# Verification-Aware Planning for Multi-Agent Systems

**Tianyang Xu***
Purdue University, USA
xu1868@purdue.edu

**Dan Zhang**
Megagon Labs, USA
dan_z@megagon.ai

**Kushan Mitra**
Megagon Labs, USA
kushan@megagon.ai

**Estevam Hruschka**
Megagon Labs, USA
estevam@megagon.ai

## Abstract

Large language model (LLM) agents are increasingly deployed to tackle complex tasks, often necessitating collaboration among multiple specialized agents. However, multi-agent collaboration introduces new challenges in planning, coordination, and verification. Execution failures frequently arise not from flawed reasoning alone, but from subtle misalignments in task interpretation, output format, or inter-agent handoffs. To address these challenges, we present VERIMAP[1], a framework for multi-agent collaboration with verification-aware planning. The VERIMAP planner decomposes tasks, models subtask dependencies, and encodes planner-defined passing criteria as subtask verification functions (VFs) in Python and natural language. We evaluate VERIMAP on diverse datasets, demonstrating that it outperforms both single- and multi-agent baselines while enhancing system robustness and interpretability. Our analysis highlights how verification-aware planning enables reliable coordination and iterative refinement in multi-agent systems, without relying on external labels or annotations.

## 1 Introduction

Large language model (LLM) agents are increasingly deployed to solve diverse and complex tasks. Beyond single-agent settings, multiple agents can be orchestrated to collaborate on a shared objective (Zhu et al., 2025; Cemri et al., 2025; Li et al., 2023), enabling workflows that combine the complementary strengths of different models and tools.

For example, a reasoning-oriented agent can interpret and plan around the overall goal, determine suitable task granularity and instructions, and coordinate subtasks across agents—delegating bounded actions to lightweight executors and leveraging specialized tools for retrieval or computation. This modularity creates opportunities for improved efficiency, interpretability, and control in AI systems, as practitioners can configure and combine models with tailored capabilities for planning, execution, and verification.

A central component of multi-agent collaboration is planning. Effective planning requires understanding the overall task, decomposing it into subtasks, modeling dependencies, and assigning responsibilities to agents with different capabilities. Often, plans are represented as directed acyclic graphs (DAGs) (Prasad et al., 2024; Sung et al., 2025; Erdogan et al., 2025), where the output of one agent becomes the input to another. This structure enables scalable problem-solving but introduces fragility: the workflow depends on smooth communication and accurate handoffs.

In multi-agent systems, agents fail in diverse ways. Verification is commonly used to monitor performance, typically checking factual accuracy, logical consistency, or final answer quality (Miao et al., 2024; Hao et al., 2025; Grigorev et al., 2025). However, failures can range from obvious to subtle: an agent may (i) produce no result, (ii) output in the wrong format, breaking downstream consumption, or (iii) produce results that appear reasonable but diverge from what the planner or downstream agents expect. For example, in a document summarization pipeline, a "section extractor" might return raw text instead of structured JSON, or misinterpret "summarize key claim" as "summarize all paragraphs", causing downstream failures. These latter cases show that **execution failure is context-dependent** — correctness alone is insufficient if outputs violate plan expectations. Existing verification rarely captures such interpretation or handoff failures (Stechly et al., 2024; Sung et al., 2025).

To address these limitations, we propose VERIMAP, a framework that integrates planning and verification in a contextualized manner. The planner decomposes tasks, models dependencies, and

*Work done during internship at Megagon Labs.
[1]Code available at https://github.com/megagonlabs/veriMAP



*Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics Volume 1: Long Papers*, pages 7528–7546

specifies requirements for each agent's output. These requirements are encoded as verification functions (VFs), which distill global context into local checks, allowing verifiers to operate independently while ensuring outputs satisfy both local and global expectations. Analogous to good project management, tasks are assigned with clear acceptance criteria, and handoffs follow well-defined "passing standards". With these explicit requirements, agents gain a precise understanding of what is expected, can self-refine when their outputs do not conform, and can offload global reasoning to the planner while focusing their efforts on localized subtasks. This allows smaller models to participate effectively, avoids redundant correction of upstream errors, and ultimately improves system robustness.

In this paper, we make the following contributions:

- We introduce VERIMAP, a system for multi-agent collaboration with verification-aware planning. The planner decomposes tasks, models dependencies, and specifies requirements for each agent's output using Structured I/O and per-subtask VFs in Python and natural language, which distill global context into local checks. This enables agents to self-refine, share context efficiently, focus on localized subtasks, and improve overall robustness.

- We evaluate VERIMAP on five datasets covering mathematics, coding, and question answering. It outperforms single- and multi-agent baselines, particularly on harder datasets (up to 4.05% on BigCodeBench-Hard and 9.8% on Olympiads, compared with the next-best tool-enabled ReAct agents), and our analysis of cost, VF characteristics, and a case study demonstrates how verification-aware planning improves robustness, and enables iterative refinement.

## 2 VERIMAP

We propose VERIMAP, a multi-agent LLM system integrating planning and verification. As illustrated in Figure 1, the system consists of four key modules: **Planner**, **Executor**, **Verifier**, and **Coordinator**.

### 2.1 Verification-Aware Planner

The Verification-Aware Planner lies at the core of VERIMAP, responsible for decomposing complex tasks into executable subtasks while simultaneously generating **Verification Functions** (VFs) for each subtask. Note that the planner generates VFs by itself and has no access to the unit tests provided by the benchmark, eliminating the possibility of VERIMAP "cheating" over the benchmarks. Like conventional multi-agent planners, it constructs a directed acyclic graph (DAG) in which nodes represent subtasks and edges capture model dependencies. Each node is assigned a unique identifier and contains the subtask instruction, its dependency edges, and metadata that regulates both context sharing and dependency resolution.

**Planning for Information Flow** In addition to planning subtask decomposition and dependencies, the VERIMAP planner also orchestrates message passing across agents. To ensure a precise and transparent flow of information, VERIMAP enforces: (i) Structured I/O, requiring that agent inputs and outputs adhere to well-defined formats, such as JSON objects; and (ii) Named Variables, requiring that each I/O object has a unique, consistent name across all nodes in the plan. Each node thus receives an input object consisting of `(name: value)` pairs, and the plan explicitly defines the expected outputs along with their variable names. This guarantees that downstream agents can reliably reference and consume upstream outputs, reducing execution ambiguity.

**Planner-Defined Subtask Verification** A second key contribution of the planner is the generation of **subtask-specific verification functions**. The planner's structured and consistently named I/O provides a rigorous foundation for verification, supporting the generation of functions aligned with each subtask's expected structure and semantics. For each node, the planner specifies precise criteria that determine whether an execution result is correct by generating verification functions (VFs) for that subtask. This design allows a verifier to focus on evaluating these planner-generated VFs, rather than reasoning about the overall task or other subtasks, effectively offloading much of the reasoning burden from the verifier to the planner.

For each subtask, the VERIMAP planner generates VFs of two categories based on the task requirements.

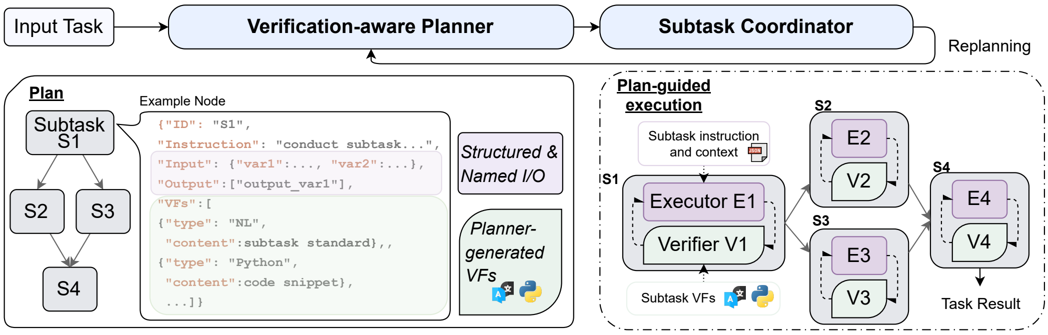



Figure 1: Overview of VeriMAP. VeriMAP integrates planning and verification by decomposing tasks into a DAG of subtasks, where the planner specifies Structured and Named I/O and Verification Functions (VFs) at subtask-level. Executors produce JSON outputs verified by paired VFs, while the coordinator manages contexts, retries, and replanning to ensure reliable final results.

