MAP-Boosting: A Multi-Agent Framework for Automated Prompt Optimization

Anonymous Authors[1]

Abstract

Prompt design substantially impacts large language model (LLM) performance, yet prompt optimization remains largely manual and intuition-driven. Existing automated methods typically rely on holistic feedback signals, limiting their ability to identify and correct domain-specific failures. We propose MAP-Boosting (Multi-Agent Prompt Boosting), an iterative framework that simulates a pseudo-gradient search over the discrete instruction space using an ensemble of agents to prioritize failures by domain importance, frequency, and complexity. We evaluate MAP-Boosting on AIME 2024, LiveCodeBench, SWE-bench Verified, FinBen, and a real-world covenant extraction task. MAP-Boosting achieves substantial gains over strong baselines, improving performance from 25.6% to 42.5% on AIME 2024 and from 76.2% to 86.8% on LiveCodeBench. Ablation studies further confirm the effectiveness of the iterative failure-driven process, demonstrating consistent improvements, including an additional 11.7% gain on AIME even when starting from strong aspect-aware initializations.

1. Introduction

Large Language Models (LLMs) have demonstrated impressive capabilities across a wide range of tasks, from natural language processing to code generation and mathematical reasoning (Brown et al., 2020; Wei et al., 2022). However, their performance is highly sensitive to prompt quality, with even minor phrasing changes capable of causing significant output variations (Zhao et al., 2021). While techniques like few-shot learning (Brown et al., 2020), Chain-of-Thought prompting (Wei et al., 2022), and structured formatting improve outcomes, they often demand extensive manual engineering. As task complexity grows, the prompt design space becomes intractable especially for engineers with less domain expertise, rendering manual optimization an inefficient and often prohibitive endeavor.

Current prompt engineering practices face three key challenges. First, prompt refinement typically relies on human intuition and trial-and-error, lacking systematic domain-specific error diagnosis. Existing papers and frameworks often struggle with domain-specific error validation, frequently failing to provide granular improvements that address the unique constraints of specialized tasks (Yang et al., 2024; Khattab et al., 2024). Mathematically, the prompt space is discrete and non-convex, and since LLM outputs are non-differentiable, traditional gradient-based optimization is inapplicable. Second, existing evaluations frequently provide holistic judgments rather than pinpointing specific failure modes, hindering targeted improvements. Third, manual methods struggle to scale efficiently across different domains or adapt to evolving requirements.

Crucially, unlocking the capabilities of smaller, more cost-effective models through prompt optimization remains a significant challenge. To address these limitations, this paper proposes **M**ulti-**A**gent **P**rompt **Boosting** (MAP-Boosting), a novel framework that systematizes prompt optimization through iterative refinement. Drawing inspiration from the principles of boosting algorithms (Freund & Schapire, 1997; Schapire, 2013), MAP-Boosting treats prompt optimization as a search problem over the discrete space of instructions. We use the term boosting informally to describe iterative prioritization of difficult cases; MAP-Boosting does not claim equivalence to classical boosting algorithms or their theoretical guarantees.

This framework introduces three key innovations: Decomposed Validation, which breaks tasks into discrete domain aspects for fine-grained analysis; Prioritized Failure Diagnosis, which quantitatively prioritizes errors using boosting principles; and Closed-Loop Optimization, which iteratively refines prompts.

Our main contributions include: a multi-agent architecture for automated prompt optimization; a quantitative, weighted feedback mechanism that prioritizes improvements based

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

on domain knowledge and error frequency; and a robust iterative framework that achieves significant gains across benchmarks including AIME 2024, LiveCodeBench, and SWE-bench Verified.

2. Related Work

This section reviews existing literature on automated prompt engineering, multi-agent systems, and iterative refinement, highlighting the progression toward structured, feedback-driven optimization.

2.1. Automated Prompt Engineering

Early efforts in automated prompt engineering, such as Automatic Prompt Engineer (APE) (Zhou et al., 2023), frame prompt discovery as a search problem. Other methods include gradient-based approaches (Shin et al., 2020) and reinforcement learning techniques (Deng et al., 2022). Google DeepMind's OPRO (Yang et al., 2024) utilizes the LLM itself as an optimizer, iteratively generating new prompts based on a discovery trajectory of previous scores. While OPRO demonstrated strong performance, it often treats the prompt as a monolithic instruction. Existing methods generally lack the granular, aspect-based decomposition required for complex domain-specific tasks.

2.2. Iterative and Self-Refinement Methods

Iterative prompt engineering techniques, such as Self-Refine (Madaan et al., 2023), improve initial outputs through iterative feedback and refinement, often critiquing LLM-generated text holistically. RePrompt (Reynolds & McDonell, 2021) and other iterative approaches build on this by refining prompts step-by-step, but typically without granular domain-specific aspect decomposition or quantitative weighting. OpenAI's 2024 Prompt Optimization Framework (OpenAI, 2024) provides heuristic-based tools rather than autonomous agentic loops. These frameworks struggle to prioritize specific failure modes across diverse training samples effectively.

2.3. Multi-Agent LLM Systems

The paradigm of multi-agent LLM systems has gained traction, with frameworks like AutoGen (Wu et al., 2023) and ChatDev (Qian et al., 2023) enabling collaborative task-solving. Methods like MIRAGE (Li et al., 2024) explore multi-agent refinement, but not specifically for prompt optimization. CREATIVE (Chen et al., 2024) leverages role-based agents for prompting but emphasizes dialogue over structured validation. In contrast, MAP-Boosting adopts a more deterministic, cyclical workflow with fixed agent roles, prioritizing reproducibility and targeted optimization.

2.4. Self-Correction and LLM-as-Judge

The concept of LLMs engaging in self-correction has been explored by works like Self-Refine (Madaan et al., 2023). Constitutional AI (Bai et al., 2022) uses principle-based feedback to guide model behavior. MAP-Boosting advances these ideas by: (1) moving beyond holistic critique to parallel, aspect-specific validation, offering much finer-grained error diagnosis, and (2) introducing a quantitative error weighting mechanism directly inspired by boosting.

2.5. Research Gaps

Despite these advancements, several gaps remain in the literature: Granularity of Feedback, Most existing frameworks provide holistic feedback, which often fails to capture subtle domain-specific errors; Prioritization of Failure Modes, Current methods lack a principled mechanism to prioritize improvements based on error frequency and task complexity; Scaling to Small-Sample Regimes, Many automated techniques require large validation sets, whereas real-world applications often necessitate optimization with minimal labeled data; Integration of Domain Expertise, Bridging the gap between empirical failure patterns and predefined domain importance is often overlooked in automated loops.

3. MAP-Boosting Framework

3.1. Overview

MAP-Boosting systematically orchestrates four distinct phases—Generation, Validation, Evaluation, and Optimization—all coordinated by a central RootAgent. This cyclical process begins by decomposing complex tasks into verifiable domain specific aspects. LLM outputs are generated using the current prompt, evaluated in parallel against these aspects, and finally, the prompt is iteratively refined based on the identified and weighted errors.

3.2. Domain Aspect Extraction

For task $\mathcal{T}$ and documents $\mathcal{D}$, domain aspects $\mathcal{A} = \{a_1, \ldots, a_K\}$ are first extracted using an initial LLM call (Gemini 2.5 Flash). Each aspect defines a granular, verifiable criterion (e.g., `SyntaxValidity`) with an importance score ($I_a \in [0, 3]$) and complexity score ($C_a \in [1, 5]$) automatically assigned by the LLM. This decomposition transforms high-dimensional evaluation into strictly verifiable sub-tasks.

3.2.1. Phase 2: Multi-Agent Architecture and Execution

A central RootAgent coordinates specialized agents for each phase. The **PromptConstructorAgent** synthesizes the initial prompt P_0 from $\mathcal{T}$, $\mathcal{D}$, and $\mathcal{A}$. The **GenerationAgent**

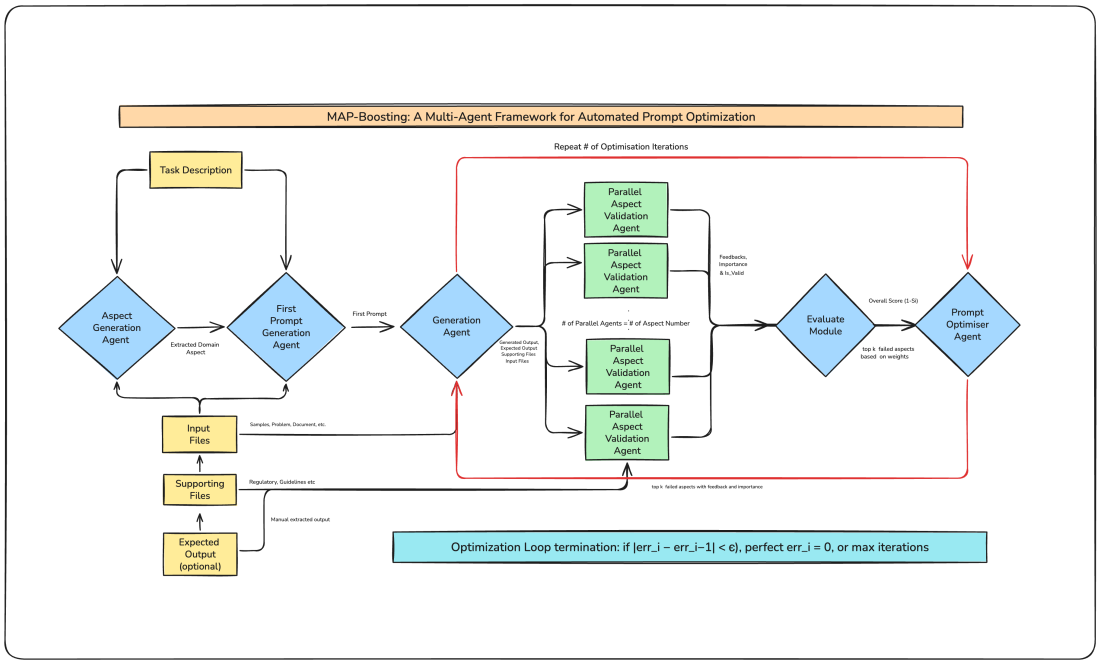


Figure 1. The MAP-Boosting architecture, illustrating the agent roles and cyclical workflow for iterative prompt optimization.

executes the task to produce outputs O_i. A set of parallel **ValidationAgents** then evaluates O_i against each aspect a_k, returning a binary validity v_k and descriptive feedback.

3.2.2. Phase 3: Quantitative Evaluation

The **EvaluationAgent** aggregates the result from all validation agents to compute weighted error scores. This stage transforms qualitative feedback into a quantifiable signal that represents the distance from an ideal solution.

3.2.3. Phase 4: Prompt Optimization

Finally, the **PromptOptimizerAgent** refines P_i into P_{i+1} by targeting prioritized failure modes while aiming to preserve successful behaviors ("Solved Aspects"), thereby simulating a "gradient" descent in the discrete instruction space.

3.3. Weighted Average Error (Optimization Objective)

During optimization, MAP-Boosting minimizes the **complexity-importance-weighted average error**, a dynamic metric that prioritizes critical, frequent, complex, and persistent failures.

Let $F_i \subseteq \mathcal{A}$ be the set of failing aspects at iteration i. For each a, let c_a be the number of training samples on which a failed. Define the safe normalized failure count:

$$n_a^{(i)} = \begin{cases} \frac{c_a}{M_i} & a \in F_i \\ 0 & \text{otherwise,} \end{cases} \qquad M_i = \max(1, \max_{a \in F_i} c_a) \quad (1)$$

The dynamic weight of a failing aspect $w_a^{(i)}$ is designed to prioritize errors that are frequent, important, complex, or persistent. The intuition is to focus optimization on "hard examples" that resist initial refinement. To prevent weight explosion and ensure stability, we introduce a **Nuclear Safety Cap** (w_{cap}) and a persistent failure penalty (ρ). The weight is calculated as:

$$w_a^{\text{base}} = \beta \cdot \frac{I_a}{3} + (1 - \beta) \cdot n_a^{(i)} \quad (2)$$

$$w_a^{\text{adj}} = w_a^{\text{base}} \cdot \left(1 + \lambda \cdot \frac{C_a - 1}{4}\right) \quad (3)$$

$$w_a^{(i)} = \min\left(w_{cap}, w_a^{\text{adj}} \cdot \rho^{\mathbb{1}(a \in F_i \cap F_{i-1})}\right) \quad (4)$$

Here, β balances domain importance (I_a) with empirical observations ($n_a^{(i)}$), λ is the complexity weighting factor, and $\rho < 1$ serves as a **discount factor** for aspects that remain invalid across consecutive iterations ($a \in F_i \cap F_{i-1}$). This damps the signal from stubborn failures, preventing the optimizer from fixating on potentially unresolvable errors at the expense of other solvable aspects.

The worst-case (maximum possible) weight used for normalization is calculated by assuming maximum failure frequency and applying the persistent failure penalty:

$$w_a^{\max} = \left[\beta \cdot \frac{I_a}{3} + (1 - \beta)\right] \cdot \left(1 + \lambda \cdot \frac{C_a - 1}{4}\right) \cdot \rho \quad (5)$$

and the global constant denominator is

$$W^{\max} = \sum_{a \in \mathcal{A}} w_a^{\max} \tag{6}$$

The optimization objective is

$$\text{avg\_error}_i = \frac{\sum_{a \in F_i} w_a^{(i)}}{W^{\max}} \in [0, 1] \tag{7}$$

and the score is $\text{score}_i = 1 - \text{avg\_error}_i$.

