DualLane: Fast and Reliable LLM Agents for Interactive AIOps via Dual-Path Planning

Haoyu Wang
Alibaba Cloud Computing
Hangzhou, China
lingzun.why@alibaba-inc.com

Wenxuan Ma
Alibaba Cloud Computing
Beijing, China
wenxuan.mwx@alibaba-inc.com

Bing Hu
Alibaba Cloud Computing
Shanghai, China
yedao.hb@alibaba-inc.com

Haozhe Li
Alibaba Cloud Computing
Beijing, China
leon.lhz@alibaba-inc.com

Qingqian Si
Alibaba Cloud Computing
Beijing, China
siqingqian.sqq@alibaba-inc.com

Hongke Guo
Alibaba Cloud Computing
Beijing, China
guohongke.ghk@alibaba-inc.com

Boyang Huang
Alibaba Cloud Computing
Beijing, China
yuejia.yj@alibaba-inc.com

Zhaoliang Zhu
Alibaba Cloud Computing
Hangzhou, China
rongdi.zzl@alibaba-inc.com

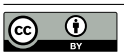
You Zhang
Alibaba Cloud Computing
Hangzhou, China
zhangyou.zy@alibaba-inc.com

Yu Zhou
Alibaba Cloud Computing
Beijing, China
tuhu@alibaba-inc.com

Xudong Zheng
Alibaba Cloud Computing
Hangzhou, China
xudong.zxd@alibaba-inc.com

Abstract

Ticket support in cloud services involves technical engineers using expert tools to resolve customer queries—a process closely resembling information integration. To enhance both operational efficiency and resolution accuracy, we propose DualLane, a novel parallel dual-path planning architecture designed for AI agents. This framework adaptively manages the highly skewed frequency distribution characteristic of real-world user queries. For low-frequency long-tail scenarios, the slow path employs a two-stage planning mechanism that decouples task decomposition from parameter generation, effectively optimizing dependency propagation and reducing context complexity. In contrast, for high-frequency routine scenarios, the fast path bypasses expensive LLM-based full-plan generation by utilizing pre-validated execution templates, thereby improving response accuracy and reducing latency. After more than one year of extensive deployment in Alibaba Cloud's ECS production environment, DualLane has demonstrated remarkable robustness and stability. Offline benchmarks indicate a high accuracy rate of 96.5%, accompanied by superior latency performance. Crucially, online metrics reveal a median plan-execution latency of merely 4.2 seconds, with an agent-induced error rate maintained at a low 7.1%. These results underscore the practical viability and effectiveness of adaptive dual-path planning in large-scale, interactive AIOps systems.



KDD '26, Jeju Island, Republic of Korea

ACM ISBN 979-8-4007-2259-2/2026/08
https://doi.org/10.1145/3770855.3818380

CCS Concepts

• **Information systems** → *Information integration*; • **Computing methodologies** → **Planning and scheduling**; • **Software and its engineering** → **Software maintenance tools**.

Keywords

LLM agent, AIOps, Information integration

ACM Reference Format:
Haoyu Wang, Wenxuan Ma, Bing Hu, Haozhe Li, Qingqian Si, Hongke Guo, Boyang Huang, Zhaoliang Zhu, You Zhang, Yu Zhou, and Xudong Zheng. 2026. DualLane: Fast and Reliable LLM Agents for Interactive AIOps via Dual-Path Planning. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2 (KDD '26), August 09–13, 2026, Jeju Island, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3770855.3818380

1 Introduction

Artificial Intelligence for IT Operations (AIOps) [11, 28] has been widely applied in the field of cloud services. Extensive research has concentrated on developing and refining automated operations and maintenance frameworks [21, 31], including data monitoring [16], data cleaning [32], anomaly detection [25, 30], fault prediction [12, 26], log diagnosis [27], and elastic resource allocation [29]. However, automated mechanisms cannot resolve all issues, and a large number of service tickets still necessitate manual resolution by technical support engineers.

1.1 Problem

Alibaba Cloud Elastic Compute Service (ECS) [4] faces similar challenges. As illustrated in Figure 1 with grayish blue lines, technical

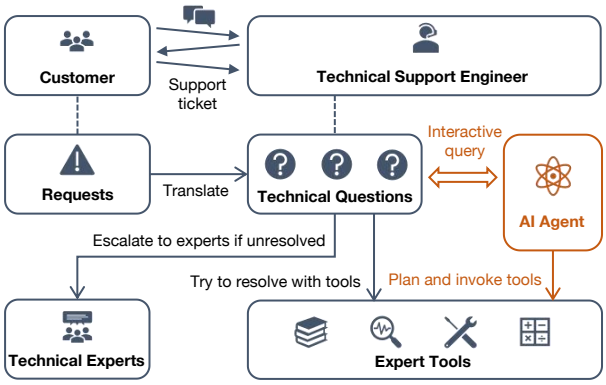


Figure 1: Technical support workflow

support engineers translate customer requests into several technical questions through interactive communications. They then attempt to resolve these questions using expert tools, escalating unresolved cases to senior experts when necessary.

The process of resolving service tickets can be viewed as a form of information integration [22]. For each issue, engineers select tools (data sources) from increasingly diverse and expanding systems and interfaces, transform and integrate heterogeneous outputs using their domain expertise, and synthesize the results into a unified natural language response.

With the rapid advancement of large language model (LLM) technologies, we aim to introduce an AI agent to address the inefficiencies and heavy reliance on human expertise in manual integration processes. As depicted by the orange components in Figure 1, the AI agent will autonomously plan sequences of tool calls and integrate their outputs, providing an intelligent and automated solution for the integration task.

1.2 Intuition

In building AI agents, ReAct [35] has emerged as a widely adopted architectural paradigm. It enables autonomous problem-solving through an iterative cycle of reasoning, acting, and observing. However, this sequential process incurs high latency, which significantly degrades user experience in interactive applications.

To mitigate this limitation, LLMCompiler [19] introduces a directed acyclic graph (DAG) representation of tool invocations and their dependencies, enabling parallel execution to minimize the time cost of tool invocation.

In Sections 2.1 and 2.2, we analyze the efficiency and accuracy challenges of these methods, respectively, and identify the planning phase as a potential direction for optimization. As discussed in Section 2.3, certain usage scenarios exhibit clearly identifiable high frequency. To exploit this pattern, we propose a dual-path planning strategy **DualLane**. For low-frequency scenarios, we fully leverage the LLM's intrinsic planning capability to maintain flexibility and generality. In contrast, for high-frequency scenarios, we bypass LLM-driven full-plan generation by leveraging pre-designed and human-verified execution plans, significantly reducing response latency while improving accuracy.

1.3 Contributions

Our key contributions are summarized as follows:

(1) We identify a pivotal domain characteristic in interactive AIOps: the **distinct clustering** of problem distributions. This insight exposes high-frequency patterns as exploitable structures, offering a dual advantage of reduced latency and improved planning accuracy.