- **Python VFs.** For tasks with well-defined functional or structural requirements, the planner generates self-contained Python assertions. These validate output type, format, and correctness against specified criteria. Being programmatic, they provide deterministic and reproducible guarantees.
- **Natural language VFs.** For tasks requiring semantic or open-ended judgments (e.g., summaries, natural language answers), the planner generates instructions in natural language to guide a verifier agent. These complement the strictness of Python tests with broader semantic coverage.

## 2.2 Executor

The **Executor** is responsible for solving its assigned subtask using planner-generated instructions and necessary context (output of upstream nodes) and then produces structured outputs that can be passed to Verifiers and downstream Executors. To limit scope and reduce reasoning complexity, Executors are not exposed to the original global task. We implement the Executor using LLMs with general-purpose agent paradigms, such as *ReAct*-style agents (Yao et al., 2023) and tool-calling agents, where planner-generated instructions are provided as task prompts. The Executor is capable of making lightweight decisions and invoking external tools as needed. While this work assumes general-purpose Executors, VeriMAP can be easily extended to incorporate specialized agents (e.g., an information extraction agent or an NL2SQL agent) if available. Since each Executor typically handles a simple, well-scoped subtask, it can rely on smaller and more cost-efficient models; in our experiments, we use `gpt-4o-mini`. Additional implementation details are provided in Appendix B.2.

## 2.3 Verifier

As discussed earlier, the plan specifies a set of predefined verification functions (VFs) that capture the expected behavior of each subtask executor. After execution, the Verifier evaluates all VFs against the structured output of the Executor. For natural language VFs, it issues LLM calls with `(executor_output, VF)` pairs; for Python-based VFs, it invokes a Python interpreter. The Verifier then aggregates the results from all VFs to make a final decision. In this paper, we adopt a strict logical AND strategy, where a subtask is marked as failed if any VF fails. In practice, softer aggregation methods could weight VFs differently according to their importance; we leave such extensions to future work. When a VF fails, the Verifier collects signals to guide retries: LLM-based VFs return explanations for the failure, while Python-based VFs provide error traces. Further implementation details are given in Appendix B.3.

## 2.4 Coordinator

The **Subtask Coordinator** serves as the central orchestrator of multi-agent task execution, following the task plan (represented as a DAG) to support reliable and adaptive execution. Its responsibilities focus on task sequencing, context management, execution and verification monitoring, error handling, and dynamic replanning.

**Task Sequencing and Context Management** The coordinator first determines a topological order of task nodes to respect dependencies. Before executing each node, it compiles a context that includes outputs from prior tasks. This ensures that subtasks receive all necessary inputs and maintain consistency across dependent operations.

**Execution and Verification Management** For each node, the coordinator executes the task within a controlled retry loop, invoking the executor with proper context and invoking the verifier to check the executor output. If the verification passes, the coordinator proceeds to the next node in the plan sequence. If the verification fails, the coordinator retries the execution-verification loop up to a maximum number of attempts (defaulted to 3), updating the context with previous execution results to guide subsequent retries.

**Error Handling and Replanning** If a node fails verification after all retries, the coordinator will collect execution traces and trigger replanning to generate a revised task plan. Execution restarts from the top of the new plan. This approach adapts to failures, prevents the propagation of errors, and improves overall task correctness. Iteration limits (defaulted to 5) ensure termination, either with a successful final result or a failure report if maximum replanning cycles are exceeded. Replanning follows the same instructions from the VERIMAP planner, adding only minimal guidance to inspect the execution trace and generate a new plan (details in Appendix A.1).

The complete coordinator pseudocode (Algorithm 1) and additional implementation details are provided in Appendix B.4.

## 3 Evaluation

### 3.1 Setup

**Evaluation Datasets** We evaluate VERIMAP on five datasets (full details in Appendix B.1 and Table 5) spanning Question Answering (QA), Programming, and Math tasks:

- **MultiHopRAG** (Tang and Yang, 2024), a question-answering dataset designed to require multi-hop retrieving and reasoning to find the correct answer;
- **HumanEval** (Chen et al., 2021), a human-written programming dataset to evaluate the functional correctness of generated code;
- **BigCodeBench-Hard** (Zhuo et al., 2024), a challenging subset of BigCodeBench with complex, real-world programming tasks;
- **GSM8K** (Cobbe et al., 2021), a high-quality grade school math problems dataset;
- **Olympiads** (Li et al., 2024), a curated collection of challenging problems from various mathematics olympiads.

**Tools** We equip VERIMAP and baselines with tools tailored to the task type: for Programming, we provide a sandboxed environment with tools for file operations (create, edit, delete) and for executing bash commands, Python scripts, and pytest tests. For Math tasks, the agent has a sandboxed Python executor, a simple arithmetic calculator, and a computer algebra system-enabled calculator. For QA tasks, we provide a specialized search tool built on Sentence Transformers, which supports pagination and conditional filtering of articles by attributes like title, author, category, date, source, and content.

**Baselines** We first compared VERIMAP against *ReAct* agents instantiated with different backbone LLMs. Each agent was given access to the full set of tools available in our system. The backbone models were chosen to mirror the stronger and weaker models used in VERIMAP's planner and other components (i.e., executor, verifier), respectively.

For multi-agent baselines, we implemented two variants, denoted as **MAP** and **MAP-V**.

- **MAP** The Multi-Agent-Planning (MAP) baseline follows AOP[2] (Li et al., 2025) setup, where multiple agents collaborate through planning without verification. Each agent is equipped with tool calling and ReAct-style reasoning capabilities.

- **MAP-V** adds generic LLM verifiers to the MAP baseline, following the setup of VeriLA (Sung et al., 2025) and SelfCheck (Miao et al., 2024)[3]. The verifier is implemented as a **ReAct agent** with access to all available tools. This allows it to make independent decisions about which tools to invoke when verifying subtask outputs, given the corresponding instruction and context. This baseline can be viewed as a simplified variant of

[2] Unlike AOP, functionalities such as math, search, and code are abstracted as tools that executors can flexibly invoke, rather than being modeled as independent agents directly addressed in the plan.

[3] Note that SelfCheck performs step-wise verification of chain-of-thought reasoning, which is comparable but not identical to verifying subtasks through a separate agent.

| Method | RAG | Programming | | Math | |
|---|---|---|---|---|---|
| | MultiHopRAG | HumanEval | BigCodeBench-Hard | GSM8K | Olympiads |
| ReAct (`gpt-4o-mini`) | 61.20% | 81.10% | 27.03% | 90.00% | 25.00% |
| ReAct (`gpt-4.1`) | 68.40% | 90.24% | 36.49% | 91.80% | 40.80% |
| MAP | 67.00% | 78.88% | 28.38% | 57.20% | 21.40% |
| MAP-V | 77.60% | 88.96% | 28.38% | 87.00% | 29.00% |
| VERIMAP (ours) | **78.20%** | **93.92%** | **40.54%** | **93.60%** | **41.20%** |
| MAP-V-1it | 73.20% | 86.50% | 28.38% | 82.00% | 23.60% |
| VERIMAP-1it | 75.40% | 91.22% | 31.08% | 89.20% | 32.00% |

Table 1: Accuracy of single- and multi-agent baselines, VERIMAP, and their variants. `-1it` denotes ablation variants limited to a single iteration without replanning. All multi-agent systems use `gpt-4.1` as the planner and `gpt-4o-mini` as the executor. The best results are shown in **bold**.

VERIMAP, generating a single NL-based verification function that treats the subtask instruction itself as the verification criterion.

More details can be found in Appendix B.5.

**Evaluation Metrics** Across all datasets, we report the outcome accuracy as our primary evaluation metric. For the math and QA tasks, we determine the correctness of a generation using a heuristic-based criterion. For the programming tasks, a solution is considered correct if it passes all the accompanying test cases on the first attempt (`pass@1`).

**Implementation Details** For the underlying LLMs, we utilize `gpt-4.1` for the planner and `gpt-4o-mini` for the executor, unless otherwise specified. The temperature is set to 1.0 and top_p is 1.0. Additional implementation details are provided in Appendix B.

### 3.2 Results

Table 1 reports accuracy across five benchmarks, showing that VERIMAP consistently achieves the best performance on all tasks.

