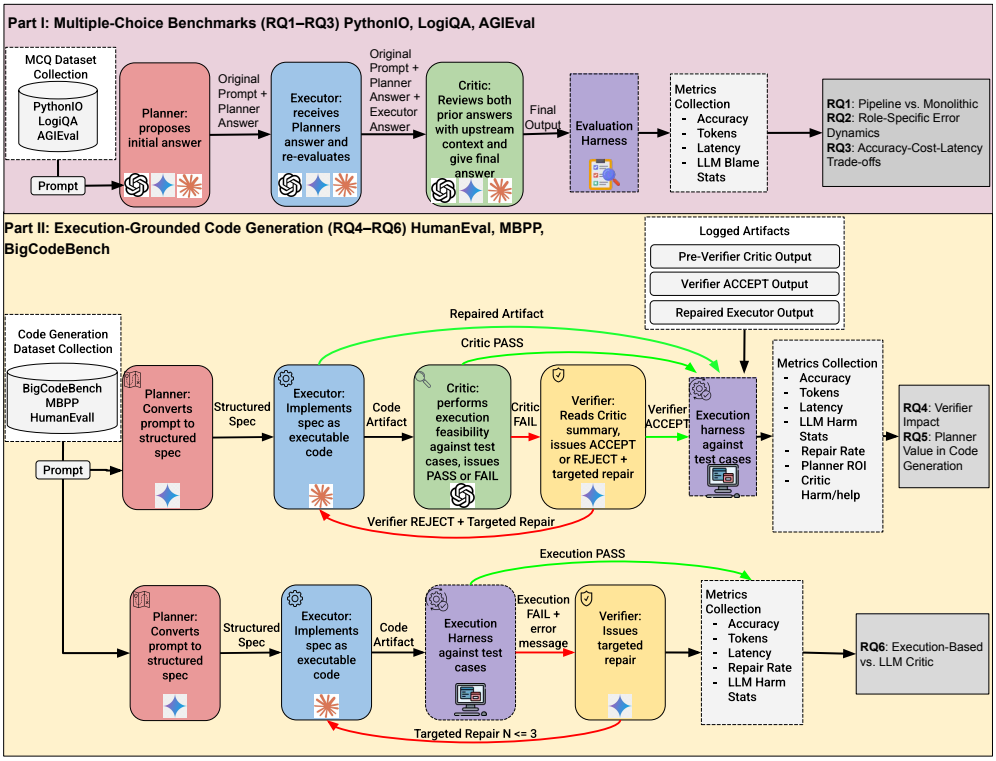


Fig. 2: Overview of the two-part study design. Part I evaluates a three-role pipeline (`Planner` → `Executor` → `Critic`) on multiple-choice benchmarks using label-matching blame attribution. Part II extends the pipeline with a `Verifier` agent and evaluates on code generation benchmarks using execution-grounded blame attribution.

3.1.2 Model Selection

For Part I, we selected three frontier LLMs that held top positions on public leaderboards during our evaluation period. Table 1 summarizes the models and their characteristics. All three models are used interchangeably across all pipeline roles (Planner, Executor, Critic), yielding eight distinct pipeline configurations.

3.1.3 Experimental Conditions

We compare three inference regimes:
(1) Single-model baselines. Each model answers end-to-end with the prompt above. These runs establish individual capability and serve as reference points.

(2) Pipelines without traceability. We compose three-step pipelines (length 3) using ordered model triples from $\{A, B, C\}$, e.g., $\mathrm{AAA}, \mathrm{BBB}, \mathrm{CCC}, \mathrm{ABC}, \mathrm{ACB}, \ldots$. Each step consumes the previous step's artifact (question and current answer) and emits a new one-letter answer. We score only the final output.

Table 1: Models used in Part I (MCQ study). All models serve interchangeably in all three pipeline roles.

Model	Label	Key Characteristics
GPT-4o	(A)	Stable, low-variance responses; strong text + code reasoning [43]
Claude Sonnet 4	(B)	Strict instruction-following; effective multi-step revision [44]
Gemini 2.5 Pro	(C)	High-capability reasoning; reliable schema compliance [45]

(3) Pipelines with traceability (*blame tracking*). We instrument the same configurations to capture stagewise inputs/outputs and deltas against gold labels. For each item we compute whether the intermediate step *repaired* a wrong answer, *harmed* a correct answer, or performed a *no-op*.

3.1.4 Pipeline Architecture and Blame Attribution

We use a three-role pipeline: a **Planner** proposes an initial answer, an **Executor** solves the task conditioned on the planner's output, and a **Critic** reviews or revises the executor's answer. The pipeline publishes a single final answer by preferring later stages (critic → executor → planner). To analyze accountability, we define a *blame function* that compares each stage's answer to the ground truth and sets binary flags indicating whether a downstream stage *repaired* a wrong upstream answer or *harmed* a correct one; the *error origin* is the earliest stage whose mistake remains unrepaired in the final output (NONE if the final answer is correct). The operational logic is summarized in Algorithm 1.

Algorithm 1 Traceable Pipeline: Roles and Blame Logic (Part I)

Require: Dataset $D = \{(x_i, y_i)\}_{i=1}^{N}$; models M_P (Planner), M_E (Executor), M_C (Critic)

Ensure: For each i: stage answers (P, E, C), final F, blame flags, error origin

for $i \leftarrow 1$ **to** N **do**
  $P \leftarrow M_P(x_i)$
  $E \leftarrow M_E(x_i, P)$
  $C \leftarrow M_C(x_i, P, E)$
  Final selection:
  if C is defined **then**
    $F \leftarrow C$
  else if E is defined **then**
    $F \leftarrow E$
  else
    $F \leftarrow P$
  end if
  Blame flags:
  `planner_error`$[i] \leftarrow (P \neq y_i)$
  `executor_repair`$[i] \leftarrow (P \neq y_i) \wedge (E = y_i)$
  `executor_harm`$[i] \leftarrow (P = y_i) \wedge (E \neq y_i)$
  `critic_repair`$[i] \leftarrow (E \neq y_i) \wedge (C = y_i)$
  `critic_harm`$[i] \leftarrow (E = y_i) \wedge (C \neq y_i)$
  Error origin:
  if $F = y_i$ **then**
    `origin`$[i] \leftarrow$ NONE
  else
    if $(E = y_i) \wedge (C \neq y_i)$ **then**
      `origin`$[i] \leftarrow$ CRITIC
    else if $(P = y_i) \wedge (E \neq y_i)$ **then**
      `origin`$[i] \leftarrow$ EXECUTOR
    else
      `origin`$[i] \leftarrow$ PLANNER
    end if
  end if
  Record $(P, E, C, F, \text{flags}, \texttt{origin}[i])$
end for

3.1.5 Cost and Latency

To assess the operational expense of each approach, we compute the monetary cost. Let $R_{\text{in}}(m)$ and $R_{\text{out}}(m)$ be the provider's prices (USD) per 1,000 tokens for input and output respectively for model m. When per-call cost is needed, it is computed itemwise as $\frac{\text{prompt}_i}{1000} R_{\text{in}}(m) + \frac{\text{completion}_i}{1000} R_{\text{out}}(m)$ and summarized by the median across

items. For the three-role pipeline, the total cost for a single task is:

$$\text{Pipeline Cost} = \text{Cost}_{\text{Planner}} + \text{Cost}_{\text{Executor}} + \text{Cost}_{\text{Critic}}$$

All reported monetary totals were computed with the fixed per-token rates in Table 2. We also record the end-to-end latency for each task in seconds and report the median.

Table 2: Per-token pricing for Part I models.

Model	Input (USD/1K)	Output (USD/1K)	Source
GPT-4o (A)	$0.00500	$0.02000	[46]
Claude Sonnet 4 (B)	$0.00300	$0.01500	[47]
Gemini 2.5 Pro (C)	$0.00125	$0.01000	[48]

3.2 Part II: Execution-Grounded Code Generation (RQ4–RQ6)

Part II extends the methodology established in Part I to the target setting: real code generation with execution-based correctness. The pipeline is expanded with a fourth agent (the Verifier), blame attribution is grounded in a sandboxed execution oracle rather than label matching, and a new configuration replaces the LLM Critic with direct execution gating.

3.2.1 Benchmarks

Part II evaluates on three code generation benchmarks representing a difficulty spectrum from well-specified function synthesis to complex, library-heavy multi-step tasks.

HumanEval [20] (164 tasks) consists of hand-crafted Python function-level problems defined by a natural-language docstring, a required function signature, and a set of deterministic unit tests. Problems are typically short-horizon, with clear specifications and well-defined expected behavior. HumanEval represents the easiest end of the difficulty spectrum and serves as a controlled baseline where we expect high accuracy from frontier models, allowing us to study pipeline dynamics under near-optimal conditions.

MBPP [34] (200 tasks, randomly sampled; task identifiers and selection hash are provided in the replication package) provides crowd-sourced Python programming tasks with concise natural-language descriptions and compact unit tests. Specification quality is more variable than HumanEval, with some problems requiring interpretation of underspecified requirements. MBPP represents medium complexity and tests how the pipeline handles moderate ambiguity in task descriptions.

BigCodeBench [21] (200 tasks, randomly sampled; task identifiers and selection hash are provided in the replication package) contains longer-horizon, library-heavy code generation problems requiring multi-step reasoning, integration of multiple APIs, and handling of complex data structures. BigCodeBench represents the hard end of the difficulty spectrum and is the closest approximation to real-world software engineering tasks, where we expect the pipeline's recovery mechanisms to have the greatest impact.

