FLOWMAKER: FROM TASKS TO COST-EFFICIENT GENERATIVE AI WORKFLOW DEPLOYMENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Generative AI workflows execute user tasks by composing sequences of LLM calls, where each call performs a sub-task (e.g., summarization or translation), forming a workflow. Executing such workflows requires decisions over workflow structure, model assignments, and deployment configurations, which jointly determine latency, cost, and accuracy under dynamic workloads and resource constraints. Existing systems optimize these decisions in isolation or fix them early, leading to suboptimal cost and goodput. We make two key observations. First, efficient serving requires jointly exploring all combinations of workflows, model choices, and deployment configurations under dynamic cluster state. Second, reuse of deployed LLMs is critical for reducing cost, but must be performed on-the-fly, as reuse depends on current load and resource availability. Together, these make selecting a workflow-variant—a combination of workflow, models, and deployment configuration—a time-critical problem. We present FlowMaker, which exposes a task-level interface and selects workflow-variants on behalf of users. To realize this, we design a constraint-guided load-aware pruning algorithm for fast, on-the-fly decisions. FlowMaker improves goodput by up to 10.14× and reduces cost by up to 2.77× compared to state-of-the-art baselines, including Murakkab.

1 INTRODUCTION

Generative AI applications increasingly execute complex user tasks by composing multiple LLM calls into workflows (Zaharia et al., 2024; Lin et al., 2025; Chaudhry et al., 2026; Amazon Web Services, 2025; Laju et al., 2026; Zhang et al., 2025; Chase, 2022; LangChain, 2025; CrewAI, 2024; Wu et al., 2024; Google Cloud, 2025; Microsoft Azure, 2025). Consider answering questions over enterprise data when relevant documents must be summarized and the result translated. This task can be executed by (a) summarizing the documents and then translating the summary, or (b) translating the documents before summarization, or (c) performing summarization and translation jointly. After choosing a workflow, the system must assign a model to each LLM sub-task (e.g., Llama-3.1-8B or Qwen-2.5-7B) and select a deployment configuration for each model (e.g., tensor- and pipeline-parallelism settings). We call each combination of workflow, model assignments, and deployment configurations a *workflow-variant*. For the task in Figure 1, assigning 11 models and 10 deployment configurations to the LLM sub-tasks in three candidate workflows yields over $24K$ workflow-variants. Across these variants, latency varies by $189.3\times$ (0.6–119s), cost by $8725\times$ (2×10^{-5}–0.17 ¢), and accuracy by $3.3\times$ (14.4–48.0%). Thus, the selected workflow-variant directly affects the quality, performance, and cost of serving the task.

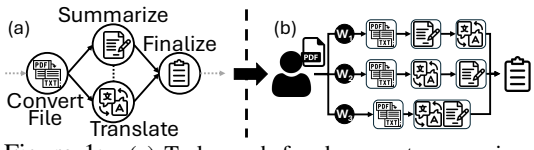

Figure 1: (a) Task graph for document conversion, summarization, and translation, showing the sub-tasks and their dependencies. (b) The three candidate workflows considered in our example. W_1 converts the user-provided PDF to text, summarizes it, and then translates the summary; W_2 and W_3 use alternative organizations of the sub-tasks. Assigning models and deployment configurations to these workflows yields over 24K variants, with significant variation in latency and cost.

Being multi-dimensional, these decisions are already complex to make. This complexity is further exacerbated by dynamic conditions: Task arrival rates vary over time (Sinha et al., 2026; Zhou et al., 2022; Bhasi et al., 2022), workload mixes evolve (Wang et al., 2025; Patel et al., 2024; Stojkovic et al., 2025; Qiu et al., 2026), and resource availability changes (Albalawi, 2025; Cortez et al.,

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.

2017). However, existing workflow frameworks typically fix these decisions statically (Chase, 2022; LangChain, 2025; CrewAI, 2024; Wu et al., 2024; Hong et al., 2024), while managed platforms leave them to manual tuning (Amazon Web Services, 2025; Google Cloud, 2025; Microsoft Azure, 2025). Across our experiments, such fixed choices incur up to $4.00\times$ the serving cost ($2.93\times$ on average) and achieve up to $12.84\times$ lower goodput than a system that jointly adapts these decisions (§ 5.1).

In this paper, we study how to serve generative AI tasks under dynamic workloads and resource constraints. Our goal is to increase goodput, the rate of requests completed while satisfying their Service-Level Objectives (SLOs), while reducing serving cost. Existing systems optimize only subsets of the decisions that determine a workflow-variant (Chase, 2022; LangChain, 2025; CrewAI, 2024; Wu et al., 2024; Google Cloud, 2025; Microsoft Azure, 2025; Zhang et al., 2025; Chen et al., 2025; Ong et al., 2025; Liakopoulos et al., 2025; Agrawal et al., 2024; Li et al., 2023). AFlow (Zhang et al., 2025) focuses on task decomposition; LLMSelector (Chen et al., 2025) and RouteLLM (Ong et al., 2025) focus on model selection; and MaverIQ (Liakopoulos et al., 2025), Vidur (Agrawal et al., 2024), and AlpaServe (Li et al., 2023) support deployment selection. Murakkab (Chaudhry et al., 2026), the closest prior system, jointly considers model and deployment choices but fixes the user-provided workflow. Although effective within their respective dimensions, these approaches do not jointly revisit workflows, model assignments, and deployment configurations as workload and cluster conditions change. Consequently, they fail to capture opportunities that arise only when these decisions are made jointly: particularly reuse of LLMs across tasks and adaptation to the current cluster state (e.g., already deployed models, load, and resource availability). We find that this leads to suboptimal performance and resource utilization (§ 2.1–2.2). For example, when serving seven tasks, we observed that an oracle that searches across all three dimensions reduces serving cost by $1.85\times$ and increases goodput by $1.69\times$ relative to Murakkab (§ 2.1).

We argue that serving systems need to make these decisions on behalf of users, rather than relying on statically fixed choices or requiring manual configuration. We propose a *task-level interface*: users specify only the task along with accuracy and performance SLOs, and the system selects a *workflow-variant*—a combination of workflow, model choices, and deployment configurations. To support this interface, the serving system needs to *explore the full design space of workflow-variants and select one per task* to improve serving cost and goodput.

Joint selection is especially important as different tasks can reuse the same deployed LLM. When a suitable model is already deployed and lightly loaded, reusing it can avoid the cost of an additional deployment. The same decision may violate latency SLOs when that model is heavily utilized. Because deployed models, their load, and available resources change as tasks arrive, reuse decisions cannot be fixed offline. Instead, the serving system must select a workflow-variant and make *on-the-fly reuse* decisions to determine which deployed models can be reused based on the current cluster state. This turns workflow-variant selection into an online constrained-search problem.

However, existing off-the-shelf algorithms are not well-suited for this setting. For the task in Figure 1, exhaustively evaluating the variants induced by three workflows, 11 models, and 10 deployment configurations per model consumes approximately 40 CPU-minutes. On a single CPU, this is approximately $4.7\times$ the task's most relaxed latency SLO of ~ 510s. Recent profilers (Liakopoulos et al., 2025; Agrawal et al., 2024; Hsia et al., 2025; Lee et al., 2025) reduce the cost of profiling individual LLM configurations, but do not eliminate the cost of evaluating workflows composed of multiple models. General-purpose methods such as Bayesian optimization (Močkus, 1975; Garnett, 2023; Zhou et al., 2022; Acun et al., 2026) and evolutionary search (Eiben & Smith, 2015; Walter et al., 2022) also rely on repeated candidate evaluations. In our setting, this evaluation cost must be paid whenever changing load or reuse opportunities alter the feasible choices. Thus, meeting the online decision budget requires exploiting the structure of this problem to eliminate candidates before evaluating complete workflow-variants.

To this end, we build `FlowMaker`, an inference serving system for generative AI workflows that meets user SLOs while minimizing serving cost. `FlowMaker` exposes the task-level interface: users specify only the task along with any latency and accuracy SLOs. `FlowMaker` generates candidate workflows for each task and lets users select the suitable ones. Involving users in selecting among the generated candidate workflows is intentional for two reasons: (1) task decomposition is often ambiguous and can yield multiple valid interpretations (Huang et al., 2025), and (2) current LLMs are unreliable as autonomous planners (Valmeekam et al., 2023; 2025; Kambhampati et al., 2024). `FlowMaker` then selects a workflow-variant for each task by jointly reasoning over work-

flows, model choices, deployment configurations, and on-the-fly reuse under dynamic cluster state. To make this search fast, we design a novel load-aware pruning algorithm that progressively narrows the candidate space using accuracy feasibility, operational cost, and performance under the current load. It then selects a low-cost workflow-variant estimated to satisfy the user's latency and accuracy SLOs, when one exists, while considering opportunities to reuse already deployed models.

We make the following contributions:

- We formulate serving generative AI workflows as a joint selection problem over workflows, model assignments, and deployment configurations. We show that considering all three dimensions and making reuse decisions from the current cluster state materially improve goodput and serving cost.
- We design a load-aware workflow-variant selection algorithm that searches across this joint space without exhaustively evaluating variants. In our evaluation, it reduces search time by $3.7\times$ relative to Aquatope's (Zhou et al., 2022) Bayesian optimization while accounting for reuse across tasks.
- We implement `FlowMaker` atop TensorRT-LLM. Across loads and arrival patterns, `FlowMaker` increases goodput by 1.45–$10.14\times$ ($4.73\times$ avg) and reduces serving cost by 1.56–$2.77\times$ ($1.95\times$ avg) over Murakkab (Chaudhry et al., 2026). Joint search alone improves average goodput and cost by $4.34\times$ and $1.48\times$, respectively, while on-the-fly reuse further reduces cost by $1.31\times$.

2 Problem Setting and Motivation

We first define the decisions involved in deploying a generative AI task. We then show why effective serving requires (1) jointly selecting workflows, models, and deployment configurations and (2) deciding whether to reuse deployed LLM instances based on the current cluster state.

Task, request, and workflow. A *task* is a high-level user objective accomplished through one or more stages, including LLM calls. A *request* is an invocation of a deployed task with a concrete input. A *workflow* is an executable decomposition of a task into stages, their dependencies, and execution order. A stage may be a non-LLM operation (e.g., PDF-to-text conversion) or an LLM call (e.g., summarize or translate). Figure 1(a) shows the stages and relationships for our example task; Figure 1(b) shows three candidate workflows with different decompositions or execution orders.

Workflow variant. A workflow can be realized using different models and deployment configurations. A *workflow-variant* specifies a workflow and, for each LLM stage, the model used to execute it (e.g., Llama-3.1-8B or Qwen-2.5-7B) and the model's deployment configuration (e.g., its tensor- and pipeline-parallelism degrees). Variants of the same workflow can therefore differ in accuracy, latency, memory consumption, and serving cost. For the example in Figure 1, evaluating 3 workflows, 11 models, and 10 deployment configurations generates $\sim 24K$ variants across which latency varies by $189.3\times$ (0.6–119s), cost by $8725\times$ (2×10^{-5}–0.17 ¢), and accuracy by $3.3\times$ (14.4–48.0%).

Deployment and reuse. A selected workflow-variant must be mapped to model deployments. For each LLM stage, the system can deploy a new model instance or reuse a compatible one already deployed for another task. We define *reuse* as sharing a deployed LLM instance across stages of different tasks. Reuse can reduce serving cost, but adds load to the shared instance and can increase request latency. Thus, reuse is a cluster-state-dependent decision for realizing a workflow-variant.

Problem formulation. Given a task, its user-approved candidate workflows, latency and accuracy SLOs, and the current cluster state, the serving system must make two decisions: (1) select a workflow-variant, and (2) realize that variant by mapping each LLM stage to either a new or a compatible existing model instance. This decision is made when the task is deployed; subsequent requests use the selected deployment. The system seeks a low-cost selection predicted to satisfy both SLOs. We measure its cluster-level effectiveness using *goodput*: the rate of requests that complete within their latency SLO while using a workflow-variant that satisfies the task's accuracy SLO.

2.1 Why Select Workflows, Models, and Deployment Configurations Jointly?

Interacting decision dimensions. A task may admit multiple user-approved workflows; each LLM stage may use one of several compatible models; and each model can run under multiple tensor- and pipeline-parallelism configurations. These choices cannot be made independently: the workflow determines the stages and dependencies, model assignments determine attainable proxy accuracy and resource requirements, and deployment configurations determine latency, capacity, and cost.