**Comparison with single-agent methods** VERIMAP outperforms the strong tool-enabled ReAct agent (backed by the same reasoning model `gpt-4.1`) by a significant margin of +9.8% on MultiHopRAG and +4.05% on the challenging BigCodeBench-Hard benchmark, demonstrating the effectiveness of planning and verification in complex reasoning tasks.

**Comparing with multi-agent systems** When multi-agent coordination is ineffective, single-agent solutions powered by strong models can outperform them (ReAct with `gpt-4.1` outperforms MAP on all five datasets and beats MAP-V on four). In contrast, VERIMAP effectively leverages planner-generated VFs to enhance performance, achieving notable gains of +12.2% on Olympiads and +12.16% on BigCodeBench-Hard.

**Effect of replanning** To investigate the impact of replanning and the role of VFs in enhancing replanning quality, we report single-iteration variants (VERIMAP-1it and MAP-V-1it) that do not perform replanning at the bottom of Table 1. As expected, replanning has a clearly positive effect. However, even without replanning, VERIMAP-1it outperforms the MAP-V baseline with replanning on most benchmarks as well as the ReAct baseline with `gpt-4o-mini`. More importantly, when replanning is applied, VERIMAP achieves substantial gains of +9.46% on BigCodeBench-Hard and +9.2% on Olympiads benchmarks, whereas replanning provides limited benefit for MAP-V (0% and +5.39%, respectively). These results indicate that planner-generated VFs capture more informative runtime signals, which significantly enhance performance during the replanning phase. Additional statistics on the average number of iterations are provided in Appendix D.2.

**Results on other LLMs** Additionally, we experiment with `o3` and `claude-sonnet-4` as planners to assess the generality of VERIMAP (Table 2). Overall, the results further validate the effectiveness of the verification-aware planning design: VERIMAP with a reasoning-oriented planner achieves the strongest performance on three datasets. Notably, on the challenging Olympiads benchmark, VERIMAP attains substantial improvements of +9.8% over ReAct with `o3` and +34.2% over ReAct with `claude-sonnet-4`. At the same time, the analysis reveals that different models excel

| Method | RAG | Programming | | Math | |
|---|---|---|---|---|---|
| | MultiHopRAG | HumanEval | BigCodeBench-Hard | GSM8K | Olympiads |
| *Single-agent baselines* | | | | | |
| ReAct (`o3`) | 31.80% | 95.45% | 39.19% | **95.60%** | 58.60% |
| ReAct (`claude-sonnet-4`) | **84.40%** | **95.68%** | 27.84% | 61.40% | 34.20% |
| `o3` *as planner* | | | | | |
| MAP | 52.00% | 88.41% | 24.32% | 57.80% | 47.80% |
| MAP-V | 65.80% | 87.80% | 28.38% | 89.60% | 56.90% |
| VERIMAP | 55.80% | 89.04% | **40.54%** | **95.60%** | **68.40%** |
| `claude-sonnet-4` *as planner* | | | | | |
| MAP | 74.20% | 84.00% | 20.27% | 50.40% | 26.05% |
| MAP-V | 79.00% | 86.40% | 22.22% | 84.60% | 38.07% |
| VERIMAP | 64.60% | 87.60% | 32.39% | **95.60%** | 47.69% |

Table 2: Accuracy of single- and multi-agent baselines and VERIMAP on other LLMs. All multi-agent systems use `gpt-4o-mini` as the executor unless otherwise stated. The best results are shown in **bold**.

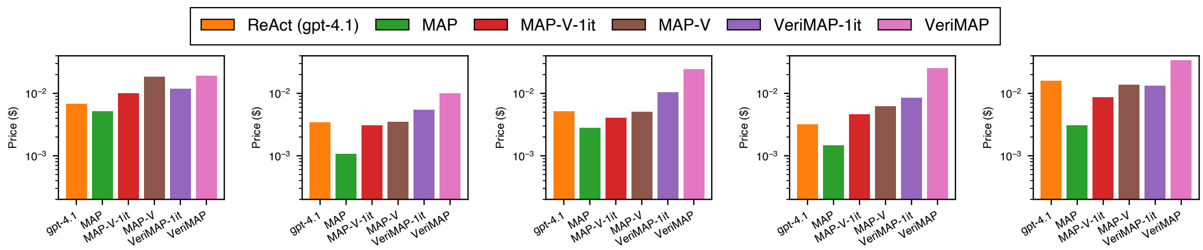
in different domains. For instance, ReAct with `o3` performs poorly on MultiHopRAG (31.8%), whereas `claude-sonnet-4` achieves markedly higher accuracy (84.4%), yet `o3` outperforms `claude-sonnet-4` on math-related tasks. This discrepancy likely stems from model specialization: `claude-sonnet-4` is optimized for tool-use behaviors such as retrieval, core to question answering in MultiHopRAG, while `o3` demonstrates stronger formal reasoning capabilities essential for mathematical problem-solving. Furthermore, when a model already performs exceptionally well on datasets like MultiHopRAG or HumanEval (potentially due to training exposure), generating high-quality plans may become more difficult than directly solving the problem, as evidenced by the observed performance drops of multi-agent solutions on these tasks. We also observe that ReAct with `claude-sonnet-4` yields lower-than-expected performance on GSM8K. Manual inspection suggests this stems from prompt–model mismatch and fragile formatting assumptions inherited from GPT-4.1-tuned templates, rather than intrinsic limitations of Claude's mathematical abilities. Importantly, VERIMAP mitigates these issues: with `claude-sonnet-4`, it attains 95.6% on GSM8K, indicating that verification-aware planning can still effectively harness the backbone even when the corresponding ReAct pipeline struggles. Also, Section D.1 describes experiments on `gpt-oss-120b`, an open-source LLM.

**Cost Analysis** Figure 2 reports the average LLM API cost per task across benchmarks and methods, using `gpt-4.1` as the planner (Full calculation details in Appendix B.6). Compared to the single-agent (ReAct) baseline, VERIMAP generally incurs higher costs due to the added verification and retry mechanisms. However, since most of the reasoning is handled by the planner, VERIMAP can delegate execution to less costly models, partially offsetting the overhead. Despite a modest increase in cost (on the order of \$0.001), these additional computations directly enable the performance gains reported in Table 1.

We further observe that the cost gap between VERIMAP and the next-best performing method, MAP-V, depends on task complexity. For simpler tasks such as GSM8K, which likely overlap with model pre-training data or require limited reasoning, models can solve them efficiently, resulting in a larger cost gap where VERIMAP 's additional verification offers limited utility. Conversely, for harder benchmarks like Olympiads, where models cannot easily derive efficient solutions, the cost gap narrows as VERIMAP 's verification and replanning become essential for achieving correct results. This balance between cost and utility highlights the importance of selecting systems according to task difficulty and performance needs. Further analysis about cost breakdown is in Appendix D.3.

### 3.3 Verification Function Analysis

**VF statistics** Table 3 presents the distribution and characteristics of planner-generated VFs from VERIMAP. The planner adapts the VF type to the task domain: for programming benchmarks such as BigCodeBench-Hard, it predominantly produces Python VFs, averaging 14.15 executable checks per task, whereas for text-based tasks like MultiHopRAG, it primarily relies on LLM-based VFs (7.36 per task) to assess semantic correctness. The plan-




(a) MultiHopRAG (b) HumanEval (c) BigCodeBench-Hard (d) GSM8K (e) Olympiads

Figure 2: Total API cost (planner, executor, verifier) across five benchmarks. The y-axis indicates average per-task cost in USD, using `gpt-4.1` as the planner and `gpt-4o-mini` as the executor and verifier (when applicable).

| Dataset | Average Number | | Average Length | |
|---|---|---|---|---|
| | Python VF | NL VF | Python VF | NL VF |
| MultiHopRAG | 0.25 | 7.36 | 159.7 | 190.0 |
| HumanEval | 8.71 | 1.75 | 283.8 | 161.3 |
| BigCodeBench-Hard | 14.15 | 2.41 | 381.6 | 174.3 |
| GSM8K | 6.39 | 1.46 | 139.8 | 156.5 |
| Olympiads | 10.96 | 5.07 | 243.8 | 165.0 |

Table 3: Verification function profiling across datasets. Average Number refers to the average number of VF calls per task. Length is in characters.

| Method | VERIMAP | | MAP-V | |
|---|---|---|---|---|
| | FP | FN | FP | FN |
| MultiHopRAG | 21.24% | 4.81% | 18.80% | 14.20% |
| HumanEval | 2.14% | 18.57% | 13.51% | 0.68% |
| BigCodeBench-Hard | 22.97% | 21.62% | 71.62% | 0.00% |
| GSM8K | 6.40% | 0.40% | 12.80% | 1.80% |
| Olympiads | 55.40% | 1.00% | 65.40% | 1.40% |

Table 4: Error analysis of VERIMAP and MAP-V on five benchmarks. FP stands for false positive, and FN stands for false negative.

ner also varies the complexity of VFs with task difficulty, as reflected in the average VF length, 139.8 for GSM8K compared to 381.6 for BigCodeBench-Hard. These patterns suggest that the planner adjusts both the form and complexity of verification according to the nature of the task.

