DBAgent: An RL-Based Agent for Autonomous Database Operations and Maintenance

Xu Chen*
University of Electronic Science and Technology of China, China
xuchen@std.uestc.edu.cn

Junming Chen*
University of Electronic Science and Technology of China, China
junmingchen@std.uestc.edu.cn

Shuncheng Liu
Huawei Technologies Co., Ltd., China
liushuncheng1@huawei.com

Yuyang Xia
University of Electronic Science and Technology of China, China
xiayuyang@std.uestc.edu.cn

Zibo Liang
University of Electronic Science and Technology of China, China
zbliang@std.uestc.edu.cn

Xiaoming Qin
Huawei Technologies Co., Ltd., China
qinxiaoming5@huawei.com

Kai Zeng†
Huawei Technologies Co., Ltd., China
kai.zeng@huawei.com

Han Su
University of Electronic Science and Technology of China, China
hansu@uestc.edu.cn

Kai Zheng†
University of Electronic Science and Technology of China, China
zhengkai@uestc.edu.cn

ABSTRACT

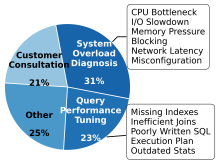
Database Operations and Maintenance (O&M) is a critical but complex and labor-intensive task. Recent LLM-based assistants promise to lower the barrier by reading manuals/tickets and exploring diagnostic search trees. However, existing LLM-based solutions fall short due to fundamental limitations in learning from expert demonstration. Such design fails to internalize domain dynamics (how interventions change plans, resources, etc.), and struggles under workload and statistics drift. To address this, we present DBAgent, an autonomous agent for Huawei Cloud Data Warehouse Service (DWS) integrated with Autopilot (DWS 's production monitoring, alerting, and auto-remediation service). DBAgent consumes DWS telemetry (e.g., KPIs and execution plans) and Autopilot alerts to diagnose and remediate incidents in production clusters. DBAgent emulates an expert's iterative *Think–Act–Observe* problem-solving loop with a policy trained via reinforcement learning (RL). It couples dynamic tool use for information gathering, a multimodal perception module for database-native signals, and an RL-based reasoning engine that plans, verifies, and generates evidence-grounded remediation recommendations. Extensive experiments show that DBAgent handles a broad range of complex O&M tasks. It surpasses the strongest baseline by +23% success rate and produces evidence-grounded reports with low hallucination (~5%) on DWS benchmark.

PVLDB Reference Format:
Xu Chen, Junming Chen, Shuncheng Liu, Yuyang Xia, Zibo Liang, Xiaoming Qin, Kai Zeng, Han Su, and Kai Zheng. DBAgent: An RL-Based Agent for Autonomous Database Operations and Maintenance. PVLDB, 19(12): 3928 - 3940, 2026.
doi:10.14778/3827998.3828006

*Xu Chen and Junming Chen contributed equally to this work.
†Kai Zheng and Kai Zeng are corresponding authors.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.



(a) DBA Requests Breakdown

DBA Task	ADT	Tools (No.)	MCC
Query Performance Tuning	8.4	6.2	27
System Overload Diagnosis	7.1	5.6	32
Routine Health Check	5.2	4.3	19
Client Consultation	3.5	1.8	N/A

(b) Analysis of Task Complexity on O&M Tasks

Figure 1: Understanding the landscape of DBA operations: (a) Breakdown of diverse DBA request types; (b) Analysis of Avg. Diagnostic Turns (ADT), Tools Invoked (Tools), Metric Columns Correlated (MCC) for complex DBA O&M tasks.

1 INTRODUCTION

Modern cloud data platforms increasingly hinge on productized, end-to-end operations. *Huawei Cloud Data Warehouse Service (DWS)*, a cloud-native, distributed database, supports business-critical analytics and services, where sustained performance and availability depend on rigorous Operations and Maintenance (O&M). Within DWS, *Autopilot* serves as the operational backbone, consolidating telemetry, KPI dashboards (e.g., system and workload indicators), and alerting across clusters. Yet transforming these signals into correct and timely actions remains challenging: practitioners must reason about the interplay of DWS 's architecture and resource policies, evolving multi-tenant workloads, and dynamic execution contexts. They must perform root-cause analysis, tune queries and plans, and recommend safe configuration changes manually. These product-level constraints jointly shape system behavior and performance, making O&M in DWS both high-stakes and intrinsically complex, and motivating intelligence that can understand, reason and act within the DWS environment.

Manual O&M. Traditionally, Database Administrators (DBAs) have been responsible for handling these critical operational tasks. An analysis of internal data from DWS reveals the demanding landscape of their work. As shown in Figure 1(a), the O&M works are remarkably diverse, spanning system overload diagnosis (31% of

doi:10.14778/3827998.3828006

requests), query performance tuning (23%), and customer consultation (21%). This breadth requires DBAs to possess deep domain knowledge across various aspects of the database system, from resource contention issues like I/O slowdown to the intricacies of SQL execution plans. Beyond their diversity, these responsibilities are complex and time-consuming. This nature is a direct result of the laborious diagnostic process DBAs must undertake. These tasks typically require multiple turns of reasoning and action, frequent invoking of tools, and correlation of heterogeneous signals, which is illustrated in Figure 1(b). For example, query tuning averages 8.4 turns, 6.2 tools, and 27 correlated metrics.

Autonomous O&M. Motivated by the cost and latency of manual operations, recent work seeks to automate O&M with data-driven assistants. Recent deep models analyze high-dimensional metrics [8, 21, 25, 26] and fuse heterogeneous signals [18, 28]. However, they are typically task-specialized (e.g., root-cause classifiers), limiting generalization across diverse O&M scenarios.

To enable more comprehensive automation and lower the operational threshold, recent systems introduce LLMs as agents or as a general reasoning backbone [3, 7, 32, 40]. These approaches center on injecting domain knowledge into pretrained LLMs and fall into two families: (i) **RAG-grounded agents** [3, 7, 32], which retrieve manuals, documentation, and tickets at inference time to ground decisions; and (ii) **tree-search agents** which orchestrate diagnosis through *LLM-guided tree search* rather than free-form reasoning: at each step, the LLM scores and expands candidate nodes drawn from a fixed library of templated subtasks and tool calls. D-Bot [40], exemplifies this design, coupling retrieval/tool access with LLM-steered expansion and relying on periodic "reflection" (typically once after each subtask). While LLMs lower the operational threshold, this paradigm constrains the agent's diagnostic behavior space. It remains bounded by the scoring heuristics and the templated action set, yielding rigid behavior under distribution shift and novel incidents that require sustained, adaptive reasoning.

In essence, the fundamental idea of current O&M agents is to learn from *expert demonstrations*—either by injecting retrieved knowledge or by conducting expert-designed tree search. This paradigm has two core limitations.

Limitation (1). Demonstration-driven behavior tends to overfit seen patterns and suppress exploration of alternative remediation sequences or trade-offs. For a slow query, for instance, a scripted recipe may default to (a) increasing `work_mem` or (b) creating an index. On the one hand, higher `work_mem` can reduce spill but risks memory pressure under concurrency. On the other hand, an index may speed up the target query yet degrade write-heavy workloads. Therefore, remediation is not a single step but an evidence-conditioned sequence whose order and branching depend on live signals (e.g., `pg_stats`, `EXPLAIN ANALYZE`, wait events). Expert rules canonize one or two default moves, ignore order sensitivity and trade-offs, and thus fail to adapt when the underlying cause demands a different sequence of diagnostics and fixes.

Limitation (2). Executing O&M tasks (e.g., adjusting `work_mem` or issuing `VACUUM`) reveals how each change affects the system. We refer to each such change as an *action* by the agent, and the resulting outcomes expose *action–effect* regularities. However, existing agents do not internalize these learnable regularities. For example, issuing `VACUUM FULL` during peak traffic can amplify locks and stall short-latency workloads; repeatedly forcing hash joins under key skew inflates spill and I/O wait. Internalizing these dynamics helps in two ways. The agent can predict side effects before making a change and adjust the order of hypotheses when new evidence arrives. It also carries experience across incidents, enabling transfer to new cases that share similar patterns. By contrast, expert rules cannot cover the combinatorial interactions of configs, data drift, and cache state, many of which are context-dependent.

Therefore, an effective O&M decision strategy must *internalize and adapt* these dynamics (e.g., `VACUUM FULL` → lock contention) and *explore* beyond demonstrated paths, guided by live operational feedback. Moreover, recent evidence suggests that context alone is insufficient: even with explicit domain knowledge in-context, frontier LMs often fail to apply it reliably [10].

We argue that the key is to *learn from experience*, i.e., training the decision strategy on the outcomes of its own actions, not just on demonstrations. Through multi-step diagnostic trials, the agent explores alternative action sequences and uses feedback to quantify their trade-offs. This reduces overfitting to scripted recipes. The agent also internalizes action–effect regularities, e.g., raising `work_mem` reduces spill but increases memory pressure. Then it adjusts its behavior based on feedback and expected consequences.

To this end, we present `DBAgent`, an LLM-based O&M agent driven by experience learning. We make two designs:

Design (1). Database O&M is not a single, linear task, but an iterative cycle of forming hypotheses, conducting investigations, and verifying findings. Therefore, we construct `DBAgent` as a multi-round interactive agent to unify O&M tasks and emulate this loop. The agent's core operational loop is defined by a Think-Act-Observe cycle, inspired by ReAct [35]. We give an example as shown in Figure 2. Faced with an operational issue (e.g., "Diagnose query performance"), the agent first Thinks, generating an internal hypothesis and a corresponding action plan (e.g., "I need to check for system resource contention by examining CPU and I/O metrics"). This thought drives an Action, such as invoking a tool to query the relevant metrics from the database. The tool's output becomes the Observation, which the agent then uses to inform its next thought, either to drill down further or to pivot to a new hypothesis.