The progression from HumanEval through MBPP to BigCodeBench provides an increasing difficulty spectrum, which allows us to investigate how pipeline dynamics, and in particular the value of the Verifier and the Critic, change with task complexity.

3.2.2 Model Selection

The model assignment for Part II was derived directly from the role-specific performance findings of Part I [14], using the successor versions of the same model families. Table 3 summarizes the assignments and their rationale.

Table 3: Model assignments for Part II (code generation study). Each role is filled by a fixed model selected based on Part I performance findings.

Role	Model	Selection Rationale
Planner	Gemini 2.5 Pro Preview	Lowest Planner error rate across all Part I benchmarks (0.79% to 15.19%)
Executor	Claude Sonnet 4.5	Highest Executor repair rate (10.01%) and lowest harm rate (0.25%) in Part I
Critic	GPT-5.1	Highest Critic repair rate (5.20%) in Part I
Verifier	Gemini 2.5 Pro Preview	Strong schema compliance and low harm rates in Part I, suitable for non-generative binary gating

Gemini 2.5 Pro Preview was selected as Planner because it exhibited the lowest error rate across all benchmarks in Part I. Claude Sonnet 4.5, the successor to Claude Sonnet 4, was selected as Executor because Claude demonstrated the highest repair rate and lowest harm rate in the Executor role. GPT-5.1, the successor to GPT-4o, was selected as Critic because GPT-4o achieved the highest Critic repair rate in Part I. Gemini 2.5 Pro Preview was also assigned the Verifier role because its strong schema compliance and instruction-following, evidenced by its low harm rates across both Part I roles, make it suitable for a non-generative gating function that requires precise, reliable binary decisions.

3.2.3 Agent Roles and Pipeline Configurations

Four agent roles are defined, each with strict functional boundaries to maintain clean blame attribution. The pipeline also supports a variant in which the Critic is removed and replaced by direct execution gating.

The **Planner** converts a problem description into a structured specification by defining intent, constraints, interfaces, and acceptance criteria without producing executable artifacts. It performs high-level decomposition and requirement scoping to bound the solution space while leaving implementation choices open. The Planner is prohibited from generating code or observing test outcomes.

The **Executor** produces an executable implementation that satisfies either the Planner's specification (in configurations with a Planner) or the raw problem statement (in configurations without a Planner). It translates abstract requirements into concrete code artifacts and may include minimal internal checks, but does not reinterpret requirements or assess global correctness. The Executor is the only agent that produces code.

The **Critic** evaluates functional correctness by examining test execution results and summarizing outcomes as PASS/FAIL signals with failure localization. The Critic operates as a black-box evaluator that observes execution behavior rather than implementation intent, and it does not propose alternative implementations or patches. The Critic's output is a structured verdict with a justification, not revised code. The Critic is present in the PECV configuration but absent in the PEGV (Exec Gate) configuration.

The **Verifier** is a conservative gatekeeper that is only invoked when the test suite execution fails. If the Executor's code passes all test cases, the pipeline terminates successfully without invoking the Verifier. When invoked, the Verifier issues a binary ACCEPT or REJECT decision and, on REJECT, provides a targeted repair request that is routed back to the Executor for re-generation. The Verifier's behavior differs across configurations. In the PECV configuration, the Verifier acts on the Critic's evaluation summary together with the execution results; it is invoked at most once, and after the Executor re-generates the pipeline accepts the new artifact regardless of its test outcome. In the PEGV configuration, the Verifier acts directly on the test harness execution results without any LLM-mediated interpretation; it supports an iterative loop of up to three re-generation iterations ($k \leq 3$), re-invoking the Verifier after each failed re-execution. The Verifier is deliberately non-generative: it decides, but does not create.

Five configurations form a controlled ablation design that isolates the contributions of the Planner, Critic, and Verifier:

- **Monolithic**: A single LLM (Claude Sonnet 4.5) receives the problem statement and produces code end-to-end. Claude Sonnet 4.5 was selected because it serves as the Executor in all pipeline configurations, making the comparison meaningful: the question is whether adding Planner, Critic, and Verifier agents around the same code-generating model improves over that model operating alone.

- **Pipeline (No Planner) [EC]**: The Executor receives the raw problem statement directly. The Critic evaluates the output. This configuration ablates the Planner.
- **Pipeline [PEC]**: The full three-role pipeline. The Planner produces a specification, the Executor implements it, and the Critic evaluates.
- **Pipeline + Verifier [PECV]**: The four-role pipeline with LLM Critic. The initial pass follows Planner → Executor → Critic → test execution. If all tests pass, the pipeline terminates. If any test fails, the Verifier is invoked once; on REJECT, it sends a repair request back to the Executor, which re-generates code. The Critic is not re-invoked; the pipeline accepts the re-generated artifact as the final output. The Verifier acts as a single-pass gatekeeper: at most one Verifier invocation and two Executor invocations per task. PECV answers RQ4.
- **Pipeline (Exec-Based) + Verifier [PEGV]**: The four-role pipeline without the LLM Critic. The initial pass follows Planner → Executor → test execution. If all tests pass, the pipeline terminates. If any test fails, execution results go directly to the Verifier, which issues ACCEPT or REJECT. On REJECT, the Executor re-generates, the new code is executed, and if it still fails the Verifier is invoked again. This iterative loop continues for up to three iterations ($k \leq 3$). For k=3, the Executor runs four times (one initial plus three re-generations) and the Verifier runs three times. This configuration ablates the LLM Critic, replacing it with the execution oracle. PECV versus PEGV answers RQ6.

3.2.4 Execution Oracle

Correctness in Part II is determined by a sandboxed Python execution environment. For each task, the Executor's generated code is run against the benchmark's provided test suite in an isolated environment with a fixed timeout. A solution either passes every test case or it fails.

All model invocations across both study parts use temperature 0.0 (greedy decoding), making each pipeline run fully deterministic for a given input. Single-run reporting is therefore appropriate, as repeated runs on the same input produce identical outputs.

In the Monolithic, EC, and PEC configurations, the Executor generates code once and the result is evaluated directly. The PECV and PEGV configurations extend this with Verifier-mediated recovery, but with different iteration strategies.

In the **PECV** configuration, the initial pass follows Planner → Executor → Critic → test execution. If all test cases pass, the pipeline terminates successfully and the Verifier is never invoked. If any test case fails, the Verifier is invoked once. When it issues REJECT, a targeted repair request is routed back to the Executor, which re-generates a new solution. The Critic is not re-invoked; the pipeline accepts this re-generated artifact as the final output. PECV therefore supports at most one Verifier invocation and at most two Executor invocations per task.

In the **PEGV** configuration, the initial pass follows Planner → Executor → test execution (with no LLM Critic). If all test cases pass, the pipeline terminates and the Verifier is never invoked (k=0). If any test case fails, the Verifier is invoked and, on REJECT, a repair request is routed back to the Executor. The Executor re-generates

a new solution, which is executed again. If the new execution still fails, the Verifier is invoked again, and the cycle repeats. Each such Verifier-triggered iteration increments k by one. The pipeline permits up to three iterations ($k \leq 3$), after which the most recent artifact is accepted regardless of the Verifier's verdict. For k=3, the Executor is invoked a total of four times (one initial generation plus three re-generations) and the Verifier three times.

We report the final pass@1 of the pipeline, that is, whether the last accepted artifact passes all test cases, as this reflects the artifact the pipeline actually delivers.

The execution harness handles runtime errors, infinite loops (via timeout), and import failures. All execution outcomes are logged as structured records including the boolean correctness verdict, error type, and execution time. In the PECV configuration, the harness output is provided to the Critic during the initial pass; if re-generation occurs, the final artifact is accepted without re-invoking the Critic. In the PEGV configuration, the harness output goes directly to the Verifier when tests fail.

3.2.5 Blame Attribution via Execution

The blame attribution methodology is extended from Part I to accommodate execution-based correctness, the addition of the Verifier, and the Exec Gate variant.

Executor blame is assessed by executing the Executor's code artifact through the test harness before any downstream evaluation. The Executor's strict pass@1 result establishes its raw contribution, isolated from critique or verification.

Critic blame (PECV configuration only) is assessed by comparing the Critic's PASS/FAIL verdict to the actual execution result. The Critic false-fail rate is the fraction of cases where the Critic issues FAIL but the code actually passes all tests. The Critic false-pass rate is the fraction of cases where the Critic issues PASS but the code actually fails.

Verifier blame is assessed through help and harm rates. The Verifier help rate is the fraction of Verifier invocations that convert a failing pipeline to a passing one. The Verifier harm rate is the fraction that convert a passing pipeline to a failing one. In PEGV, the Verifier is only invoked when execution genuinely fails, which eliminates the possibility of harm from false triggering.

Planner blame is assessed by comparing PEC versus EC pass@1 per benchmark.