**Error Analysis of Verification Functions** Ideally, verification functions should neither incorrectly mark failed executions as passing (false positives) nor incorrectly flag correct executions as failing (false negatives). Table 4 reports the frequency of false positives and false negatives for generic verifications from MAP-V and planner-generated verifications from VERIMAP. MAP-V verifications tend to have low false negative rates due to their flexibility, but this same flexibility often leads to high false positive rates, indicating that the functions can be "too loose." In contrast, VERIMAP generally exhibits lower false positive rates[4], reflecting a more structured and comprehensive verification strategy. However, this occasionally comes at the cost of higher false negatives, particularly in HumanEval and BigCodeBench-Hard, where the approach can be "overly conservative." Overall, even without external signals, VERIMAP is able to generate VFs that more effectively monitor the execution of multi-agent systems.

### 3.4 Case study (Example from Olympiads)

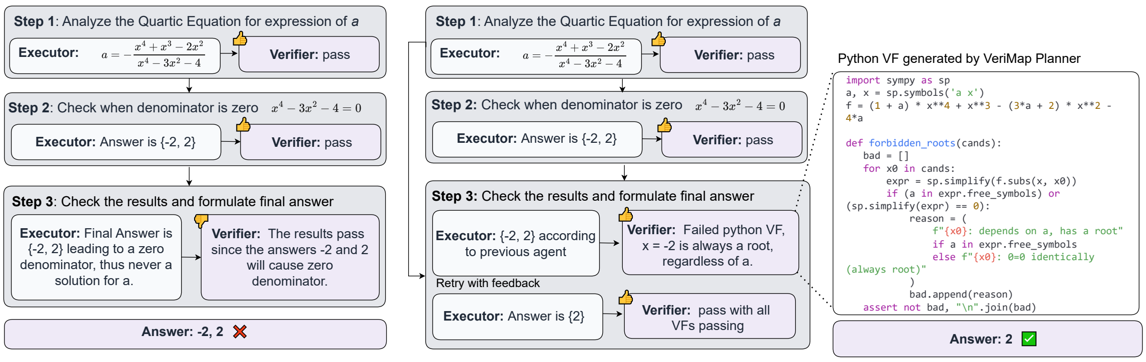
**Example Task**

Given that $(1+a)x^4+x^3-(3a+2)x^2-4a=0$, does there exist a $x_0 \in \mathbb{R}$ that there is never a solution for any $a \in \mathbb{R}$?

Comparison between MAP-V and VERIMAP solutions are shown in Figure 3. Both reached a reasonable plan and arrived at a seemingly correct execution at step 2, identifying that $a = -\frac{x^4+x^3-2x^2}{x^4-3x^2-4}$, and that the denominator must be zero for $a$ to have no solution, yielding $x = \pm 2$. However, $(x+2)$ is a common factor of both the numerator and denominator. When $x = -2$, the equation always holds regardless of the value of $a$. Thus, $-2$ should not be a valid answer.

With the generic natural language–based verification using only the context from step 2, MAP-V failed to verify the answer against either the numerator and the full expression, leading to a false positive verification result. In contrast, in VERIMAP, even though the executor made the same initial mistake, Step 3 included (i) necessary context from Step 1 (see the connecting edge) and (ii) a structured Python-based verification that checks for `free_symbols`. This allowed VERIMAP to correctly capture the error, provide feedback to the

[4]Except for MultiHopRAG, where most VFs are NL-based




(a) MAP-V

(b) VeriMAP

Figure 3: Comparison of planning and execution–verification traces between MAP-V and VERIMAP. At step 3, the MAP-V verifier failed to detect the incorrect answer $-2$ (marked with a thumbs-down), whereas VERIMAP, guided by the planner-generated VF, successfully identified the error and used the feedback to guide the executor to the correct answer in the next attempt.

executor, and ultimately converge on the correct solution.

This comparison highlights the effectiveness of planner-generated verifications in capturing nuanced and complex errors. Moreover, the subtle but necessary structural differences in the plan demonstrate how enforcing verification generation, in turn, improves overall plan quality.

## 4 Related Work

**Planning with Language Models** Planning has become a central focus in enabling LLMs to perform complex reasoning tasks, usually by decomposing them into intermediate and actionable steps. Yao et al. (2023) introduced ReAct, which interleaves reasoning and action to improve transparency and adaptability, though it remains limited on more complex tasks. ADaPT (Prasad et al., 2024) extends this idea by adaptively deciding the level of decomposition, enabling more flexible planning across different task types. Recent work emphasizes more structured frameworks, such as Plan-and-Act (Erdogan et al., 2025) which proposes a two-stage system with fine-tuned planners and executors, while Kim et al. (2024) use retrieval-augmented decomposition and trajectory exemplars. ExploraCoder (Wang et al., 2025) distinguishes reactive, preactive, and exploratory strategies, showing exploratory hybrids to be more robust with unseen APIs in program synthesis. These systems highlight the promise of single-agent planning but also expose their susceptibility to compounding reasoning errors.

**Multi-Agent Systems** Unlike single-agent planning, multi-agent systems (MAS) distribute responsibilities across collaborating or competing agents, enabling specialization but introducing new challenges in coordination. Benchmarks such as MultiAgentBench (Zhu et al., 2025) provide systematic evaluations of communication topologies and coordination protocols, showing how explicit planning can improve collective outcomes. Architectures such as MALMM (Singh et al., 2024) introduce hierarchical roles with supervisory agents to balance efficiency and adaptability. RopMura (Wu et al., 2025) employs a router to dynamically select agents and planners to solve a query, enabling accurate and efficient question answering. Despite these advances, failures remain common. Cemri et al. (2025) analyze why multi-agent LLM systems often underperform, pointing to unstable communication, misaligned objectives, and cascading coordination errors. These findings highlight that while MAS can mitigate single-agent limitations through distributed reasoning, they also introduce new challenges that demand stronger mechanisms for coordination, alignment, and reliability.

**Verification** Verification has emerged as a complementary component to planning. LLM-based evaluators include AutoCalibrate (Liu et al., 2024), which aligns evaluators to human labels, and EvoPrompt (Guo et al., 2024), which evolves prompts to improve evaluator reliability. Stechly et al. (2024) further show that LLMs struggle with reliable self-verification in reasoning and planning tasks, underscoring broader concerns about the stability of LLM evaluators (Shankar et al., 2024).

In collaborative settings, planners may generate verifier agents (Suzgun and Kalai, 2024), or frameworks may define verification steps automatically (Hao et al., 2025; Grigorev et al., 2025). Formal approaches such as AdaPlanner (Sun et al., 2023) introduce symbolic verification, offering guarantees at the cost of flexibility. Recent work such as VeriLA (Sung et al., 2025) emphasizes human-centered evaluation for interpretable verification of agent failures, highlighting how interpretability and reliability remain open challenges. Together, these studies suggest verification is essential but often treated as an external module, raising questions about tighter integration with planning processes. Gu et al. (Gu et al., 2025) explore iterative verification in a code-centric setting, using program analysis to drive LLM-based unit test generation. In contrast, VERIMAP completes verification in a more generic setting spanning multiple tasks. Thus, while conceptually related, the methods and contributions differ substantially. A non-exhaustive comparison of existing systems with VERIMAP is shown in Table 6.

## 5 Conclusion

We introduced VERIMAP, a framework for multi-agent collaboration with verification-aware planning. Planner-generated verification functions monitor and guide agent execution, enabling reliable coordination and self-refinement. Experiments show that VERIMAP improves robustness, interpretability, and performance across diverse complex tasks.

## Limitations

While VERIMAP demonstrates consistent improvements across diverse benchmarks, it is not without limitations.