Design (2). Although we adopt the powerful LLM as the core reasoning engine, making the Think-Act-Observe cycle practical is non-trivial. We operationalize the O&M ReAct framework with three cooperating modules: (1) **Knowledge & Tools Layer** exposes both static domain facts and live context through a unified interface. We distill manuals and tickets into a structured knowledge base of key facts and causal links. And we wrap low-level system commands (metric/log queries, `EXPLAIN ANALYZE`, config reads) as safe, high-level tool APIs so the agent can retrieve evidence at any step. (2) **Perception Layer** turns raw tool outputs into summaries tailored to data modality, including robust spike/trend summaries for time-series KPIs and compact plan analysis from `EXPLAIN`. So downstream reasoning consumes clean signals rather than verbose artifacts. (3) **RL-based Reasoning** provides the control logic. It proposes a hypothesis and plan, selects the next probe or remedy, incorporates observations to revise the hypothesis, and justifies the final decision with gathered evidence. We utilize Reinforcement

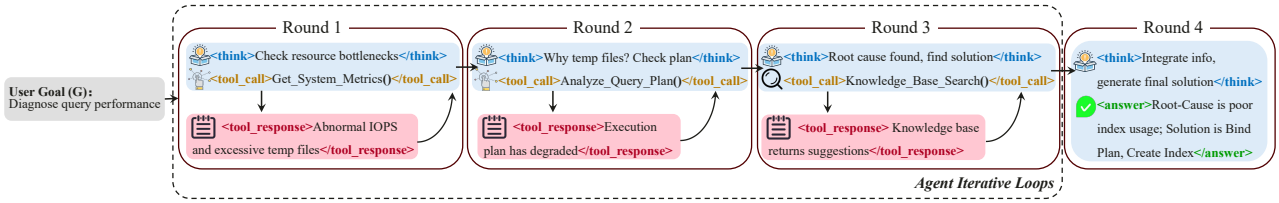


Figure 2: An example of a DBAgent reasoning chain.

Learning (RL) to optimize control logic based on rule-based feedback. Our RL algorithm encourages informative probing, factual grounding, and consistent long-horizon behavior. Together, these modules realize an O&M agent that executes general-purpose diagnostic and remediation processes without pre-scripted rules and learns from experience.

To conclude, we make the following contributions:

- We propose DBAgent, a deployed O&M agent integrated into DWS Autopilot, which provides unified support for query tuning, system diagnosis, health checks, and consultation.
- We design a curated knowledge and tool layer and database-signal perception layer for heterogeneous database-native signals.
- We model O&M as a Think–Act–Observe interaction process and train a tool-using reasoning policy with RL and rule-based process rewards distinguishing our approach from RAG-grounded and tree-search agents.
- We construct a DWS-derived replayable benchmark, compare against strong baselines and report real-world deployment results.

2 RELATED WORK

Autonomous Database Diagnosis. Autonomous database diagnosis has evolved from rule-based expert systems [2, 9, 14, 16, 24, 36] to ML/DL-based methods that detect or attribute anomalies from database metrics and heterogeneous signals [7, 8, 13, 18, 21, 25–28]. These methods are effective for specific diagnosis tasks, but are often designed as detectors or classifiers and require task-specific features or labels. Recent LLM-based systems further introduce retrieval and tool use for database or cloud diagnosis, including Panda [32], RCACopilot [7], and D-Bot [40].

Panda and RCACopilot mainly ground a diagnostic inference with retrieved documents, logs, traces, or metric snapshots, while D-Bot is closer to our setting because it performs iterative diagnosis with tools through an LLM-guided search tree. DBAgent differs in its control mechanism: both KB retrieval and diagnostic tools are selected as actions by a unified Think–Act–Observe policy, and the policy is optimized with RL feedback from interaction outcomes rather than only following a fixed retrieval prompt or a hand-designed subtask tree.

Autonomous Configuration Tuning. Autonomous configuration tuning optimizes database parameters for predefined objectives such as latency or throughput. Representative systems include ML-based tuning such as OtterTune [33], Bayesian optimization methods [12, 38], and RL-based tuning systems [23, 37]. These systems are complementary to DBAgent: they mainly optimize configuration choices under a target workload, whereas DBAgent focuses on diagnosing diverse natural-language O&M requests, collecting evidence through tools, and producing grounded root-cause and remedy reports. Recent LLM-based tools such as Andromeda [3] use RAG over technical documents for configuration debugging, while DBAgent targets broader O&M tasks.

3 PRELIMINARIES

3.1 Database Operation and Maintenance

The daily operation of DWS involves specialized O&M tasks that sustain performance, ensure high availability, and preserve system reliability. Based on the distribution in Figure 1(a), we group DBA workloads into four representative categories:

- **Routine Database Inspection.** Proactively verify cluster health by monitoring node-level CPU, memory, storage, and I/O, and by querying system views to uncover latent risks (e.g., data skew, high dirty-page rates) before they impact service.
- **Query Performance Tuning.** Optimize distributed SQL execution by analyzing query plans to mitigate performance bottlenecks. A common workflow is to use a `Top SQL` view to locate slow queries, then inspect their execution plans for inefficiencies like costly data redistribution between nodes.
- **System Overload Diagnosis.** Perform root-cause analysis under resource saturation by synthesizing evidence from performance metrics, logs, and workload history to isolate cluster-wide instability or node-level failures.
- **Database Consultation.** Provide evidence-backed recommendations and system feature explanations tailored to the cluster's configuration and workload. Complex consultation tasks often require synthesizing insights from multiple documents. For example, to answer "Can I limit CPU and memory usage for a single query in DWS?", one must consult workload-management documentation for query-level memory controls, then review resource-pool materials to clarify the granularity of CPU limitations, and finally integrate findings to conclude that per-query CPU hard-capping is unsupported while memory can be constrained.

3.2 Data Sources

We categorize data sources into two types: (i) for training and evaluation; and (ii) for O&M analysis.

Operation and Maintenance Ticket

1. **Symptom**: On 2025/6/2 22:17 (cluster: …, project ID: …), users reported query latency spiking from ~300ms to >15s, …
2. **Initial Analysis**: Recent changes suggest three likely causes: (a) an unoptimized large transaction causing lock contention, or (b) …
3. **Validation**: Primary CPU usage was stable (<60%), ruling out resource exhaustion. pg_stat_activity revealed a 40+ minute uncommited transaction (PID 12845) …
4. **Root Cause & Remedies**: *Root cause*: Missing index on 'batch_id'. *Resolution*: Added a single-column index on batch_id and ….

Figure 3: An example of an O&M ticket.

3.2.1 Data Sources for Training and Evaluation. We use O&M tickets as the training and evaluation benchmark. O&M tickets originate from real production incidents and record the diagnostic trace of human experts. As shown in Figure 3, a typical ticket usually includes the (1) symptom, (2) initial analysis, (3) validation, (4) root cause and remedies. During diagnosis, the ticket captures the entire process by which DBAs obtain and analyze KPIs, query plans, logs, and other relevant information from the customer database cluster. Finally, it describes the confirmed root cause along with remedies.

3.2.2 Data Sources for Analysis. Our agent consumes the same product surfaces that DWS engineers use in production, integrated through DWS and its Autopilot stack. These sources provide sufficient signals for diagnosis and safe remediation.

(1) Textual Knowledge Corpus. It comprises semi-structured and unstructured sources used by DBAs for diagnosis and operations. Semi-structured content is lightly organized (hierarchies/indices), e.g., product documentation, internal O&M manuals, and standard operating procedures. Unstructured content is free-form without a uniform schema, e.g., community Q&A, technical blogs.

(2) Operating System-Level Metrics. Structured OS/hardware time-series capturing resource supply and pressure: 9 CPU metrics (utilization, idle percentage, context switches), 8 memory metrics (total/used memory, page faults), 8 disk metrics (usage, IOPS, throughput), and 8 network metrics (throughput, retransmits, connections).

(3) Database System-Level Metrics. It comprises two types:

• **Query and Workload Metrics.** Structured time-series/log data covering 30 transactional metrics (commit/rollback, sessions, lock waits), 70 `Top SQL` metrics (SQL/plans, runtime, I/O), query logs (timestamps, results, errors), and 54 resource-pool telemetry metrics (CPU, fast/slow_wait, read/write_speed).

• **Metadata and Configuration.** Semi-static metadata on database structure and operating parameters, covering 37 schema fields (tables, indexes, partitions), 13 statistical descriptors (row counts, histograms, NDV), configuration (GUCs; e.g., `shared_buffers`), and 12 resource-policy settings (WLM queues, resource groups, connection pools).

3.3 Problem Definition

We consider four DBA O&M categories defined in §3.1. For each category $k \in \mathcal{K}$, we assume: (i) a finite root-cause label space C_k; (ii) a remedy space $\mathcal{M}_k$ (e.g., parameter change, SQL/index rewrite, capacity action).

An O&M incident/task is a tuple $\mathcal{I} = (k, \Sigma, \sigma, c^*, \mathbf{m}^*)$, where $k \in \mathcal{K}$ is the category, Σ is a frozen, read-only snapshot of the DB/system state (metrics, logs, plans, configs) detailed in §5.3.1 or reference documentation, σ is a short natural-language symptom description, $c^* \in C_k$ is the expert root-cause label, and $\mathbf{m}^* \in \mathcal{M}_k$ is the ground-truth remedy specification used for verification.

Let $\mathcal{U}$ denote the catalog of callable tools. Each tool $u \in \mathcal{U}$ returns reproducible observations from Σ or documentation from knowledge base. After a finite number of interactions via tool calls, the agent produces a final report $\hat{Y} = (\hat{c}, \hat{\mathbf{m}}, \hat{V})$, where $\hat{c} \in C_k$ is the predicted root cause; $\hat{\mathbf{m}}$ is a concrete remedy specification (either a single remedy $\hat{m} \in \mathcal{M}_k$ or an ordered list $\hat{\mathbf{m}} = \langle \hat{m}_1, \ldots, \hat{m}_m \rangle$ with $\hat{m}_i \in \mathcal{M}_k$). and $\hat{V}$ records the complete diagnostic process for secondary analyses and does *not* enter the pass/fail criterion.

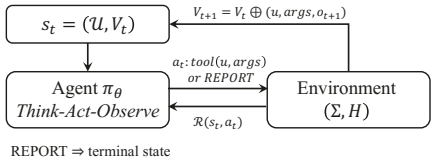


Figure 4: Compact MDP view for a $\mathcal{I}$: the agent takes an action (tool or REPORT), the environment returns observation and reward, and the diagnostic state V_t updates.

