VineLM: Trie-Based Fine-Grained Control for Agentic Workflows

Nikos Pagonas
Columbia University
n.pagonas@columbia.edu

Matthew Lou
Columbia University
ml4855@columbia.edu

Tianyi Peng
Columbia University
tianyi.peng@columbia.edu

Dan Rubenstein
Columbia University
danr@cs.columbia.edu

Kostis Kaffes
Columbia University
kkaffes@cs.columbia.edu

Abstract

Agentic workflows interleave configurable LLM stages with tool stages and often include retries or refinement loops. Existing workflow managers profile full workflow configurations offline and assign each request a static workflow-level plan that binds each configurable LLM stage to a single model, reuses that model across repeated loop iterations, and does not revisit those choices at runtime. We present VineLM, a workflow manager that enables fine-grained control by choosing the model for each stage invocation as execution unfolds under request-level objectives such as maximizing accuracy under cost or latency budgets. VineLM represents feasible executions as an annotated trie of model-choice prefixes and uses checkpointing and cascade profiling to estimate path accuracy, cost, and latency without exhaustively profiling every request on every path. At runtime, VineLM re-roots the trie after each stage invocation and replans over the remaining subtrie using the realized execution prefix and remaining latency budget. On NL2SQL and math reasoning workflows, VineLM improves the cost-latency-accuracy frontier over coarse workflow-level baselines, achieving up to 18% higher accuracy at the same per-request budget with its sparse profiling reducing offline profiling cost by 98–99.8% when compared to exhaustive profiling.

1 Introduction

Agentic workflows: Modern AI applications increasingly take the form of agentic workflows rather than a single monolithic LLM call [24, 47, 54, 61]. A workflow is a stateful program composed of stages: some stages are configurable LLMs, while others are tools such as retrieval, SQL execution, or external API calls [41, 42, 46, 53, 57]. A request is handled by a sequence of stage invocations, and intermediate results determine what stage, if any, happens next, allowing the workflow to retry, refine, and maintain state across steps. A workflow is conceptually the analog of a program where stages are the instructions. Like programs, workflows can conceptually implement branching and looping [6, 48, 55, 60, 62]. Frameworks such as LangGraph [27], LlamaIndex Workflows [29], AG2 [56], among others [3, 5, 7, 16, 25, 39] already expose this programming model.

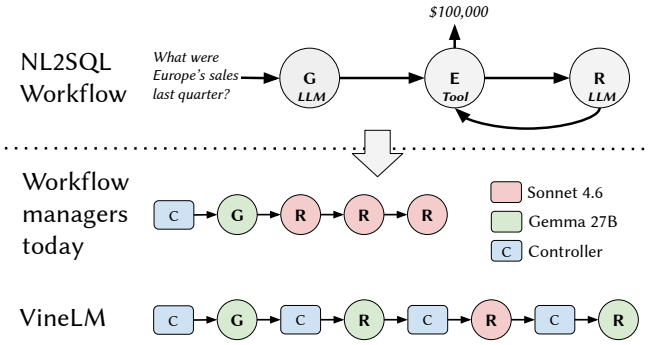

Figure 1. A simplified example NL2SQL workflow containing a SQL Generation LLM stage (G), a SQL Execution tool stage (E), and a potentially repeated Repair LLM stage (R). Existing workflow managers such as Murakkab [9] commit to one workflow-level model assignment when the request arrives, with repeated loop iterations always reusing the same model. VineLM instead reconsiders the choice after each stage and can mix LLMs across different iterations of the same loop. This finer-grain control yields a much larger decision space, but enables better end-to-end optimization.

A representative example is NL2SQL shown in Figure 1. Given a question such as "What were Europe's sales last quarter?", the workflow generates a candidate SQL query, executes it, and uses execution feedback (e.g., runtime error) or available offline information (e.g., I need two iterations to reach my target average accuracy) to decide whether to stop or continue in a bounded repair loop. Generation and Repair are configurable LLM stages, while SQL Execution is a tool stage. This feedback-and-repair pattern is now common well beyond NL2SQL, from self-refinement systems [33] to tool-using agents that improve through execution feedback and retry [49].

Workflow management today: Managing such workflows requires choosing a model for each configurable LLM stage, where models differ in cost, latency, and accuracy. State-of-the-art managers such as Murakkab [9] profile a space of full workflow configurations offline, where each configuration binds a specific model to each configurable

stage in the workflow template. Crucially, this binding is to the stage template, not to each dynamic stage invocation. If a workflow contains a loop, the manager does not unroll that loop into separate first-, second-, and third-iteration decisions; instead, it chooses one model for the stage and must reuse that same model on every loop iteration within the request. At serving time, each request is therefore assigned one static configuration before the workflow begins. In the NL2SQL example of Figure 1, the system can choose one model for Generation and a different model for Repair, but it cannot use intermediate execution feedback to switch to a different model on later repair rounds. We refer to this decision granularity as coarse-grained control: the manager chooses one workflow-level model assignment up front and keeps that assignment fixed for the lifetime of the request.

As we show in §2, coarse-grained control can severely limit performance. In refinement loops, the best model for an early repair attempt is often not the best model for a later one, and mixed-model trajectories can dominate plans that bind a single model to the repair stage for all iterations. Runtime variability makes static plans even weaker: a downstream choice that looked best at the start of the request can become the wrong one after an unexpectedly slow or expensive earlier stage. Current systems stop at offline workflow-level plans to avoid combinatorial explosion of the space to be considered. Once each refinement step is allowed to choose its model independently, the number of feasible paths grows rapidly. For example, in our NL2SQL workflow, with 8 candidate models at Generation and at each of two Repair rounds, the workflow already induces 584 feasible model-choice paths. Exhaustively profiling that space is expensive even before considering online adaptation.

Our approach: In contrast, this paper targets fine-grained control: choosing the model separately for each stage invocation as the workflow unfolds. Under fine-grained control, the first Repair invocation and the second Repair invocation are distinct decision points, even though they correspond to the same logical Repair stage in the workflow template.

Our key insight is to represent this space as an execution trie. Each node corresponds to a prefix of model choices for the stage invocations encountered so far in a workflow run, including repeated invocations of the same logical stage inside a refinement loop. Thus, every root-to-node path represents a feasible workflow prefix: if the workflow can terminate at that node, it is a complete feasible run; otherwise, it is a partial run that can be extended to one of the node's descendants. The trie exposes shared prefixes across many candidates, turning a large set of full plans into a structured search space. We show that this structure helps in two ways. First, it enables efficient offline annotation: using checkpointing, cascade-style profiling, and subtree reuse, we can estimate the accuracy, cost, and latency of many prefixes and terminating runs from a small number of sampled executions. Second, it enables efficient online adaptation: after each stage invocation completes, we can re-root the trie at the realized prefix, revise latency to reflect actual time spent thusfar, and search only the remaining subtrie, allowing VineLM to adapt later model choices to the observed progress of the request and, when available, the current system state.

We build VineLM, an agentic workflow manager. VineLM constructs an annotated trie offline from representative traces and uses it online to perform per-invocation model selection under request-level objectives such as maximizing accuracy under a budget or minimizing cost subject to an accuracy floor. Rather than commit to a single workflow-level plan at request start, VineLM interleaves execution and control throughout the workflow run. After each stage invocation completes, as depicted in Figure 1, the VineLM controller uses its offline estimates together with the latency so far to choose the model for the next stage invocation.

We evaluate VineLM on NL2SQL and math reasoning workflows. Our results show three main takeaways. First, per-invocation control materially improves the cost–latency–accuracy frontier over coarse workflow-level baselines, achieving up to 18% higher accuracy with the same available models and the same workflow under the same budget. Second, sparse cascade profiling annotates the trie accurately enough for control while requiring only 0.2%–2% of the cost of exhaustive profiling. Third, rerooting after each stage reduces latency-SLO violations relative to static workflow-level plans by up to 85%.

In summary, this paper makes four contributions:

- We show that coarse-grained control leaves performance on the table in agentic workflows.
- We formulate fine-grained workflow control as search over an annotated execution trie of workflow prefixes.
- We design and implement VineLM, which combines the trie formulation with checkpointing, efficient offline estimation, and dynamic rerooting to enable fine-grained per-invocation control.
- We demonstrate on NL2SQL and math reasoning workflows that VineLM improves the cost-latency-accuracy frontier and reduces latency-SLO violations relative to baselines with very low profiling cost.

2 Motivation

Our work here is motivated by the observation that existing workflow system controls are coarse-grained, in the sense that each LLM stage of a workflow binds to a fixed model. We show in this paper that there are clear advantages to a more fine-grained, flexible controller whose assignment of a model to a stage can vary across iterations of that stage. This is the case even when multi-iteration assignment is offline (a priori planned per stage iteration), but especially when assignment is online (i.e., selected on-the-fly during the execution of the workflow).