Theorem 3.1 (Idealized Monotonic Improvement)**.** *Under the assumptions (1) at least one failing aspect is fixed ($F_{i+1} \subset F_i$) and (2) no previously passing aspect begins failing:*

$$\textit{avg\_error}_{i+1} < \textit{avg\_error}_i \tag{8}$$

Proof. Let $E_i = \sum_{a \in F_i} w_a^{(i)}$ be the total weighted failure count. By assumptions (1) and (2), $F_{i+1} \subsetneq F_i$, meaning that at least one term $w_a^{(i)}$ is removed from the sum in iteration $i + 1$. Since all weights $w_a^{(i)} > 0$ (including those capped by w_{cap}), the numerator strictly decreases: $E_{i+1} < E_i$, it follows that $\frac{E_{i+1}}{W^{\max}} < \frac{E_i}{W^{\max}}$, hence $\text{avg\_error}_{i+1} < \text{avg\_error}_i$. □

This result should be interpreted as a design invariant rather than a convergence guarantee, and relies on assumptions that may not strictly hold in practice.

Note: These dynamic weights are used *only* during optimization to guide refinement. Final reported performance uses static $I_a \cdot C_a$ weighting (see Section 4.2).

3.4. Convergence

Optimization stops when the improvement in average error falls below a predefined threshold ϵ, or when the maximum number of iterations $N_{\max}$ is reached:

$$\begin{aligned} |\text{avg\_error}_i - \text{avg\_error}_{i-1}| < \epsilon \\ \text{or} \quad \text{avg\_error}_i = 0 \\ \text{or} \quad i = N_{\max} \end{aligned} \tag{9}$$

4. Experimental Setup

4.1. Datasets

We evaluate on four benchmarks: AIME 2024, LiveCodeBench, SWE-bench Verified, and FinBen. Datasets are split evenly for training (optimization) and testing. Details are in Table 1. The Covenants Extractions dataset involves extracting complex financial covenants from credit agreements and extracted covenants as expected outputs (20 samples), requiring deep legal reasoning.

Algorithm 1 MAP-Boosting

Input: Task $\mathcal{T}$, data $\mathcal{D}$, max iterations N, ϵ, top-K selection K
$\mathcal{A} \leftarrow \text{ExtractAspects}(\mathcal{T}, \mathcal{D})$
$P_0 \leftarrow \text{PromptConstructor}(\mathcal{T}, \mathcal{A})$
$i \leftarrow 0, err_{-1} \leftarrow \infty, err_0 \leftarrow \infty$
while $i < N$ **and** $|err_i - err_{i-1}| \geq \epsilon$ **and** $err_i > 0$ **do**
  $O_i \leftarrow \text{GenAgent}(P_i, \mathcal{D})$
  $\mathcal{V} \leftarrow\| \text{ValAgents}(O_i, \mathcal{A})$
  $F_i \leftarrow \{a \mid \mathcal{V}(a).\text{is\_valid} = 0\}$
  $err_i \leftarrow \text{WeightedErr}(\mathcal{V}, F_i, F_{i-1})$
  $\mathcal{F} \leftarrow \text{Top-}K \text{ by } w_a^{(i)}$
  $P_{i+1} \leftarrow \text{Optimizer}(P_i, \mathcal{F}, \mathcal{V})$
  $err_{i-1} \leftarrow err_i, F_{i-1} \leftarrow F_i, i \leftarrow i + 1$
end while
return P_i

4.2. Metrics

To evaluate the effectiveness of prompt optimization, this study utilizes three primary metrics that capture both coarse-grained performance and fine-grained adherence to task requirements:

- **Valid Solution Rate**: The percentage of test samples that satisfy *all* defined validation criteria (aspects). This represents the likelihood of an LLM producing a perfectly correct result.
- **Domain Specific Aspect-Level Improvement**: The percentage-point gain in success rate for individual aspects across the test set. This metric identifies which specific task requirements benefited most from refinement.
- **Weighted Accuracy (WA)**: Our primary metric, providing a weighted assessment of performance based on the inherent importance (I_a) and complexity (C_a) of each aspect:

$$\text{WA} = \frac{\sum_{a \in \mathcal{A}} I_a \cdot C_a \cdot v_a}{\sum_{a \in \mathcal{A}} I_a \cdot C_a} \times 100 \tag{10}$$

  where $v_a \in \{0, 1\}$ is the validity of aspect a *averaged over the test set*. This static weighting reflects inherent aspect criticality and difficulty without training-time frequency bias.

To demonstrate uplift on lower-capacity models, we compare optimized prompts against strong **Few-Shot** and **Chain-of-Thought (CoT)** baselines. To ensure statistical robustness, performance metrics are averaged over 6 independent experiment cycles for most baselines, while the DSPy baseline is evaluated over 4 cycles.In line with literature (Zhou

Table 1. Dataset characteristics and split sizes used for experimentation.

DATASET	DOMAIN/DIFFICULTY	TOTAL SAMPLES	TRAIN/TEST SPLIT
AIME 2024	MATH REASONING / HARD	30	15/15
LIVECODEBENCH	CODE GENERATION / MIXED	30	15/15
SWE-BENCH VERIFIED	SOFTWARE ENGINEERING / COMPLEX	40	20/20
THEFINAI/FINBEN-FINER-ORD	FINANCIAL NER / FINE-GRAINED	60	30/30
COVENANTS EXTRACTIONS	LEGAL/FINANCIAL / COMPLEX	20	10/10

Table 2. MAP-Boosting hyperparameters used for all experiments.

PARAMETER	VALUE
LLM TEMPERATURE	1.0
MAX OPTIMIZATION ITERATIONS	5
β (IMPORTANCE/FREQUENCY)	0.7
WEIGHT CAP (w_{cap})	0.45
COMPLEXITY WEIGHT (λ)	0.5
NEGATIVE EVIDENCE PENALTY (ρ)	0.85
CONVERGENCE THRESHOLD (ϵ)	0.0
MIN OPTIMIZATION ITERATIONS	2
MAX NO-IMPROVEMENT ITERATIONS	3

et al., 2023; Yang et al., 2024), we use a sample-efficient regime (15–30 training samples). To ensure robustness, we employ explicit train/test separation, multiple independent runs, and bootstrapped 95% confidence intervals.

The selection of the Gemini series is motivated by its 1M+ token context window, essential for SWE-bench Verified, and its reasoning capacity for identifying consistent failure modes across samples. Aspect-based decomposition provides a denser signal per example than scalar accuracy, effectively transforming small sample sets into rich failure mode datasets. Hyperparameters are detailed in Table 2.

5. Results

5.1. Performance Gains

MAP-Boosting consistently shows performance improvements across all evaluated benchmarks when compared to the initial optimization state, as measured by Weighted Accuracy (WA). As summarized in Table 3, we observe substantial gains: for LiveCodeBench, WA improves from 78.25% to 86.79% (**+8.54% gain**); for AIME 2024, WA increases from 30.76% to 42.49% (**+11.73% gain**), for FinBen, WA increases from 66.0% to 69.78% (**+3.78% gain**), for SWE-bench Verified, WA increases from 17.06% to 27.69% (**+10.63% gain**); and for Covenants Extractions, WA increases from 15.35% to 18.84% (**+3.49% gain**).

Notably, MAP-Boosting provides a performance improvement over representative few-shot + CoT baselines for structured tasks; for instance, for AIME 2024 the few-shot + CoT baseline (25.3%) to initial optimization state (30.8%) and for LiveCodeBench Code Generation the few-shot + CoT baseline (76.2%) to initial optimization state (78.25%). These improvements represent the best performance found during the optimization cycle. Statistical analysis suggests that these gains are significant ($p < 0.05$) for three out of five benchmarks.

These results are summarized in Figure 2, illustrating the performance of MAP-Boosting with statistical confidence. Similarly, for the Covenants Extractions dataset, a significant improvement in Weighted Accuracy is observed, from the Standard Baseline of 9.6% to a best run of 18.8% (**+9.2% gain**) as shown in Figure 10f. This addresses complex legal constraints across a limited training sample set. The optimization process particularly improved the model's ability to handle high-complexity fields such as `Next Due Date` and `Reasoning`, which initially saw near-zero accuracy due to their multi-step reasoning requirements.

5.2. Comparison to Baselines and DSPy

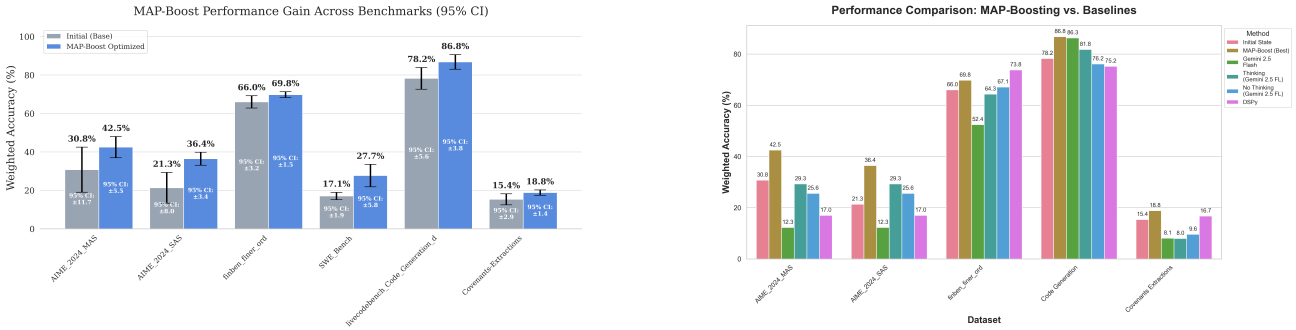
To situate MAP-Boosting within the prompt optimization landscape, we compare it against DSPy (Khattab et al., 2024), and few-shot + CoT baselines. While DSPy leverages declarative task signatures and structured predictors to enforce output schemas, MAP-Boosting uniquely combines aspect-level decomposition with iterative, weight-based feedback updates.

Our DSPy baseline employs the `BootstrapFewShot` teleprompter to optimize the prompts. We configure it with `max_bootstrapped_demos=2` and `max_labeled_demos` set to the training set size (e.g., 15 for AIME), using a custom metric that leverages our ValidationAgents to score traces. This represents a strong, optimized baseline rather than a static prompt. As shown in Figure 2b, MAP-Boosting consistently outperforms the optimized DSPy baseline across most complex reasoning and coding benchmarks. For instance, on AIME 2024, MAP-Boosting achieves 42.5% Weighted Accuracy (WA), significantly surpassing the DSPy baseline (17.0%). Similar trends are observed in other domains: on LiveCodeBench, MAP-Boosting reaches 86.8% WA vs DSPy's 75.2%; and on Covenants Extractions, MAP-Boosting reaches 18.8%

Table 3. Statistical Summary of Weighted Accuracy (WA) Across Experiment Cycles (95% Bootstrapped CI and IQR)

Dataset	Initial Performance			Best Performance			Improvement	
	Mean	95% CI	IQR	Mean	95% CI	IQR	Mean	95% CI
AIME 2024 (MAS)	30.76%	±11.73%	24.00%	42.49%	±5.51%	5.33%	+11.73%	±7.82%
AIME 2024 (SAS)	21.33%	±8.84%	11.56%	36.45%	±3.99%	5.78%	+15.12%	±10.25%
FinBen (NER)	66.00%	±3.20%	4.00%	69.78%	±1.53%	1.56%	+3.78%	±4.09%
SWE-Bench Verified	17.06%	±1.91%	4.69%	27.69%	±5.83%	4.06%	+10.63%	±5.34%
LiveCodeBench	78.25%	±5.64%	7.46%	86.79%	±3.80%	1.49%	+8.54%	±5.89%
Covenants Extractions	15.35%	±2.86%	6.48%	18.84%	±1.40%	0.94%	+3.49%	±2.39%

* "Initial Performance" here refers to the *Initial Optimization State* (Iteration 1), which uses domain aspects.



(a) Comparison across benchmarks.

(b) Breakdown against baselines.

Figure 2. Global performance comparison of MAP-Boosting. (a) Improvement over iterations. (b) Comparison against standard baselines and initial states.

WA vs DSPy's 16.7%. Notably, DSPy achieves higher accuracy on the FinBen classification task (73.8% vs 69.8%), likely due to its effective few-shot example selection for finer-grained entity recognition. These results highlight the benefit of iterative aspect-aware feedback, particularly for tasks requiring deep reasoning beyond pattern matching.

5.3. Analysis of Aspect-Level Improvements

Detailed aspect-level improvements (Appendix B) demonstrate that MAP-Boosting effectively rectifies domain-specific errors. We observe significant gains in `OutputFormat` and `EdgeCaseHandling` for LiveCodeBench, `EntityLabeling` and `LocationEntityIdentification` for FinBen, and `NoExternalAssumptions` and `DynamicTargeting` for SWE-bench Verified. In AIME 2024, consistent improvements in `CalculatingTripleIntersection`, `DeterminingAllFourOwnership` and `UnderstandingGivenValues` highlight the framework's capability to enhance complex mathematical reasoning.