(2) We propose an **original dual-path planning architecture** DualLane that enables parallel fast and slow path planning. It integrates two-stage planning, LLM-driven scenario matching, and resource-based pruning to achieve both efficiency and accuracy.

(3) Extensive evaluations show that DualLane significantly outperforms ReAct and LLMCompiler in both latency and accuracy under real-world workloads, achieving 96.5% overall accuracy and up to 44.6% faster in plan and execution.

(4) We deploy DualLane architecture in Alibaba Cloud ECS for over a year. In real-world production environment, the median plan-execution latency is only 4.2 seconds and only 7.1% of queries are incorrectly addressed due to agent-induced errors, providing effective support for ticketing services.

2 Motivation

Efficiency and accuracy are significant for LLM agents. In this section, we first analyze the challenges of existing architectures and then explore potential optimization strategies.

2.1 Challenge I: Plan-Execution Latency

Although LLMs support streaming output, allowing users to start reading the response before the entire process completes, users still have to wait during the agent's planning and execution phases. Therefore, the agent should minimize the plan-execution latency to optimize user experience. Below is an example analyzing the latency of existing agent architectures such as ReAct and LLMCompiler.

Example 2.1. Suppose a user wants to know whether there is an operation action affecting a VM[1] (called *vm-1*). In addition to actions directly performed on the VM, those on the host machine can also affect the VM. Thus, this question can be addressed using the following two tools:

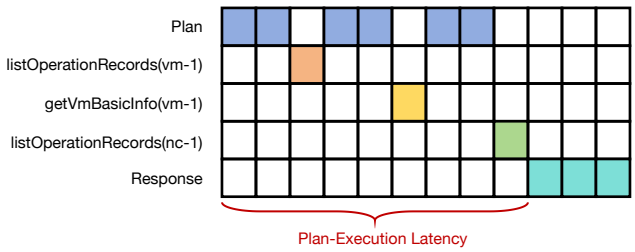
(1) **getVmBasicInfo**: This tool returns the basic information of a VM, including its host machine.

(2) **listOperationRecords**: This tool returns the operation actions performed on a specific machine. The machine can be either a VM or a host machine.

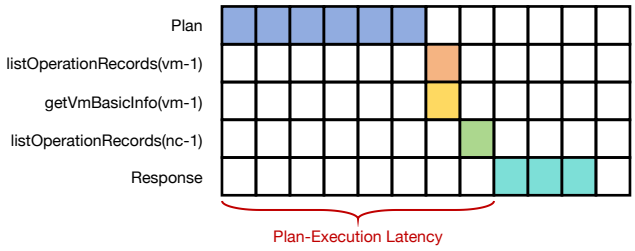
To solve this problem, ReAct first plans and executes **listOperationRecords** to query the operation actions performed on *vm-1*. Then, it invokes **getVmBasicInfo** to get the host machine of *vm-1* (denoted as *nc-1*). Subsequently, **listOperationRecords** is called again to query the operation actions performed on *nc-1*. Finally, the results from the two **listOperationRecords** calls are synthesized to generate the response. Figure 2 (a) shows the temporal flow chart.

To improve execution efficiency, LLMCompiler generates a DAG-based execution plan that enables parallel execution of independent

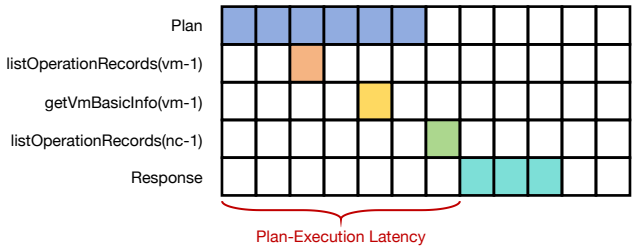
[1]Virtual machine (VM) is the software emulation of a physical computer. Its host machine is called node controller (NC).



(a) ReAct



(b) LLMCompiler



(c) LLMCompiler in stream mode

Figure 2: Temporal flow of LLM agent architectures

tools. As shown in Figure 2 (b), it concurrently queries the operation actions on the VM and the host machine. The former relies solely on the **listOperationRecords** tool, while the latter involves a sequential execution of **getVmBasicInfo** followed by **listOperationRecords**.

Furthermore, the stream mode of LLMCompiler initiates tool execution at an earlier stage. Each step is immediately enqueued for execution upon completion, rather than waiting for the entire DAG plan to be fully generated. This concurrent execution during the planning phase further improves efficiency, especially for plans with inter-step dependencies. Its temporal flow chart is shown in Figure 2 (c).

Theoretically, the DAG-based tool execution is optimal. Thus, our optimization efforts are focused solely on the planning phase.

2.2 Challenge II: Inaccuracy from Planning

Incorrect agent responses arise from several sources: first, the agent may make errors in tool selection, failing to select certain critical tools; second, the parameters generated by the agent for tool calls may contain issues such as missing parameters, incorrect parameter names, or wrong parameter types; third, when handling complex tasks with dependencies, the agent may fail to accurately extract required parameters from prior-step results, leading to dependency errors; finally, when synthesizing tool outputs into the final response, the agent may misinterpret the results or omit certain key information, resulting in an incorrect final answer.

Table 1: Error categories for LLM Agents

Reason	ReAct	LLMCompiler
Failure in tool selection	68.3%	62.1%
Incorrect tool parameters	18.3%	18.2%
Broken inter-step dependency	3.3%	10.6%
Incorrect or incomplete conclusion	10.0%	9.1%

Table 2: Top 5 user query scenarios

Scenario	Percentage
Query general knowledge	19.9%
Query the information of a VM	7.9%
Evaluate the success likelihood of migration	7.2%
Query the system events of a VM	6.5%
Query the information of a physical machine	5.4%

Table 1 presents the classification of error causes and their corresponding proportions. It can be observed that deficiencies in planning capability, corresponding to the first three types of causes, are the primary reasons for inaccurate responses. Therefore, the planning phase is the key focus of our optimization.

2.3 Opportunity: High-Frequency Scenarios

Based on the above analysis, we need to improve the planning phase. In this phase, LLM is used to generate tool execution parameters, which contain complex structures such as parameter dependencies and nested constructs.

Empirical analysis of user queries reveals that a significant proportion exhibit recurring, semantically similar patterns—what we define as **high-frequency scenarios**. While these queries may vary in phrasing or reference different resources, they share nearly identical intents and require the same tool execution workflows. For example, "What is the instance type of VM 1?" and "Please give VM 2's instance type" both seek to retrieve a VM's configuration, reflecting the same underlying task. Table 2 shows the top 5 scenarios, covering 47.0% queries.

This observation presents a key optimization opportunity: for queries belonging to high-frequency scenarios, they can be directly mapped to pre-defined and validated execution plans. This strategy is analogous to semantic-level caching, where queries from frequent scenarios hit the cache and bypass the high-cost LLM inference to generate the full plan. As a result, agent latency can be significantly reduced while maintaining correctness.