In Figure 1, W_1 and W_2 each contain two LLM stages, whereas W_3 performs summarization and translation through one joint LLM stage; PDF conversion is a fixed non-LLM stage. With 11 models and 10 deployment configurations per LLM stage, this yields $2(11 \cdot 10)^2 + (11 \cdot 10) = 24{,}310$ workflow variants. Fixing any dimension in advance can therefore exclude low-cost variants satisfying

the end-to-end SLOs. Across the evaluated models and configurations, latency, memory, cost, and proxy accuracy expose different tradeoffs; Appendix C provides the complete characterization.

Prior systems restrict this joint space by fixing at least one dimension. AFlow (Zhang et al., 2025) varies workflows, LLMSelector (Chen et al., 2025) and RouteLLM (Ong et al., 2025) select models, and MaverIQ (Liakopoulos et al., 2025) and related systems (Agrawal et al., 2024; Lee et al., 2025; Li et al., 2023) support deployment selection. Murakkab (Chaudhry et al., 2026), the closest prior system, jointly selects models and deployment configurations but assumes a user-provided workflow.

Figure 2 quantifies the resulting opportunity. We serve seven tasks under bursty, medium-load request arrivals from the Azure code trace (Patel et al., 2024). The *Static* policy uses a fixed workflow-variant; Murakkab receives a default workflow and selects models and deployment configurations; and the *Oracle* exhaustively searches the joint space of candidate workflows, models, and deployment configurations. Murakkab incurs $1.85\times$ the oracle's serving cost, while the oracle achieves $1.69\times$ Murakkab's goodput by also considering alternative workflows.

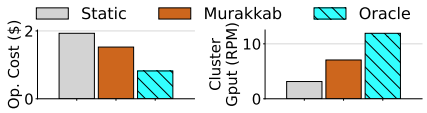

Figure 2: Serving cost and cluster goodput as the system is given progressively more choices: a fixed workflow variant (Static), model and deployment choices within a fixed workflow (Murakkab), and joint workflow, model, and deployment choices from a high-level task (Oracle). Joint optimization reduces cost and improves goodput.

LLM routing alone is insufficient. Routing systems (Yue et al., 2025; Dai et al., 2025; Ong et al., 2025; Jitkrittum et al., 2026) select among already-deployed models, whereas `FlowMaker` also determines which workflow, models, and deployment configurations should be deployed. So, LLM routing is complementary; for details, see Appendix B. We also evaluate against RouteLLM (Ong et al., 2025) in § 5.

Requirement 1. The serving system should jointly search workflows, models, and deployment configurations without fixing any dimension in advance, while keeping search efficient. Exhaustively evaluating 24,310 workflow-variants takes about 40 CPU-minutes, nearly $4.7\times$ the task's most relaxed ~ 510s latency SLO, making direct enumeration impractical at deployment time.

2.2 Why Reuse Must Account for Current Load?

Selecting across workflows and models can expose opportunities for stages from different tasks to share a deployed LLM instance. Such reuse avoids a separate deployment for each task, but its feasibility depends on the load already placed on the shared instance. A deployment that can accommodate another task under low load may violate latency SLOs when its existing request streams approach its SLO capacity—the maximum request rate it can sustain while meeting its latency target. Because request rates and resource availability change over time (Sinha et al., 2026; Zhou et al., 2022; Bhasi et al., 2022; Wang et al., 2025; Patel et al., 2024; Stojkovic et al., 2025; Qiu et al., 2026; Albalawi, 2025; Cortez et al., 2017), reuse decisions based on a fixed load estimate can either provision unnecessary capacity or overload shared instances.

Murakkab (Chaudhry et al., 2026) makes reuse decisions offline. We instantiate its offline planner in two ways: *OfflineReuseAvg* provisions shared instances for the average request load, while *OfflineReuseMax* provisions them for the maximum load. We evaluate both while serving seven tasks whose request streams follow the bursty Azure code trace (Patel et al., 2024). Figure 3 shows that OfflineReuseMax incurs $1.36\times$ OfflineReuseAvg's cost but achieves $1.29\times$ its goodput. Thus, provisioning for average load reduces cost but creates contention during bursts, while provisioning for peak load preserves more capacity at higher cost. *OnTheFlyReuse* instead makes reuse decisions as a task is deployed, using the current load and available capacity of existing model instances. Compared with Murakkab's offline policies, OnTheFlyReuse improves both cost and goodput by 1.89–$2.57\times$ and 1.13–$1.45\times$, respectively.

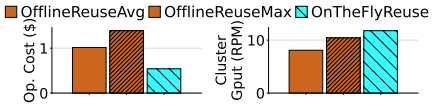

Figure 3: Serving cost and cluster goodput for reuse policies provisioned for average, maximum, or current cluster load (OnTheFlyReuse). Adapting reuse decisions to current load reduces cost and improves goodput.

Requirement 2. The serving system should therefore make reuse decisions based on the current load and capacity of deployed model instances. Together, efficient joint selection and load-aware reuse motivate `FlowMaker`'s design in the following section.

3 FlowMaker

The previous section established two requirements: selection must jointly consider workflows, models, and deployment configurations, and reuse decisions must account for the current cluster state.

FlowMaker meets these requirements by separating reusable offline profiling from state-dependent selection at task-deployment time. Figure 4 shows its architecture.

3.1 Task-Level Interface and System Lifecycle

FlowMaker exposes a task-level interface (see Appendix F-APIs). Through deploy_task, users submit a natural-language description of a task and its accuracy and latency SLOs. FlowMaker generates candidate workflows, from which the user identifies the semantically acceptable ones. This validation is necessary because task decomposition can admit multiple valid interpretations (Huang et al., 2025), and current LLMs remain unreliable as autonomous planners (Valmeekam et al., 2023; 2025; Kambhampati et al., 2024). FlowMaker then selects a workflow-variant, determines which deployed model instances can be reused, and deploys any remaining models. Subsequent inference_task requests use this selected deployment. Expert users may add constraints or directly specify workflows, models, or deployment configurations; FlowMaker respects these choices while still exploiting compatible deployed instances when possible. FlowMaker's lifecycle consists of three steps.

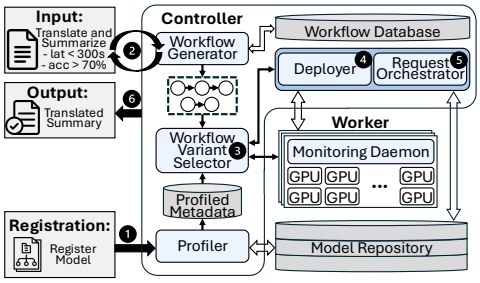

Figure 4: FlowMaker. See § 3.1.

Model registration and profiling. When a model is registered through register_model, FlowMaker stores its weights and profiles its supported tensor- and pipeline-parallelism configurations. The Profiler builds on MaverIQ (Liakopoulos et al., 2025) to estimate the latency and memory requirements of each configuration. These profiles are stored and reused across tasks.

Task deployment. When deploy_task is invoked, the Workflow Generator uses an LLM to decompose the task into candidate workflows and refines them through user interaction (§ 3.1). It caches the user-approved workflows and passes them to the Workflow-Variant Selector. The Selector combines these workflows with model profiles, user SLOs, and the current cluster state, including deployed models, their load, and available GPU memory, to select a workflow-variant and reuse plan. The Deployer then launches only the model instances that cannot be served through reuse.

Request execution. After deployment, the Orchestrator executes incoming requests using the selected workflow, models, configurations, and reuse mappings. Intermediate outputs are passed according to the workflow dependencies, and the final result is returned to the user. Workflow-variant selection is therefore performed when the task is deployed, rather than for every request.

3.2 Workflow-Variant Selection Problem

Inputs. The selector receives a task $\mathcal{T}$, its accuracy SLO α, latency SLO λ, and expected request arrival rate ρ, estimated from historical request traces (Li et al., 2023; Chaudhry et al., 2026). It also receives the task's user-approved candidate workflows $\mathcal{W}$, supported models $\mathcal{M}$, and deployment configurations $\mathcal{C}$. Finally, it observes the current cluster state, including the set of deployed model instances $\mathcal{D} = \{n_1, \ldots, n_d\}$, their configurations and serving loads, and available GPU memory.

Output. For a workflow with k LLM stages, the selector returns a workflow-variant and its deployment mapping, represented as $\mathcal{WV} = \langle \mathbf{n}, \mathbf{gpu} \rangle$, where $\mathbf{n} = (n_1, \ldots, n_k)$ specifies each stage's model and deployment configuration, and $\mathbf{gpu}$ specifies its GPU mapping. If a selected node n_i is in $\mathcal{D}$, the stage reuses that model instance; otherwise, FlowMaker deploys a new instance.

Objective and constraints. FlowMaker seeks a low-cost selection that satisfies the task's SLOs without exceeding the capacity of any reused model. For a candidate $\mathcal{WV}$, it computes:

$$\text{cost}(\mathcal{WV}) = \Big(\sum_{n_i \in \mathbf{n},\, n_i \notin \mathcal{D}} \text{mem}(n_i) \Big) \times \Big(\sum_{n_i \in \mathbf{n}} \text{lat}(n_i) \Big), \tag{1}$$

subject to latency, accuracy, and load capacity constraints:

$$\underbrace{\textstyle\sum_{n_i \in \mathbf{n}} \text{lat}(n_i) \le \lambda}_{\text{latency}}, \qquad \underbrace{\min_i \text{acc}(n_i) \ge \alpha}_{\text{accuracy}}, \qquad \underbrace{\text{cap}(n_i) \ge \text{load}(n_i) + \sigma_i \cdot \rho}_{\text{capacity}},$$

where σ_i counts the subtasks of $\mathcal{WV}$ routed through n_i and $\text{load}(n_i) = 0$ for new nodes. The capacity constraint ensures that a deployed model is shared only if the added load keeps it below the rate at which queuing begins. A candidate must also admit a placement under the GPUs' available memory. We define $\text{cost}(\varnothing) = \infty$.

Algorithm 1 Workflow-variant selection

Input: Task $\mathcal{T}$, accuracy SLO α, latency SLO λ, expected load ρ, model set $\mathcal{M}$, configuration set $\mathcal{C}$
Output: Workflow-variant selection $\mathcal{WV}^*$, or $\varnothing$

$\mathcal{WV}^* \leftarrow \varnothing$
for each workflow $[s_1, \ldots, s_k]$ in GETWORKFLOWS($\mathcal{T}$) **do**
▷ *Stage 1: Pruning by accuracy*
  $\mathcal{M}_i \leftarrow \{m \in \mathcal{M} \mid s_i \in \text{SKILLS}(m) \wedge \text{ACC}(m) \geq \alpha\}$, **for** $i = 1, \ldots, k$
  if $\exists i : \mathcal{M}_i = \varnothing$ **then continue** ▷ no admissible model: skip workflow
▷ *Stage 2: Pruning by operational cost*
  $\beta \leftarrow \sum_{i=1}^{k} \min_{m \in \mathcal{M}_i} \text{SIZE}(m)$; *candidates* $\leftarrow [\,]$ ▷ parameter budget
  for each $\mathbf{m} = (m_1, \ldots, m_k) \in \mathcal{M}_1 \times \cdots \times \mathcal{M}_k$ **do**
    $active_s \leftarrow \text{COMPUTEACTIVESIZE}(\mathbf{m})$; **if** $active_s \leq \beta$ **then** *candidates*.APPEND($(\mathbf{m}, active_s)$)
▷ *Stage 3: Pruning by system constraints*
  for each $(\mathbf{m}, active_s)$ in *candidates* sorted by $active_s$ ascending **do**
    $\mathcal{C}_{m_i} \leftarrow \{c \in \mathcal{C} \mid c \text{ is valid for } m_i\},\ \forall m_i \in \mathbf{m}$; $\mathcal{WV}_{\mathbf{m}} \leftarrow \varnothing$
    for each $(c_1, \ldots, c_k) \in \mathcal{C}_{m_1} \times \cdots \times \mathcal{C}_{m_k}$ **do**
      $\mathbf{n} \leftarrow [\text{NODE}(m_i, c_i)]_{i=1}^{k}$
      if $\sum_i \text{LAT}(n_i) > \lambda$ **then continue** ▷ latency SLO
      if $\exists i : \text{CAP}(n_i) < \text{LOAD}(n_i) + \sigma_i \rho$ **then continue** ▷ capacity
      $ok, \mathbf{gpu} \leftarrow \text{LEASTLOADED}(\mathbf{n})$; **if** $\neg ok$ **then continue** ▷ placement
      $\mathcal{WV} \leftarrow (\mathbf{n}, \mathbf{gpu})$
      if $\text{COST}(\mathcal{WV}) < \text{COST}(\mathcal{WV}_{\mathbf{m}})$ **then** $\mathcal{WV}_{\mathbf{m}} \leftarrow \mathcal{WV}$
    if $\mathcal{WV}_{\mathbf{m}} \neq \varnothing$ **then** $\mathcal{WV}^* \leftarrow \arg\min_{\text{COST}}(\mathcal{WV}^*, \mathcal{WV}_{\mathbf{m}})$; **break and go to line 2** ▷ first feasible
return $\mathcal{WV}^*$

Estimating workflow accuracy. To our knowledge, no general method estimates the end-to-end accuracy of an arbitrary generative AI workflow from per-model accuracies before execution. Following prior work (Liakopoulos et al., 2025), we use the average score across six Open LLM Leaderboard benchmarks (Fourrier et al., 2024) as each model's accuracy estimate. Motivated by evidence that per-stage and end-to-end accuracy are correlated (Chen et al., 2025; Dai et al., 2025), we approximate workflow accuracy as the minimum accuracy across its stages. More advanced task- or workflow-specific estimators can be incorporated without changing the remainder of the selector.