**Replanning** In the current VERIMAP, when a node fails verification after all retries, the coordinator collects all execution traces and augments the original planning prompt with this history to generate a revised plan. The iterative refinement is essential to VERIMAP as shown in its comparison with VERIMAP-1it. While effective, this simple mechanism calls for further improvements. A key question is what signals to collect from failed executions and how to incorporate them effectively. Future work could move beyond appending the full history and instead analyze specific failure signals, such as error types or patterns across retries,to enable more targeted plan corrections, potentially improving both efficiency and robustness.

**Centralized Planning** VERIMAP relies on a centralized planner that dictates task decomposition and context sharing. This makes the system heavily reliant on the capability of the planner model: weaker planners may generate suboptimal task decompositions or verification functions, leading to degraded performance. Although explicit and planner-driven coordination is generally more effective for tasks with well-specified requirements, there exists a more decentralized, "group-chat" style of planning where agents operate with greater autonomy under only lightweight guidance. This may be better suited for more creative or open-ended tasks. We leave the exploration of such decentralized approaches to future work.

**Resource Constraints** VERIMAP currently assumes reliable access to multiple large models and external tools, which may limit its applicability in resource-constrained settings. Future work could focus on reducing cost through adaptive verification strategies, improving planner robustness, and adapting VERIMAP to resource-constrained settings.

## Ethical Considerations

Our work does not involve sensitive personal data or human subjects. However, as with any system that relies on large language models, there are potential ethical risks. First, reliance on powerful LLMs may perpetuate biases embedded in training data, and our framework does not explicitly mitigate such biases. Second, the verification process, while designed to improve robustness, could inadvertently reinforce flawed reasoning if verification functions are poorly specified. Third, multi-agent coordination introduces additional complexity for transparency: while agents can correct each other, the planning and verification chain may reduce interpretability for end users. Finally, increased computational demand raises sustainability concerns due to energy consumption. We encourage future work to investigate bias-aware verification functions, develop more interpretable coordination mechanisms, and design energy-efficient verification strategies.

**Disclosures on AI Usage** The authors confirm that the majority of the text in this paper was written by the authors themselves. We used GPT-5 to assist with brainstorming ideas, editing for clarity, and improving language fluency, and Copilot to assist with coding. Any content generated by AI tools has been reviewed and edited by the authors, and we take full responsibility for the final content of the paper.

## References


Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. 2025. Why do multi-agent llm systems fail? *Preprint*, arXiv:2503.13657.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *Preprint*, arXiv:2110.14168.

Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2025. Plan-and-act: Improving planning of agents for long-horizon tasks. *Preprint*, arXiv:2503.09572.

Danil S. Grigorev, Alexey K. Kovalev, and Aleksandr I. Panov. 2025. Verifyllm: Llm-based pre-execution task plan verification for robots.

Sijia Gu, Noor Nashid, and Ali Mesbah. 2025. Llm test generation via iterative hybrid program analysis. *Preprint*, arXiv:2503.13580.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. 2024. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *The Twelfth International Conference on Learning Representations*.

Yilun Hao, Yongchao Chen, Yang Zhang, and Chuchu Fan. 2025. Large language models can solve real-world planning rigorously with formal verification tools. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3434–3483, Albuquerque, New Mexico. Association for Computational Linguistics.

Minsoo Kim, Victor Bursztyn, Eunyee Koh, Shunan Guo, and Seung-won Hwang. 2024. RaDA: Retrieval-augmented web agent planning with LLMs. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13511–13525, Bangkok, Thailand. Association for Computational Linguistics.

Ao Li, Yuexiang Xie, Songze Li, Fugee Tsung, Bolin Ding, and Yaliang Li. 2025. Agent-oriented planning in multi-agent systems. In *The Thirteenth International Conference on Learning Representations*.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for "mind" exploration of large language model society. *Preprint*, arXiv:2303.17760.

Jia Li, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Costa Huang, Kashif Rasul, Longhui Yu, Albert Jiang, Ziju Shen, Zihan Qin, Bin Dong, Li Zhou, Yann Fleureau, Guillaume Lample, and Stanislas Polu. 2024. NuminaMath: The largest public dataset in ai4maths with 860k pairs of competition math problems and solutions. `https://huggingface.co/datasets/AI-MO/NuminaMath-CoT`. Hugging Face repository.

Yuxuan Liu, Tianchi Yang, Shaohan Huang, Zihan Zhang, Haizhen Huang, Furu Wei, Weiwei Deng, Feng Sun, and Qi Zhang. 2024. Calibrating LLM-based evaluator. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 2638–2656, Torino, Italia. ELRA and ICCL.

Ning Miao, Yee Whye Teh, and Tom Rainforth. 2024. Selfcheck: Using LLMs to zero-shot check their own step-by-step reasoning. In *The Twelfth International Conference on Learning Representations*.

Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. 2024. ADaPT: As-needed decomposition and planning with language models. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 4226–4252, Mexico City, Mexico. Association for Computational Linguistics.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Preprint*, arXiv:2305.18290.

Shreya Shankar, J.D. Zamfirescu-Pereira, Bjoern Hartmann, Aditya Parameswaran, and Ian Arawjo. 2024. Who validates the validators? aligning llm-assisted evaluation of llm outputs with human preferences. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, UIST '24, New York, NY, USA. Association for Computing Machinery.

Harsh Singh, Rocktim Jyoti Das, Mingfei Han, Preslav Nakov, and Ivan Laptev. 2024. Malmm: Multi-agent large language models for zero-shot robotics manipulation. *Preprint*, arXiv:2411.17636.

Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. 2024. On the self-verification limitations of large language models on reasoning and planning tasks. *Preprint*, arXiv:2402.08115.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. 2023. Adaplanner: Adaptive planning from feedback with language models. *Preprint*, arXiv:2305.16653.

Yoo Yeon Sung, Hannah Kim, and Dan Zhang. 2025. Verila: A human-centered evaluation framework for interpretable verification of llm agent failures. *Preprint*, arXiv:2503.12651.

Mirac Suzgun and Adam Tauman Kalai. 2024. Meta-prompting: Enhancing language models with task-agnostic scaffolding.

Yixuan Tang and Yi Yang. 2024. Multihop-rag: Benchmarking retrieval-augmented generation for multi-hop queries. *Preprint*, arXiv:2401.15391.

Yunkun Wang, Yue Zhang, Zhen Qin, Chen Zhi, Binhua Li, Fei Huang, Yongbin Li, and Shuiguang Deng. 2025. ExploraCoder: Advancing code generation for multiple unseen APIs via planning and chained exploration. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 18124–18145, Vienna, Austria. Association for Computational Linguistics.

Feijie Wu, Zitao Li, Fei Wei, Yaliang Li, Bolin Ding, and Jing Gao. 2025. Talk to right specialists: Routing and planning in multi-agent system for question answering. *Preprint*, arXiv:2501.07813.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Kunlun Zhu, Hongyi Du, Zhaochen Hong, Xiaocheng Yang, Shuyi Guo, Zhe Wang, Zhenhailong Wang, Cheng Qian, Robert Tang, Heng Ji, and Jiaxuan You. 2025. MultiAgentBench : Evaluating the collaboration and competition of LLM agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8580–8622, Vienna, Austria. Association for Computational Linguistics.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, and 14 others. 2024. BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. *Preprint*, arXiv:2406.15877.

# A Prompts

## A.1 VERIMAP Planner Prompts

Figure 4 and Figure 5 provides the detailed prompt used for the VERIMAP planner, where {available_tools} is a placeholder for the list of tools the executors and verifiers can use, {task_instruction} is the specific user task to be solved, and {demo_example} is an optional few-shot task-specific demonstration of a VERIMAP-generated plan.

Figure 6 provides the replanning prompt inserted into the planning prompt when the initial plan fails, where {previous_dag_str} is the JSON representation of the previous failed DAG-based plan, and {failed_context} is the traceback of the input/output of the failed node and two generations of ancestors of the failed node in the failed execution.

## A.2 VERIMAP Executor Prompts

Figure 7 and Figure 8 provide the prompts used for the VERIMAP executor. It includes two parts: the system prompt that sets up the agent's behavior (Figure 7) and the task prompt (Figure 8) that provides specific instructions for each subtask. In the system prompt, {tool_desc} is a placeholder for the list of tools the agent can use.