A representative design of a coarse-grained control system, exemplified by Murakkab [9], profiles a set of candidate workflow configurations offline—e.g., one model assignment per configurable stage together with a fixed retry horizon—and then, at serving time, assigns each request one of those pre-profiled configurations. This is an important step forward, but it is not the same as the fine-grained control we target. The chosen configuration fixes the model assignment of each stage template. In particular, if a workflow contains a refinement loop, the manager does not treat the each visit to that stage as a separate decision point; instead, it chooses one model up front and reuses that same model on every loop iteration within the request. Any adaptation therefore happens at the level of switching among whole-workflow configurations, not at the level of choosing the next model after each stage invocation. The system reasons about the workflow as one configurable unit, rather than as a sequence of stage invocations whose best next model can change as execution unfolds.

This distinction is the starting point for our motivation. Below, we show that this coarse-grained control leaves performance on the table in two ways: it cannot mix models for a given stage across loop iterations, and it cannot revisit downstream choices after earlier stages consume unexpected (more or less) cost or latency budget than anticipated.

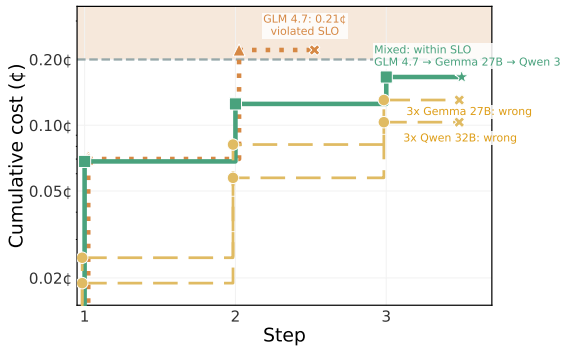

Figure 2. For a specific NL2SQL request, the objective is to return the correct answer under a fixed cost SLO. Static single-model plans fail in opposite ways: always using GLM 4.7 reaches the correct answer but exceeds the budget, while always using Gemma 27B or always using Qwen 32B stays within budget but returns the wrong answer. A mixed path—GLM 4.7 for generation, then Gemma 27B and Qwen 32B on successive repair rounds—stays within the SLO and returns the correct answer. This illustrates why refinement loops should be optimized as a sequence of model choices, not by binding one model to the entire loop.

2.1 Fixed loop choices

Coarse-grained control is especially restrictive in workflows with refinement loops. In an NL2SQL workflow, a manager such as Murakkab profiles configurations of the form "Generation = L_i, Repair = L_j, allow up to h repairs," then assigns one such configuration to the request. This lets the system choose one model for Generation and one model for Repair, but every visit to the Repair stage must reuse that same Repair model. The first and second Repair invocations are therefore not treated as separate decision points, even though they arise after different failures and under different remaining budgets.

Figure 2 shows why this matters on a concrete NL2SQL request with a fixed cost SLO. Static single-model strategies fail in opposite ways: always using `GLM 4.7` reaches the correct answer but exceeds the budget, while always using `Gemma 27B` or always using `Qwen 32B` stays within budget but returns the wrong answer. The successful path is mixed: `GLM 4.7` for the initial generation, then `Gemma 27B` and `Qwen 32B` on successive repair rounds. The figure uses the simplest static alternatives to make the effect visually clear, but the underlying point is broader: any coarse workflow-level plan that binds one model to the Repair stage cannot realize this successful path, because it requires different model choices on different loop iterations.

More generally, the right object to optimize is the sequence of model choices across loop iterations, not a single model bound to the loop as a whole. An early repair may be best served by a cheaper model that quickly resolves easy cases, while a later repair may justify escalating to a stronger model because the marginal value of avoiding another failure is now higher. Coarse-grained control cannot express such trajectories because it never treats individual loop iterations as separate decision points.

2.2 No per-invocation adaptation

Coarse-grained control fixes more than the model reused within a refinement loop: it fixes the remaining workflow plan before execution reveals how the current request is actually progressing. Offline profiling can identify the workflow-level configuration with the best expected cost–latency–accuracy tradeoff for a target objective, but that choice is based on averages across prior runs. For any particular request, the realized latency of an invocation depends not only on the chosen model, but also on the request itself, the amount of generated text, and transient backend conditions while the request is in flight [19, 20]. As a result, a configuration that looks optimal at request start can become suboptimal—or even infeasible—after one unexpectedly slow step. Existing workflow managers can react to load by periodically refreshing workflow-level configurations or by selecting among pre-profiled ones, but they do not reconsider the next model after each stage invocation [9].

Figure 3 shows a concrete example on a self-reflective MathQA workflow for a request with a 15 s latency SLO and the objective of maximizing accuracy. Using offline average profiles, the best workflow-level plan is `Gemma27B` → `Sonnet`

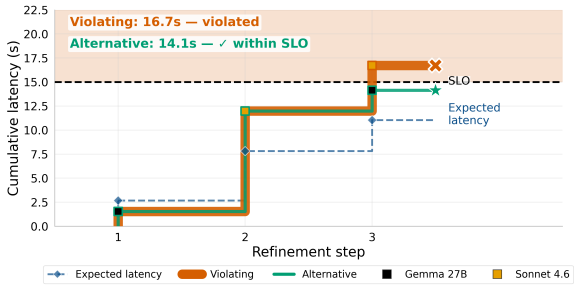

Figure 3. For this MathQA request with three stages, the objective is to maximize accuracy while meeting a 15 s latency SLO. Offline averages favor the plan `Gemma 27B` → `Sonnet 4.6` → `Sonnet 4.6`, but the realized second step runs long, so following that plan finishes in 16.7 s and violates the SLO (orange). Replanning after step 2 and switching the final step to `Gemma 27B` yields `Gemma 27B` → `Sonnet 4.6` → `Gemma 27B`, finishing in 14.1 s and staying within the SLO (green).

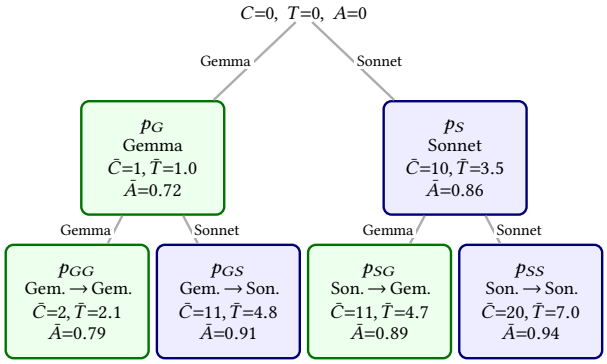

Figure 4. Illustrative execution trie with two configurable stages and two admissible models per stage. Each node p is annotated with expected cumulative cost $\bar{C}(p)$, expected cumulative latency $\bar{T}(p)$, and expected accuracy $\bar{A}(p)$ if execution terminated at that prefix. All three metrics are monotone along each root-to-leaf path.

`4.6` → `Sonnet 4.6`. At runtime, however, the second step—the first `Sonnet 4.6` invocation—takes longer than expected. If the system continues with the original plan, the request exceeds the 15 s budget. A controller with per-invocation adaptation can instead replan after observing that delay and switch the final step to `Gemma27B`, yielding `Gemma27B` → `Sonnet 4.6` → `Gemma27B`, which remains within the latency SLO.

The key point is that the best remaining suffix depends on the realized execution prefix, not just on offline averages. Thus, what is missing is fine-grained adaptation within a workflow run. If an early stage has already consumed too much latency budget, the system should be able to switch the next Repair invocation to a faster model or to a shorter remaining refinement strategy. If earlier stages finish quickly, it should be able to spend the saved slack on a stronger downstream model where additional accuracy is most valuable.

3 Trie-Based Workflow Formulation

3.1 Setting, scope, and request objectives

We consider a fixed agentic workflow template W that may contain branches and refinement loops. Some stages in W are tool stages, such as a SQL engine or retrieval component, while others are configurable LLM stages at which the system may choose among multiple models. Let

$$\mathcal{L} = \{L_1, \ldots, L_m\}$$

be the set of available LLMs. Each model L_i has its own quality, latency, and monetary cost characteristics, and each configurable stage s in the workflow admits a subset $\mathcal{L}(s) \subseteq \mathcal{L}$ of valid model choices. We are also given an offline dataset of representative requests

$$Q = \{q_1, \ldots, q_n\},$$

together with ground truth (i.e., the accurate output) for each request.

Our goal is to support request-specific objectives over accuracy, latency, and cost. Each incoming request is therefore modeled as a pair (q, o), where q is the request itself and o is an objective specification. We write

$$o = (f, C),$$

where f is the quantity to optimize and C is a set of constraints. Typical objectives include:

minimize cost	subject to accuracy $\geq a$,
maximize accuracy	subject to cost $\leq c$,
maximize accuracy	subject to latency $\leq l$.

This formulation and our approach is also easily extended to optimizations of one criteria subject to constraints on the other 2, e.g., maximize accuracy subject to both cost $\leq c$ and latency $\leq l$.

A key feature of our setting is that these targets are absolute and may vary from request to request; we do not assume a fixed global SLO or a small set of coarse quality tiers.

3.2 Workflow templates and execution tries

A workflow path is a sequence of model choices at the configurable stages encountered during one realization of the workflow:

$$p = (\ell_1, \ell_2, \ldots, \ell_k), \quad \ell_i \in \mathcal{L}(s(i))$$

where $s(i)$ is the ith stage of path p. Different paths may have different lengths k depending on how many configurable stages are traversed. The same model may appear at multiple positions due to loops. We assume each loop is bounded by a maximum retry horizon so that the set of feasible execution paths is finite.