3.3 CONSTRAINT-GUIDED WORKFLOW-VARIANT SELECTION

Directly enumerating all workflow-variants is too expensive at task-deployment time. As shown in § 2.1, even our translation–summarization example contains 24,310 variants and requires approximately 40 CPU-minutes to evaluate with MaverIQ (Liakopoulos et al., 2025). Precomputed lookup tables also cannot directly capture changing reuse opportunities and model loads. Evolutionary search and Bayesian optimization, often used to manage resources for CPU-based workflows deployed on serverless platforms (Zhou et al., 2022; Bilal et al., 2023), avoid complete enumeration of the search space. However, for generative AI inference serving, they still require repeated evaluations of complete candidate configurations and introduce additional modeling or tuning overhead (Močkus, 1975; Garnett, 2023; Eiben & Smith, 2015). We further discuss these alternatives, along with graph-based formulations, in Appendix D and why generative AI workflows differ from traditional serverless workflows in Appendix B.

`FlowMaker` instead exploits when different constraints become evaluable. Accuracy can be checked after choosing models; incremental deployment footprint can be evaluated after constructing a model assignment and accounting for reuse; and latency, capacity, and memory feasibility can be evaluated after selecting deployment configurations and placement. `FlowMaker` orders its search according to this dependency structure, applying inexpensive constraints before expanding the next decision dimension. Algorithm 1 implements this strategy in three stages. It prioritizes finding a low-cost feasible workflow-variant quickly over slow yet globally optimal selection.

Stage 1: Pruning by accuracy. For each workflow stage, `FlowMaker` retains only models that support the required skill and satisfy the accuracy SLO. Under our minimum-stage accuracy estimator, any assignment containing a model below α is infeasible. If no model can serve a stage, `FlowMaker` skips the workflow. Because this check is independent of deployment configuration, it eliminates model–configuration combinations before constructing them.

Stage 2: Pruning by operational cost. After accuracy pruning, `FlowMaker` constructs candidate model assignments. Evaluating every deployment configuration at this point would remain expensive, so it uses model size as a proxy for deployment cost: model memory and latency generally increase with parameter count (Liakopoulos et al., 2025). The parameter budget β is the aggregate size of the smallest accuracy-feasible model for each stage, assuming no reuse. For an assignment $\mathbf{m}$, COMPUTEACTIVESIZE counts the parameters of the distinct models that must be newly deployed after accounting for existing deployments and reuse within the workflow. `FlowMaker` retains assignments whose active size does not exceed β and considers them in ascending active-size order. This heuristic favors assignments that exploit reuse or require a small incremental deployment footprint, but does not guarantee that every potentially lower-cost assignment is retained.

Stage 3: Pruning by system constraints. For each retained model assignment, in ascending active-size order, `FlowMaker` enumerates its valid tensor- and pipeline-parallelism configurations. It discards configurations that violate the end-to-end latency SLO or would cause a model instance to exceed its serving capacity. It then invokes LEASTLOADED, which attempts to map the required model shards to GPUs with sufficient free memory while preferring less-loaded GPUs.

`FlowMaker` keeps each assignment's lowest-cost feasible configuration. It compares the first feasible assignment with the current best selection and proceeds to the next workflow. Thus, it selects the smallest-footprint feasible assignment per workflow and returns the lowest-cost selection across workflows.

Search cost. The worst-case search space remains exponential in the number of workflow stages. For $|\mathcal{W}|$ workflows with $|\mathcal{S}|$ stages, exhaustive selection considers $O\left(|\mathcal{W}| \cdot (|\mathcal{M}| \cdot |\mathcal{C}| \cdot |\mathcal{D}|)^{|\mathcal{S}|}\right)$ model–configuration assignments, in addition to placement checks (see Appendix E for details).

4 IMPLEMENTATION AND EXPERIMENTAL SETUP

Prototype and testbed. We implement `FlowMaker` in $\sim$3K lines of Python atop TensorRT-LLM (NVIDIA Corporation, 2024), with a centralized CPU controller and per-worker daemons reporting model load and free GPU memory; the Profiler uses MaverIQ (Liakopoulos et al., 2025). Experiments run on $8\times$NVIDIA RTX A6000 GPUs (48GB each) with a 64-core AMD EPYC 7763 CPU, and all systems execute inference through the same backend (see Appendix F for details).

Workloads and models. We evaluate seven generative-AI tasks (Data and Beyond, 2025; Khurana et al., 2023; AIMonk Team, 2026), each with three candidate workflows of 1–3 stages, using inputs from six public datasets (See et al., 2017; Hermann et al., 2015; Chen et al., 2021a; Zhang et al., 2015; Chen et al., 2021b; Fan et al., 2018) and a catalog of 11 open-weight LLMs (1B–72B) from the Llama, Qwen, and Mistral families (Grattafiori et al., 2024; Qwen et al., 2025; Jiang et al., 2023). Following prior LLM-serving evaluations (Liakopoulos et al., 2025; Zhong et al., 2024; Li et al., 2023), we replay a bursty code and a sustained conversation Azure trace (Patel et al., 2024) at four offered-load levels (25–100%), preserving each trace's arrival structure and yielding 212–484 requests per run. Appendix G provides more information regarding the experimental details.

Comparison methods. The baselines differ in which decisions they fix. *LangChain* (Chase, 2022) fixes all three, receiving a default workflow, the smallest proxy-accuracy-feasible models, and TP $= 1$, PP $= 1$. *RouteLLM* (Ong et al., 2025) frees model choice within a fixed workflow, routing each request between Qwen-2.5-14B and Qwen-2.5-7B at thresholds favoring maximum quality, minimum cost, or balance the two; we provision enough replicas to avoid queuing, intentionally favoring its latency, and count all replicas in its cost. *Murakkab* (Chaudhry et al., 2026) also frees deployment configurations but fixes the workflow and decides reuse offline; as its implementation is unavailable, we reconstruct its optimizer and run its selections on our backend, so our comparison concerns this configuration rather than its broader feature set. We also evaluate *`FlowMakerNoReuse`*, which applies `FlowMaker`'s selector without reuse, and compare the selector against brute force and Aquatope-style Bayesian optimization (Zhou et al., 2022) (see Appendix G for more details).

SLOs and metrics. To set latency SLOs, for each task we measure its latency in isolation and set STRICT–RELAXED latency SLOs as 2–5$\times$ this value, following prior work (Zhong et al., 2024; Ahmad et al., 2024; Liakopoulos et al., 2025; Li et al., 2023). To set accuracy SLOs, we define a model's proxy accuracy as its average score across six Open LLM Leaderboard benchmarks (Fourrier et al., 2024), and a variant's score as the minimum across its models. Following Murakkab, BEST, GOOD, and BASIC accuracy SLOs correspond to the p95, p80, and p50 achievable variant scores. Because we do not evaluate generated outputs, all accuracy compliance refers to this proxy.

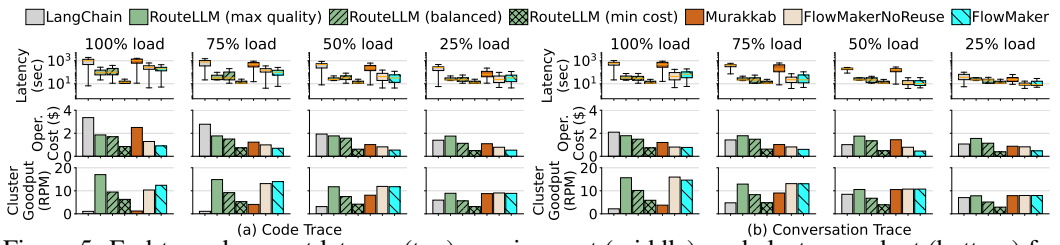

Figure 5: End-to-end request latency (top), serving cost (middle), and cluster goodput (bottom) for 7 tasks under the Azure (a) code and (b) conversation traces. See § 5.1 and § 5.2.

We report task latency, serving cost $\sum_j M_j T_j r$ (allocated GPU memory $\times$ residency time $\times$ GB-second rental rate), and cluster goodput: the rate of requests meeting their latency SLO under a variant that satisfies their proxy-accuracy SLO. Additional details appear in Appendix G.

5 Evaluation

We evaluate `FlowMaker`'s (1) end-to-end serving performance (§ 5.1), (2) gains from joint workflow-variant selection and on-the-fly reuse (§ 5.2), and (3) selection quality, efficiency, and scalability relative to brute-force search and Aquatope that leverages Bayesian optimization (§ 5.3). Appendix H reports fixed-capacity scaling, SLO sensitivity, and system overheads.

5.1 End-to-End Serving Performance

Figure 5 compares the systems on the same seven-task workload and task-specific SLOs, under bursty code and sustained conversation arrivals across four load levels (see Appendix H.1 for additional details). The systems differ in which of three decisions they can adapt: the workflow, the models used, and their deployment configurations. Fixing any of these decisions limits adaptation to workload and cluster conditions, reducing efficiency or SLO compliance.

LangChain fixes all three decisions and never reuses models. It deploys 15 instances (290GB) for 6 tasks but cannot place the 7th because memory fragmentation leaves no GPU with enough free capacity. At 50% code-trace load, it colocates 4 instances on one GPU, increasing average latency to 426s versus 38s for `FlowMaker`, which serves all 7 tasks with only 3–5 instances. Overall, `FlowMaker` achieves 1.13–12.84$\times$ higher goodput (5.2$\times$ avg) at 2.2–4$\times$ lower cost (2.93$\times$ avg).

RouteLLM can choose between models, but fixes the workflow and restricts deployment to a pre-deployed strong/weak pair. Its routing threshold therefore trades quality for cost only by shifting requests between these two models. At maximum quality, it meets all proxy-accuracy SLOs and achieves on average 5.8% higher goodput than `FlowMaker`, but reserves 206–235GB and costs 2.06–3.84$\times$ (2.95$\times$ avg) more. Its minimum-cost and balanced settings meet only 34.9–36.8% and 62.6–65.6% of proxy-accuracy SLOs, versus 100% for `FlowMaker`, yielding 2.62$\times$ and 1.52$\times$ lower average goodput. RouteLLM's lower latency comes from provisioning enough replicas to avoid queuing, which also drives up cost. More details are in Appendix H.1.

Murakkab additionally adapts deployment configurations, but still fixes the workflow and chooses reuse offline. At low load, goodput is similar (up to 1.46$\times$, 1.21$\times$ avg), while `FlowMaker` is 1.84–3.15$\times$ (2.23$\times$ avg) cheaper by selecting shorter workflows when beneficial. At high load, Murakkab's fixed workflow and reuse choices increase latency and queuing; `FlowMaker` achieves 1.45–10.14$\times$ (4.73$\times$ avg) higher goodput at 1.56–2.77$\times$ (1.95$\times$ avg) lower cost.