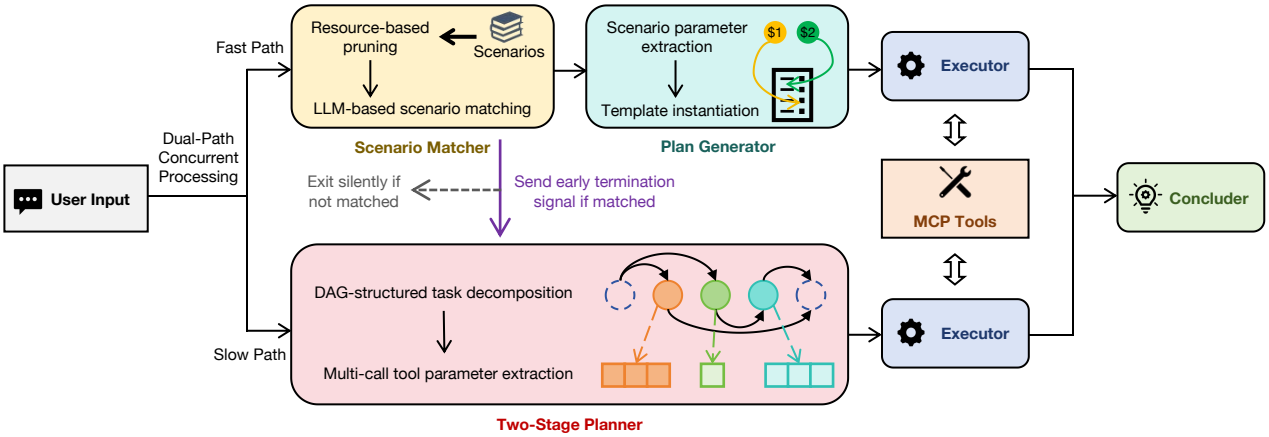


Figure 3: Architecture of dual-path planning

3 DualLane

3.1 Overview

Building on the insights from Section 2.3, we propose a dual-path planning architecture, **DualLane**, illustrated in Figure 3. For each user input, processing proceeds in parallel through two complementary pathways: a *slow path* designed for complex, long-tail scenarios, and a *fast path* optimized for frequent, routine queries.

The slow path handles intricate and diverse tasks through a two-stage planning process (Section 3.2), where the first stage generates a DAG of interdependent steps and the second stage maps each step to one or more tool calls, leveraging the LLM's reasoning to orchestrate complex workflows. This path serves as a robust foundation for addressing challenging queries and informs the design of efficient, pattern-driven mechanisms in the fast path.

In parallel, the fast path targets high-frequency scenarios by performing efficient scenario matching (Section 3.3.1) and rapidly generating predefined plans (Section 3.3.2). A resource-based scenario pruning mechanism (Section 3.3.3) is implemented to ensure feasibility and efficiency.

The two paths operate concurrently (Section 3.4), ensuring both coverage and efficiency. Upon a fast path hit, the slow path is terminated; upon a miss, the fast path silently exits. Therefore, only one path will eventually complete. The resulting plan and actual tool outputs are sent to the LLM Concluder. This minimal context reduces noise, enabling a focused and reliable final response.

3.2 Slow Path

The slow path serves as a general-purpose solution applicable to all scenarios. To minimize latency, the slow path also organizes tool invocations into a DAG like LLMCompiler. However, the planning approach of LLMCompiler, which relies on a single LLM call with placeholders, is ill-suited for AIOps.

It struggles with complex dependencies, especially when tool responses contain nested or structured data (e.g., lists of objects) requiring filtering, mapping, or conditional logic. Generating accurate placeholders and extraction rules in such cases is error-prone.

Moreover, since each step maps to exactly one tool invocation, the framework lacks runtime adaptability. Even simple tasks such as iterating over a variable-length list and issuing a tool invocation per element necessitate expensive re-planning mechanisms. This rigidity makes it poorly suited for dynamic, data-driven workflows typical in AIOps.

Thus, we propose a **two-stage planning** framework powered by LLMs. In the first stage, the LLM performs task decomposition, breaking down the user's high-level instruction into a sequence of actionable steps, each aligned with a specific tool. Importantly, this process requires only a brief description of each tool. In the second stage, the LLM is employed for parameter extraction, identifying the precise inputs required to execute one or more tool calls with the detailed tool description. Crucially, both stages are driven by the LLM's reasoning capabilities, ensuring flexible and context-aware planning. When steps involve dependencies, the outputs from prior tool executions are fed back into the prompt for subsequent stages, allowing the LLM to dynamically reason over intermediate results. Similar to the stream mode in LLMCompiler, the two stages are executed in a pipelined manner, ensuring efficient and coherent planning. We illustrate this workflow in the following Example 3.1.

Example 3.1. Consider a set of physical machines, where the user's objective is to shut down those that are idle. Suppose the machine IPs are 10.20.30.1, 10.20.30.2, and 10.20.30.3. Figure 4 illustrates how the two-stage planning approach solves this problem.

The first stage is task decomposition using a LLM. The generated solution consists of two steps: Step 1 invokes the **getNcBasicInfo** tool to check whether each machine is idle; Step 2 calls the **submitOps** tool to submit shutdown requests for machines identified as idle.

Once Step 1 is generated, it immediately proceeds to the parameter extraction stage. Since the **getNcBasicInfo** tool only supports a single machine, the LLM-based parameter extractor generates three distinct parameter sets, one for each machine. Each parameter set is submitted to the executor as soon as it is generated.

Due to the dependency, the parameter extractor processes Step 2 after all three tool calls in Step 1 complete successfully. By analyzing

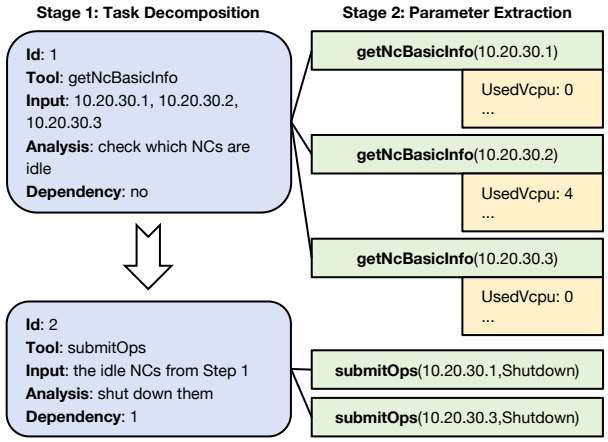


Figure 4: Example of two-stage planning

the *UsedVcpu* field in the returned results, it determines that only 10.20.30.1 and 10.20.30.3 are idle. Consequently, it generates two corresponding shutdown requests, which are then executed.

In contrast, if we directly plan the tool invocation parameters, it becomes difficult to determine during the planning phase how many times the **submitOps** tool needs to be invoked. Therefore, single-stage planning is not suitable for handling such problems with complex dependencies.

3.3 Fast Path

The fast path is designed to process the high-frequency scenarios. This section describes the techniques behind it.