3.2.6 Cost and Latency

Cost computation follows the same methodology as Part I, using updated per-token pricing for the successor models. For configurations with a Verifier, the cost becomes variable. The Critic is only invoked during the initial pass; re-generation iterations involve only the Verifier and the Executor. The cost formulas for the two Verifier configurations are:

PECV ($k \in \{0, 1\}$, at most one Verifier invocation):

$$\text{Cost}_{\text{PECV}} = \text{Cost}_{\text{Planner}} + \text{Cost}^{(0)}_{\text{Executor}} + \text{Cost}_{\text{Critic}} + k \cdot \left(\text{Cost}_{\text{Verifier}} + \text{Cost}^{(1)}_{\text{Executor}} \right)$$

PEGV ($k \in \{0, 1, 2, 3\}$, up to three iterative re-generation cycles):

$$\text{Cost}_{\text{PEGV}} = \text{Cost}_{\text{Planner}} + \text{Cost}^{(0)}_{\text{Executor}} + \sum_{j=1}^{k} \left(\text{Cost}^{(j)}_{\text{Verifier}} + \text{Cost}^{(j)}_{\text{Executor}} \right)$$

In both formulas, superscript (0) denotes the initial pass and k is the number of Verifier invocations. PEGV has no Critic cost because the LLM Critic is replaced by the execution oracle. For PECV with k=1, the Executor runs twice and the Verifier once. For PEGV with k=3, the Executor runs four times and the Verifier three times.

All reported monetary totals were computed with the fixed per-token rates in Table 4. The PEGV configuration saves the cost and latency of the Critic LLM call but may incur higher iteration costs because its iterative loop permits up to three re-generation cycles.

Table 4: Per-token pricing for Part II models.

Role	Model	Input (USD/1K)	Output (USD/1K)	Source
Planner & Verifier	Gemini 2.5 Pro Preview	\$0.00125	\$0.01000	[48]
Executor	Claude Sonnet 4.5	\$0.00300	\$0.01500	[47]
Critic	GPT-5.1	\$0.00125	\$0.01000	[46]

3.3 Evaluation Metrics

Both study parts share a common set of operational metrics (token usage, cost, latency) while employing distinct correctness and blame metrics that reflect their respective evaluation paradigms. We describe the shared metrics first, followed by the metrics specific to each part.

3.3.1 Shared Metrics

Token statistics. For every model invocation, we record the number of input (prompt) tokens and output (completion) tokens. These counts capture the complexity of the context passed to each agent and form the basis for cost computation. We report the median across all tasks to provide a robust central measure of token consumption per role and per configuration.

Cost. Monetary cost is computed per invocation as $\frac{\text{prompt}_i}{1000} R_{\text{in}}(m) + \frac{\text{completion}_i}{1000} R_{\text{out}}(m)$, where $R_{\text{in}}(m)$ and $R_{\text{out}}(m)$ are the provider's per-1K-token rates for model m (Tables 2 and 4).

Latency. We record the wall-clock time (in seconds) for each complete pipeline run, measured from the first API call to the receipt of the final output. For Part II configurations with re-generation loops, latency includes all iterations. We report the median latency per task to characterize the time overhead of each configuration.

3.3.2 Part I Metrics: Label-Matching Evaluation

Accuracy (label matching). The primary correctness metric for Part I is the percentage of tasks for which the pipeline's final output matches the gold-standard answer. Since all benchmarks use a discrete answer space ($\{A, B, C, D, E\}$), matching is exact string comparison. This metric answers RQ1 by comparing pipeline configurations against single-model baselines.

Repair rate. The fraction of tasks where an agent corrects a wrong answer inherited from its upstream stage. Formally, the Executor repair rate is $\frac{|\{i:P_i \neq y_i \wedge E_i = y_i\}|}{|\{i:P_i \neq y_i\}|}$, and the Critic repair rate is defined analogously over Executor errors. This metric quantifies each role's error-correction capability and is central to RQ2.

Harm rate. The fraction of tasks where an agent converts a correct upstream answer into an incorrect one. The Executor harm rate is $\frac{|\{i:P_i = y_i \wedge E_i \neq y_i\}|}{|\{i:P_i = y_i\}|}$, and the Critic harm rate is defined analogously. This metric captures the risk that a downstream role may degrade rather than improve the pipeline's output, and is also part of RQ2.

Error origin. For each task where the pipeline produces an incorrect final answer, we trace the earliest stage whose mistake remains unrepaired in the final output. The error origin is assigned as follows: CRITIC if the Executor was correct but the Critic introduced a wrong answer; EXECUTOR if the Planner was correct but the Executor introduced a wrong answer that persists; PLANNER otherwise. When the final answer is correct, the origin is NONE. The distribution of error origins across configurations reveals which roles are the dominant sources of failure (RQ2) and informs the model-role assignment strategy used in Part II.

3.3.3 Part II Metrics: Execution-Grounded Evaluation

Pass@1 (execution-based accuracy). The primary correctness metric for Part II is strict pass@1: the percentage of tasks for which the pipeline's final code artifact passes every test case in the benchmark's test suite. Unlike Part I's label matching, this metric reflects functional correctness validated by actual code execution in a sandboxed environment. For configurations with Verifier-mediated recovery (PECV, PEGV), pass@1 is computed on the final accepted artifact, reflecting the output the pipeline would actually deliver.

Executor repair rate (re-generation). In PECV and PEGV configurations, when the Verifier rejects the Executor's code, the Executor receives a targeted repair request and generates a new solution. The Executor repair rate is the fraction of re-generation attempts that convert a previously failing solution into a passing one. In PECV, this is measured over the single re-generation attempt; in PEGV, it is measured across up to three attempts. This metric captures the Executor's ability to self-correct given feedback and is relevant to both RQ4 and RQ5.

Critic false-fail rate. The fraction of tasks (in the PECV configuration) where the Critic issues a FAIL verdict but the code actually passes all test cases upon execution. False failures trigger an unnecessary Verifier invocation and a wasted re-generation, increasing both cost and latency without improving correctness. This metric is central to RQ6's comparison of LLM-based and execution-based gating.

Critic false-pass rate. The fraction of tasks (in the PECV configuration) where the Critic issues a PASS verdict but the code actually fails at least one test case upon execution. False passes allow incorrect code to bypass the repair loop, resulting in lower final accuracy. Together with the false-fail rate, the false-pass rate characterizes the Critic's calibration as a correctness judge (RQ4, RQ6).

Verifier help rate. The fraction of Verifier invocations that result in a net improvement, defined as cases where the Verifier triggers re-generation and the final artifact passes after the loop, whereas the pre-Verifier artifact would have failed. This metric quantifies the Verifier's constructive contribution and is central to RQ4.

Verifier harm rate. The fraction of Verifier invocations that result in a net degradation, defined as cases where the Verifier triggers re-generation and the final artifact fails, whereas the pre-Verifier artifact would have passed. In the PEGV configuration, the Verifier is only invoked when execution genuinely fails, which eliminates the possibility of harm from false triggering. Comparing Verifier harm rates across PECV and PEGV isolates the impact of Critic miscalibration on the downstream repair loop (RQ4, RQ6).

Planner ROI. The Planner's value in code generation is assessed by comparing the pass@1 of configurations with and without the Planner (PEC vs. EC). A positive delta indicates that the Planner's specification improves downstream code quality enough to justify its token and latency overhead. A negative or negligible delta suggests that the Planner's decomposition may be unnecessary or even counterproductive for certain benchmark difficulties. This metric directly answers RQ5.

Re-generation dynamics. For the PEGV configuration, which supports an iterative re-generation loop, we report the distribution of the number of Verifier-triggered re-generation iterations per task ($k \in \{0, 1, 2, 3\}$, where k=0 means the initial execution passed all tests and the Verifier was never invoked, and k=3 means the maximum number of iterations was reached). For k=3, the Executor runs a total of four times and the Verifier three times. We also report the cumulative pass@1 after each iteration to characterize the diminishing-returns curve of the re-generation loop. For PECV, where the Verifier is invoked at most once ($k \in \{0, 1\}$), we report the fraction of tasks that required Verifier intervention and the success rate of the single re-generation attempt. These distributions inform practical deployment decisions about the value of iterative repair versus single-pass verification.

4 Results

This section presents the empirical findings organized by research question. For each RQ, we state the motivation behind the question, describe the analytical approach used to answer it, and present the results. Part I (RQ1–RQ3) addresses the multiple-choice benchmarks; Part II (RQ4–RQ6) addresses execution-grounded code generation.

4.1 RQ1: Pipeline vs. Monolithic

4.1.1 Motivation

Before investing in multi-agent architectures, practitioners need evidence that decomposing inference into specialized roles yields measurable gains over a single model operating end-to-end. Furthermore, simply chaining models is insufficient if intermediate artifacts are passed without structure; errors may compound silently across stages. RQ1 therefore asks two complementary questions: (i) do sequential pipelines improve over monolithic baselines, and (ii) does a structured, accountable handoff protocol matter?

4.1.2 Approach

We evaluate three inference regimes. First, each of the three models (GPT-4o, Claude Sonnet 4, Gemini 2.5 Pro) is run as a monolithic baseline, answering each benchmark item end-to-end. Second, we compose three-step pipelines from all eight ordered triples of models and run them in a *simple* mode where each stage passes its output to the next without any structured protocol. Third, we run the same configurations with the *accountable* handoff protocol described in Section 3.1, which instruments each stage with blame tracking and structured artifact formatting. We compare the final accuracy of all three regimes, alongside median latency and cost per prompt.

4.1.3 Results

Table 5: Baseline performance of single models across all datasets.