Let H be the verifier that determines success for a proposed report. The verifier returns pass iff the predicted cause matches the ground truth and the remedy check succeeds:

$$H(\hat{Y}; \Sigma) = \begin{cases} 1, & \text{if } \hat{c} = c^* \text{ and } H^{\text{rem}}(\hat{\mathbf{m}}; \Sigma) = 1, \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Let $v \in \{0, 1\}$ indicate whether $\mathcal{I}$ is executable verification (e.g., query rewrites). We define H^{rem} as:

$$H^{\text{rem}} = \begin{cases} 1, & \text{if } v = 1 \text{ and } \text{Lat}(\hat{\mathbf{m}}; \Sigma) \le t_{\text{ref}} \text{ and } \text{Out}(\hat{\mathbf{m}}) = \text{Out}(\text{ref}), \\ 1, & \text{if } v = 0 \text{ and } \hat{\mathbf{m}} \equiv \mathbf{m}^*, \\ 0, & \text{otherwise.} \end{cases}$$

Here $\text{Lat}(\hat{\mathbf{m}}; \Sigma)$ and $\text{Out}(\hat{\mathbf{m}})$ denote the latency and execution result after applying $\hat{\mathbf{m}}$ in Σ, while t_{ref} and $\text{Out}(\text{ref})$ denote the latency and result before remedy application. The symbol $\equiv$ is the per-incident equivalence relation for non-executable remedies (capturing fixed normalization such as aliases, units, and benign reorderings) detailed in §5.3.1.

We say $\mathcal{I}$ is *solved* iff $H(\hat{Y}; \Sigma) = 1$. Given a test set $\mathcal{D}$, the primary metric is the *Success Rate*:

$$\text{SR} = \frac{1}{|\mathcal{D}|} \sum_{\mathcal{I} \in \mathcal{D}} H(\hat{Y}; \Sigma), \tag{2}$$

reported overall and per-category (macro-averaged across $\mathcal{K}$).

We additionally report complementary process metrics based on the agent's cumulative diagnostic state $\hat{V}$; e.g., a *hallucination rate* that checks whether claims in the final report are supported by tool observations from Σ.

Objective. Let Π be the family of agent policies under comparison. The goal is to find $\pi^* \in \arg\max_{\pi \in \Pi} \text{SR}(\pi, \mathcal{D})$, with model selection on a held-out split and final reporting on $\mathcal{D}$.

Limitation. In production, the closed label spaces are used only for automatic verification and offline evaluation. `DBAgent` is not forced to choose a known cause when evidence is insufficient. We maintain an explicit escalation path: if the retrieved evidence and verifier checks are inconsistent, the agent emits an "insufficient evidence" report and routes the incident to human DBAs.

3.4 O&M as a Markov Decision Process

We model database O&M as an MDP $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$ [1] (Figure 4), where the policy π_θ invokes tools in the environment (Σ, H) and is optimized from reward feedback.

• **State $\mathcal{S}$.** A state $s_t \in \mathcal{S}$ contains only information visible at step t: $s_t = (\mathcal{U}, V_t)$, where V_t is the diagnostic state accumulated up to step t, comprising the initial symptom σ and all previous tool-observation pairs, with $V_0 = [\sigma]$.

• **Action $\mathcal{A}$.** An action $a_t \in \mathcal{A}$ takes one of two forms: (i) tool invocations (u, args) with $u \in \mathcal{U}$ (e.g., `Get_System_Metrics()`), or (ii) a terminal REPORT action that emits the final report $\hat{Y} = (\hat{c}, \hat{\mathbf{m}}, \hat{V})$ as defined in §3.3.

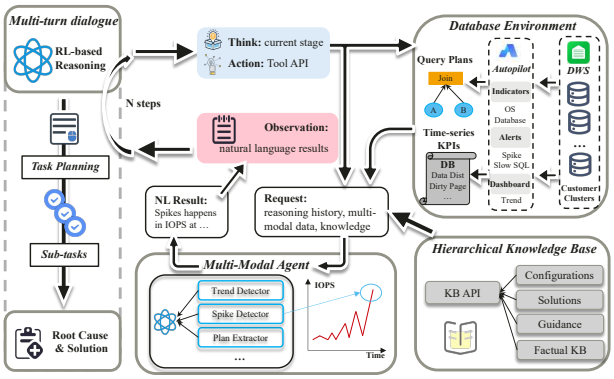


Figure 5: DBAgent framework overview.

• **Transition $\mathcal{T}$.** The kernel $\mathcal{T}(s_{t+1} \mid s_t, a_t)$ updates the revealed view. For a tool action, it returns an observation $o_{t+1} = u(\Sigma, \text{args})$ and sets $V_{t+1} = V_t \oplus (u, \text{args}, o_{t+1})$. The REPORT action transitions to a terminal state.

• **Reward $\mathcal{R}$.** The reward provides feedback (§5.3.2), with a terminal component aligned to the success definition in §3.3.

• **Discount γ.** We use $\gamma = 1$ for these episodic interactions.

For an MDP, a policy π maps states to action distributions, $\pi(a_t \mid s_t)$. A *trajectory* is $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T)$ generated by sampling $a_t \sim \pi(\cdot \mid s_t)$ and $s_{t+1} \sim \mathcal{T}(\cdot \mid s_t, a_t)$ until the terminal REPORT action at step T. The agent maximizes the expected cumulative reward: $\pi^* = \arg\max_\pi \mathbb{E}_{a_t \sim \pi(\cdot|s_t),\, s_{t+1} \sim \mathcal{T}(\cdot|s_t,a_t)} \left[\sum_{t=0}^{T-1} \mathcal{R}(s_t, a_t) \right]$. At termination, the environment computes a success indicator using H and $(\hat{c}, \hat{\mathbf{m}})$, yielding the terminal component of the return.

4 FRAMEWORK OVERVIEW

4.1 Overall Architecture

DBAgent employs an LLM as the *core policy* that plans, decides, and explains its reasoning, while surrounding it with three tightly-coupled modules (Figure 5). (i) **Knowledge & Tools Layer** (§ 5.1): It exposes a curated, queryable knowledge base and a suite of callable OS/DB tools, which covers (a) knowledge retrieval, (b) live metric acquisition at OS/DB levels. (ii) **Multi-Modal Agent (MMA)** (§ 5.2): It is a modality-specific perception layer for database-native signals (time-series KPIs, execution plans, and textual snippets). It converts database-native signals into concise, structured natural-language observations. The MMA provides perception primitives: anomaly detection for KPIs and structure-aware plan analysis, feeding compact textual evidence back to the policy. (iii) **RL-based Reasoning Engine** (§ 5.3): It optimizes the core policy based on RL to maximize O&M success rate, strengthening multiple capabilities, such as multi-step tool use and evidence-grounded report quality. Specifically, we use Group Relative Policy Optimization (GRPO) [30] as the RL algorithm under a multi-component reward.

We train the DBAgent policy by *trial and error* in a reproducible environment that mirrors production tool interfaces including Autopilot-defined KPIs and DWS native views (Top SQL, WLM, etc). Each episode begins from a snapshot (containing metrics, logs, execution plans, configurations). The DBAgent policy proposes intermediate analyses and final report. A task-specific verifier returns scalar rewards reflecting success. DBAgent updates its policy based on the feedback. As the feedback loop repeats, DBAgent improves its policy to achieve a higher success rate. Note that we do not provide any handcrafted task rules for DBAgent to achieve the task.

After training, DBAgent can be executed on *held-out* scenarios and symptoms/causes without learning to assess generalization. It must diagnose and remediate with the same tool APIs and reporting protocol, without prior exposure to those specific cases. We empirically validate its out-of-distribution generalization on a held-out split that withholds entire root-cause scenarios from training in §6.2.2. Accordingly, DBAgent is deployed in DWS 's Autopilot auto-diagnosis pipeline to handle alerts and recommend evidence-grounded remedies in production.

Algorithm 1 Think–Act–Observe loop

Require: symptom σ, tool catalog $\mathcal{U}$, KB API, snapshot Σ, policy π_θ
$V_0 \leftarrow [\sigma]$
for $t = 0, \ldots, T-1$ **do**
  π_θ generates <think> and one of:
  (a) <tool_call> KNOWLEDGE_BASE_SEARCH(type, query)
  (b) <tool_call> metric/plan/config/log tool
  (c) <answer>
  if an <answer> is generated **then**
    return final report
  end if
  MMA summarizes environment raw result into observation o_{t+1}
  $V_{t+1} \leftarrow V_t \oplus (\text{action}, o_{t+1})$
end for
return timeout or escalation report

4.2 End-to-End Inference Process

DBAgent executes a single serialized Think–Act–Observe dialogue. At step t, the policy conditions on the current trajectory V_t and emits a <think> block followed by either a <tool_call> or a terminal <answer>. Tool calls include both diagnostic APIs and the KB retrieval API KNOWLEDGE_BASE_SEARCH. Their raw outputs are converted by the MMA into <tool_response> observations and appended to the trajectory. Figure 2 shows a concrete query-performance diagnosis example, and Algorithm 1 summarizes the generic loop. The same rollout format is used during RL training, where trajectories are scored by the reward components in §5.3.2.

5 DBAGENT

5.1 Knowledge Base and Tool Construction

To acquire both static knowledge and real-time data, DBAgent adopts a unified function-calling paradigm [29] that exposes its curated knowledge base and environmental tools as callable APIs. Unlike conventional RAG pipelines [3] that pre-inject retrieved passages into the prompt via fixed retrievers, our policy initiates retrieval itself during reasoning: at each turn, it decides *whether* to retrieve, *what* to query, and *which* API to call. This agent-driven knowledge injection makes DBAgent more flexible and versatile.

5.1.1 Knowledge Base Construction. The foundational knowledge for DBAgent is derived from a vast corpus of human-oriented textual information for DWS, including user manuals, operational guides, and historical maintenance tickets. As this raw information is often too verbose and unstructured for direct use by an LLM, we implement a systematic curation process to bridge this gap. This

process, which involves three main steps, converts the raw documentation into a concise and structured knowledge source that DBAgent can actively query during problem-solving.