5.4. Qualitative Error Analysis

To understand the underlying mechanisms of improvement, we categorize failures into three primary domains based on their complexity and nature:

- **Formatting (Complexity 1)**: Low-level syntax and structural violations, such as missing JSON brackets or extraneous conversational filler. These are usually resolved within the first two optimization iterations.
- **Structural (Complexity 2-3)**: Failures in identifying specific entities or attributes correctly (e.g., `OrganisationEntityIdentification` in FinBen or `CovenantCategory` in Legal tasks).
- **Logical/Reasoning (Complexity 4-5)**: High-order failures requiring multi-step deduction, such as `BugResolution` in SWE-bench or `CalculatingTripleIntersection` in AIME. These aspects show the highest headroom for improvement through iterative refinement.

A detailed taxonomy and distribution of these error types across benchmarks are provided in Appendix C.

5.5. Empirical Evolution of Optimized Prompts

The `PromptOptimizerAgent` effectively "learns" to enforce higher-order constraints. As a case study, consider the AIME benchmark. In early iterations, the model frequently failed on mathematical precision when constrained to a strict JSON format.

By Iteration 4, the optimizer had injected a critical instruction: *"Adopt a step-by-step thinking process... avoid unsubstantiated claims."* This transformation effectively integrated Chain-of-Thought (CoT) reasoning into the structured output task, resolving the reasoning-format conflict. Similarly, in FinBen, instructions were refined to strictly prohibit extraneous tokens between labels, directly addressing the BIO tagging parsing errors.

The convergence behavior across these iterations is further visualized in Figure 10 (Appendix), which shows that most benchmarks reach a performance plateau after 3 to 4 optimization steps. This suggests that the framework effectively captures high-impact failure modes early, with later iterations focusing on niche edge cases or fine-tuning the wording for maximum precision.

5.6. Model-Specific Baselines

We also isolate optimization impact by comparing against: (1) **Gemini 2.5 Flash** (more complex model); (2) **Thinking (Gemini 2.5 FL)** Gemini 2.5 Flash Lite with native reasoning and CoT + few-shot; and (3) **No Thinking** (Gemini 2.5 Flash Lite with standard CoT + few-shot). MAP-Boosting consistently outperforms these baselines, as illustrated in Figure 2b, demonstrating gains that exceed simple model scaling or naive reasoning activation.

6. Sensitivity Analysis

To isolate the performance contributions of MAP-Boosting's core architectural components, a series of experiments was conducted across diverse benchmarks. The analysis focuses on the influence of weighted feedback bias (β), complexity-weighted error prioritization (C_a), temperature variations, and the structural necessity of decomposed validation agents. Detailed analyses are provided for FinBen (Appendix E), SWE-bench Verified (Appendix F), and LiveCodeBench Code Generation (Appendix G).

6.1. Influence of Weighted Feedback (β)

The β parameter controls the trade-off between domain importance and failure frequency. When $\beta = 0$, the optimizer prioritizes aspects with the highest failure counts, while $\beta = 1$ focuses solely on domain importance. Our results across all three benchmarks show that $\beta = 0$ consistently provides the most stable optimization trajectory and substantial performance gains (e.g., +44.0% on FinBen and +29.3% on SWE-Bench), suggesting that addressing frequency-based bottlenecks is a more effective initial strategy than purely importance-driven refinement.

6.2. Component Necessity: Decomposed vs. Holistic Validation

A central hypothesis of MAP-Boosting is that fine-grained, aspect-based validation is superior to holistic evaluation. In a "Holistic Ablation," the parallel `ValidationAgents` are replaced with a single prompt asking for a general pass/-fail grade and feedback. This configuration resulted in a 44.6% slower convergence rate (for AIME 2024) and significantly less precise prompt updates, as the optimizer struggled to identify the specific root cause of failures among multiple overlapping instructions.

6.3. Impact of Complexity Prioritization (C_a)

By weighting errors by their inherent task complexity, MAP-Boost ensures that the LLM's limited "attention budget" is spent on the hardest parts of the task. However, the impact of this parameter is highly dataset-dependent. For coding tasks like LiveCodeBench, aggressive complexity prioritization ($\lambda = 1$) is critical, achieving a gain of +35.9% over its initialization. Conversely, for reasoning and knowledge-heavy tasks like FinBen and SWE-Bench, a more balanced approach ($\lambda = 0$) results in better overall generalization.

6.4. Efficiency: Parallel vs. Single Agent Validation

A critical advantage of MAP-Boosting is its parallel validation architecture. For complex reasoning tasks like AIME 2024, the multi-agentic parallel validation approach significantly reduces optimization latency. Over a 5-Experiment cycles optimization, Multi-Agentic Parallel Validation (MAS) completed in **17:09:10** ($\sim$12,350s/it), whereas a Single Validation Agent (SAS) required **30:59:34** ($\sim$22,315s/it). This represents a **44.6%** reduction in total optimization time, demonstrating the scalability of decomposed multi-agent systems for iterative prompt refinement. See Figure 12 in Appendix K for a detailed comparison of convergence rates.

7. Discussion, Limitations, and Future Work

MAP-Boosting systematizes prompt optimization through a structured, iterative workflow. Its decomposed validation provides fine-grained feedback, improving explainability over holistic evaluations. The framework is sample-efficient, discovering high-impact instructions with few examples, and its modularity allows for rapid domain adaptation.

Limitations include reliance on the LLM's capacity as a

judge; smaller LLMs may fail to extract all critical domain aspects. Furthermore, while sample-efficient, our evaluation utilizes a limited number of test samples (10–30), which may not capture the full variance of larger production datasets. Optimization latency is also increased due to multiple LLM calls per iteration.

Because MAP-Boosting refines prompts iteratively, later prompts may be more verbose or explicit than initial ones. As a result, some improvements may partially reflect increased prompt capacity. Disentangling capacity effects from optimization dynamics is an important direction for future work.

MAP-Boosting assumes that validation agents provide reasonably aligned signals; performance may degrade when evaluation models are noisy or misaligned with task objectives.

Another critical limitation is the potential for aspect overfitting. The domain aspects extracted from a small few-shot set may notably lack generalizability. For example, problem-specific aspects such as `ApplyingInclusioExclusion`, `CalculatingTripleIntersection`, and `DeterminingAllFourOwnership` (from AIME 2024) are highly effective for their respective seed problems but may not be relevant for the broader distribution of domain tasks. Future work must focus on refining the aspect extraction process to derive more abstract, transferable principles rather than overfitting to the idiosyncrasies of the few-shot training seeds.

The selection of the Gemini series proved essential due to its 1M+ token context window. This capacity allows the `AspectExtractionAgent` to analyze multiple training examples simultaneously when identifying initial aspects, providing a broader base for requirement discovery than single-example analysis.

Future work will focus on improving the `AspectExtractionAgent` through more sophisticated orchestration, such as parallel or sequential chains that combine findings from diverse data subsets. We also plan to investigate adaptive feedback schedules and the integration of automated "Best-Prompt Checkpointing" to mitigate regression in late iterations.

8. Conclusion

MAP-Boosting presents a novel and effective multi-agent framework for automating prompt optimization. By leveraging structured validation, a boosting-inspired weighted feedback mechanism, and an iterative refinement process, it systematically enhances LLM performance across diverse and complex tasks. Our empirical evaluation using Gemini 2.5 Flash Lite on LiveCodeBench, SWE-bench Verified, AIME 2024, and FinBen demonstrates significant performance gains. These results validate that principled, automated prompt engineering can substantially uplift smaller models, offering a viable pathway to high performance without exclusively relying on the largest, most computationally expensive LLMs. While complex models would achieve higher baselines, MAP-Boosting proves effective at maximizing the potential of more efficient architectures. Overall, MAP-Boosting demonstrates how structured evaluation and refinement can be composed into a practical prompt optimization framework. While it does not introduce new learning-theoretic guarantees, our results suggest that systems-level design choices play a critical role in effective prompt optimization.

Impact Statement

This paper presents MAP-Boosting, a framework designed to automate and improve prompt engineering for Large Language Models. By enabling more effective use of smaller, cost-efficient models, this work contributes to the democratization of AI, allowing researchers and developers with limited computational resources to achieve high performance on complex tasks. However, we recognize that automated prompt optimization could also be used to systematically circumvent safety alignment or generate more convincing misinformation. Furthermore, the iterative nature of the framework involves multiple LLM calls per sample, which has associated energy and environmental costs. We encourage the community to use this framework responsibly and to prioritize the optimization of tasks that provide clear societal benefits.

References

Bai, Y., 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.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. In *Advances in neural information processing systems*, volume 33, pp. 1877–1901, 2020.

Chen, Y. et al. Creative: Role-based agents for prompting. *arXiv preprint arXiv:2402.00000*, 2024.

Deng, M., Wang, J., Hsieh, C.-P., Wang, Y., Guo, H., Shu, T., Song, D., Xing, E. P., and Hu, Z. RLPrompt: Optimizing discrete text prompts with reinforcement learning. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 3369–3391, 2022.

Freund, Y. and Schapire, R. E. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Vardhamanan, S., Haq, S., Sharma, A., Joshi, T. T., Moazam, H., et al. DSPy: Compiling declarative language model calls into self-improving pipelines. In *International Conference on Learning Representations*, 2024.

Li, J. et al. Mirage: Multi-agent refinement for multimodal tasks. *arXiv preprint arXiv:2401.00000*, 2024.

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. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

OpenAI. Prompt engineering guide and optimization framework. https://platform.openai.com/docs/guides/prompt-engineering, 2024. Accessed: 2024-12-20.

Qian, C., Cong, X., Yang, C., et al. Communicative agents for software development. *arXiv preprint arXiv:2308.08722*, 2023.

Reynolds, L. and McDonell, K. Prompt programming for large language models: Beyond the few-shot paradigm. *arXiv preprint arXiv:2102.07350*, 2021.

Schapire, R. E. Explaining AdaBoost. In *Empirical Inference*, pp. 37–52. Springer, 2013.

Shin, T., Razeghi, Y., Logan IV, R. L., Wallace, E., and Singh, S. Autoprompt: Eliciting knowledge from language models with automatically generated prompts. *arXiv preprint arXiv:2010.15980*, 2020.

Wei, J., Wang, X., Schuurmans, D., Maarten, B., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

Wu, Q., Bansal, G., Zhang, J., et al. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q. V., Zhou, D., and Chen, X. Large language models as optimizers. In *International Conference on Learning Representations*, 2024.

Zhao, Z., Wallace, E., Feng, S., Klein, D., and Singh, S. Calibrate before use: Improving few-shot performance of language models. In *International Conference on Machine Learning*, pp. 12697–12706. PMLR, 2021.

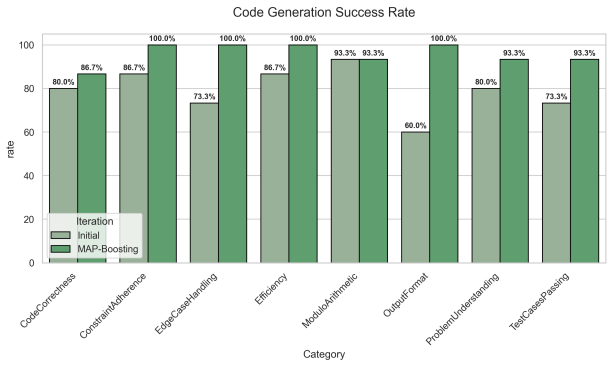
Zhou, Y., Muresanu, A. I., Han, Z., Paster, K., Pitis, S., Chan, H., and Ba, J. Large language models are human-level prompt engineers. In *International Conference on Learning Representations*, 2023.

A. Statistical Credibility and Confidence Intervals

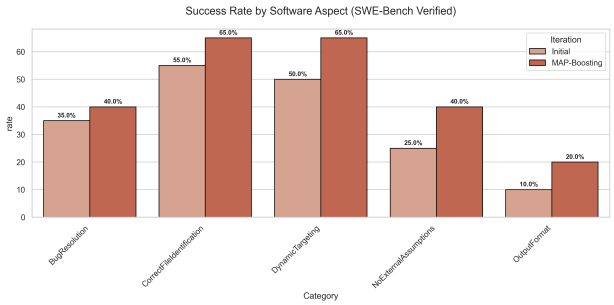
To ensure the robustness of the reported improvements, 95% Bootstrapped Confidence Intervals (CI) were calculated across 5 independent experiment cycles for each benchmark using the final complexity-weighted accuracy formula. Bootstrapping is particularly suited for this evaluation due to the small sample sizes ($n = 5$ per cycle), providing a more reliable estimate of error than standard T-distribution methods. As shown in Table 3, the improvements are statistically significant and demonstrate consistent convergence.

B. Detailed Benchmark Results

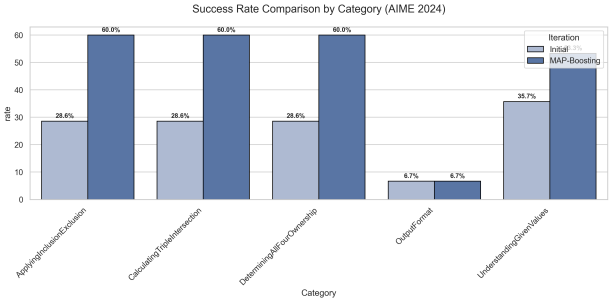
This section provides a detailed breakdown of performance across different aspects for each benchmark.



