FLOWCOMPILE: AN OPTIMIZING COMPILER FOR STRUCTURED LLM WORKFLOWS

Anonymous authors
Paper under double-blind review

ABSTRACT

Structured LLM workflows, in which specialized LLM sub-agents are executed according to a predefined execution graph, have become a powerful abstraction for solving complex tasks. Optimizing such workflows, *i.e.*, selecting configurations for each sub-agent to balance accuracy and latency, is fundamentally challenging due to the combinatorial design space over model choices, reasoning budgets, and workflow structures. Routing-based cost-aware methods select a configuration at inference time for each query according to an accuracy–latency objective. We argue that, beyond runtime routing, structured LLM workflows can also be optimized from a *compilation* perspective: before deployment, the system can globally explore the workflow design space and construct a reusable set of workflow-level configurations spanning diverse accuracy–latency trade-offs. Drawing inspiration from machine learning compilers, we introduce **FlowCompile**, a structured LLM workflow compiler that performs compile-time design space exploration to identify such a high-quality, reusable trade-off set. FlowCompile decomposes a workflow into sub-agents, profiles each sub-agent under diverse configurations, and composes these measurements through a structure-aware proxy to estimate workflow-level accuracy and latency. It then identifies a diverse set of high-quality configurations in a single compile-time pass, without retraining or online adaptation. Experiments across diverse workflows and challenging benchmarks show that FlowCompile consistently outperforms heuristically optimized workflow configurations and routing-based baselines by a large margin, delivering up to $\mathbf{6.4}\times$ speedup while maintaining strong task performance. Furthermore, the compiled configuration set serves as a reusable optimization artifact, enabling flexible deployment under varying runtime preferences and naturally supporting downstream selection or routing for additional gains.

1 INTRODUCTION

Recent advances in machine learning compilers, such as TVM (Chen et al., 2018a), Glow (Rotem et al., 2018), and XLA (Sabne, 2020), have enabled efficient optimization of neural networks, including large language models (LLMs). TVM, in particular, illustrates a compiler-based approach that statically analyzes computation graphs, profiles low-level operator performance, and searches over execution configurations to optimize a target computation for a given deployment setting. It provides a scalable framework for exploring large design spaces and identifying efficient configurations.

As LLM systems evolve beyond single-model inference, they are increasingly instantiated as *structured LLM workflows* composed of multiple specialized LLM sub-agents (Zhang et al., 2024; Hu et al., 2024). A structured LLM workflow connects these sub-agents through a predefined execution graph, which may include sequential, parallel, branching, or iterative control flow. This abstraction is particularly useful for complex tasks that require multi-step problem solving rather than a single generation step. For example, a clinical decision-support workflow may retrieve relevant patient information and medical guidelines, invoke specialized agents for diagnosis and verification, and aggregate their outputs into an auditable recommendation. By enforcing structured execution, such workflows improve reliability, reproducibility, and controllability.

These benefits stem from the explicit, program-like execution graphs underlying structured LLM workflows. In this work, we focus on this structured setting, where the control flow and sub-agents

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

are specified before execution. Such explicit graphs enable systematic analysis of workflow-level behavior and expose a well-defined design space for optimization. This scope is distinct from open-ended agentic systems such as ReAct (Yao et al., 2022), which dynamically interleave reasoning and tool use and may produce substantially different execution traces across queries.

Within this structured setting, optimization is still substantially more challenging than optimizing a single LLM deployment. Each sub-agent can be configured through model selection and reasoning budget, and the workflow structure itself may also expose configurable choices, yielding a combinatorial workflow design space that quickly becomes very large. More importantly, workflow optimization differs from conventional machine learning compilation: instead of optimizing latency while preserving the model's original computation, it must navigate configurations that trade output quality against inference cost. The natural output is therefore not a single fastest implementation, but a set of optimized operating points that support diverse deployment requirements and user preferences.

This frontier-level problem is fundamentally difficult: related formulations of multi-module model assignment are NP-hard (Chen et al., 2025), and our setting further increases the complexity by expanding the search space to include reasoning budgets and workflow-structure choices. Routing-based workflow optimization methods (Zhang et al., 2025; Yue et al., 2025; Su et al., 2025; Nie et al., 2025) learn or tune an inference-time policy to select configurations according to an accuracy–latency objective specified during training. As a result, each policy typically targets a single trade-off point and must be retrained or re-optimized to accommodate different deployment requirements.

Inspired by the machine learning compilers introduced at the beginning, we take a different perspective and argue that structured LLM workflow optimization can be formulated as a compilation problem rather than only as runtime routing. The key distinction is that compilation explores the workflow design space before deployment and produces a reusable set of workflow-level configurations, rather than selecting one configuration online for a particular trade-off objective. We introduce **FlowCompile**, an optimizing compiler that performs a single compile-time search over the workflow design space and outputs a reusable set of configurations spanning diverse accuracy–latency trade-offs. FlowCompile profiles sub-agents under different model and reasoning-budget choices, composes these sub-agent-level profiles through a workflow-level proxy, and uses the resulting estimates to efficiently explore the workflow configuration space. This compiler-style decomposition avoids exhaustive full-workflow profiling while preserving a flexible set of operating points for deployment. Experiments across diverse workflows and benchmarks show that FlowCompile consistently outperforms heuristically optimized workflows and routing-based baselines by a large margin. We summarize our contributions as follows.

- We formulate joint optimization of per-role models, per-role reasoning budgets, and workflow structure as compile-time exploration of reusable accuracy–latency trade-off sets for structured LLM workflows.
- We validate a structure-aware compositional proxy that reuses sub-agent profiles to guide workflow-level accuracy–latency exploration across the evaluated reasoning workflows.
- We present FlowCompile, an optimizing compiler that performs a single compile-time search over model choices, reasoning budgets, and workflow structures, consistently improving accuracy–latency trade-offs across diverse workflows and benchmarks.

2 Related Work

Structured LLM Workflow Optimization. Structured LLM workflows coordinate multiple LLM-based sub-agents under a predefined execution graph, but often incur substantial latency and inference overhead. Existing efficiency-oriented methods predominantly follow a routing-based paradigm. Representative methods include MaAS (Zhang et al., 2025), MasRouter (Yue et al., 2025), and DAAO (Su et al., 2025), which make inference-time decisions over models and collaboration strategies. Similarly, Nie et al. (2025) rewrites a workflow into a fixed program and learns an online policy to allocate backends to its components under streaming feedback. LLMSELECTOR (Chen et al., 2025) is closely related because it also leverages module-level assessments to optimize multi-module workflows, but it selects a single static configuration that maximizes accuracy without explicitly modeling cost or latency. DSPy (Khattab et al., 2023) also frames LM pipeline optimization as

compilation, but it mainly optimizes prompts and demonstrations for improving pipeline accuracy, rather than workflow-level execution trade-offs.

Our work is complementary to these approaches but addresses a different compiler-inspired formulation: instead of optimizing prompts or learning an inference-time routing policy, FlowCompile performs compile-time workflow-level design-space exploration and produces a reusable set of configurations spanning accuracy–latency trade-offs, without retraining or online adaptation.

Machine Learning Compilers. Machine learning compilers optimize high-level computational graphs by decomposing them into lower-level optimization units and searching over implementation choices under hardware-aware cost models. TVM (Chen et al., 2018a) is a representative end-to-end deep learning compiler that combines graph-level optimizations, such as operator fusion and layout transformation, with operator-level code generation and autotuning. AutoTVM (Chen et al., 2018b) further automates tensor-operator optimization by using learned cost models to guide search over large implementation spaces. Ansor (Zheng et al., 2020) extends this idea by automatically constructing search spaces and optimizing multiple subgraphs of a neural network through a task scheduler, providing a particularly relevant example of decomposing a full computation graph into local optimization tasks while targeting end-to-end performance.

FlowCompile draws inspiration from this compiler-style decomposition, but targets a different objective and optimization level. ML compilers typically optimize system-level metrics such as latency or memory while preserving the intended computation and output quality of the model. FlowCompile instead performs workflow-level optimization over structured LLM workflows, where model choices and reasoning budgets jointly affect answer quality and inference efficiency, creating an inherent accuracy–latency trade-off. The desired output is therefore a set of operating points rather than a single optimized implementation. Accordingly, in addition to reporting accuracy and latency, we use scalarization-based metrics from multi-objective optimization, such as expected utility, to evaluate trade-off quality (Hayes et al., 2021; Yang et al., 2019).

3 FLOWCOMPILE

3.1 PROBLEM DEFINITION AND OVERVIEW

We first formalize *structured LLM workflow compilation*. A structured LLM workflow consists of LLM-based sub-agents connected by a predefined execution graph that specifies sequential, parallel, conditional, or iterative control flow. We denote a structured LLM workflow by $\mathcal{W} = (\mathcal{A}, G)$, where $\mathcal{A}$ is the set of sub-agents and G is the workflow execution graph.

Let $\mathcal{C}$ denote the workflow design space. A workflow configuration $c \in \mathcal{C}$ instantiates the executable choices of the workflow, including role-level model assignments, reasoning budgets, and optional structural decisions such as branch or refinement-stage execution. The supplied graph specifies the legal structure family, including optional branches and bounded retries; compilation jointly selects these structural choices and sub-agent settings. Fan-in is represented by an aggregation sub-agent consuming the active branch outputs; each instantiated graph preserves the required input–output interfaces (Appendix A). A reasoning budget is the maximum number of generated reasoning tokens allocated to a sub-agent call. Executing c on a labeled validation set $\mathcal{D}_{\text{val}}$ induces a workflow-level performance vector $y(c) = (\text{Acc}(c), \text{Lat}(c))$, where $\text{Acc}(c)$ and $\text{Lat}(c)$ denote task accuracy and LLM inference latency.

Given $\mathcal{W}$, $\mathcal{D}_{\text{val}}$, and $\mathcal{C}$, the goal of workflow compilation is to identify a reusable set of high-quality configurations that spans the workflow's accuracy–latency trade-off space, enabling selection under different inference-time latency budgets or performance preferences. Exhaustively evaluating all configurations on $\mathcal{D}_{\text{val}}$ to construct this trade-off set is infeasible due to the combinatorial design space: with V sub-agents, M model choices, and B reasoning-budget options, model–budget assignment alone yields $(MB)^V$ configurations, before structural choices. Even a five-sub-agent workflow with five models and four budgets gives 3.2M configurations, making exhaustive evaluation impractical.

As shown in Figure 1, FlowCompile compiles a structured LLM workflow in three stages: sub-agent profiling, workflow-level estimation, and design-space exploration. Given a workflow specification and a labeled validation set, these stages produce reusable configurations spanning diverse accuracy–latency trade-offs.

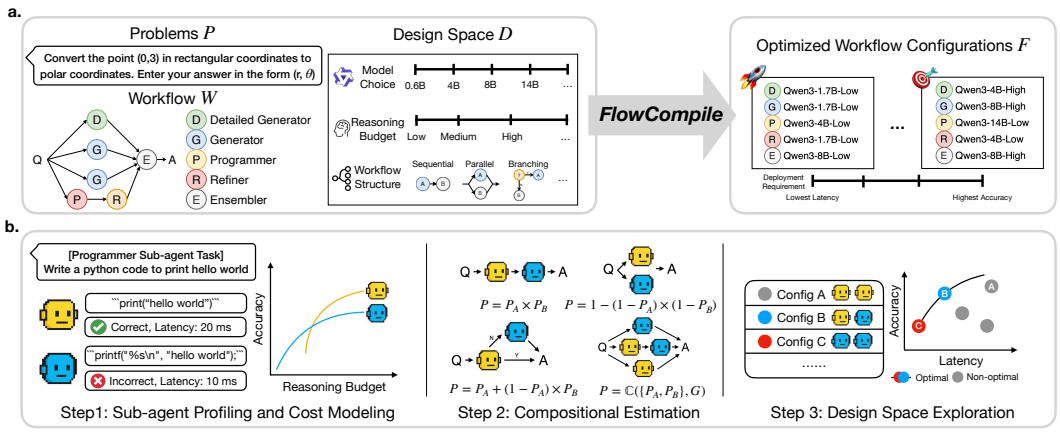


Figure 1: **Overview of FlowCompile.** (a) FlowCompile treats structured LLM workflow optimization as compilation: given a problem set, an input workflow, and a design space, it outputs a compiled set of optimized configurations spanning low-latency to high-accuracy deployment regimes. (b) FlowCompile compiles the workflow through three stages: sub-agent profiling and cost modeling, structure-aware compositional estimation of workflow-level accuracy and latency, and design-space exploration to identify configurations spanning the accuracy–latency trade-off frontier.

3.2 Sub-Agent Data Induction and Profiling

FlowCompile first constructs component-level cost models for each sub-agent. Since supervision in $\mathcal{D}_{\text{val}}$ is available only for final workflow outputs, FlowCompile induces sub-agent-level datasets from workflow traces. It executes the workflow on $\mathcal{D}_{\text{val}}$ using a high-capacity reference model, such as GPT-5 (Singh et al., 2026), records intermediate inputs and outputs for each sub-agent call, and applies an LLM-as-a-judge filter to retain calls that are well-executed and contribute to a correct final answer. The reference model executes the supplied workflow; it does not determine its decomposition or structure space. The judge prompt is provided in Appendix K.1. For each sub-agent $a \in \mathcal{A}$, the retained examples form an induced dataset $\mathcal{D}_a$, which serves as pseudo ground truth for profiling.