The set of all feasible workflow paths induces an execution trie. The root corresponds to the empty prefix. Each node

corresponds to a prefix of model assignments,

$$u = (\ell_1, \ldots, \ell_r),$$

and there is an edge from u to

$$u' = (\ell_1, \ldots, \ell_r, \ell_{r+1})$$

whenever appending the next model choice is legal under the workflow's control flow and loop bounds. A root-to-leaf path represents one complete workflow instance, while an internal node represents a partial execution prefix shared by many complete paths. Unlike a full $|\mathcal{L}|$-ary tree, this trie respects workflow semantics: not every model is valid at every position, and some branches terminate earlier than others.

This trie view is useful because it makes the combinatorial structure of the decision space explicit while also exposing shared prefixes across many candidate executions. Two workflow instances that make the same early choices but diverge later share the same prefix nodes, and repeated loop iterations appear naturally as deeper prefixes of the same stage family. In the ideal case, each node in the trie would be annotated with the cumulative latency, expected cost, and expected accuracy associated with executing the corresponding prefix. The next subsection formalizes these path metrics and shows how monotonicity induces a structured feasible region for request-level optimization.

3.3 Path metrics

For the remainder of this section, we abuse notation slightly and use p to denote both a node in the execution trie and the workflow plan induced by the model assignments on the root-to-p path. Descendants of p correspond to strictly longer workflow plans that allow additional refinement rounds.

Let $p = (\ell_1, \ldots, \ell_k)$, and let $R_i(q, p)$ indicate that the ith stage on that prefix is actually reached when serving request q under the workflow's normal stop conditions. For each request $q \in \mathcal{Q}$ and workflow path p, we define three end-to-end metrics:

$$A(q, p), \qquad C(q, p), \qquad T(q, p),$$

corresponding to task accuracy, monetary cost, and latency.

In the simplest case, $A(q, p) \in \{0, 1\}$ is an offline success indicator computed against the request's ground truth. Monetary cost is treated in expectation:

$$C(q, p) = \sum_{i=1}^{k} R_i(q, p)\, c_i(q, \ell_i),$$

where $c_i(q, \ell_i)$ is the realized dollar cost of stage i if it is executed. Thus, if the workflow terminates early at an ancestor of p, all later stages on the prefix contribute zero cost. The node annotation

$$\bar{C}(p) = \mathbb{E}_{q \sim \mathcal{D}}[C(q, p)]$$

therefore already accounts for early termination and should be read as the expected spend of choosing prefix p.

Latency is handled more conservatively. Because cost budgets are naturally expectation-based but latency SLOs are per-request constraints that should not be violated by continuing too deep into the workflow, we do not discount later-stage latency by the probability of early stopping when annotating a node. Instead, if $\tau_i(q, \ell_i)$ denotes the latency of stage i when it is executed, we define the node's latency annotation as

$$\bar{T}(p) = \sum_{i=1}^{k} \mathbb{E}_{q \sim \mathcal{D}}[\tau_i(q, \ell_i) \mid R_i(q, p) = 1],$$

that is, the sum of expected per-stage latencies along the prefix. Intuitively, a request that continues to a later stage must still have enough remaining wall-clock budget to execute that stage; discounting its latency by the probability of early termination would understate the time required on the requests that actually continue.

Finally,

$$\bar{A}(p) = \mathbb{E}_{q \sim \mathcal{D}}[A(q, p)].$$

Figure 4 illustrates a small execution trie annotated with $(\bar{C}, \bar{T}, \bar{A})$ at each node.

3.4 Oracle path selection

Our approach relies on the observation that when the offline set of query samples $\mathcal{Q}$ is sufficiently large, our mean estimates of $A(p), C(p)$, and $T(p)$ accurately describe the means of future query samples. Specifically, when the $q \in \mathcal{Q}$ form a distribution that is stationary the law of large numbers ensures that a sufficiently large set of subsequent alternate online queries $\mathcal{Q}'$ would, if similarly measured offline, yield identical mean esimates. With this assumption, if the execution trie were fully annotated with exact path metrics $\bar{A}, \bar{C}, \bar{T}$, request handling would reduce to a constrained search over workflow paths. Let $\mathcal{P}$ denote the set of terminating nodes in the trie. A request objective $o = (f, C)$ specifies an optimization target f and constraints C. Three representative objectives are:

Minimize cost subject to accuracy floor:

$$p^\star = \arg\min_{p \in \mathcal{P}} \bar{C}(p) \quad \text{s.t.} \quad \bar{A}(p) \geq a.$$

In Figure 4, for $a = 0.90$, only p_{GS} ($\bar{A} = 0.91$) and p_{SS} ($\bar{A} = 0.94$) satisfy the accuracy floor; the optimizer selects p_{GS} because $\bar{C}(p_{GS}) = 11 < 20 = \bar{C}(p_{SS})$.

Maximize accuracy subject to latency cap:

$$p^\star = \arg\max_{p \in \mathcal{P}} \bar{A}(p) \quad \text{s.t.} \quad \bar{T}(p) \leq t.$$

For $t = 5.0$ in the figure, all four terminating paths satisfy the cap, and p_{SS} ($\bar{A} = 0.94$) is selected.

Maximize accuracy subject to cost budget:

$$p^\star = \arg\max_{p \in \mathcal{P}} \bar{A}(p) \quad \text{s.t.} \quad \bar{C}(p) \leq c.$$

For $c = 11$, the feasible paths are p_{GG}, p_{GS}, and p_{SG}; the optimizer selects p_{GS} ($\bar{A} = 0.91$) as the most accurate within budget. These examples show that the optimal path is not

fixed: it depends on the request objective and the induced feasible region in the trie.

Remark (Monotonicity enables pruning): One observation to speed up the above search is that all three metrics are monotone along any root-to-leaf path ($\bar{A}(u) \le \bar{A}(v)$, $\bar{C}(u) \le \bar{C}(v)$, $\bar{T}(u) \le \bar{T}(v)$ whenever u prefixes v): cost and latency grow because descendants run strictly more stages; we also assume accuracy cannot decrease because additional stages only add refinement opportunities. Consequently, each feasibility constraint defines a contiguous boundary on any root-to-leaf chain, allowing subtrees that cannot contain feasible or optimal descendants to be pruned.

Remark (Online Adaptation): The path selected above is an offline recommendation based on expected metrics. Online, if intermediate stage outcomes deviate from expectation (e.g. a stage invocation finishes sooner than expected), the system can re-evaluate the remaining subtrie and switch to a different path. We explore this setting in §4.3.

3.5 Challenges in estimating the annotated trie

The oracle formulation is conceptually simple once the trie is fully annotated. In practice, obtaining those annotations is the central challenge.

Scale: If we index requests by rows and workflow paths by columns, the accuracy outcomes form a request–path table

$$A \in \{0,1\}^{|\mathcal{Q}|\times|\mathcal{P}|},$$

with analogous tables C and T for cost and latency. In our NL2SQL experiments, $|\mathcal{Q}| = 1{,}529$ and $|\mathcal{P}| = 584$ ($8+64+512$ paths at depths 1–3), giving a $1{,}529 \times 584$ table with over 890,000 entries. Fully observing A would require running every request on every path — infeasible even for moderate workflow sizes. The central problem is therefore to estimate A under a limited profiling budget.

This is naturally viewed as a matrix completion problem [8, 23, 26]: we observe a sparse subset of entries and wish to estimate what is missing. Our setting, however, differs from standard matrix completion in two important ways.

Column mean estimation, not entry recovery: Standard matrix completion aims to recover every entry of a low-rank matrix. Here we only need the column means $\bar{A}(p) = \frac{1}{|\mathcal{Q}|}\sum_q A(q,p)$ for path selection. This relaxed goal allows estimators that exploit the known cascade structure directly without attempting to recover the full matrix.

Missing Not At Random (MNAR) observation pattern: Our approach to sampling paths is for each query, to repeatedly pick a random node on the trie. We make 2 observations: a depth d node can be executed when either it is explicitly chosen or when one of its descendants is chosen. Also, the number of depth d nodes increases monotonically in d, reducing the likelihood of sampling a specific depth d node than depth $d' < d$ counterparts. Figure 5 shows the result: depth-1 columns are 97% observed, depth-2 only 17%, and depth-3 just 1.8%. This is not only a sparsity issue: the accuracy monotonicity property (§3.3) also imposes additional constraints on A. Typical matrix completion does not exploit these structures.

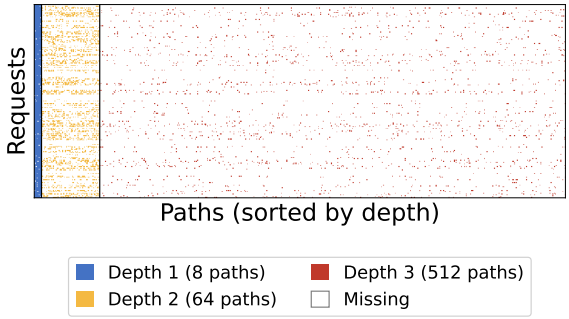

Figure 5. The request–path accuracy table A under 5% cascade sampling. Columns are sorted left to right by depth: 8 depth-1 paths, 64 depth-2 paths, and 512 depth-3 paths (separated by vertical lines). Missingness is severe and depth-dependent: depth-1 columns are 97% observed, depth-2 only 17%, and depth-3 just 1.8%, because later stages are only reached when earlier stages are executed. Each color indicates a depth-level observed entry; white indicates missing.