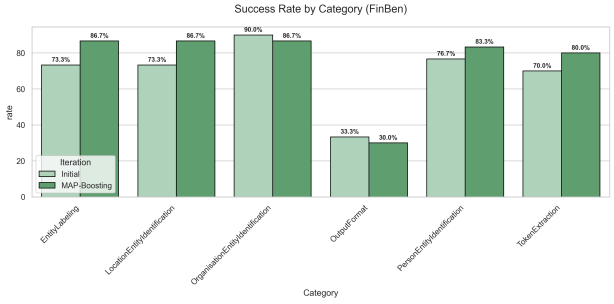
(a) LiveCodeBench Performance (Grouped by Category)



(b) SWE-bench Verified Performance (Grouped by Aspect)



(c) AIME 2024 Performance (Grouped by Category)



(d) FinBen Performance (Grouped by Aspect)

Figure 3. Detailed Benchmark Results: (a) LiveCodeBench, (b) SWE-bench Verified, (c) AIME 2024, and (d) FinBen.

C. Error Analysis and Convergence

C.1. Qualitative Error Analysis

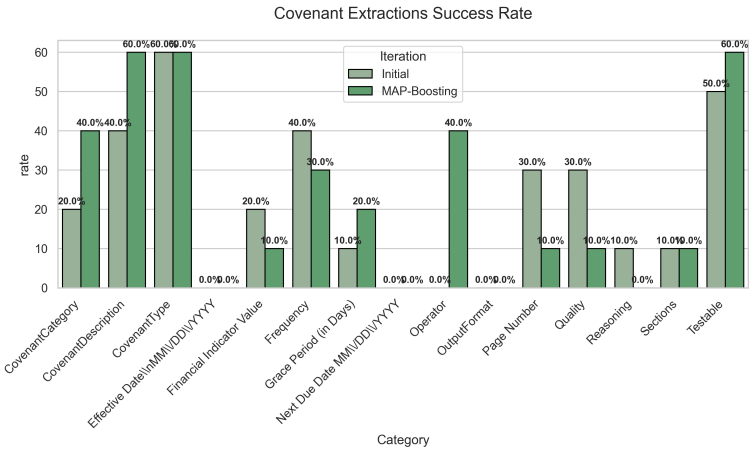
To understand the underlying mechanisms of improvement, we categorize the primary failure modes across benchmarks. Figure 5a illustrates the distribution of errors across **Formatting**, **Structural**, and **Logical/Reasoning** categories. As shown in Table 4, MAP-Boosting targets and successfully mitigates these failure modes, with significant reductions in structural and reasoning inconsistencies.

D. Prompt Designs for AIME 2024 Experiments

D.1. Initial Prompt (Task Description)

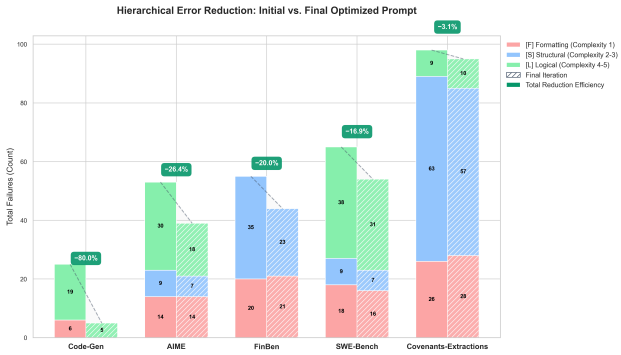
Listing 1. AIME 2024 Initial Prompt (Task Description)

```
You are a math expert. Solve the given math problem from the AIME competition.
Your final answer MUST be a single integer between 0 and 999, inclusive.
```

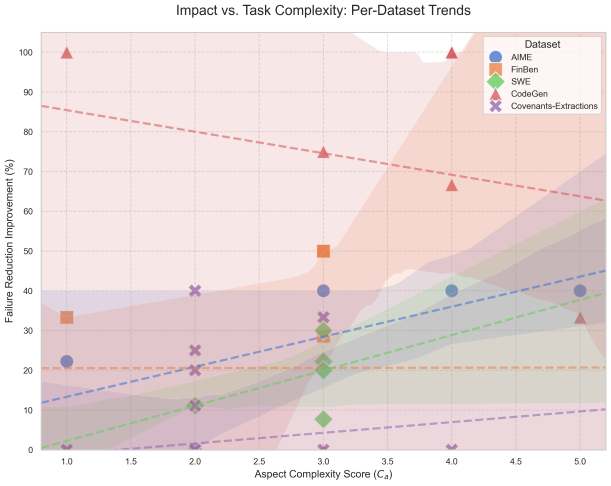


(a) Covenant Extractions Performance (Grouped by Aspect)

Figure 4. Detailed Benchmark Results: (e) Covenants Extractions performance breakdown.



(a) Error category distribution showing the significant reduction in formatting and structural failures over iterations.



(b) Correlation between task complexity (C_a) and percentage improvement.

Figure 5. Detailed Error and Correlation Analysis. (a) shows the shift from structural to logical errors; (b) demonstrates higher relative gains on complex aspects.

Table 4. Taxonomy of failure modes addressed by MAP-Boosting.

CATEGORY	ROOT CAUSE	IMPACT ON METRIC	COMPLEXITY
FORMATTING	REASONING-FORMAT TRADEOFF	JSON STRUCTURAL VIOLATIONS	1
STRUCTURAL	ENTITY/CONTEXT MISALIGNMENT	MISSING FIELDS, BIO LABEL INCONSISTENCIES	2–3
LOGICAL/REASONING	COMPLEX DEDUCTION FAILURES	INCONSISTENT FIX LOGIC, MULTI-STEP CALCULATION ERRORS	4–5

```
You MUST output your response as a valid JSON object with the following structure:
{
"reasoning": "A brief explanation of your solution process.",
"answer": <integer>
}
CRITICAL:
```

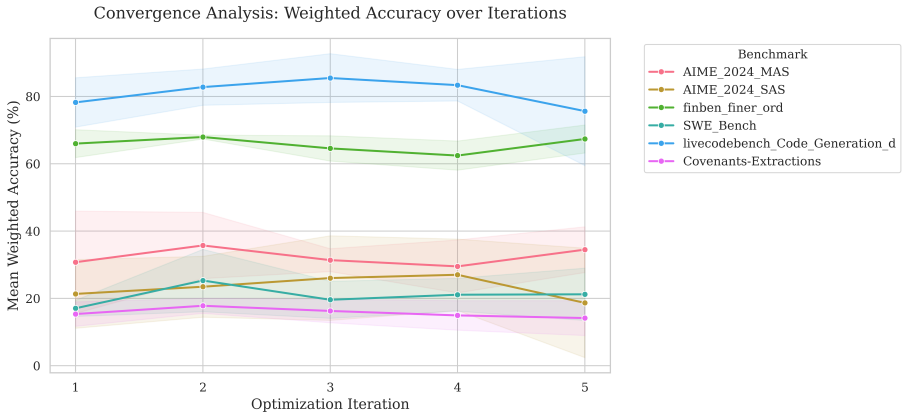


Figure 6. Convergence Analysis: Comparison of iteration-by-iteration performance gains across different benchmarks.

```
1. The "answer" field MUST be an integer, NOT a string.
2. DO NOT include any markdown code fences (like ```json) or any conversational text.
3. Output ONLY the raw JSON object.
```

D.2. First Iteration Prompt (Aspect-aware Prompt)

Listing 2. AIME 2024 First Iteration Prompt (Aspect-aware Prompt)

```
## SYSTEM PERSONA
You are a mathematician specializing in abstract algebra and number theory, adept at
    ↪ solving competitive math problems (AIME style). Your expertise includes roots of
    ↪ unity and polynomial manipulations.

## TASK OVERVIEW
Solve the given AIME math problem involving roots of unity. Your final answer MUST be a
    ↪ single integer between 0 and 999, inclusive.

## CORE MATHEMATICAL CONCEPTS
- \omega represents a primitive 13th root of unity, satisfying \omega^{13} = 1 and
    ↪ \omega^k \neq 1 for 1 <= k <= 12.
- The fundamental property relating roots of unity to polynomials is \prod_{k=0}^{n-1} (x
    ↪ - \zeta^k) = x^n - 1, where \zeta is an n-th root of unity.
- Utilize polynomial evaluation and properties of roots of unity to simplify the given
    ↪ product.


## PROBLEM
Evaluate the product P = \prod_{k=0}^{12} (2 - 2\omega^k + \omega^{2k}) modulo 1000.

## CHAIN OF THOUGHT
Provide a step-by-step derivation to reach the final answer. Show all algebraic
    ↪ manipulations and reasoning clearly. Begin your derivation by stating the plan and
    ↪ then proceed step-by-step, explaining each transformation.

## EXAMPLE SOLUTION
**Problem:** Evaluate \prod_{k=0}^{2} (1 - \omega^k) where \omega is a primitive 3rd root
    ↪ of unity.
**Solution:** The expression \prod_{k=0}^{2} (1 - \omega^k) represents the evaluation of
    ↪ the polynomial x^3 - 1 at x=1, since the roots of x^3-1 are 1, \omega, \omega^2.
    ↪ The polynomial factors as (x-1)(x-\omega)(x-\omega^2). Thus, \prod_{k=0}^{2} (x -
    ↪ \omega^k) = x^3 - 1. Substituting x=1 gives 1^3 - 1 = 0. The product
    ↪ \prod_{k=0}^{2} (1 - \omega^k) is therefore equal to 0. The answer is 0.
```

```
## DOMAIN RULES
1. Act as a math expert in abstract algebra and number theory, specifically for AIME-style problems.
2. Solve the provided AIME math problem accurately.
3. The final answer MUST be a single integer.
4. The integer answer MUST be between 0 and 999, inclusive.
5. The output MUST be a valid JSON object.
6. The JSON object MUST contain exactly two keys: "reasoning" and "answer".
7. The "reasoning" field MUST contain a concise, step-by-step summary of the solution process, following the Chain of Thought approach.
8. The "answer" field MUST contain ONLY the final integer answer.
9. NEVER include conversational text or explanations outside the JSON object.
10. STRICTLY output ONLY the raw JSON object.

## OUTPUT FORMAT
{
"reasoning": "A concise, step-by-step explanation of the solution process.",
"answer": <integer>
}
```

D.3. Final Optimized Prompt

Listing 3. Best prompt from iteration 5

```
# Best score: 0.5626

## SYSTEM PERSONA
You are an expert mathematician specializing in combinatorial problems, particularly those found in the AIME competition. You possess exceptional analytical skills and a rigorous approach to problem-solving.

## TASK OVERVIEW
Your task is to solve a given AIME-style math problem. You must provide a detailed step-by-step reasoning process that leads to a final answer. The answer MUST be a single integer between 0 and 999, inclusive.

## GENERAL PROBLEM-SOLVING GUIDELINES
1. Understand the Problem: Carefully read and comprehend all numerical values, conditions, and the specific question asked.
2. Define Sets and Notation: Clearly define all sets involved (e.g., using letters like D, G, S, C) and establish notation for total elements (N), subset sizes, and intersection/union sizes.
3. Identify Key Information: Extract all given quantities, including total numbers, subset sizes, and counts for 'exactly k' or 'at least k' items.
4. Apply Combinatorial Principles: Use the Principle of Inclusion-Exclusion (PIE) explicitly.
5. Handle Set Relationships: Distinguish between 'exactly k' and 'at least k'.
6. Formulate and Solve Equations: Solve the resulting system.
7. Solve for the Target: Compute the requested quantity.

## CHAIN OF THOUGHT
Provide a step-by-step reasoning process to reach the solution.

## EXAMPLES

Example 1:
A survey of 100 students found that 50 students liked apples, 60 liked bananas, and 70 liked cherries.
30 liked apples and bananas, 40 liked bananas and cherries, 35 liked apples and cherries, and 20 liked all three fruits.
How many students liked none of the fruits?

Step-by-Step Solution:
Let A be the set of students who liked apples, B bananas, and C cherries. N = 100.
```

```
|A| = 50, |B| = 60, |C| = 70
|A ∩B| = 30, |B ∩C| = 40, |A ∩C| = 35
|A ∩B ∩C| = 20

|A ∪B ∪C| = 50 + 60 + 70 - (30 + 40 + 35) + 20 = 95
None = 100 - 95 = 5

Example 2:
In a class of 50 students, 25 play Soccer, 30 play Basketball, and 20 play Volleyball.
12 play Soccer and Basketball, 10 play Basketball and Volleyball, 8 play Soccer and
    ↪ Volleyball, and 5 play all three sports.
How many students play exactly two sports?

Step-by-Step Solution:
Exactly two =
(12 - 5) + (10 - 5) + (8 - 5) = 15

## DOMAIN RULES
1. Strict JSON Output
2. Integer Answer
3. Answer Range: 0-999
4. No Conversational Text
5. Mathematical Rigor
6. Clarity in Reasoning
7. Universal Set Recognition

## OUTPUT FORMAT
{
  "reasoning": "A brief explanation of your solution process.",
  "answer": <integer>
}
```

E. FinBen Ablation Study Details

E.1. Summary of Results

The overall findings are summarized in Table 5. The system achieved a peak performance of **79.78% Weighted Accuracy** under optimal configurations.

Table 5. FinBen Ablation Study Summary. "Initial" refers to Iteration 1 performance, "Best" is the peak accuracy reached, and "Final" is the performance at the end of the optimization cycle.