For each sub-agent a, we define a discrete sub-agent configuration space $\mathcal{Q}_a = \mathcal{M}_a \times \mathcal{R}_a$, where $\mathcal{M}_a$ is the set of candidate models and $\mathcal{R}_a$ is the set of reasoning budgets. For each configuration $q = (m, r) \in \mathcal{Q}_a$, FlowCompile evaluates sub-agent a on $\mathcal{D}_a$ and records empirical accuracy and latency: $\phi_a(q) = f_{\text{profile}}(a, q; \mathcal{D}_a) = \big(\hat{p}_a(q), \hat{\ell}_a(q)\big)$, where $\hat{p}_a(q)$ denotes the profiled sub-agent accuracy and $\hat{\ell}_a(q)$ denotes the profiled sub-agent latency. The accuracy $\hat{p}_a(q)$ is computed against the pseudo ground truth in $\mathcal{D}_a$, using task-specific matching when available and an LLM-as-a-judge otherwise; details of the profiling evaluation protocol are provided in Appendix K.2. The collection of profiles $\{\phi_a(q) : a \in \mathcal{A}, q \in \mathcal{Q}_a\}$ forms the component-level cost model used by the compiler and can be reused across workflow-level configurations during design space exploration.

3.3 Workflow-Level Performance Proxy

Given sub-agent profiles, FlowCompile estimates workflow-level performance with a compositional proxy. Directly obtaining the true performance $y(c) = (\text{Acc}(c), \text{Lat}(c))$ for every configuration $c \in \mathcal{C}$ would require full workflow execution and is infeasible at scale. Instead, FlowCompile estimates $\hat{y}(c) = (\widehat{\text{Acc}}(c), \widehat{\text{Lat}}(c))$ from reusable sub-agent profiles.

For a configuration c, let G_c denote the instantiated workflow graph and $q_a(c)$ the configuration assigned to sub-agent a. The proxy is defined as $\hat{y}(c) = \mathcal{M}_\theta\left(\{\phi_a(q_a(c))\}_{a \in \mathcal{A}_c}, G_c, E\right)$, where E denotes the deployment execution model, such as an edge deployment setting where LLM calls are queued and executed sequentially. The mapping $\mathcal{M}_\theta$ composes sub-agent-level profiles according to the workflow structure G_c and execution model E to produce workflow-level accuracy and latency estimates. The proxy is used to compare executable configurations during compilation, with its quality assessed by the resulting trade-offs and deployment choices.

Proxy objective. FlowCompile does not require the proxy to predict absolute performance exactly; it requires the proxy to identify the right configurations. We assess this objective through two properties, evaluated empirically rather than assumed. *Frontier consistency*: configurations that are non-dominated under measured performance remain so under the proxy estimates, and clearly dominated configurations are not selected into the estimated frontier $\widehat{\mathcal{F}}$. *Local order preservation*: among candidate configurations, the proxy preserves the measured ordering of accuracy and of latency, i.e., $\widehat{\mathrm{Acc}}(c_i) > \widehat{\mathrm{Acc}}(c_j)$ when $\mathrm{Acc}(c_i) > \mathrm{Acc}(c_j)$, and likewise for latency.

Proxy instantiation. The proxy $\mathcal{M}_\theta$ can use analytical rules, learned estimators, or hybrid models. We adopt a lightweight, training-free analytical instantiation that composes sub-agent profiles according to workflow structure and deployment execution.

Accuracy proxy. Let $\hat{p}_a$ denote the profiled accuracy of sub-agent a. FlowCompile composes these values according to workflow control-flow semantics to obtain a structure-aware estimate of workflow-level accuracy. Here, "sequential" and "parallel" describe the logical structure of the workflow graph, rather than the physical execution schedule of LLM calls, batching, or hardware parallelism. Bounded loops are handled by unrolling them into the corresponding sequence of conditional workflow stages:

$$\begin{aligned}
\hat{p}_{\mathrm{seq}} &= \prod_{i=1}^{N} \hat{p}_{a_i} && \text{(logical sequential composition)},\\
\hat{p}_{\mathrm{or}} &= 1 - \prod_{i=1}^{N} (1 - \hat{p}_{a_i}) && \text{(disjunctive parallel branches)},\\
\hat{p}_{\mathrm{and}} &= \prod_{i=1}^{N} \hat{p}_{a_i} && \text{(conjunctive parallel branches)},\\
\hat{p}_{\mathrm{cond}} &= \hat{p}_{a_1} + (1 - \hat{p}_{a_1})\hat{p}_{a_2} && \text{(conditional composition)}.
\end{aligned} \tag{1}$$

Together, these rules define the recursive estimator: $\widehat{\mathrm{Acc}}(c) = \mathcal{C}_{\mathrm{acc}}(\{\hat{p}_a(q_a(c))\}, G_c)$. For simplicity and computational efficiency, these rules approximate workflow accuracy without explicitly modeling input shifts, correlated errors, or downstream error correction.

Latency proxy. Let $\hat{\ell}_a$ denote the profiled latency of sub-agent a. We estimate workflow-level latency with an expected-latency rule, $\widehat{\mathrm{Lat}}(c) = \mathcal{C}_{\mathrm{lat}}(\{\hat{\ell}_a(q_a(c))\}_{a \in A_c}, G_c, E)$, where E is the deployment execution model. Under our edge execution model, LLM calls run sequentially, so unconditional stages are summed. Conditional branches are weighted by execution probabilities; e.g., if a_2 runs only when a_1 fails, then $\widehat{\mathrm{Lat}}_{\mathrm{cond}} = \hat{\ell}_{a_1} + (1 - \hat{p}_{a_1})\hat{\ell}_{a_2}$. Bounded retry loops are unrolled and composed similarly. Other execution models, such as critical-path latency under parallel execution, can be handled by replacing $\mathcal{C}_{\mathrm{lat}}$ without re-profiling sub-agents.

Workflow-specific proxy instantiations are provided in Appendix B.3.

From estimation error to selection quality. These compositional estimates need not be exact: selection depends on how their errors vary across candidates. For a fixed preference α and candidate set S, let $U_\alpha(c)$ and $\widehat{U}_\alpha(c)$ denote measured and predicted utility (Appendix J), and define $e_\alpha(c) = \widehat{U}_\alpha(c) - U_\alpha(c)$. Let c^*_α maximize measured utility over S. For a selected $\hat{c}_\alpha \in S$, define its predicted-utility shortfall $\delta_\alpha = \max_{c \in S} \widehat{U}_\alpha(c) - \widehat{U}_\alpha(\hat{c}_\alpha)$. Then

$$0 \le U_\alpha(c^*_\alpha) - U_\alpha(\hat{c}_\alpha) \le \delta_\alpha + \max_{c \in S} e_\alpha(c) - \min_{c \in S} e_\alpha(c). \tag{2}$$

The bound separates the selection rule's shortfall from *differences* in proxy error: a shared offset cancels. The omitted interactions contribute to e_α and can increase its variation across configurations. Richer estimators, such as conditional profiling on candidate-generated inputs, can be plugged into the same framework to reduce these errors (Appendix M). Appendix L gives the proof and separates within-set selection from coverage of the full design space.

Section 4.2 evaluates the instantiated proxy through exhaustively measured frontiers in restricted design spaces and predicted–measured rank agreement. Despite these simplifications, the proxy preserves most accuracy and latency orderings, and its selected set matches the measured frontier in the exhaustively evaluated spaces.

3.4 Design Space Exploration and Deployment

Trade-off set construction. Given the workflow-level proxy, FlowCompile performs lightweight, CPU-only compile-time exploration over the cached profiles in two steps. First, it applies sub-agent-level pruning: for each sub-agent, configuration q_1 is removed if it is dominated by q_2, i.e., $\hat{p}_a(q_2) \geq \hat{p}_a(q_1)$ and $\hat{\ell}_a(q_2) \leq \hat{\ell}_a(q_1)$, with at least one strict inequality. Under a monotone workflow-level proxy, this pruning preserves the attainable non-dominated proxy performance vectors while reducing the search space. FlowCompile then enumerates the remaining configurations $\widetilde{\mathcal{C}}$, computes $\hat{y}(c) = (\widehat{\mathrm{Acc}}(c), \widehat{\mathrm{Lat}}(c))$, and applies non-dominated sorting (Kung et al., 1975) to obtain the proxy-estimated trade-off set $\widehat{\mathcal{F}}$.

Using the compiled set. Deployment selects from $\widehat{\mathcal{F}}$ under three settings: latency-constrained deployment maximizes predicted accuracy within a predicted latency budget; preference-based deployment selects an accuracy–latency operating point; and routing-based adaptation uses $\widehat{\mathcal{F}}$ for per-query routing. Predicted-utility maximization gives $\delta_\alpha = 0$ in Equation 2; the latency-rank rule used in our evaluation can have a nonzero shortfall (Appendix F). Section 4.3 evaluates operating points and preference-based selection; Section 4.4 evaluates routing.

Compilation cost. FlowCompile incurs no model-training cost. Its main cost is sub-agent profiling, which scales as $\sum_{a \in \mathcal{A}} |\mathcal{Q}_a|$ for a fixed profiling set rather than as the full workflow design space $|\mathcal{C}|$. The remaining combinatorial search uses cached profiles rather than LLM calls. In our experiments, profiling takes about one hour per benchmark on 32 H100 GPUs; numerical exploration takes 0.01–0.99s on a CPU. By contrast, evaluating all 1.04M HotpotQA configurations in our search space end to end would take an estimated 173K hours on the same hardware. After compilation, runtime deployment selects from the compiled configuration set and does not rerun the compilation process. Appendix C details the cost analysis and scalability.

4 Experiments

4.1 Settings

Metric. Our evaluation metrics include task accuracy and LLM inference latency. Latency is measured on a single H100 GPU with vLLM (Kwon et al., 2023) as the inference engine. For preference-aware evaluation, we follow the standard utility-based view in multi-objective decision making (Hayes et al., 2021) and use expected utility as the scalar evaluation metric. In our two-objective (accuracy and latency) setting, the utility weight reduces to a single preference parameter α: smaller α prioritizes latency efficiency, while larger α prioritizes accuracy. To assess proxy estimation quality and ranking consistency, we report Spearman correlation ρ, pairwise agreement, and calibrated mean absolute error (cMAE). Formal definitions are provided in Appendix J.

Design Space. FlowCompile searches a unified configuration space spanning three dimensions: (1) *Model size*, using the Qwen-3 family (Yang et al., 2025) with sizes 0.6B, 1.7B, 4B, 8B, 14B; (2) *Reasoning budget*, with discrete budgets ranging from 10 to 16,000 tokens, enforced via budget forcing (Muennighoff et al., 2025); and (3) *Workflow structure*, based on AFlow workflows (Zhang et al., 2024), with optional stages subject to the benchmark-specific candidate constraints. Additional details are provided in Appendix B.

Datasets. We evaluate on four public benchmarks spanning three domains: (i) mathematical reasoning, including **GSM8K** (Cobbe et al., 2021) and **MATH-500** (Lightman et al., 2023); (ii) multi-hop question answering, including **HotpotQA** (Yang et al., 2018); and (iii) code reasoning, including **LiveCodeBench** (Jain et al., 2024). We construct disjoint profiling and evaluation subsets: the profiling subset is used for sub-agent profiling and baseline training, while all reported results are measured on the held-out evaluation subset. Detailed split protocols are provided in Appendix B.1.

Baselines. We compare FlowCompile against three categories of baselines: (1) *Single-model baselines*, which directly apply a single large reasoning model without workflows, including Qwen3-32B (Yang et al., 2025) and QwQ-32B (QwenTeam, 2025); (2) *Fixed-workflow baselines*, which execute the same AFlow workflow with a fixed model assignment for all sub-agents; and (3) *Routing-based baselines*, which adapt model assignment or workflow structure at runtime using query-dependent routers. These include a KNN-based model router implemented with the LLM Router

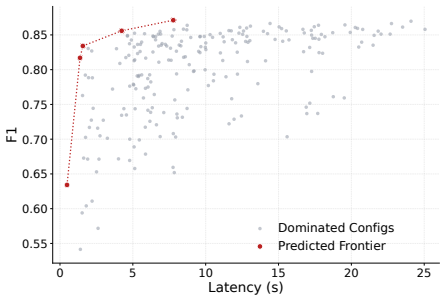


Figure 2: **Frontier consistency.** The proxy-selected set matches the measured frontier of the evaluated HotpotQA configurations.

Table 1: **Local order preservation within compiled configurations.** The proxy preserves the ordering of high-quality configurations and yields low absolute error after systematic bias correction.