Estimation for optimization: The goal of offline profiling is not merely to minimize estimation error uniformly. Rather, the estimates serve downstream path selection: small errors near a feasibility boundary can flip the selected path, while larger errors far from the boundary are inconsequential.

Taken together, the execution trie, MNAR estimation under a limited profiling budget, and downstream path optimization form a rich problem space. The next section describes how VineLM addresses these challenges.

4 VineLM Design

4.1 Design Overview

Figure 6 shows the architecture of VineLM which has two main components: an offline trie construction pipeline that builds an annotated execution trie, and a runtime controller that uses that trie to make invocation-level decisions for each request. Offline, the developer provides a workflow template W together with a representative dataset $\mathcal{Q}$ for which ground truth is available. The profiler then evaluates a selected set of request–path pairs rather than exhaustively running every request on every workflow instance. These sampled outcomes are passed to the trie estimator, which constructs an annotated trie whose nodes correspond to execution prefixes and whose annotations estimate the expected metrics $\bar{A}(p), \bar{C}(p), \bar{T}(p)$ for each prefix or terminating path p. Intuitively, this trie is the compact summary that lets VineLM reason about a large space of workflow executions without materializing every possible run in full.

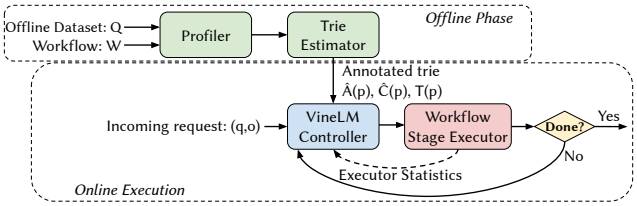

Figure 6. VineLM overview.

At runtime, each incoming request arrives together with an objective $o = (f, C)$, such as minimizing cost subject to an accuracy floor and latency cap. The VineLM controller begins at the root of the trie and repeatedly chooses the next stage/model decision based on the current execution prefix, the remaining budget implied by o, and the estimated metrics stored in the trie. The chosen action is then executed by the workflow stage executor, which may invoke either an LLM stage or a tool stage. If the workflow terminates, the result is returned to the user; otherwise, execution advances to a new trie prefix and the controller makes another decision. In this way, VineLM does not commit to a single workflow-level plan at admission time. Instead, it interleaves execution and control, revising its choices as the request progresses through the workflow.

4.2 Offline Trie Annotation

Building the annotated trie requires estimating, for each workflow path p, the expected path-level metrics $\hat{\bar{A}}(p)$, $\hat{\bar{C}}(p)$, and $\hat{\bar{T}}(p)$. In principle, all three must be learned from limited profiling data. In practice, the hard part is accuracy. Path-level cost and latency are largely determined by the chosen model, stage, and infrastructure, so their averages are comparatively stable and easier to estimate; moreover, VineLM can observe realized cost and latency online and update the remaining budget after each stage. We therefore focus on offline accuracy estimation in this subsection.

Checkpointing: Many workflow paths share a long prefix. If

$$p_1 = (\ell_1, \ldots, \ell_k), \qquad p_2 = (\ell_1, \ldots, \ell_b, \ell'_{b+1}, \ldots, \ell'_{k'}),$$

then the state after the common prefix $(\ell_1, \ldots, \ell_b)$ can be checkpointed once and reused to evaluate both suffixes. This reduces profiling cost and also removes spurious variance from rerunning the shared prefix independently for each descendant. Although reuse lowers the number of independent samples of the prefix itself, this can be offset by replaying a query along multiple balanced paths if needed. Storage space to hold checkpointed an be constrained by ordering execution of the randomly selected queries that operate over similar parts of the trie (i.e., use the same checkpoints) successively.

Cascade sampling: To populate the annotated trie under a limited profiling budget, VineLM profiles the workflow in cascade order. For each sampled query q, it invokes a randomly selected depth-1 model ℓ_1; if ℓ_1 fails, it continues to a depth-2 extension (ℓ_1, ℓ_2); and so on until the query succeeds or the path is exhausted. Cascade execution also yields free additional labels through subtree fill-in. Under our path semantics, if a query succeeds at some prefix, then every extension of that prefix is also counted as successful, since success anywhere on the path makes the whole path successful. Thus, when a node succeeds, the entire subtree rooted at that node can be marked successful at no extra cost. A budget of N cascade runs therefore yields substantially more than N observed request–path entries, with shallow columns effectively close to fully observed.

Why standard matrix completion remains biased: Despite subtree fill-in, the resulting observation pattern is still MNAR. Consider a depth-2 path (ℓ_1, ℓ_2). Cascade execution reaches this column only on queries for which ℓ_1 failed, so the raw sample mean estimates $\Pr[\ell_2 \text{ succeeds} \mid \ell_1 \text{ fails}]$, not the target

$$\bar{A}(\ell_1, \ell_2) = \Pr[\ell_1 \text{ or } \ell_2 \text{ succeeds}].$$

More generally, deeper columns are observed on increasingly hard subpopulations. A method that fits directly to these observed means will therefore systematically underestimate the true path-level accuracy of deeper paths.

Cascade decomposition: The same cascade structure that creates the MNAR bias also provides an exact correction. For a depth-1 path, the observed mean is already an unbiased estimate:

$$\hat{\mu}(\ell_1) = \bar{A}_{\ell_1}.$$

For a depth-2 path (ℓ_1, ℓ_2), success occurs if ℓ_1 succeeds, or if ℓ_1 fails and ℓ_2 then succeeds. These events are disjoint, so

$$\mu(\ell_1, \ell_2) = \mu(\ell_1) + \big(1 - \mu(\ell_1)\big) \cdot \Pr[\ell_2 \text{ succeeds} \mid \ell_1 \text{ fails}]. \quad (1)$$

The conditional term is exactly what cascade sampling observes for column (ℓ_1, ℓ_2). Hence, if $\bar{A}_{(\ell_1,\ell_2)}$ denotes the raw sample mean of that column, then

$$\hat{\mu}(\ell_1, \ell_2) = \hat{\mu}(\ell_1) + \big(1 - \hat{\mu}(\ell_1)\big) \cdot \bar{A}_{(\ell_1,\ell_2)}.$$

The same recursion applies at depth 3 and beyond: each observed column mean estimates the conditional success probability at that depth given that all earlier stages failed. The estimator is therefore consistent for every path p, and the MNAR pattern that breaks direct averaging becomes, under this decomposition, exactly the right conditioning.

Low-rank smoothing: At a 5% profiling budget, depth-3 columns receive only ≈ 20–80 observations each, making the cascade decomposition estimates noisy for deep paths. As a practical variance-reduction step, VineLM applies a rank-1 SVD projection to the depth-3 conditional accuracy estimates before substituting them into the decomposition.

End-to-end effectiveness: These techniques are complementary: checkpointing reduces redundant execution, cascade sampling and subtree fill-in maximize useful observations, and cascade decomposition corrects the MNAR bias

that defeats direct averaging and vanilla matrix completion. As we show in §5.3, together they reduce path-level MAE to roughly 1% using only 2% of the cost of exhaustive profiling, whereas standard matrix-completion baselines remain at 5% error. This makes sparse offline trie annotation accurate enough to support online model selection.

4.3 Online Model Selection

At runtime, VineLM decides per configurable stage. After each stage finishes, the controller observes the realized execution prefix u together with the cumulative latency so far T_u. It then re-roots the annotated trie at u, searches only the descendants of u, executes the next stage/model action on the best feasible suffix, updates (u, T_u), and repeats. Thus, VineLM never commits to a fixed workflow-level plan at admission time; it interleaves execution and control, allowing later choices to depend on the actual progress of the request.

This receding-horizon design is useful whenever realized behavior differs from offline averages [35]. If an early stage is slower than expected, the remaining budget shrinks and the controller can switch to a faster suffix. If earlier stages finish quickly, VineLM can spend the saved budget on a stronger downstream model or an additional refinement step. The same mechanism also handles loops naturally: each loop iteration moves the request to a deeper prefix in the trie, and VineLM solves the same optimization problem again on the remaining subtree.

Because expected accuracy, cost, and latency are monotone along every root-to-leaf path, search can prune large parts of the trie. For objectives that minimize cost subject to an accuracy floor, we use pruned DFS. Once a node already satisfies the accuracy target, exploring its descendants cannot improve that branch, since all descendants have weakly higher cost and latency. After the first feasible node is found, its objective value becomes an incumbent bound, so any prefix whose current cost or latency already exceeds that bound can be discarded. For objectives that maximize accuracy under cost or latency caps, pruning is weaker: internal-node accuracy does not justify pruning, because descendants may still improve accuracy while remaining feasible. In that case, pruning is driven only by prefixes that already violate the relevant budgets.

