Agentic AI Pipeline Optimization for IT Operations

Linshuo Li **Kexin Wang** **Moxi Yuan** **Shouyi Li**
COMSW6998 High Performance Machine Learning, Columbia University
{ll3815, kw3164, my2833, sl5632 }@columbia.edu

1 Introduction

Agentic AI systems have recently emerged as promising tools for automating and augmenting complex IT operations. By combining large language models with planning, tool use, and execution capabilities, these systems can assist in Site Reliability Engineering (SRE), security and compliance operations (CISO), Financial Operations (FinOps), and broader DevOps workflows. Such agents are increasingly capable of interpreting operational signals, diagnosing failures, reasoning about system state, and proposing or executing corrective actions. As organizations consider deploying agentic AI in production environments, understanding not only what these systems can do, but how to evaluate, compare, and optimize them, has become a central challenge.

Despite rapid progress in agent architectures and supporting techniques, the evaluation landscape for agentic AI in IT operations remains underdeveloped in several key respects. Existing efforts provide valuable task coverage but differ substantially in evaluation criteria, reported metrics, and assumptions about operational context. In practice, evaluations often emphasize task-specific correctness while providing limited insight into operational performance dimensions such as latency, cost efficiency, safety, and robustness. This fragmentation makes it difficult to compare agent capabilities, assess system-level tradeoffs, or reason about how improvements in agent design translate to real-world operational impact.

Recent benchmarks such as ITBench and DevOps-Gym illustrate both progress and fragmentation in this space. ITBench evaluates agents on realistic IT operations scenarios across SRE, security, and cost management domains using operational artifacts and end-to-end task formulations, while DevOps-Gym focuses on software delivery workflows with real execution environments and tool interactions. Although these benchmarks target different operational settings, they adopt distinct task designs and evaluation criteria, limiting direct comparison across domains. Importantly, such diversity is not inherently problematic: different operational domains impose legitimately different constraints and success criteria. The challenge lies in the absence of shared evaluation dimensions that enable principled comparison while preserving domain-specific nuance.

Beyond benchmarking, a growing body of work explores techniques for building and optimizing agentic systems for IT operations, including architectural choices, tool orchestration strategies, inference-time optimizations, and mechanisms for improving robustness and efficiency. However, these techniques are often evaluated in isolation or under narrowly defined task settings, making it unclear how they interact with operational metrics or generalize across domains. As a result, there is a gap between advances in agent design and the evaluation frameworks needed to assess their practical effectiveness.

The goal of this survey is to provide a structured overview of agentic AI for IT operations through the lens of evaluation. We review existing benchmarks and analyze their strengths and limitations, survey techniques used to construct and optimize agentic systems, and examine empirical case studies that illustrate how these techniques perform under realistic operational constraints. Rather than advocating for a single unified benchmark, we argue for interoperable evaluation frameworks that enable comparison along shared operational dimensions while respecting domain-specific requirements.

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

Overall, this survey aims to bridge the gap between the rapidly advancing capabilities of agentic AI systems and the practical needs of enterprise IT operations. By connecting benchmarks, techniques, and case studies within a common evaluative framework, we seek to provide guidance for both future research and responsible real-world deployment.

2 Benchmarks for Agentic IT-Operations

2.1 Existing Benchmarks for Agentic IT Operations

Although agentic AI systems are rapidly gaining adoption in IT operations, benchmarking frameworks for this domain remain limited. In this subsection, we review existing benchmarks that are relevant to IT operations tasks, specifically ITBench and DevOps-Gym, and analyze their strengths and limitations in the context of operational evaluation. We also briefly discuss adjacent benchmarks that focus on tool-use or software engineering tasks to illustrate the broader landscape.

2.1.1 ITBench

ITBench is a recently proposed benchmark designed to evaluate language model–based agents on a range of IT operations tasks spanning domains such as Site Reliability Engineering (SRE), security and compliance operations (CISO), and cloud cost optimization (FinOps). The benchmark grounds its scenarios in realistic operational artifacts, including logs, alerts, configuration files, and policy documents, and evaluates an agent's ability to reason about and resolve domain specific operational problems.

From a metrics perspective, ITBench reports task-level success and qualitative reasoning outcomes but omits several operationally critical dimensions. In particular, metrics such as latency, throughput, and execution cost are not captured. Additionally, performance is reported separately across domains, without a unified scoring mechanism that enables holistic comparison of agent capabilities across different IT functions.

This make ITBench a valuable foundation for evaluating reasoning in IT operations, while simultaneously underscoring the need for unified evaluation metrics that account for both task correctness and system-level operational performance.

2.1.2 DevOps-Gym

DevOps-Gym evaluates agent performance on software delivery workflows, including building codebases, running tests, and debugging CI/CD pipelines using real repositories and build tools. Although it focuses on software engineering tasks rather than IT operations as defined in this survey, we include DevOps-Gym as a methodological reference point.

Specifically, DevOps-Gym demonstrates an agent-centric benchmarking architecture involving multi-step tool-driven interactions, dynamic execution environments, and semantic outcome evaluation. These architectural elements are directly transferable to IT operations domains such as SRE, security, and cost management.

2.1.3 Summary

Existing benchmarks for IT operations provide valuable coverage of agent reasoning and tool interaction across individual domains, but they lack a unified evaluation framework that captures the full spectrum of operational requirements. In particular, current benchmarks do not incorporate key operational performance metrics such as latency and throughput, business-oriented metrics such as cost efficiency, or risk and governance considerations such as safety and policy compliance. Moreover, methodological gaps remain in the form of domain-specific scoring schemes that prevent consistent cross-domain comparison of agent capabilities. These limitations motivate the need forunified metrics to support future research in agentic AI for IT operations.

2.2 Evaluation Dimensions and the Need for Unified Metrics

A central challenge in benchmarking agentic AI for IT operations is the lack of a unified evaluation framework. Existing benchmarks—including ITBench, DevOps-Gym, and tool-use benchmarks in

related domains—primarily assess task-specific correctness. While correctness is essential, real-world IT operations require agents to satisfy a broader set of operational and performance criteria. This subsection provides an overview of the key evaluation dimensions that matter in IT operations and motivates the need for a unified metrics framework.

2.2.1 Operational Evaluation Dimensions

Based on industry practices in SRE, CISO, FinOps, and general DevOps workflows, several dimensions recur as fundamental to agent performance:

- **Task Efficacy (Accuracy and Usefulness).** Measures whether the agent ultimately produces a correct and practically useful solution to the operational problem. This includes accurate diagnoses, correct recommendations, and alignment with domain expectations.
- **Autonomy.** Evaluates the agent's ability to operate without human intervention. Metrics include the number of required hints, escalations, or supervisor queries. One practical design is to provide a limited set of optional *hint tools* or *clarification tools* that the agent may invoke when it cannot progress. Each invocation is automatically counted as a reduction in autonomy, enabling reproducible measurement without requiring human evaluators. Similarly, *escalations* can be defined as explicit agent actions, such as requesting human approval or signaling inability to complete a task and treated as observable events recorded by the evaluation environment. Supervisor queries or clarification requests thus become standardized tool calls that the benchmark can measure directly.
- **Efficiency (Latency, Throughput, and Cost).** Operational environments require timely responses, especially in incident handling or security contexts where delays may violate service-level objectives. Efficiency includes: (i) latency (time to generate a response or complete a task), (ii) throughput (capacity to handle multiple tasks), and (iii) cost efficiency (monetary cost of running the model or agent over sustained periods).
- **Safety and Compliance.** IT operations often involve sensitive systems governed by security, compliance, and access-control policies. Unsafe recommendations, policy violations, or actions that would introduce operational risk must be penalized.
- **Adaptivity.** Production systems evolve over time. Agents must handle variations in logs, metrics schemas, alert types, and cloud environment changes. Adaptivity measures robustness under such variations and the ability to generalize across unseen tasks or slightly altered conditions.