Latency Estimation			
Benchmark	Spearman (↑)	Pairwise (↑)	cMAE (↓)
GSM8K	0.98	0.96	2.8
MATH-500	0.97	0.96	10.8
HotpotQA	0.99	0.98	0.1
LiveCodeBench	0.90	0.89	3.8
Average	**0.96**	**0.95**	**4.4**
Accuracy Estimation			
Benchmark	Spearman (↑)	Pairwise (↑)	cMAE (↓)
GSM8K	0.93	0.89	0.015
MATH-500	0.81	0.81	0.047
HotpotQA	0.98	0.95	0.013
LiveCodeBench	0.97	0.95	0.015
Average	**0.92**	**0.90**	**0.023**

framework (Feng et al., 2025), and MaAS (Zhang et al., 2025), which performs query-dependent structural routing. Two further baselines adapt to accuracy–latency preferences. *Pref-Aware Router* performs model-size routing within a fixed workflow, while *Pref-Aware MaAS* combines MaAS-style structural routing with model-size selection; both select by latency rank. Fixed-workflow baselines use the AFlow base graphs; all workflow baselines use Qwen-3 executors.

4.2 Proxy Validation

We test the two proxy properties defined in Section 3.3 against measured workflow performance.

Frontier consistency. We first test whether the proxy can recover the empirical accuracy–latency frontier by comparing selected configurations with evaluated alternatives in restricted design spaces. The HotpotQA and LiveCodeBench comparisons retain the workflows and proxy composition rules with fewer component choices (Appendix D). Figures 2 and 6 show that the proxy-estimated configurations (red dots) closely align with the empirically measured non-dominated region, while most remaining configurations are dominated in the measured accuracy–latency space. The selected set's best measured utility is within 0.003 of the best evaluated alternative for every preference. Because every alternative, including each uniform model–budget assignment, is executed end to end, this frontier bounds any search over these restricted spaces; FlowCompile reaches it from component profiles alone, without executing the alternatives.

Local order preservation. We next test whether the proxy preserves ordering within the compiled set. For each benchmark, we sample about 20 configurations from the compiled set, run full test-set evaluation, and compare proxy-estimated and measured accuracy/latency using the metrics in Section 4.1. Table 1 shows that the proxy largely preserves the ordering of high-quality configurations, with average pairwise agreement of 0.90 for accuracy and 0.95 for latency. MATH-500 is the most challenging case for accuracy estimation, but the proxy still preserves most pairwise orderings. After correcting systematic bias, the proxy achieves low absolute error, with an average latency error of 4.4 seconds and an average accuracy error of 2.3 percentage points. The calibration is a post hoc diagnostic and is not used for configuration selection.

4.3 End-to-End Quality of Compiled Configurations

Having validated the proxy, we evaluate the compiled configurations on the test set by comparing their measured accuracy–latency trade-offs with baselines and assessing their preference-aware performance using expected utility.

Accuracy–latency trade-offs. We evaluate the compiled configurations and all baselines under the same inference setting across four benchmarks. As shown in Figure 3, FlowCompile consistently achieves lower latency at comparable or higher accuracy than the baselines, yielding substantially better accuracy–latency trade-off curves. These gains reflect the complete system, including its model, budget, and structural choices; Table 4 measures their incremental contributions within

Table 2: **Preference-aware evaluation under heterogeneous preferences.** Expected utility across four benchmarks under randomly sampled per-query preferences, reported as mean ± std over ten preference draws.

Method	Expected Utility (↑)				
	GSM8K	MATH-500	HotpotQA	LiveCodeBench	**Average**
		Single Model			
Qwen3-32B	74.1 ± 0.6	82.2 ± 0.3	67.4 ± 0.4	55.9 ± 0.3	69.9
QwQ-32B	61.6 ± 1.2	80.6 ± 0.4	62.6 ± 0.7	54.6 ± 0.4	64.9
		Fixed Workflow			
Qwen3-4B	69.9 ± 0.7	70.3 ± 0.7	68.3 ± 0.4	61.0 ± 0.3	67.4
Qwen3-8B	57.6 ± 0.9	48.5 ± 1.4	58.1 ± 0.8	49.1 ± 0.6	53.3
		Workflow with Router			
MaAS (Qwen3-4B)	87.2 ± 0.1	78.4 ± 0.9	74.2 ± 0.5	58.2 ± 0.6	74.5
MaAS (Qwen3-8B)	82.2 ± 0.4	71.9 ± 0.6	65.2 ± 0.5	58.0 ± 0.5	69.3
KNN Router	81.9 ± 0.2	77.6 ± 0.5	74.1 ± 0.4	53.6 ± 0.3	71.8
Pref-Aware Router	68.0 ± 0.7	70.1 ± 1.1	69.3 ± 0.4	59.7 ± 0.6	66.8
Pref-Aware MaAS	80.7 ± 0.2	78.2 ± 1.3	78.7 ± 0.4	65.5 ± 0.8	75.8
FlowCompile	$\mathbf{89.2} \pm 0.6$	$\mathbf{84.2} \pm 0.8$	$\mathbf{88.4} \pm 0.4$	$\mathbf{80.1} \pm 0.6$	**85.5**

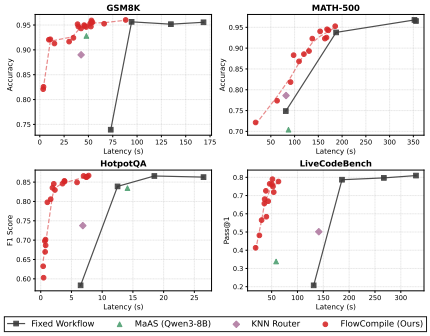


Figure 3: **Empirical accuracy–latency trade-offs.** FlowCompile produces configurations with substantially better measured accuracy–latency trade-offs than baselines.

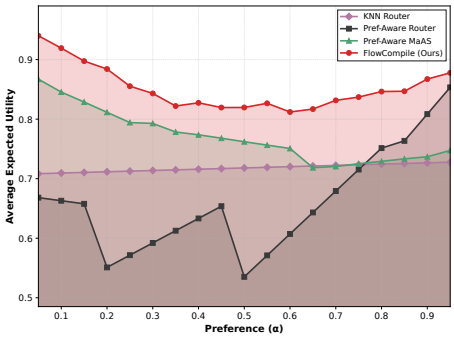


Figure 4: **Preference-aware evaluation under fixed preferences.** FlowCompile achieves the highest expected utility averaged across four benchmarks as the preference varies.

FlowCompile. We also provide numerical accuracy and latency results in Table 5 in Appendix E. Since FlowCompile outputs a set of configurations rather than a single operating point, we report two representative measured operating points: an accuracy-priority configuration closest to the full Qwen3-14B workflow, and a latency-priority configuration; prediction-based selection is evaluated below. The accuracy-priority setting achieves nearly the same accuracy as the full baseline with an average $3.4\times$ speedup and an even greater speedup of $\mathbf{6.4}\times$ in LiveCodeBench, while the latency-priority setting achieves an average $\mathbf{10.8}\times$ speedup with task-specific accuracy trade-offs reported in the same table. Against the Qwen3-4B fixed workflow, compiled configurations are more accurate at $3.3\times$ and $3.6\times$ lower latency on HotpotQA and LiveCodeBench, and at similar latency on GSM8K and MATH-500.

Preference-aware evaluation. Expected utility combines accuracy and latency efficiency under a specified preference weight, enabling direct comparison across methods (Hayes et al., 2021). We vary the preference parameter $\alpha \in (0, 1)$ under two settings. In the heterogeneous setting, each query is assigned a randomly sampled α to represent mixed-preference workloads; we repeat this ten times and report the mean and standard deviation. In the fixed setting, all queries share the same α, and we sweep α across the preference spectrum to evaluate robustness under different global preferences. FlowCompile maps α to a rank in the compiled set ordered by predicted latency, then scores the selected configuration on held-out data (Appendices F and J).

Table 3: **Cross-benchmark transfer.** Reusing MATH-500 sub-agent profiles for GSM8K preserves proxy quality and good expected utility.

	Correlation (ρ)		Expected Utility
	Latency	Accuracy	
GSM8K	0.98	0.93	89.2
GSM8K (transfer)	0.94	0.76	86.4

Table 4: **Design space ablation.** Expected utility improves as FlowCompile expands the searchable design space.

Design Space	HotpotQA
Model	69.7
Model + Budget	83.2
Model + Budget + Structure (Ours)	**88.4**

Table 2 shows that under the heterogeneous setting, FlowCompile achieves the highest expected utility across all benchmarks, outperforming the strongest workflow baseline by an average of $+8.0$. Figure 4 further shows that, in the fixed setting, FlowCompile achieves the highest average utility across the four benchmarks for every preference value. Preference-aware routing baselines are effective only over narrower ranges. Per-benchmark results in Figure 7 of Appendix F show similar patterns. The same compiled set serves every preference without re-profiling or retraining.

Qualitative analysis. We further inspect the compiled configurations and find that they form interpretable operating regimes: latency-priority configurations favor simpler workflows and lower-cost model-budget choices, while accuracy-priority configurations allocate compute to task-critical stages. Detailed analysis is provided in Appendix I.

4.4 Additional Analysis

We further analyze FlowCompile's behavior and design choices. Unless otherwise specified, all experiments use the heterogeneous preference-aware evaluation setting.

Per-query routing. FlowCompile is complementary to routing-based methods because the compiled configuration set provides a compact candidate pool for query-level adaptation. To demonstrate this, we combine FlowCompile with a simple KNN router ($k = 20$) on GSM8K and MATH-500, allowing the router to select among compiled configurations for each query. As shown in Table 6 in Appendix G, this further improves expected utility over FlowCompile alone, with gains of $+2.6$ and $+6.3$ on GSM8K and MATH-500, respectively. Appendix G shows that the router's selections correlate with problem difficulty. These results highlight the complementarity between compile-time optimization and query-level selection from the compiled set.

Cross-benchmark transfer. Since sub-agent profiling is the main cost of FlowCompile, we evaluate whether profiles can be reused across related tasks sharing the same workflow. We reuse sub-agent data profiled on MATH-500 to compile configurations for GSM8K, without re-profiling on the GSM8K validation set, and evaluate the resulting configurations on the GSM8K test set. As shown in Table 3, transferred profiles preserve strong latency correlation, reasonable accuracy correlation, and competitive expected utility. This suggests that FlowCompile can reuse profiles across related tasks to reduce profiling cost while still producing competitive compiled configurations.

Ablation study. We ablate two design choices on HotpotQA. First, we ablate the design space by progressively adding model choice, reasoning-budget choice, and workflow-structure choice; adding the per-role reasoning-budget dimension improves utility by 13.5 points, and structure choices contribute a further 5.2 points (Table 4). Each space contains the previous one, including all uniform assignments, and each expansion improves measured utility. Second, we test sensitivity to the reference model used for sub-agent data induction by replacing GPT-5 with GPT-5-mini or Qwen3-1.7B for sub-agent data induction. The resulting frontiers are highly consistent (Figure 10), suggesting robustness as long as the reference model can generate reasonable workflow traces.

5 Conclusion

FlowCompile is an optimizing compiler for structured LLM workflows. Our results show that composing reference-trace profiles enables joint model, budget, and structural optimization. By turning workflow optimization into a reusable compile-time artifact, FlowCompile enables configurations to be selected and adapted under diverse deployment requirements. More broadly, our results motivate *workflow compilation* as a systems abstraction for future LLM applications, where efficiency, controllability, and adaptability must be managed at the workflow level.

AI USE STATEMENT

Generative AI is used within FlowCompile to induce sub-agent profiling data and to judge intermediate outputs, as described in Section 3.2 and Appendix K. Generative AI tools also assisted with paper drafting and editing, supplementary experiment code and analysis, interpretation of results, formulation of the selection-error analysis, and the writing of its proof. These research and writing activities were conducted under close author supervision, with rigorous checks throughout the AI-assisted generation process. The authors carefully reviewed all AI-assisted content, including code, analyses, claims, and proof arguments. The selection-error analysis includes a direct proof with explicit conditions. The authors take full responsibility for the final text, claims, and artifacts.

REPRODUCIBILITY STATEMENT

Algorithm 1 specifies the compilation procedure. Appendix B details the data splits, hardware, reasoning budgets, workflow structures, and proxy instantiations; Appendix C reports compilation cost. Appendix J defines the evaluation metrics and selection protocol, and Appendix K provides the judge prompts. Appendix L proves the selection-error bound and relates within-set selection to candidate-space coverage. Upon acceptance, we will release all code and the compiled configurations obtained by the search.

REFERENCES

Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Lingjiao Chen, Jared Quincy Davis, Boris Hanin, Peter Bailis, James Zou, Matei Zaharia, and Ion Stoica. Llmselector: Learning to select models in compound ai systems. In *ICML 2025 Workshop on Collaborative and Federated Agentic Workflows*, 2025.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pp. 578–594, 2018a.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31, 2018b.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Tao Feng, Haozhen Zhang, Zijie Lei, Haodong Yue, Chongshan Lin, Ge Liu, and Jiaxuan You. Llmrouter: An open-source library for llm routing. `https://github.com/ulab-uiuc/LLMRouter`, 2025. GitHub repository.