(1) Classification. We first categorize the documents based on their function within a typical diagnostic workflow. It mirrors the logical steps an agent would take, from gathering initial facts to finding a final solution. The distinct categories include: configuration parameter manuals ("configuration"), troubleshooting solutions ("solution"), best practice guides ("guide"), and foundational factual knowledge ("fact"). This taxonomy allows the agent to perform highly targeted searches at different stages of its reasoning process. For instance, it might first query for "fact" or "guide" to understand the system architecture before checking a specific "configuration" setting.

(2) Semantic Refinement. Since raw documentation is often lengthy and narrative-driven, we employ semantic chunking techniques to make it digestible. Our approach is adaptive: for narrative-heavy content like guides and solutions, we use recursive chunking to break down documents into smaller, semantically coherent paragraphs. For structured content, particularly configuration manuals, we apply a more structure-aware method. This involves segmenting the information by individual parameters or tables, ensuring that a parameter's description, default value, and range are kept together as a single, logical unit.

(3) Causality and Fact Extraction. Finally, from these refined chunks, we distill key information into a format that is both *concise* and *actionable*. In this way, the knowledge is efficient to process and directly useful for decision-making.

To achieve conciseness, we employ summarization and entity extraction to distill verbose text into its essential entities and their relationships by removing narrative filler. For instance, a lengthy configuration parameter description is condensed into a structured object containing only its name, value range, and functional impact.

More importantly, we make domain knowledge *actionable* by structuring it for the Think–Act–Observe loop. Concretely, we apply a hybrid extraction pipeline (LLM-based parsing plus heuristics) to operational guides and historical maintenance tickets, yielding records that (i) normalize *symptoms* (error codes, KPI anomalies, degradation patterns), (ii) map them to candidate *causes* consistent with the root cause label space C_k, and (iii) bind them to executable *actions* in an action space $\mathcal{A}$. It covers diagnostic probes (e.g., targeted measurements, plan inspection, configuration checks) or remedies (e.g., specific tool invocations with arguments). The resulting entries are persisted as two catalogs: guide and solution. For example, a record may encode *Symptom: high query queue times → Candidate causes: {misconfigured WLM (slot limits), lock waits, long-running transactions} → Actions: [*`Get_WLM_Config()`, `Get_Lock_Waits()`, `Get_Top_SQL()`*]*, which the policy can query to form hypotheses and select the next tool.

5.1.2 Tool API Construction. To obtain data sources for analysis (in §3.2.2), we build a suite of tool APIs that the agent's reasoning engine can invoke. Each tool is accompanied by clear natural language descriptions of its purpose, parameters, and output.

Our tool catalog is grounded in the administrator commands routinely used by DBAs. However, these sources expose schema oriented outputs (e.g., `delta_tup_fetched`) that lack linguistic context and are hard for an agent to interpret. We therefore semantically enrich each tool: LLM ingests (i) a formal specification from manuals or view schemas and (ii) curated notes from the KB's `fact`/`configuration` sections, and synthesizes a compact NL description (purpose, interface, output schema) consumable by the policy. For example, `Get_Top_SQL()` will extract DWS native view `PGXC_WLM_SESSION_INFO`. It exposes schema such as `session_id`, `resource_pool` and `spill_info`. Additional details in *KB notes (facts/config)*: `spill_info` value can be `All` or `[a:b]` and indicates memory pressure or skew. Cross-check with plan buffers and `work_mem`. Then the *synthesized doc* is: "Purpose: Inspect session-level resource-pool assignment and spill distribution to explain query latency or memory pressure. Interface: `Get_Top_SQL(...)` → `TABLE<session_id, resource_pool, spill_info, ...>`". Based on the categories of data source in §3.2.2, tools include:

• **Knowledge Retrieval**: The agent queries the knowledge base through a single API: `Knowledge_Base_Search(kb_type: string, query_string: string)`, which performs a similarity search and returns the top-k (e.g., k=5) most relevant chunks as its observation.

• **OS-Level Metric Acquisition**: The agent retrieves operating system metrics (CPU, memory, disk, and network) using a tool called `Get_System_Metrics(metric_type: string, start_time: timestamp, end_time: timestamp)`.

• **DB-Level Metric Acquisition**: It includes 7 workload-related tools, such as `Get_Top_SQL(start_time: timestamp, end_time: timestamp)`, which retrieves execution plans of specific queries and diagnoses performance bottlenecks. It also includes 4 metadata or configuration tools, such as `Get_Schema_Info(object_type: string, object_name: string, database_name: string)`, for querying schema information of database objects like tables.

5.2 Multi-Modal Agent

To address perception limitation, we introduce a dedicated sub-agent, the **Multi-Modal Agent (MMA)**, that converts raw tool outputs into structured natural-language observations for the core policy. The MMA conditions on (i) the current diagnostic trajectory (turn history), (ii) the observation data from the most recent tool call, and (iii) a library of modality-specific analysis modules including KPI anomaly detectors and query plan analyzers. Given the modality type, it selects and applies the appropriate module and emits a concise summary of findings. This summary serves as the *Observation* for the next cycle of the Think–Act–Observe loop.

The MMA is designed for extensibility across both supported modalities and analytical techniques. In our current implementation, it provides specialized modules for three data types. We skip MMA for simple tool results like metadata and parameters, which agents can understand directly.

Knowledge Base Results. When tools retrieve information from the textual knowledge corpus, they may return multiple scattered facts or lengthy passages. The MMA processes these results, synthesizing them into a coherent and condensed summary.

Time-series KPIs. To interpret KPI time series, we employ a statistically grounded pipeline to extract trends and detect anomalies [3, 15], rather than passing raw numeric sequences directly to the LLM. For a given KPI with observations $\{x_t\}_{t=1}^{n}$, we apply Seasonal–Trend decomposition via LOESS (STL) to obtain x_t =

$x_t^{\text{seas}} + x_t^{\text{trend}} + x_t^{\text{res}}$, where x_t^{seas} captures seasonal structure, x_t^{trend} captures long-term drift, and x_t^{res} denotes the residual component. Anomaly detection is then performed on $\{x_t^{\text{res}}\}$ using a generalized Extreme Studentized Deviate (ESD) procedure with robust location and scale estimates (median and median absolute deviation, MAD), which attenuate the influence of extreme points. The test iteratively flags at most $k_{\max}$ outliers at significance level α, yielding a set of anomalous indices $I_{\text{anom}} \subseteq \{1, \ldots, n\}$. Detected events are subsequently rendered as concise, natural-language observations that reference the KPI, the time of occurrence, and deviation relative to the seasonal–trend baseline (e.g., "CPU utilization exhibited an anomalous spike at 10:35 AM").

Query Plans. Raw query execution plans are intricate and deeply nested, which makes them difficult for an LLM to parse and interpret directly. To simplify and analyze this data, our agent employs a structured two-stage procedure: `Plan Extractor` followed by `Plan Analyzer`, that together implement `Analyze_Query_Plan()`. First, the `Plan Extractor` parses the textual output of `EXPLAIN` and normalizes it into an operator graph (typically a tree or DAG). Each node represents a plan operator (e.g., `HashJoin`) and is populated with attributes, including estimated and (when available) actual row counts, cumulative and local costs, and relevant physical properties (e.g., ordering, parallelism, or data distribution).

Subsequently, the `Plan Analyzer`, a rule-based engine, traverses the structured plan to identify common performance pathologies. It applies a set of pre-defined heuristics at operator nodes and over their local neighborhoods, such as: (1) detecting large discrepancies between estimated and observed row counts to flag cardinality errors (often due to stale statistics); (2) identifying `SeqScan` on large relations with low selectivity, suggesting a missing or unusable index; and (3) checking for data skew in parallel or distributed joins and aggregations. For each issue, the analyzer emits a structured warning that records the implicated operator(s) and supporting evidence drawn from parent/child context. The resulting summary, together with the normalized plan, is then passed to the MMA.

Analyzing plans with LLMs remains challenging due to a structural mismatch between sequential token processing and the inherent graph topology of execution plans. This difficulty is compounded by the scale and heterogeneity of real-world plans, which can span hundreds of operators with intricate join patterns [4–6]. Therefore, the development of methods to interpret such complex structures is a crucial area for future research.

5.3 Reasoning Engine

5.3.1 Environment Construction. RL training requires many interactions, which requires the constructed environment to meet the following requirements: (1) **efficient sampling** to support massive agent-environment interactions within feasible time and resource budgets, and (2) **stable and accurate feedback** to ensure a stable training process and provide the agent with supervision. To satisfy these requirements, we construct a high-fidelity, replayable environment (Σ, H) that mirrors production APIs.

For **efficient sampling**, we extract database instances from small-batch production data to reproduce target problem scenarios. Each instance Σ stores all important data in the realistic environment (including time-series KPIs, logs, execution plans, and configuration metadata), obtaining a frozen system snapshot that mirrors O&M scenarios. Query tuning tasks are directly executed using internal data in the DWS cluster for realistic interaction.

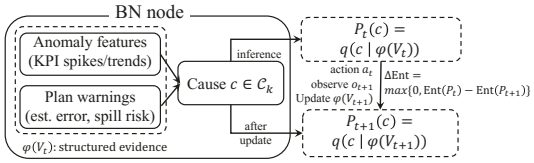


Figure 6: Bayesian estimation of $P_t(c)$ from structured evidence $\varphi(V_t)$, and update to $P_{t+1}(c)$ after action a_t and observation o_{t+1}; the stepwise reward uses entropy reduction ΔEnt.