3.3.1 Scenario Matching. Scenario matching is the cornerstone of the fast path. We maintain a curated set of high-frequency scenarios, each defined by the following fields:

- *Name*: A concise identifier for the scenario.
- *Description*: A brief explanation of the scenario's semantics and purpose.
- *Examples*: Representative input examples to aid in recognition.
- *Parameters*: Key variable elements involved, typically including resources and timestamps.

An LLM performs scenario matching by analyzing user input and identifying the best-fitting predefined scenario. It leverages semantic understanding to compare the input against the *Description* and *Examples* of each scenario.

To minimize latency and ensure compatibility with downstream systems, we use prompt engineering to constrain the LLM's output to solely the scenario name. This structured format enables efficient, programmatic processing. If no match is found, the LLM returns "None", thereby terminating the fast path.

3.3.2 Plan Generation. When a high-frequency scenario is detected, we generate an execution plan along with the corresponding tool parameters—effectively replicating the output of the two-stage planning process in the slow path. Since semantically similar user inputs often exhibit consistent plan patterns, we employ a template-based approach to enable efficient plan generation.

Each scenario is associated with a predefined plan template containing placeholders for dynamic variables. The *Parameters* field defines the type and format of these variables. We utilize the LLM to extract the required variable values from the user input and output them as a JSON dictionary. These extracted values are then instantiated into the template to produce the final execution plan. This approach enables fast and deterministic plan instantiation with minimal reliance on the LLM, thereby ensuring both efficiency and consistency.

3.3.3 Resource-based Scenario Pruning. As the number of predefined scenarios increases, injecting all scenario information into the LLM prompt leads to higher latency. Moreover, a larger candidate set can introduce semantic noise, reducing matching accuracy. To address these issues, we introduce resource-based scenario pruning—a preprocessing step that filters out irrelevant scenarios before invoking the LLM, thereby enhancing both efficiency and precision.

Resource IDs in user queries carry significant informational value and typically follow fixed, predictable formats, making them easily extractable via regular expressions. For example, physical machines are commonly identified by IPv4 addresses, while virtual machines are denoted by alphanumeric instance IDs prefixed with `i-`. Crucially, resource IDs frequently appear as key parameters in high-frequency scenarios.

Our pruning method operates as follows: for each high-frequency scenario, we analyze the *Parameters* field to identify expected resource ID types and attempt to extract corresponding IDs from the user query using regex patterns. If no matching ID is found, the scenario is discarded early. For instance, if the query contains no valid VM ID, scenarios such as "query the information of a VM" can be safely ruled out.

This pruning method is lightweight and incurs minimal computational overhead, as it relies solely on fast pattern matching without requiring LLM inference. Since resource ID extraction is highly reliable due to their fixed formats, the risk of incorrectly filtering relevant scenarios is negligible. As a result, the approach effectively reduces latency by shrinking the scenario candidate set and improves matching accuracy by eliminating semantic noise from irrelevant options. Example 3.2 illustrates how fast path works with pruning.

Example 3.2. Suppose there are three high-frequency scenarios: **VmDown**, which diagnoses the cause of VM downtime; **NcInfo**, which queries basic information about physical machines; and **SubmitOps**, which submits an operation action.

Given the question, "Why did the VM i-abcdefg reboot on October 13th, 2025?", as illustrated in Figure 5, we first extract the unique resource ID "i-abcdefg" from the query and compare it against the resource parameters (highlighted in red) of each scenario. Both the **VmDown** and **SubmitOps** scenarios accept this resource ID, whereas the **NcInfo** scenario requires an IP address—a parameter that does not match the key resource in the query. Hence, the **NcInfo** scenario is pruned.

Next, we employ a LLM to perform fine-grained scenario matching. The query is classified into the **VmDown** scenario, and the model is subsequently used to extract all relevant key parameters

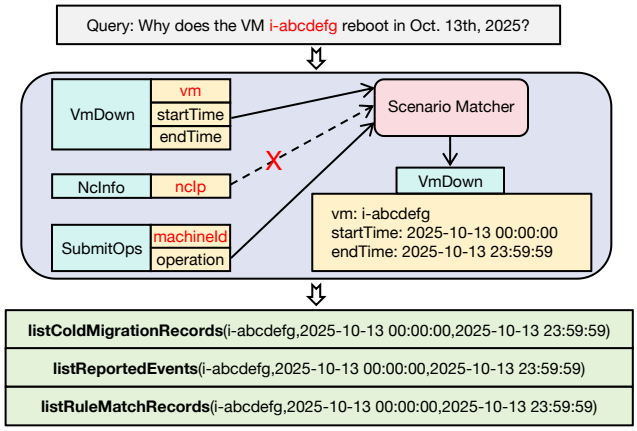


Figure 5: Example of fast path with pruning

for this scenario. These parameters are then injected into a predefined execution plan, triggering the parallel invocation of three tools.

The **listColdMigrationRecords** tool reports a successful cold migration, the **listReportedEvents** tool detects an unplanned reboot event, and the **listRuleMatchRecords** tool identifies a match with rule *cable_error_recover*. The timestamps of these three events align exactly. Therefore, the VM reboot was caused by a network cable failure and was automatically resolved via cold migration.

3.4 Dual-Path Concurrent Processing

To minimize end-to-end latency, the system executes the fast and slow paths concurrently rather than sequentially. This design ensures no additional latency penalty, even when the fast path misses.

Critically, when the fast path detects a high-frequency scenario, it triggers an early termination signal, halting ongoing LLM reasoning and tool calls in the slow path. In such cases, the fast path outputs only a brief scene name (1–2 tokens), while the slow path generates little to no output. Given that a single slow path invocation consumes approximately 3,000 input tokens, which cost less than $0.001, this overhead is negligible. This dynamic coordination between fast and slow paths achieves an optimal trade-off between responsiveness and generality, enabling low-latency inference without sacrificing coverage.

3.5 Discussion

3.5.1 Scenario Construction and Update. Our system supports the dynamic conversion of slow-path (low-frequency) scenarios into fast-path (high-frequency) ones. Based on structural similarity (measured by DAG edit distance[15]) and semantic similarity (quantified via cosine similarity of their embeddings[37]), slow-path execution plan DAGs could be offline clustered. Thus, when a "new" complex scenario is resolved frequently enough in the slow path, it becomes a cluster. Each cluster center represents a candidate scenario, which is distilled into a candidate fast path template and subsequently added to the scenario set after manual review.

To maintain effectiveness, we periodically analyze unmatched slow path executions, checking whether their DAGs resemble current fast path templates. Additionally, we regularly re-execute fast path templates against updated tools to detect opportunities for improved plans due to new tool capabilities. All changes are subject to human review to ensure correctness and safety.

In practice, the maintenance cost is low. In our experience, approximately 30 core high-frequency scenarios are maintained, each requires only 10 to 20 minutes of human effort for construction and validation. Through this adaptive loop, the system continuously integrates well-resolved new scenarios into the predefined fast path, endowing the overall mechanism with strong scalability and sustainability.