Conor F Hayes, Roxana Rădulescu, Eugenio Bargiacchi, Johan Källström, Matthew Macfarlane, Mathieu Reymond, Timothy Verstraeten, Luisa M Zintgraf, Richard Dazeley, Fredrik Heintz, et al. A practical guide to multi-objective reinforcement learning and planning. *arXiv preprint arXiv:2103.09568*, 2021.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

Hsiang-Tsung Kung, Fabrizio Luccio, and Franco P Preparata. On finding the maxima of a set of vectors. *Journal of the ACM (JACM)*, 22(4):469–476, 1975.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step, 2023. URL https://arxiv.org/abs/2305.20050.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori B Hashimoto. s1: Simple test-time scaling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 20286–20332, 2025.

Lunyiu Nie, Zhimin Ding, Kevin Yu, Marco Cheung, Chris Jermaine, and Swarat Chaudhuri. Resource-efficient inference with foundation model programs. *arXiv preprint arXiv:2504.07247*, 2025.

QwenTeam. Qwq-32b: Embracing the power of reinforcement learning, March 2025. URL https://qwenlm.github.io/blog/qwq-32b/.

Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2026.

Jinwei Su, Qizhen Lan, Yinghui Xia, Lifan Sun, Weiyou Tian, Tianyu Shi, Xinyuan Song, and Lewei He. Difficulty-aware agentic orchestration for query-specific multi-agent workflows. *arXiv preprint arXiv:2509.11079*, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Runzhe Yang, Xingyuan Sun, and Karthik Narasimhan. A generalized algorithm for multi-objective reinforcement learning and policy adaptation. *Advances in neural information processing systems*, 32, 2019.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.

Yanwei Yue, Guibin Zhang, Boyang Liu, Guancheng Wan, Kun Wang, Dawei Cheng, and Yiyan Qi. MasRouter: Learning to route LLMs for multi-agent systems. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15549–15572, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.757. URL https://aclanthology.org/2025.acl-long.757/.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. *arXiv preprint arXiv:2502.04180*, 2025.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*, 2024.

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pp. 863–879, 2020.

A FLOWCOMPILE ALGORITHM

Algorithm 1: FLOWCOMPILE: Structured LLM Workflow Compilation

Input: Workflow $\mathcal{W} = (\mathcal{A}, G)$ with sub-agents $\mathcal{A}$ and structure space;
Workflow-level labeled validation set $\mathcal{D}_{\text{val}}$;
Reference model m_{ref};
Sub-agent model choices $\{\mathcal{M}_a\}_{a\in\mathcal{A}}$ and reasoning budgets $\{\mathcal{R}_a\}_{a\in\mathcal{A}}$;
Deployment execution model $\mathcal{E}$.

Output: Proxy-estimated optimized configuration set $\widehat{\mathcal{F}}$.

Step 1: Sub-agent data induction and profiling
Execute $\mathcal{W}$ on $\mathcal{D}_{\text{val}}$ using m_{ref} to collect workflow traces;
Apply LLM-as-a-judge filtering to retain high-quality intermediate input–output pairs;
Construct induced sub-agent datasets $\{\mathcal{D}_a\}_{a\in\mathcal{A}}$;
foreach *sub-agent* $a \in \mathcal{A}$ **do**
  Define $\mathcal{Q}_a = \mathcal{M}_a \times \mathcal{R}_a$;
  foreach *sub-agent configuration* $q = (m, r) \in \mathcal{Q}_a$ **do**
    Profile a on $\mathcal{D}_a$ under q to obtain $\phi_a(q) = \big(\hat{p}_a(q), \hat{\ell}_a(q)\big)$;
  Remove locally dominated configurations from $\mathcal{Q}_a$ under the monotone proxy;

Step 2: Workflow-level compositional estimation
Enumerate the pruned workflow configuration space $\widetilde{\mathcal{C}}$;
foreach *workflow configuration* $c \in \widetilde{\mathcal{C}}$ **do**
  Let G_c be the instantiated workflow graph and $q_a(c)$ be the configuration assigned to sub-agent a;
  Estimate workflow accuracy $\widehat{\mathrm{Acc}}(c) = \mathcal{C}_{\text{acc}}\big(\{\hat{p}_a(q_a(c))\}_{a\in\mathcal{A}_c}, G_c\big)$;
  Estimate workflow latency $\widehat{\mathrm{Lat}}(c) = \mathcal{C}_{\text{lat}}\big(\{\hat{\ell}_a(q_a(c))\}_{a\in\mathcal{A}_c}, G_c, \mathcal{E}\big)$;
  Set $\hat{y}(c) = \big(\widehat{\mathrm{Acc}}(c), \widehat{\mathrm{Lat}}(c)\big)$;

Step 3: Trade-off set construction
Apply non-dominated sorting to $\{\hat{y}(c) : c \in \widetilde{\mathcal{C}}\}$, treating higher accuracy and lower latency as better;
Let $\widehat{\mathcal{F}}$ be the resulting proxy-estimated non-dominated configuration set;

return $\widehat{\mathcal{F}}$

Executable structure choices and aggregation. The input graph specifies the roles, interfaces, and allowed control-flow choices. Each G_c is a legal instantiation of this structure family: a retained consumer receives its required inputs, and at least one answer-producing branch remains active. The reference model executes this supplied workflow to induce profiling examples; it is not asked to invent the decomposition. Fan-in is represented by the aggregation role consuming the active branch outputs. For example, in HotpotQA with two active generators g_1, g_2, ensemble s, and formatter f, the proxy is

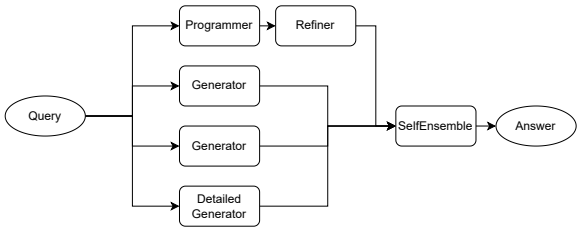
$$\widehat{\mathrm{Acc}}(c) = \big[1 - (1 - \hat{p}_{g_1})(1 - \hat{p}_{g_2})\big]\hat{p}_s\hat{p}_f, \qquad \widehat{\mathrm{Lat}}(c) = \hat{\ell}_{g_1} + \hat{\ell}_{g_2} + \hat{\ell}_s + \hat{\ell}_f.$$

With a single generator, the ensemble is bypassed; a legal formatter bypass removes its accuracy factor and latency term. These are alternative executable graphs within the declared structure space, jointly searched with model and budget assignments. Appendix B.3 gives the corresponding math and bounded-repair instantiations.

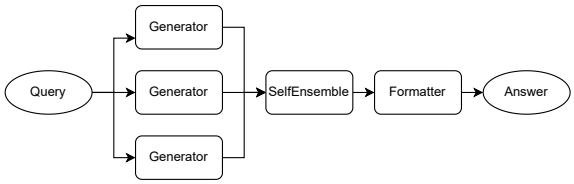
B ADDITIONAL EXPERIMENT SETUP DETAILS

B.1 DATASET SPLITS

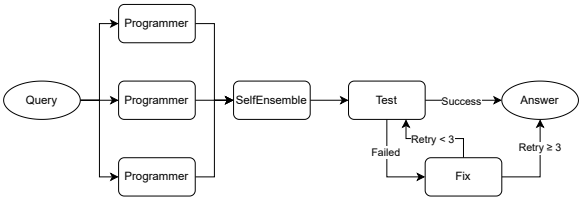
FlowCompile requires a profile set for sub-agent data induction, sub-agent profiling, configuration selection, and baseline training, while final results must be evaluated on held-out examples. We



(a) Workflow structure used for GSM8K and MATH-500.



(b) Workflow structure used for HotpotQA.



(c) Workflow structure used for LiveCodeBench.

Figure 5: Workflow structures used by FlowCompile across benchmarks.

therefore construct disjoint profile and test sets for each benchmark. Unless otherwise specified, we use a 1:4 split, with 20% of the examples used for profiling and 80% held out for testing.

The profile/test counts are 264/1,055 for GSM8K, 98/402 for MATH-500, 200/800 for HotpotQA, and 175/704 for LiveCodeBench. GSM8K uses the official test set. MATH-500 is stratified by difficulty level, assigning $\lfloor n_l/5 \rfloor$ examples per level to profiling. Following AFlow (Zhang et al., 2024), HotpotQA samples 1,000 examples from the distractor training split and partitions them after shuffling. LiveCodeBench uses `release_v5`, stratified by difficulty with $\lfloor n_d/5 \rfloor$ examples per stratum assigned to profiling. MATH-500, HotpotQA, and LiveCodeBench use seed 42. These are source-example counts; judge filtering can yield fewer retained examples for an individual role.

The profile set is used only for compile-time profiling, configuration selection, and baseline training. All primary benchmark accuracy, F1, Pass@1, latency, and expected-utility results are measured only on the held-out test set.

B.2 Latency Measurement and Reasoning Budgets

Latency measurement. All latency measurements are conducted on a single NVIDIA H100 80GB GPU using the vLLM inference engine. We measure end-to-end model inference latency with a batch size of 1 to ensure accurate measurements in our setting. Profiling generates sub-agent outputs in large batches for throughput; sub-agent latency profiles use batch-size-1 timing on the same H100 and vLLM setup as the workflow measurements.

Reasoning budget. We consider a discrete set of reasoning budgets for each benchmark, chosen to cover a wide range of accuracy–latency trade-offs. For GSM8K, we use [10, 200, 400, 800, 1000, 1500, 2000, 3000, 4000, 5000, 6000, 8000, 10000]. For MATH-500, we use [10, 200, 400, 800, 1000, 1500, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 12000, 16000]. For HotpotQA, we use [10, 50,

100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1500, 2000, 4000, 8000]. For LiveCodeBench, we use [10, 200, 400, 800, 1000, 1500, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 12000, 16000].

B.3 Workflow Structures and Proxy Instantiations

For each benchmark, we use the full workflow discovered by AFlow (Zhang et al., 2024) as the base workflow for FlowCompile (Figure 5). Importantly, FlowCompile treats the workflow structure itself as part of the configuration space, rather than assuming a fixed execution graph. For GSM8K and MATH-500, we use a shared math workflow in which each of the four branches can be independently activated or skipped. The SelfEnsemble sub-agent is executed only when at least two branches are activated; otherwise, it is skipped. We apply the same SelfEnsemble activation rule to all three workflow types. For HotpotQA, Generator branches and Formatter are optional subject to a minimum of two active LLM calls: a single Generator is paired with Formatter; multiple Generators execute SelfEnsemble and may bypass Formatter. For LiveCodeBench, each of the three programmer sub-agents is optional, and the maximum number of retry attempts is additionally configurable. These structural choices are jointly optimized with model choices and reasoning budgets, and therefore may differ across compiled configurations.

We next provide workflow-specific details for the proxy used in our experiments. All modules, including SelfEnsemble, Formatter, Refiner, and Fix, are treated as profiled sub-agents. Their accuracy and latency are measured on induced sub-agent examples and then composed according to the workflow structure. In particular, SelfEnsemble is not modeled by a closed-form majority-vote rule; FlowCompile directly profiles its aggregation behavior on induced aggregation examples.

GSM8K and MATH-500. The math workflow contains four optional solving branches: a Programmer–Refiner branch, two Generator branches, and a Detailed Generator branch. For the Programmer–Refiner branch, the branch-level accuracy is computed by sequential composition, $\hat{p}_{\mathrm{prog}}\hat{p}_{\mathrm{ref}}$. The other solving branches use their profiled accuracies directly. Given the active branches in a configuration, FlowCompile first composes their branch-level accuracies using the disjunctive rule, estimating the probability that at least one active branch produces a correct candidate solution. When at least two branches are active, SelfEnsemble is executed and treated as a profiled aggregation sub-agent; its profiled accuracy is then composed sequentially with the candidate-solution stage. When only one branch is active, SelfEnsemble is skipped and the branch output is used directly. Under the edge execution model, the latency proxy sums the profiled latencies of all active branch calls, including Refiner when the Programmer branch is selected, and adds the profiled SelfEnsemble latency when SelfEnsemble is executed.

HotpotQA. The HotpotQA workflow contains up to three optional Generator branches, followed by an optional SelfEnsemble aggregation step and an optional Formatter. The active Generator branches are composed using the disjunctive rule, reflecting that any correct generated answer can support the final prediction. If multiple Generator branches are active, SelfEnsemble is executed as a profiled aggregation sub-agent and composed sequentially with the generator stage. The Formatter, when enabled, is also treated as a profiled sub-agent and composed sequentially with the preceding output. The latency proxy sums the profiled latencies of executed Generator calls and adds the SelfEnsemble and Formatter latencies when those modules are present.

LiveCodeBench. The LiveCodeBench workflow contains up to three optional Programmer branches, followed by SelfEnsemble, test execution, and bounded repair through the Fix module. The active Programmer branches are composed using the disjunctive rule, since any correct candidate program can solve the task. SelfEnsemble is treated as a profiled aggregation sub-agent that selects or combines candidate programs before testing. The Test node is not an LLM sub-agent; it deterministically executes the generated code against the public test cases provided by LiveCodeBench and returns a success or failure signal. Since test execution is deterministic and does not involve LLM inference, its latency is excluded from the LLM latency proxy. The repair stage is modeled as a bounded conditional workflow: Fix is executed only when the current program fails the public tests and the retry budget has not been exhausted. Public tests guide repair; final Pass@1 uses a separate private-test evaluator. Accordingly, the accuracy proxy follows the conditional composition rule over the initial program-generation stage and subsequent repair attempts. The latency proxy uses expected latency: unconditional generation and aggregation stages are summed, while each Fix attempt is weighted by the estimated probability that all previous attempts fail and the retry stage is executed.