2.2.2 Motivation for Unified Metrics

Current benchmarks for agentic IT operations evaluate correctness withinindividual domains, but they do so using heterogeneous and incompatible scoring rules. For example, ITBench evaluates an SRE diagnosis task based on whether the agent names the correct incident class, whereas CISO tasks are scored on whether the agent identifies specific indicators of compromise. FinOps tasks, in contrast, evaluate whether the agent recommends the correct optimization action or identifies a cost anomaly. Because each domain defines success differently, scores from these tasks cannot be meaningfully compared. This fragmentation makes it difficult to answer basic questions such as: "Which agent performs best overall?".

These limitations are particularly salient in scenarios where operational domains interact. Consider a cloud cost anomaly that ultimately results from a security breach. Under existing domain specific scoring rules, an agent might receive full credit in FinOps for correctly identifying the cost spike, even if it entirely misses the underlying security cause. This illustrates a structural issue: current benchmarks measure domain success, but not the broader operational competencies that IT teams care about, such as the quality of the agent's reasoning, the efficiency with which it reaches conclusions, or its robustness under input variation. Addressing such issues requires more than domain-specific correctness; it requires evaluation along shared dimensions that reflect operational priorities.

The goal of unified metrics is not to replace domain-specific scoring, nor to automatically infer cross-domain causal chains. Rather, unified metrics provide a common evaluative language that allows heterogeneous tasks to be compared on dimensions that matter across IT operations, such as task efficacy, operational efficiency, and adaptivity. This common structure enables more principled assessment of agent performance and supports analysis of system-level optimizations.

2.2.3 Proof-of-Concept Unified Metrics

To illustrate how unified metrics can be constructed across heterogeneous IT operations tasks, we present a simple proof-of-concept framework that standardizes evaluation along three core dimensions: *Task Efficacy*, *Efficiency*, and *Adaptivity*. By decomposing each dimension into interpretable subcomponents — such as latency, memory usage, throughput under concurrent load, and the cost impact of incorrect actions — the framework aligns closely with the types of system-level optimizations used in practice (e.g., quantization, batching, speculative decoding).

Task Efficacy. Task efficacy represents the operational quality of an agent's output and integrates two dimensions: *Accuracy*, which measures technical correctness, and *Usefulness*, which measures operational appropriateness and actionability.

Accuracy. Accuracy evaluates the correctness of the agent's reasoning steps or diagnostic conclusions. Because correctness in IT operations is multi-dimensional (e.g., identifying symptoms, isolating root causes, selecting viable remediations), we define accuracy as a weighted combination of subcomponents:

$$\text{Accuracy} = w_{\text{diag}} \cdot \text{DiagCorrect} + w_{\text{rca}} \cdot \text{RootCauseCorrect} + w_{\text{rem}} \cdot \text{RemediationCorrect},$$

where each term is scored independently on a continuous scale from 0 to 1. This formulation captures nuanced cases such as a correct diagnosis but an incorrect remediation plan, or a correct root cause but incomplete analysis.

Each subcomponent corresponds to a distinct aspect of operational reasoning:

- **DiagCorrect** measures whether the agent correctly interprets the observed symptoms (e.g., alerts, logs, metric spikes) and identifies the relevant signals without being misled by noise. This captures the quality of the agent's initial diagnostic reasoning.
- **RootCauseCorrect** evaluates whether the agent accurately isolates the underlying cause of the issue rather than only describing surface symptoms.
- **RemediationCorrect** measures whether the agent proposes a technically valid remediation that would resolve the problem without introducing additional risk. This component evaluates whether the agent's final recommendation is not only technically valid but also safe and operationally appropriate, distinguishing correct reasoning that results in harmful or impractical actions from reasoning that produces actionable solutions.

The weights $w_{\text{diag}}, w_{\text{rca}}, w_{\text{rem}}$ control the relative importance of these components and allow the metric to adapt to different operational contexts. For example, an SRE incident-management task may assign greater weight to root-cause identification (w_{rca}), whereas a CISO triage task may place higher emphasis on safe remediation (w_{rem}). By normalizing the weights such that

$$w_{\text{diag}} + w_{\text{rca}} + w_{\text{rem}} = 1,$$

the final accuracy score remains in the interval $[0, 1]$ while preserving task-specific flexibility.

Usefulness. Usefulness captures whether the agent's output is operationally actionable and aligns with domain constraints, policies, and risk considerations. We compute usefulness as:

$$\text{Usefulness} = 1 - \text{Penalty}_{\text{operational}},$$

where $\text{Penalty}_{\text{operational}}$ captures violations of security policy, unsafe recommendations, impractical remediations, or outputs that fail to reduce operational risk. This dimension distinguishes between technically correct answers that are helpful in practice and those that are not.

Integration. Finally, task efficacy combines these components:

$$\text{TaskEfficacy} = \lambda \cdot \text{Accuracy} + (1 - \lambda) \cdot \text{Usefulness},$$

where λ controls the relative weighting of correctness and operational appropriateness. This integrated metric reflects how real-world IT operations evaluate agent performance: an agent must not only reason correctly but also produce outputs that are safe and actionable.

Efficiency. Efficiency captures how effectively an agent uses computational, economic, and operational resources while completing a task. In IT operations, efficiency is not limited to low latency or small token usage; it also includes resource footprint, concurrency behavior, infrastructure impact, and the cost of mistaken or suboptimal actions. To reflect these considerations, we decompose Efficiency into three components: *Computational Efficiency*, *Economic Efficiency*, and *Operational Efficiency*.

Computational Efficiency. Computational efficiency measures how well the agent uses computing resources during inference. It includes latency, memory usage, and throughput under concurrent load:

$$\text{CompEff} = w_{\text{lat}} \cdot \text{LatencyScore} \ + \ w_{\text{mem}} \cdot \text{MemoryScore} \ + \ w_{\text{thr}} \cdot \text{ThroughputScore}.$$

Each term in this formulation captures a distinct aspect of computational performance. *LatencyScore* measures how quickly the agent produces results under operational constraints, reflecting responsiveness during incidents or security events. *MemoryScore* quantifies how efficiently the agent uses available GPU or CPU memory, an important factor for deployability and for supporting high concurrency without out-of-memory failures. *ThroughputScore* evaluates how well the agent maintains performance when handling multiple simultaneous requests.