Runtime budget updates: These pruning rules are reapplied after every stage invocation using the realized execution so far. If the current prefix is u, the controller optimizes over descendants $v \succeq u$ using the remaining budgets, equivalently the incremental estimates

$$\Delta\hat{T}_u(v) = \hat{\bar{T}}(v) - \hat{\bar{T}}(u).$$

The trie's accuracy and cost estimates do not change during execution; what changes is which suffixes remain feasible after accounting for realized latency. Dynamic adaptation is therefore not a different optimization problem, but the same trie search repeated as new execution feedback arrives.

Load-aware latency adjustment: If VineLM has a current estimate of queueing delay or service slowdown for each serving engine e, it can incorporate that signal directly into search. For any suffix $v \succeq u$, VineLM replaces the offline latency estimate with

$$\Delta\hat{T}_u^{\text{live}}(v) = \Delta\hat{T}_u(v) + \sum_{e \in \text{engines}(v \setminus u)} \delta_e(t),$$

where $\delta_e(t)$ is the current expected delay for engine e. This inflates paths that rely on congested backends and steers the controller toward suffixes less likely to violate the request's latency target.

4.4 Implementation

VineLM is implemented as a control layer on top of an existing workflow runtime, LangGraph [27]. The runtime maintains a typed workflow state that stores the request context, intermediate artifacts, retry counters, the realized execution prefix, and per-stage metadata. VineLM wraps each configurable LLM stage with a routing layer that selects the model or endpoint for the next invocation, executes it, and records the resulting measurements. Fixed tool stages (e.g., SQL execution) are executed by the underlying workflow and return structured feedback that determines the next control-flow transition.

The prototype supports multiple serving backends, including Amazon Bedrock [1], Gemini, OpenAI-based backends, and SGLang [63]. Each invocation logs the selected model or endpoint together with token usage and latency statistics. For self-hosted backends such as SGLang, VineLM records time to first token, decode time, and token counts; for API-hosted backends such as Bedrock, it records provider-reported latency together with prompt and output tokens. These traces are used both to reconstruct offline cost and latency annotations and to update remaining budgets online during execution.

The execution trie is materialized through checkpointed workflow prefixes rather than by replaying every path from the root. The Profiler expands the search space depth by depth and runs each request–prefix pair in an isolated subprocess, which localizes failures and produces one structured execution record per node. Deeper workers resume from serialized parent checkpoints, execute only the remaining suffix, and emit updated checkpoints containing prefix metadata, evaluation results, and termination status. The Trie Estimator merges checkpoints by request and prefix, reconstructs path-level cost and latency, aggregates node statistics, applies subtree fill-in and the sparse estimator from §4.2, and produces the annotated trie used by the online VineLM Controller.

4.5 Discussion

Context reuse: Switching models across refinement iterations can reduce context reuse: a later step served by

a different model or endpoint may require re-sending the accumulated context and paying the prefill cost again [12, 30–32, 40, 53, 59, 63]. In our workloads this overhead was modest, and VineLM naturally accounts for it because profiling is performed over full workflow paths, so any extra latency or cost from re-prefill is already reflected in the trie annotations.

Non-LLM stages: Tool stages such as retrieval, SQL execution, or external API calls are treated as fixed transitions in the workflow template. They do not introduce branching in the trie, but their measured latency and cost are folded into cumulative path metrics, allowing VineLM to reason jointly about LLM choices and tool overheads.

Distribution mismatch: The trie also serves as a monitoring abstraction: VineLM can compare live path statistics against offline annotations and detect when observed latency or success rates drift away from the profiling distribution [28, 34]. When that happens, the right response is to refresh or recalibrate the trie using newer requests.

Resource allocation and hardware choices: Resource provisioning and hardware allocation are complementary to VineLM's per-invocation control. A higher-level planner can choose replicas, hardware, or endpoints [14, 45], while VineLM selects among those options online; the trie can be extended so that each action is a model–endpoint pair rather than just a model. We leave this joint, multi-timescale optimization to future work.

5 Evaluation

We evaluate VineLM along the following questions: (i) How much does finer-grain control improve over Murakkab-style workflow-level control in refinement loops? (ii) How closely can sparse profiling recover the decisions of a fully profiled trie and at what cost? (iii) How well does dynamic per-step control handle latency SLOs under runtime variance and load?

5.1 Experimental Setup

Workflows: We evaluate VineLM on two refinement-heavy workloads. The first workload is an NL2SQL pipeline based on a long-context NL2SQL system [13]. We use two variants of this workflow. NL2SQL-8 exposes eight candidate models and reaches total depth 3, consisting of one generation step and up to two refinement steps. NL2SQL-2 exposes two candidate models and reaches total depth 4, consisting of one generation step and up to three refinement steps. The second workload is a self-reflection math question-answering workflow [43]; we refer to this workload as MathQA. MathQA is a single-stage reflection workflow with up to six invocations of the same logical stage and four candidate models. These depths define the maximum horizon we profile, but the loop budget used for each request is chosen based on the target SLOs (accuracy, cost or latency) and the offline estimation of the managers rather than fixed arbitrarily or by workflow-specific stop semantics.

Models: For NL2SQL-8, we use Gemma-3-27B [22], Sonnet-4.6 [4], Kimi-K2.5 [51], Qwen3-32B [58], GLM-4.7 [64], Llama-3.3-70B [36], DeepSeek-V3.2 [17], and gpt-oss-120b [38]. For NL2SQL-2, we use Gemma-3-27B and Sonnet-4.6. MathQA uses Gemma-3-27B, Sonnet-4.6, Kimi-K2.5, and Qwen3-32B.

Baselines: Our main baseline is the workflow-level control space of Murakkab [9]. Murakkab can choose different models for different stage templates and can also choose the maximum number of refinement iterations, but it cannot choose a different model for different invocations of the same stage within one request. In NL2SQL, Murakkab can choose one model for generation, one model for refinement, and a refinement-depth cap. In MathQA, because the workflow has only one repeated stage, Murakkab can choose only one model for the entire workflow. Unless stated otherwise, Murakkab and VineLM (full) use full offline profiling. VineLM (sparse) uses only 2% of the full offline LLM profiling cost with the trie-estimation methods of §4.2.

Execution environment: We serve all workflow models through Amazon Bedrock [1], and we use Amazon EC2 [2] for the client machines. The client machine for NL2SQL is an `m6a.24xlarge` instance and the client machine for MathQA is an `m6a.8xlarge` instance.

5.2 Gain over Murakkab's Workflow-Level Control

We compare VineLM to the best Murakkab-style workflow-level configuration under the same cost SLO. Figure 7 reports the accuracy gain of VineLM over Murakkab on NL2SQL-8, NL2SQL-2, and MathQA. The solid line uses the fully profiled trie; the dashed line uses sparse profiling at 0.2-2% of the full profiling cost. Across all three workloads, VineLM consistently outperforms the best static workflow-level choice, and sparse VineLM preserves most of that gain despite using only a tiny fraction of the profiling cost.

VineLM optimizes over a richer action space. For the case of NL2SQL-8, this is 136 Murakkab configurations versus 584 trie paths; for NL2SQL-2, it is 14 versus 30. Even in NL2SQL-8, where Murakkab already has a relatively rich workflow-level space, VineLM still delivers a consistent positive accuracy delta across the cost range. This is the hardest setting for VineLM to separate from Murakkab, because Murakkab already has many static configurations to choose from. The gain is larger in NL2SQL-2. With fewer models, Murakkab has even less ability to approximate the right repair sequence, while VineLM can still adapt each repair invocation separately. As a result, the advantage of modeling the loop as a sequence of prefix-conditioned decisions becomes more pronounced. Sparse VineLM retains most of that advantage.

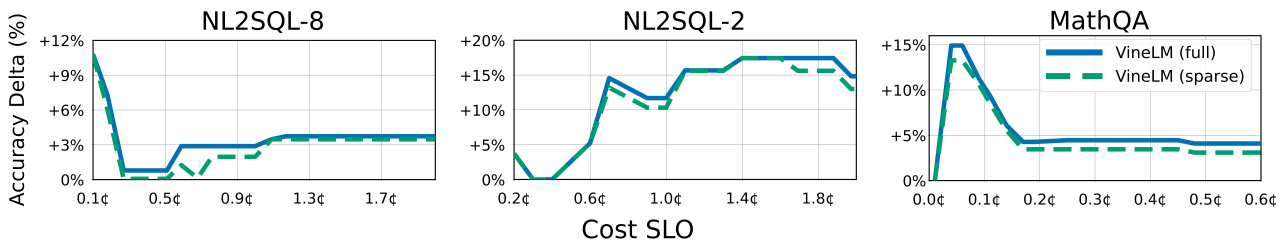

Figure 7. Accuracy delta over Murakkab for three workflows: NL2SQL-8, NL2SQL-2, and MathQA.

MathQA shows a smaller but still positive delta for a different reason. Because MathQA is a repeated-reflection workflow, Murakkab effectively commits to one model for the reflection process, whereas VineLM can mix models across rounds. The gain is smaller not because fine-grained control stops helping, but because baseline accuracy is already higher, leaving less headroom for any controller to improve end-to-end success.