C Additional Details on Compilation Cost

FlowCompile performs compile-time optimization and does not involve any model training. Its compilation cost consists of three main stages: (1) inducing sub-agent data and profiling sub-agents under candidate configurations, (2) composing workflow-level accuracy and latency estimates from the profiled sub-agent statistics, and (3) constructing the proxy-estimated accuracy–latency trade-off set. Suppose the profiling set contains D data points, the profiling procedure uses A distinct reusable sub-agent profile types, and each sub-agent role has M candidate models and R reasoning-budget options. For simplicity, we describe the cost assuming the same model and budget choices for all sub-agent roles; the expression directly generalizes to heterogeneous sub-agent configuration spaces.

C.1 Step-by-Step Cost Analysis

Stage 1: Sub-agent data induction and profiling. FlowCompile first executes the workflow on the profile set using a reference model to collect workflow traces and induce sub-agent-level profiling data. Bounded retries are unrolled for compositional estimation, while profiling cost counts reusable role profiles. LiveCodeBench uses generation, aggregation, and repair profile types, giving $3 \times 5 \times 15 = 225$ role–model–budget settings. This requires D reference workflow executions. Let $n_a = |\mathcal{D}_a|$ be the number of retained examples for profile type a; repeated calls can contribute multiple examples per reference workflow.

FlowCompile then profiles each sub-agent role independently under candidate model and reasoning-budget configurations. This requires $\sum_a n_a |\mathcal{Q}_a|$ sub-agent inference calls, or $MR \sum_a n_a$ with homogeneous configuration choices. The number of profiled role–model–budget settings is AMR; these settings each process their retained examples. These calls are independent single-model inferences and do not require executing the full workflow control logic, making this stage highly parallelizable. In our implementation, we use vLLM with large batch sizes to execute this stage efficiently. For example, in our experiments, we use 32 H100 GPUs with a total batch size of 256, yielding a typical profiling time of about **1 hour** per benchmark.

Stage 2: Workflow-level compositional estimation. Given the profiled sub-agent statistics, FlowCompile estimates workflow-level accuracy and latency by applying the structure-aware proxy to candidate workflow configurations. This stage involves no model inference or end-to-end workflow execution; it only performs numerical composition over cached sub-agent profiles.

For V independently configurable call positions after unrolling, each with MR model–budget choices and an optional bypass, a worst-case bound is $O((1 + MR)^V)$ candidates. Role sharing and legal-structure constraints reduce this count. In practice, FlowCompile first applies sub-agent-level Pareto pre-filtering, which removes locally dominated model–budget choices and effectively replaces MR with a much smaller pruned set size. With vectorized numerical operations, the resulting composition step completes in under one second on a CPU.

Stage 3: Trade-off set construction. Finally, FlowCompile identifies the proxy-estimated non-dominated configurations from the composed candidates. Let N denote the number of remaining workflow configurations after sub-agent-level pruning and structural enumeration. We apply standard non-dominated sorting using an $O(N \log N)$ algorithm (Kung et al., 1975), treating higher accuracy and lower latency as preferable. This stage also involves only lightweight numerical computation and completes in under one second on a CPU.

C.2 Comparison with Exhaustive Workflow Evaluation

A naive alternative to FlowCompile is to evaluate every workflow configuration end-to-end and then extract the empirical accuracy–latency frontier. This is computationally infeasible in our setting. The full design spaces contain approximately 2.37B, 4.84B, 2.56M, and 1.04M workflow configurations for GSM8K, MATH-500, LiveCodeBench, and HotpotQA, respectively. FlowCompile avoids exhaustive workflow-level evaluation by replacing it with one-time sub-agent profiling and analytic workflow-level composition.

The difference is substantial in practice. FlowCompile profiles only 325, 375, 225, and 240 sub-agent settings in total for GSM8K, MATH-500, LiveCodeBench, and HotpotQA, respectively, and this profiling stage takes about one hour per benchmark. In contrast, on HotpotQA, one full-workflow

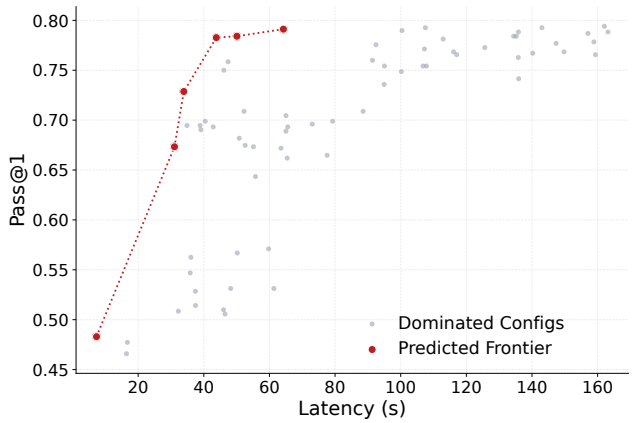


Figure 6: **Frontier consistency on restricted LiveCodeBench configurations.** The proxy-estimated frontier recovers high-quality empirically measured configurations across the accuracy–latency trade-off space, while most remaining configurations are dominated after full workflow execution.

evaluation takes approximately 10 minutes on the same hardware. Exhaustively evaluating the full HotpotQA configuration space would therefore require roughly $1.04\text{M} \times 10$ minutes, or about **173,333 hours**, compared with about **one hour** for FlowCompile's sub-agent profiling stage.

FlowCompile further reduces the analytic search space through exact sub-agent-level Pareto filtering before workflow-level composition. Under the monotone proxy rules, replacing a locally dominated choice by a locally Pareto-superior choice cannot worsen estimated workflow accuracy or latency, provided the replacement preserves the graph and its interfaces. The replacement can tie at workflow level, so pruning preserves attainable non-dominated proxy performance vectors rather than every configuration realizing them. This guarantee concerns the proxy objective; empirical performance is evaluated separately. In practice, this reduces each sub-agent role from roughly 65–80 settings to 6–19 settings. After pruning, the final workflow-level search spaces contain 146,751, 1,074,866, 13,319, and 4,017 configurations for GSM8K, MATH-500, LiveCodeBench, and HotpotQA, respectively. Since the remaining search is purely analytic, it completes in 0.16s, 0.99s, 0.05s, and 0.01s on a CPU for the four benchmarks.

C.3 Scalability Discussion

The main scalability advantage of FlowCompile is that expensive model inference is performed at the sub-agent level rather than at the workflow-configuration level. For a fixed profiling set, the profiling cost scales with the number of sub-agent roles and candidate model–budget settings, i.e., AMR in the homogeneous case or $\sum_{a \in \mathcal{A}} |\mathcal{Q}_a|$ in the heterogeneous case. In contrast, the full workflow design space grows combinatorially. Even without structural choices, independently assigning one of MR model–budget settings to each of A sub-agent roles yields $(MR)^A$ workflow configurations; optional structural choices further enlarge this space. FlowCompile avoids this combinatorial inference cost by profiling each sub-agent choice once and reusing the cached profiles to estimate many workflow-level configurations through lightweight numerical composition.

This separation makes the method scalable in practice. The expensive profiling calls are independent across sub-agents, examples, models, and reasoning budgets, and can therefore be parallelized across accelerators. Once collected, the profiles are reused across all workflow-level configurations, deployment preferences, and latency budgets. The remaining workflow-level composition and trade-off set construction require only lightweight numerical computation over cached profiles and complete in under one second on a CPU in our experiments.

Overall, FlowCompile shifts workflow optimization from combinatorial end-to-end evaluation to linear, reusable, and parallelizable sub-agent profiling. This makes compile-time exploration scalable even when the full workflow design space contains millions or billions of configurations.

Table 5: **Representative accuracy–latency operating points.** Bold marks the best value excluding the full baseline; MaAS (Qwen3-4B) on MATH-500 is scored on the 401 examples with completed outputs. FlowCompile outputs a set of optimized configurations, from which we report accuracy-priority and latency-priority selections. The two selections expose different accuracy–latency trade-offs from the same compiled set, relative to the full Qwen3-14B workflow baseline.

Method	GSM8K		MATH-500		HotpotQA		LiveCodeBench	
	Accuracy	Latency (s)	Accuracy	Latency (s)	F1	Latency (s)	Pass@1	Latency (s)
Full Baseline (Qwen3-14B)	95.55	168.3	96.52	355.5	86.29	26.5	80.97	329.3
Qwen3-4B	95.64	94.4	93.78	187.4	83.89	12.5	78.69	186.2
Qwen3-8B	95.17	134.8	**96.77**	352.3	86.58	18.5	**79.69**	267.2
MaAS (Qwen3-4B)	92.42	30.5	71.82	**55.0**	71.82	6.3	26.56	33.7
MaAS (Qwen3-8B)	92.80	47.9	70.40	87.1	83.44	14.1	33.81	58.6
KNN Router	89.00	42.5	78.61	82.0	73.75	6.9	50.14	141.3
Ours (Acc.-first)	**96.02**	88.3	95.27	185.5	**86.69**	7.8	78.98	51.5
Ours (Lat.-first)	92.13	**10.9**	88.31	98.0	85.32	**3.8**	41.34	**19.0**

D Additional Details on Frontier Consistency in Restricted Spaces

To further evaluate frontier consistency in Section 4.2, we compare proxy-selected configurations with evaluated alternatives on HotpotQA and LiveCodeBench under restricted configuration spaces. We then compare the proxy-estimated frontier with the empirically measured accuracy–latency trade-off space.

Restricted design space. For HotpotQA, we restrict the model choices to Qwen3-1.7B, Qwen3-8B, and Qwen3-14B, and the reasoning budgets to 10 and 10,000 tokens. We consider two workflow structures: the full workflow and a reduced workflow with one generator and one formatter. For LiveCodeBench, we restrict the model choices to Qwen3-4B and Qwen3-8B, and the reasoning budgets to 10 and 10,000 tokens. We consider two workflow structures: the full workflow and a reduced workflow with a single programmer. For each restricted template, we enumerate its admissible role-level model–budget assignments and evaluate the resulting configurations end-to-end. These restricted spaces preserve the key design dimensions of the full search space, including model choice, reasoning budget, and workflow structure, while keeping workflow-level comparisons tractable.

Results. As shown in Figure 2, the proxy-estimated frontier on HotpotQA matches the measured frontier: the proxy-selected configurations dominate every other configuration in the figure. Figure 6 shows a similar pattern on LiveCodeBench: the configurations selected by the proxy form a high-quality empirical frontier after workflow execution. The selected configurations cover the main trade-off regimes from low-latency to high-accuracy operating points, and on both benchmarks the selected set's best measured utility is within 0.003 of the best evaluated configuration for every $\alpha \in [0, 1]$ and any common positive latency normalizer. Because the evaluated alternatives include every uniform model–budget assignment, this also compares the compiled set against uniform budget sweeps over the same models and structures.

Comparison with equal-size random subsets. We use the same evaluated configurations to quantify how compact-set quality compares with selecting points uniformly without replacement. For a subset of size k, let $u_{(0)} \le \cdots \le u_{(n-1)}$ be the utilities of all n evaluated configurations at a fixed preference. Its expected best-in-set utility is

$$\mathbb{E}\left[\max_{c \in S_k} U_\alpha(c)\right] = \sum_{j=k-1}^{n-1} \frac{\binom{j}{k-1}}{\binom{n}{k}} u_{(j)}.$$

Using the primary shared benchmark normalizers (Appendix J) and averaging over $\alpha = 0.05, 0.10, \ldots, 0.95$, random subsets of the same size as the compiled sets have expected gaps to the best evaluated configuration of 6.79 utility points on HotpotQA and 3.68 points on LiveCodeBench, whereas the compiled sets have zero gap on this grid. Random subsets need to be twice as large to reach gaps of 4.57/2.20 points, and four times as large to reach 2.92/1.11 points.