3.5.2 Generalization. Although DualLane was originally proposed in the context of cloud service AIOps, its architecture exhibits strong generalization capability. The key to successful adaptation lies in whether the target scenario satisfies two core conditions:

(1) **Structured resource as the center:** User issues are predominantly tied to well-defined resources that possess structured and standardized identifiers. Problem resolution relies on invoking domain-specific tools to retrieve or manipulate resource-related information, and accurate tool invocation—guided by correctly parsed resource IDs—forms the foundation for reliable automation.

(2) **Significantly aggregated query distribution:** The workload follows a pronounced long-tail pattern, where a small set of issue types accounts for the majority of requests. This phenomenon is commonly observed in large-scale systems with stable user behavior, consistent with the Pareto principle[9].

Indeed, beyond cloud computing, numerous domains such as industrial IoT (e.g., PLC communication failures or sensor malfunctions) naturally satisfy these conditions, enabling effective deployment of the DualLane architecture. However, in cases lacking clear resource references, such as software application AIOps involving code-level logic or end-to-end business flow anomalies, DualLane may fail to leverage its advantages.

4 Evaluation

4.1 Experiment Setup

4.1.1 Dataset. We collect real user queries directed to our DualLane system to form the test dataset. Due to factors such as data expiration and evolving system states, tool outputs may vary over time. To ensure experimental reproducibility, we preserve the outputs of all tools as part of the dataset. For each query, domain experts manually review and refine the system response to establish accurate expected results. The dataset contains 370 cases, spanning diverse scenarios including information retrieval, problem diagnosis, and resource operations.

Among these cases, 208 are easy ones that require only a single tool invocation. Another 84 are medium, involving multiple tool calls that are independent of one another, such as Example 3.2. The remaining 78 cases are hard, requiring multiple sequential tool calls with dependencies between them, such as Example 3.1.

4.1.2 Evaluation Metrics. We evaluate the performance with two main criteria: *latency* and *accuracy*.

Latency is measured from the moment a user question is received to key milestones in the response process. We define two latency metrics:

- *Total Response Latency*: the time from question receipt to the completion of the final answer generation.
- *Plan-Execution Latency*: the time from plan and tool execution, which also corresponds to the point when answer generation begins.

Accuracy measures how closely the actual response aligns with the expert-refined expected response. Since semantic flexibility is allowed, exact string matching is insufficient. Therefore, we employ *qwen3-max-2025-09-23* as a semantic verifier to determine if the agent output is semantically equivalent to the expected one.

We conducted a human audit on a random sample of 200 instances. A domain expert manually verified whether the LLM's binary judgments aligned with the true semantic equivalence. The results yielded an overall agreement rate of 98%. Crucially, all discrepancies (the 2% error rate) were False Negatives. This high reliability confirms that the LLM judge effectively replicates human judgment in assessing semantic equivalence while maintaining a safety margin against hallucinated correctness.

4.1.3 Baselines. We compare the proposed DualLane framework against two state-of-the-art baselines: ReAct and LLMCompiler. The configurations for all methods are detailed below to ensure a fair and consistent evaluation.

DualLane. The slow path and the concluder utilize *qwen3-235b-a22b*, while the fast path employs the lighter-weight *qwen-plus-2025-07-28*. DualLane has access to a total of 27 distinct tools, with the fast path specifically optimized for 25 predefined high-frequency scenarios to enable rapid responses.

LLMCompiler. We adopt the open-source implementation [5] and adapt it to our evaluation setup through prompt engineering. The backbone LLM is replaced with *qwen3-235b-a22b* to ensure both a fair comparison and strong data security. To evaluate performance in interactive settings, we disable its replanning mechanism. Stream mode is enabled by default, as it reduces latency without sacrificing accuracy. The framework has access to the same set of 27 tools as DualLane, along with the default *Join* tool for task coordination.

ReAct. Similarly, we use a modified version of the open-source implementation [5], with the underlying LLM also replaced by *qwen3-235b-a22b*. It operates with the identical set of 27 tools and follows the standard reasoning-and-acting paradigm.

4.1.4 Environment. The experiments are conducted within a containerized pod equipped with 4 CPU cores and 8 GB of memory. All language models are invoked exclusively through the DashScope API from Alibaba Cloud Model Studio [1] to ensure consistent deployment conditions and eliminate potential biases from local inference environments. Meanwhile, all models work in instruct mode to accelerate response generation.

4.2 Overall Evaluation

To demonstrate the advantages of DualLane over existing methods, we conducted a comprehensive performance evaluation of DualLane, ReAct, and LLMCompiler. Table 3 presents the average total response and plan-execution latency, as well as the accuracy of each method across queries of varying difficulty levels.

Table 3: Overall results of different methods

Difficulty	Method	Latency		Acc.
		Total Resp.	Plan-Exec.	
Easy	DualLane	**6.89s**	**3.69s**	**97.6%**
	LLMCompiler	13.15s	4.34s	84.6%
	ReAct	7.64s	4.33s	85.6%
Medium	DualLane	**13.69s**	**7.08s**	**97.5%**
	LLMCompiler	20.43s	9.64s	69.0%
	ReAct	18.36s	14.23s	71.4%
Hard	DualLane	**13.06s**	7.29s	**92.3%**
	LLMCompiler	14.28s	**5.88s**	64.1%
	ReAct	14.77s	8.92s	83.3%
All	DualLane	**9.73s**	**5.22s**	**96.5%**
	LLMCompiler	15.04s	5.87s	76.7%
	ReAct	11.58s	7.55s	81.9%

4.2.1 Accuracy. Overall, ReAct achieves slightly higher accuracy than LLMCompiler, while DualLane significantly outperforms both across all difficulty levels. We now analyze the sources of this performance gap.

Context Management. ReAct and LLMCompiler embed full descriptions of all 27 tools directly into the system prompt, resulting in excessively long contexts that can overwhelm the model. For easy problems, errors often manifest as incorrect tool selection, improper parameter formatting, or missing parameters. For medium and hard problems, in addition to these issues, errors also arise from incomplete tool invocation or incomplete conclusion. For instance, when querying migration records, the model may only retrieve cold migration records while neglecting to query live migration records, or it may overlook a tool that has already been invoked during the concluding phase. In contrast, DualLane adopts a two-stage planning approach that decouples tool selection from parameter generation. By isolating these decisions, DualLane reduces context length at each stage, minimizing noise and cognitive load.

Fast Path. DualLane incorporates a fast path that leverages human-verified, pre-defined plans for high-frequency, routine scenarios, ensuring consistent and reliable performance. Neither ReAct nor LLMCompiler employs a similar shortcut mechanism, requiring full generative planning even for simple or repetitive tasks, which increases the likelihood of avoidable mistakes and inefficiencies.