For **stable and accurate feedback**, we ensure robustness at two levels. *During agent-environment interaction*, we embed comprehensive validity checks and structured error feedback across the full tooling pipeline—tool parsing, tool invocation, and response formatting—so each step either succeeds with well-formed output or returns deterministic error signals. For example, when the agent executes a tool that the system does not allow (such as adjusting worker_mem), the environment returns the corresponding error signal. *At episode termination*, verifier H serves as a rule-based final check: the agent emits a final report $\hat{Y} = (\hat{c}, \hat{\mathbf{m}}, \hat{V})$ enclosed in `<answer>...</answer>`, which we parse with a deterministic extractor. If parsing fails, H returns 0. Otherwise, verification applies two criteria: **(1) Cause matching**, using exact string equality after light normalization (lowercasing, whitespace and punctuation cleanup); **(2) Remedy verification**, where executable cases (query rewrites) undergo direct execution validation, and non-executable cases are compared against the incident-specific specification $\mathbf{m}^*$ under fixed normalization rules for aliases, units, and benign reorderings (e.g., "increase `work_mem` to `128MB`" ≡ "`set work_mem=131072kB`"). For customer consultation tasks, $\mathbf{m}^*$ specifies a required evidence set (KB document IDs); we parse cited documents from the agent's answer to obtain normalized $\hat{\mathbf{m}}$ and require exact set equality $\hat{\mathbf{m}} \equiv \mathbf{m}^*$. This dual-layer design—robust interaction feedback coupled with deterministic final verification—ensures stable, failure-aware RL training.

5.3.2 Reward Modeling. A reward model r_ϕ evaluates a trajectory on a concrete instance $\mathcal{I}$. Given a generated trajectory $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T)$, the total reward is

$$r_\phi(\mathcal{I}, \tau) = w_1 r_{\text{format}}(\tau) + w_2 r_{\text{tool}}(\tau) + w_3 r_{\text{answer}}(\tau) + w_4 r_{\text{stepwise}}(\tau),$$

where $w_i \geq 0$ and $\sum_i w_i = 1$. Each r_ϕ is normalized to $[0, 1]$.

Format Reward ($w_1 = 0.2$). This term enforces well-formed turns under the Think–Act–Observe schema. For a trajectory τ with T agent turns, let K be the number of turns that satisfy the format checks (e.g., a `<tool_call>` is preceded by `<think>`, the tool name/arguments are valid, and tags are properly closed). We define $r_{\text{format}}(\tau) = \frac{K}{T} \in [0, 1]$.

Tool Reward ($w_2 = 0.1$). This term measures the coherence of plan-action throughout the trajectory. Let *Plan* denote actions planned in `<think>` blocks and *Exec* the executed `<tool_call>`. We align plans to executions by name and compatible arguments within a window of turns. Define *unfulfilled plans* $Plan_{\text{ue}} = \{p \in Plan : p \text{has no aligned execution}\}$ and *unplanned executions* $Exec_{\text{up}} = \{e \in$

Exec : e has no aligned plan}. We compute $r_{\text{tool}} = 1 - \frac{|Plan_{ue}| + |Exec_{\text{up}}|}{|Plan| + |Exec|}$ clipped to $[0, 1]$, penalizing both speculative and unexecuted calls.

Answer Reward ($w_3 = 0.5$). We instantiate the outcome term with the success indicator in §3.3. We set $r_{\text{answer}}(\tau) = H(\hat{Y}; \Sigma)$ (Eq. (1)).

Stepwise Reward ($w_4 = 0.2$).. This shaping term rewards diagnostic actions that reduce uncertainty over the root cause. For category k, let C_k be the cause set and let $P_t(c) = q(c \mid \varphi(V_t))$ be the belief estimated by a Bayesian network [17] trained on historical tickets, where $\varphi(\cdot)$ encodes structured evidence such as KPI anomalies, plan warnings, and WLM checks. The network architecture is shown in Figure 6. After action a_t yields observation o_{t+1}, we update $P_{t+1}(c) = q(c \mid \varphi(V_{t+1}))$ and assign the normalized entropy reduction: $r^{(t)}_{\text{stepwise}} = \frac{\max\{0, \text{Ent}(P_t) - \text{Ent}(P_{t+1})\}}{\log |C_k|}$, $\text{Ent}(P) = -\sum_{c \in C_k} P(c) \log P(c)$. The reward is injected per turn rather than aggregated only at the episode end, providing denser supervision for exploration and credit assignment. To prevent over-probing, if one decision triggers more than L_{tool} tool calls, we set $r^{(t)}_{\text{stepwise}} = 0$ for that turn. This Bayesian signal may encode historical frequency bias: rare or newly emerging root causes can receive lower prior probability, and entropy reduction may favor actions that confirm common diagnoses. We mitigate this by using it only as an auxiliary shaping term ($w_4 = 0.2$); terminal success is still determined by the rule-based, evidence-based verifier; and in production, low-confidence or conflicting evidence triggers human escalation rather than forced classification.

5.3.3 RL Training Framework. We train the reasoning policy with a multi-turn, GRPO-style objective [19, 30] under the framework in §3.4. The LLM itself is the policy π_θ that generates a trajectory τ. To stabilize learning and prevent linguistic drift, we regularize toward a frozen reference policy π_{ref} via a KL penalty:

$$\max_{\pi_\theta} \; \mathbb{E}_{\mathcal{I} \sim \mathcal{D},\, \tau \sim \pi_\theta(\cdot \mid \mathcal{I})} \Big[r_\phi(\mathcal{I}, \tau) \Big] - \beta\, D_{\text{KL}}\big[\pi_\theta(\tau \mid \mathcal{I}) \,\big\|\, \pi_{\text{ref}}(\tau \mid \mathcal{I})\big], \quad (3)$$

where β controls the trust penalty. In our implementation, r_ϕ combines terminal outcomes with dense per-turn shaping (§5.3.2).

Rollout phase. For each incident $\mathcal{I}$, the current policy π_θ interacts with the simulated environment (§5.3.1) to produce a *group* of G distinct multi-turn trajectories $\{\tau_1, \ldots, \tau_G\}$. Each trajectory is a full Think–Act–Observe episode with stepwise rewards injected during interaction and terminal rewards at the end.

Update phase. We evaluate each τ_j with r_ϕ and compute *within-group* advantages by subtracting the group mean (the GRPO baseline). Policy gradients are applied to the agent's tokens only, with a KL penalty to π_{ref}. We use *loss masking*: tokens generated by the environment (e.g., inside `<tool_response>...</tool_response>`) are masked out so the model learns reasoning and action (`<think>`, `<tool_call>`, `<answer>`) rather than memorizing tool outputs.

6 EXPERIMENTS

6.1 Experiment Setup

6.1.1 Setup. To rigorously evaluate `DBAgent`, we construct a multi-task, multi-difficulty benchmark from DWS Autopilot operational scenarios. It captures multi-tenant dynamics and cross-cluster drift, mirroring the variability faced by DWS engineers in daily O&M. Each incident is constructed based on a corpus of O&M tickets (§3.2.1) and production artifacts. We normalize tickets into categories (e.g., lock contention, misconfiguration) and assemble the multi-modal artifacts per scenario including Autopilot KPIs (multi-signal telemetry) and DWS-native views (e.g., `Top SQL`, execution plans, WLM/resource policies). When artifacts are incomplete, we reproduce the incident on an isolated cluster to capture the missing signals, then de-identify and version the traces. Following §5.3.1, this dataset enables safe, efficient, and repeatable evaluation by mapping function calls to deterministic responses. Our replay design cannot synthesize truly novel telemetry beyond what was captured. We mitigate this with noise injection. Future work will integrate stochastic generators and fault injectors to expand coverage.

Specifically, snapshot telemetry contains at most 500 samples per time-series metric (per instance, typically over 2–10 h windows). KPI categories and their metric counts are detailed in §3.2.2. Query plan traces are sourced from real ticket cases with per-plan operator counts ranging from 1 to 90. This complex environment poses significant challenges for the agent, requiring it to meticulously sift through vast amounts of data to identify critical signals.

As Table 1 shows, the benchmark spans four task categories across easy and hard levels, distinguished by indicative metric count and information volume. Harder diagnostic instances involve more signals, requiring agents to detect subtle anomalies and disambiguate competing root causes. For customer consultations, complex tasks (e.g., §3.1) demand synthesizing multiple data sources to produce coherent responses. Notably, the diagnostic scope includes 10 distinct root causes for query tuning tasks and 14 root causes for system overload and health check tasks.

Dataset Split. We evaluate both **in-distribution (ID)** and **out-of-distribution (OOD)** performance. For ID, we use a random 50/50 train–test split, ensuring test cases come from different database instances and contexts. For OOD, we evaluate on a held-out split of DWS diagnostic scenarios comprising four scenarios (143 cases total): three categories of easy task and one category of hard task. The split removes overlap along multiple axes of novelty including workload/schema, root causes and KPI patterns.

Evaluation Metric. Unless otherwise noted, we evaluate the agent using Success Rate (SR), which is given in §3.3 as Eq. (2).

Implementation. We conducted all experiments on a Huawei BMS server with Euler OS 2.10, featuring a 192-core CPU, 1.5TB RAM, 2TB SSD, and eight Ascend 910B3 NPUs. For RL training, we implemented our multi-turn GRPO algorithm by extending VeRL [31] using Qwen3-8B as the base model. Our architecture employs: (1) Qwen3-32B for offline document processing (yielding 10,040 chunks, <1K tokens each), (2) Qwen3-8B for the multi-modal agent, and (3) a Faiss [11] vector database retrieving top-5 relevant chunks per query. All models except Qwen3-235B (accessed via API) were locally deployed using vLLM [22].

6.1.2 Methods Compared. We compare `DBAgent` against three categories of strong baselines. For fair comparison, all baselines implement an agent loop sharing the same knowledge corpus and tool APIs. We omit RAG-only single-turn QA systems [3, 7, 32], as they cannot support the multistep use of stateful tools.

ReAct Agent. This category evaluates unfine-tuned LLMs implementing the ReAct loop with access to the same tools and knowledge base. The agent iteratively generates "thoughts" and "actions"

Table 1: A real-world benchmark task categories and difficulty criteria.