The weights w_{lat}, w_{mem}, and w_{thr} allow the benchmark designer to assign relative importance to these components depending on operational priorities. For example, production SRE workloads may emphasize low latency, whereas FinOps batch analyses may prioritize high throughput. Consistent with the weighting scheme introduced in the Task Efficacy metric, we normalize these weights such that they sum to one.

Economic Efficiency. Economic efficiency quantifies the monetary implications of using the agent:

$$\text{EconEff} = w_{\text{inf}} \cdot \text{InferenceCost} + w_{\text{infra}} \cdot \text{InfraCost} + w_{\text{err}} \cdot \text{ErrorCost}.$$

Each term in the Economic Efficiency formulation captures a different dimension of monetary cost in IT operations. *InferenceCost* represents the direct cost of running the model, including token usage and model invocation counts. *InfraCost* captures the infrastructure expenses resulting from agent initiated actions, such as provisioning new compute resources or modifying autoscaling settings. *ErrorCost* reflects the operational or financial penalties associated with incorrect or suboptimal actions, such as unnecessarily scaling up cloud resources or misclassifying a security incident.

The weights w_{inf}, w_{infra}, and w_{err} enable the benchmark designer to adjust the relative importance of these components based on operational constraints. For cost-sensitive FinOps environments, higher weight may be placed on InferenceCost, whereas security-focused workloads may emphasize ErrorCost due to the higher risk of false positives or missed threats. As in the Computational Efficiency metrics, we normalize these weights so that they sum to one.

Operational Efficiency. Operational efficiency measures the human effort required to supervise, verify, or correct the agent's behavior. Agents that require frequent clarifications, manual validation, or supervisory input create hidden operational costs:

$$\text{OpEff} = 1 - \frac{\text{HumanInterventionCount}}{\text{MaxAllowed}}.$$

HumanInterventionCount captures how often a human must step in to clarify instructions, provide missing context, or prevent an unsafe action. Higher intervention frequency indicates that the agent places additional operational burden on the team. *MaxAllowed* represents the upper bound on acceptable interventions for a task, enabling the score to be normalized between 0 and 1. An Operational Efficiency score of 1 corresponds to fully autonomous behavior, while lower values indicate increasing levels of required oversight.

Integration. The overall Efficiency score is a weighted combination of these three components:

$$\text{Efficiency} = w_{\text{comp}} \cdot \text{CompEff} + w_{\text{econ}} \cdot \text{EconEff} + w_{\text{op}} \cdot \text{OpEff}.$$

This weighted formulation aggregates the three components of Efficiency into a single unified score. w_{comp}, w_{econ}, and w_{op} control the relative contribution of Computational Efficiency, Economic Efficiency, and Operational Efficiency, respectively. These weights are normalized to sum to one, ensuring that the overall Efficiency score remains within $[0, 1]$ and that each component reflects its intended share of importance. The choice of weights depends on the operational context. For example,

latency-sensitive SRE incident response may assign higher weight to Computational Efficiency, cost-governed FinOps workflows may emphasize Economic Efficiency, and settings with limited on-call staffing or high supervision burden may prioritize Operational Efficiency.

Adaptivity. Adaptivity measures the agent's robustness to variations in operational inputs. In real IT environments, such variations occur frequently due to software updates, deployment changes, or evolving cloud services. An agent that performs well only on fixed, static scenarios but fails when confronted with these perturbations is not production-ready.

We propose an *Adaptivity Score* based on the agent's ability to maintain performance under controlled input perturbations. Let D_0 denote the original task inputs (e.g., logs, metrics, configuration files), and let $\{D_1, D_2, \ldots, D_k\}$ denote perturbed variants that reflect realistic operational drift, such as schema changes or reordered fields. We define:

$$\text{Adaptivity} = \frac{1}{k} \sum_{i=1}^{k} \frac{\text{TaskEfficacy}(D_i)}{\text{TaskEfficacy}(D_0)}.$$

This formulation evaluates whether the agent preserves its reasoning quality when encountering new or shifted input formats. A value of 1 indicates perfect robustness: the agent is unaffected by such changes. Lower values reveal sensitivity to input drift or rigid reliance on memorized patterns rather than generalizable reasoning.

Discussion. These simple formulations demonstrate how heterogeneous tasks can be mapped into a shared metric space while preserving core operational concerns. Although simplified, the framework shows that unified metrics for Task Efficacy, Efficiency, and Adaptivity are feasible and can serve as building blocks for more comprehensive, multi-dimensional evaluation schemes in future benchmarks.

2.3 Future Directions

The analysis of existing benchmarks and the proof-of-concept unified metrics presented in this section point toward a clear research agenda for agentic AI in IT operations. While current benchmarks provide valuable coverage of agent reasoning and tool use within individual domains, they lack a unified evaluation framework that spans the full range of operational requirements across SRE, security, and cost management.

The proof-of-concept framework presented in this section demonstrates that heterogeneous IT operations tasks can be mapped into a shared metric space without discarding domain-specific nuance. Future research should investigate how such unified metrics can be calibrated, validated, and aggregated at scale, as well as how they interact with system-level optimization techniques such as batching, quantization, and adaptive inference strategies. Establishing robust, unified evaluation methodologies is a necessary step toward enabling principled comparison and optimization of agentic AI systems in complex IT environments.

3 Optimization Techniques

Performance bottlenecks in agentic AI workflows for IT operations manifest primarily in two areas: memory pressure from processing extensive context, such as long logs and documentation, and high latency from the generative inference of large language models (LLMs) during agent planning and reasoning. This section surveys state-of-the-art techniques designed to alleviate these bottlenecks, focusing on inference optimization algorithms that directly improve agent responsiveness and processing capabilities.

3.1 Latency Reduction via Speculative Decoding

As discussed earlier, existing benchmarks for IT operations, such as ITBench, primarily focus on task efficacy while overlooking efficiency metrics, particularly latency. In real-world IT workflows—such as incident mitigation, anomaly triage, or service outage remediation—response time is not merely an efficiency metric but a functional requirement. Agentic AI systems typically operate in multi-step

loops involving planning, tool invocation, and reflection, where each stage may invoke an LLM. This sequential dependency leads to compounding latencies that can quickly accumulate, rendering otherwise capable agents impractical in time-sensitive environments.

Speculative decoding offers a practical approach to mitigating this autoregressive bottleneck. By leveraging a lightweight "propose" stage to generate multiple candidate tokens in parallel and validating them through a single forward pass of the target model, it replaces multiple memory-bandwidth-bound sequential computations with a highly parallelizable operation. This design can substantially reduce end-to-end wall-clock latency—often by a factor of 2–3—while preserving output fidelity. Such gains are particularly relevant for IT workloads where agents must generate long, structured outputs (e.g., shell commands or remediation plans) under tight response-time constraints.

3.1.1 The Foundational Approach: Two-Model Speculative Decoding

A foundational speculative decoding framework, introduced by Leviathan et al. (2023) and Chen et al. (2023), adopts a decompositional architecture: a small "draft model" first generates multiple token candidates in parallel, and a large "target model" then verifies them in a single pass. Using token-level rejection sampling, the system validates each candidate sequentially and accepts the prefix of all verified tokens.