Dependency. For hard problems involving complex inter-step dependencies, LLMCompiler's placeholder-based strategy proves inadequate. Subsequent steps fail to correctly reference the results of preceding steps through simple placeholders, as illustrated in Example 3.1. This limitation severely degrades its accuracy in multi-step reasoning. Considering that ReAct performs sequential planning and DualLane utilizes two-stage planning to deal with dependencies, resulting in a significant accuracy gap between LLMCompiler and these methods on hard problems.

Table 4: Overall results of different variants

Difficulty	Var.	Latency		Acc.	Hit Rate
		Total Resp.	Plan-Exec.		
Easy	I	6.89s	3.69s	**97.6%**	95.2%
	II	**6.51s**	**3.67s**	94.7%	95.6%
	III	9.08s	5.95s	89.9%	-
	IV	6.82s	4.17s	87.0%	-
Medium	I	13.69s	7.08s	**97.5%**	89.3%
	II	**12.29s**	**6.56s**	94.0%	91.7%
	III	20.75s	14.47s	89.3%	-
	IV	16.06s	10.76s	71.4%	-
Hard	I	13.06s	7.29s	**92.3%**	26.9%
	II	14.01s	7.58s	89.7%	28.2%
	III	13.53s	7.70s	**92.3%**	-
	IV	**12.82s**	**5.89s**	64.1%	-
All	I	9.73s	5.22s	**96.5%**	79.5%
	II	**9.40s**	**5.15s**	93.5%	80.5%
	III	12.67s	8.25s	90.3%	-
	IV	10.18s	6.03s	78.6%	-

4.2.2 Latency. On average, DualLane speedups end-to-end response by 19.0% compared to ReAct and by 54.6% compared to LLMCompiler. Given that the concluder prompt in LLMCompiler often produces lengthy outputs, we place greater emphasis on plan-execution time. In this regard, DualLane is 44.6% faster than ReAct and 12.5% faster than LLMCompiler.

The latency advantage of DualLane primarily stems from its fast path mechanism. For easy and medium problems, the fast path achieves a high hit rate, leading to significant performance gains over both ReAct and LLMCompiler. This improvement is particularly evident in medium tasks that require multiple tool calls, where DualLane's efficiency is further amplified.

However, most hard problems cannot be resolved through the fast path and instead require full reasoning. In such cases, LLMCompiler demonstrates the lowest latency, benefiting from its ability to directly generate tool parameters and support parallel execution.

4.3 Ablation Study

To better understand the contribution of each component in our proposed method, we conduct ablation studies on the following four key variants of DualLane:

- **Variant I**: Both slow and fast paths are enabled—this is equivalent to the original DualLane.
- **Variant II**: Both slow and fast paths are enabled, but high-frequency scenarios are not pruned based on resources.
- **Variant III**: The fast path is disabled; all queries are processed through the slow path.
- **Variant IV**: The fast path is disabled, and the slow path adopts a single-stage planning approach, where tool invocation parameters are directly predicted, with dependency relationships managed via placeholder variables.

Table 4 presents the performance of these variants in terms of average latency, accuracy and fast path hit rate.

4.3.1 Accuracy. Variant I (the original DualLane) achieves the highest accuracy. As components are incrementally removed, accuracy consistently decreases, demonstrating that each designed component, including resource-based pruning, fast path and two-stage planning, plays a critical role in maintaining high accuracy.

For easy problems, Variant II shows reduced accuracy as the absence of pruning expands the fast path matching space. Variant III sees a sharper decline by disabling the fast path entirely. Without the precise descriptions and few-shot examples of fast path scenarios, the slow path suffers from a higher rate of incorrect tool selections. Finally, Variant IV degrades accuracy by directly generating parameters, a process that requires full tool descriptions. This significantly bloats the context window, thereby reducing accuracy.

Notably, Variant IV for medium and hard problems suffers significant accuracy degradation, as single-stage planning is ill-suited for handling complex task dependencies (just like LLMCompiler). Meanwhile, Variant III does not degrade accuracy for hard problems. This is because most hard problems do not trigger the fast path; therefore, disabling the fast path mechanism has a negligible impact on the final outcome.

4.3.2 Latency. Variants I and II demonstrate significantly lower latency than Variants III and IV, attributed to the activation of the fast path, which validates its efficacy in accelerating inference. Specifically, for queries routed through the fast path in Variant I, we evaluate their latency against a forced slow-path execution (Variant III). This comparison reveals that leveraging the fast path reduces total response latency from 11.95s to 8.27s (44.5% speedup) and slashes plan-execution latency from 8.15s to 4.39s, yielding a substantial 85.6% improvement.

When comparing Variant I and Variant II, the pruned fast path in Variant I results in marginally higher latency than the unpruned version in Variant II. This occurs because the unpruned fast path generates more false positives (i.e., incorrect scenario matches), which paradoxically lowers the average measured latency.

Furthermore, for hard problems, Variant IV achieves the lowest latency. This aligns with observations in LLMCompiler system: when the problems are unsuitable for fast path, single-stage planning incurs less overhead than the two-stage alternative.

4.4 Real-World Deployment Evaluation

Based on the DualLane architecture described in Section 3, we deployed an AIOps agent in the production environment of Alibaba Cloud ECS. Built on LangGraph [8] and integrated with LangFuse [7] for comprehensive trace observability, the agent enables engineers to interactively query system information, diagnose issues, and execute operational tasks across both virtual and physical machines. Interaction is supported through multiple channels, including DingTalk bots [3] and web interfaces. Appendix A shows the details of implementations.

We conducted a 30-day online evaluation following deployment. During this period, the agent processed over 18,000 user requests. Table 5 presents the human-evaluated results and the distribution of failure modes. The intrinsic error rate attributable to the DualLane

Table 5: Distribution of failure modes in the real world

Category	Percentage	Type
Correct	73.9%	-
Uncovered knowledge	9.7%	OOD
Unsupported tools	6.4%	OOD
Incorrect or incomplete conclusion	3.4%	Internal
MCP server anomaly	2.9%	Infra
Failure in tool selection	2.8%	Internal
Incorrect tool parameters	0.5%	Internal
Broken inter-step dependency	0.4%	Internal

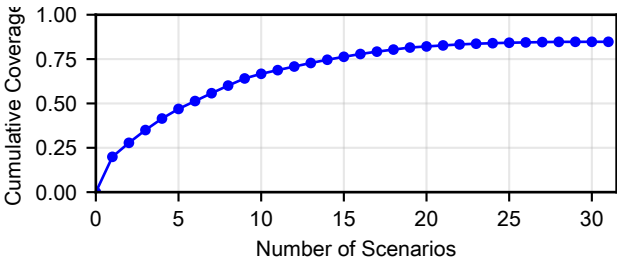


Figure 6: Coverage CDF of fast path scenarios

logic itself is only 7.1%, comparable to the offline results. A primary source of inaccuracies was out-of-scope queries—e.g., questions referencing knowledge not covered by the agent's knowledge base or involving unsupported tools. To handle OOD queries, we use a tiered strategy:

- **Out-of-domain**: We restricted the domain scope in the prompt, guiding the LLM to actively reject them. Additionally, we treated out-of-domain questions as a fast-path scenario for pattern matching.
- **In-domain gaps**: Queries within our scope but lacking current tool/knowledge support are flagged and routed to human experts.