In the task prompt, {subtask} is the name of the subtask to be executed, {instruction} is the detailed instruction for the subtask provided by the planner, and {contexts} are the relevant contexts provided by the planner in the structured input format.

## A.3 VERIMAP Verifier Prompts

Figure 9 provides the prompt used for the VERIMAP verifier for LLM-based verifications, where {agent_input} is the original task input, {verify_prompt} is the verification requirement provided by the planner, and {agent_output} is the structured output produced by the executor.

# B Implementation Details

## B.1 Dataset Details

A detailed description of the datasets used in our experiments is provided in Table 5. Due to the limited computation budget and to balance the dataset sizes, for all datasets, we choose at most 500 random samples as the training set and at most 500 random samples as the test set. For datasets with fewer than 500 samples, we use half of the samples as the training set and the other half as the test set.

**Prompt for VERIMAP planner (Part I: Objective and Structure)**

You are a planner responsible for generating high-level action plans to solve arbitrary tasks.
**Objective**
Decompose the input task into smaller, manageable **subtasks** that can be executed by **agents capable of calling external tools**. Tools available are {available_tools}
Each subtask should:

- Be represented as a **node** in a dependency graph.
- Contain a **high-level name**.
- Include a **detailed, executable instruction** specifying how to use dependency context.
- Focus on goal, required inputs, and expected outputs.
- Be self-contained and executable by another agent.
- Choose granularity carefully.

**Structure**
Represent the plan as a **Directed Acyclic Graph (DAG)** where:
Each node includes:

- `id`
- `name`
- `instruction`
- `input`
- `output`
- `verification`

Each edge represents a dependency between nodes.
**Task Instructions**
{task_instruction} {demo_example}

Figure 4: Prompt for VERIMAP Planner (Part I).

| Dataset | Description | # Training | # Test |
|---|---|---|---|
| MultiHopRAG* | Multi-hop QA over a news knowledge base (Tang and Yang, 2024) | 500 | 500 |
| HumanEval | Python function synthesis from docstrings; evaluated by unit tests (Chen et al., 2021) | 164 | 164 |
| BigCodeBench-Hard | Execution-based code generation with diverse library calls and complex instructions; evaluated by unit tests (Zhuo et al., 2024) | 74 | 74 |
| GSM8K* | Grade-school math word problems requiring multi-step reasoning (Cobbe et al., 2021) | 500 | 500 |
| Olympiads* | Competition math problems with standardized chain-of-thought solutions (Li et al., 2024) | 500 | 500 |

Table 5: Dataset statistics. * represents samples of the original datasets.

The training set is only used for few-shot learning and Training-based VERIMAP, while the test set is used for evaluation. Note that the main results of VERIMAP is not affected by the training- and test-set division, since the basic VERIMAP is a training-free framework.

Across the five datasets, MultiHopRAG, HumanEval, BigCodeBench-Hard, and GSM8K are used as-is. The Olympiads dataset is originally from the NuminaMath-CoT benchmark (Li et al., 2024), which contains 32,926 problems. We filter out competition math problems with standardized chain-of-thought solutions, resulting in 1,337 problems. We then randomly select 500 problems as the training set and the other 500 problems as the test set. The full dataset is available at `https://huggingface.co/datasets/Metaskepsis/Olympiads`.

### B.2 Executor Implementation Details

In our implementation, the Executor is a *ReAct*-style single agent. It is provided with a ReAct system prompt, the task name, detailed instructions, and relevant contexts from the Coordinator. The system prompt explicitly directs the agent to produce its final answer in a structured output format, and the task prompt provides a detailed description of the subtask along with a structured output guide. The structured output guide specifies the expected output variables, their types, and any constraints or formats that need to be adhered to. An example of the ReAct system prompt and task prompt is

## Prompt for VERIMAP planner (Part II: Verification and Output Format)

**Verification Instruction**

Each subtask must include verification functions to evaluate correctness.
Verification types:

- **Python unit tests**
- **LLM-based criteria**

If `type == "python"`:

- Structural validation
- Functional validation
- Edge cases

If `type == "llm"`:

- Provide a clear judging instruction.

**Formatting Instructions**

Output JSON structure:

```
{
  "nodes": [...],
  "edges": [...]
}
```

- Do not include additional explanations.
- Do not wrap JSON in markdown.

Figure 5: Prompt for VERIMAP Planner (Part II).

## Replanning Prompt for VERIMAP Planner

The previous attempt to solve this task failed. Please generate a new plan based on the following guidance and context:
Previous DAG that failed: {previous_dag_str}
Context of previous failure: {failed_context}
Generate a revised DAG plan. Ensure the new plan addresses the previous issues.

Figure 6: Replanning Prompt for VERIMAP Planner

provided in Appendix A.2.

The agent is allowed a maximum of 20 rounds to complete a given subtask. If the agent fails to produce a final, structured output within these 20 rounds, the execution is considered a failure, and the process proceeds directly to the error handling stage.

### B.3 Verifier Implementation Details

In our implementation, each executor node can be equipped with a suite of verification functions. A subtask is deemed successfully completed only when its output passes all associated verification functions. If any single verifier fails, the execution is considered a failure, and the process is escalated to the Coordinator's error handling stage.

The feedback mechanism is tailored to the type of verification being performed:

1. For **Python-based verification**, if a programmatic assertion fails, the system automatically captures and returns the complete traceback and the specific assertion message as feedback. This provides precise, actionable information for debugging and correction.

2. For **LLM-based verification**, the verifier LLM is set to be the same as the executor LLM. Similarly, it is powered by a ReAct-style agent and has access to the same set of tools as executors. It is prompted to produce a structured output containing a binary success/-fail evaluation and a detailed textual feedback given the LLM-based verification prompt and the output from the executor. This feedback is designed to be constructive, explicitly identifying which parts of the original answer were incorrect and providing a comprehensive solution for how to amend it. Prompts used for LLM-based verification are provided in Appendix A.3.

**System Prompt for VeriMAP Executor**

You are designed to help with a variety of tasks, from answering questions to providing summaries to other types of analyses—and to prepare your outputs so that any downstream "child" agents can pick up and continue work seamlessly.
**Tools**
You have access to a wide variety of tools. You are responsible for using the tools in any sequence you deem appropriate to complete the task at hand. This may require breaking the task into subtasks and using different tools to complete each subtask.
You have access to the following tools: {tool_desc}
**Coordination with Other Agents**
You need to anticipate downstream needs because you are a part of a multi-agent system. Structure your output so that each child agent can identify what to do next—e.g. by providing context summaries, data snippets, or explicit instructions.
You may also receive additional contexts from upstream agents. These contexts may include relevant information, redundant details, or noise. **Use only relevant information** from the provided context; do not copy or summarize irrelevant content.
**Do not repeat work** already handled by upstream agents.
**Output Format**
Please answer in the same language as the question and use the following format:

```
Thought: The current language of the user is: (user's language). I need to use a tool to help me answer the question.
Action: tool name (one of tool_names) if using a tool.
Action Input: the input to the tool, in a JSON format representing the kwargs (e.g. {"input": "hello world", "num_beams": 5})
```

Please ALWAYS start with a Thought.
NEVER surround your response with markdown code markers. You may use code markers within your response if you need to.
Please use a valid JSON format for the Action Input. If you include the "Action:" line, then you MUST include the "Action Input:" line too, even if the tool does not need kwargs, in that case you MUST use "Action Input: {}".
If this format is used, the tool will respond in the following format:

```
Observation: tool response
```

You should keep repeating the above format till you have enough information to answer the question without using any more tools. At that point, you MUST respond in one of the following two formats:

```
Thought: I can answer without using any more tools. I'll use the user's language to answer
Answer: [your answer here (In the same language as the user's question)]
```

```
Thought: I cannot answer the question with the provided tools.
Answer: [your answer here (In the same language as the user's question)]
```

MAKE SURE that you include EVERYTHING needed in the answer part, as the verifier and downstream agents won't be able to see your thinking process, even though that might make the answer longer, that is necessary.
**Current Conversation**
Below is the current conversation consisting of interleaving human and assistant messages.