This method established speculative decoding as a viable strategy for speeding up autoregressive generation in large language models, often achieving 2–3$\times$ speedups without compromising output fidelity. In the context of IT operations, this can be especially impactful for agents required to emit long, structured outputs—such as Kubernetes commands, remediation scripts, or audit rules—where fast, consistent generation is critical for automation. For example, an agent composing a command like `kubectl get pods -n monitoring` can produce the entire command in the time a standard decoder might take to emit just one or two tokens.

However, this approach also introduces deployment complexity: it requires maintaining and serving two distinct models, which increases memory footprint and engineering overhead. Whether benchmarks like ITBench should evaluate such techniques depends on whether latency-sensitive agent workloads are within the intended scope. If so, benchmark designers may need to incorporate task scenarios that stress responsiveness and throughput, rather than only accuracy.

3.1.2 Architectural Integration: Medusa for Single-Model Speculation

To address the deployment complexity of two-model speculative decoding, later work proposed a model-integrated design that embeds speculation directly within a single network. **Medusa** (Cai et al., 2023) exemplifies this approach by augmenting the base model with multiple lightweight prediction heads. In a single forward pass, while the main head predicts the next token ($t + 1$), auxiliary heads simultaneously predict future tokens ($t + 2$, $t + 3$, etc.), forming a speculative tree. A custom tree attention module then efficiently validates these predictions.

This design shifts complexity from the system level to the model level: although it requires a one-time architectural modification and retraining, it enables significant speedups without the need for an external draft model. Empirical evaluations show that Medusa can achieve up to 2.2$\times$ throughput improvements and reduce decoding latency by 30–50% across various sequence generation benchmarks. Such gains are particularly beneficial for IT operations agents generating structured outputs like multi-line JSON patches, YAML configurations, or compliance rules—where intermediate token dependencies are sparse and tree-based speculation can explore multiple plausible paths concurrently.

Moreover, because Medusa simplifies runtime deployment by avoiding multi-model orchestration, it may be better aligned with IT systems' operational constraints. If benchmarks such as ITBench aim to evaluate model-integrated optimizations, they may require tasks with complex structured outputs and fine-grained measurement of step-wise latency within decoding pipelines.

3.1.3 Algorithmic Innovation: Lookahead Decoding for Zero-Overhead Speculation

The most recent frontier in speculative decoding is the development of *algorithmic*, non-intrusive methods that require neither architectural changes nor auxiliary models. **Lookahead Decoding** (Fu et al., 2023) exemplifies this direction by enabling self-speculation through purely scheduling-level innovations. It exploits the parallelizable nature of Transformer layers to propose and verify multiple

N-grams in a single forward pass, thereby accelerating autoregressive decoding with no additional memory or computational footprint.

Despite its simplicity, Lookahead achieves non-trivial performance gains: empirical results show latency reductions of up to 35% and throughput improvements of 1.5–1.8$\times$ on diverse generation workloads. These gains make it particularly attractive for agentic AI systems that rely on fast intermediate reasoning—such as during multi-hop log analysis or real-time compliance audits in IT workflows—where deploying or retraining custom models is impractical.

Lookahead's minimal integration cost makes it a promising "drop-in" candidate for accelerating the internal CoT-style subroutines that often dominate end-to-end latency in IT agents. However, its general-purpose nature may limit maximum gains compared to more specialized techniques like Medusa. For benchmarks aiming to capture such low-overhead optimizations, task designs may need to expose fine-grained decoding latencies and track efficiency metrics beyond task success, further reinforcing the need to treat latency as a first-class concern in future evaluation protocols.

3.2 Memory Optimization via Activation-aware Weight Quantization

While speculative decoding alleviates latency in agent inference, agentic workflows in IT operations face a concurrent and equally critical constraint: memory capacity. Tasks in ITBench, such as SRE debugging or compliance analysis, often require agents to process thousands of log lines, lengthy policy documents, or complex system configurations. Such long-context workloads generate large key-value (KV) caches that can quickly exhaust GPU memory, limiting both model scale and usable context length.

Activation-aware Weight Quantization (AWQ) is a state-of-the-art compression technique that addresses this constraint by optimizing the model weights themselves. Unlike traditional post-training quantization methods such as GPTQ, which focus on minimizing numerical reconstruction error, **AWQ** (Lin et al., 2023) accounts for the interaction between weights and activations during inference. It observes that weights associated with large-magnitude activations have a disproportionate impact on model outputs. AWQ therefore identifies these salient weights through a calibration forward pass and selectively preserves their precision via per-channel scaling, while aggressively quantizing the remaining weights (typically to INT4). This activation-aware strategy enables significant model size reduction (approximately 4$\times$) while largely preserving task performance.

For agentic AI systems in IT operations, the benefits of AWQ extend beyond cost reduction along two key dimensions:

1. **Enabling Larger Model Deployment:** Quantization makes it feasible to deploy larger language models—such as 65B or 70B parameter models—on commodity or enterprise-grade GPUs. Larger models are generally hypothesized to better handle complex IT failures and security threats than smaller models (e.g., 13B), due to their increased representational capacity and reasoning depth. AWQ enables such higher-capacity models to be deployed under realistic hardware constraints.
2. **Supporting Longer Contexts:** By reducing the memory footprint of model weights, AWQ frees GPU memory that can be allocated to the KV cache. This additional memory headroom allows agents to process longer logs, richer configuration states, or more comprehensive documentation without truncation, which is particularly important for tasks such as root-cause analysis or policy compliance checking.

Consequently, quantization should not be viewed merely as a model compression or cost-saving measure. In the context of IT operations agents, it is a **key enabling technology** that directly influences agent **capability and scalability**. A comprehensive benchmark should therefore move beyond baseline efficiency metrics and explicitly consider the trade-offs between numerical precision, model scale, and supported context length—dimensions that AWQ helps to navigate in practical deployments.

3.3 Batching for LLM Inference

Batching techniques play a central role in improving the efficiency of LLM inference, which is directly amplified in agentic AI systems where rapid, repeated model calls determine overall responsiveness.

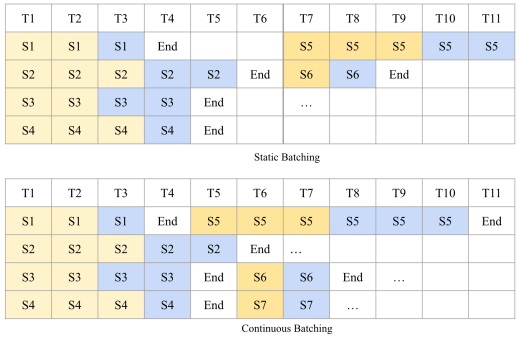

Figure 1: Static vs. continuous batching during decoding. Columns T1–T11 denote decoding time steps, and rows S1–S7 represent different sequences. Yellow blocks indicate prompt tokens, and blue blocks indicate generated tokens. In static batching (top), the batch is fixed once decoding starts, and new sequences (e.g., S5, S6) cannot join until all earlier ones finish. In continuous batching (bottom), the scheduler removes completed sequences and admitting new ones immediately.