Meanwhile, Figure 6 shows the coverage of fast path scenarios. The top 5 scenarios accounted for 47.0% of queries, and the top 10 covered 66.7%. Overall, 84.8% of all requests were handled via the fast path, demonstrating the efficiency and effectiveness of the DualLane architecture.

We also measured system latency under real-world conditions. For plan-execution latency, the median was 4.2s, with the 90th percentile at 10.1s. While for total response latency, it was 8.5s and 18.4s, respectively. These results indicate that the system maintains responsive performance, effectively preventing users from experiencing prolonged waiting times.

5 Related Work

5.1 Information Integration in LLM Era

Information integration is a highly classical research topic. Traditional approaches suffer from various limitations, including generalization, accuracy, robustness, and so on. To overcome these limitations, pre-trained LLM, which encode vast amounts of world knowledge and contextual understanding, are considered a promising direction [23, 36].

A growing body of research explores the application of LLMs across various stages of the information integration pipeline. In data cleaning, for example, LLMClean [10] employs LLMs to automatically extract data dependencies and construct context models, while RetClean [13] enhances cleaning performance with data lake-based RAG (Retrieval Augmented Generation) and LLM. LLMs are also applied to schema matching [24] and end-to-end data integration [17]. Furthermore, their ability to translate natural language into SQL queries [14] or support multimodal retrieval [33] enables more intuitive and user-friendly interfaces.

In this proposal, components including databases and APIs are abstracted and encapsulated as tools. Based on dual-path planning, it achieves fast and accurate information integration by invoking tools and integrating their results. It effectively meets the responsiveness and precision demands of interactive applications.

5.2 Planning of LLM Agent

Planning capability is crucial for LLM agents. Some approaches focus on a single potential path: for instance, Decomposed Prompts [18] explicitly break down a complex problem into several sub-problems and solve them individually, while Chain-of-Thought (CoT) [20] enhances prompts to elicit step-by-step reasoning from the model. Furthermore, to enable exploration over multiple reasoning paths, Tree-of-Thought (ToT) [34] constructs a tree-structured search space that allows parallel planning and backtracking. However, its high computational cost limits applicability.

ReAct [35] integrates tool usage with reasoning, dynamically planning the next step based on environmental observations (e.g., tool outputs). Commercial agents like Claude Code [2] and iFlow [6] adopt similar paradigms, maintaining an explicit to-do list to ensure execution fidelity. To improve efficiency, LLMCompiler [19] models inter-step dependencies as a DAG to enable parallelized tool invocation.

Building upon this, our DualLane achieves dual-path planning: it accelerates high-frequency scenarios through a fast path with semantic caching, and optimizes dependency handling via a slow path that employs two-stage planning. This design balances speed and accuracy, making it well-suited for interactive AIOps applications.

6 Conclusion

This paper presents DualLane, an LLM agent architecture designed for interactive AIOps. Inspired by the existence of high-frequency scenarios, we propose the dual-path planning architecture: a fast path to optimize and accelerate processing in high-frequency scenarios, and a slow path to handle long-tail cases. Experimental results demonstrate that, compared to existing approaches, DualLane achieves significant improvements in both accuracy and latency.

Acknowledgments

We thank all reviewers and chairs for their constructive comments, as well as our colleagues at Alibaba Cloud who served as early users and offered valuable optimization suggestions. Haoyu Wang (https://wanghy.pages.dev/) is the corresponding author.

References

[1] 2025. Alibaba Cloud Model Studio. https://www.alibabacloud.com/help/en/model-studio/

[2] 2025. Claude Code. https://claude.com/product/claude-code

[3] 2025. DingTalk. https://www.dingtalk.com/en

[4] 2025. Elastic Compute Service (ECS) - Alibaba Cloud. https://www.alibabacloud.com/en/product/ecs

[5] 2025. GitHub - SqueezeAILab/LLMCompiler. https://github.com/SqueezeAILab/LLMCompiler

[6] 2025. iFlow. https://platform.iflow.cn/cli/quickstart

[7] 2025. LangFuse. https://langfuse.com/

[8] 2025. LangGraph. https://www.langchain.com/langgraph

[9] Chris Anderson. 2006. *The Long Tail.* Hyperion.

[10] Fabian Biester, Mohamed Abdelaal, and Daniel Del Gaudio. 2024. LLMClean: Context-Aware Tabular Data Cleaning via LLM-Generated OFDs. In *New Trends in Database and Information Systems - ADBIS 2024 Short Papers, Workshops, Doctoral Consortium and Tutorials, Bayonne, France, August 28-31, 2024, Proceedings (Communications in Computer and Information Science, Vol. 2186)*, Joe Tekli, Johann Gamper, Richard Chbeir, Yannis Manolopoulos, Salma Sassi, Mirjana Ivanovic, Genoveva Vargas-Solar, and Ester Zumpano (Eds.). Springer, 68–78. doi:10.1007/978-3-031-70421-5_7

[11] Yingnong Dang, Qingwei Lin, and Peng Huang. 2019. Aiops: real-world challenges and research innovations. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 4–5.

[12] Lingfei Deng, Yunong Wang, Haoran Wang, Xuhua Ma, Xiaoming Du, Xudong Zheng, and Dongrui Wu. 2024. Time-Aware Attention-Based Transformer (TAAT) for Cloud Computing System Failure Prediction. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2024, Barcelona, Spain, August 25-29, 2024*, Ricardo Baeza-Yates and Francesco Bonchi (Eds.). ACM, 4906–4917. doi:10.1145/3637528.3671547

[13] Mohamed Y. Eltabakh, Zan Ahmad Naeem, Mohammad Shahmeer Ahmad, Mourad Ouzzani, and Nan Tang. 2024. RetClean: Retrieval-Based Tabular Data Cleaning Using LLMs and Data Lakes. *Proc. VLDB Endow.* 17, 12 (2024), 4421–4424. doi:10.14778/3685800.3685890

[14] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. *Proc. VLDB Endow.* 17, 5 (2024), 1132–1145. doi:10.14778/3641204.3641221

[15] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. 2010. A survey of graph edit distance. *Pattern Analysis and applications* 13, 1 (2010), 113–129.

[16] Peng Huang, Chuanxiong Guo, Jacob R Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 1–16.

[17] Moe Kayali, Fabian Wenz, Nesime Tatbul, and Çagatay Demiralp. 2024. Mind the Data Gap: Bridging LLMs to Enterprise Data Integration. *CoRR* abs/2412.20331 (2024). arXiv:2412.20331 doi:10.48550/ARXIV.2412.20331

[18] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2023. Decomposed Prompting: A Modular Approach for Solving Complex Tasks. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=_nGgzQjzaRy

[19] Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. 2024. An LLM Compiler for Parallel Function Calling. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net. https://openreview.net/forum?id=uQ2FUoFjnF

[20] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large Language Models are Zero-Shot Reasoners. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html

[21] Sebastien Levy, Randolph Yao, Youjiang Wu, Yingnong Dang, Peng Huang, Zheng Mu, Pu Zhao, Tarun Ramani, Naga Govindaraju, Xukun Li, et al. 2020. Predictive and Adaptive Failure Mitigation to Avert Production Cloud {VM} Interruptions. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1155–1170.