Figure 7: System Prompt for VeriMAP Executor

**Task Prompt for VeriMAP Executor**

You are working on a subtask, which is part of a complete task.
**Input**
**Your task:** {subtask}
**Instructions for your subtask:** {instruction}
**Contexts for your subtask:** {contexts}
**Structure**
Please first go through the Thought-Action process.
**Instructions**

- Do **not** include any additional text or explanations.
- Do **not** wrap the JSON output in code blocks or markdown formatting.
- Do **not** output again the observations. They are provided to you, don't repeat them.
- Do **not** put any symbols like "," "'" after your "Action Input:"'s json. The only thing after your "Action Input" should be a valid json.

Figure 8: Task Prompt for VeriMAP Executor

### B.4 Coordinator Implementation Details

The Coordinator's orchestration logic is governed by specific parameters and data handling rules to ensure deterministic behavior.

- **Execution Model:** To simplify implementation, the Coordinator executes the nodes in the DAG sequentially, following a standard topological sort. Parallel execution is left for future work.
- **Inter-node Data Flow:** When a node has multiple parent nodes, the Coordinator merges their structured outputs into a single dictionary. It

**Prompt for VERIMAP Verifier**

You are an expert evaluator charged with judging whether an agent's response meets a given verification requirement, based on three pieces of information: the original task input, the verification requirement, and the agent's output.
**Objective:** Assess the correctness of the agent's output against the verification requirement. Always leverage your tools to perform independent verification of any facts, data, or claims.
**Scoring Scale:**

**0 (Failure):** Output misses key requirements or contains serious errors, or output is wrong in format.

**1 (Success):** Output adequately satisfies the requirement.

**Response Format (JSON only):**

```
{
"success_score": <0 or 1>,
"reasoning": "<1-2 sentence justification, explain IN DETAIL, SPECIFICALLY about how you will amend

             the answer>"
}
```

**Context Variables:**

- **Agent Input:** {agent_input}
- **Verification Requirement:** {verify_prompt}
- **Agent Output:** {agent_output}

**Guidelines:**

1. Return *only* the JSON object—no extra text, markdown, or code blocks.
2. `success_score` must be exactly 0 or 1.
3. **MANDATORY:** Independently verify all claims using your available tools (e.g., fact-checks, data lookups, calculation scripts). Never assume the agent's explanation is complete or correct without tool-based confirmation.
4. Focus strictly on correctness relative to the requirement; level of explanatory detail in the agent's output beyond correctness is *not* required.

Figure 9: Prompt for VERIMAP Verifier

then provides this dictionary as context to the child node's Executor, which can access the required inputs by their corresponding keys.

- **Final Output:** The DAG contains one node explicitly marked as the final output node. The Coordinator returns only the structured output from this specific node upon successful completion of the entire graph.

### B.5 Baseline Implementation Details

**Single-Agent Baselines (ReAct)** These agents utilize the ReAct framework and have access to the same set of tools as the VERIMAP executor. Their internal reasoning process is limited to a maximum of 20 steps per task.

**MAP Data Handling** In the **MAP** ablation, where structured I/O is disabled, an executor node is provided with the complete prompt and raw text output from all of its parent nodes up to two generations back. This historical context helps compensate for the missing structured data. The prompts given to the executor are identical to those in the full VERIMAP system, with the exception of clauses related to generating structured outputs and verification criteria.

**MAP-V Verifier** The verifier in the **MAP-V** baseline is powered by the same LLM as the executor. Its implementation is consistent with a standard LLM-based verification, where its goal is to check for alignment between the generated output and the specific instructions the executor received for that subtask.

**Single-Round Execution** In the **MAP-V-1it** and **VERIMAP-1it** configurations, the system completes one full pass of the DAG. If a node's verification fails, its output (whether incorrect or empty)

**Algorithm 1** Task Coordinator for Multi-Agent Execution with Multiple Verifiers per Node

**Require:** $P = (V, E)$ ▷ Task plan as DAG with nodes $V$ and edges $E$
**Require:** $\{VF_v \mid v \in V\}$, where $VF_v = \{VF_{v,1}, VF_{v,2}, \dots\}$ ▷ Node-specific verification functions
**Require:** $R_{\max}$ ▷ Maximum retry limit per node
**Require:** $I_{\max}$ ▷ Maximum replanning iteration limit
**Ensure:** Result $\in$ {exec_out, failure} with execution traces

1: $order \leftarrow$ TOPOLOGICAL-SORT$(P)$
2: $iter \leftarrow 0$ ▷ Replanning iteration counter
3: **while** $iter < I_{\max}$ **do**
4:     **for** each node $v \in order$ **do**
5:         $r \leftarrow 0$ ▷ Retry counter
6:         **repeat**
7:             **if** $r = 0$ **then**
8:                 $ctx \leftarrow$ COMPILE-CONTEXT$(v)$
9:             **else**
10:                 $ctx \leftarrow$ COMPILE-CONTEXT$(v, exec_out)$ ▷ Use previous output
11:             **end if**
12:             $exec\_out \leftarrow$ EXECUTE$(v, ctx)$ ▷ Invoke executor
13:             $ver\_res \leftarrow$ VERIFY$(exec_out, V)$ ▷ Invoke verifier
14:             **if** $ver\_res =$ pass **then**
15:                 **break** ▷ Proceed to next node
16:             **else**
17:                 $r \leftarrow r + 1$
18:             **end if**
19:         **until** $ver\_res =$ pass **or** $r \geq R_{\max}$
20:         **if** $ver\_res \neq$ pass **then**
21:             $traces \leftarrow$ COLLECT-TRACES$(v, exec_out, ver_res)$
22:             $P \leftarrow$ REPLAN$(P, traces)$ ▷ Generate new plan
23:             $iter \leftarrow iter + 1$
24:             **goto line 1** ▷ Start next iteration from new plan
25:         **end if**
26:     **end for**
27:     **return** $exec\_out$ ▷ Return latest execution result of last node
28: **end while**
29: **return** failure

is still passed as context to any subsequent child nodes for the remainder of that single iteration. No re-execution or re-planning is triggered.

### B.6 Cost calculation

For `gpt-4o-mini` as the executor, we assume the cost per 1M tokens as \$0.15 for input, \$0.08 for cached input (inputs that have been seen before), and \$0.60 for output. For `gpt-4.1` as the planner, we assume the cost per 1M tokens as \$2.00 for input, \$0.50 for cached input, and \$8.00 for output.

## C Training-based VERIMAP

To further enhance the quality of plan generation, we fine-tuned the Verification-Aware Planner using Direct Preference Optimization (DPO) (Rafailov et al., 2024). DPO is a methodology for aligning language models with human (or, in this case, machine-labeled) preferences. It operates on a dataset of triplets, each containing a prompt ($p$), a preferred response ($y_w$), and a less-preferred response ($y_l$). By training on these triplets, the model learns to increase the likelihood of generating responses similar to $y_w$ while decreasing the likelihood of those similar to $y_l$. Our process for creating this dataset and training the model is detailed

| System | MAS? | Verification Entity | Planner Generated Verification Module? | Python Verification? | Structured I/O? | Retry? |
|---|---|---|---|---|---|---|
| **VeriMAP** (ours) | ✓ | Subtask-level | ✓ | ✓ | ✓ | ✓ |
| **AdaPlanner** (Sun et al., 2023) | ✗ | Plan-level | ✗ | ✗ prompt-based check | ✗ | ✓ |
| **MetaPrompt** (Suzgun and Kalai, 2024) | ✗ single LM Meta Model | Plan & Execution level | ✗ | ✓ | ✗ | ✗ |
| **VerifyLLM** (Grigorev et al., 2025) | ✗ | Plan-level | ✗ | ✗ | ✗ | ✗ |
| **Self-Verification** (Stechly et al., 2024) | ✗ | Plan-level | ✗ | ✓ | ✗ | ✓ |
| **Formal Verification Tools** (Hao et al., 2025) | ✗ | Plan & Execution-level | ✗ | ? SMT solver | ? SMT format | ✓ |

Table 6: Comparison of planning and verification systems. **MAS?**: Multi-Agent System; **Verification Entity**: Granularity of verification process; **Planner Generated Verification Module?**: Verification module generated by the planner; **Python Verification?**: Verification implemented in Python; **Structured I/O?**: System uses structured input/output; **Retry?**: Retry planning after verification or execution failure; ✓: feature supported; ✗: feature not supported; ?: partial/uncertain support.

below.

### C.1 Data Collection

Our first step was to construct a dataset of preference pairs derived from the execution traces of VeriMAP. We designated a portion of our datasets to serve as a training set for this purpose as in Appendix B.1. On these tasks, we ran VeriMAP using `gpt-4.1` as the Planner and `gpt-4o-mini` for both the Executor and Verifier roles. To capture a variety of outcomes, each task was executed three times.