Early deployments relied on static batching, grouping requests into fixed batches to amortize kernel-launch overhead. However, static batching struggles with the heterogeneous prompt lengths and asynchronous tool-calling patterns common in agent workflows, leading to substantial underutilization of GPU compute and memory. These constraints motivated increasingly adaptive approaches: first allowing batches to refresh more frequently under continuous request streams, then recognizing the fundamental imbalance between compute-heavy prefill and memory-bound decode, and eventually optimizing these phases separately. The following paragraphs introduce these techniques in order, highlighting how each development stems naturally from the limitations of its predecessor.

3.3.1 Continuous Batching

The first major advance beyond static batching is continuous batching, sometimes referred to as dynamic batching or token-level scheduling. As in Figure 1, instead of fixing batch composition per request, the scheduler dynamically admits new requests and removes finished ones at every decoding step, forming a new micro-batch each iteration. KV-cache memory is reused across iterations, preventing idle GPU cycles and mitigating divergence in sequence lengths. [2]

Given this mechanism, the effectiveness of continuous batching naturally appears in throughput under concurrent load and latency when processing many requests, since both depend on how efficiently the scheduler keeps the GPU occupied. Benchmarks such as ITBench (used later in our agent evaluation) are already able to measure these effects, and the cited Anyscale study evaluates the technique using exactly these metrics across different serving frameworks. In the context of IT operations, continuous batching can be particularly beneficial for workloads that involve bursts of tool calls or sustained log-processing traffic; however, it provides limited improvement for the earliest requests when the GPU is initially idle, making its gains more pronounced only once requests begin to accumulate.

3.3.2 Hybrid Batching

While continuous batching improves token-level concurrency, it does not explicitly address the imbalance between the compute-intensive prefill phase and the memory-bound decode phase. Hybrid batching was introduced to jointly schedule these phases within the same iteration. As shown in Figure **??**, the Sarathi framework [1] implements this by chunking long prefills into smaller segments and interleaving them with concurrent decode requests. Each hybrid batch therefore fuses prefill chunks—rich in matrix multiplications—with lightweight decode steps, allowing decodes to ride on the prefill phase's high arithmetic intensity and reducing pipeline bubbles that otherwise stall SMs.

Given this mechanism, hybrid batching is particularly advantageous under multi-request, multi-GPU workloads, where the scheduler can distribute prefill and decode chunks across devices with higher overlap. Although this technique is newer and less widely deployed than continuous batching, its implementation atop vLLM improves compatibility with popular benchmarking setups. Multi-GPU benchmarks such as NVIDIA's `tensorrt-llm-bench` support evaluating multi-requests scenario.

In IT operations settings, hybrid batching can accelerate workloads with long prompts or extended reasoning chains, though improvements appear mainly in steady-state decode throughput rather than the earliest requests, mirroring the behavior observed in continuous batching but amplified under multi-GPU conditions.

3.3.3 Disaggregated Scheduling

Beyond single-GPU optimization, disaggregated scheduling extends batching efficiency to multi-GPU and distributed serving environments. The DistServe architecture (Mo et al., 2024) separates the prefill and decode phases into dedicated GPU clusters connected by a high-throughput communication layer. A central controller dispatches each request first to a prefill cluster, which performs the computationally intensive prompt processing, and then transfers cached activations to a decode cluster specialized for low-latency, memory-bound generation. By decoupling compute- and memory-bound workloads, DistServe eliminates resource contention and allows independent scaling of each cluster. Under production-scale loads, it achieves 1.3–2.4× higher goodput and up to 10× lower tail latency than monolithic inference systems. This system complements hybrid batching by addressing the same prefill–decode imbalance at a higher architectural level, ensuring that throughput-oriented and latency-oriented resources are utilized optimally.

3.3.4 Discussion

The batching strategies discussed above are integral to scaling agentic AI systems used in IT automation and operations management. Frameworks such as ITBench and vAgentBench rely on efficient scheduling to handle concurrent reasoning agents (e.g., SRE, FSCO, and CISO agents) interacting continuously with LLMs. These techniques reduce latency, improve GPU utilization, and stabilize throughput across heterogeneous workloads. Table 1 summarizes their practical applications in agentic workflows.

Table 1: Batching Techniques and Their Applications in Agentic AI and IT-Automation

Technique	**Representative System / Framework**	**Agentic Workflow / IT-Automation Impact**
Continuous Batching	vLLM, TensorRT-LLM [3]	Maintains GPU utilization and responsiveness for concurrent reasoning threads
Hybrid Batching	Sarathi-Serve (Agrawal et al., 2023)	Interleaves prefill and decode phases for smoother utilization; improves throughput for multi-step diagnostic or planning agents.
Disaggregated Scheduling	DistServe (Mo et al., 2024)	Separates compute-bound prefills and memory-bound decodes across clusters; lowers tail latency in CISO threat-triage workflows.
Multi-Tenant Adapter Batching	Punica, S-LoRA, vLLM (with LoRA enabled)	Enables concurrent LoRA agents to share GPU compute while preserving isolation; supports multi-department CISO or customer-support automation.

3.4 Attention-Level Optimization Techniques

While batching policies orchestrate request- and iteration-level scheduling, the attention kernel itself remains the dominant computational and memory bottleneck during LLM inference. Each token requires computing attention over a growing key–value (KV) cache, whose size scales linearly with sequence length and batch width. As context lengths extend to tens of thousands of tokens and serving concurrency increases, the efficiency of attention memory layout, kernel design, and cache management becomes critical. A series of attention-level optimizations have been developed to alleviate these constraints, progressing from memory-efficient KV cache organization to fused kernel execution and adaptive compression.

3.4.1 PagedAttention

PagedAttention introduced a paradigm shift in KV-cache memory management for inference. Instead of allocating contiguous memory per sequence, it divides the KV cache into fixed-size pages that can be dynamically allocated, reused, and reclaimed. This paging mechanism allows non-contiguous storage of KV blocks, enabling the GPU to serve heterogeneous sequences without preallocating large, contiguous buffers. When a request completes or a sequence finishes decoding, its pages are immediately returned to a free list, preventing memory fragmentation and wastage. The approach achieves an order-of-magnitude improvement in memory utilization, with empirical reports of up to 20× less memory waste compared to naive contiguous allocation under mixed-length workloads. By decoupling memory allocation from sequence length, PagedAttention supports significantly larger batch sizes and longer context windows on the same hardware. Since its introduction in vLLM (2023), it has become the standard memory management layer in modern inference engines and serves as the foundation for nearly all subsequent attention-level optimizations.