Task Category	Difficulty	Criteria	# Types	Example Scenarios	Cases
Query Performance Tuning	Easy	Root cause evident from single anomaly in query plan operator	7	Inefficient operators (e.g., `Nested Loop`, `NOT IN`); table/computation skew; poor index usage	152
	Hard	Requires cross-operator analysis or combining with performance metrics	3	Excessive small CUs; predicate pushdown failures; complex subquery issues	64
System Overload & Health Check	Easy	Identifiable by single indicative system metric	9	High SQL concurrency; data skew; low disk space	250 + 120
	Hard	Requires inferring from multiple correlated metrics	5	High `INSERT`/`COPY` volume; dirty page rate/temp file spills; transaction log accumulation	174 + 70
Customer Consultation	Easy	Answerable with 0-1 knowledge base queries	–	Parameter definitions; simple "how-to" requests	172
	Hard	Requires synthesizing from multiple documents	–	Feature comparisons; multi-step troubleshooting	80

(tool calls), processing environment observations in each round. We use few-shot prompting that includes the task description, tool definitions, and several high-quality reasoning trajectories as examples. To study scaling effects, we evaluate three ReAct-based baselines: **ReAct-8B**, **ReAct-32B**, and **ReAct-235B**, which are built respectively on Qwen3-8B, Qwen3-32B, and Qwen3-235B [34].

D-Bot Agent. D-Bot [40] orchestrates database diagnosis with a multi-agent mechanism and an LLM-guided tree search algorithm, where each node corresponds to a candidate (hypothesis, action) tuple. Since the original D-Bot system is not directly executable in our proprietary DWS environment, we make a best-effort adaptation to DWS. Specifically, we retain its original processing methodology, including offline summary-tree-based knowledge extraction and BM25-based passive knowledge matching, and map its diagnostic actions to the available DWS tool APIs. For fairness, we provide D-Bot with the same raw DWS corpus used by `DBAgent`, but do not use our curated KB construction. Since D-Bot's passive matching mechanism is not designed for open-ended consultation and document synthesis, we exclude it from the customer consultation task and evaluate it on the diagnostic and tuning tasks. We instantiate this baseline with Qwen3-235B and denote it as **DBot-235B**.

Tree-Search Agent. Building on D-Bot's tree search, we implement a system-native baseline with three key adjustments: (1) a single-agent design to preserve global context and eliminate consensus overhead; (2) a Tool API for our hierarchical knowledge base, enabling active, real-time retrieval over passive matching; and (3) DWS-specific few-shot prompts that guide branch expansion by defining necessary signals, tools, and acceptance criteria. We evaluate three Qwen3-based variants: **TreeSearch-8B**, **TreeSearch-32B**, and **TreeSearch-235B**.

6.2 Overall Performance

6.2.1 In-Distribution Performance. We first evaluate `DBAgent` on the multi-task, multi-difficulty benchmark defined in §6.1 with ID dataset split, reporting Success Rate (SR) and summarizing results in Figure 7. Baselines are denoted *ReAct-8B/32B/235B*, *TreeSearch-8B/32B/235B*, and *DBot-235B*, where LLMs are the Qwen3 series.

`DBAgent` *vs. Baselines*. As shown in Figure 7, `DBAgent` attains higher SR across difficulty strata. On *easy* tasks, `DBAgent` reaches 88% SR, exceeding the best baseline (TreeSearch-235B) at 65% by +23pp. On *hard* tasks, `DBAgent` maintains 68% SR versus 50% for the strongest baseline (TreeSearch-235B; +18pp). Although baseline SR scales positively with model size (ReAct-235B > ReAct-32B > ReAct-8B and TreeSearch-235B > TreeSearch-32B > TreeSearch-8B), `DBAgent`, built on an 8B backbone, surpasses even the 235B variants. This indicates that, for structured DBA O&M tasks, an agentic framework with RL-based reasoning yields larger gains than scaling the base LLM alone. General-purpose LMs struggle to adapt to tool use and dynamic reasoning, while tree-search agents lack the flexibility required for diverse, open-ended incidents. Among tree-search-based systems, DBot-235B generally underperforms our system-native TreeSearch at both difficulty levels. Although we make a best-effort adaptation of D-Bot to the DWS, its original passive BM25 matching is difficult to align with DWS-specific domain knowledge and tool semantics. As a result, it often fails to retrieve the precise diagnostic guidance needed for the current trajectory, and instead falls back to generic database solutions from its pretrained knowledge. This leads to generic conclusions and occasional hallucinations of unsupported metrics or tools, as further discussed in §6.3.

Multi-Task Performance. `DBAgent` delivers robust, well-balanced SR across the four task categories. For system overload & health checks, `DBAgent` attains 89% SR on easy cases and exceeds 60% on hard cases, whereas baselines struggle on scenarios requiring correlation across multiple KPIs. For query performance tuning, which demands cross-operator reasoning over execution plans, `DBAgent` reaches 83% SR (easy) and 71% (hard), substantially outperforming baselines on the hard split. For client consultation (knowledge retrieval and synthesis), `DBAgent` maintains a clear margin: over 90% SR on easy and over 70% on hard cases. However, the *TreeSearch*-series (especially TreeSearch-235B) is relatively weak. This is consistent with its design: a restricted diagnostic routine that is effective for structured troubleshooting but ill-suited to the flexible retrieval and synthesis required by client consultation.

Performance Across Difficulty Levels. Difficulty stratification is critical for assessing deep reasoning. Baselines struggle even on easy tasks (e.g., 8B LLM achieves 0% SR on easy query performance tuning) and tend to degrade further on hard cases requiring multi-step analysis. Within query performance tuning, we also observe an abnormal baseline pattern: TreeSearch-235B performs strongly on hard instances but underperforms on easy ones. This is likely a consequence of its complex tree search that *overthinks* simple cases. By contrast, `DBAgent` sustains high SR on both easy and hard splits (e.g., routine health checks drop only 10pp from 83% to 73%, vs. a

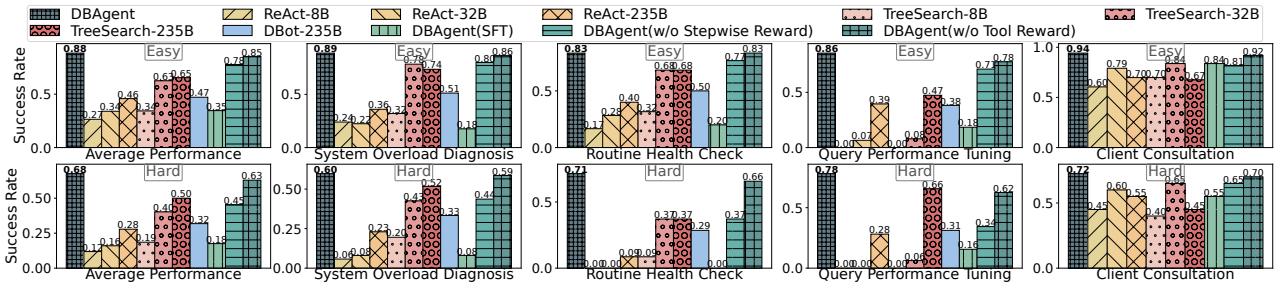


Figure 7: Overall performance comparison of DBAgent and baseline models under ID dataset split.

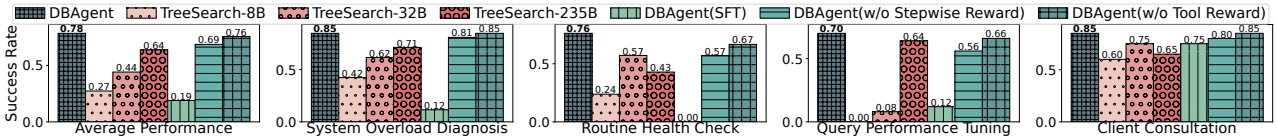


Figure 8: Performance comparison of generalization within our held-out DWS scenario split.

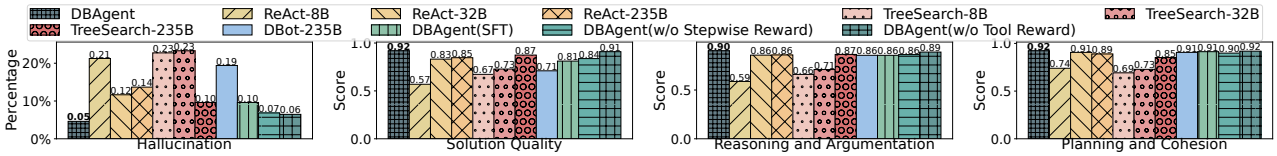


Figure 9: Agent reasoning capability and solution quality measurement.

31pp drop from 68% to 37% for the strongest baseline). This reflects a learned *adaptive* policy from DBAgent. The token-length analysis in §6.3 corroborates this behavior.

6.2.2 Out-of-Distribution Performance. We compare DBAgent with tree-search baselines (TreeSearch-8B/32B/235B) under the OOD split. These OOD cases require composing evidence across unfamiliar KPI trajectories or invoking tool chains not seen together during training. For tree-search agents, we explicitly isolate OOD scenarios in the node design: the node library (templated actions/tool calls) and their few-shot prompts are derived solely from the ID scenarios (i.e., no OOD-specific prompts are introduced). The expansion functions are identical across ID/OOD. We report SR in Figure 8 (ReAct-style agents are omitted as non-competitive).

Overall, DBAgent attains 78% SR, surpassing the best tree-search baseline (TreeSearch-235B) at 64% (+14pp). DBAgent consistently leads across tasks: Routine Health Check 76% vs. 57% (TreeSearch-32B; +19pp), System Overload Diagnosis 85% vs. 71% (TreeSearch-235B; +14pp), Query Performance Tuning 70% vs. 64% (TreeSearch-235B; +6pp), and Client Consultation 85% vs. 75% (TreeSearch-32B; +10pp). We also note that TreeSearch-235B's relative strength on Query Performance Tuning appears to stem primarily from raw model capacity and longer effective context handling (e.g., parsing and comparing long execution plans), rather than from the tree search itself. Overall, Tree-Search agents are brittle under distribution shift. DBAgent instead learns an adaptive policy, composing probes and remedies from live evidence to maintain advantages across tasks and held-out cases.