Dataset	Model	Accuracy (%)	Median Latency (s)	Median Cost/Prompt (USD)
AgiEval	Gemini 2.5 Pro (C)	92.40	11.84	$0.0112
	Claude Sonnet 4 (B)	58.20	4.21	$0.0036
	GPT-4o (A)	43.70	0.74	$0.0006
PythonIO	Gemini 2.5 Pro (C)	99.21	11.20	$0.0120
	Claude Sonnet 4 (B)	53.54	5.48	$0.0054
	GPT-4o (A)	69.29	0.71	$0.0009
LogiQA	Gemini 2.5 Pro (C)	84.45	14.78	$0.0134
	Claude Sonnet 4 (B)	79.51	6.74	$0.0051
	GPT-4o (A)	67.49	0.77	$0.0009

Monolithic baselines.

Table 5 reports the accuracy, median latency, and median cost per prompt for each model acting alone.

Gemini 2.5 Pro leads on all three benchmarks, reaching 99.21% on PythonIO and 92.40% on AgiEval. The gap between the strongest and weakest model is substantial:

nearly 49% on AgiEval and 46% on PythonIO. These baselines establish the ceiling that any pipeline must approach or exceed to justify its added complexity.

Simple vs. accountable pipelines.

Table 6 compares the final accuracy of simple (unmonitored) and accountable pipelines across all eight configurations.

Table 6: Accuracy (%) of simple (S) vs. accountable (A) pipelines and accuracy delta (Δ) across all datasets and configurations.

	AgiEval			**PythonIO**			**LogiQA**		
Config	**S**	**A**	Δ	**S**	**A**	Δ	**S**	**A**	Δ
AAA	57.41	77.19	+19.78	73.23	88.98	+15.75	71.48	71.38	−0.10
ABC	81.37	90.87	+9.50	99.21	99.21	0.00	79.58	80.57	+0.99
BAB	88.97	86.69	−2.28	63.78	96.85	+33.07	78.52	79.86	+1.34
BBB	88.97	89.73	+0.76	61.42	97.64	+36.22	79.58	79.51	−0.07
BCC	91.25	90.49	−0.76	99.21	98.43	−0.78	82.75	83.39	+0.64
CBA	87.07	92.40	+5.33	71.65	99.21	+27.56	81.34	83.75	+2.41
CCA	91.25	92.40	+1.15	71.65	98.43	+26.78	78.52	82.28	+3.76
CCC	91.63	93.54	+1.91	99.21	99.21	0.00	81.34	85.16	+3.82

The accountable protocol produces consistent gains on PythonIO, where the BBB configuration jumps from 61.42% to 97.64% and BAB from 63.78% to 96.85%. On AgiEval, the largest gain is AAA (from 57.41% to 77.19%), followed by ABC (from 81.37% to 90.87%). On LogiQA, gains are positive but more modest, with CCC improving from 81.34% to 85.16% and CBA from 81.34% to 83.75%. Two configurations (BAB on AgiEval, BCC on PythonIO) show small decreases, indicating that the protocol is not universally beneficial but that regressions are minor compared to the gains.

Figure 3 summarizes the key comparison: the AAA (Simple) pipeline consistently underperforms even the weakest monolithic baseline, demonstrating that unstructured chaining is actively harmful. The AAA (Accountable) pipeline recovers dramatically on AgiEval and PythonIO, showing that the protocol raises the performance floor. The strongest accountable pipelines (CCC, CBA) match or slightly surpass the best monolithic model on all three benchmarks.

Findings for RQ1

F1. Unstructured pipelines degrade accuracy below monolithic baselines on the majority of configurations across AgiEval and PythonIO.
F2. Structured accountability recovers accuracy dramatically over unstructured pipelines (e.g., from 61.42% to 97.64% on PythonIO), with the best accountable configurations matching or exceeding the strongest monolithic model on all three benchmarks.

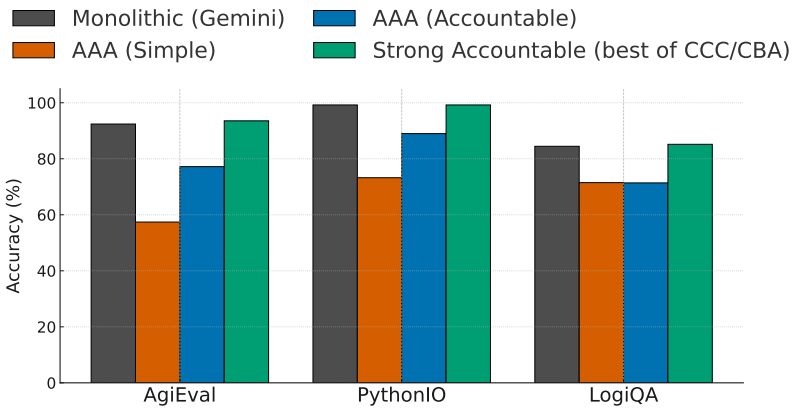


Fig. 3: Impact of pipeline architecture on accuracy. Monolithic baselines (best single model) are compared against the weakest and strongest pipeline configurations under both simple and accountable protocols.

4.2 RQ2: Role-Specific Error Dynamics

4.2.1 Motivation

RQ1 established that accountable pipelines can match or exceed monolithic baselines, but it treated the pipeline as a black box. For practitioners who need to decide which model to assign to which role, a deeper question arises: where do errors originate, how do they propagate, and which roles are most effective at correcting them? Understanding these dynamics is essential for informed model-role casting and for identifying the weakest link in a given configuration.

4.2.2 Approach

We use the blame attribution methodology defined in Algorithm 1 to decompose every pipeline run into per-stage contributions. For each task, we record whether the Planner produced a correct or incorrect initial answer, whether the Executor repaired or harmed the Planner's output, and whether the Critic repaired or harmed the Executor's output. We aggregate these flags across all configurations and benchmarks to compute three sets of metrics: (i) Planner error rates per model per benchmark, (ii) Executor and Critic repair and harm rates per model (aggregated across benchmarks), and (iii) error origin distributions that trace each final incorrect answer to its earliest unrepaired source.

4.2.3 Results

Planner error rates.

Table 7 reports the error rate for each model when assigned the Planner role. The "Total Cases" column reflects the number of times each model served as Planner across

all relevant configurations for a given dataset (e.g., GPT-4o appears as Planner in two configurations, so its total for AgiEval is $2 \times 263 = 526$).

Table 7: Planner error rate by model and dataset.

Planner Model	Dataset	Total Cases	Errors Introduced	Error Rate (%)
Gemini 2.5 Pro (C)	AgiEval	789	58	7.35
	PythonIO	381	3	0.79
	LogiQA	849	129	15.19
Claude Sonnet 4 (B)	AgiEval	789	106	13.43
	PythonIO	381	15	3.94
	LogiQA	849	187	22.03
GPT-4o (A)	AgiEval	526	213	40.49
	PythonIO	254	28	11.02
	LogiQA	566	162	28.62

A consistent hierarchy emerges: Gemini is the most reliable Planner (0.79% to 15.19%), followed by Claude (3.94% to 22.03%), with GPT-4o being the least reliable (11.02% to 40.49%). Error rates are also task-dependent. On the well-structured PythonIO tasks, all models perform well as Planners, with Gemini's error rate below 1%. On the more abstract reasoning tasks (LogiQA, AgiEval), error rates climb substantially, reaching 40.49% for GPT-4o on AgiEval. This pattern suggests that the Planner's reliability is sensitive to specification clarity: well-defined tasks produce fewer planning errors.

Repair and harm rates.

Table 8 reports the repair and harm rates for the Executor and Critic roles, aggregated across all configurations and benchmarks.

Each model exhibits a distinct behavioral profile. Claude Sonnet 4 is the most effective Executor, with the highest repair rate (10.01%) and a minimal harm rate (0.25%), meaning it frequently corrects Planner errors without introducing new ones. GPT-4o is the strongest Critic, achieving the highest final-stage repair rate (5.20%) with a moderate harm rate (0.89%). Gemini 2.5 Pro is the safest agent overall, with the lowest harm rates in both roles (0.25%), but its repair rates are also the lowest, indicating a conservative behavior that preserves correctness at the expense of error recovery.

These profiles are complementary. Gemini minimizes upstream errors as Planner, Claude recovers from the remaining errors as Executor, and GPT-4o provides a high-reward final correction as Critic. This complementarity yields a data-driven casting recommendation: Gemini (Planner) $\rightarrow$ Claude (Executor) $\rightarrow$ GPT-4o (Critic), which directly informs the heterogeneous model assignment used in Part II.

Table 8: Role-specific repair and harm rates by model (aggregated across all benchmarks).

Model	Role	Repair Rate (%)	Harm Rate (%)
Gemini 2.5 Pro (C)	Executor	1.27	0.25
	Critic	2.66	0.25
Claude Sonnet 4 (B)	Executor	10.01	0.25
	Critic	3.04	1.90
GPT-4o (A)	Executor	2.28	1.33
	Critic	5.20	0.89

Findings for RQ2

F3. The Planner is the highest-leverage role: its error rate ranges from 0.79% to 40.49% and is the strongest predictor of pipeline failure.
F4. Models exhibit complementary role-specific aptitudes: Claude is the best Executor (10.01% repair, 0.25% harm), GPT-4o is the strongest Critic (5.20% repair), and Gemini is the safest overall (0.25% harm in both roles).
F5. These complementary profiles yield a data-driven casting strategy adopted in Part II: Gemini (Planner) → Claude (Executor) → GPT-4o (Critic).