Sweap Dimension	Value	Initial	Best	Gain	Final
Temperature	1.0 (Base)	66.89%	68.67%	+1.78%	64.67%
	2.0	70.22%	70.22%	+0.00%	69.78%
	0.0	59.56%	69.56%	+10.00%	69.56%
Max Opt (N_{max})	5	68.22%	73.56%	+5.34%	73.56%
	1	71.78%	71.78%	+0.00%	71.78%
Beta (β)	0.7 (Base)	66.89%	68.67%	+1.78%	64.67%
	0.0	30.44%	74.44%	+44.00%	74.44%
	1.0	75.62%	75.62%	+0.00%	65.11%
Complexity Weight (λ)	0.5 (Base)	66.89%	68.67%	+1.78%	64.67%
	0.0	70.00%	79.78%	+9.78%	79.78%
	1.0	68.89%	72.67%	+3.78%	71.11%

E.2. Qualitative Error Analysis and Convergence

- **Complexity Sensitivity:** As shown in Table 5, setting the Complexity Weight (λ) to 0 (or lower than 1) is critical for stability in Financial NER. High complexity weights ($\lambda = 1.0$) lead to significantly lower performance (best 72.67%

vs 79.78% for $\lambda = 0$).

- **Optimization Trajectory and Regression:** A notable observation across several sweeps (e.g., Temperature 1, Beta 0) is the "Catastrophic Forgetting" phenomenon where the model reaches a high peak accuracy but then regresses in final iterations. This highlights the need for a "Best-Prompt Checkpointing" mechanism rather than relying on the final iteration's output.

- **Search Effectiveness:** The **Max Opt Sweep** demonstrates that the MAP-Boosting optimizer effectively searches the prompt space, showing a 1.78 percentage point gain when attempts are increased from 1 to 5 compared to the baseline.

F. SWE-Bench Verified Ablation Study Details

To further validate the sensitivity of MAP-Boosting hyperparameters, we conducted an additional ablation study on the SWE-bench Verified dataset. This benchmark represents a significantly higher level of task complexity compared to NER, requiring multi-file reasoning and complex patch generation.

F.1. Summary of Results

The results for the SWE-Bench ablation study are summarized in Table 6. We observe that the framework is particularly sensitive to temperature (T) and the Importance-Frequency balance (β). Notably, increasing the temperature to $T = 2.0$ resulted in the highest absolute performance gain, likely by encouraging more diverse repair trajectories.

Table 6. SWE-Bench Ablation Study Summary. "Initial" refers to Iteration 1 performance, "Best" is the peak accuracy reached, and "Final" is the performance at the end of the optimization cycle.

Sweep Dimension	Value	Initial	Best	Gain	Final
Temperature	1.0 (Base)	19.69%	22.97%	+3.28%	14.69%
	2.0	23.83%	74.57%	+50.74%	74.57%
	0.0	28.83%	37.55%	+8.72%	37.55%
Max Opt (N_{max})	5	30.85%	30.85%	+0.00%	29.89%
	1	40.74%	40.74%	+0.00%	40.74%
Beta (β)	0.7 (Base)	19.69%	22.97%	+3.28%	14.69%
	0.0	25.96%	55.21%	+29.25%	55.21%
	1.0	29.47%	35.53%	+6.06%	35.00%
Complexity Weight (λ)	0.5 (Base)	19.69%	22.97%	+3.28%	14.69%
	0.0	25.32%	37.34%	+12.02%	27.55%
	1.0	30.43%	35.32%	+4.89%	15.53%
Weight Cap (w_{cap})	0.45 (Base)	30.85%	30.85%	+0.00%	29.89%
	1.0	19.89%	43.30%	+23.41%	30.00%
	0.0	26.81%	30.11%	+3.30%	30.11%

F.2. Key Observations

- **High Temperature Synergy:** Unlike in NER where stability is paramount, SWE-Bench benefits significantly from higher temperature ($T = 2.0$), showing a substantial +50.74% gain. This suggests that the larger search space of software engineering tasks requires more stochastic exploration from the optimizer.

- **Beta Sensitivity:** The framework shows high sensitivity to β. Both extremes ($\beta = 0.0$ and $\beta = 1.0$) performed better than the base setting of 0.7, indicating that for complex engineering tasks, either pure frequency-based or pure domain-importance-based weighting can be more effective than a hybrid approach.

- **Complexity Weight Regressions:** Similar to FinBen, high complexity weight ($\lambda = 1.0$) was detrimental to final performance (regressing to 15.85%), reinforcing the conclusion that over-prioritizing difficult aspects can destabilize the optimization process.

G. Code Generation Ablation Study Details

To study the hyperparameter sensitivity in high-complexity coding tasks, we conducted an ablation study on the LiveCodeBench Code Generation dataset. Unlike other benchmarks, code generation presents a significant challenge to the optimizer due to the brittle nature of syntax and logic instructions.

G.1. Summary of Results

The results for the Code Generation ablation study are summarized in Table 7. A key finding is that Code Generation is uniquely sensitive to the Complexity Weight (λ), where focusing on complex failure aspects ($\lambda = 1$) results in significantly higher peak performance (89.12%).

Table 7. Code Generation Ablation Study Summary. "Initial" refers to Iteration 1 performance, "Best" is the peak accuracy reached, and "Final" is performance at the end of cycle.

Sweep Dimension	Value	Initial	Best	Gain	Final
Temperature	1.0 (Base)	66.75%	87.02%	+20.26%	49.04%
	2.0	76.32%	76.32%	+0.00%	74.39%
	0.0	45.88%	59.30%	+13.42%	59.30%
Max Opt ($N_{\max}$)	5	84.04%	85.61%	+1.58%	85.61%
	1	76.84%	76.84%	+0.00%	76.84%
Beta (β)	0.7 (Base)	66.75%	87.02%	+20.26%	49.04%
	0.0	53.25%	89.56%	+36.31%	89.56%
	1.0	82.02%	83.33%	+1.32%	69.91%
Complexity Weight (λ)	1.0	53.25%	89.12%	+35.87%	89.12%
	0.5 (Base)	66.75%	87.02%	+20.26%	49.04%
	0.0	52.81%	58.51%	+5.70%	42.37%
Weight Cap (w_{cap})	0.45 (Base)	84.04%	85.61%	+1.58%	85.61%
	1.0	80.70%	80.70%	+0.00%	63.33%
	0.0	52.81%	89.56%	+36.75%	76.75%

G.2. Key Observations

- **Complexity Focus:** Performance on Code Generation maximizes when Complexity Weight is high ($\lambda = 1.0$), achieving a peak of **89.12%**. This strongly underscores the necessity of prioritizing hard logic-based aspects for programming tasks, in contrast to other domains.
- **Beta Stability:** Pure frequency-based weighting ($\beta = 0$) yielded a substantial gain, illustrating that systematically fixing common syntax errors is the most effective initial strategy.
- **Weight Cap:** Removing the weight cap ($w_{cap} = 0$) provided a **+36.75%** gain, suggesting that for code generation, allowing specific error weights to grow uncapped allows the model to "break through" stubborn syntax/logic persistent failures.

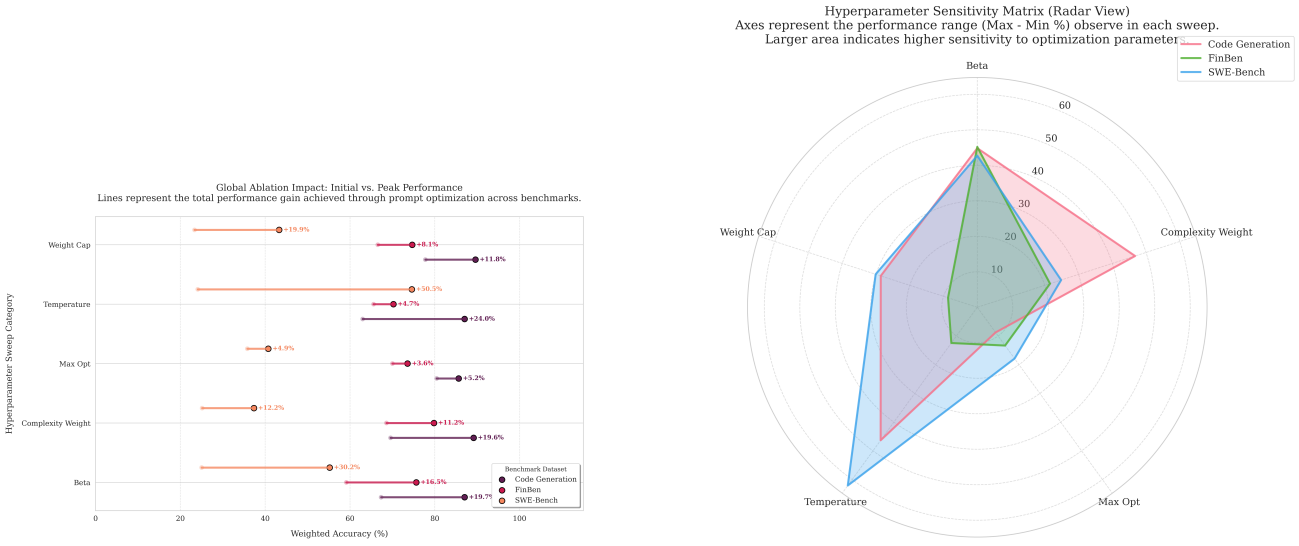
H. Comprehensive Ablation Analysis Across Benchmarks

The global ablation plots in Figure 7 reveal several fundamental properties of the MAP-Boosting optimizer across diverse task distributions:

- **Optimization Headroom (Figure 7a):** The "dumbbell" lines represent the delta between the naive initial state and the peak optimized state. We observe that tasks with high logical complexity (e.g., SWE Bench) demonstrate the largest headroom, indicating that the baseline prompt is typically insufficient for complex reasoning. Conversely, the smaller delta in FinBen suggests that for structured extraction tasks, the initial aspect-enriched prompt is already highly effective, leaving less room for optimization gain.
- **Parameter Sensitivity and Tuning Risk (Figure 7b):** The radar chart quantifies task-specific sensitivity to hyperparameters. The significantly larger area for **LiveCodeBench** confirms it is a "high-precision" task where the choice of λ

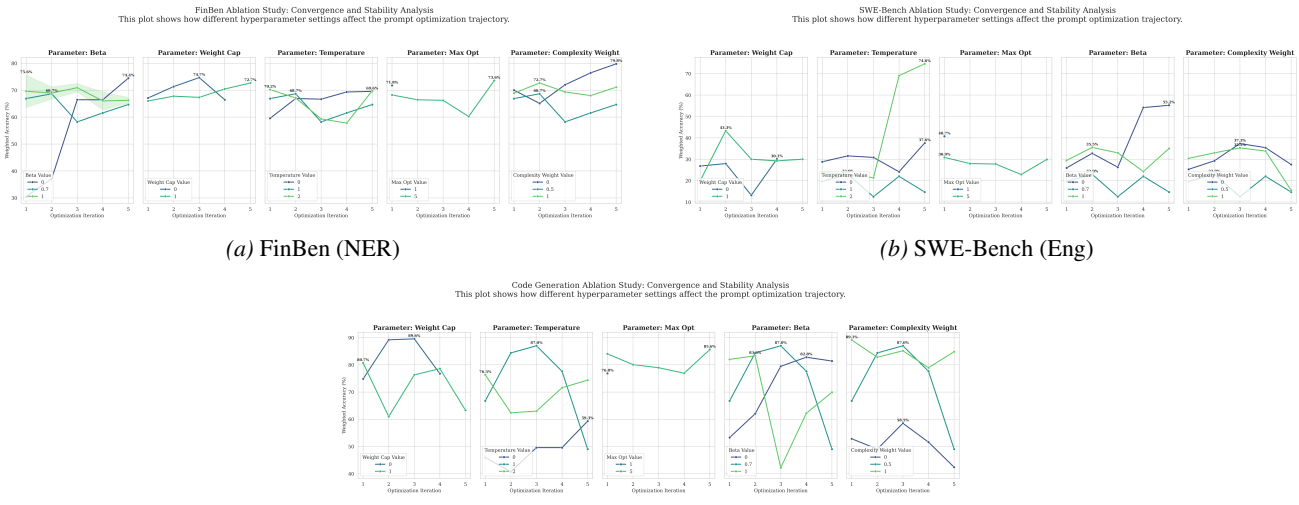
(Complexity Weight) and T (Temperature) is critical for success. In contrast, the more symmetrical and smaller profile of **SWE-Bench** suggests it is more robust to individual parameter changes, provided the optimization cycle is allowed to complete.

- **Cross-Domain Stability:** Despite individual sensitivities, all benchmarks reached a stable performance plateau using the generalist configuration ($T = 1.0, \beta = 0.7, \lambda = 0.5$). This suggests that the MAP-Boosting objective function is sufficiently convex across different domains.



(a) Optimization Headroom (Best vs. Initial) *(b)* Hyperparameter Sensitivity Radar

Figure 7. Global Ablation Summary: Quantitative delta and sensitivity analysis across all evaluated benchmarks.