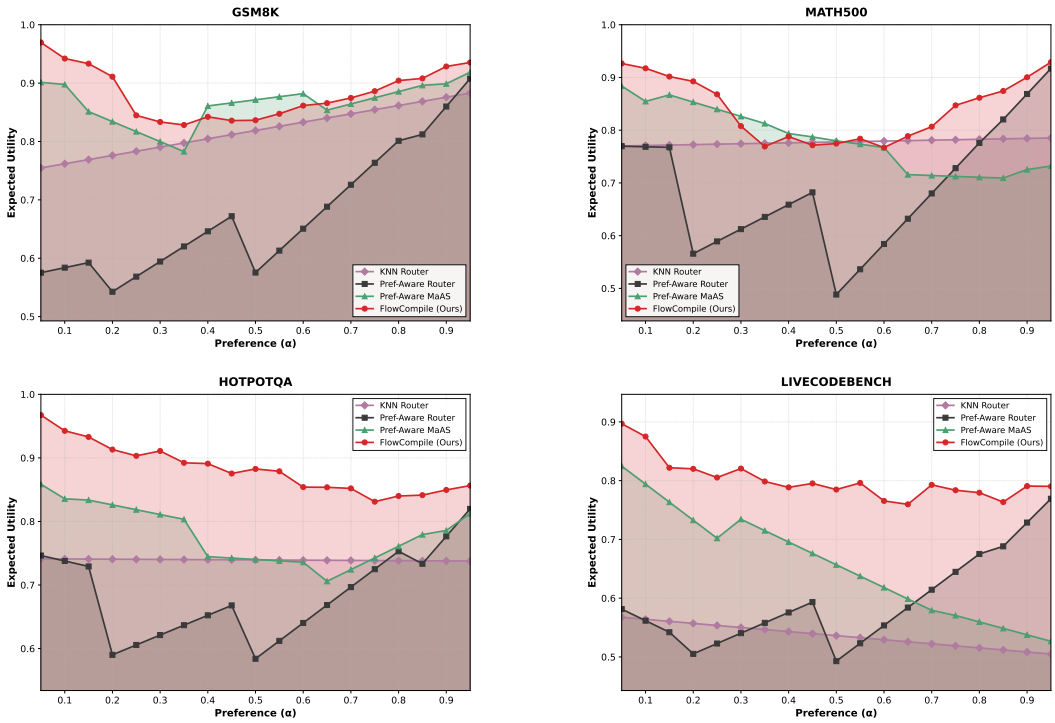


Figure 7: Per-benchmark average expected utility under fixed accuracy–latency preferences.

E Additional Details on Accuracy–Latency Trade-offs

Table 5 provides numerical accuracy and latency results, enabling a direct comparison of FlowCompile and the baselines in the accuracy–latency trade-off space. Since FlowCompile outputs a compiled set of configurations rather than a single operating point, we report two representative selections from this set. The accuracy-priority selection aims to match the full Qwen3-14B workflow baseline while reducing latency, whereas the latency-priority selection prioritizes low latency.

The accuracy-priority configuration reduces latency with small observed task-score differences from the full baseline. For example, on HotpotQA, it improves F1 from 86.29 to 86.69 while reducing latency from 26.5s to 7.8s. On LiveCodeBench, it reduces latency from 329.3s to 51.5s while maintaining similar Pass@1. The latency-priority configuration yields larger speedups, especially on GSM8K and HotpotQA, while retaining competitive accuracy. These results complement the frontier plots in Figure 3 by showing concrete operating points selected from the compiled configuration set.

F Additional Details on Preference-Aware Evaluation

We provide additional details on the preference-aware evaluation protocol. FlowCompile constructs its candidate set from proxy-estimated accuracy and latency. Deployment uses the predicted-latency ordering of that set, and reported utilities use held-out outcomes.

Heterogeneous preferences. In the heterogeneous setting used in Table 2, each test query is assigned an independently sampled preference parameter $\alpha \sim \mathrm{Uniform}(0, 1)$. This setting represents mixed workloads containing both latency-sensitive and accuracy-sensitive requests. For each query, FlowCompile maps the sampled α to a normalized rank in the predicted-latency-ordered compiled set (Appendix J). The reported utility uses the selected configuration's measured test-set accuracy and latency. We repeat the sampling ten times and report the mean and standard deviation.

Fixed preferences. In the fixed-preference setting, all queries share the same α. We sweep α from 0.05 to 0.95 with a step size of 0.05 to evaluate how robust each method is under different global accuracy–latency preferences. For each α, FlowCompile applies the same latency-rank selection rule

Table 6: **Per-query routing over compiled configurations.** A lightweight KNN router further improves expected utility by selecting among the configurations produced by FlowCompile for each query.

Method	GSM8K	MATH-500
FlowCompile	89.2 ± 0.6	84.2 ± 0.8
FlowCompile + KNN Routing	$\mathbf{91.8 \pm 0.5}$	$\mathbf{90.5 \pm 0.9}$

and reports the selected configuration's measured test-set utility. Figure 7 reports the per-benchmark results for this setting.

Across benchmarks and most preference values, FlowCompile achieves the highest expected utility, indicating that the compiled trade-off set provides robust operating points across diverse accuracy–latency preferences.

G Additional Details and Analysis on Per-Query Routing

Setup and main result. FlowCompile performs compile-time optimization and produces a global accuracy–latency trade-off set. This compiled set is complementary to runtime routing: instead of routing over the full combinatorial design space, a router only needs to select among the compact set of configurations produced by FlowCompile.

We evaluate this use case on GSM8K and MATH-500 using a simple KNN router with $k = 20$. For each test query, the router uses Longformer (Beltagy et al., 2020) embeddings to retrieve nearest neighbors from the validation set, and then selects a configuration from the compiled set based on the neighbors' performance. The required neighborhood statistics are available from query-linked, per-example component profiles: averaging these over the retrieved neighbors and applying the workflow proxy yields neighborhood-level predictions for a configuration. This adds only lightweight query-level selection and requires no additional workflow profiling, online optimization, or retraining.

As shown in Table 6, routing over the compiled set further improves expected utility from 89.2 to 91.8 on GSM8K and from 84.2 to 90.5 on MATH-500. These results show that FlowCompile can be used either directly as a compiled deployment artifact or as a high-quality candidate pool for downstream per-query adaptation.

Analysis setup. To better understand how query-level selection uses the compiled set, we further analyze the routing assignments on MATH-500, since it provides difficulty labels that support post hoc interpretation. The difficulty labels are not used by the router during selection; they are used only to examine whether the selected configurations correlate with problem difficulty. For each test query and preference value α, the assignment artifact records the selected compiled workflow configuration, including its workflow structure, sub-agent model choices, and reasoning budgets.

We group preference values into three regimes: Latency-first ($\alpha \leq 0.3$), Balanced ($0.4 \leq \alpha \leq 0.6$), and Accuracy-first ($\alpha \geq 0.7$). We also classify the selected workflow structures into Simple, Balanced, and Complex classes, corresponding to two, three, and four active sub-agents in the MATH-500 workflow, respectively.

Routing assignment patterns. Figure 8 shows that per-query routing does not select a single global configuration for all MATH-500 problems. Instead, it induces different configuration distributions across difficulty levels and preference regimes, even though difficulty labels are not available to the router. In the Latency-first regime, all difficulty levels are dominated by Simple workflows, and the mean proxy latency remains low. Thus, when latency is prioritized, the router preserves inexpensive execution even for harder questions.

The difficulty-conditioned pattern becomes more pronounced in the Balanced and Accuracy-first preference regimes. In the Balanced regime, the share of Complex workflows increases from 1.9% for level-1 questions to 11.4% for level-5 questions, while mean proxy latency rises from 10.6s to 15.9s. In the Accuracy-first regime, the Complex workflow share further increases from 18.6% for level-1 questions to 39.6% for level-5 questions, with mean proxy latency increasing from 20.8s to

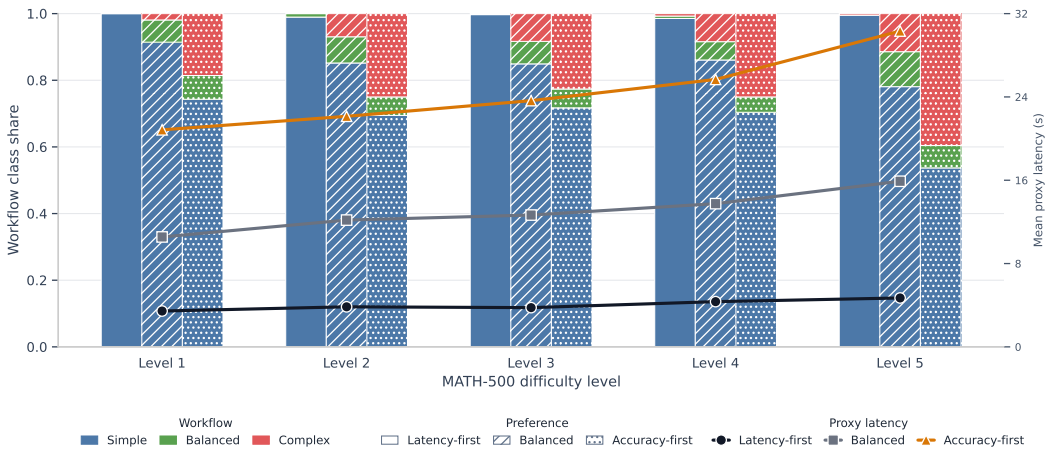


Figure 8: **Routing assignment patterns on MATH-500.** Difficulty labels are used only for post hoc analysis and are not observed by the router during selection. For each difficulty level, stacked bars show the workflow-class distribution of selected configurations under Latency-first, Balanced, and Accuracy-first preference regimes; hatching distinguishes preference regimes and colors indicate workflow classes. Lines with markers report the corresponding mean proxy latency. The selected configurations remain mostly Simple under Latency-first preferences, but shift more often toward Balanced or Complex workflows for harder questions as accuracy is prioritized.

30.3s. This indicates that the router spends more workflow-level compute on harder questions, but mainly when the preference value makes the additional latency worthwhile.

Latency allocation by difficulty. Figure 9 provides a complementary view by directly comparing the mean proxy latency across MATH-500 difficulty levels, with and without per-query routing. FlowCompile without routing applies a single global configuration, and therefore operates at essentially the same latency level for all difficulty levels. In contrast, FlowCompile + KNN Routing produces a difficulty-sensitive latency profile: easier questions are assigned cheaper configurations, while harder questions receive progressively more expensive ones.

This comparison shows that per-query routing changes how configurations are selected within the compiled set. Without routing, FlowCompile selects configurations according to the global preference objective, so the selected latency is nearly constant across MATH-500 difficulty levels after grouping by difficulty. In contrast, KNN routing makes query-conditioned selections from the same compiled set: easier questions receive lower-latency configurations, while harder questions receive relatively higher-latency configurations. Although the routed configurations remain lower-latency than the non-routed FlowCompile selection across all difficulty levels, the increasing trend shows that routing decisions correlate with problem difficulty, although difficulty labels are never observed by the router.

Implication. Together, these analyses illustrate the complementarity between compile-time optimization and runtime routing. FlowCompile first constructs a compact set of high-quality configurations spanning workflow structures, model choices, and reasoning budgets. The KNN router then performs lightweight query-conditioned selection within this set. Although the router does not observe difficulty labels, its selected configurations correlate with problem difficulty: easier questions are more often assigned to simpler, lower-latency compiled workflows, while harder MATH-500 questions are routed more frequently to configurations with additional reasoning branches and higher latency when extra compute is likely to help. The improvement from per-query routing therefore comes from using the compiled set as a query-conditioned deployment menu, rather than retraining the router or searching the full configuration space online.

H Additional Details on Reference-Model Ablation

FlowCompile uses a reference model to induce sub-agent-level profiling data from workflow traces. We evaluate whether the compiled trade-off set is sensitive to this reference model choice. In

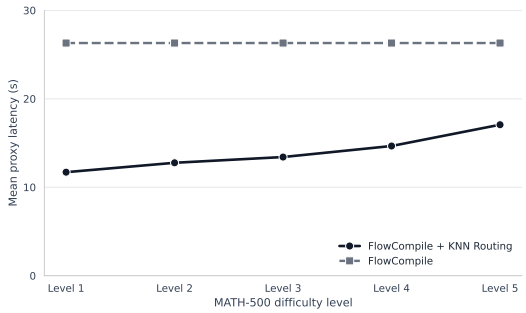


Figure 9: **Mean proxy latency by MATH-500 difficulty level, with and without per-query routing.** Difficulty labels are used only for post hoc analysis and are not observed by the router during selection. FlowCompile without routing applies a single global configuration, resulting in an approximately constant latency across difficulty levels. In contrast, FlowCompile + KNN Routing produces a difficulty-sensitive latency profile: easier questions are assigned lower-latency configurations, while harder questions receive progressively higher-latency configurations.

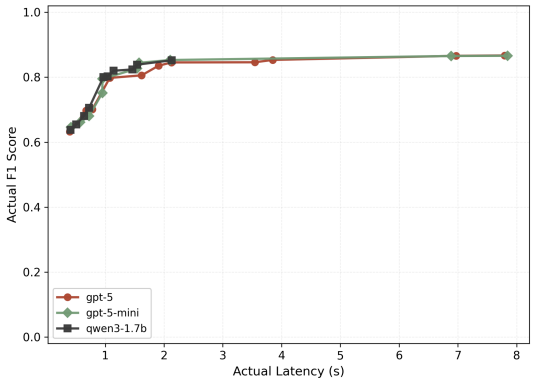


Figure 10: **Reference-model ablation on HotpotQA.** FlowCompile produces similar optimized trade-off sets when sub-agent data are induced using different reference models, suggesting that the compilation pipeline is robust to the reference model choice.

addition to the default GPT-5 reference model, we repeat the sub-agent data induction, profiling, compositional estimation, and frontier search pipeline on HotpotQA using GPT-5-mini and Qwen3-1.7B as alternative reference models.

Figure 10 shows that the resulting optimized configuration sets achieve largely consistent measured performance across reference models. This suggests that FlowCompile is not tightly coupled to a particular high-capacity reference model, provided that the reference model can generate reasonable workflow traces for sub-agent profiling.