The gain is not just that VineLM has more choices. Murakkab optimizes over workflow-level profiles: one generation model, one repair model, and a loop horizon or round count, fixed before the request begins. That view collapses the loop into a single average plan. In particular, when estimating the value of later repair rounds, it cannot condition on which requests actually survive to those rounds. Requests that would already have finished after Generation or after the first Repair are still mixed into the same workflow-level expectation. VineLM does not have this problem. Because the execution trie is indexed by prefixes and the cascade estimator recovers prefix-conditioned success probabilities, VineLM prices a continuation only on the subset of requests that actually reaches that prefix. This lets VineLM value later repair rounds more accurately and decide whether another step is still worth the remaining budget.

5.3 A Deep Dive into Trie-Filling Methods

We next ask how closely VineLM can recover the fully profiled frontier after observing only a small fraction of request–path executions. Coverage denotes the fraction of the full offline LLM profiling cost spent on sparse sampling. The goal in this subsection is not to reconstruct every missing entry of A, but to predict the column means that determine path selection.

We compare six estimators. First, is direct average: for each path, predict its column mean by averaging the observed entries in that column. Under cascade sampling, this estimator is badly biased for deeper paths because those paths are only observed on harder requests. Second, prefix fill-in + avg: first apply prefix-success closure, then average the observed entries in each column. Prefix fill-in marks all descendants of an observed success as successful, because once a prefix succeeds, every extension of that prefix also succeeds. Third, prefix fill-in + hard impute: after prefix fill-in, apply a low-rank matrix-completion step to impute the remaining unobserved cells, then average the completed columns. Fourth, prefix fill-in + XGBoost: after prefix fill-in, apply XGBoost [11] which predicts each column mean from hand-designed path and observation features, including path depth, row and column observation counts, row and column mean success rates, prefix values and prefix means at several depths, sibling statistics, and model power-score summaries along the path. Fifth, VineLM-Lite: estimate path means through the cascade decomposition of §4.2, which corrects the MNAR bias by treating deeper observations as conditional accuracies. Sixth, VineLM: adds the low-rank smoothing step on top of conditional decomposition.

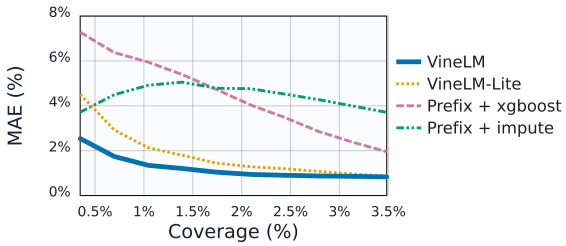

Figure 8. Column-mean prediction error versus profiling coverage. Coverage is the fraction of full offline LLM profiling cost used for sparse sampling.

Table 1. Column-level error summary at 2% cost coverage for NL2SQL. Signed error is prediction minus ground-truth column mean.

Method	Mean Signed	Mean Abs.	Max Abs.
average	-49.53%	49.53%	76.46%
prefix + avg	20.50%	20.50%	33.17%
prefix + impute	4.78%	4.78%	12.98%
prefix + xgboost	3.08%	4.73%	15.57%
VineLM-Lite	**0.04%**	1.45%	6.29%
VineLM	0.07%	**1.04%**	**4.33%**

Figure 8 plots column-mean MAE as a function of coverage. VineLM attains the lowest error across the full range and reaches about 1% MAE at 2% coverage. The ordering matches the MNAR analysis in §3.5: methods that ignore the depth-dependent sampling bias misestimate the deeper columns that determine the high-accuracy end of the frontier.

To analyze the methods more closely, we fix coverage at 2% and examine the full error distribution. Table 1 shows both the scale and the direction of the mispredictions. Signed

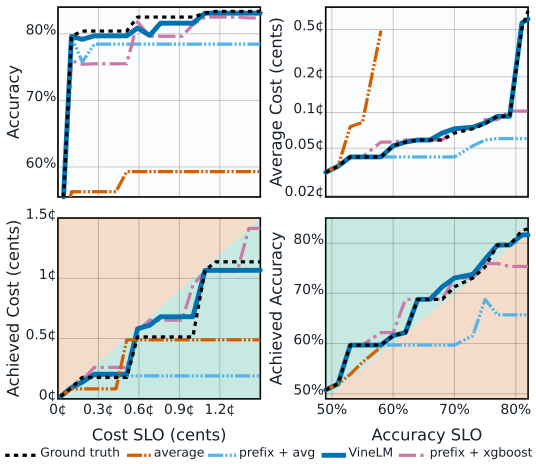

Figure 9. Maximum accuracy under a cost SLO (left) and minimum average cost under an accuracy SLO (right), at 2% profiling coverage.

error separates pessimism from optimism. The direct average is strongly pessimistic because direct averaging observes only the suffix-conditioned hard subpopulation. Prefix-based methods are optimistic on average. VineLM variants are nearly unbiased in mean signed error, which shows that conditional decomposition removes the main source of bias. The remaining difference between VineLM and VineLM-Lite is mostly in variance and tail behavior: VineLM reduces mean absolute error from 0.0145 to 0.0104 and max absolute error from 0.0629 to 0.0433. This difference matters because one large column misprediction can change the selected path even when average error remains small.

We then keep coverage fixed at 2% and rerun the policy search with predicted column means. Figure 9 compares the resulting maximum-accuracy-under-cost curves against the fully profiled ground truth. VineLM tracks the ground-truth line closely in both the achieved-accuracy panel and the achieved-cost panel, which shows that low column-mean error translates into high-fidelity policy selection.

The same figure shows the comparison for the minimum-cost-under-accuracy objective. VineLM again follows the ground-truth frontier closely. Some baselines appear cheaper in the top panel, especially prefix fill-in + avg, but the bottom panel shows why: those methods fall below the $y = x$ line and therefore violate the accuracy SLO. The same failure mode appears for XGBoost at higher accuracy targets. These methods do not adequately correct the sampling bias in §3.5, and they also underprofile the deeper paths that are usually necessary to reach higher accuracy. The direct-average baseline fails even earlier. Its curve terminates around 57% accuracy because it severely mispredicts the deep paths that dominate the high-accuracy regime.

Profiling Cost: Table 2 reports three profiling regimes: VineLM's sparse profiler (VineLM), full checkpointed exhaustive profiling (Chkpt), and naive exhaustive profiling

Table 2. Profiling cost in dollars.

Workflow	VineLM	Chkpt	Full	Ratio
MathQA-4	28.16	563.18	15073.67	535.29x
NL2SQL-2	5.28	105.56	250.49	47.44x
NL2SQL-8	48.19	963.89	2764.21	57.36x

from the root on every leaf path (Full). Even without sparsity, checkpointing substantially reduces full-profiling cost by reusing shared prefixes: compared with the naive Full baseline, checkpointing alone reduces cost by 2.37× on NL2SQL-2, 2.87× on NL2SQL-8, and 26.77× on MathQA-4. Combining checkpointing with sparse profiling yields much larger savings. Relative to Full, VineLM reduces profiling cost by 47.44× on NL2SQL-2, 57.36× on NL2SQL-8, and 535.29× on MathQA-4. These results show that shared-prefix reuse is already important for exhaustive profiling, and that VineLM's full sparse-trie pipeline makes offline trie annotation cheap enough to be practical. As expected, gains are larger in complex, deep workflows with many available models to choose from.

5.4 Latency SLO and Online Model Selection

So far, we have focused on cost and accuracy. Latency is different because stage-time variability can create per-request SLO violations even when the offline plan is optimal in expectation. We therefore compare three policies: (i) Murakkab, which commits to a single full path at admission time, (ii) a dynamic load-unaware policy that recomputes the best remaining suffix after each stage using realized elapsed time, and (iii) a dynamic load-aware policy that also inflates latency estimates using the current load model. All three policies optimize for accuracy subject to a latency SLO.

The load model comes from a separate queueing experiment on SGLang setup. All requests were real NL2SQL problems. We used one fixed problem as the target request and injected artificial load by sending N higher-priority dummy requests in parallel, with $N \in \{0, 1, 2, 4, 8, 16, 32\}$. Dummy requests were drawn randomly from the full NL2SQL set. We delayed the target request by 2000 ms so that most dummy requests were already queued, repeated each load level 50 times, and flushed the KV cache between runs. We then fit the utilization-conditioned slowdown curve used to inflate recorded latencies during evaluation.

Figure 10 shows that per-stage replanning reduces latency-SLO violations relative to Murakkab across the tested SLO range. Under additional load, the load-aware policy reduces violations further by steering requests away from more overloaded, slower paths. Violation counts do not necessarily decrease as the latency SLO increases. A looser latency budget lets the controller choose slower and more accurate paths, so the operating point remains close to the latency frontier. Residual latency misprediction can therefore still produce violations even at larger SLOs.