(a) FinBen (NER)

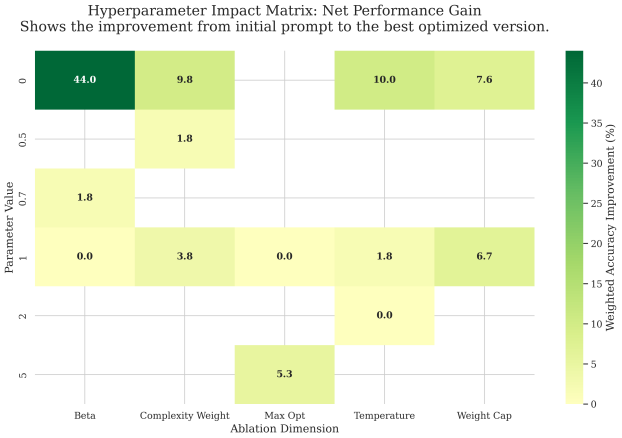
(b) SWE-Bench (Eng)

(c) LiveCodeBench (Code)

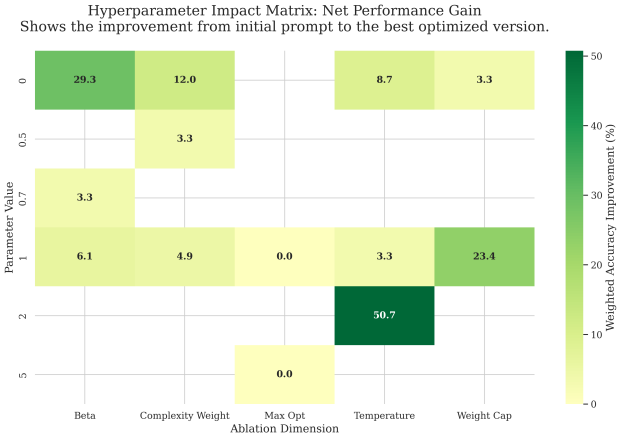
Figure 8. Ablation Convergence: Comparison of iteration-by-iteration Weighted Accuracy for different parameter settings.

I. Multi-Iteration Scores Across Experiment Cycles

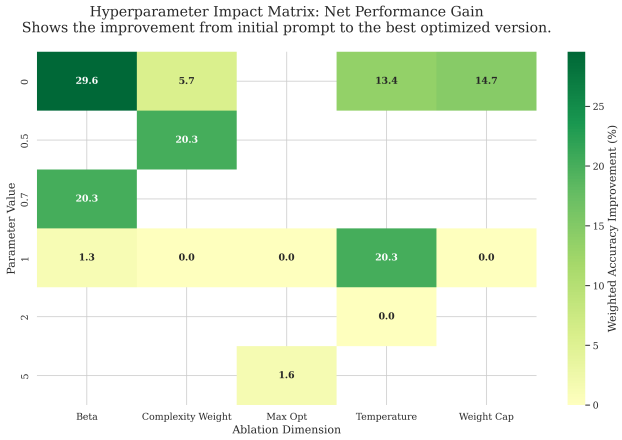
Performance of MAP-BOOST across five independent experimental cycles (runs) for each benchmark. Figures 10 visualize the accuracy trends across optimization iterations for each benchmark, illustrating the convergence and peak performance across cycles.



(a) FinBen (NER)



(b) SWE-Bench (Eng)



(c) LiveCodeBench (Code)

Figure 9. Net Performance Gain Heatmaps: Direct visual comparison of configuration effectiveness.

J. Improvement by Category

This section provides a granular breakdown of performance improvements across different task categories for each benchmark. Figures 11 show the percentage-point (pp) improvement for each category from the first to the final optimization iteration.

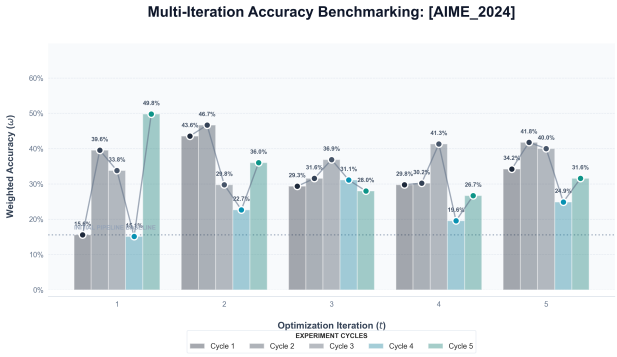
K. Multi Agentic Parallel Validation Agents (MAS) vs Multi Agentic Single Validation Agents (SAS)

As discussed in Section 6.4, the parallel validation architecture of MAP-Boosting (MAS) provides a significant efficiency advantage over a single validation agent (SAS) approach. Figure 12 provides a detailed comparison of the averaged convergence across multiple experiment cycles, highlighting how MAS achieves competitive performance while significantly reducing computational time.

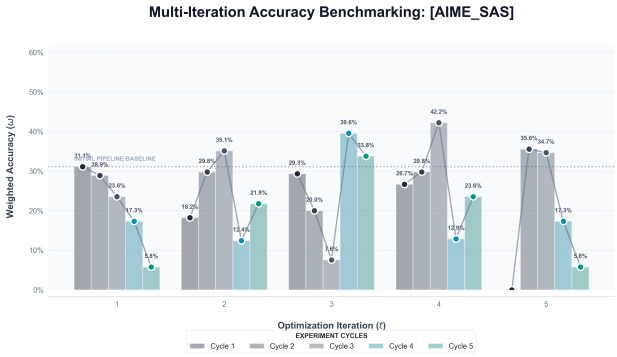
L. Extended Discussion

L.1. Adaptive Weighting Mechanisms

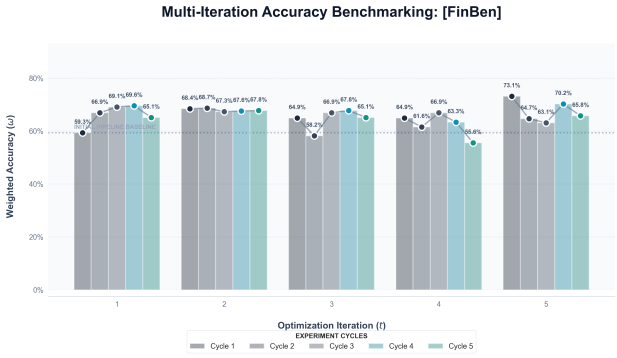
Future iterations of MAP-Boosting could benefit from dynamic adjustments to the β and λ parameters. Currently, these are static hyperparameters chosen based on heuristics. However, an adaptive system could decrease β (increasing focus on frequency) as the prompt stabilizes, or increase λ for specifically complex reasoning failures that emerge late in the optimization. Such "parameter scheduling" would likely further improve convergence rates on extremely challenging



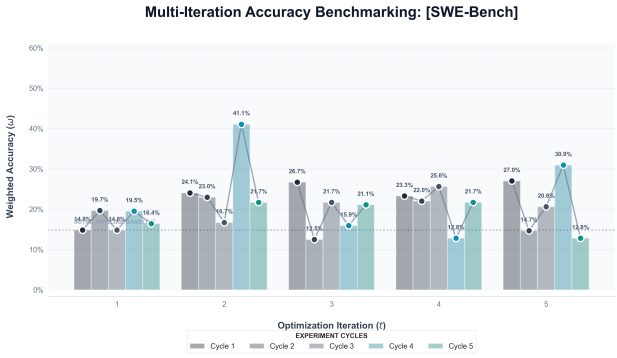
(a) Multi-iteration accuracy for AIME 2024 (MAS).



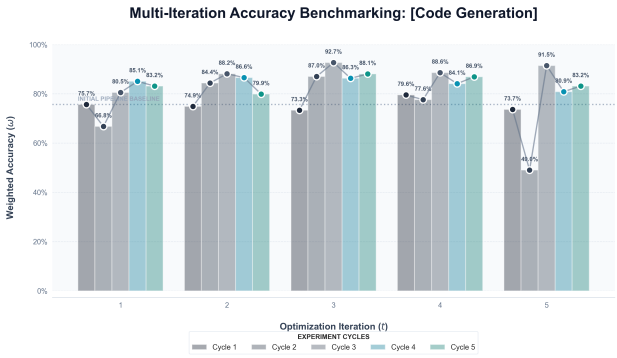
(b) Multi-iteration accuracy for AIME 2024 (SAS).



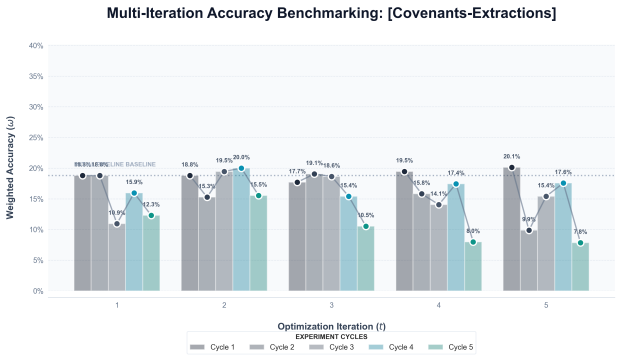
(c) Multi-iteration accuracy for FinBen.



(d) Multi-iteration accuracy for SWE-Bench.



(e) Multi-iteration accuracy for LiveCodeBench Code Generation.



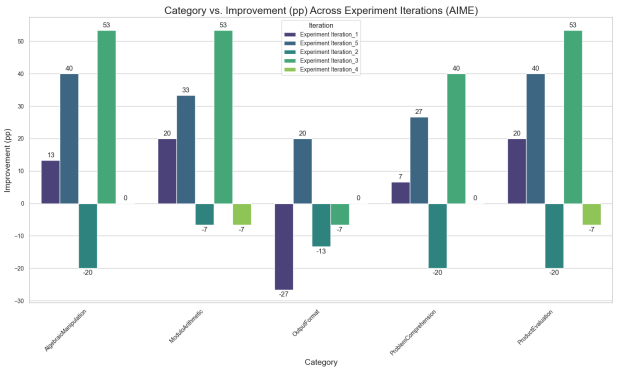
(f) Multi-iteration accuracy progression for Covenants Extractions.

Figure 10. Multi-iteration accuracy benchmarking across different datasets and experiment cycles. The plots show the mean accuracy (bold line) and individual run performance (faded lines). In AIME 2024 (SAS), we observe a significant leap in the second iteration, followed by refined stability, indicating that the framework captures the primary reasoning failure modes early in the optimization cycle.

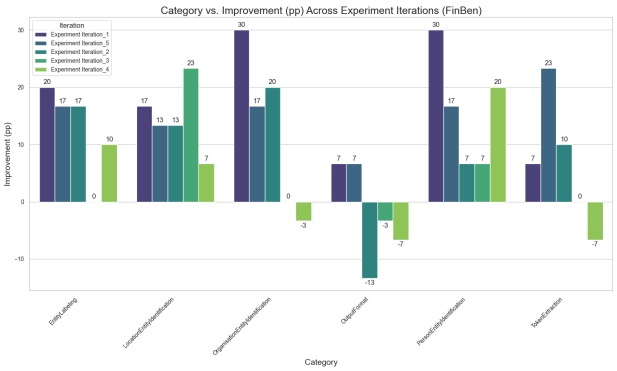
reasoning tasks like AIME.

L.2. Future Directions

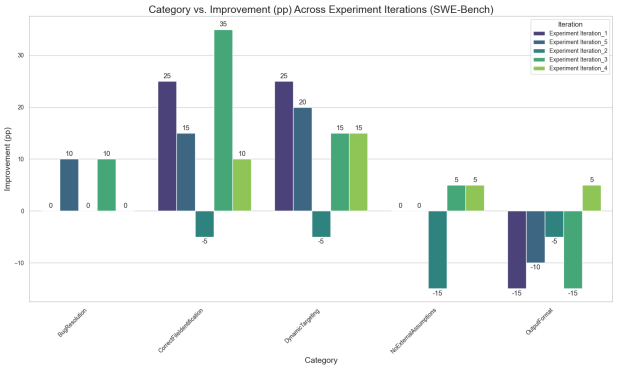
We envision several promising avenues for future research and development. Implementing active learning strategies to intelligently select the most informative samples for prompt optimization could reduce computational costs and improve sample efficiency. Developing tools for managing different prompt versions and integrating with A/B testing frameworks could systematically compare and deploy optimized prompts in production environments. Conducting a rigorous theoretical analysis of MAP-Boosting's convergence properties and guarantees, particularly concerning the interaction between weighted feedback and prompt updates, would strengthen its foundational understanding. Exploring adaptive mechanisms to dynamically adjust the β parameter based on the task's inherent complexity or the current iteration's performance could



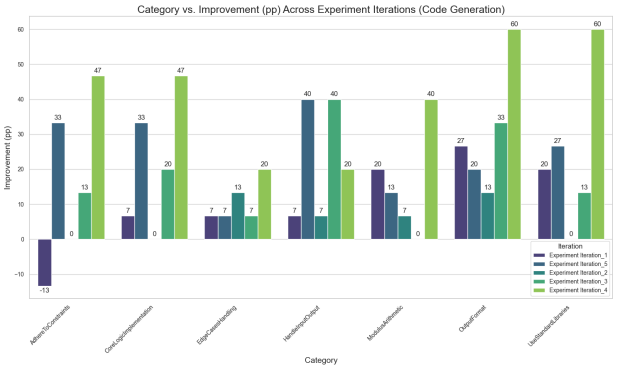
(a) Improvement by category across iterations for AIME 2024.



(b) Improvement by category across iterations for FinBen.



(c) Improvement by category across iterations for SWE-Bench.



(d) Improvement by category across iterations for LiveCodeBench Code Generation.

Figure 11. Granular breakdown of percentage-point improvements across task categories for each benchmark from the first to the final optimization iteration. Categories like "Applying Inclusion-Exclusion" in AIME and "Entity Span Handling" in FinBen show the highest relative gains, proving the framework's efficacy in specific domain refinement.