I Analysis of Compiled Workflow Configurations

We further inspect the compiled configuration sets for all four benchmarks to characterize how FlowCompile allocates workflow structure, reasoning budget, and model capacity across different accuracy–latency operating regimes.

Analysis setup. We group compiled configurations into three accuracy–latency trade-off regions: Latency-priority, Balanced-priority, and Accuracy-priority, corresponding to low-latency, intermediate, and high-accuracy regions of the compiled set, respectively. These trade-off regions describe where a configuration lies on the compiled accuracy–latency frontier and are distinct from workflow-structure classes.

Table 7: **Qualitative structure of compiled workflow configurations.** We summarize dominant low-latency and high-accuracy patterns using Simple, Balanced, and Complex workflow classes.

Benchmark	Latency-priority pattern	Accuracy-priority pattern	Main trade-off mechanism
GSM8K	Mostly Simple workflows with short budgets and smaller models.	Mostly Complex workflows with larger budgets and stronger solving or aggregation models.	FlowCompile activates additional math reasoning stages and allocates more capacity to solving and aggregation.
MATH-500	Mostly Simple workflows with modest budgets.	Mostly Complex workflows with substantially larger budgets.	FlowCompile spends latency on additional reasoning paths for harder math problems and shifts capacity toward stronger solving and aggregation calls.
HotpotQA	A Simple fixed-size workflow with smaller models and short budgets.	The same Simple workflow with stronger model assignments and larger budgets.	The workflow structure remains stable; the trade-off is mainly governed by model and budget allocation within answer generation and formatting.
LiveCodeBench	Repair-based workflows with short generation and repair budgets in the low-latency region.	Complex repair workflows with substantially larger code-generation and repair budgets.	Accuracy is improved primarily by allocating more compute and stronger models to code generation and execution-guided repair.

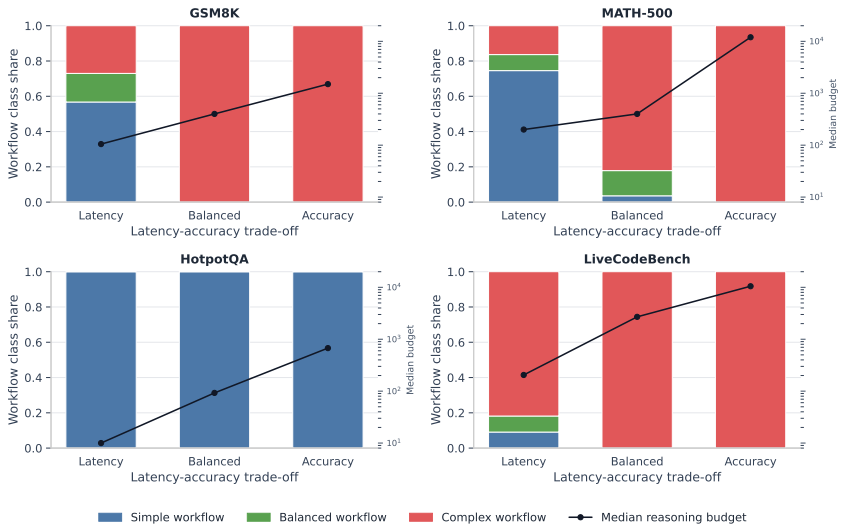


Figure 11: Workflow-structure patterns across latency-ranked regions of the full compiled artifacts. For each benchmark, configurations are grouped into low-latency, middle, and high-latency regions. Stacked bars show the proportion of Simple, Balanced, and Complex workflow classes, and black markers show the median reasoning budget across configured sub-agent calls on a log scale.

We also use a benchmark-specific three-way structural classification. For GSM8K and MATH-500, Simple, Balanced, and Complex correspond to configurations with two, three, and four active sub-agents, respectively. For LiveCodeBench, the number of active sub-agents in the compiled set remains fixed across configurations, so Simple, Balanced, and Complex correspond to one, two, and three maximum repair attempts. For HotpotQA, the compiled set uses a fixed two-sub-agent workflow, so the structural class remains Simple, and the main variation comes from model and budget allocation.

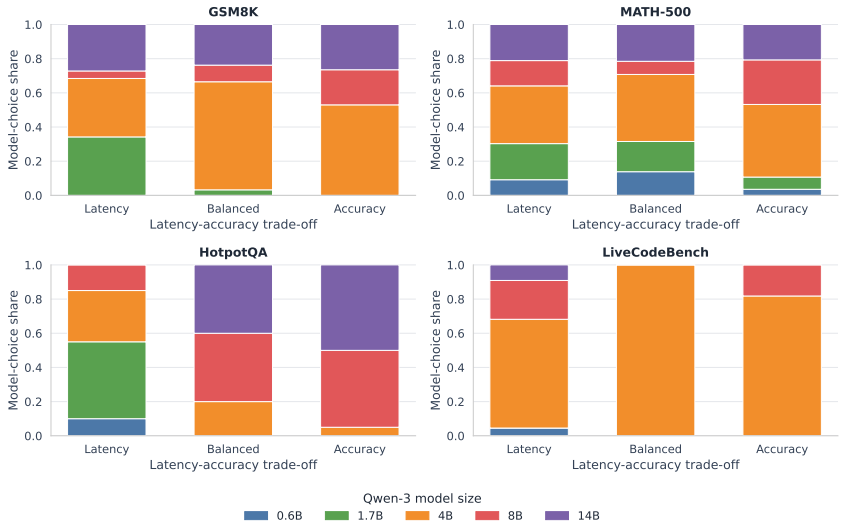


Figure 12: Model choices across latency-ranked regions of the full compiled Pareto artifacts. Stacked bars show the distribution of Qwen-3 model sizes used by configured sub-agent calls in each region. Moving from latency-priority to accuracy-priority configurations generally shifts capacity from smaller and mid-sized models toward 8B and 14B models, with task-dependent allocation patterns.

Overall pattern. Table 7 summarizes the dominant patterns for each benchmark, while Figures 11 and 12 provide aggregated views of workflow structure, reasoning budget, and model choices across the three trade-off regions. Across benchmarks, the compiled configurations exhibit interpretable structure rather than arbitrary combinations of models, budgets, and workflow variants. Moving from Latency-priority to Accuracy-priority configurations is mainly explained by three changes: the workflow becomes more complex when additional reasoning stages are useful, reasoning budgets increase for task-critical sub-agents, and model assignments shift toward stronger models. Low-latency configurations therefore tend to use simpler workflows, smaller or mid-sized models, and short reasoning budgets, while high-accuracy configurations allocate more compute to bottleneck roles such as solving, aggregation, formatting, refinement, or repair.

Math reasoning workflows. GSM8K and MATH-500 show the clearest structural transition. In the latency-priority region, FlowCompile often selects Simple configurations. As the objective shifts toward accuracy, the compiled set increasingly favors Complex configurations. This suggests that, for math reasoning, accuracy is not obtained only by uniformly scaling every LLM call. Instead, FlowCompile selectively activates additional reasoning stages and allocates larger budgets or stronger models to stages that provide complementary reasoning signals.

Multi-hop QA workflows. HotpotQA follows a different pattern. The compiled configurations keep the same Simple workflow class across the trade-off range. Low-latency configurations use smaller models and shorter budgets, while high-accuracy configurations increase capacity within the same workflow structure. This suggests that, for this workflow, the dominant optimization choice is not whether to activate additional sub-agents, but how much capacity to allocate to answer generation and final formatting.

Code reasoning workflows. LiveCodeBench also keeps a stable set of active sub-agents, but exposes a different structural knob through the number of execution-guided repair attempts. Latency-priority configurations use shorter generation and repair budgets, while balanced and accuracy-priority configurations are dominated by more repair trials. The main difference between balanced and accuracy-priority regions is therefore budget and model allocation: when accuracy is prioritized, FlowCompile spends substantially more compute on producing and repairing executable code.

Implication. Together, Table 7, Figure 11 and Figure 12 show that the compiled set can be interpreted as a deployment menu rather than a collection of unrelated operating points. Latency-priority configurations remove optional reasoning stages or use small-budget versions of the same

task skeleton, while accuracy-priority configurations spend compute where the workflow benefits most: additional reasoning paths for math, stronger generation and formatting for question answering, and more repair attempts together with larger generation and repair budgets for code reasoning. This supports the compiler view of FlowCompile: its output is a reusable set of workflow-level configurations spanning qualitatively different operating regimes.

J Metric Description

J.1 Expected Utility

We use expected utility to summarize the accuracy–latency trade-off under different deployment preferences. For a workflow configuration c, let $\mathrm{Acc}(c)$ denote its task performance and $\mathrm{Lat}(c)$ denote its measured latency. We first convert latency into a latency-efficiency score,

$$\mathrm{LES}(c) = 1 - \frac{\mathrm{Lat}(c)}{L_{\max}}, \tag{3}$$

where $L_{\max}$ is the fixed Qwen3-14B workflow baseline's mean latency, shared by all methods within a benchmark. Expected utility is then defined as

$$U(c;\alpha) = \alpha\,\mathrm{Acc}(c) + (1-\alpha)\,\mathrm{LES}(c), \tag{4}$$

where $\alpha \in (0,1)$ controls the accuracy–latency preference. Smaller α places more weight on latency efficiency, while larger α places more weight on accuracy. For QA tasks, $\mathrm{Acc}(c)$ denotes the F1 score; for LiveCodeBench, it denotes Pass@1.

For a method with one fixed configuration, we directly report its measured test-set utility. For a compiled candidate set $S = \{c_{(0)}, \ldots, c_{(N-1)}\}$ sorted by increasing predicted latency, our preference-dependent selection is

$$\hat{c}_\alpha = c_{(\mathrm{round}(\alpha(N-1)))}, \qquad U(S;\alpha) = U(\hat{c}_\alpha;\alpha).$$

This moves from faster to slower operating points as accuracy receives greater weight. The rule uses only the predicted ordering, so it is insensitive to proxy calibration and to the latency normalizer. For FlowCompile, $S = \widehat{\mathcal{F}}$, and held-out scores are used only to evaluate the selected point. Preference-aware baselines apply the same normalized-rank mapping to their latency-ordered candidate menus; their routing parameters remain fixed across preferences. The shared $L_{\max}$ is used for utility scoring, not rank selection. Its values are 168.314, 355.523, 26.492, and 329.349 seconds for GSM8K, MATH-500, HotpotQA, and LiveCodeBench, respectively.

In the heterogeneous-preference setting, α is sampled independently for each query and results are averaged across repeated samples. In the fixed-preference setting, a single α is shared across all queries and swept over $(0,1)$.

J.2 Proxy Validation Metrics

Let $\mathcal{C} = \{c_1, \ldots, c_N\}$ denote the set of evaluated workflow configurations. For each configuration c_i, let $\hat{y}_i$ and y_i denote the estimated and empirically measured values of a given metric, respectively.

Spearman Correlation. Spearman correlation measures the consistency between the relative orderings induced by the estimated and measured values. Let $\mathrm{rank}(\hat{y}_i)$ and $\mathrm{rank}(y_i)$ denote the ranks of $\hat{y}_i$ and y_i among all configurations in $\mathcal{C}$. The Spearman rank correlation coefficient is defined as

$$\rho = \frac{\sum_{i=1}^{N}\left(\mathrm{rank}(\hat{y}_i) - \overline{\mathrm{rank}(\hat{y})}\right)\left(\mathrm{rank}(y_i) - \overline{\mathrm{rank}(y)}\right)}{\sqrt{\sum_{i=1}^{N}\left(\mathrm{rank}(\hat{y}_i) - \overline{\mathrm{rank}(\hat{y})}\right)^2 \sum_{i=1}^{N}\left(\mathrm{rank}(y_i) - \overline{\mathrm{rank}(y)}\right)^2}}, \tag{5}$$

where $\overline{\mathrm{rank}(\hat{y})}$ and $\overline{\mathrm{rank}(y)}$ denote the mean ranks. Spearman correlation takes values in $[-1,1]$, with higher values indicating stronger agreement in relative ordering.

Pairwise Agreement. Pairwise agreement directly measures the fraction of configuration pairs whose relative ordering is correctly preserved. Formally, it is defined as

$$\mathrm{PA} = \frac{1}{\binom{N}{2}} \sum_{1 \le i < j \le N} \mathbb{I}\left[\mathrm{sign}(\hat{y}_i - \hat{y}_j) = \mathrm{sign}(y_i - y_j)\right], \tag{6}$$

where $\mathbb{I}[\cdot]$ is the indicator function. Pairwise agreement lies in $[0, 1]$ and equals 1 if and only if the estimated values induce exactly the same total ordering as the measured values.

Calibrated MAE. Raw mean absolute error (MAE) is sensitive to systematic discrepancies between estimated and measured values, including both global offsets and scale mismatches. Since our objective is to evaluate how well the compiler captures configuration-dependent accuracy–latency trade-offs, we apply an affine calibration to remove systematic bias and scale distortion before computing MAE. We use a two-point calibration rather than fitting an affine regressor over all measured configurations because the latter would require measuring many workflow configurations, whereas our goal is to assess calibrated absolute error under a minimal-calibration setting consistent with proxy-based compilation.