3.4.2 POD-Attention

While PagedAttention optimizes memory layout, POD-Attention targets the computational imbalance between prefill and decode phases at the kernel level. Traditional inference systems employ separate kernels for prefill (matrix–matrix attention) and decode (matrix–vector attention), leading to poor hardware utilization when the two phases co-exist in hybrid batches. POD-Attention (Parallel Overlapping Dual-phase Attention) unifies these operations within a single GPU kernel, co-scheduling prefill and decode tasks across streaming multiprocessors (SMs). By exploiting complementary resource usage—prefill is compute-bound, whereas decode is memory-bound—POD-Attention maximizes overlap and minimizes idle execution units. This kernel fusion aligns with the philosophy of hybrid batching but implements it at the micro-kernel level. Experimental evaluations report up to 59% acceleration in attention computation (mean 28 $\%$) and roughly 22 $\%$ end-to-end throughput gain in hybrid-batched LLM inference compared to phase-specific kernels. The technique exemplifies how fine-grained kernel co-execution can extend the benefits of high-level scheduling to the GPU instruction level.

3.4.3 FlashInfer

As model architectures and inference patterns diversify, static kernels struggle to support variants such as grouped-query attention, rotary embeddings, or block-sparse matrices. FlashInfer (2025) generalizes attention computation through a unified kernel engine that supports block-sparse formats, JIT-compiled fusion, and runtime reconfiguration. Instead of hand-tuning kernels for each attention type, FlashInfer generates optimized kernels on demand, adapting to different sequence layouts, precision modes, and parallel generation strategies. The design builds upon ideas from earlier work such as FlashAttention but extends them for online serving rather than offline training. By dynamically selecting optimal memory layouts and operation orders, FlashInfer reduces inter-token latency by 29–69

3.4.4 KV-Cache Compression and Eviction

Even with efficient paging, the KV cache grows linearly with sequence length, imposing a hard limit on memory capacity. Recent research therefore explores KV-cache compression and adaptive eviction to further scale long-context inference. Techniques such as KV-Compress and HashEvict compress or discard low-importance tokens based on attention statistics. KV-Compress performs per-head quantization and variable-ratio compression of the KV cache, reducing its footprint by up to 8× with negligible degradation in output quality. HashEvict uses locality-sensitive hashing (LSH) to identify redundant keys that contribute little to future attention scores, freeing 30–70 $\%$ of cache memory without noticeable performance loss. These strategies are complementary to PagedAttention: while paging mitigates fragmentation, compression and eviction reduce total memory demand. Together, they enable serving longer sequences, more concurrent requests, and denser multi-tenant deployments under fixed GPU memory budgets.

3.4.5 FastTree and Context-Shared Attention

In interactive or multi-agent scenarios, many requests share common prefix contexts, leading to redundant KV-cache storage and repeated attention over identical tokens. FastTree introduces context-shared attention kernels that exploit this structural redundancy. It organizes KV caches into a radix-tree representation, where shared prefixes are stored once and referenced by multiple decoding branches. During attention computation, queries traverse only the divergent nodes of the tree, reusing precomputed attention results for shared segments. This design substantially reduces memory traffic and arithmetic cost for workloads such as beam search or hierarchical text generation. Experimental results show up to 5.1× speedup in attention computation over FlashAttention baselines and 2.2× end-to-end improvement for models such as LLaMA and Mistral under tree-structured decoding. Although its applicability is scenario-specific, FastTree highlights the potential of structural reuse in attention kernels when serving multiple correlated sessions.

3.4.6 Discussion

Attention-level optimizations refine how self-attention and KV-cache operations are executed within LLM inference, complementing batching by addressing memory and bandwidth bottlenecks. Systems such as vLLM, Sarathi-Serve, and FastTree integrate these kernels to enable longer contexts and higher throughput under concurrent multi-agent workloads. Table 2 summarizes their representative frameworks and practical relevance to agentic IT-automation tasks.

Table 2: Attention-Level Techniques and Their Applications in Agentic AI and IT-Automation

Technique	**Representative System / Framework**	**Agentic Workflow / IT-Automation Impact**
Paged Attention	vLLM	Implements a virtual-memory–style paged KV-cache to reduce fragmentation and memory waste; supports persistent long-context sessions for SRE and CISO agents analyzing extended logs or playbooks.
POD Attention	Sarathi-Serve (Agrawal et al., 2023)	Overlaps attention computation across decoding steps to eliminate idle GPU cycles; improves responsiveness for incident-triage or diagnostic agents processing long reasoning chains.
FastTree Attention	SGLang	Builds hierarchical prefix-sharing trees to reuse attention results across similar requests; accelerates multi-agent orchestration and plan expansion where agents share system prompts.
FlashAttention	DAO et al. (2022)	Reformulates attention as an IO-aware exact algorithm, reducing HBM traffic and enabling longer sequences; decreases per-step latency for streaming or continuous reasoning agents.
KV-Cache Compression / Reuse	TensorRT-LLM	Compresses or prunes KV-cache entries to free GPU memory and maintain throughput; allows more concurrent agent sessions during high-load SRE or CISO response events.

4 Case Study: vLLM Optimization for Qwen2.5-Based Agents on Commodity GPUs

This section evaluates how vLLM accelerates sequential agentic workflows through its suite of inference optimizations. We frame this as a systems study of which optimization techniques matter—and why—for long-context, multi-step workloads. We benchmarked two distinct agentic frameworks: the **ITBench SRE Agent** and the **SWE-agent**.

Although SWE-agent focuses on software engineering (resolving GitHub issues) rather than strictly IT operations, we include it as a comparative baseline because it shares the critical architectural constraints of SRE workloads: a strictly sequential loop (Thought $\rightarrow$ Action $\rightarrow$ Observation) and heavy context dependency. By contrasting infrastructure log analysis (ITBench) with code repository navigation (SWE-agent), we demonstrate that the system-level bottlenecks—and their solutions—are universal across long-horizon autonomous agents.

Four inference configurations were benchmarked: a naive HuggingFace baseline (without any optimizations), vLLM without batching (Serial), vLLM with continuous batching enabled (Full), and vLLM augmented with speculative decoding. Through this comparative analysis, we differentiate between optimizations that meaningfully reduce agentic latency and those whose benefits are structurally limited by workload.

4.1 Experimental Setup

Model and Runtime Configuration Experiments were conducted using the `Qwen2.5-14B-Instruct-AWQ` model deployed on vLLM. The environment consisted of a single NVIDIA L4 GPU with a maximum sequence length of 32,768 tokens and an effective GPU memory utilization target of 0.90. We selected this configuration to represent a realistic "commodity" deployment scenario, where organizations seek to host capable mid-sized agents on cost-effective, single-GPU instances rather than massive clusters.

Task Selection and Reproducibility To ensure rigorous interpretability, we targeted specific representative tasks rather than the entire dataset. For ITBench, we focused on SRE Scenarios 1–3, reporting detailed profiling metrics for Incident #1 (Kubernetes CrashLoopBackOff). For SWE-agent, we evaluated performance on 20 randomly selected instances from the 300 tasks in the SWE-bench Lite dataset. All data points reported in Section 4.3 represent the average of 5 independent runs to account for system variance.

Agent Workflows We evaluated two distinct agent architectures:

- **ITBench SRE Agent:** Executes a diagnostic loop involving Kubernetes and IT tools. The context grows dynamically with system logs and command outputs.
- **SWE-agent:** Addresses GitHub issues by navigating file systems, reproducing bugs, and generating patches. Like the SRE agent, it follows a serial dependency structure: the agent must wait for the environment (e.g., a test suite or linter) to return an observation before generating the next thought.