4.3 RQ3: Accuracy-Cost-Latency Trade-offs

4.3.1 Motivation

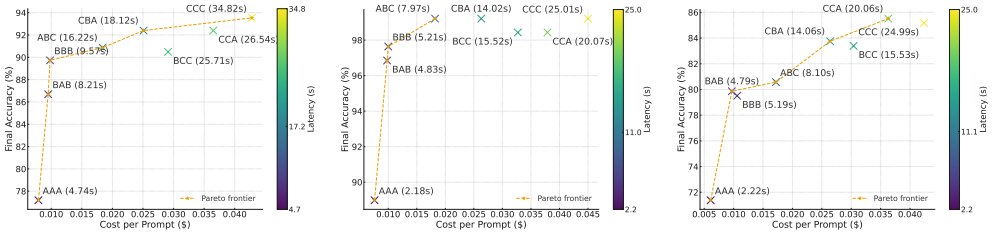
Maximizing accuracy is rarely the sole objective in production deployments. Operational budgets and response-time requirements constrain which pipeline configurations are feasible. RQ3 asks whether there exist configurations that offer near-optimal accuracy at substantially lower cost or latency, and whether the optimal trade-off point varies across task types.

4.3.2 Approach

For each of the eight accountable pipeline configurations, we compute the total cost per task (summing across all three roles using the per-token rates in Table 2) and the median wall-clock latency. We plot accuracy against cost for each benchmark, with latency encoded as point color, and identify the Pareto frontier: the set of configurations for which no other configuration achieves higher accuracy at equal or lower cost.

4.3.3 Results

Figure 4 visualizes the trade-off space across all three benchmarks.



AgiEval PythonIO LogiQA

Fig. 4: Accuracy vs. cost with latency color-coded; Pareto frontier shown (dashed).

The optimal trade-off is task-dependent. On AgiEval, the CCC configuration achieves the highest accuracy (93.54%) but is also the most expensive. CBA and CCA sit on the Pareto frontier, offering near-peak accuracy (92.40%) at significantly lower cost, making them practical balanced choices. On PythonIO, several configurations (ABC, CBA, CCC) reach 99.21% accuracy, so the discriminating factor becomes cost: ABC is the most efficient of this group, while CCC is substantially more expensive for no additional accuracy gain. On LogiQA, the Pareto frontier shows a direct accuracy-cost relationship, with CCA providing the highest accuracy (85.51%) and CBA offering a slightly lower accuracy (83.75%) at meaningfully lower cost.

Across all benchmarks, heterogeneous configurations (CBA, CCA, ABC) consistently occupy the Pareto frontier, while homogeneous configurations tend to be dominated. This reinforces the RQ2 finding that models have complementary strengths: pipelines that exploit role-specific aptitudes achieve better accuracy per dollar than those that use the same model throughout.

Findings for RQ3

F6. The optimal accuracy-cost-latency trade-off is task-dependent, with distinct Pareto frontiers across all three benchmarks.
F7. Heterogeneous pipelines consistently dominate the Pareto frontier, delivering higher accuracy per dollar than homogeneous configurations.

4.4 RQ4: Verifier Impact

4.4.1 Motivation

Part I's three-role pipeline had no structured recovery path when the Critic detected a failure: the Critic could flag an error, but the pipeline had no mechanism to act on this signal. RQ4 asks whether adding a non-generative Verifier agent that gates Critic verdicts and routes failures back to the Executor for re-generation improves pipeline correctness, and under what conditions it helps or harms.

4.4.2 Approach

We compare PEC (three-role pipeline without Verifier) against PECV (four-role pipeline with Verifier) across all three code generation benchmarks. In PECV, after

the Critic issues a verdict, the Verifier reads the Critic's evaluation summary and issues a binary ACCEPT or REJECT decision. On REJECT, the Verifier provides a targeted repair request routed back to the Executor for a single re-generation attempt. The marginal impact of the Verifier is measured as the accuracy delta between PEC and PECV (Δ PECV in Table 9). We additionally measure the Critic's false-fail and false-pass rates (Table 10) to explain the mechanistic cause of Verifier help and harm, and report Verifier invocation rates, help rates, and harm rates (Table 11).

4.4.3 Results

Pass@1 across configurations.

Table 9 presents the pass@1 accuracy for all five configurations, together with the Verifier-attributable deltas.

Table 9: Pass@1 accuracy (%) across all configurations (Part II). Δ PECV and Δ PEGV are measured against PEC. Best result per benchmark in bold.

Benchmark	Mono	EC	PEC	PECV	PEGV	Δ PECV	Δ PEGV
BigCodeBench	56.50	54.00	53.50	63.50	**71.50**	+10.00	+18.00
HumanEval	**96.95**	95.73	96.34	64.02	96.34	−32.32	+0.00
MBPP	80.50	79.00	75.00	**86.00**	81.00	+11.00	+6.00

The Verifier's impact when paired with the LLM Critic (PECV) is task-complexity conditional: PECV improves accuracy on BigCodeBench (from 53.50% to 63.50%) and MBPP (from 75.00% to 86.00%), but drops HumanEval from 96.34% to 64.02%, well below the Monolithic baseline. PEGV recovers this regression and achieves the highest accuracy on BigCodeBench.

Critic calibration.

Table 10 presents the Critic's false-fail and false-pass rates, which explain the mechanistic cause of Verifier help and harm.

Table 10: Critic calibration metrics per benchmark (Part II).

Benchmark	False-Fail (%)	False-Pass (%)
BigCodeBench	36.72	20.83
MBPP	59.13	3.53
HumanEval	93.00	0.00

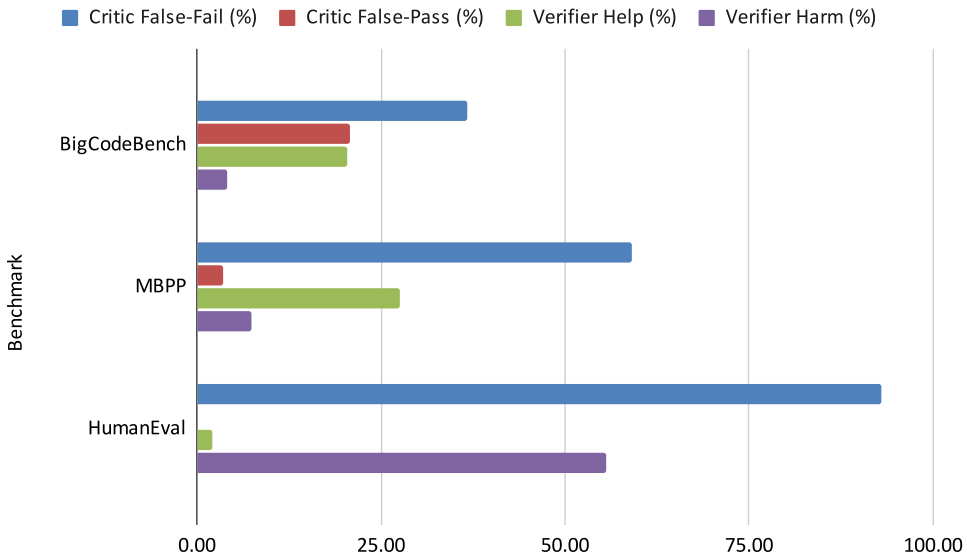


Fig. 5: Critic false-fail rate versus Verifier help and harm rates per benchmark. Higher false-fail rates predict higher Verifier harm, explaining the sign flip in PECV's impact across benchmarks.

Figure 5 shows the Critic's false-fail rate alongside the Verifier's help and harm rates, revealing a clear mechanistic relationship. On HumanEval, the 93.00% false-fail rate triggers re-generation of correct solutions, producing a 55.67% Verifier harm rate. On MBPP (59.13% false-fail), the help rate (27.52%) still exceeds the harm rate (7.34%), yielding a net positive impact. On BigCodeBench (36.72% false-fail), help dominates harm, yet even here PEGV outperforms PECV (71.50% vs. 63.50%).

Re-generation loop statistics.

Table 11 reports the Verifier invocation and overhead metrics for PECV.

Table 11: Re-generation loop statistics and pipeline overhead for the PECV configuration (Part II).

Benchmark	V. Inv. (%)	Repair (%)	Help (%)	Harm (%)	Lat. (s)	Tokens
BigCodeBench	64.00	61.50	20.33	4.07	60.23 s	7,118
HumanEval	60.98	59.15	2.06	55.67	45.20 s	6,164
MBPP	57.50	54.50	27.52	7.34	34.15 s	4,613

The Verifier invocation rate ranges from 57.50% (MBPP) to 64.00% (BigCodeBench), indicating that the Critic issues FAIL on a majority of tasks across all

benchmarks. This high invocation rate is itself a signal of Critic over-rejection, particularly on HumanEval where 93% of flagged solutions are actually correct. The Critic's false-fail rate predicts both the direction and magnitude of Verifier impact in PECV.

Findings for RQ4

F8. Adding the Verifier improves accuracy on complex tasks (BigCodeBench: from 53.50% to 63.50%) and medium tasks (MBPP: from 75.00% to 86.00%), but degrades near-saturated benchmarks (HumanEval: from 96.34% to 64.02%).
F9. LLM-based test evaluation is unreliable: the Critic's false-fail rate reaches 93% on HumanEval, triggering unnecessary re-generation that overwrites correct solutions with Verifier harm rates up to 55.67%.