Formally, let $i_{\min} = \arg\min_i \hat{y}_i$ and $i_{\max} = \arg\max_i \hat{y}_i$ denote the indices of the configurations with the minimum and maximum estimated values, respectively. We define

$$\hat{y}_{\min} = \hat{y}_{i_{\min}}, \quad \hat{y}_{\max} = \hat{y}_{i_{\max}}, \quad y_{\min} = y_{i_{\min}}, \quad y_{\max} = y_{i_{\max}}. \tag{7}$$

That is, the calibration anchors are determined by the extrema of the predicted values, and the corresponding measured values are taken from the same configurations.

We then define an affine mapping that aligns these two anchor points:

$$\hat{y}_i^{\text{cal}} = \frac{y_{\max} - y_{\min}}{\hat{y}_{\max} - \hat{y}_{\min}} (\hat{y}_i - \hat{y}_{\min}) + y_{\min}. \tag{8}$$

The calibrated MAE is computed as

$$\text{cMAE} = \frac{1}{N} \sum_{i=1}^{N} \left| \hat{y}_i^{\text{cal}} - y_i \right|. \tag{9}$$

This two-point calibration is a post hoc diagnostic computed on the evaluated configurations, including the two anchors. It is not applied when selecting configurations for the expected-utility experiments, and cMAE is not a held-out calibration error. A positive fitted slope preserves the ordering of each individual metric; separate rescaling of accuracy and latency need not preserve their weighted utility ordering. We therefore report rank diagnostics alongside cMAE and assess preference-aware deployment using the measured utility of configurations selected from the original proxy estimates.

K LLM-AS-A-JUDGE PROTOCOLS

FlowCompile uses LLM-as-a-judge evaluation in two distinct stages. During sub-agent data generation, the judge filters intermediate calls from reference-model workflow traces and retains calls that are well-executed and useful for producing a successful final answer. During isolated sub-agent profiling, the judge evaluates outputs from candidate model–budget configurations by comparing them against the induced pseudo target for the corresponding sub-agent, without observing downstream workflow traces. All judge-based filtering and profiling are performed only on the profile set.

K.1 PROMPT FOR SUB-AGENT DATA GENERATION

```
You are an expert evaluator judging whether a specific AI agent call
↪  helped solve a problem within a multi-agent workflow.

**Original Problem:**
{agent_data['problem']}

**Ground Truth Answer:**
{agent_data['ground_truth']}

**Workflow Final Answer:**
{final_answer}{quality_info}

**Full Workflow Trace (showing all steps leading to the solution):**
{trace_context}
```

```
**Current Agent Being Evaluated:**
- Agent Name: {agent_data['agent_name']}
- Step Number: {agent_data['step_number']}
- Agent-Level Input: {agent_data['agent_input']}

**What the LLM Received (Raw Prompt):**
{agent_data['raw_llm_prompt']}

**What the LLM Generated (Raw Output):**
{agent_data['raw_llm_output']}

**Final Agent Output (after any post-processing):**
{actual_agent_output}

**Evaluation Task:**
Determine if THIS SPECIFIC agent call was helpful and well-executed.
↪  Consider:

1. **Understanding**: Did the agent correctly understand its task from
↪  the input prompt?
2. **Execution Quality**: Is the agent's raw output (what the LLM
↪  generated) sound, logical, and well-reasoned?
3. **Correctness**: Does the output align with the ground truth and
↪  contribute to a high-quality final answer?
4. **Value**: In the context of the full workflow, did this agent's
↪  contribution help?
5. **Intermediate Utility**: If this is an intermediate step, did it
↪  provide useful information for subsequent agents?

**Important Notes:**
- The overall workflow produced a HIGH-QUALITY answer
- Your job is to assess if THIS PARTICULAR agent call was done well
- An agent can be marked correct even if it's an intermediate step (e.g.,
↪  programmer generating code, answer_generate producing one of multiple
↪  candidate solutions)
- Focus on: Was the agent's output reasonable, helpful, and properly
↪  executed given its role?

Respond in JSON format:
{{
    "is_correct": true/false,
    "reasoning": "Brief explanation (2-3 sentences) of whether this agent
    ↪  call was helpful and well-executed"
}}
```

K.2 Prompt for Isolated Sub-Agent Profiling

After sub-agent data generation, FlowCompile profiles each candidate model–budget configuration independently on the induced sub-agent dataset. This stage does not generate or observe downstream workflow traces. For each induced example, the profiled configuration receives the same sub-agent input and produces a candidate output, which is compared against the induced pseudo target output for that sub-agent role. When exact or executable evaluation is available, we use deterministic evaluation instead of an LLM judge, such as normalized answer matching for math, task F1 for HotpotQA final answers, and execution against public test cases for LiveCodeBench programs. The profiling judge is used only when deterministic matching is not applicable, such as for evaluating aggregation outputs. We use GPT-OSS-120B for this isolated-profile judge.

```
You are an expert evaluator for a sub-agent in a structured LLM workflow.

**Original Problem:**
{agent_data['problem']}

**Ground Truth Answer:**
```

```
{agent_data['ground_truth']}

**Current Agent Being Evaluated:**
- Agent Name: {agent_data['agent_name']}
- Agent Description: {agent_data['agent_description']}
- Agent-Level Input: {agent_data['agent_input']}

**Pseudo Target Output:**
{pseudo_target_output}

**Candidate Output:**
{candidate_output}

**Evaluation Task:**
Determine whether the Candidate Output is correct for this sub-agent role.
↪  A correct output should be semantically equivalent to the Pseudo
↪  Target Output or provide the same useful information needed by
↪  downstream workflow stages. Consider:

1. **Role Fulfillment**: Does the Candidate Output fulfill the intended
↪  role of this sub-agent?
2. **Semantic Correctness**: Is it semantically equivalent to, or at
↪  least as useful as, the Pseudo Target Output?
3. **Completeness**: Does it contain the key information needed by
↪  downstream stages?
4. **Format Compatibility**: Is the output in a format that can be
↪  consumed by the subsequent workflow stage?

**Important Notes:**
- Do not require identical wording.
- Focus on correctness, usefulness, and compatibility with the sub-agent
↪  role.
- Do not judge whether the entire workflow would succeed; judge only this
↪  sub-agent output relative to its induced pseudo target.
- If the Candidate Output is partially correct but misses key information
↪  needed by downstream stages, mark it incorrect.

Respond in JSON format:
{{
    "is_correct": true/false,
    "reasoning": "Brief explanation (1-2 sentences) of whether this
    ↪  candidate output is correct for the sub-agent role"
}}
```

The held-out test set is used only for final workflow evaluation and is never used for sub-agent data generation, sub-agent profiling, configuration selection, or judge-based filtering.

L Selection Error and Candidate-Space Coverage

L.1 A deterministic selection-error bound

Fix a nonempty finite candidate set S, preference α, and latency normalizer $L_{\max} > 0$. Let

$$U_\alpha(c) = \alpha \mathrm{Acc}(c) + (1-\alpha)\left(1 - \frac{\mathrm{Lat}(c)}{L_{\max}}\right),$$

with $\widehat{U}_\alpha(c)$ computed from the proxy estimates using the same α and $L_{\max}$. Here measured performance refers to the fixed evaluation data; the following is a deterministic comparison, not a population generalization bound. Define $e_\alpha(c) = \widehat{U}_\alpha(c) - U_\alpha(c)$, and choose any

$$c^*_\alpha \in \arg\max_{c \in S} U_\alpha(c), \qquad \hat{c}_\alpha \in S, \qquad \delta_\alpha = \max_{c \in S} \widehat{U}_\alpha(c) - \widehat{U}_\alpha(\hat{c}_\alpha).$$

Then Equation 2 holds, including in the presence of ties.

Proof. Optimality of c^*_α gives $U_\alpha(c^*_\alpha) - U_\alpha(\hat{c}_\alpha) \geq 0$. The definition of δ_α gives $\widehat{U}_\alpha(c^*_\alpha) - \widehat{U}_\alpha(\hat{c}_\alpha) \leq \delta_\alpha$. Substituting the definition of e_α yields

$$\begin{aligned} U_\alpha(c^*_\alpha) - U_\alpha(\hat{c}_\alpha) &= \widehat{U}_\alpha(c^*_\alpha) - \widehat{U}_\alpha(\hat{c}_\alpha) + e_\alpha(\hat{c}_\alpha) - e_\alpha(c^*_\alpha) \\ &\leq \delta_\alpha + e_\alpha(\hat{c}_\alpha) - e_\alpha(c^*_\alpha) \\ &\leq \delta_\alpha + \max_{c \in S} e_\alpha(c) - \min_{c \in S} e_\alpha(c). \end{aligned}$$

□

Shared error and pairwise reversals. If $e_\alpha(c) = b_\alpha + r_\alpha(c)$ for a constant b_α, the bound depends only on the range of r_α. In particular, $|r_\alpha(c)| \leq \epsilon$ for every $c \in S$ implies utility loss at most $\delta_\alpha + 2\epsilon$, regardless of the shared offset. Predicted-utility maximization gives $\delta_\alpha = 0$. More locally, if $U_\alpha(c_i) > U_\alpha(c_j)$, their ordering can reverse only if

$$e_\alpha(c_j) - e_\alpha(c_i) \geq U_\alpha(c_i) - U_\alpha(c_j).$$

This identifies the relevant interaction between error variation and the margin separating configurations. It does not assume that all sub-agent outcomes are independent or that every pairwise ordering is preserved.

Relation to the metric errors. Writing $\Delta_A(c) = \widehat{\mathrm{Acc}}(c) - \mathrm{Acc}(c)$ and $\Delta_L(c) = \widehat{\mathrm{Lat}}(c) - \mathrm{Lat}(c)$ gives

$$e_\alpha(c) = \alpha \Delta_A(c) - \frac{1-\alpha}{L_{\max}} \Delta_L(c).$$

A configuration-independent offset in either metric therefore induces a shared utility offset. Configuration-dependent errors can change the preferred accuracy–latency trade-off. Separate rank correlations and calibrated MAEs describe aspects of the proxy but do not, by themselves, bound this utility error range. Section 4.3 evaluates deployed selections.

L.2 Selection, pruning, and coverage

For $S = \widehat{\mathcal{F}}$, Equation 2 compares the selected configuration with the best measured configuration in the compiled set. For any candidate subset $S \subseteq \mathcal{C}$, the gap to the best measured configuration in the full declared design space decomposes as

$$\begin{aligned} \max_{c \in \mathcal{C}} U_\alpha(c) - U_\alpha(\hat{c}_\alpha) = &\underbrace{\max_{c \in \mathcal{C}} U_\alpha(c) - \max_{c \in S} U_\alpha(c)}_{\text{candidate-space coverage gap}} \\ &+ \underbrace{\max_{c \in S} U_\alpha(c) - U_\alpha(\hat{c}_\alpha)}_{\text{selection error within } S}. \end{aligned}$$

When local replacement cannot decrease proxy accuracy or increase proxy latency, local pruning preserves the best predicted utility for every $\alpha \in (0, 1)$. Non-dominated sorting likewise preserves a predicted-utility maximizer. Thus, if the remaining legal configuration space is enumerated completely, maximizing predicted utility on its compiled frontier also maximizes predicted utility over $\mathcal{C}$. Equation 2 then applies with $S = \mathcal{C}$, but requires the error range over that full space. Measurements on a subset do not establish such a uniform bound. In particular, a bound for one scalarized preference is not a claim of recovering every non-dominated point of an arbitrary workflow.

Interpretation of compositional scores. For a sequential pair with upstream correctness event A and downstream final correctness event B, the exact identity is

$$P(B) = P(A)P(B \mid A) + (1 - P(A))P(B \mid \neg A).$$

A multiplicative score emphasizes the successful upstream path, while downstream recovery contributes through the second term. Likewise, reference-input profiles can differ from downstream performance under candidate-generated inputs, and parallel branches can share errors. These effects enter $e_\alpha(c)$ jointly. The bound clarifies when they affect configuration selection, rather than positing that their magnitude is always small. Frontier measurements and deployment utility provide complementary empirical checks of the compiler's output.

M LIMITATIONS

FlowCompile is designed for structured LLM workflows whose execution graphs and sub-agent interfaces are specified in advance. This scope matches many program-like workflow systems, but it is less directly applicable to open-ended agentic systems whose execution traces are dynamically constructed at inference time, such as ReAct-style agents with unbounded tool-use patterns (Yao et al., 2022). Extending workflow compilation to such dynamic settings would require additional mechanisms for trace abstraction or online workflow-graph construction.

Scope of the compositional proxy. The analytical proxy composes independently profiled sub-agents, which is most natural when sub-agent outcomes are loosely coupled. The evaluated workflows already contain the main sources of coupling in structured workflows, namely aggregation over parallel branches, sequential refinement, and bounded test-driven repair, and on these workflows the proxy preserves the measured accuracy and latency orderings (Section 4.2). Workflows with tighter dependencies, such as long chains in which upstream choices substantially shift downstream inputs or deeper iterative control flow, may benefit from richer estimators. Because selection depends on differences in proxy error rather than on absolute accuracy (Equation 2), such estimators, including conditional profiling on candidate-generated inputs or learned models of downstream recovery, can be integrated directly into the same compilation framework.