[22] Guoliang Li. 2017. Human-in-the-loop data integration. *Proceedings of the VLDB Endowment* 10, 12 (2017), 2006–2017.

[23] Guoliang Li, Xuanhe Zhou, and Xinyang Zhao. 2024. LLM for Data Management. *Proc. VLDB Endow.* 17, 12 (2024), 4213–4216. doi:10.14778/3685800.3685838

[24] Xuanqing Liu, Runhui Wang, Yang Song, and Luyang Kong. 2024. GRAM: Generative Retrieval Augmented Matching of Data Schemas in the Context of Data Security. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2024, Barcelona, Spain, August 25-29, 2024*, Ricardo Baeza-Yates and Francesco Bonchi (Eds.). ACM, 5476–5486. doi:10.1145/3637528.3671602

[25] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, et al. 2023. Perseus: A {Fail-Slow} Detection Framework for Cloud Storage Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 49–64.

[26] Xianting Lu, Yunong Wang, Yu Fu, Qi Sun, Xuhua Ma, Xudong Zheng, and Cheng Zhuo. 2024. MISP: A Multimodal-based Intelligent Server Failure Prediction Model for Cloud Computing Systems. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2024, Barcelona, Spain, August 25-29, 2024*, Ricardo Baeza-Yates and Francesco Bonchi (Eds.). ACM, 5509–5520. doi:10.1145/3637528.3671568

[27] Markos Markakis, Brit Youngmann, Trinity Gao, Ziyu Zhang, Rana Shahout, Peter Baile Chen, Chunwei Liu, Ibrahim Sabek, and Michael J. Cafarella. 2024. From Logs to Causal Inference: Diagnosing Large Systems. *Proc. VLDB Endow.* 18, 2 (2024), 158–172. https://www.vldb.org/pvldb/vol18/p158-markakis.pdf

[28] Laxmi Rijal, Ricardo Colomo-Palacios, and Mary Sánchez-Gordón. 2022. Aiops: A multivocal literature review. *Artificial Intelligence for Cloud and Edge Computing* (2022), 31–50.

[29] Mufeed Ahmed Naji Saif, SK Niranjan, and Hasib Daowd Esmail Al-Ariki. 2021. Efficient autonomic and elastic resource management techniques in cloud environment: taxonomy and analysis. *Wireless Networks* 27, 4 (2021), 2829–2866.

[30] Haoyu Wang, Hongke Guo, Zhaoliang Zhu, You Zhang, Yu Zhou, and Xudong Zheng. 2024. BacktrackSTL: Ultra-Fast Online Seasonal-Trend Decomposition with Backtrack Technique. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2024, Barcelona, Spain, August 25-29, 2024*, Ricardo Baeza-Yates and Francesco Bonchi (Eds.). ACM, 5848–5859. doi:10.1145/3637528.3671510

[31] Haoyu Wang, Zhicheng Liu, Yeliang Qiu, Haozhe Li, Hongke Guo, Zhaoliang Zhu, You Zhang, Yu Zhou, and Xudong Zheng. 2025. Stability is Not Downtime: Comprehensive Stability Evaluation for Large-Scale Cloud Servers in Alibaba Cloud. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19-23, 2025*. IEEE, 4169–4182. doi:10.1109/ICDE65448.2025.00311

[32] Haoyu Wang, Aoqian Zhang, Shaoxu Song, and Jianmin Wang. 2024. Streaming data cleaning based on speed change. *VLDB J.* 33, 1 (2024), 1–24. doi:10.1007/S00778-023-00796-Y

[33] Mengzhao Wang, Haotian Wu, Xiangyu Ke, Yunjun Gao, Xiaoliang Xu, and Lu Chen. 2024. An Interactive Multi-modal Query Answering System with Retrieval-Augmented Large Language Models. *Proc. VLDB Endow.* 17, 12 (2024), 4333–4336. doi:10.14778/3685800.3685868

[34] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/271db9922b8d1f4dd7aaef84ed5ac703-Abstract-Conference.html

[35] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=WE_vluYUL-X

[36] An Zhang, Yang Deng, Yankai Lin, Xu Chen, Ji-Rong Wen, and Tat-Seng Chua. 2024. Large Language Model Powered Agents for Information Retrieval. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2024, Washington DC, USA, July 14-18, 2024*, Grace Hui Yang, Hongning Wang, Sam Han, Claudia Hauff, Guido Zuccon, and Yi Zhang (Eds.). ACM, 2989–2992. doi:10.1145/3626772.3661375

[37] Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, et al. 2025. Qwen3 Embedding: Advancing Text Embedding and Reranking Through Foundation Models. *arXiv preprint arXiv:2506.05176* (2025).

A Deployment Details

A.1 Design and Implementation of MCP Tools

Model Context Protocol (MCP) tools go beyond simple API wrappers—they are intelligent, resource-centric components designed to streamline and enhance interactions with external systems. To maximize usability and consistency, tool interfaces are systematically optimized to eliminate redundancy, enforce standardized parameter naming, and leverage contextual dependencies. For example, when querying a VM's cold migration records, the original API requires both the VM ID and its region. However, since a VM is inherently located within a single region, the region is redundant when the VM is provided. In the MCP tool, the region parameter is removed and automatically inferred from the VM, simplifying the interface and reducing the likelihood of input errors.

We enhance MCP tool outputs by adding descriptive annotations for each field to improve interpretability by LLMs. For fields that are inherently ambiguous—such as exception names—we include supplementary explanation fields. Additionally, we address suboptimal design choices in the original APIs. For instance, the *LockReason* field is not automatically cleared, causing it to persist in responses even when the machine is unlocked. This can mislead LLMs into inferring an incorrect state, so we rectify such inconsistencies to ensure accurate and reliable output.

A.2 Safety, Security, and Robustness

To ensure the secure and reliable operation in production environments, we implement a multi-layered protection mechanism:

Human-in-the-loop for Critical Operations. All operational tools (e.g., migration, restart) require manual confirmation and are prohibited from automatic execution, ensuring that critical actions remain under strict control.

Fine-Grained Access Control. Tool access is restricted based on roles. For low-privilege users, sensitive tools are disabled and sensitive fields are masked to prevent leakage of confidential information (e.g., operation rules).

Strict Parameter Validation. Incorrect outputs are more harmful than outright failures. LLMs may hallucinate critical parameters (e.g., VM IDs), leading to errors. To prevent this, we use regex matching to verify that such parameters explicitly appear in the prior context; otherwise, the tool call is rejected immediately, blocking error propagation.