4.5 RQ5: Planner Value in Code Generation

4.5.1 Motivation

Part I established Planner primacy as a design principle: the Planner error rate was the strongest predictor of pipeline failure (F3). However, in code generation the Planner's contribution is qualitatively different: rather than selecting a correct answer from options, the Planner produces a natural-language specification that the Executor must interpret and implement. RQ5 asks whether the Planner yields positive return on investment in code generation tasks.

4.5.2 Approach

We compare EC (Executor + Critic, no Planner) against PEC (Planner + Executor + Critic) across all three benchmarks. This comparison directly ablates the Planner while holding the Executor and Critic constant. We measure the accuracy delta, the Planner's latency overhead, and token overhead at both the pipeline level (Table 12) and the Executor level (Table 13). The Executor-level comparison isolates where the overhead originates by measuring the Executor's own input and output tokens with and without the Planner's specification.

4.5.3 Results

Planner ROI.

Table 12 presents the pass@1 comparison between EC and PEC, together with latency and token overhead.

The Planner's return on investment is negative or near-zero on all three benchmarks. On BigCodeBench, adding the Planner changes accuracy from 54.00% to 53.50% at 1.8× more tokens and 20.60 seconds of additional latency. On HumanEval, the Planner raises accuracy marginally from 95.73% to 96.34% (approximately one additional correct task out of 164) at 2.3× the token cost. On MBPP, adding the Planner drops accuracy from 79.00% to 75.00% at 3.0× the token cost.

Table 12: Planner cost: adding the Planner increases token consumption and latency with no accuracy benefit (Part II).

	BigCodeBench	**HumanEval**	**MBPP**
Accuracy			
Without Planner (EC)	54.00%	95.73%	79.00%
With Planner (PEC)	53.50%	96.34%	75.00%
Δ	−0.50	+0.61	−4.00
Median total tokens			
Without Planner (EC)	2,822	1,536	969
With Planner (PEC)	4,996	3,485	2,951
Overhead	1.8×	2.3×	3.0×
Planner latency added (avg)			
	20.60 s	14.27 s	16.50 s

Source of overhead.

Table 13 isolates the Executor's own token consumption to show where the Planner overhead originates.

Table 13: Executor token consumption with and without the Planner. The Planner's specification inflates the Executor's input tokens while output tokens remain similar (Part II).

	BigCodeBench	**HumanEval**	**MBPP**
Executor input tokens (median)			
Without Planner (EC)	753	407	216
With Planner (PEC)	865	651	621
Increase	+15 %	+60 %	+187 %
Executor output tokens (median)			
Without Planner (EC)	479	226	111
With Planner (PEC)	526	251	101
Change	+10 %	+11 %	−9 %

The Executor's input tokens increase substantially (+15% to +187%) because it receives the Planner's specification on top of the problem statement. However, the Executor's output tokens remain essentially unchanged (+10%, +11%, −9%), confirming that the Planner's specification does not produce longer or more detailed code. The overhead is specification passthrough, not improved code generation. For well-defined tasks (HumanEval, MBPP), the problem statement already provides a function

signature, docstring, and example inputs/outputs, making the Planner's specification redundant. For complex tasks (BigCodeBench), the Executor appears sufficiently capable of performing implicit planning during code generation.

Findings for RQ5

F10. Adding the Planner yields no meaningful accuracy gain on code generation: accuracy changes from 54.00% to 53.50% on BigCodeBench, from 95.73% to 96.34% on HumanEval, and from 79.00% to 75.00% on MBPP.
F11. The Planner adds 1.8–3.0× token overhead and 14–21 seconds of latency per task, with the overhead originating from specification passthrough rather than improved code output.
F12. The Planner's contribution is task-type dependent, qualifying Part I's finding of Planner primacy (F3): strong Executors produce comparable output with or without the Planner's formal specification on well-defined tasks.

4.6 RQ6: Execution-Based vs. LLM Critic

4.6.1 Motivation

RQ4 identified the Critic's false-fail rate as the mechanistic cause of Verifier-induced accuracy regressions: on HumanEval, the 93.00% false-fail rate triggered unnecessary re-generation of correct solutions, dropping accuracy from 96.34% to 64.02% (F9). Even on BigCodeBench, where the Critic was best calibrated (36.72% false-fail), PECV left room for improvement. In code generation, an execution oracle provides ground-truth correctness independently of any LLM. RQ6 asks: what happens when we remove the Critic entirely and let the execution oracle drive the Verifier?

4.6.2 Approach

We compare PECV (Critic-mediated verification) against PEGV (execution-gated verification) across all three benchmarks. In PEGV, after the Executor produces code, the test harness executes it directly. If all tests pass, the pipeline terminates. If any test fails, execution results go directly to the Verifier, which issues ACCEPT or REJECT and routes REJECT decisions back to the Executor for re-generation, in an iterative loop of up to three iterations ($k \leq 3$). The LLM Critic is removed entirely. This comparison directly ablates the Critic while holding the Planner, Executor, and Verifier constant.

4.6.3 Results

PECV vs. PEGV accuracy.

Table 14 presents the side-by-side comparison.

Figure 6 presents the accuracy and harm comparison side by side. Execution gating produces the highest accuracy on BigCodeBench (71.50%, up from 56.50% Monolithic), recovers the HumanEval regression entirely (96.34% vs. 64.02% with PECV), and trails PECV only on MBPP (81.00% vs. 86.00%). On HumanEval, the Verifier

Table 14: Critic-mediated (PECV) vs. execution-gated (PEGV) verification accuracy (%). Best per benchmark in bold.

Benchmark	Monolithic	PEC	PECV	PEGV	PECV vs. PEGV
BigCodeBench	56.50	53.50	63.50	**71.50**	+8.00 (PEGV)
HumanEval	96.95	96.34	64.02	**96.34**	+32.32 (PEGV)
MBPP	80.50	75.00	**86.00**	81.00	−5.00 (PECV)

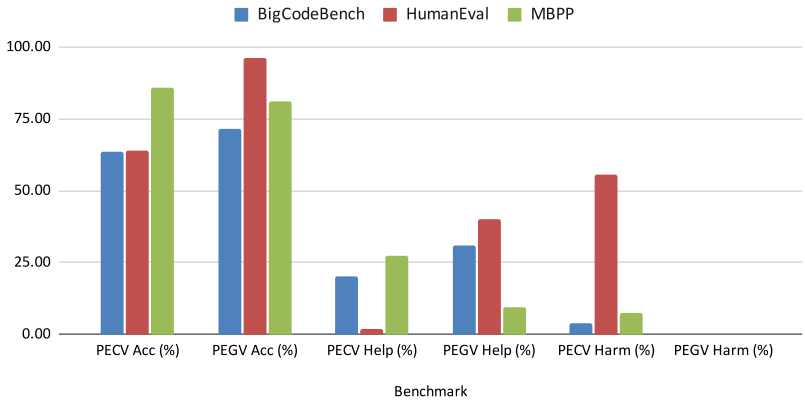


Fig. 6: PECV vs. PEGV: accuracy and Verifier harm rate per benchmark. Execution gating eliminates all Verifier harm (0.00%) while matching or exceeding PECV accuracy on two of three benchmarks.

is invoked on only 6.10% of tasks in PEGV versus 60.98% in PECV, because execution gating triggers the Verifier only when code genuinely fails. On MBPP, PECV's advantage (86.00% vs. 81.00%) suggests the Critic's failure localization provides richer context than bare execution output.

Verifier harm elimination.

Table 15 provides the detailed harm rate comparison.

Table 15: Verifier harm rate comparison: PECV vs. PEGV across benchmarks.

Benchmark	PECV Harm (%)	PEGV Harm (%)
BigCodeBench	4.07	**0.00**
HumanEval	55.67	**0.00**
MBPP	7.34	**0.00**

The Verifier harm rate is 0.00% across all three benchmarks in PEGV. In PECV, Verifier harm ranges from 4.07% (BigCodeBench) to 55.67% (HumanEval). Every point of Verifier harm in PECV is directly attributable to the Critic's false-fail verdict triggering re-generation of correct code.

Cost efficiency.

Table 16 compares latency and token overhead between the two configurations.

Table 16: Latency and token overhead comparison: PECV vs. PEGV.

Benchmark	PECV Lat. (s)	PEGV Lat. (s)	PECV Tokens	PEGV Tokens
BigCodeBench	60.23 s	54.92 s	7,118	3,851
HumanEval	45.20 s	25.30 s	6,164	2,625
MBPP	34.15 s	36.08 s	4,613	2,356

PEGV uses 46–57% fewer tokens across all benchmarks. The savings come from two sources: PEGV eliminates the Critic's LLM call entirely, and PEGV invokes the Verifier far less frequently (e.g., 6.10% vs. 60.98% on HumanEval) because it only fires on genuinely failing code.

Findings for RQ6

F13. Replacing the LLM Critic with execution gating eliminates Verifier harm entirely (0.00% across all benchmarks) and raises BigCodeBench accuracy from 56.50% (Monolithic) to 71.50%.
F14. Execution gating recovers the accuracy loss caused by Critic-mediated verification on HumanEval (from 64.02% back to 96.34%) while reducing token cost by up to 57%.
F15. The LLM Critic adds marginal value only on MBPP (86.00% with PECV vs. 81.00% with PEGV), where its failure localization provides richer context to the Verifier than bare execution output.