End-to-end takeaway. At high load, `FlowMaker` achieves 1.45–10.14$\times$ Murakkab's goodput (4.73$\times$ on average) while reducing serving cost by 1.56–2.77$\times$ (1.95$\times$ on average). RouteLLM lowers latency through replication, but its maximum-quality setting costs 2.06–3.84$\times$ more, while only 34.9–36.8% of requests satisfy their proxy-accuracy SLOs in its minimum-cost setting.

5.2 Impact of Joint Selection and On-the-Fly Reuse

Joint selection across workflows, models, and deployment configurations. We compare `FlowMakerNoReuse`, which jointly selects across all three dimensions but deploys separate instances per task, with Murakkab, which fixes each task's workflow but retains offline reuse (Figure 5). Because the systems also differ in reuse policy, this is conservative evidence for joint selection rather than a controlled isolation: `FlowMakerNoReuse` operates without reuse, while Murakkab benefits from it. At lower loads, `FlowMakerNoReuse` achieves up to 1.47$\times$ (1.13$\times$ on average) higher goodput and reduces cost by up to 1.83$\times$ (1.38$\times$ on average). Murakkab consumes less memory through reuse—117.11GB rather than 186.46GB in one conversation-trace setting—

but its fixed workflows contain at least two stages, whereas `FlowMakerNoReuse` can select one- or two-stage alternatives. Across tasks and loads, these choices increase Murakkab's latency by 2.41–10.73×, offsetting its memory savings. At higher loads, the additional latency causes queues to grow. Across both traces, `FlowMakerNoReuse` achieves 1.45–8.46× higher goodput (4.34× on average) and reduces cost by 1.24–1.95× (1.48× on average).

On-the-fly reuse. We isolate reuse by comparing `FlowMaker` with `FlowMakerNoReuse`; both apply the same joint-selection heuristic, but only `FlowMaker` can map stages from different tasks to shared instances. Their goodput remains similar across loads. At lower loads, `FlowMaker` reuses 3–4 instances, whereas `FlowMakerNoReuse` deploys 8, reducing cost by 1.47–1.73× (1.61× avg). At higher loads, `FlowMaker` shares less aggressively and deploys five instances to preserve capacity, but still reduces cost by 1.04–1.42× (1.31× avg). For example, under the code trace at 100% load, `FlowMaker` selects a three-stage variant that reuses an existing Qwen-2.5-3B instance, whereas `FlowMakerNoReuse` selects a two-stage variant requiring new deployments. Thus, the lower-cost choice depends on both the workflow structure and the current deployment state.

Ablation takeaway. Despite disabling reuse, `FlowMakerNoReuse` achieves 1.45–8.46× (4.34× on average) higher goodput and 1.24–1.95× (1.48× on average) lower cost than Murakkab at high load. Adding load-aware reuse further reduces `FlowMaker`'s cost by 1.04–1.42× (1.31× on average) without materially changing goodput.

5.3 Selection Quality, Efficiency, and Scalability

We compare `FlowMaker` with brute-force (BF) and Aquatope-style Bayesian optimization (BO) (Močkus, 1975; Garnett, 2023; Zhou et al., 2022; Acun et al., 2026). BF enumerates the candidate space and returns the minimum-cost feasible variant under Equation 1. BO uses Optuna's TPE sampler (Optuna Team, 2024), performs 200 sequential trials, and penalizes SLO violations.

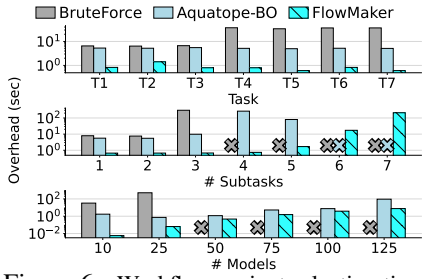

Figure 6: Workflow-variant selection time across tasks (top), workflow depth (middle), and model-catalog size (bottom).

Selection quality. Across the seven tasks and every comparable point in the model-catalog and workflow-depth scaling experiments, `FlowMaker`, BF, and BO select the same workflow variant. Thus, `FlowMaker` has a 100% observed match rate, a 0% observed objective gap relative to BF, and the same feasibility and SLO-satisfaction outcome as both reference methods. This provides empirical evidence that `FlowMaker`'s algorithm matches the selections of BF without incurring its substantial overheads.

Time efficiency. Figure 6 (top) shows that `FlowMaker` is 3.7–8.5× faster than BO and 4.5–61× faster than BF across the seven tasks. BF takes approximately 6.5s for the simpler tasks (T1–T3) and approximately 40s for the larger candidate spaces of T4–T7. BO requires approximately 5s per task for its 200 sequential trials. In contrast, `FlowMaker`'s staged heuristic filters candidates using proxy accuracy and active deployment size before evaluating deployment configurations, selecting a variant in approximately one second. The approximately 40s reported here measures BF enumeration over already-computed model profiles. In contrast, the approximately 40 CPU-minutes reported in § 2.1 measure end-to-end exhaustive evaluation of the example task, including computing the latency and memory estimates required to evaluate its 24,310 variants.

Scalability. Workflow depth most strongly affects search time (Figure 6, middle). BF's and BO's search time exceeds 15 minutes at ≥ 6 stages (indicated by `x` in the Figure), while `FlowMaker` stays below two seconds for 3–5 stages, yielding 14.5–443× speedups. At 7 stages, however, its search time rises to 210s, reflecting the underlying combinatorial growth. `FlowMaker` also scales better with model-catalog size (bottom): BF times out at 50 models, BO slows by 12.2× at 125 models, while `FlowMaker`'s early filtering reduces the model–configuration combinations explored.

6 Conclusion

`FlowMaker` is a serving system that takes a task and its latency and accuracy SLOs and jointly selects a workflow variant and state-aware reuse plan across workflows, models, and deployment configurations. Its constraint-guided selector matches brute-force and Bayesian-optimization selections while reducing selection time by 4.5–61× and 3.7–8.5×, respectively. At high load, `FlowMaker` achieves 4.73× Murakkab's goodput while reducing serving cost by 1.95× on average, showing the benefits of joint, state-aware selection for cost-efficient generative AI workflow serving.

AI USE STATEMENT

In this work, we used generative AI tools to help implement and tune the baseline systems we compare against, to write and edit software code for our prototype and experiment scripts, and to improve the readability of parts of the manuscript. We have not used generative AI tools to develop the conceptual framework, formulate claims, design the methodology or experiments, or interpret results, and the remaining categories in the ICLR policy are not applicable. We have reviewed all AI-assisted work: AI-assisted code was verified and tested by the authors, and baseline implementations were checked against their original papers. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REFERENCES

Bilge Acun, Prasoon Sinha, Newsha Ardalani, Sangmin Bae, Alicia Golden, Meghana Madhyastha, Chien-Yu Lin, Fei Sun, Neeraja J. Yadwadkar, and Carole-Jean Wu. Composer: A search framework for hybrid neural architecture design. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=m00gjQfpCc.

Ajit Agrawal, Philip Klein, and R. Ravi. When trees collide: an approximation algorithm for the generalized Steiner problem on networks. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, STOC '91, pp. 134–144, New York, NY, USA, 1991. Association for Computing Machinery. ISBN 0897913973. doi: 10.1145/103418.103437. URL https://doi.org/10.1145/103418.103437.

Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S. Gulavani, Ramachandran Ramjee, and Alexey Tumanov. VIDUR: A LARGE-SCALE SIMULATION FRAMEWORK FOR LLM INFERENCE. In P. Gibbons, G. Pekhimenko, and C. De Sa (eds.), *Proceedings of Machine Learning and Systems*, volume 6, pp. 351–366, 2024. URL https://proceedings.mlsys.org/paper_files/paper/2024/file/b74a8de47d2b3c928360e0a011f48351-Paper-Conference.pdf.

Sohaib Ahmad, Hui Guan, Brian D. Friedman, Thomas Williams, Ramesh K. Sitaraman, and Thomas Woo. Proteus: A High-Throughput Inference-Serving System with Accuracy Scaling. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, pp. 318–334, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703720. doi: 10.1145/3617232.3624849. URL https://doi.org/10.1145/3617232.3624849.

AIMonk Team. 10+ Amazing Natural Language Processing Examples in 2026. https://aimonk.com/natural-language-processing-examples/, 2026. Published March 10, 2026; accessed April 2026.

Nasser S. Albalawi. Dynamic scheduling strategies for cloud-based load balancing in parallel and distributed systems. *J. Cloud Comput.*, 14(1), July 2025. ISSN 2192-113X. doi: 10.1186/s13677-025-00757-6. URL https://doi.org/10.1186/s13677-025-00757-6.

Amazon Web Services. Amazon Quick Flows. https://aws.amazon.com/quick/flows/, 2025.

Amazon Web Services. AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/, 2026.

Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. DeepSpeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022. ISBN 9784665454445. URL https://dl.acm.org/doi/abs/10.5555/3571885.3571946.

Lixiang Ao, George Porter, and Geoffrey M. Voelker. FaaSnap: FaaS made fast using snapshot-based VMs. In *Proceedings of the Seventeenth European Conference on Computer Systems*,

EuroSys '22, pp. 730–746, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391627. doi: 10.1145/3492321.3524270. URL https://doi.org/10.1145/3492321.3524270.

Ricardo Baptista and Matthias Poloczek. Bayesian optimization of combinatorial structures. In *International conference on machine learning*, pp. 462–471. PMLR, 2018. URL https://proceedings.mlr.press/v80/baptista18a/baptista18a.pdf.

Vivek M. Bhasi, Jashwant Raj Gunasekaran, Aakash Sharma, Mahmut Taylan Kandemir, and Chita Das. Cypress: input size-sensitive container provisioning and request scheduling for serverless platforms. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, pp. 257–272, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394147. doi: 10.1145/3542929.3563464. URL https://doi.org/10.1145/3542929.3563464.

Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. With great freedom comes great opportunity: Rethinking resource allocation for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, pp. 381–397, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394871. doi: 10.1145/3552326.3567506. URL https://doi.org/10.1145/3552326.3567506.

Harrison Chase. LangChain. https://www.langchain.com/, 2022.

Gohar Irfan Chaudhry, Esha Choukse, Haoran Qiu, Inigo Goiri, Rodrigo Fonseca, Adam Belay, and Ricardo Bianchini. Murakkab: Resource-Efficient Agentic Workflow Orchestration in Cloud Platforms. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pp. 567–587, Seattle, WA, July 2026. USENIX Association. ISBN 978-1-939133-55-7. URL https://www.usenix.org/conference/osdi26/presentation/chaudhry.

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. URL https://openreview.net/forum?id=NphowWHYJj.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, 2021a. URL https://arxiv.org/abs/2107.03374.

Yulong Chen, Yang Liu, Liang Chen, and Yue Zhang. DialogSum: A real-life scenario dialogue summarization dataset. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pp. 5062–5074, Online, August 2021b. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-acl.449. URL https://aclanthology.org/2021.findings-acl.449.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022. URL https://mitpress.mit.edu/9780262367509/introduction-to-algorithms/.

Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pp. 153–167, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132772. URL https://doi.org/10.1145/3132747.3132772.

CrewAI. CrewAI: Multi-Agent Platform for Automating AI Workflows, 2024. URL https://crewai.com/.

Yinwei Dai, Zhuofu Chen, Anand Iyer, and Ravi Netravali. Aragog: Just-in-time model routing for scalable serving of agentic workflows, 2025. URL https://arxiv.org/abs/2511.20975.

Data and Beyond. Practical generative ai: 9 nlp tasks with hugging face transformers. https://medium.com/data-and-beyond/practical-generative-ai-9-nlp-tasks-with-hugging-face-transformers-2ec7754a76e2, 2025. Medium article, accessed April 2026.

E. W. Dijkstra. *A Note on Two Problems in Connexion with Graphs*, pp. 287–290. Association for Computing Machinery, New York, NY, USA, 1 edition, 2022. ISBN 9781450397735. URL https://doi.org/10.1145/3544585.3544600.

Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pp. 467–481, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378512. URL https://doi.org/10.1145/3373376.3378512.

Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Springer, 2015. URL https://doi.org/10.1007/978-3-662-05094-1.

Simon Eismann, Johannes Grohmann, Erwin van Eyk, Nikolas Herbst, and Samuel Kounev. Predicting the Costs of Serverless Workflows. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, pp. 265–276, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450369916. doi: 10.1145/3358960.3379133. URL https://doi.org/10.1145/3358960.3379133.

Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. In Iryna Gurevych and Yusuke Miyao (eds.), *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 889–898, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1082. URL https://aclanthology.org/P18-1082/.

Clémentine Fourrier, Nathan Habib, Alina Lozovskaya, Konrad Szafer, and Thomas Wolf. Open LLM Leaderboard. https://huggingface.co/open-llm-leaderboard, 2024. Accessed: 2026-03-18.

Peter I. Frazier. A Tutorial on Bayesian Optimization, 2018. URL https://arxiv.org/abs/1807.02811.

Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. Serverlessllm: low-latency serverless inference for large language models. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://dl.acm.org/doi/abs/10.5555/3691938.3691946.

Roman Garnett. *Bayesian Optimization*. Cambridge University Press, 2023.

Michel X. Goemans and David P. Williamson. A general approximation technique for constrained forest problems. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, pp. 307–316, USA, 1992. Society for Industrial and Applied Mathematics. ISBN 089791466X. URL https://dl.acm.org/doi/10.5555/139404.139468.

Google Cloud. Vertex ai agent engine overview. https://docs.cloud.google.com/agent-builder/agent-engine/overview, 2025.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. URL https://arxiv.org/abs/2407.21783.

Anupam Gupta and Amit Kumar. Online steiner tree with deletions. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pp. 455–467, USA, 2014. Society for Industrial and Applied Mathematics. ISBN 9781611973389. URL https://dl.acm.org/doi/10.5555/2634074.2634108.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi: 10.1109/TSSC.1968.300136. URL https://doi.org/10.1109/TSSC.1968.300136.

Zijian He, Reyna Abhyankar, Vikranth Srivatsa, and Yiying Zhang. Cognify: Supercharging Gen-AI Workflows With Hierarchical Autotuning. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, KDD '25, pp. 932–943, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400714542. doi: 10.1145/3711896.3736884. URL https://doi.org/10.1145/3711896.3736884.

Karl Moritz Hermann, Tomás Kociský, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching Machines to Read and Comprehend. In *NIPS*, pp. 1693–1701, 2015. URL http://papers.nips.cc/paper/5945-teaching-machines-to-read-and-comprehend.

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. URL https://openreview.net/forum?id=VtmBAGCN7o.

Samuel Hsia, Alicia Golden, Bilge Acun, Newsha Ardalani, Zachary DeVito, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. MAD-Max Beyond Single-Node: Enabling Large Machine Learning Model Acceleration on Distributed Systems. In *Proceedings of the 51st Annual International Symposium on Computer Architecture*, ISCA '24, pp. 818–833. IEEE Press, 2025. ISBN 9798350326581. doi: 10.1109/ISCA59077.2024.00064. URL https://doi.org/10.1109/ISCA59077.2024.00064.

Shengran Hu, Cong Lu, and Jeff Clune. Automated Design of Agentic Systems. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=t9U3LW7JVX.

Sukai Huang, Nir Lipovetzky, and Trevor Cohn. Planning in the dark: LLM-symbolic planning pipeline without experts. In *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'25/IAAI'25/EAAI'25. AAAI Press, 2025. ISBN 978-1-57735-897-8. doi: 10.1609/aaai.v39i25.34855. URL https://doi.org/10.1609/aaai.v39i25.34855.

InferX AI. InferX. https://inferx.net/, 2026.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7B, 2023. URL https://arxiv.org/abs/2310.06825.

Wittawat Jitkrittum, Harikrishna Narasimhan, Ankit Singh Rawat, Jeevesh Juneja, Congchao Wang, Zifeng Wang, Alec Go, Chen-Yu Lee, Pradeep Shenoy, Rina Panigrahy, Aditya Krishna Menon, and Sanjiv Kumar. Universal Model Routing for Efficient LLM Inference. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=ka82fvJ5f1.

Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Paul Saldyt, and Anil B Murthy. Position: LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=Th8JPEmH4z.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines, 2023. URL https://arxiv.org/abs/2310.03714.

Diksha Khurana, Aditya Koli, Kiran Khatter, and Sukhdev Singh. Natural language processing: state of the art, current trends and challenges. *Multimedia tools and applications*, 82(3):3713–3744, 2023. URL https://doi.org/10.1007/s11042-022-13428-4.

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*, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Marco Laju, Donghyun Son, Saurabh Agarwal, Nitin Kedia, Myungjin Lee, Jayanth Srinivasa, and Aditya Akella. Nalar: An agent serving framework, 2026. URL https://arxiv.org/abs/2601.05109.

LangChain. LangGraph: Agent Orchestration Framework for Reliable AI Applications. https://www.langchain.com/langgraph, 2025.

Seonho Lee, Amar Phanishayee, and Divya Mahajan. Forecasting GPU Performance for Deep Learning Training and Inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, pp. 493–508, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400706981. doi: 10.1145/3669940.3707265. URL https://doi.org/10.1145/3669940.3707265.

Zelong Li, Shuyuan Xu, Kai Mei, Wenyue Hua, Balaji Rama, Om Raheja, Hao Wang, He Zhu, and Yongfeng Zhang. AutoFlow: Automated Workflow Generation for Large Language Model Agents, 2024. URL https://arxiv.org/abs/2407.12821.

Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 663–679, Boston, MA, July 2023. USENIX Association. ISBN 978-1-939133-34-2. URL https://www.usenix.org/conference/osdi23/presentation/li-zhouhan.

Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. FaaSFlow: enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pp. 782–796, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507717. URL https://doi.org/10.1145/3503222.3507717.

Dimitrios Liakopoulos, Prasoon Sinha, Tianrui Hu, Myungjin Lee, and Neeraja J. Yadwadkar. MaverIQ: Fingerprint-Guided Extrapolation and Fragmentation-Aware Layering for Intent-Based LLM Serving. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '25, pp. 1676–1696, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400714665. doi: 10.1145/3712285.3759867. URL https://doi.org/10.1145/3712285.3759867.

Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient Serving of LLM-based Applications with Semantic Variable. In *18th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pp. 929–945, Santa Clara, CA, July 2024a. USENIX Association. ISBN 978-1-939133-40-3. URL https://www.usenix.org/conference/osdi24/presentation/lin-chaofan.

Fangru Lin, Emanuele La Malfa, Valentin Hofmann, Elle Michelle Yang, Anthony G. Cohn, and Janet B. Pierrehumbert. Graph-enhanced large language models in asynchronous plan reasoning. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024b. URL https://dl.acm.org/doi/10.5555/3692070.3693283.

Matthieu Lin, Jenny Sheng, Andrew Zhao, Shenzhi Wang, Yang Yue, Victor Shea Jay Huang, Huan Liu, Jun Liu, Gao Huang, and Yong-Jin Liu. Scaffolded Language Models with Language Supervision for Mixed-Autonomy: A Survey, 2025. URL https://arxiv.org/abs/2410.16392.

Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 303–320, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/mahgoub.

Kai Mei, Xi Zhu, Wujiang Xu, Mingyu Jin, Wenyue Hua, Zelong Li, Shuyuan Xu, Ruosong Ye, Yingqiang Ge, and Yongfeng Zhang. AIOS: LLM agent operating system. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=L4HHkCDz2x.

Microsoft Azure. Azure AI Foundry: Build, Deploy, and Manage AI Applications and Agents. https://azure.microsoft.com/en-us/products/ai-foundry/, 2025.

J. Močkus. On bayesian methods for seeking the extremum. In G. I. Marchuk (ed.), *Optimization Techniques IFIP Technical Conference Novosibirsk, July 1–7, 1974*, pp. 400–404, Berlin, Heidelberg, 1975. Springer Berlin Heidelberg. ISBN 978-3-540-37497-8. URL https://doi.org/10.1007/3-540-07165-2_55.

E.F. Moore. *The Shortest Path Through a Maze*. Bell Telephone System. Technical publications. monograph. Bell Telephone System., 1959. URL https://books.google.co.in/books?id=IVZBHAAACAAJ.

NVIDIA Corporation. TensorRT-LLM: Optimized Inference for Large Language Models. https://github.com/NVIDIA/TensorRT-LLM, 2024.

NVIDIA Corporation. NVIDIA Management Library (NVML). https://developer.nvidia.com/management-library-nvml, 2025. Accessed: 2026-03-31.

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. URL https://openreview.net/forum?id=8sSqNntaMr.

OpenAI. gpt-oss-120b Model. https://developers.openai.com/api/docs/models/gpt-oss-120b, 2025. OpenAI API Documentation.

Optuna Team. TPE Sampler. https://hub.optuna.org/samplers/tpe/, 2024.

Zaifeng Pan, AJJKUMAR PATEL, Yipeng Shen, Zhengding Hu, Yue Guan, Wan-Lu Li, Lianhui Qin, Yida Wang, and Yufei Ding. KVFlow: Efficient prefix caching for accelerating LLM-based multi-agent workflows. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=5Iw1nDtYmT.

Leonard Papenmeier, Matthias Poloczek, and Luigi Nardi. Understanding high-dimensional bayesian optimization. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=1d2fpvyKvJ.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *ISCA*, June 2024. URL https://www.microsoft.com/en-us/research/publication/splitwise-efficient-generative-llm-inference-using-phase-splitting/.

Haoran Qiu, Anish Biswas, Zihan Zhao, Jayashree Mohan, Alind Khare, Esha Choukse, Íñigo Goiri, Zeyu Zhang, Haiying Shen, Chetan Bansal, Ram Ramjee, and Rodrigo Fonseca. ModServe: Modality- and Stage-Aware Resource Disaggregation for Scalable Multimodal Model Serving. In *Proceedings of the 2025 ACM Symposium on Cloud Computing*, SoCC '25, pp. 817–830, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400722769. doi: 10.1145/3772052.3772254. URL https://doi.org/10.1145/3772052.3772254.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 Technical Report, 2025. URL https://arxiv.org/abs/2412.15115.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009. ISBN 0136042597. URL https://dl.acm.org/doi/10.5555/1671238.

Abigail See, Peter J. Liu, and Christopher D. Manning. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1073–1083, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1099. URL https://www.aclweb.org/anthology/P17-1099.

Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 205–218. USENIX Association, July 2020. ISBN 978-1-939133-14-4. URL https://www.usenix.org/conference/atc20/presentation/shahrad.

Prasoon Sinha, Kostis Kaffes, and Neeraja J. Yadwadkar. ALAP: Intent-Based Serverless Computing via Delayed Decision-Making. In *Proceedings of the 2025 ACM Symposium on Cloud Computing*, SoCC '25, pp. 831–846, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400722769. doi: 10.1145/3772052.3772262. URL https://doi.org/10.1145/3772052.3772262.

Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1348–1362, 2025. doi: 10.1109/HPCA61900.2025.00102. URL https://doi.org/10.1109/HPCA61900.2025.00102.

Jinwei Su, Qizhen Lan, Yinghui Xia, Lifan Sun, Weiyou Tian, Tianyu Shi, Xinyuan Song, Lewei He, and Yang Jingsong. Difficulty-Aware Agentic Orchestration for Query-Specific Multi-Agent Workflows, 2026. URL https://arxiv.org/abs/2509.11079.

Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2): 146–160, 1972. doi: 10.1137/0201010. URL https://doi.org/10.1137/0201010.

Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pp. 559–572, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446714. URL https://doi.org/10.1145/3445814.3446714.

Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. On the planning abilities of large language models - a critical investigation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=X6dEqXIsEW.

Karthik Valmeekam, Kaya Stechly, Atharva Gundawar, and Subbarao Kambhampati. A Systematic Evaluation of the Planning and Scheduling Abilities of the Reasoning Model o1. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=FkKBxp0FhR.

Noppanat Wadlom, Junyi Shen, and Yao Lu. Efficient LLM Serving for Agentic Workflows: A Data Systems Perspective, 2026. URL https://arxiv.org/abs/2603.16104.

Mathew J. Walter, David J. Walker, and Matthew J. Craven. An explainable visualisation of the evolutionary search process. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '22, pp. 1794–1802, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392686. doi: 10.1145/3520304.3533984. URL https://doi.org/10.1145/3520304.3533984.

Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. BurstGPT: A Real-World Workload Dataset to Optimize LLM Serving Systems. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, KDD '25, pp. 5831–5841, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400714542. doi: 10.1145/3711896.3737413. URL https://doi.org/10.1145/3711896.3737413.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088. URL https://dl.acm.org/doi/10.5555/3600270.3602070.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

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/.

Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. The Shift from Models to Compound AI Systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/, 2024.

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. URL https://openreview.net/forum?id=z5uVAKwmjf.

Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, pp. 649–657, Cambridge, MA, USA, 2015. MIT Press. URL https://dl.acm.org/doi/10.5555/2969239.2969312.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, OSDI'24, USA, 2024. USENIX Association. ISBN 978-1-939133-40-3. URL https://dl.acm.org/doi/10.5555/3691938.3691949.

Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. AQUATOPE: QoS-and-Uncertainty-Aware Resource Management for Multi-stage Serverless Workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, pp. 1–14, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450399159. doi: 10.1145/3567955.3567960. URL https://doi.org/10.1145/3567955.3567960.

A Appendix Organization

The appendix is organized as follows. Appendix B extends the related work, positioning `FlowMaker` relative to workflow construction, model routing, generative-AI workflow serving, and serverless workflow systems. Appendix C characterizes the three dimensions of the workflow-variant space: workflows, models, and deployment configurations. Appendix D discusses alternative selection approaches (lookup tables, Bayesian optimization and evolutionary search, and graph algorithms) and their limitations. Appendix E summarizes notation, describes progressive SLO relaxation, and analyzes the complexity of Algorithm 1. Appendix F details `FlowMaker`'s APIs and prototype implementation. Appendix G describes the experimental setup, including trace construction, tasks, the model catalog, baseline configurations, SLO definitions, the testbed, and metrics. Appendix H reports additional results: a per-baseline breakdown of the end-to-end comparison (§H.1), scaling under fixed cluster capacity (§H.2), sensitivity to latency SLOs (§H.3), and model-loading overheads (§H.4).

B Extended Related Work

Workflow construction and application-level optimization. Prior work automates different aspects of generative-AI application design. DSPy (Khattab et al., 2023) compiles declarative LM programs by optimizing prompts and demonstrations. AutoFlow (Li et al., 2024) generates natural-language agent workflows using fine-tuning- and in-context-based optimization, while ADAS (Hu et al., 2025) searches over code-defined agentic systems and AFlow (Zhang et al., 2025) refines code-represented workflows using Monte Carlo tree search and execution feedback. Cognify (He et al., 2025) autotunes an existing workflow through hierarchical search over structural changes, step operators—including model choice—and prompts, using workflow-level measurements of output quality, latency, and monetary cost. DAAO (Su et al., 2026) instead adapts workflow depth, reasoning operators, and heterogeneous model assignments to individual queries based on their estimated difficulty.

These systems optimize application logic, prompts, or per-query model use. They do not jointly choose physical deployment configurations, such as tensor- and pipeline-parallelism, or determine whether stages should use new or already deployed model instances based on current load and GPU availability. They are therefore complementary to `FlowMaker`: their generated or tuned workflows can supply candidate workflows, after which `FlowMaker` selects a deployable workflow-variant and reuse plan at task-deployment time.

Model selection and routing. LLMSelector (Chen et al., 2025) assigns models to modules in a fixed compound-AI architecture. RouteLLM (Ong et al., 2025) routes each query between a fixed strong–weak model pair, while MasRouter (Yue et al., 2025) additionally selects collaboration modes, agent roles, and models for multi-agent systems. Other LLM routers similarly select among available models to balance output quality and inference cost (Jitkrittum et al., 2026). These approaches determine which model processes an input, but do not jointly select alternative workflow decompositions, physical deployment configurations, and new-versus-reused model instances under current cluster state. They can be combined with `FlowMaker` to route individual requests among the instances selected when a task is deployed.

Generative-AI workflow serving. Aragog (Dai et al., 2025) assumes a supplied workflow DAG and a configured pool of model-serving engines. For each request, it identifies accuracy-preserving model assignments and selects among them stage by stage using current engine load. Thus, Aragog adapts model routing during the execution of a fixed workflow but does not select the workflow structure or determine which model configurations and instances should be deployed.

Murakkab (Chaudhry et al., 2026) exposes a declarative workflow interface and uses offline workflow and model profiles with periodic MILP optimization. Its optimizer selects workflow parameters, executors, model and hardware configurations, parallelism strategies, and instance assignments under predicted load. Murakkab can multiplex model instances across workflows and periodically reconfigure its deployment as demand and resource availability change. `FlowMaker` addresses a different decision scope and timescale: when a task is deployed, it selects among user-approved alternative workflows and their induced model and deployment choices, and maps each stage to a new or compatible deployed instance using the observed cluster state. Subsequent requests use this

selected deployment. Aragog's request-time routing and Murakkab's periodic resource planning are therefore complementary to `FlowMaker`'s task-deployment-time workflow-variant and reuse selection.

Workflow orchestration and agent execution. Application frameworks such as LangChain, LangGraph, CrewAI, AutoGen, and MetaGPT provide abstractions for composing LLM calls, tools, agents, state, and control flow (Chase, 2022; LangChain, 2025; CrewAI, 2024; Wu et al., 2024; Hong et al., 2024). Their programming models range from explicitly structured workflows to conversational multi-agent coordination, but the application structure or agent logic is supplied by the developer.

Systems work provides complementary execution-time mechanisms. Parrot exposes dependencies among LLM calls through semantic variables to enable application-aware scheduling and context sharing (Lin et al., 2024a). Nalar uses futures and managed state to support policy-driven scheduling, routing, and resource management for dynamic agent programs (Laju et al., 2026), while AIOS provides services for agent scheduling, context, memory, storage, and tool management (Mei et al., 2025). KVFlow and Helium exploit workflow structure to improve cache management, prefetching, or scheduling (Pan et al., 2025; Wadlom et al., 2026). These systems improve the construction or execution of an application, whereas `FlowMaker` selects its deployable workflow variant and reuse plan under task-level SLOs and current cluster conditions.

LLM inference, profiling, and deployment. Inference runtimes such as vLLM, TensorRT-LLM, and DeepSpeed provide complementary mechanisms for efficiently executing a selected model, including KV-cache management, batching, optimized kernels, and distributed parallelism (Kwon et al., 2023; NVIDIA Corporation, 2024; Aminabadi et al., 2022). Profiling and performance-modeling systems reduce the cost of evaluating deployment choices. MaverIQ estimates latency and memory across parallelism configurations through lightweight profiling (Liakopoulos et al., 2025); Vidur simulates LLM inference and supports cost-effective deployment selection (Agrawal et al., 2024); NeuSight forecasts model latency across model–GPU combinations (Lee et al., 2025); and MAD-Max models distributed mappings and parallelization strategies (Hsia et al., 2025). AlpaServe jointly selects placement and model parallelism for a collection of models under bursty demand (Li et al., 2023).

`FlowMaker` builds on these execution and prediction mechanisms but operates at a different optimization scope. Rather than optimizing the deployment of a predetermined model or model collection, it selects an end-to-end workflow variant, including workflow structure and accuracy-dependent model choices, and accounts for reuse across tasks.

Serverless workflow resource management. Serverless workflow systems execute user-specified DAGs of functions under latency and cost objectives. Aquatope (Zhou et al., 2022) and Cypress (Bhasi et al., 2022) provision workflow stages under performance uncertainty; Orion (Mahgoub et al., 2022) sizes, bundles, and prewarms stages; FaaSFlow (Li et al., 2022) optimizes distributed workflow execution and data movement; and ALAP (Sinha et al., 2026) delays resource-allocation decisions to improve efficiency. These systems generally treat the workflow structure and function implementations as fixed and optimize how their resources are provisioned, placed, or scheduled.

Generative-AI workflow serving introduces additional decisions. Different models provide different output quality, each model admits multiple parallel configurations, and reusing an instance changes its load and capacity available to other tasks. LLM instances also have large, persistent GPU-memory footprints and may span multiple GPUs. Finally, the serving system must select among alternative user-approved workflows rather than receiving a single fixed DAG. Serverless resource-management mechanisms remain complementary, but their fixed-workflow formulations do not directly solve this joint selection problem. We nevertheless compare against the Bayesian-optimization search strategy used by serverless systems (Zhou et al., 2022; Bilal et al., 2023) in § 5.3.

Cold starts and model loading. A complementary line of work reduces the time required to initialize serverless functions or load model state. Prior systems mitigate cold starts and accelerate restoration (Du et al., 2020; Ustiugov et al., 2021; Ao et al., 2022), prewarm instances (Mahgoub et al., 2022; Shahrad et al., 2020), or optimize checkpoint loading and live migration for LLM serving (Fu et al., 2024). `FlowMaker` addresses the preceding decision: it determines which models and con-

figurations to deploy and whether an already deployed instance can be reused without exceeding its capacity. These techniques can accelerate any new deployment selected by `FlowMaker`.

C Workflow-Variant Space Characterization

We provide additional characterization of the three dimensions forming the workflow-variant space.

Workflows. A task can admit multiple candidate workflows that differ in their stages, dependencies, or execution order. Our analysis of approximately 1.6K tasks in AsyncHow (Lin et al., 2024b) finds that 42% have at most three candidate workflows, 18% have at least 20, and the median is five (Dai et al., 2025; Yang et al., 2024) (see Figure 7). This diversity expands the selection space but also creates opportunities to choose workflows better suited to the task's SLOs and the current cluster state.

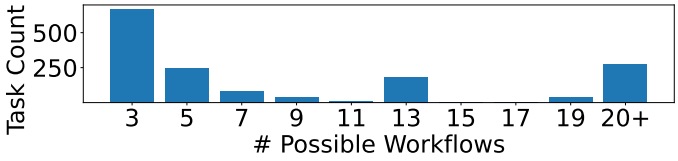

Figure 7: Each task in the AsyncHow dataset can be executed using multiple workflows, each with distinct characteristics. Those choices expand the search space.

Models. Each LLM stage can be served by models with different latency, GPU-memory, serving-cost, and proxy-accuracy characteristics. Figure 8 characterizes the 11 models used in our evaluation. Latency, memory, and cost are measured while serving a query with an output length of 100 tokens. Proxy accuracy is the model's average score across six Open LLM Leaderboard benchmarks (Fourrier et al., 2024), rather than a direct evaluation of generated outputs. The models occupy different points across these metrics, requiring selection to account for both user SLOs and resource requirements.

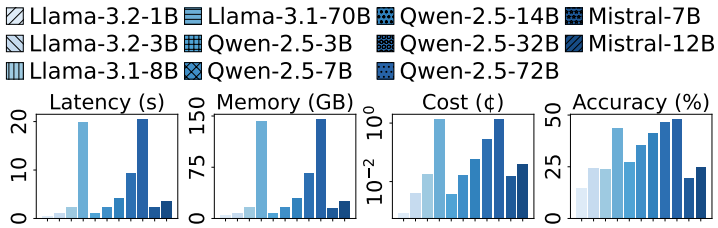

Figure 8: Latency, GPU-memory consumption, serving cost, and benchmark-derived proxy accuracy of the 11 evaluated models. Latency, memory, and cost are measured while serving a query with an output length of 100 tokens.

Deployment configurations. Even after selecting a model, its tensor- and pipeline-parallelism configuration can substantially change latency, memory requirements, capacity, and cost (Liakopoulos et al., 2025). On our eight-GPU testbed, Llama-3.1-8B supports 10 evaluated configurations, with up to 3.5× variation in latency and 3.6× variation in cost. Thus, selecting a model without considering its deployment configuration can eliminate otherwise feasible or less expensive workflow variants.

D Design Alternatives and their Limitations

Lookup table. A straightforward approach constructs a lookup table per task that records latency, throughput, and memory for all workflow variants, selecting the one that meets the objective. However, populating this table is expensive, even with state-of-the-art profilers, such as MaverIQ (Liakopoulos et al., 2025), NeuSight (Lee et al., 2025), or Vidur (Agrawal et al., 2024), which only

reduce per-model profiling cost. For a simple translate-and-summarize task (3 workflows, 11 models, 10 configurations), this requires evaluating over 24K variants and takes ~ 40 CPU-minutes with MaverIQ (Liakopoulos et al., 2025). The cost further increases with reuse, as the space expands to include previously deployed models, making this approach impractical in an online setting.