6.3 Reasoning Capability and Answer Quality

Beyond SR, we assess whether trajectories exhibit *sound reasoning* and *reliable answers* with four rubric-based metrics. These dimensions are central to agent *usability*, yet they are seldom reported in

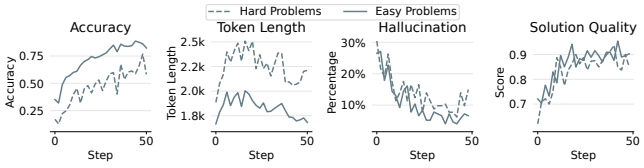


Figure 10: DBAgent performance and reasoning capability scales with RL training.

prior works. Given an instance $\mathcal{I}$ and an agent-generated trajectory τ (including complete diagnostic process $\hat{V}$) , we utilize an widely used *LLM-as-a-Judge* method [39] with a Qwen3-235B model with fixed rubric prompts. Each metric evaluation generates a binary result (pass/fail). The metrics include:

Planning & Cohesion. Pass if the trajectory presents a clear diagnostic plan and executes it coherently: every planned action in <think> is eventually executed and every <tool_call> is justified by a preceding/concurrent plan (cf. r_{tool} in §5.3.2).

Reasoning & Argumentation. Pass if intermediate conclusions are logically derived and explicitly supported by prior observations in V_t (no unsupported "jumps").

Solution Quality. Pass if the final <answer> is specific and actionable and its claims are justified by evidence in $\hat{V}$ (e.g., cited KPIs, plan warnings, KB entries). This complements SR by grading explanatory adequacy, not only correctness.

Hallucination. We flag hallucination when the trajectory contains fabricated content (e.g., non-existent APIs or KB facts from $\hat{V}$). Lower is better. Detection leverages our tool list and KB provenance.

Reasoning and Answer Quality. Figure 9 reports pass rates on the four rubrics. DBAgent outperforms all baselines across the board: it exceeds 0.9 on Planning & Cohesion and Reasoning & Argumentation, indicating that it proposes structured diagnostic plans and follows them with evidence-grounded logic. It also achieves

high Solution Quality (0.92), providing actionable and well-justified answers. Conversely, baselines, particularly smaller variants, frequently fail plan–execution cohesion or make unsupported inferences, and exhibit higher hallucination rates (e.g., invented tools). Notably, as its passive retrieval mechanism often fails to surface specific domain knowledge, DBot-235B suffers from a high hallucination rate of 19%—double that of TreeSearch-235B. In contrast, DBAgent remains highly grounded with a hallucination rate below 5%. Overall, these evaluations demonstrate significantly better usability of DBAgent in practice.

Impact of RL on Reasoning Dynamics. Figure 10 traces how these properties emerge during RL. At iteration 0, the untrained policy displays *high* hallucination, which is a known behavior of base LLMs [20], but hallucinations decrease sharply as training proceeds, while both SR and Solution Quality rise. Token-length trends further indicate an *adaptive* policy: trajectories become more concise on easy instances and longer/more detailed on hard ones, reflecting effort allocation proportional to task complexity. These dynamics are consistent with our reward design: the stepwise information-gain term encourages informative actions each turn, and the plan–action coherence term regularizes execution against stated plans, yielding reliable reasoning.

6.4 Design Choices

6.4.1 Effectiveness of RL and Reward Design. In this section, we study (1) why we choose to learn from experience (i.e., training with RL) instead of expert demonstration and (2) the significance of reward design. We evaluate three ablated versions of DBAgent:

- **DBAgent (SFT):** This baseline is the DBAgent reasoning engine after only the supervised fine-tuning without any RL training. It is trained on a curated dataset of 1800 expert diagnostic trajectories.
- **DBAgent (w/o Tool Reward):** This variant is trained using our full RL framework, but with the tool reward component (r_{tool}) removed from the reward model.
- **DBAgent (w/o Stepwise Reward):** This variant is also RL-trained but omits the entropy-based step-wise reward (r_{stepwise}) to demonstrate its importance for learning multi-step reasoning.

Impact of RL. Under the ID setup, Figure 7 shows that DBAgent outperforms its SFT variant by a wide margin. On the *hard* split, DBAgent-SFT attains 18% SR versus 68% for DBAgent, indicating that RL is essential for learning adaptive decision strategies beyond expert demonstrations. On the OOD split, as shown in Figure 8, the gap widens: DBAgent achieves 78% SR, while DBAgent-SFT obtains only 19%. This pattern highlights that SFT struggles to internalize environment dynamics from experience, thereby limiting generalization and tends to show more hallucination (as shown in Figure 9).

Impact of Reward Design. Under the ID setting (Figure 7), the benefits of r_{stepwise} and r_{tool} are most pronounced on the *hard* split. Removing the entropy–reduction term r_{stepwise} drops SR to 45%, whereas removing the plan–action coherence term r_{tool} yields 63%. Hard cases demand deeper, multi-step reasoning and precise tool use, so reward shaping that encourages uncertainty reduction and coherent interventions has greater effect. Across all experiments, ablating r_{stepwise} hurts more than ablating r_{tool}, likely because dense per-turn feedback is critical for stabilizing exploration and credit assignment in multi-round diagnosis. Such observation and arguments still hold on the OOD setup in Figure 8.

Table 2: Impact of different DBAgent modules.

Metric	DBAgent	DBAgent (raw KB)	DBAgent (w/o MMA)	DBAgent (w/o both)	ReAct-8B	ReAct-8B (w/o both)
SR (%)	**84.5**	29.2	60.3	22.4	22.9	11.8
Hallu. (%)	**4.6**	35.5	23.3	37.9	21.3	26.6
Plan.	**0.92**	0.65	0.75	0.72	0.74	0.66

Table 3: DBAgent multi-modal agent analysis.

(a) Time-series Analysis (F1 %)

Length	Metric Dimension		
	1–10	10–20	20–30
10	57.1	57.5	48.7
20	64.5	35.0	24.0
100	18.2	9.0	6.3
500	3.2	1.7	1.1

(b) Plan Analysis (%)

Method	# of Operators		
	1–30	30–60	60–90
DBAgent	95.2	100.0	100.0
(Rule)	90.5	85.7	100.0
(LLM)	81.0	100.0	75.0
(w/o MMA)	76.2	42.9	0.0

6.4.2 Effectiveness of DBAgent modules. To validate the effectiveness of our key architectural components, we conducted an ablation study focusing on two critical modules: the curated Knowledge Base (KB) and the Multi-Modal Agent (MMA). We measured the impact of ablating these modules on SR, hallucination rate, and planning quality, with the results summarized in Table 2.

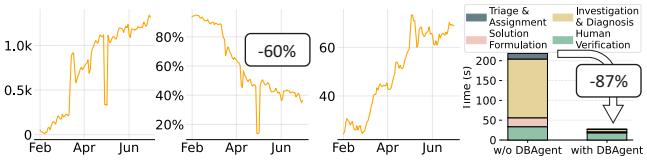
Impact of KB. For the KB, the importance of our knowledge curation is clear when replacing the curated KB with raw documentation (DBAgent (raw KB)). When the curated KB is replaced with raw unstructured documentation, the agent's performance collapses: SR drops from 84.5% to 29.2%, and the hallucination rate explodes from 4.6% to 35.5%. This demonstrates that the reasoning engine cannot effectively parse and utilize verbose, human-oriented text for complex problem-solving. Our curation process is essential for providing the agent with actionable knowledge.

Impact of MMA. For the MMA, removing the MMA (DBAgent (w/o MMA)) also causes a severe degradation in performance. Without the MMA to interpret and translate complex data like time-series KPIs and query plans, SR drops significantly to 60.3%, and the planning score falls from 0.92 to 0.75. This suggests that the MMA is a crucial perception layer. MMA also benefits untuned agents: allowing them in ReAct-8B nearly doubles SR (11.8% → 22.9%) and improves planning (0.66→0.74).

We further examine the MMA's modality-specific preprocessing functions: time-series anomaly detection and query plan analysis.

For time-series KPIs, Table 3a reports F1 scores under different sequence lengths and metric dimensions. Performance decreases as sequences become longer and higher-dimensional, which is expected due to increased noise and correlation complexity. In our deployment setting, diagnostic windows are typically 2–3 hours with 60–120 second sampling intervals, where the detector is sufficient for localizing suspicious periods. We tune the module toward high precision, so flagged anomalies are reliable even when the returned window is slightly broader. For query plans, Table 3b compares the full MMA with rule-based, LLM-only, and w/o-MMA variants on plans with different operator counts. The full MMA remains stable across plan sizes, while the alternatives degrade on more complex plans; in particular, w/o MMA drops to 0% SR on 60–90-operator plans. These results show that structured plan

preprocessing is important for converting complex execution plans into usable evidence for downstream reasoning.



(a) Click Counts Per Day (b) Tickets / Historical Average Tickets (%) (c) Click Counts Per Day (d) Average Handling Time (min)

Figure 11: Deployment effect of DBAgent.

Table 4: Distribution of O&M failures during deployment.

Category	Outdated KB	Rare Faults	Massive Telemetry	Retrieval Misses	Insufficient Evidence	Model Hallucination	Others
Proportion	19%	17%	13%	12%	11%	9%	19%

6.5 Deployment at Huawei with Real Customers

6.5.1 Deployment Setup. DBAgent was deployed in two real-world scenarios in 2025. It was first integrated into Huawei's internal ERP system from February to June as a first-line support tool for user queries and consultation tickets. It was later deployed from July to September in the analytical processing system of one of China's largest banking institutions for database diagnostics.

To bridge the gap between offline benchmark capability and production reliability, DBAgent is used as diagnosis and recommendation service rather than an unconstrained executor. In production, it is triggered by fault alerts or scheduled health checks, collects evidence from DWS/Autopilot tools, and generates a report containing the suspected root cause, supporting evidence, and recommended remedies. The online tool access exposed to the agent is read-only, rate-limited, and restricted by a whitelist. Operations that may change cluster state (e.g., configuration changes, DDL operations, session termination, or resource-policy updates) are not directly executed by the agent; they are risk-tagged by rule-based checks and require human approval through the existing O&M workflow.

Before a report is surfaced to DBAs, an independent LLM-based audit module reviews the logged diagnostic trajectory with fixed rubric prompts. It checks whether each root-cause claim and remedy is supported by tool observations or KB evidence. It flags unsupported reasoning or missing evidence, and marks low-confidence or high-risk cases for human escalation. Risky operations still require human approval through the existing O&M workflow.

Failed or escalated diagnoses are logged for periodic review by DWS engineers. An LLM-assisted extraction pipeline summarizes these cases into candidate *symptom–cause–action* records. They are integrated into the guide/solution category of the KB only after human review, de-identification, and versioned approval. This human-in-the-loop process allows DBAgent to improve from production experience while preventing unverified failures from being fed back into the knowledge base.