Crucially, this serial dependency implies that at no point do multiple LLM requests overlap within a single agent instance.

4.2 vLLM Optimization Features Overview

vLLM incorporates a collection of architectural and kernel-level optimizations designed to serve large language models efficiently. Table 3 summarizes these techniques and their conceptual impact on agentic workloads.

4.3 End-to-End Performance Results

The performance comparison across both agents is summarized in Table 4. The results demonstrate a dramatic improvement when switching from the baseline to vLLM, with SWE-agent achieving generation speeds of approximately 26 tokens/second. However, adding batching or speculation provided negligible gains.

Task Efficacy and Output Consistency. In addition to latency and throughput, we evaluated whether different vLLM configurations affected task-level correctness. Across all tested configurations—including prefix caching, continuous batching, speculative decoding, and AWQ quantization—all agents achieved identical task success rates. No configuration produced divergent outputs or altered final task outcomes. These results confirm that the observed performance gains stem purely from system-level optimizations and do not degrade reasoning quality or task accuracy.

Table 3: vLLM Optimization Features and Their Impact on Agents

Optimization	Status in vLLM	Impact on Agentic Workloads
PagedAttention	Core / Always On	Eliminates memory fragmentation. Crucial for agents with contexts exceeding 10k+ tokens, preventing OOM errors during long debugging sessions.
Prefix Caching	Optional (Enabled)	Caches KV-blocks for the system prompt and history. Since logs and file contents persist across steps, this allows the engine to skip redundant computation.
Chunked Prefill	Default	Splits long prompts (e.g., massive log files) into micro-batches. Prevents the GPU from stalling and improves inter-token responsiveness.
Continuous Batching	Core / Always On	Maximizes throughput by processing requests at the token level. Provides minimal benefit for single-stream, sequential agents.
CUDA Graphs	Default (Decoding)	Reduces CPU launch overhead. Stabilizes decoding speed, ensuring consistent token generation rates.
Optimized Kernels	Default	Fast implementations of Norms, Activations, and AWQ Quantization. Significantly reduces latency for quantized models.

Table 4: Performance Comparison of Inference Configurations

Workload	Configuration	Latency (ms/tok)	Speed (tok/s)	Total Runtime (s)
ITBench SRE	Baseline (HF)	121.0	8.2	$\approx$10,800 s
	vLLM Serial	40.1	24.9	422 ± 23 s
	vLLM Full	43.0	23.3	446 ± 24 s
	vLLM + Spec	43.9	23.0	443 ± 26 s
SWE-agent	Baseline (HF)	109.9	9.1	$\approx$5,300 s
	vLLM Serial	38.5	26.0	531 ± 21 s
	vLLM Full	38.8	25.8	552 ± 18 s
	vLLM + Spec	38.8	25.8	549 ± 25 s

4.4 Analysis

4.4.1 Deconstructing the Speedup: The Prefill Dominance

A naive reading of Table 4 presents a paradox: vLLM's generation speed (26 tok/s) is only $3\times$ faster than the baseline (9 tok/s), yet the total runtime drops by a factor of $10\times$ (from over an hour to 7 minutes). This discrepancy highlights that **generation latency is not the bottleneck**.

The massive time savings are driven almost entirely by the optimization of **Prompt Processing (Prefill)**. We can formalize the cumulative cost over a session of T turns, where the context length grows from N_0 by δ tokens per turn.

- **The Baseline Cost ($O(T \cdot N^2)$):** In a naive implementation, the attention mechanism must re-process the entire history at every step. The total cost is the sum of quadratic operations

over all turns:

$$\text{Cost}_{\text{base}} \approx \sum_{t=1}^{T} O((N_0 + t \cdot \delta)^2) \approx O(T \cdot N_{\text{final}}^2)$$

This cumulative quadratic growth explains why the baseline becomes significantly slower as the context length increases; the cost of the 50^{th} step is significantly higher than the 1^{st}.

- **The vLLM Advantage ($O(T)$):** Through **Prefix Caching**, vLLM retrieves the pre-computed Key-Value (KV) cache for these tokens instantly, processing only the new tokens (δ) at each step. Prompt processing becomes nearly instantaneous. The average prompt throughput achieves $\approx$ 2900 tokens/second.

$$\text{Cost}_{\text{vLLM}} \approx \sum_{t=1}^{T} O(\delta^2) \approx O(T)$$

This transforms the complexity from quadratic to linear with respect to the number of turns, effectively reducing the prefill latency for cached segments to near-zero.

4.4.2 Memory Profiling: PagedAttention vs. Contiguous Allocation

To validate the claim that PagedAttention enables deeper context, we profiled the peak GPU memory usage of the `Qwen2.5-14B-AWQ` model on the 24GB NVIDIA L4 GPU.

Table 5: Memory Consumption and Context Limits (24GB L4 GPU)

Configuration	Model Weights	KV-Cache Allocation	Fragmentation	Max Safe Context
Baseline (HF)	9.8 GB	Contiguous	$\sim$50–80%	$\approx$ 8,000 tokens
vLLM	9.8 GB	Non-Contiguous (Paged)	$< 4\%$	$\approx$ 28,000 tokens

As shown in Table 5, both configurations reserve $\approx$10GB for model weights, but manage the remaining VRAM differently:

- **The Baseline Bottleneck:** The naive HuggingFace implementation mandates contiguous KV-cache allocation. Dynamic log growth forces over-provisioning, resulting in 60–80% fragmentation waste. Consequently, the baseline triggered premature OOM errors at just $\approx$ 8,000 tokens.
- **vLLM's Paged Efficiency:** vLLM manages memory in small, non-contiguous pages (similar to OS virtual memory), reducing fragmentation to $< 4\%$. This allowed the SWE-agent to robustly process contexts exceeding 28,000 tokens, unlocking the long-context capabilities required for massive server logs.

4.4.3 Why Continuous Batching Shows Limited Benefits

As surveyed in Section 3.3, continuous batching is designed to maximize throughput when multiple requests arrive concurrently. However, our experiments reveal that these benefits do not transfer to single-agent workflows.

- **Sequential Dependency:** Each LLM call must wait for external tools (e.g., executing a Kubernetes command or running a test suite) to complete before the next call is issued.
- **Batch Size of One:** With only a single active request at any point in time, batching opportunities never arise. The overhead of the batching scheduler slightly outweighs its benefits, which is why *vLLM Full* performs marginally worse than *vLLM Serial*. It is important to note that while this limitation holds for our single-agent experimental setup, production deployments managing multiple concurrent agent instances (e.g., parallel incident remediation) would leverage the throughput benefits of continuous batching.