Traditional search algorithms. More sophisticated approaches avoid exhaustive enumeration. Bayesian optimization (BO) and evolutionary search are widely used (Močkus, 1975; Garnett, 2023; Zhou et al., 2022; Acun et al., 2026). BO builds a probabilistic surrogate (e.g., Gaussian processes) but incurs modeling overhead and scales poorly to high-dimensional, discrete spaces (Frazier, 2018; Papenmeier et al., 2025; Baptista & Poloczek, 2018). Evolutionary methods avoid explicit modeling but require many evaluations to converge (Eiben & Smith, 2015). Both require careful tuning of surrogates, acquisition functions, and hyperparameters, and incur high decision latency—tens of seconds to minutes—to identify a suitable variant (§ 5.3). This makes them unsuitable for our setting, where selection lies on the critical path.

Graph algorithms. Graph-based formulations (e.g., Steiner Forest (Agrawal et al., 1991; Goemans & Williamson, 1992; Gupta & Kumar, 2014)) assume a pre-constructed graph, ignoring the cost of building it. In generative AI workflows, the graph $\mathcal{G}$—capturing all workflow-variants—must be constructed on-the-fly, as subtasks and workflows are not known a priori. Even for a single task $\mathcal{T}$, without reusing deployed models, constructing $\mathcal{G}$ costs $O(|\mathcal{W}| \cdot (|\mathcal{M}| \cdot |\mathcal{C}|)^{|\mathcal{S}|})$, where $\mathcal{W}$, $\mathcal{S}$, $\mathcal{M}$, and $\mathcal{C}$ denote workflows, subtasks, models, and configurations. This makes runtime graph construction prohibitive and precludes classical traversal algorithms (BFS (Moore, 1959; Cormen et al., 2022), DFS (Tarjan, 1972; Cormen et al., 2022), Dijkstra (Dijkstra, 2022; Cormen et al., 2022), A* (Hart et al., 1968; Russell & Norvig, 2009)).

E Notation, SLO Relaxation & Complexity of FlowMaker's Algorithm

Notation. Table 1 summarizes the symbols used in the problem formulation (§ 3.2) and in Algorithm 1. Throughout, a *node* n_i denotes a deployed (model, configuration) instance together with its current serving load, so a workflow-variant $\mathcal{WV} = \langle \mathbf{n}, \mathbf{gpu} \rangle$ fixes one node per subtask and the GPUs each node occupies.

Table 1: Notation.

Symbol	Description	Symbol	Description
$\mathcal{T}$	Task (in natural language)	$\mathbf{n}$	Nodes $\{n_i\}$
$\mathcal{W}$	Set of workflows	n_{id}	Node (model, config)
$\mathcal{S}$	Set of subtasks	id	Node's ID
$\mathcal{M}$	Set of models	$\mathbf{gpu}$	GPU mapping
$\mathcal{C}$	Set of configs (TP, PP)	λ	Latency SLO
$\mathcal{D}$	Set of deployed LLMs	α	Accuracy SLO
$\mathcal{WV}$	Workflow-variant $(\mathbf{n}, \mathbf{gpu})$	ρ	Arrival rate

Progressive SLO relaxation. While users may specify both latency and accuracy SLOs, satisfying both may be infeasible (e.g., no model is both fast and accurate enough). We therefore relax constraints progressively, preserving user intent as much as possible. We first drop the accuracy constraint and select the highest-accuracy workflow variant that meets the latency SLO. If no variant satisfies the latency SLO, we select the lowest-latency variant; if multiple variants have the same latency, we choose the one with the highest accuracy. If only an accuracy SLO is specified and no variant satisfies it, `FlowMaker` selects the highest-accuracy variant. If no variant is feasible, `FlowMaker` reports that no deployable workflow exists under the given conditions.

Time complexity analysis. A brute-force approach enumerates all workflow-variants across all workflows, assigning each subtask a model, deployment configuration, and reuse option. For $|\mathcal{W}|$ workflows, this requires enumerating the Cartesian product across $|\mathcal{S}|$ subtasks, each with $|\mathcal{M}|$ model choices, $|\mathcal{C}|$ configuration options, and $|\mathcal{D}|$ reuse candidates. This yields a complexity of $O\left(|\mathcal{W}| \cdot (|\mathcal{M}| \cdot |\mathcal{C}| \cdot |\mathcal{D}|)^{|\mathcal{S}|}\right)$, which is exponential in $|\mathcal{S}|$.

FlowMaker makes this tractable by shrinking the base of the exponent through pruning: (i) pruning by accuracy reduces $|\mathcal{M}| \rightarrow |\mathcal{M}'|$, (ii) pruning by operational cost reduces $|\mathcal{D}| \rightarrow |\mathcal{D}'|$, and (iii) pruning by latency, capacity, and memory constraints reduces $|\mathcal{C}| \rightarrow |\mathcal{C}'|$. We also exit early, avoiding full enumeration of the remaining search space after pruning. Hence, the effective complexity becomes $O\left(|\mathcal{W}| \cdot (|\mathcal{M}'| \cdot |\mathcal{C}'| \cdot |\mathcal{D}'|)^{|\mathcal{S}|}\right)$, where $|\mathcal{M}'| \ll |\mathcal{M}|$, $|\mathcal{C}'| \ll |\mathcal{C}|$, and $|\mathcal{D}'| \ll |\mathcal{D}|$. This significantly reduces the base of the exponent. In § 5.3, we empirically demonstrate the time-efficiency of our algorithm.

F APIs & Implementation Details

APIs. Table 2 lists FlowMaker's three APIs. Operators call register_model to add a model to the catalog, optionally supplying a measured accuracy score; if omitted, FlowMaker falls back to the Open LLM Leaderboard proxy (see Appendix G). Users call deploy_task with the task in natural language and its latency and accuracy SLOs, after which FlowMaker generates candidate workflows, selects a workflow-variant, and deploys it. Expert users may instead pass a workflow_variant directly, pinning the workflow, models, and configurations; FlowMaker then honors those constraints while still reusing matching deployed instances. Once a task is deployed, inference_task submits requests against it, and the Request Orchestrator routes each through the task's selected stages.

Table 2: FlowMaker's APIs.

API	Parameters
register_model	model, [accuracy]
deploy_task	task, latSLO, accSLO | workflow_variant
inference_task	task, inputText

Prototype architecture. We implement FlowMaker in approximately 3K lines of Python atop TensorRT-LLM (NVIDIA Corporation, 2024). FlowMaker consists of a logically centralized Controller and a set of GPU workers. The Controller hosts the Workflow Generator, Workflow-Variant Selector, Profiler, Deployer, and Request Orchestrator. Each worker runs a lightweight Monitoring Daemon, and the components communicate through TCP sockets.

Profiling and metadata. When a model is registered, the Profiler uses MaverIQ's fingerprint-based profiling (Liakopoulos et al., 2025) to estimate its latency and memory consumption across supported tensor- and pipeline-parallelism configurations. Profiles are approximately 30KB and are stored as CSV files, while model weights are stored in the Model Repository. Profiles are computed once and reused across task deployments.

Workflow generation and selection. The Workflow Generator runs as a CPU process. It first queries the Workflow Database, an on-disk JSON catalog of previously approved decompositions. On a cache miss, it prompts OpenAI's gpt-oss-120b (OpenAI, 2025) using chain-of-thought prompting (Wei et al., 2022) to generate candidate workflows, which are presented to the user for approval. Approved workflows are added to the database.

The Workflow-Variant Selector also runs as a CPU process. It maintains an in-memory view of deployed model instances, their load, and available GPU memory. At task deployment, it combines this state with stored model profiles, user-approved workflows, and the task's SLOs to execute Algorithm 1.

Deployment and request execution. For each model instance that cannot be served through reuse, the Deployer launches a TensorRT-LLM process on the selected GPUs using the chosen parallelism configuration. Each process is assigned an IP/port endpoint stored in an in-memory routing directory. Additional mappings associate tasks with workflow stages and stages with model processes.

The Request Orchestrator serves requests in first-come-first-served order. It looks up the endpoint for the first stage, attaches 100–1000 bytes of routing metadata, and forwards the request. This metadata identifies subsequent stages and their endpoints, allowing intermediate outputs to flow directly between model processes over TCP and keeping the Controller off the inter-stage data path.

Cluster monitoring. Each Monitoring Daemon contains two CPU components. The Memory Monitor samples per-GPU memory availability using NVML (NVIDIA Corporation, 2025). The Load Monitor computes each deployed model's load over a two-minute sliding window. Both update the Controller's global in-memory state every five seconds. The selector uses the latest snapshot when evaluating reuse and placement.

G Experimental Details

Trace construction. We derive request arrivals from the Azure code and conversation traces (Patel et al., 2024). We construct four relative offered-load levels by discretizing each trace into intervals, aggregating adjacent intervals, and applying the factors in Table 3. These percentages denote relative levels within each trace, rather than absolute cluster utilization.

Table 3: Trace-scaling factors used for each offered-load level.

Trace	25%	50%	75%	100%
Code	0.100	0.133	0.167	0.200
Conversation	0.050	0.067	0.083	0.100

The resulting workloads contain 212–484 requests. Because the traces provide timestamps and input/output token counts but no semantic inputs, we pair their entries with examples from CNN/DailyMail (See et al., 2017; Hermann et al., 2015), HumanEval (Chen et al., 2021a), Yelp Reviews and AG News (Zhang et al., 2015), DialogSum (Chen et al., 2021b), and WritingPrompts (Fan et al., 2018).

Tasks. We evaluate seven representative generative-AI tasks (Data and Beyond, 2025; Khurana et al., 2023; AIMonk Team, 2026). Each task has three candidate workflows containing one to three stages. The tasks span six categories: (1) cross-lingual understanding, which translates and summarizes text across languages; (2) text processing, which summarizes and simplifies text; (3) analysis and reporting, which extracts key phrases and sentiment to synthesize a report; (4) code documentation; (5) opinion mining over product reviews; and (6) document classification and tagging.

Model catalog. Our catalog contains 11 open-weight models from three families: Llama-3.2-1B, 3B and Llama-3.1-8B, 70B (Grattafiori et al., 2024); Qwen2.5-3B, 7B, 14B, 32B, 72B (Qwen et al., 2025); and Mistral-7B, 12B (Jiang et al., 2023). We treat Llama-3.1-70B and Qwen2.5-72B as fallback models and the remaining nine as the primary catalog. We restrict Llama-3.2-1B/3B and Qwen2.5-3B to stages that perform a single operation because these models do not reliably execute compound operations such as joint translation and summarization.

We construct each stage's candidate-model set using a score that combines model quality from the Open LLM Leaderboard (Fourrier et al., 2024), a logarithmic parameter-count penalty favoring deployment efficiency, and a model-family diversity bonus. This procedure provides multiple candidate models per stage when possible.

LangChain. LangChain (Chase, 2022) represents static workflow execution. We port its execution logic to TensorRT-LLM and provide each task's default workflow, the smallest model satisfying its proxy-accuracy SLO, and $\text{TP} = 1$, $\text{PP} = 1$. These choices remain fixed as cluster conditions change.

RouteLLM. RouteLLM (Ong et al., 2025) performs request-level routing over a fixed workflow. For every task, we pre-deploy Qwen-2.5-14B and Qwen-2.5-7B as its strong/weak pair and use RouteLLM's pretrained BERT router. A request uses the routed model for every LLM stage in its workflow. We evaluate three routing thresholds selected to favor maximum quality, minimum cost, or a balance between them. We provision sufficient replicas to serve each model's offered load without queuing; all replicas are included in the reported cost.

Murakkab. Murakkab (Chaudhry et al., 2026) uses profile-guided optimization to select workflow parameters, models, deployment configurations, and resource allocations. In our evaluated configuration, we supply the same default workflow used by the other fixed-workflow configurations and execute Murakkab's selected models and configurations using TensorRT-LLM. Because its implementation is unavailable, we reconstruct the optimizer described in the paper. Our comparison

therefore characterizes this evaluated configuration and does not imply that Murakkab is inherently restricted to fixed workflows.

`FlowMakerNoReuse`. `FlowMakerNoReuse` uses the same workflow-variant selector as `FlowMaker`, including selection across workflows, models, and deployment configurations, but deploys separate model instances for each task. Comparing `FlowMaker` with `FlowMakerNoReuse` isolates the effect of cross-task reuse.