5 Discussion and Practical Guidance

The two parts of this study jointly yield actionable design rules for multi-agent LLM pipelines, each grounded in specific empirical findings (F1–F15).

5.1 Design Rules for Practitioners

Rule 1: Always use structured, accountable handoffs. The accuracy lift over unstructured pipelines is consistent across both parts and all task types. In Part I, accountable pipelines recovered accuracy from 61.42% to 97.64% on PythonIO (F2). In Part II, the PEGV pipeline raised BigCodeBench accuracy from 56.50% (Monolithic) to 71.50%. Structured accountability is the most transferable finding of this work.

Rule 2: Include the Planner only when task specification is ambiguous or underspecified. Part I established Planner primacy on multiple-choice tasks (F3), but Part II showed that adding the Planner yields no meaningful accuracy gain on code generation (e.g., from 54.00% to 53.50% on BigCodeBench, from 79.00% to 75.00% on MBPP; F10). For well-defined, short-horizon tasks, omitting the Planner saves 14 to 21 seconds of latency per task with no accuracy cost (F11). The Planner's value is task-type dependent (F12).

Rule 3: Use execution gating as the default verification strategy. When a test harness or execution oracle is available, the Verifier should be driven directly from execution results rather than from an LLM Critic's verdict. Execution gating eliminates Verifier harm entirely (0.00% across all benchmarks, F13), achieves the highest accuracy on complex tasks (71.50% on BigCodeBench), and reduces token cost by 46–57% compared to Critic-mediated verification (F14). The LLM Critic should be added only when its calibration on the target task distribution has been validated to provide net benefit, which in this study occurred only on MBPP (F15).

Rule 4: When an LLM Critic is necessary, measure its false-fail rate before deploying a Verifier. For tasks without execution oracles (e.g., UI generation, documentation, requirements analysis), the LLM Critic remains the only available evaluation signal. In these settings, the Critic false-fail rate is the binding constraint on Verifier utility: a false-fail rate above approximately 60% makes Verifier-based regeneration net harmful (F9). A practical pre-deployment step is to evaluate the Critic's false-fail rate on a representative sample and recalibrate or omit the Verifier if the rate is too high.

Rule 5: Cast roles by empirically established strengths. The role-specific aptitudes observed in Part I (F4, F5) informed Part II's heterogeneous model assignment, which produced the best results wherever the pipeline adds value. Heterogeneous pipelines consistently dominate homogeneous configurations on the Pareto frontier (F7). Running a small profiling benchmark to measure model-role fit before committing to a pipeline design should be standard practice.

5.2 Threats to Validity

Construct validity is substantially improved from Part I to Part II. Part I relies on multiple-choice benchmarks that simplify open-ended tasks into discrete answers, making blame attribution feasible but limiting ecological validity. Part II replaces label matching with execution-based evaluation on genuine code generation tasks. The remaining concern is scope: the three code generation benchmarks cover single-function to moderate-complexity problems but do not include multi-file, repository-level tasks. Additionally, HumanEval's likely presence in frontier model training data means that absolute accuracy figures on this benchmark reflect model familiarity as well as capability, though the internal pipeline comparisons (e.g., PECV vs. PEGV) remain valid because all configurations use the same models. This contamination also likely inflates the Critic's 93% false-fail rate on HumanEval: the Executor produces memorized, correct solutions that the Critic cannot recognize as correct through static reasoning alone. We mitigate this concern by evaluating across three benchmarks of varying contamination risk and by demonstrating that the execution-based gating

strategy (PEGV) resolves the issue entirely, further supporting the case that execution oracles are essential for reliable verification.

Internal validity is substantially improved. Part I's blame attribution required ground-truth labels that are unavailable at runtime; Part II replaces this with an execution oracle that provides ground-truth correctness independently. The PEGV ablation provides a clean comparison of Critic-mediated versus execution-gated verification, isolating the Critic's contribution. All model invocations use temperature 0.0 (greedy decoding), making pipeline runs fully deterministic and eliminating stochastic variance as a confound. Three remaining concerns are: (a) the PEGV configuration includes a Planner, so the PECV-versus-PEGV comparison conflates Critic removal with potential Planner interaction effects, though the RQ5 analysis (F10–F12) shows the Planner's impact is minimal, suggesting this confound is negligible; (b) the iterative re-generation loop in PEGV ($k \leq 3$) versus the single-pass Verifier in PECV ($k \leq 1$) introduces a structural asymmetry, though this reflects the intended design difference between the two configurations; and (c) at the sample sizes used in this study (164–200 tasks per benchmark), small accuracy differences (below approximately 3–4%) may not be statistically distinguishable from noise and should be interpreted cautiously. The paper's core findings rest on large-magnitude effects (e.g., accountability raising accuracy from 61.42% to 97.64%, HumanEval dropping from 96.34% to 64.02%, 93% false-fail rate, 0% PEGV harm) that are well above this threshold.

External validity remains a concern. Both parts use proprietary, frontier LLMs during fixed evaluation windows. The pipeline is evaluated exclusively on Python benchmarks. The model selection for Part II uses successor models (Claude Sonnet 4.5, GPT-5.1, Gemini 2.5 Pro Preview) rather than the exact models evaluated in Part I (Claude Sonnet 4, GPT-4o, Gemini 2.5 Pro), introducing a confound between model improvement and role-specific dynamics. We mitigate this by selecting successor models from the same families and by grounding the role assignments in Part I's empirical findings. While the data-driven casting methodology transfers across model generations, the specific optimal role assignments may differ and should be re-profiled when models are updated. All prompts, execution logs, and evaluation scripts are released to support replication and encourage extension to open-sourc

6 Conclusion

This paper presents a two-part empirical investigation that opens the black box of sequential multi-agent LLM pipelines through traceable blame attribution.

Part I evaluates a three-role Planner-Executor-Critic pipeline on three multiple-choice benchmarks (PythonIO, LogiQA, AGIEval) using label-matching blame attribution. The results show that structured, accountable handoffs are essential: unstructured pipelines degrade accuracy below monolithic baselines, while accountable pipelines recover accuracy dramatically (e.g., from 61.42% to 97.64% on PythonIO; F1, F2). Blame attribution reveals that the Planner is the highest-leverage role (F3), that models exhibit complementary role-specific aptitudes (F4), and that heterogeneous configurations dominate the Pareto frontier in accuracy-cost-latency trade-offs (F6,

F7). These findings yield a data-driven casting recommendation, Gemini (Planner) → Claude (Executor) → GPT-4o (Critic), that is adopted in Part II (F5).

Part II extends the methodology to execution-grounded code generation on HumanEval, MBPP, and BigCodeBench, introducing a fourth Verifier agent and replacing label matching with runtime test-suite execution. Three findings reshape the design landscape. First, the Verifier's benefit is conditional on Critic calibration: it improves accuracy on complex tasks (BigCodeBench: from 53.50% to 63.50%; MBPP: from 75.00% to 86.00%) but causes substantial accuracy degradation on near-saturated benchmarks (HumanEval: from 96.34% to 64.02%), driven mechanistically by the Critic's false-fail rate (F8, F9). Second, the Planner's primacy does not transfer to code generation: its ROI is negative or near-zero on all three benchmarks, with overhead originating from specification passthrough rather than improved code output (F10–F12). Third, replacing the LLM Critic with direct execution gating eliminates Verifier harm entirely (0.00% across all benchmarks), achieves the highest accuracy on complex tasks (71.50% on BigCodeBench), and reduces token cost by 46–57% (F13–F15).

Together, the two parts establish that multi-agent pipeline design is not a monolithic practice but a task-sensitive engineering discipline. The study contributes traceable blame attribution methodologies for both label-matching and execution-grounded settings, 15 empirical findings (F1–F15) across six research questions, and five actionable design rules for practitioners building accountable multi-agent systems.

Acknowledgments

This study is partially supported by our industrial collaborator *zuvu.ai*, which provided LLM usage resources for this work.

Author Contributions

J.B. conducted the experiments, collected the results, and prepared the figures and tables. A.B. designed the study, contributed to the experiments, and led the supervision. E.K. contributed to supervision and study design. All authors wrote the main manuscript text and reviewed the manuscript.

Data Availability

To facilitate reproducibility and further research, the replication materials are publicly available. The results corresponding to the first 3 RQs (RQ1, RQ2, and RQ3) can be found at [49]. The results for the last 3 RQs (RQ4, RQ5, and RQ6) are available through the replication package [50].

Declarations

Competing interests The authors declare no competing interests.