6.5.2 Results. In the Huawei ERP deployment (>30 departments, >1,000 active employees), the system recorded 80,000 clicks (Figure 11(a)) and achieved a 60% reduction in human-handled consultation tickets (Figure 11(b)).

In the banking deployment (>15 database clusters, 10 departments), the system handled 5,000 diagnostic clicks (Figure 11c). It reduced Average Handling Time (AHT) by 87% (Figure 11d) by automating time-intensive diagnosis and solution formulation. This allowed DBAs to focus on verification, shifting resolution times from hours to minutes. Furthermore, we analyzed the causes of diagnostic failures encountered during the deployment. As shown in Table 4, an outdated knowledge base (19%) and rare faults (17%) are the primary causes, highlighting the ongoing challenge of maintaining context freshness and handling unprecedented anomalies.

6.5.3 Lessons Learned. Production deployment revealed several gaps between offline evaluation and real O&M usage.

Safety Gates Are Mandatory. Strict action boundaries are essential for production trust. DBAgent is deployed as a diagnosis and recommendation service: online tool access is read-only, rate-limited, and whitelisted, while risky operations require human approval through the existing O&M workflow.

Assisting Rather Than Replacing DBAs. DBAgent does not replace DBAs in high-risk environments. Its main value is to reduce repetitive evidence gathering by correlating metrics, plans, configurations, etc, allowing DBAs to focus on validation and final decisions.

Evidence-Grounded Reports Drive Adoption. DBAs trust the agent more when conclusions are backed by traceable evidence, such as KPI anomalies, plan warnings, or KB entries. Evidence citation is therefore critical for both evaluation and production adoption.

Knowledge Freshness Matters. Many deployment errors come from outdated knowledge, version mismatch, or retrieval misses. Since reasoning quality is bounded by retrieved context, the KB must remain fresh, source-traceable, and precisely indexed.

Toward Continual Domain Learning. Current deployment still relies on engineering heuristics, manual KB updates, and fallback rules. A key future direction is to make DBAgent learn from real production interaction like human DBAs: failed or escalated trajectories should become long-term experience, from which the system can discover new *symptom–cause–action* patterns, refine tool-use strategies, update domain knowledge, and improve the policy through continual training. Human governance remains necessary, but should shift from handcrafting every fix to approving, auditing, and safely integrating agent-discovered knowledge.

7 CONCLUSION

In this paper, we address the limitations of LLMs in complex database O&M by introducing DBAgent, an autonomous agent using a novel RL-based framework. By integrating a curated knowledge base for information acquisition, a multi-modal agent for perception, and an RL-based engine for reasoning, DBAgent significantly outperforms strong baselines in diagnostic accuracy and solution quality, particularly on held-out fault scenarios. Our experiments confirm that these components are crucial for high performance and reliability. Within Autopilot for DWS, DBAgent complements detection and alerting with diagnosis and remedy, improving success on hard incidents and producing grounded reports suitable for Autopilot dashboards.

ACKNOWLEDGMENTS

This work is partially supported by NSFC (No. 62472068 and No. 62272086).

REFERENCES

[1] Richard Bellman. 1957. A Markovian decision process. *Journal of mathematics and mechanics* (1957), 679–684.

[2] Darcy G Benoit. 2005. Automatic diagnosis of performance problems in database management systems. In *Second International Conference on Autonomic Computing (ICAC'05)*. IEEE, 326–327.

[3] Sibei Chen, Ju Fan, Bin Wu, Nan Tang, Chao Deng, Pengyi Wang, Ye Li, Jian Tan, Feifei Li, Jingren Zhou, et al. 2025. Automatic Database Configuration Debugging using Retrieval-Augmented Language Models. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–27.

[4] Xu Chen, Haitian Chen, Zibo Liang, Shuncheng Liu, Jinghong Wang, Kai Zeng, Han Su, and Kai Zheng. 2023. Leon: A new framework for ml-aided query optimization. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2261–2273.

[5] Xu Chen, Zhen Wang, Shuncheng Liu, Yaliang Li, Kai Zeng, Bolin Ding, Jingren Zhou, Han Su, and Kai Zheng. 2023. BASE: Bridging the Gap between Cost and Latency for Query Optimization. *Proc. VLDB Endow.* 16, 8 (2023), 1958–1966.

[6] Xu Chen, Ximu Zeng, Yuze Wang, Zibo Liang, Kai Zeng, Han Su, and Kai Zheng. 2026. LEON+: towards robust ML-aided query optimization: X. Chen et al. *The VLDB Journal* 35, 3 (2026), 21.

[7] Yinfang Chen, Huaibing Xie, Minghua Ma, Yu Kang, Xin Gao, Liu Shi, Yunjie Cao, Xuedong Gao, Hao Fan, Ming Wen, et al. 2024. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 674–688.

[8] Ailin Deng and Bryan Hooi. 2021. Graph neural network-based anomaly detection in multivariate time series. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 35. 4027–4035.

[9] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. 2005. Automatic Performance Diagnosis and Tuning in Oracle.. In *CIdR*. 84–94.

[10] Shihan Dou, Ming Zhang, Zhangyue Yin, Chenhao Huang, Yujiong Shen, Junzhe Wang, Jiayi Chen, Yuchen Ni, Junjie Ye, Cheng Zhang, et al. 2026. CL-bench: A Benchmark for Context Learning. *arXiv preprint arXiv:2602.03587* (2026).

[11] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. (2024). arXiv:2401.08281 [cs.LG]

[12] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. 2009. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1246–1257.

[13] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, Vol. 96. 226–231.

[14] Li Hai-Xiang, Li Xiao-Yan, Liu Chang, Du Xiao-Yong, Lu Wei, and Pan An-Qun. 2021. Systematic definition and classification of data anomalies in DBMS (English Version). *arXiv preprint arXiv:2110.14230* (2021).

[15] Xiao He, Ye Li, Jian Tan, Bin Wu, and Feifei Li. 2023. Oneshotstl: One-shot seasonal-trend decomposition for online time series anomaly detection and forecasting. *arXiv preprint arXiv:2304.01506* (2023).

[16] Zengyou He, Xiaofei Xu, and Shengchun Deng. 2003. Discovering cluster-based local outliers. *Pattern recognition letters* 24, 9-10 (2003), 1641–1650.

[17] David Heckerman. 1998. A tutorial on learning with Bayesian networks. *Learning in graphical models* (1998), 301–354.

[18] Shiyue Huang, Ziwei Wang, Yinjun Wu, Yaofeng Tu, Jiankai Wang, and Bin Cui. 2025. OpDiag: Unveiling Database Performance Anomalies Through Query Operator Attribution. *IEEE Transactions on Knowledge and Data Engineering* (2025).

[19] Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516* (2025).

[20] Adam Tauman Kalai, Ofir Nachum, Santosh S. Vempala, and Edwin Zhang. 2025. Why Language Models Hallucinate. arXiv:2509.04664 [cs.CL] https://arxiv.org/abs/2509.04664

[21] Tung Kieu, Bin Yang, and Christian S Jensen. 2018. Outlier detection for multidimensional time series using deep neural networks. In *2018 19th IEEE international conference on mobile data management (MDM)*. IEEE, 125–134.

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

[23] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2118–2130.

[24] Zheng Li, Yue Zhao, Nicola Botta, Cezar Ionescu, and Xiyang Hu. 2020. Copod: copula-based outlier detection. In *2020 IEEE international conference on data mining (ICDM)*. IEEE, 1118–1123.

[25] Ping Liu, Shenglin Zhang, Yongqian Sun, Yuan Meng, Jiahai Yang, and Dan Pei. 2020. Fluxinfer: Automatic diagnosis of performance anomaly for online database system. In *2020 IEEE 39th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 1–8.

[26] Xiaoze Liu, Zheng Yin, Chao Zhao, Congcong Ge, Lu Chen, Yunjun Gao, Dimeng Li, Ziting Wang, Gaozhong Liang, Jian Tan, et al. 2022. PinSQL: Pinpoint root cause SQLs to resolve performance issues in cloud databases. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2549–2561.

[27] Minghua Ma, Zheng Yin, Shenglin Zhang, Sheng Wang, Christopher Zheng, Xinhao Jiang, Hanwen Hu, Cheng Luo, Yilin Li, Nengjun Qiu, et al. 2020. Diagnosing root causes of intermittent slow queries in cloud databases. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1176–1189.

[28] Biao Ouyang, Yingying Zhang, Hanyin Cheng, Yang Shu, Chenjuan Guo, Bin Yang, Qingsong Wen, Lunting Fan, and Christian S Jensen. 2025. RCRank: Multimodal Ranking of Root Causes of Slow Queries in Cloud Database Systems. *arXiv preprint arXiv:2503.04252* (2025).

[29] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems* 36 (2023), 68539–68551.

[30] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300* (2024).

[31] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. HybridFlow: A Flexible and Efficient RLHF Framework. *arXiv preprint arXiv: 2409.19256* (2024).

[32] Vikramank Singh, Kapil Eknath Vaidya, Vinayshekhar Bannihatti Kumar, Sopan Khosla, Murali Narayanaswamy, Rashmi Gangadharaiah, and Tim Kraska. 2024. Panda: Performance debugging for databases using LLM agents. (2024).

[33] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*. 1009–1024.

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

[35] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

[36] Dong Young Yoon, Ning Niu, and Barzan Mozafari. 2016. Dbsherlock: A performance diagnostic tool for transactional databases. In *Proceedings of the 2016 international conference on management of data*. 1599–1614.

[37] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 international conference on management of data*. 415–432.

[38] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. Restune: Resource oriented tuning boosted by meta-learning for cloud databases. In *Proceedings of the 2021 international conference on management of data*. 2102–2114.

[39] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems* 36 (2023), 46595–46623.

[40] Xuanhe Zhou, Guoliang Li, Zhaoyan Sun, Zhiyuan Liu, Weize Chen, Jianming Wu, Jiesi Liu, Ruohang Feng, and Guoyang Zeng. 2023. D-Bot: Database Diagnosis System using Large Language Models. CoRR abs/2312.01454 (2023).