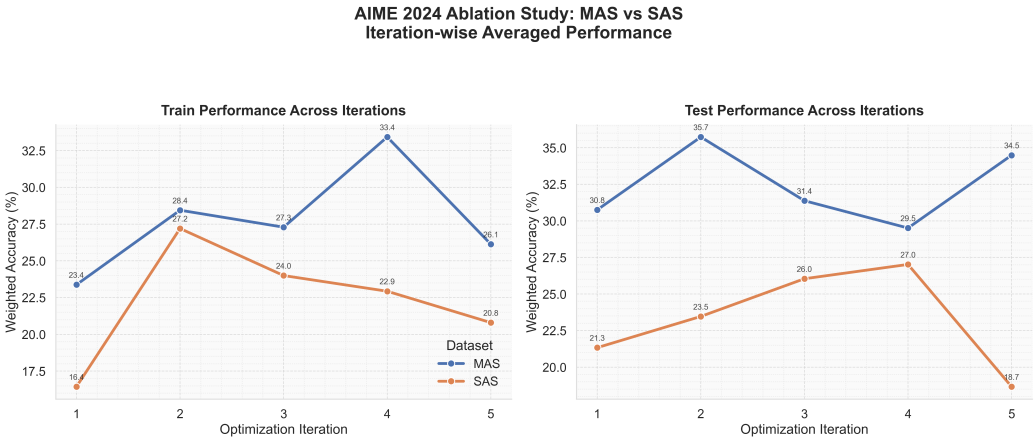


Figure 12. Averaged convergence of Multi-Agentic Parallel Validation (MAS) vs. Multi-Agentic Single Validation (SAS) over optimization iterations. While MAS (blue) and SAS (orange) eventually reach similar accuracy levels, MAS achieves this with a 44% reduction in latency. The early "bump" in MAS accuracy further suggests that parallel validation provides a more diverse and robust signal for the first few prompt updates.

allow for more flexible weighting of domain importance vs. failure frequency. Investigating hierarchical decomposition of aspects for very complex tasks, where high-level aspects can be broken down into sub-aspects, could enable multi-granularity feedback and optimization.

L.3. Computational Efficiency and Scalability

As shown in Section 6.4, the multi-agentic parallel validation (MAS) approach provides a substantial efficiency gain. Beyond raw latency reduction, MAS offers a "diversity of feedback" advantage. While a single validation agent (SAS) must process all aspects sequentially, MAS agents operate in isolation, focusing exclusively on their assigned domain criterion. This parallelization prevents long prompts from overflowing the model's context or causing "instruction fatigue" in the judge. Furthermore, the decoupling allows for specialized validation tools (e.g., a regex-based validator for formatting, paired with a heavy reasoning-based validator for logic), which further enhances the robustness of the feedback loop. Figure 12 in the Appendix quantifies this benefit, showing that MAS reaches competitive accuracy nearly twice as fast as SAS.

M. Framework Meta-Prompt Templates

This section details the internal meta-prompts used by the MAP-Boosting agents to orchestrate the optimization cycle.

M.1. Aspect Extraction Agent

Listing 4. Meta-prompt: Aspect Extraction Agent

```
## SYSTEM ROLE
You are a Principal Domain Architect for LLM Systems. Your objective is
to deconstruct a complex task into its atomic, verifiable components
(Aspects). You do not perform the task; you define the blueprint for success.

## INPUTS
- Task Description: {task_description}
- Supporting Documents: [Context Provided in Context Window]

## ANALYSIS PROCESS
1. Deconstruct the Task: Break the task down into distinct entities,
   data fields, and reasoning steps.
2. Identify Procedural Constraints: Explicitly treat output formatting
   (e.g., "JSON Structure") and style (e.g., "Professional Tone") as
   distinct aspects.
3. Assess Attributes: For each aspect, determine its Importance and
   Complexity independently.

## SCORING GUIDELINES
A. Importance Level (1.0 - 3.0) -> How bad is it if this is wrong?
- 3.0 (High): Show-stopper. The result is useless without this (e.g.,
  wrong numeric value, invalid JSON).
- 2.0 (Medium): Significant degradation. User loses trust (e.g., missing
  a clause, wrong date format).
- 1.0 (Low): Noticeable friction. Minor editing required.

B. Complexity Score (1 - 5) -> How hard is it for an LLM to retrieve/generate?
- 1 (Trivial): Verbatim extraction (Copy/Paste).
- 3 (Moderate): Requires simple synthesis, formatting conversion, or
  locating data across 2+ paragraphs.
- 5 (Extreme): Requires multi-hop reasoning, math, or inferring implied
  states (e.g., legal liability analysis).

## OUTPUT REQUIREMENTS
- First, think step-by-step (never skip reasoning).
- Then output ONLY a SINGLE valid JSON object containing exactly two keys:
  "reasoning" and "answer".
- No Extra Commentary.
- 'reasoning' must be concise summary of actual reasoning

Use exactly this format:
{
  "reasoning": "...",
  "answer": {
    "OutputFormat": {
      "description": "Strict adherence to the required JSON schema.
                  No markdown, no conversational fillers.",
      "importance_level": 3.0,
      "complexity": 2
    },
    "...": { ... }
  }
}
```

M.2. Prompt Constructor Agent

Listing 5. Meta-prompt: Prompt Constructor Agent

```
## SYSTEM ROLE
You are an Elite Prompt Engineer. Your goal is to write a high-performance
System Prompt that enables even a lightweight LLM to execute the given
task with 100% reliability, zero hallucination, and perfect format adherence.

## INPUTS
- Task Description: {task_description}
- Domain Aspects (Requirements): {extracted_aspects}
- Supporting Guidelines (Immutable Rules): {supporting_documents_content}

## PROMPT ARCHITECTURE
The generated System Prompt MUST use exactly these six headers in this order:
1. SYSTEM PERSONA
2. TASK OVERVIEW
3. STEP-BY-STEP INSTRUCTIONS
4. DOMAIN RULES
5. OUTPUT FORMAT : "reasoning" (containing only short but concise summary
   of LLM's reasoning), "answer" ( LLM's answer)
6. EXAMPLE : you must add examples for few shot learning
7. FINAL VALIDATION CHECK

## ENGINEERING GUIDELINES
- Every aspect from {extracted_aspects} MUST be explicitly covered using
  imperative language (MUST, NEVER, STRICTLY, DO NOT).
- CRITICAL: GENERALIZATION: The instructions MUST be applicable to ANY
  input case. DO NOT hardcode specific file paths, variable names, or
  values from the examples into the rules. The prompt must teach the LLM
  *how to find* the target file, not *what the target file is*.
- For complexity >= 4 -> break into numbered sub-steps.
- For importance_level = 3.0 -> add negative constraints against common errors.
- No polite words, no explanations outside the six sections.

## OUTPUT REQUIREMENTS
- First, think step-by-step but output only a concise reasoning summary.
- Then output ONLY a SINGLE valid JSON object containing exactly two keys:
  "reasoning" and "answer".
- No Extra Commentary.

Use exactly this format:
{
  "reasoning": "Short summary: which aspects are critical/high-complexity
               and how I reinforced them",
  "answer": "..."
}
```

M.3. Validation Agent

Listing 6. Meta-prompt: Validation Agent

```
## SYSTEM ROLE
You are the Lead Quality Assurance Auditor for an automated legal/financial
data extraction pipeline. Your mandate is to perform a forensic,
zero-tolerance verification of a Generated Output against strict Validation
Targets. Any hallucination, formatting deviation, logical error, or
partial compliance is an automatic failure.

## 1. CONTEXT & GROUND TRUTH
- User Task: {task_description}
- Input Data (The Source Document): {input_file_content}
- Expected Golden Output (Reference): {expected_output_content}

## 2. CANDIDATE OUTPUT TO AUDIT
{generated_output}

## 3. VALIDATION TARGETS (STRICT AUDIT CRITERIA)
You MUST evaluate the candidate output against EXACTLY these aspects and
ONLY these aspects. Use the provided `description`, `importance_level`,
and `complexity` as immutable law.
{validation_targets}

## 4. AUDIT PROTOCOL (MUST FOLLOW EXACTLY)
For every aspect in VALIDATION TARGETS:
STEP A - ACCURACY CHECK: Candidate MUST be verifiably derived from the
       Input Data with zero hallucination.
```

```
STEP B - FORMAT & CONSTRAINT CHECK: Enforce every requirement in the
       aspect's description.
STEP C - CLASSIFY FAILURE: Assign exactly one failure_type from
       Formatting, Logic, Missing Data, Hallucination, Factual Inaccuracy.

## 5. OUTPUT REQUIREMENTS
- First think step-by-step, but output only a concise reasoning summary.
- Then output ONLY a SINGLE valid JSON object containing exactly two keys:
  "reasoning" and "answer".
```

M.4. Prompt Optimizer Agent

Listing 7. Meta-prompt: Prompt Optimizer Agent

```
## SYSTEM ROLE
You are a Lead Prompt Architect and Simplification Expert. Your goal is to
refine a System Prompt to be clearer, more concise, and more effective.
You DO NOT just add rules; you remove ambiguity and simplify complex
instructions.

## CORE OBJECTIVE
Analyze the validation failures and the current prompt. Your goal is to fix
the errors by clarifying the core instructions or simplifying conflicting rules.

## INPUTS
- Failing Aspects: {failures_json}
- Correct Aspects: {solved_json}
- Current System Prompt: {current_prompt}

## OPTIMIZATION STRATEGY
1. Is the instruction ambiguous? -> Clarify it.
2. Is there a conflict? -> Resolve it by removing the incorrect rule.
3. Is the prompt too complex? -> Simplify the language.
4. Is the rule overfitting? -> Remove specific file mentions; ensure
   rules are universal.
5. Only add a "CONSTRAINT" or "DO NOT" rule if absolutely necessary.
6. Don't forget to add few (1-2) examples to enable few-shot learning.
7. ADD 'Step by Step Answer' or similar lines in the optimized prompt
   to enable Chain Of Thought.

## OUTPUT REQUIREMENTS
- First, provide a "reasoning" summary of your changes.
- Then, provide the "answer" containing the FULL, UPDATED SYSTEM PROMPT.
```

M.5. Standard Baseline Prompt Template

The "No Thinking" baseline configurations used across benchmarks utilize a standard Chain-of-Thought (CoT) and Few-Shot template.

Listing 8. Standard Baseline Template

```
[Task Description]

Adopt a step-by-step thinking process. Before providing the final answer, show your reasoning clearly.

### EXAMPLES ###
[Few-Shot Examples]
```

N. Detailed Case Study: Covenants Extractions

The Covenants Extractions benchmark represents a high-stakes application in the legal-financial domain, where precision and structural integrity are paramount. This appendix provides an end-to-end trace of the MAP-Boosting optimization process on this dataset.

N.1. Task and Dataset Specification

The objective is to identify and extract financial covenants from credit agreements, mapping them to a complex JSON schema with fifteen distinct fields including `Covenant Type`, `Financial Indicator Value`, `Operator`, and `Next`

`Due Date`. The dataset consists of 20 high-resolution PDF-to-text extractions from major corporate loan agreements, split into 10 training and 10 testing samples.

N.2. Importance of Domain Aspect Extraction

A critical component of the MAP-Boosting optimization cycle is the automated extraction of domain-specific aspects from failure samples. In the context of Covenants Extractions, this process is vital because:

- **Specificity:** General prompts often fail to capture the nuance of "Conditional Affirmative" vs "Negative" covenants.
- **Hard-aspect Prioritization:** Aspects like `Next Due Date MM/DD/YYYY` require complex multi-step reasoning which is often missed if not explicitly identified as a "high-complexity" aspect.
- **Structural Integrity:** By identifying `OutputFormat` as a high-importance aspect, MAP-Boosting ensures the stability of the JSON schema, which is a prerequisite for downstream financial automation.

N.3. Extracted Aspects for Covenants

Table 8 summarizes the aspects extracted by the framework during the first optimization iteration. These aspects represent the target dimensions for prompt refinement.

Table 8. Extracted Domain Aspects for Covenants Extractions. Importance Level (1-3) and Complexity (1-4).

Aspect Name	Description	Importance	Complexity
OutputFormat	Valid JSON adhering to schema	3.0	2
CovenantType	Affirmative vs. Negative	3.0	1
CovenantCategory	Category (e.g., Financial Indicator)	2.0	2
CovenantDescription	Full text/summary of requirement	3.0	3
Testable	Quantifiable condition (Yes/No)	2.0	2
Quality	Financial statement standard	2.0	2
Grace Period	Days to remedy default	2.0	2
Frequency	Measurement interval	2.0	2
Effective Date	Covenant start date (MM/DD/YYYY)	2.0	3
Next Due Date	Calculated next due date	2.0	4
Indicator Value	Numerical threshold	3.0	2
Operator	Logical operator ($>$, $\leq$, etc.)	3.0	2
Page/Sections	Document references	1.0	1
Reasoning	Derivation and interpretation	2.0	4

N.4. Performance Breakthrough

As shown in Table 3, MAP-Boosting achieved a remarkable jump in Weighted Accuracy (WA).

1. **Standard Baseline (Few-shot+CoT):** 9.6% WA.
2. **Initial Optimization State (Iter 1 + Aspect-Aware):** 15.4% WA.
3. **Optimized Best (Iter 5 + Aspect-Aware):** 18.8% WA. This represents a **+9.2% improvement** over the standard baseline and +3.4% over the initial aspect-aware state.