Brute force and Aquatope-BO. For the selector study, brute-force search enumerates the complete candidate space using cached profiles and returns the minimum-cost feasible variant under Equation 1. Aquatope-BO adapts the Bayesian-optimization strategy used for serverless workflow tuning (Zhou et al., 2022) to our workflow-variant space. We implement it using Optuna's TPE sampler (Optuna Team, 2024), a budget of 200 sequential trials, and penalties for SLO violations. This is an adaptation of Aquatope's search strategy rather than an execution of the complete Aquatope system.

Default workflows. For configurations receiving a supplied workflow, we use the simplest candidate for each task, composed of stages that each perform one operation (e.g., translate $\rightarrow$ summarize). LangChain additionally receives fixed model assignments and TP $= 1$, PP $= 1$.

Latency SLOs. For each task, we measure an isolated reference latency using its simplest workflow, largest compatible models, and no model parallelism. We construct four tiers: $2\times$ the reference latency (STRICT), $3\times$ (MODERATE), $4\times$ (LENIENT), and $5\times$ (RELAXED). Unless otherwise stated, T2 and T7 use STRICT SLOs, T3 uses MODERATE, T1 and T5 use LENIENT, and T4 and T6 use RELAXED SLOs.

Proxy-accuracy SLOs. Let $b_1(m), \ldots, b_6(m)$ denote model m's scores on the six Open LLM Leaderboard benchmarks. We define

$$q(m) = \frac{1}{6}\sum_{j=1}^{6} b_j(m), \qquad q(\mathcal{WV}) = \min_{m \in \mathcal{WV}} q(m) \tag{2}$$

Following Murakkab (Chaudhry et al., 2026), the BEST, GOOD, and BASIC SLOs are the 95th, 80th, and 50th percentiles of the achievable workflow-variant proxy scores for each task. Unless otherwise stated, T2 and T7 use BASIC; T1, T3, and T5 use GOOD; and T4 and T6 use BEST.

We do not independently evaluate the semantic quality of generated outputs. A request instead inherits its selected workflow variant's proxy score and satisfies the proxy-accuracy SLO when $q(\mathcal{WV}) \geq \alpha$.

Testbed. Experiments run on one server with eight NVIDIA RTX A6000 GPUs, each with 48GB of memory. GPU pairs are connected through NVLink, while communication between pairs uses PCIe. The server has a 64-core AMD EPYC 7763 CPU and runs Ubuntu 22.04, NVIDIA driver 575.51.03, and CUDA 12.9. All LLM inference uses TensorRT-LLM with its optimized kernels, in-flight batching, and tensor- and pipeline-parallel execution.

Serving cost. We measure cluster-wide GPU-memory time and convert it to dollars:

$$\text{Cost} = r_{\text{GB-s}} \int_0^T \sum_j M_j(t)\, dt \tag{3}$$

where $M_j(t)$ is the GPU memory occupied by model instance j at time t, and $r_{\text{GB-s}}$ is derived from the hardware rental price (Amazon Web Services, 2026; Liakopoulos et al., 2025; Eismann et al., 2020). This trace-level operational metric differs from the incremental cost proxy used by the selector in Equation 1.

Goodput. For an experiment lasting T minutes, cluster goodput is

$$\text{Goodput} = \frac{|\{req : L_{req} \leq \lambda_{req} \ \wedge \ q(\mathcal{WV}_{req}) \geq \alpha_{req}\}|}{T}. \tag{4}$$

where L_{req} is request req's end-to-end latency and $\mathcal{WV}_{req}$ is the workflow variant used to serve it. We report goodput in requests per minute.

H Additional Empirical Results

H.1 Detailed comparison of FlowMaker's performance against baselines

Figure 5 compares the systems under the same seven-task workload and task-specific latency and proxy-accuracy SLOs. We evaluate both bursty code arrivals and sustained conversation arrivals across four offered-load levels.

`FlowMaker` vs. Murakkab. Murakkab is our closest end-to-end baseline: it selects models and deployment configurations for a user-specified workflow and makes reuse decisions offline. At lower loads, queues remain shallow, so both systems achieve similar goodput. Nevertheless, `FlowMaker` improves goodput by up to $1.46\times$ ($1.21\times$ on average) and reduces cost by 1.84–$3.15\times$ ($2.23\times$ on average) by considering alternative workflows. For example, for the analysis-and-reporting task, Murakkab uses a three-stage workflow, whereas `FlowMaker` selects a two-stage workflow with lower latency and cost. The difference grows at higher loads. `FlowMaker` achieves 1.45–$10.14\times$ Murakkab's goodput ($4.73\times$ on average) while reducing cost by 1.56–$2.77\times$ ($1.95\times$ on average). In these experiments, Murakkab's workflow and reuse choices increase its average latency by $4.65\times$. For example, when the seventh task arrives, `FlowMaker` reuses a Qwen2.5-3B instance configured with TP $= 2$, PP $= 1$, whose existing load is 16 requests/minute against a capacity of 34. Murakkab instead reuses a Qwen2.5-14B instance configured with TP $= 1$, PP $= 1$ that is already at capacity, causing queues to build and reducing goodput.

`FlowMaker` vs. RouteLLM. RouteLLM fixes each task's workflow and pre-deploys a strong/weak model pair; its routing threshold therefore exposes a quality–cost tradeoff. Its maximum-quality setting routes nearly all requests through Qwen2.5-14B, such that all requests satisfy their proxy-accuracy SLOs. However, it reserves 206–235GB of GPU memory across traces and loads, incurring 2.06–$3.84\times$ `FlowMaker`'s cost ($2.95\times$ on average). Its goodput differs from `FlowMaker`'s by at most 7%, except at the highest code-trace load. At that point, 14% of `FlowMaker`'s requests violate their latency SLOs, and RouteLLM achieves $1.37\times$ higher goodput (17.02 vs. 12.41 RPM) at $2.06\times$ the cost. The minimum-cost setting predominantly routes requests through Qwen2.5-7B. Its cost ranges from 0.87–$1.23\times$ that of `FlowMaker`, but only 34.9–36.8% of requests are assigned variants satisfying their proxy-accuracy SLOs, compared with 100% for `FlowMaker`. Consequently, `FlowMaker` achieves 1.97–$2.82\times$ higher goodput ($2.62\times$ on average). RouteLLM's balanced setting lies between these extremes: 62.6–65.6% of requests satisfy their proxy-accuracy SLOs, while `FlowMaker` achieves 1.31–$1.58\times$ higher goodput ($1.52\times$ on average). The balanced setting also incurs 1.88–$2.94\times$ `FlowMaker`'s cost ($2.35\times$ on average).

RouteLLM attains lower request latency in all three settings because we provision sufficient replicas to eliminate queuing. This replication accounts for the higher cost of its quality-oriented settings.

`FlowMaker` vs. LangChain. Across traces and loads, `FlowMaker` achieves 1.13–$12.84\times$ LangChain's goodput ($5.19\times$ on average) and reduces cost by 2.22–$3.99\times$ ($2.93\times$ on average). LangChain fixes workflows, models, and configurations and does not reuse model instances, so it cannot adapt its resource footprint or parallelism configuration to the cluster state. For example, LangChain deploys 15 instances consuming 290GB to serve six tasks. The remaining task requires three 29GB Qwen-2.5-14B instances, but memory fragmentation leaves no GPU with sufficient free memory. `FlowMaker` instead serves all seven tasks using three to five model instances by selecting alternative workflows and configurations and reusing compatible deployments. Static placement also increases contention: under the code trace at 50% load, LangChain colocates four model instances on one GPU, producing an average latency of 426s, compared with 38s for `FlowMaker`.

H.2 Scaling Under Fixed Cluster Capacity

We next evaluate how the systems scale as tasks are added without increasing cluster capacity. We create 14 tasks by duplicating the seven-task workload and use the Azure conversation trace at 25% load, with one task deployed every two minutes. All models are deployed within 4×NVIDIA RTX A6000 GPUs. Figure 9 reports the number of tasks admitted, serving cost, and aggregate SLO-compliant goodput.

LangChain and `FlowMakerNoReuse` exhaust GPU memory after admitting four and seven tasks, respectively, because they deploy separate instances per task. `FlowMaker` and Murakkab admit all 14 tasks through reuse. However, `FlowMaker` achieves $1.5\times$ Murakkab's goodput by jointly con-

sidering alternative workflows and adapting reuse to the current load. Murakkab concentrates traffic from the 14 tasks on two shared model instances, causing contention as tasks arrive. FlowMaker distributes the traffic across four shared instances, preserving more serving capacity.

RouteLLM also admits all 14 tasks, since we grant it an additional advantage: it pre-deploys replicas sized to the offered load it will receive. With this headroom, its maximum-quality setting achieves $1.40\times$ FlowMaker's goodput at comparable cost ($0.97\times$), as replication absorbs the queuing that FlowMaker incurs while deploying variants on demand. Its remaining settings trade that advantage away: the balanced and minimum-cost settings reduce cost by 1.16–$1.90\times$, but serve only 39% and 38% of the requests within their proxy-accuracy SLOs, leaving goodput 1.51–$1.68\times$ below FlowMaker's. Fixed replication can therefore achieve higher goodput when the load is known ahead of time, whereas FlowMaker meets both latency and proxy-accuracy SLOs while deciding deployments online.

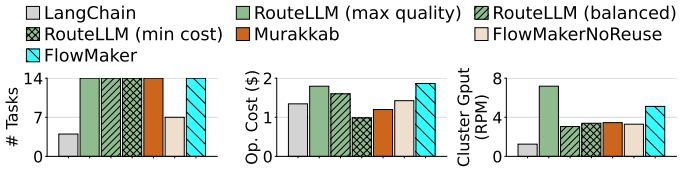

Figure 9: Tasks admitted (left), serving cost (middle), and SLO-compliant goodput (right) as 14 tasks arrive under fixed cluster capacity. FlowMaker and Murakkab admit all tasks through reuse, with FlowMaker achieving $1.5\times$ Murakkab's goodput. RouteLLM attains higher goodput in its maximum-quality setting through replication.

H.3 Sensitivity of FlowMaker to Latency SLOs

In our end-to-end analysis, we evaluate FlowMaker under heterogeneous latency SLOs (loose, moderate, lenient, and strict). Here, we consider extreme settings where all tasks have either loose or strict SLOs, using the code trace at 100% load with the same accuracy SLOs as in the end-to-end experiment. Figure 10 shows that all systems, except RouteLLM, experience a substantial drop in goodput under strict SLOs—approximately halving relative to loose SLOs—due to the difficulty of meeting tight constraints under limited resources. However, FlowMaker remains the most robust, improving goodput by 1.1–$12.4\times$ and reducing cost by 1.3–$3.8\times$ across those baselines in both settings. RouteLLM is the exception: replication keeps its latency low, so its goodput is identical under both SLO settings. Under strict SLOs, this exceeds FlowMaker by $2.29\times$ and $1.08\times$ for the max-quality and balanced variants, at $1.94\times$ and $1.77\times$ the cost, while the min-cost variant reduces cost by $1.13\times$ and goodput by $1.18\times$ compared to FlowMaker.

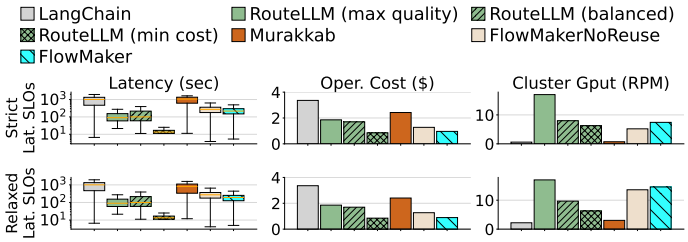

Figure 10: Per-request latency (left), serving cost (middle), and SLO-compliant goodput (right) under strict (top) and relaxed (bottom) latency SLOs, on the code trace at 100% load. FlowMaker remains the most cost-efficient in both settings.

H.4 FlowMaker Incurs Minimal Overheads

Section 5.3 shows our search algorithm efficiently selects workflow-variants; here, we quantify model loading overheads. Loading takes 1–5s (depending on size), amounting to only 0.4–2.5% of SLO time. These costs are small relative to FlowMaker's gains from on-the-fly reuse ($1.13\times$ goodput, $1.42\times$ cost reduction), making the tradeoff worthwhile. Moreover, these overheads can be further reduced using model-loading techniques from prior work (Stojkovic et al., 2025; Fu et al., 2024; InferX AI, 2026).