4.4.4 Why Speculative Decoding Does Not Improve Performance

As discussed in Section 3.1, speculative decoding typically offers significant gains in chat workloads. We evaluated this using an **N-gram draft strategy** (5-token window), which offers a lightweight, zero-memory-overhead alternative to loading a secondary draft model. However, we found that these benefits are negated by the structural characteristics of agentic reasoning:

- **Low Acceptance Rate:** We observed a negligible acceptance rate of $\approx 0.1\%$ for speculative tokens. Unlike natural language chat, where N-grams effectively capture common phrases, SRE diagnostics and code patches are highly entropy-dense. The agent's output is strictly conditioned on unique, dynamic system states (e.g., specific pod IDs, error codes, or variable names) which do not repeat predictably enough for an N-gram model to guess correctly. Consequently, the overhead of the verification step cancelled out the minimal gains from the few accepted tokens.
- **Prefill Dominance:** Even if the acceptance rate were higher, the structural impact would be limited. Speculative decoding only accelerates the *decode* phase (token generation). In our agentic workflow, the primary latency bottleneck is the *prefill* phase (processing the massive context history). Accelerating the generation of the ≈50 output tokens does little to offset the cost of processing the 20,000 input tokens, rendering the optimization bounded.

4.4.5 Cost Efficiency and Token Consumption

To validate the *Cost* dimension of our unified metrics framework mentioned in Section 2.2.4, we profiled the total token usage and inference cost for the representative ITBench SRE scenario Incident #1.

Table 6: Token Consumption and Cost Analysis (ITBench SRE Scenario)

Metric	**Value**
Total Requests	93
Total Prompt Tokens	648,971
Total Generation Tokens	8,301
Total Token Volume	**657,272**
Runtime (vLLM Serial)	422 s (≈ 7 min)
Est. Cloud Cost (L4 GPU)	**$0.09**
Est. API Cost (GPT-4o Equivalent)	≈ $1.70

As detailed in Table 6, the workload is heavily dominated by prefill, with prompt tokens accounting for 98.7% of the total traffic. By self-hosting on a commodity NVIDIA L4 GPU (approx. $0.80/hour), the total inference cost for a single execution episode was less than $0.10.

In contrast, executing the same volume of tokens via a commercial API (e.g., GPT-4o) would cost approximately 18× more (≈$1.70). While commercial models offer higher reasoning capabilities, utilizing them for workloads dominated by context ingestion incurs a significant cost inefficiency. This confirms that optimizing the inference pipeline not only improves latency (Efficiency) but also fundamentally alters the economic viability (Cost) of deploying autonomous agents at scale, particularly for high-frequency monitoring loops where execution cost is a primary constraint.

4.5 Discussion and Conclusion

4.5.1 Architectural Implications for Autonomous Agents

Our evaluation highlights that the effectiveness of inference optimization is not universal, but strictly coupled to the structural characteristics of the workload. For autonomous software agents—which operate in a sequential, state-heavy, and reasoning-dense paradigm—the optimization landscape differs fundamentally from high-throughput chat serving.

The "Memory-First" Optimization Hierarchy We observe that for agentic workflows, memory management techniques are significantly more impactful than compute-bound optimizations.

- **Prefix Caching as an Amortized Accelerator:** The most transformative finding is the impact of Prefix Caching on the agent's inner loop. Prefix caching amortizes the heavy prefill computation across turns by retaining the KV-states of the static history. This reduces the marginal cost per turn from the quadratic $O(N^2)$ of re-processing the full context to $O(\delta \cdot N)$ (where δ is the number of new tokens and $\delta \ll N$). This architectural shift reduces the time-to-first-token (TTFT) from seconds to milliseconds, preventing the "context tax" that typically makes long-horizon agents prohibitively slow.
- **PagedAttention as the Enabler of Depth:** While Prefix Caching optimizes speed, PagedAttention optimizes capacity. By eliminating external memory fragmentation, it allows the agents to maximize the remaining ≈14GB of VRAM for actual context storage rather than losing it to contiguous reservation blocks. This difference is operationally critical: it distinguishes an agent that crashes after reading a few log files from one that can robustly navigate a complex repository without truncation.

The Limits of Throughput-Oriented Techniques Conversely, industry-standard optimizations designed for serving thousands of concurrent users show diminishing returns in the "batch-size-of-one" regime of autonomous agents.

- **The Sequential Bottleneck:** Continuous Batching relies on interleaving multiple request streams to saturate GPU compute units. However, an autonomous agent is strictly blocked by its environment; it cannot generate the next thought until the previous action (e.g., `kubectl get pods` or `pytest`) returns an observation. This serial dependency means that the GPU sits idle during tool execution, rendering batching mechanisms overhead-heavy rather than throughput-enhancing.
- **The Reasoning Barrier for Speculation:** Speculative Decoding failed to produce speedups because of the high information density of the agent's output. Unlike casual conversation, where draft strategies can easily predict common phrases, SRE diagnostics and code patches are highly entropy-dense and conditioned on specific system states. The rejection rate becomes too high to justify the computational overhead of the verification step, even when using lightweight N-gram speculation.

4.5.2 Conclusion and Implications for Benchmarking

These results suggest that the standard "chatbot serving stack" is suboptimal for autonomous engineering. Deploying capable agents requires a pivot from maximizing *aggregate throughput* (tokens/sec across all users) to minimizing *single-stream latency* (time-to-solution for one problem).

Implications for Future Benchmarks These findings directly address the gaps identified in our survey of ITBench and DevOps-Gym (Section 2). Since system-level optimizations like Prefix Caching can reduce runtime by an order of magnitude (from hours to minutes), measuring task accuracy alone is insufficient for evaluating operational readiness. We propose that future iterations of IT operations benchmarks should:

- **Standardize Inference Configurations:** Benchmarks must control for hardware constraints to ensure fair comparisons. This includes defining fixed VRAM budgets and specifying allowable memory optimization techniques (e.g., PagedAttention, Prefix Caching, AWQ Quantization), as our results demonstrate that memory architecture—not just model intelligence—determines whether a long-context task completes or crashes.
- **Report Granular Operational Metrics:** Evaluation boards should expand beyond accuracy to report *Time-to-Solution* and *Inference Cost*. Crucially, metrics should decouple *Prefill Latency* (context processing) from *Decode Latency* (generation) and report *Prefix Reuse Rates*. This granularity is necessary to distinguish agents that are merely "correct" from those that are responsive and economically viable for high-volume SRE operations.

Feasibility of Sovereign Deployment The vLLM stack, when properly tuned with PagedAttention and Prefix Caching, transforms the SRE and SWE-agent workflows from theoretical prototypes into

responsive tools capable of executing multi-step remediation in minutes. While the NVIDIA L4 represents accessible enterprise hardware rather than true commodity consumer gear, our results demonstrate that mid-sized (14B) models can be hosted effectively on such infrastructure. This offers a cost-efficient and privacy-preserving alternative to external APIs for many operational tasks, significantly lowering the barrier to entry for secure, self-hosted autonomous engineering.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, Santa Clara, CA, July 2024. USENIX Association.

[2] Cade Daniel, Chen Shen, Eric Liang, and Richard Liaw. How continuous batching enables 23× throughput in LLM inference while reducing p50 latency. `https://www.anyscale.com/blog/continuous-batching-llm-inference`, 2023. Accessed: 2025-12-12.

[3] nyunAI/TensorRT-LLM Maintainers. In-flight batching — tensorrt-llm. `https://github.com/nyunAI/TensorRT-LLM#in-flight-batching`, 2025. Accessed: 2025-12-12.