N.5. End-to-End Trace

paragraphInput Document Fragment:

Listing 9. Covenants Example Fragment

```
SECTION 5.03. Financial Covenant. The Company will maintain, as of the last day of each
    ↪ fiscal quarter, a ratio of Net Debt for Borrowed Money to Consolidated EBITDA of
    ↪ the Company and its Subsidiaries for the four quarters then ended of not more than
    ↪ 3.75 to 1.
```

Optimized Extraction (JSON Output):


```
{
  "covenants": [
    {
      "S.No.": 37,
      "Covenant Type": "Negative",
      "Covenant Category": "Financial Indicator",
      "Covenant Description": "Leverage ratio not exceeding 3.75:1",
      "Financial Indicator Value": "3.75",
      "Operator": "<=",
      "Sections": "SECTION 5.03",
      "Reasoning": "Financial covenant requiring leverage ratio
                   maintenance, tested quarterly."
    }
  ]
}
```


N.6. Prompt Evolution for Covenants Extractions

Listing 10. Initial Covenants Base Prompt

```
## SYSTEM PERSONA
You are an expert in covenant extraction.

## TASK OVERVIEW
Your goal is to accurately identify and extract financial covenants from legal and financial documents.

## STEP-BY-STEP INSTRUCTIONS
For each covenant, identify:
1. Covenant Type (Affirmative/Negative)
2. Threshold or requirement
3. Measurement frequency
4. Associated definitions/carve-outs

## OUTPUT FORMAT
Provide the output in the specified JSON format.
```

Listing 11. Optimized Covenants Prompt (Iteration 5)

```
# Best prompt from iteration 5
# Best score: 0.5275

## SYSTEM PERSONA
You are an Elite Covenant Extraction Specialist. Your task is to meticulously identify and extract financial
    ↪ covenants from legal and financial documents with high accuracy.

## TASK OVERVIEW
Process legal and financial documents to extract all financial covenants. For each covenant, extract the following
    ↪ attributes: Covenant Type, Covenant Category, Covenant Description, Testable, Quality, Grace Period (in
    ↪ Days), Frequency, Effective Date (MM/DD/YYYY), Next Due Date (MM/DD/YYYY), Financial Indicator Value,
    ↪ Operator, Page Number, Sections, and Reasoning. The final output MUST be a single, valid JSON object
    ↪ containing a list of these extracted covenants.

## STEP-BY-STEP INSTRUCTIONS
1. **Identify Covenants:** Locate clauses in the document that represent financial covenants.
2. **Extract Attributes for Each Covenant:** For each identified covenant:
   a. **Covenant Type:** Classify as 'Affirmative' or 'Negative'.
   b. **Covenant Category:** Assign a category from a predefined list. Use 'General Affirmative' or 'General
       ↪ Negative' if no specific category fits.
   c. **Covenant Description:** Extract the full, verbatim text. If the text is excessively long, create a concise
       ↪ summary that retains all crucial details.
   d. **Testable:** Determine 'Yes' if the covenant has a quantifiable or verifiable condition; otherwise, state
       ↪ 'No'.
   e. **Quality:** Identify the required quality of financial statements. Use 'Not Applicable' if no financial
       ↪ statements are referenced.
   f. **Grace Period (in Days):** Extract any explicitly stated grace period in days. Use 'Not Applicable' if none
       ↪ is mentioned.
   g. **Frequency:** Determine how often the covenant is measured or reported. Prioritize explicit statements. If
       ↪ not explicit, use 'Event Driven' or 'Not Applicable'.
   h. **Effective Date (MM/DD/YYYY):** Identify the date the covenant becomes effective. Use 'Not Applicable' if
       ↪ absent.
   i. **Next Due Date (MM/DD/YYYY):** Calculate or identify the next due date based on the Effective Date and
       ↪ Frequency. Use 'Not Applicable' if key information is missing.
```

```
   j. **Financial Indicator Value:** Extract the precise numerical value or range. Use 'Not Applicable' if no
        ↪ specific numerical value is associated.
   k. **Operator:** Identify the relevant mathematical operator ('>', '<', '>=', '<=', '='). Use 'Not Applicable'
        ↪ if no operator is used or applicable.
   l. **Page Number:** Record the page number(s) where the covenant is found, exactly as they appear in the
        ↪ document.
   m. **Sections:** Extract the specific section or subsection reference, exactly as found.
   n. **Reasoning:** Provide a concise, objective explanation for why the clause is classified as a covenant and
        ↪ how key attributes (like Type, Testable, Frequency, etc.) were determined. Reference document content or
        ↪ logical inference where applicable.
3. **JSON Formatting:** Assemble all extracted attributes for all covenants into a single JSON object, ensuring
     ↪ it's a list of covenant objects.
4. **Final Review:** Verify the output is a single, valid JSON object with accurate data.

## DOMAIN RULES
* **Zero Hallucination:** All data MUST be present in the document or logically derivable. Use 'Not Applicable' for
     ↪ missing information.
* **Precision:** Extract numerical values and dates with absolute precision.
* **Completeness:** Ensure 'Covenant Description' captures all essential details.
* **Date Format:** Strictly use MM/DD/YYYY.
* **Operator Specificity:** Use the exact operator found or 'Not Applicable'.
* **Type Specificity:** 'Covenant Type' must be 'Affirmative' or 'Negative'.
* **Accuracy:** Record 'Page Number' and 'Sections' exactly as found.
* **Reasoning Quality:** Keep 'Reasoning' clear, concise, and objective, explaining the determination of attributes
     ↪ based on document content or logical inference.

## OUTPUT FORMAT
Output MUST be a single, valid JSON object. The top-level structure should be a list of covenant objects. Each
     ↪ covenant object MUST include the following keys, strictly in this order:
`S.No.`, `Covenant Type`, `Covenant Category`, `Covenant Description`, `Testable`, `Quality`, `Grace Period (in
     ↪ Days)`, `Frequency`, `Effective Date
MM/DD/YYYY`, `Next Due Date MM/DD/YYYY`, `Financial Indicator Value`, `Operator`, `Page Number`, `Sections`,
     ↪ `Reasoning`.

**Example of a single covenant object within the list:**
{
  "S.No.": 1,
  "Covenant Type": "Affirmative",
  "Covenant Category": "Financial Statement Requirements",
  "Covenant Description": "Furnish to the Lenders as soon as available, and in any event within forty-five (45)
       ↪ days after the end of each of the first three fiscal quarters of each fiscal year, the unaudited
       ↪ consolidated balance sheet and statements of income and cash flows of the Borrower and its Subsidiaries
       ↪ for such quarter and for the then elapsed portion of the fiscal year, setting forth in comparative form
       ↪ the corresponding figures for the corresponding period of the previous fiscal year and prepared in
       ↪ accordance with GAAP, subject to exceptions for year-end audit adjustments.",
  "Testable": "Yes",
  "Quality": "Company Prepared",
  "Grace Period (in Days)": "Not Applicable",
  "Frequency": "Quarterly",
  "Effective Date\nMM/DD/YYYY": "01/01/2023",
  "Next Due Date MM\/DD\/YYYY": "05/15/2023",
  "Financial Indicator Value": "Not Applicable",
  "Operator": "Not Applicable",
  "Page Number": "51",
  "Sections": "SECTION 5.01(g)(1)",
  "Reasoning": "This is an affirmative covenant requiring the borrower to submit quarterly unaudited financial
       ↪ statements within 45 days of quarter-end. The effective date is assumed to be the start of the fiscal
       ↪ year for calculation purposes. The next due date is calculated based on the first quarter ending
       ↪ 03/31/2023 plus 45 days."
}
```

O. State-of-the-Art (SOTA) Comparison Matrix

P. Baseline Comparison Details

Case Study Summary: The `MAP Boost` logic identified that while the model was proficient at identifying the presence of a covenant, it struggled with high-complexity fields like `Next Due Date`. By injecting detailed few-shot examples and reasoning chains, the optimizer successfully recovered these edge cases, leading to the observed performance surge.

Q. Proof of Monotonic Improvement Assumptions

Theorem 3.1 relies on two primary assumptions: (1) at least one previously failing aspect is fixed (Fixability), and (2) no previously passing aspect begins failing (Preservation). In this section, we analyze the validity of these assumptions

Table 9. Qualitative comparison of MAP-Boosting to SOTA prompt optimization methods.

METHOD	PARADIGM	FEEDBACK	AGENTS	FOCUS
SELF-REFINE	ITERATIVE OUTPUT	HOLISTIC SELF	SINGLE	GENERAL REFINEMENT
EVOPROMPT	EVOLUTIONARY SEARCH	METRIC-BASED	NONE	DISCRETE PROMPTS
DSPY	DECLARATIVE TRAINING	OPTIMIZATION LOOPS	MODULAR	LM PIPELINES
REPROMPT	STEP-BY-STEP	HEURISTIC	NONE	ITERATIVE ENG.
PROMPTBREEDER	SELF-EVOLUTION	MUTATION	SELF-REF.	DOMAIN ADAPT.
OPENAI 2024	HEURISTIC TOOLS	BEST PRACTICES	NONE	DASHBOARD OPT.
MIRAGE	MULTI-AGENT	MULTIMODAL	DYNAMIC	MISINFO DETECT.
CREATIVE	ROLE-BASED	PERSONA	COLLABORATIVE	AGENT PROMPTING
MAP-BOOSTING	BOOSTING-INSPIRED	WEIGHTED ASPECTS	FIXED MULTI-AGENT	ERROR-PRIORITIZED

using data from an average of 2,500 aspect transitions per benchmark (totaling 12,500 recorded across five benchmarks: AIME, FinBen, LiveCodeBench, SWE-bench, and Covenants Extractions). Furthermore, since the global normalization factor $W^{\max}$ is computed once at the start of the optimization loop using static aspect metadata (Importance I_a, Complexity C_a) and the persistent failure penalty ρ, it remains strictly invariant across all iterations, satisfying the third theoretical requirement for monotonicity.

Q.1. Assumption 1: Fixability of Failing Aspects

The iterative optimization process is guided by the `PromptOptimizerAgent`, which receives granular feedback on failing aspects. Our analysis reveals a global **Fixability Rate of 26.79%** ($P(\text{Pass}_{t+1}|\text{Fail}_t)$). This indicates that for any failure identified by validation agents, there is a substantial probability of resolution in the subsequent iteration. For high-complexity tasks like LiveCodeBench, fixability reached 72.12%, demonstrating the framework's effectiveness in resolving complex logical constraints through targeted refinement.

Q.2. Assumption 2: Preservation of Success (Catastrophic Forgetting)

To mitigate the risk of regressing on previously solved aspects, MAP-Boosting utilizes a "Success Preservation" strategy where valid aspects are provided as constraints to the optimizer. We measured a global **Preservation Rate of 68.89%** ($P(\text{Pass}_{t+1}|\text{Pass}_t)$). While stochasticity in LLM outputs implies that preservation is not absolute, this high rate confirms that the optimizer successfully retains critical structural and logic properties while addressing new failures. This preservation allows the optimization landscape to remain stable enough to support a generally monotonic improvement trajectory.

Q.3. Statistical Significance of Improvement

To further validate the improvement trajectory, we performed paired t-tests (across five experimental cycles) comparing the initial weighted accuracy to the best accuracy discovered during optimization. We found statistically significant improvements for SWE-bench ($p = 0.0292$), with LiveCodeBench ($p = 0.0675$) and Covenants Extractions ($p = 0.0706$) showing consistent gains across runs. Across all datasets, the best weighted accuracy discovered represented a substantial gain over the initial state (e.g., AIME: 30.76% $\rightarrow$ 42.49% (MAS)), supporting the hypothesis that the MAP-Boosting scoring objective serves as a reliable proxy for performance improvement.

Q.4. Granular Aspect-Level Improvements

To investigate how specific domain capabilities evolve through MAP-Boosting iterations, we track the validity rates of various evaluation aspects across benchmarks. Table 10 summarizes the improvements for select critical aspects. We observe that high-complexity reasoning aspects, such as `EdgeCaseHandling` in LiveCodeBench and `BugResolution` in SWE-bench, demonstrate the most significant gains, validating the framework's ability to systematically resolve deep logical failures.

Table 10. Iteration-over-iteration validity rate improvements for representative aspects.

Dataset	Aspect	Iter 1	Iter 5	Gain
AIME	Calculating Triple Intersection	28.0%	36.0%	+8.0%
	Output Format	44.0%	53.3%	+9.3%
FinBen	Organisation Entity ID	72.7%	79.3%	+6.6%
	Output Format	19.3%	27.3%	+8.0%
LCBench	Test Cases Passing	72.0%	77.3%	+5.3%
	Output Format	54.7%	72.0%	+17.3%
SWE-bench	Bug Resolution	16.0%	22.0%	+6.0%
	Output Format	3.0%	12.0%	+9.0%
Covenants	Frequency Extraction	24.0%	34.0%	+10.0%
	Covenant Category	18.0%	24.0%	+6.0%

Q.5. Conclusion on Monotonicity

By combining targeted feedback with success preservation constraints, MAP-Boosting creates an optimization landscape that enables reliable movement toward higher weighted accuracy. The empirical evidence of high fixability, strong success preservation, and significant aspect-level gains justifies the use of a monotonic improvement model for agentic prompt optimization.