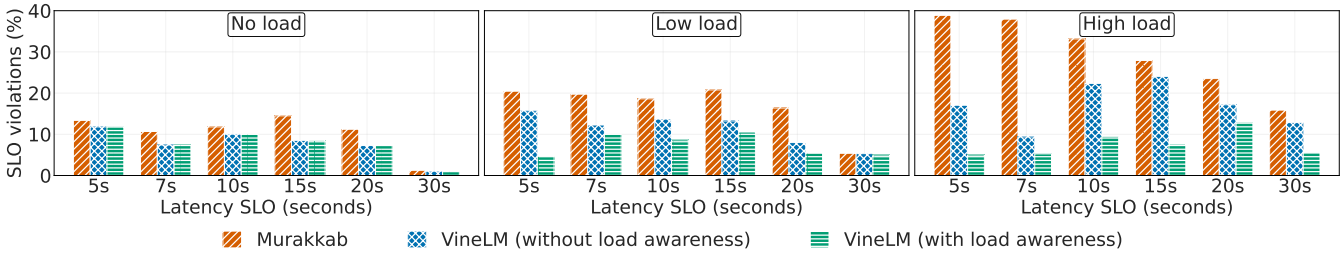

Figure 10. Latency-SLO violation rate for Murakkab, dynamic load-unaware control, and dynamic load-aware control. Dynamic replanning reduces violations, and load-aware replanning reduces them further under load.

Table 3. Overhead of the dynamic method. Overhead (%) shows the overhead of the dynamic method compared to the average latency for the fastest LLM call in the workflow.

Workflow	Mean (μs)	Overhead (%)
MathQA-4	6921.2	0.592
NL2SQL-2	33.0	0.001
NL2SQL-8	1312.0	0.052

Online selection adds a small search cost after each stage because it re-roots the trie and recomputes the best remaining suffix. Table 3 shows that this cost is negligible relative to LLM calls. Even the largest configuration we measure—the depth-6 MathQA workflow with four models—adds 6.921 ms per replanning step, which is only 0.592% of the fastest LLM call in that workflow. The smaller configurations are much cheaper: 33 μs for NL2SQL with two models and 1.312 ms for NL2SQL with eight models. As expected, the overhead increases with both workflow depth and the number of candidate models, but it remains below 1% in all measured cases.

6 Related Work

Agentic workflow serving systems. The system closest to VineLM is Murakkab [9]. Murakkab decouples logical workflow specification from execution choices and uses offline profiles, an SLO-aware optimizer, and an adaptive runtime to choose among pre-profiled workflow-level configurations. Our main distinction is decision granularity. Murakkab commits each request to a workflow-level plan that binds one model to each configurable LLM stage and fixes loop-related knobs up front. By contrast, VineLM controls a workflow at the level of stage invocations.

LLM routing and cascades. A large recent literature studies how to route a query to an appropriate LLM to improve the cost-quality tradeoff. RouteLLM [37], Hybrid LLM [21], IRT-Router [50], and Causal LLM Routing [52] learn prompt-level routing policies using preference data, predicted difficulty, or observational logs, while FrugalGPT [10] and later work on routing–cascading unification [18] study sequential escalation from cheaper to stronger models. These methods solve a flatter problem than ours: choose one model, or a short cascade, for a single query. VineLM instead controls multi-stage workflows with tools and loops, where an early decision changes not only immediate cost and latency but also the remaining budget and whether later stages are entered at all. However, similar signals such as question complexity could be used to bias VineLM's initial search toward cheaper or stronger prefixes, or to maintain separate trie annotations for different difficulty buckets.

Classical ML serving and pipeline management. Classical ML-serving systems address neighboring problems, but under different abstractions. Clipper [15] introduced online model selection and ensembling over a flat set of models. INFaaS [44] supports per-request performance and accuracy targets, but still operates over single-model variants and model/hardware profiles. InferLine [14] and Llama [45] move closer to workflow-level optimization by planning and adapting multi-stage pipelines under latency objectives. These systems demonstrate the value of workflow-aware planning and online adaptation, but they target classical pipelines where stage choices primarily affect latency, throughput, and resource efficiency. VineLM instead targets agentic LLM workflows with loops, where invocation-level model choices also affect semantic accuracy and future control flow.

7 Conclusion

VineLM uses an annotated execution trie to make fine-grained control practical for agentic workflows. Hence, its controller can avoid the common pitfall of coarse workflow-level policies that bind one model to an entire refinement loop and cannot revise downstream choices as execution unfolds. By combining checkpointing, cascade profiling, and rerooted online control, VineLM improves the cost–latency–accuracy frontier for refinement-heavy workloads while keeping profiling overhead low. Across NL2SQL and math reasoning workflows, VineLM achieves up to 18% higher accuracy at the same budget, reduces offline profiling cost by 98%, and cuts latency-SLO violations by up to 85% relative to coarse workflow-level baselines. These results suggest that agentic workflows should be controlled with online policies over execution prefixes, rather than with fixed workflow configurations.

References

[1] Amazon Web Services. Amazon Bedrock, 2026. Accessed: 2026-04-02.

[2] Amazon Web Services. Amazon Elastic Compute Cloud (Amazon EC2), 2026. Accessed: 2026-04-02.

[3] Amazon Web Services, Inc. Strands agents — open source ai agent sdk for python & typescript. https://strandsagents.com/, 2026. Accessed: 2026-04-02.

[4] Anthropic. Claude Sonnet, 2024. Accessed: 2026-04-02.

[5] Anthropic. Agent sdk overview - claude api docs. https://platform.claude.com/docs/en/agent-sdk/overview, 2026. Accessed: 2026-04-02.

[6] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of thoughts: Solving elaborate problems with large language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(16):17682–17690, March 2024.

[7] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

[8] Emmanuel J. Candès and Benjamin Recht. Exact matrix completion via convex optimization. *Foundations of Computational Mathematics*, 9(6):717–772, 2009.

[9] Gohar Irfan Chaudhry, Esha Choukse, Haoran Qiu, Íñigo Goiri, Rodrigo Fonseca, Adam Belay, and Ricardo Bianchini. Murakkab: Resource-efficient agentic workflow orchestration in cloud platforms, 2025.

[10] Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *Transactions on Machine Learning Research*, 2024. Featured Certification.

[11] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[12] Yihua Cheng, Kuntai Du, Jiayi Yao, and Junchen Jiang. Do large language models need a content delivery network?, 2024.

[13] Yeounoh Chung, Gaurav T. Kakkar, Yu Gan, Brenton Milne, and Fatma Ozcan. Is long context all you need? leveraging LLM's extended context for NL2SQL. *arXiv preprint arXiv:2501.12372*, 2025.

[14] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 477–491, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association.

[16] CrewAI. Crewai: The leading multi-agent platform. https://crewai.com/, 2026. Accessed: 2026-04-02.

[17] DeepSeek-AI. Deepseek-v3.2: Pushing the frontier of open large language models, 2025.

[18] Jasper Dekoninck, Maximilian Baader, and Martin T. Vechev. A unified approach to routing and cascading for llms. In *ICML*, 2025.

[19] Christina Delimitrou and Christos Kozyrakis. Amdahl's law for tail latency. *Commun. ACM*, 61(8):65–72, July 2018.

[20] Christina Delimitrou and Christos Kozyrakis. Amdahl's law for tail latency. *Commun. ACM*, 61(8):65–72, July 2018.

[21] Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Ruhle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. Hybrid llm: Cost-efficient and quality-aware query routing, 2024.

[22] Google. gemma-3-27b-it Model Card, 2025. Accessed: 2026-04-02.

[23] Trevor Hastie, Rahul Mazumder, Jason Lee, and Reza Zadeh. Matrix completion and low-rank svd via fast alternating least squares, 2014.

[24] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024.

[25] 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.

[26] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[27] LangChain, Inc. Langgraph, 2024.

[28] Grace A. Lewis, Sebastián Echeverría, Lena Pons, and Jeffrey Chrabaszcz. Augur: a step towards realistic drift detection in production ml systems. In *Proceedings of the 1st Workshop on Software Engineering for Responsible AI*, SE4RAI '22, page 37–44, New York, NY, USA, 2023. Association for Computing Machinery.

[29] Jerry Liu. LlamaIndex, 11 2022.

[30] Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, and Junchen Jiang. Lmcache: An efficient kv cache layer for enterprise-scale llm inference, 2025.

[31] Yuhan Liu, Hanchen Li, Yihua Cheng, Siddhant Ray, Yuyang Huang, Qizheng Zhang, Kuntai Du, Jiayi Yao, Shan Lu, Ganesh Ananthanarayanan, Michael Maire, Henry Hoffmann, Ari Holtzman, and Junchen Jiang. Cachegen: Kv cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 38–56, New York, NY, USA, 2024. Association for Computing Machinery.

[32] Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E. Gonzalez, and Ion Stoica. Autellix: An efficient serving engine for llm agents as general programs, 2025.

[33] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023.

[34] Ankur Mallick, Kevin Hsieh, Behnaz Arzani, and Gauri Joshi. Matchmaker: Data drift mitigation in machine learning for large-scale systems. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 77–94, 2022.

[35] D. Q. Mayne, J. B. Rawlings, C. V. Rao, and P. O. M. Scokaert. Survey constrained model predictive control: Stability and optimality. *Automatica*, 36(6):789–814, June 2000.