From these runs, we collected all execution iterations. For a given task (the prompt, $p$), a plan that led to a successful, verified outcome—often after one or more error-correction cycles—was labeled as the *preferred response* ($y_w$). Conversely, a plan generated within an iteration that ultimately failed verification and led to the re-planning stage was labeled as the *less-preferred response* ($y_l$). This process yielded a raw dataset of $(p, y_w, y_l)$ triplets.

### C.2 Data Cleaning and Augmentation

The raw dataset collected in the previous step had two primary limitations: 1) some failures may have been caused by random executor errors rather than genuinely poor plans, and 2) the diversity of planning errors was limited. We addressed these issues with a two-stage process.

First, to filter for meaningful preference pairs, each triplet was provided to `gpt-4.1` to judge whether the preferred and less-preferred plans exhibited a significant, structural difference in quality. Triplets where the plans were too similar, suggesting a non-deterministic execution failure, were removed from the dataset.

Second, to augment the dataset and improve its diversity, we first categorized common planning flaws observed in our experiments:

- Boundary condition removed/changed (e.g., off-by-one, empty input)
- Incorrect parameter/default value (e.g., ddof, timeout)
- Missing or overly broad exception handling
- Data-type or casting mistake (e.g., int vs float)
- Variable naming shadow or mismatch
- Inefficient algorithm (e.g., $O(n^2)$ vs. a built-in function)
- Deleted necessary pre/post-processing (e.g., dropna(), .strip())

| | ReAct | VeriMAP |
|---|---|---|
| MultiHopRAG | 59.80% | **71.60%** |
| HumanEval | 93.29% | **94.12%** |
| BigCodeBench-Hard | 37.84% | **41.89%** |
| GSM8K | **95.80%** | 95.40% |
| Olympiads | 64.40% | **68.80%** |

Table 7: Accuracy of single-agent baseline and VeriMAP on `gpt-oss-120b`. All multi-agent systems use `gpt-4o-mini` as the executor unless otherwise stated. The best results are shown in **bold**.

| | **Avg. Iter./Avg. Retr.** | |
|---|---|---|
| | VeriMAP | MAP-V |
| MultiHopRAG | 1.48 / 1.57 | 1.60 / 1.78 |
| HumanEval | 1.33 / 1.71 | 1.05 / 1.09 |
| BigCodeBench-Hard | 1.85 / 2.28 | 1.07 / 1.11 |
| GSM8K | 1.17 / 1.14 | 1.09 / 1.35 |
| Olympiads | 1.84 / 2.29 | 1.46 / 1.54 |

Table 8: Average iterations/retries of VeriMAP vs. MAP-V across benchmarks.

- Style/format only errors (e.g., extra header, comment)

We then used `gpt-4.1` to create new, synthetic training samples. For each high-quality triplet, we prompted the model to introduce one or more of the error types above into the plan to generate a new, even less-preferred version. This created additional, challenging negative examples for the DPO training.

### C.3 Hyperparameters and Training Details

Our initial data collection process yielded 150 high-quality preference samples. After the cleaning and augmentation stages, the final dataset was expanded to 300 samples.

We used `gpt-4.1` as the base model for fine-tuning, which is the same model used for the Planner in our main experiments to ensure a fair comparison. The model was fine-tuned using the Direct Preference Optimization implementation available through OpenAI's fine-tuning API. The training was configured with the following hyperparameters: a **batch size** of 2, a single training **epoch**, a **learning rate multiplier** of 0.8, and a DPO **beta** ($\beta$) of 0.2.

## D Additional Experiment Results

### D.1 Results on open-source LLMs

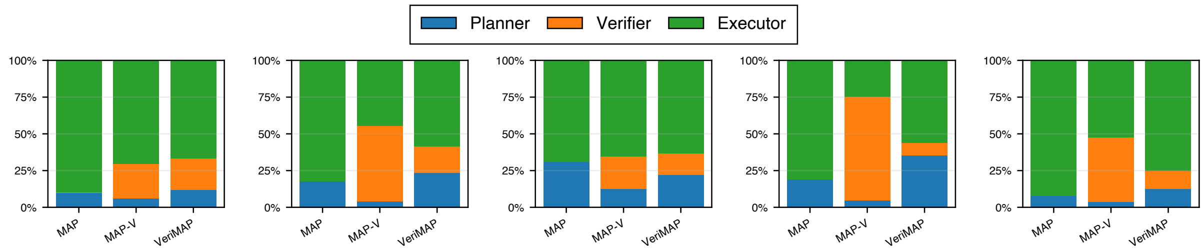
To further examine the applicability of open-source LLMs on VeriMAP, we have conducted an experiment using ReAct with `gpt-oss-120b` as the single-agent baseline and VeriMAP using `gpt-oss-120b` as the planner and `gpt-4o-mini` as the executor. Table 7 shows that VeriMAP is also applicable to `gpt-oss-120b` and achieves better performance on all datasets other than GSM8K, which is a saturated dataset, highlighting that VeriMAP can also work with open-source LLMs.

### D.2 Iteration counts

Table 8 shows the average number of iterations and retries for VeriMAP and the MAP-V baseline. Through more flexible and rigorous self-correction, VeriMAP achieves superior accuracy. While on tasks with clearer success criteria like MultiHopRAG, VeriMAP demonstrates greater efficiency with fewer iterations (1.48 vs. 1.60) and retries (1.57 vs. 1.78). On Programming and Olympiads tasks, VeriMAP exhibits significantly higher average iterations and retries (e.g., 1.85 iterations and 2.28 retries on BigCodeBench-Hard, vs. 1.07 and 1.11 for the MAP-V baseline). We attribute the higher iteration counts to its effective verification mechanism. VeriMAP's sophisticated verifiers successfully detect subtle flaws that the weaker MAP-V verifier misses. The MAP-V baseline often terminates prematurely after an undetected error, leading to lower iteration counts but also poor final accuracy. In contrast, VeriMAP correctly identifies these failures and puts more effort in retries and re-planning to find a correct solution. Therefore, the higher iteration counts on difficult problems represent the necessary cost of achieving higher quality and robustness, directly explaining the performance gains reported in Table 1.

### D.3 Cost breakdown

Figure 10 shows the detailed cost breakdown of different components in VeriMAP and the MAP and MAP-V baselines across all benchmarks. A consistent cost distribution is evident across all methods, where the Executor is the most expensive component, followed by the Verifier, and finally the Planner. A core finding is that VeriMAP's Planner cost (the blue bar) is consistently higher than that of the MAP-V baseline. This reflects the additional work required from our planner to generate not just an execution plan but also the so-




(a) MultiHopRAG (b) HumanEval (c) BigCodeBench-Hard (d) GSM8K (e) Olympiads

Figure 10: Cost decomposition of different components in VERIMAP and the MAP and MAP-V baselines across all benchmarks. The cost is broken down into the planner cost, the executor cost, and the verifier cost.

phisticated, structured verification logic. However, this upfront investment in planning leads to significant efficiency gains during the verification stage, particularly on programming tasks like HumanEval and BigCodeBench-Hard. On these benchmarks, VERIMAP's Verifier cost (the orange bar) is substantially lower because it generates token-efficient, executable Python unit tests instead of relying on lengthy, prompt-based checks. This demonstrates that VERIMAP's more intelligent planner directly enables a more effective and economical verification process.

### D.4 Training-based VERIMAP

We have experimented with the training based VERIMAP-Planner as introduced in Section C. It has achieved accuracies of 64.77%, 93.06%, 37.68%, 93.00% and 44.85% on the MultiHopRAG, HumanEval, BigCodeBench-Hard, GSM8K, and Olympiads datasets, respectively. The training-based VERIMAP-Planner achieves the best performance on the challenging Olympiads benchmark, with a significant +3.65% improvement over the standard VERIMAP. This indicates that fine-tuning the Planner to generate more robust and verifiable plans is particularly beneficial for extremely difficult tasks. However, on other benchmarks, VERIMAP-DPO shows mixed results, sometimes underperforming compared to the original VERIMAP, suggesting that while DPO training can enhance plan quality in certain contexts, it may also reduce generality in others. Further investigation is needed to fully understand the conditions under which DPO training provides consistent benefits.