References

[1] He, J., Treude, C., Lo, D.: Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. ACM Transactions on Software Engineering and Methodology **34**(5), 1–30 (2025)

[2] Jin, H., Huang, L., Cai, H., Yan, J., Li, B., Chen, H.: From llms to llm-based agents for software engineering: A survey of current, challenges and future. arXiv preprint arXiv:2408.02479 (2024)

[3] Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., Chen, Z., Tang, J., Chen, X., Lin, Y., *et al.*: A survey on large language model based autonomous agents. Frontiers of Computer Science **18**(6), 186345 (2024)

[4] Qian, C., Cong, X., Yang, C., Chen, W., Su, Y., Xu, J., Liu, Z., Sun, M.: Communicative agents for software development. arXiv preprint arXiv:2307.07924 **6**(3), 1 (2023)

[5] Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., Wang, J., Wang, Z., Yau, S.K.S., Lin, Z., *et al.*: Metagpt: Meta programming for a multi-agent collaborative framework. (2024). International Conference on Learning Representations, ICLR

[6] Lee, C., Xia, C.S., Yang, L., Huang, J.-t., Zhu, Z., Zhang, L., Lyu, M.R.: A unified debugging approach via llm-based multi-agent synergy. arXiv preprint arXiv:2404.17153 (2024)

[7] Epperson, W., Bansal, G., Dibia, V., Fourney, A., Gerrits, J., Zhu, E., Amershi, S.: Interactive debugging and steering of multi-agent ai systems. In: Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems. CHI '25 (2025)

[8] Yang, A.Z., Le Goues, C., Martins, R., Hellendoorn, V.: Large language models for test-free fault localization. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, pp. 1–12 (2024)

[9] Ronanki, K.: Facilitating trustworthy human-agent collaboration in llm-based multi-agent system oriented software engineering. In: Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, pp. 1333–1337 (2025)

[10] Pan, M.Z., Cemri, M., Agrawal, L.A., Yang, S., Chopra, B., Tiwari, R., Keutzer, K., Parameswaran, A., Ramchandran, K., Klein, D., *et al.*: Why do multiagent systems fail? In: ICLR 2025 Workshop on Building Trust in Language Models and Applications (2025)

[11] Cleland-Huang, J., Gotel, O.C., Huffman Hayes, J., Mäder, P., Zisman, A.: Software traceability: trends and future directions. In: Future of Software Engineering Proceedings, pp. 55–69 (2014)

[12] Zhang, S., Yin, M., Zhang, J., Liu, J., Han, Z., Zhang, J., Li, B., Wang, C., Wang, H., Chen, Y., et al.: Which agent causes task failures and when? on automated failure attribution of llm multi-agent systems. arXiv preprint arXiv:2505.00212 (2025)

[13] Triantafyllou, S., Singla, A., Radanovic, G.: On blame attribution for accountable multi-agent sequential decision making. Advances in Neural Information Processing Systems **34**, 15774–15786 (2021)

[14] Barrak, A.: Traceability and accountability in role-specialized multi-agent llm pipelines. In: 2025 40th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), pp. 315–322 (2025). IEEE

[15] Ridnik, T., Kredo, D., Friedman, I.: Code generation with AlphaCodium: From prompt engineering to flow engineering. arXiv preprint arXiv:2401.08500 (2024)

[16] Le, H., Wang, Y., Gotmare, A.D., Savarese, S., Hoi, S.C.H.: CodeChain: Towards modular code generation through chain of self-revisions with representative submodules. In: The Twelfth International Conference on Learning Representations (2024)

[17] Snell, C., Lee, J., Xu, K., Kumar, A.: Scaling LLM test-time compute optimally can be more effective than scaling model parameters. arXiv preprint arXiv:2408.03314 (2024)

[18] Lifshitz, S., McIlraith, S.A., Du, Y.: Multi-agent verification: Scaling test-time compute with multiple verifiers (abridged). In: Workshop on Reasoning and Planning for Large Language Models

[19] Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E., et al.: Judging LLM-as-a-judge with MT-bench and Chatbot Arena. Advances in Neural Information Processing Systems **36** (2023)

[20] Chen, M., Tworek, J., Jun, H., Yuan, Q., Oliveira, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)

[21] Zhuo, T.Y., Vu, M.C., Chim, J., Hu, H., Yu, W., Widyasari, R., Yusuf, I.N.B., Zhan, H., He, J., Paul, I., et al.: BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. arXiv preprint arXiv:2406.15877 (2024)

[22] Liu, N.F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., Liang,

P.: Lost in the middle: How language models use long contexts. Transactions of the Association for Computational Linguistics **12**, 157–173 (2024) https://doi.org/10.1162/tacl_a_00638

[23] Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., *et al.*: Autogen: Enabling next-gen llm applications via multi-agent conversations. In: First Conference on Language Modeling (2024)

[24] Yang, J., Jimenez, C.E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., Press, O.: SWE-agent: Agent-computer interfaces enable automated software engineering. arXiv preprint arXiv:2405.15793 (2024)

[25] Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., Anandkumar, A.: Chain of agents: Large language models collaborating on long-context tasks. arXiv preprint arXiv:2404.18260 (2024)

[26] Mushtaq, A., Naeem, M.R., Ghaznavi, I., Taj, M.I., Hashmi, I., Qadir, J.: Harnessing multi-agent llms for complex engineering problem-solving: A framework for senior design projects. arXiv preprint arXiv:2501.01205 (2025)

[27] Chen, W., Su, Y., Zuo, J., Yang, C., Yuan, C., Qian, C., Chan, C.-M., Qin, Y., Lu, Y., Xie, R., *et al.*: Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. arXiv preprint arXiv:2308.10848 **2**(4), 6 (2023)

[28] Barrak, A., Fourati, Y., Olchawa, M., Ksontini, E., Zoghlami, K.: Cargo: A framework for confidence-aware routing of large language models. In: 2025 IEEE International Conference on Collaborative Advances in Software and COmputiNg (CASCON), pp. 80–89 (2025). IEEE

[29] Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., Zhou, D.: Self-consistency improves chain of thought reasoning in language models. In: The Eleventh International Conference on Learning Representations (2023)

[30] Dong, L., Lu, Q., Zhu, L.: Agentops: Enabling observability of llm agents. arXiv preprint arXiv:2411.05285 (2024)

[31] Zhang, Z., Jiang, Z., Xu, L., Hao, H., Wang, R.: Multiple-choice questions are efficient and robust llm evaluators. arXiv preprint arXiv:2405.11966 (2024)

[32] Liu, J., Cui, L., Liu, H., Huang, D., Wang, Y., Zhang, Y.: Logiqa: A challenge dataset for machine reading comprehension with logical reasoning. arXiv preprint arXiv:2007.08124 (2020)

[33] Zhong, W., Cui, R., Guo, Y., Liang, Y., Lu, S., Wang, Y., Saied, A., Chen, W., Duan, N.: Agieval: A human-centric benchmark for evaluating foundation models.

arXiv preprint arXiv:2304.06364 (2023)

[34] Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al.: Program synthesis with large language models. arXiv preprint arXiv:2108.07732 (2021)

[35] Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., *et al.*: Self-refine: Iterative refinement with self-feedback. Advances in Neural Information Processing Systems **36**, 46534–46594 (2023)

[36] Chen, X., Lin, M., Schaerli, N., Zhou, D.: Teaching large language models to self-debug. In: The 61st Annual Meeting Of The Association For Computational Linguistics (2023)

[37] Tufano, R., Pascarella, L., Tufano, M., Poshyvanyk, D., Bavota, G.: Towards automating code review activities. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 163–174 (2021). IEEE

[38] Shinn, N., Cassano, F., Berman, E., et al.: Reflexion: Language agents with verbal reinforcement learning. arXiv preprint arXiv:2303.11366 (2023)

[39] Xia, C.S., Zhang, L.: Conversational automated program repair. arXiv preprint arXiv:2305.14864 (2023)

[40] Huang, D., Bu, Q., Zhang, J.M., et al.: Agentcoder: Multi-agent based code generation with iterative testing and optimization. arXiv preprint arXiv:2312.13010 (2023)

[41] Xue, E., Huang, Z., Ji, Y., Wang, H.: Improve: Iterative model pipeline refinement and optimization leveraging llm agents. arXiv preprint arXiv:2502.18530 (2025)

[42] Bai, H., Kadavath, S., Kundu, S., Askell, A., Kernion, J., Jones, A., Chen, A., Goldie, A., Mirhoseini, A., McKinnon, C., et al.: Constitutional AI: Harmlessness from AI feedback. arXiv preprint arXiv:2212.08073 (2022)

[43] Hart, R.: What AI Is The Best? Chatbot Arena Relies On Millions Of Votes. Forbes coverage of Chatbot Arena overall leaders (GPT-4o listed first). https://www.forbes.com/sites/roberthart/2024/07/18/what-ai-is-the-best-chatbot-arena-relies-on-millions-of-human-votes/

[44] Sharma, S.: Anthropic's Claude 3.5 Sonnet Surges to Top of AI Rankings in LMSYS Chatbot Arena, Challenging Industry Giants. https://venturebeat.com/ai/anthropic-claude-3-5-sonnet-surges-to-top-of-ai-rankings

[45] Shakir, U.: Google Says Its New 'reasoning' Gemini AI Models Are the Best Ones Yet. Mentions Gemini 2.5 Pro leading LMArena with a 39 Elo point margin.

https://www.theverge.com/news/635502/google-gemini-2-5-reasoning-ai-model

[46] OpenAI: API Pricing. https://openai.com/api/pricing/ Accessed 2025-08-20

[47] Anthropic: Claude Sonnet 4 API Pricing. https://www.anthropic.com/news/1m-context Accessed 2025-08-20

[48] Google AI: Gemini Developer API Pricing. https://ai.google.dev/gemini-api/docs/pricing Accessed 2025-08-20

[49] Barrak, A.: Accountability and Role Specialization in Sequential LLM Agent Pipelines. https://doi.org/10.5281/zenodo.17216904 . https://doi.org/10.5281/zenodo.17216904

[50] JacksonBeem: SWE-Agentic-Pipeline: Replication Package - Who Broke the Pipeline? Traceability and Accountability in Multi-Agent LLM Software Engineering Pipelines. https://doi.org/10.5281/zenodo.19462824 . https://doi.org/10.5281/zenodo.19462824