[36] Meta. Llama-3.3-70B-Instruct Model Card, 2024. Accessed: 2026-04-02.

[37] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M Waleed Kadous, and Ion Stoica. RouteLLM: Learning to route LLMs from preference data. In *The Thirteenth International Conference on Learning Representations*, 2025.

[38] OpenAI. gpt-oss-120b & gpt-oss-20b Model Card, 2025. Accessed: 2026-04-02.

[39] OpenAI. Openai agents sdk. https://openai.github.io/openai-agents-python/, 2026. Accessed: 2026-04-02.

[40] Zaifeng Pan, Ajjkumar Patel, Zhengding Hu, Yipeng Shen, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. Kvflow: Efficient prefix caching for accelerating llm-based multi-agent workflows. *arXiv preprint arXiv:2507.07400*, 2025.

[41] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis, 2023.

[42] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023.

[43] Matthew Renze and Erhan Guven. Self-reflection in LLM agents: Effects on problem-solving performance. *arXiv preprint arXiv:2405.06682*, 2024.

[44] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association, July 2021.

[45] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 1–17, New York, NY, USA, 2021. Association for Computing Machinery.

[46] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[47] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face, 2023.

[48] Ruijie Shi, Houbin Zhang, Yuecheng Han, Yuheng Wang, Jingru Fan, Runde Yang, Yufan Dang, Huatao Li, Dewen Liu, Yuan Cheng, and Chen Qian. Agentxray: White-boxing agentic systems via workflow reconstruction, 2026.

[49] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.

[50] Wei Song, Zhenya Huang, Cheng Cheng, Weibo Gao, Bihan Xu, Guan-Hao Zhao, Fei Wang, and Runze Wu. Irt-router: Effective and interpretable multi-llm routing via item response theory, 2025.

[51] Kimi Team. Kimi k2.5: Visual agentic intelligence, 2026.

[52] Asterios Tsiourvas, Wei Sun, and Georgia Perakis. Causal LLM routing: End-to-end regret minimization from observational data. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

[53] Noppanat Wadlom, Junyi Shen, and Yao Lu. Efficient llm serving for agentic workflows: A data systems perspective, 2026.

[54] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), March 2024.

[55] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models, 2023.

[56] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Eric Zhu, Li Jiang, Shaokun Zhang, Xiaoyun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen LLM applications via multi-agent conversation framework, 2023.

[57] Binfeng Xu, Zhiyuan Peng, Bowen Lei, Subhabrata Mukherjee, Yuchen Liu, and Dongkuan Xu. Rewoo: Decoupling reasoning from observations for efficient augmented language models, 2023.

[58] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025.

[59] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. Cacheblend: Fast large language model serving for rag with cached knowledge fusion. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 94–109, New York, NY, USA, 2025. Association for Computing Machinery.

[60] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023.

[61] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.

[62] Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2025.

[63] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc.

[64] Zhipu AI. GLM-4.7 Technical Blog, 2025. Accessed: 2026-04-02.

A Estimating Accuracy for Workflow Instances under MNAR Sampling

This appendix describes VineLM's approach to estimating the expected accuracy of every workflow instance from sparse offline profiling. We show that the cascade structure of the workflow tree, which at first appears to make the missing-data problem harder, in fact provides an exact bias correction that standard matrix completion cannot exploit.

A.1 Problem Statement

Let $\mathcal{Q} = \{q_1, \ldots, q_m\}$ be the set of profiling queries and let $\mathcal{P}$ denote the set of all workflow instances (*i.e.*, root-to-leaf paths in the workflow tree). The accuracy outcomes form a request–path table $A \in \{0,1\}^{|\mathcal{Q}|\times|\mathcal{P}|}$, where $A(q,p) = 1$ if workflow instance p succeeds on query q. The goal of offline profiling is to estimate the *expected accuracy* of each workflow instance:

$$\mu(p) = \frac{1}{|\mathcal{Q}|}\sum_{q\in\mathcal{Q}} A(q,p) = \mathbb{E}_q[A(q,p)], \quad p \in \mathcal{P}. \tag{2}$$

Given a cost or latency budget that limits the number of profiling runs, only a small fraction of entries in A can be observed directly. Estimating μ from this partial view is the core matrix-completion problem.

A.2 The MNAR Challenge

Cascade sampling is the natural way to profile the workflow tree: pick a query q uniformly at random, invoke a randomly sampled depth-1 model L_i, and continue to deeper stages only when the current execution fails. Under a profiling budget of N runs, this produces N observed entries distributed across A, but the observation pattern is **Missing Not At Random (MNAR)**: a depth-2 workflow instance (L_i, L_j) is only observed on queries for which L_i already failed. Formally, for any query q,

$$\begin{aligned}\mathbb{E}\big[A(q,\,(L_i,L_j)) \mid (q,\,(L_i,L_j)) \in \Omega\big] &= q(L_j \mid L_i \text{ fails}) \\ &\neq \mu(L_i,L_j),\end{aligned} \tag{3}$$

where Ω is the set of observed entries and $q(L_j \mid L_i \text{ fails})$ is the conditional accuracy of L_j given that L_i already failed. The two quantities differ because $\mu(L_i,L_j)$ averages over *all* queries, while the observed entries come from the harder subpopulation on which L_i failed. Depth-3 instances are even more severely affected: they are observed only on the subset of queries where both depth-1 and depth-2 stages fail, *i.e.*, the hardest queries.

Standard matrix completion applies low-rank factorization to the observed entries, fitting the latent model to this biased subpopulation. This yields large errors for depth-2 and depth-3 instances even after subtree fill-in.

A.3 Unbiased Estimation via Cascade Decomposition

The key insight is that the cascade workflow structure, despite inducing MNAR, also reveals its own bias correction. Because A is prefix-closed — a workflow instance succeeds whenever any of its prefixes succeed — the expected accuracy satisfies the following recursive decomposition:

$$\mu(L_i) = \mathbb{E}_q[A(q,L_i)], \tag{4}$$

$$\mu(L_i,L_j) = \mu(L_i) + \big(1-\mu(L_i)\big)\cdot q(L_j \mid L_i \text{ fails}), \tag{5}$$

$$\mu(L_i,L_j,L_k) = \mu(L_i,L_j) + \big(1-\mu(L_i,L_j)\big)\cdot q(L_k \mid (L_i,L_j) \text{ fails}). \tag{6}$$

The conditional accuracy $q(L_j \mid L_i \text{ fails})$ in (5) is precisely the quantity that (3) shows to be *unbiasedly* estimated by the sample mean of observed entries in column (L_i, L_j). Substituting sample means into (4)–(6) therefore yields a consistent estimator:

$$\hat{\mu}(L_i) = \bar{A}_{L_i}, \tag{7}$$

$$\hat{\mu}(L_i,L_j) = \hat{\mu}(L_i) + \big(1-\hat{\mu}(L_i)\big)\cdot \bar{A}_{(L_i,L_j)}, \tag{8}$$

$$\hat{\mu}(L_i,L_j,L_k) = \hat{\mu}(L_i,L_j) + \big(1-\hat{\mu}(L_i,L_j)\big)\cdot \bar{A}_{(L_i,L_j,L_k)}, \tag{9}$$

where $\bar{A}_p$ denotes the sample mean of observed entries in column p. As $N \to \infty$, $\hat{\mu}(p) \to \mu(p)$ for every workflow instance p.

Intuition: The MNAR observation model is harmful for estimating $\mu(p)$ directly, but it is exactly the right conditioning for estimating the incremental conditional accuracy q. The cascade decomposition turns this into an advantage: rather than estimating μ from biased column means, we estimate q from unbiased conditional means and reconstruct μ via the recursive formula.

A.4 Low-Rank Regularisation for Sparse Depth-3 Estimates

The estimator in (7)–(9) is unbiased but high-variance for depth-3 instances: with a 5% profiling budget, depth-3 columns receive only ≈20–80 observations each.

To reduce this variance, we exploit the low-rank structure of the *conditional accuracy matrix* Q. Define the $72 \times |\mathcal{L}|$ matrix Q whose rows correspond to every workflow instance prefix (8 depth-1 prefixes and 64 depth-2 prefixes) and whose columns correspond to candidate last-stage models, with $Q[p, L_k] = q(L_k \mid p \text{ fails})$. The depth-3 block (rows corresponding to depth-2 prefixes) is approximately rank-1 in practice, reflecting a single latent query-difficulty axis that drives conditional accuracy across all prefixes and models.

We apply a rank-1 SVD projection to the $64 \times |\mathcal{L}|$ depth-3 block:

$$\hat{Q}_{\text{depth-3}} = \Pi_1\big(Q_{\text{depth-3}}\big), \qquad \hat{Q}_{\text{depth-2}} = Q_{\text{depth-2}}, \tag{10}$$

where Π_1 denotes projection onto the rank-1 manifold (initialized with column means for unobserved entries). SVD is applied only to the sparse depth-3 block; the depth-2 block is well-observed (all 64 entries, 150–500 samples each) and

uses raw conditional means directly to avoid introducing bias. The smoothed rates $\hat{Q}$ are then substituted into (8)–(9) in place of $\bar{A}_p$.