From General Agents to RCA Experts: A Self-Evolving Harness for Root Cause Analysis

Haiyu Huang*, Jiewei Lyu†, Zhihan Jiang*‡, Jinyang Liu‡,
Xiao He‡, Tieying Zhang‡, Wu Xiang‡, Michael Lyu*
*The Chinese University of Hong Kong, {hyhuang25, zhjiang22, lyu}@cse.cuhk.edu.hk
†Individual Researcher, jiewei.lyu@gmail.com
‡ByteDance, {jinyang.liu, xiao.hx, tieying.zhang, xiangwu}@bytedance.com

***Abstract*—Automated root cause analysis (RCA) with large language models (LLMs) has drawn growing attention. Today, SREs typically automate RCA with LLMs in one of two ways: directly using a general-purpose agent (e.g., Codex or Claude Code) for diagnosis, or building a specialized RCA agent from scratch. As mainstream general agents grow more capable and iterate quickly, our quantitative study finds that the former now often surpasses the latter. Its accuracy, however, still falls short of production needs, and this gap stems mainly from the external adaptation layer outside the agent's general capabilities, namely the harness. We therefore argue that LLM-based RCA should focus on this external harness, reusing the strong general capabilities of a modern agent rather than rebuilding an agent from scratch. A key capability of such a harness is to self-evolve, accumulating system-specific experience from past diagnoses so that it gets better the more it is used. We introduce OpsHarness, a self-evolving RCA harness that turns diagnosis experience into reusable expertise. Its data plane combines layered operational knowledge with an idea-card tool library, while its control plane coordinates setup, diagnosis, evolution, and verification. During evolution, OpsHarness contrasts successful and failed trajectories, converts their evidence into atomic proposals, and admits updates only through a dual-gate verification process designed to prevent overfitting and regression. Across two public benchmarks and an industrial deployment, OpsHarness achieves 59.0% top-1 accuracy, improving over a bare general agent by 63.4% and over baseline RCA agents by 4.02×.**

***Index Terms*—root cause analysis, AIOps, LLM agents, agent harness, self-evolving**

I. Introduction

Modern software is built as large meshes of microservices. When one service degrades, the symptom (e.g., a latency spike or a surge of errors) often surfaces far from its cause and propagates across many dependencies. Root cause analysis (RCA) is the task of locating that cause from the metrics [1], logs [2], and traces [3] a system emits. It is one of the most time-pressured tasks in site reliability engineering. Even with mature observability tooling, a single production incident can take an on-call engineer tens of minutes to localize [4], [5].

Automated RCA has been studied for decades. Early methods localize faults with dependency graphs, causal inference, or spectrum analysis [6]–[12]. With the rise of large language models (LLMs), more recent methods build specialized LLM-based RCA agents that plan, call tools, read telemetry, and reason about the cause [13]–[17]. These methods typically start from a model and design the full pipeline from scratch, including the ReAct loop, code execution, and planning, to form a specialized RCA agent or framework.

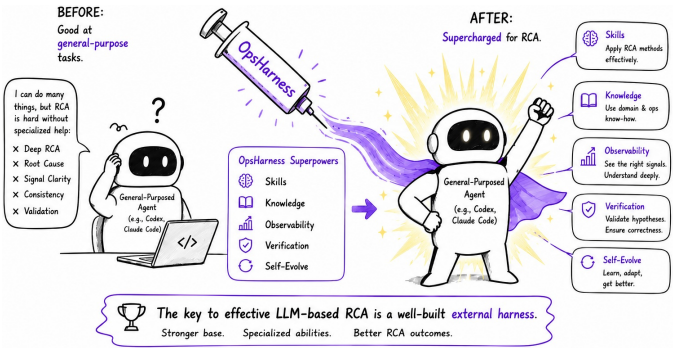

Fig. 1. The key to effective LLM-based RCA is a well-built external harness.

However, LLM agent development has moved to a new stage. General-purpose agent frameworks such as Claude Code [18], OpenAI Codex [19], and the open-source OpenCode [20] now provide mature tool use, code execution, sandboxing, and long-horizon planning out of the box, and they iterate quickly. A mainstream best practice for an agent's general capabilities has therefore emerged, and re-designing these parts (i.e., building an agent from scratch) is hard to get better. We confirm this on RCA with a quantitative study (Section III-A): a bare flagship general-purpose agent is already often better than prior RCA agents that were carefully designed from scratch. On the same backbone (e.g., GPT 5.5), bare Codex answers 45% of OpenRCA queries, well above the 32% reached by the RCA-Agent proposed in OpenRCA [21]; measured across all tested benchmarks, this lead holds on every backbone we test.

Nevertheless, even today's top-tier general-purpose agents still leave substantial room for improvement on RCA tasks, with Codex (GPT-5.5) achieving an average accuracy of only 51.2% on the test datasets, as shown in our study in Sec. III-A. Through our study of the failure cases, this room stems not from a deficit in general ability, but from the agent's lack of diagnostic expertise and unfamiliarity with the target system under diagnosis, much like a broadly capable engineer newly joining a team. Closing it is the responsibility not of the model or general ability of an agent, but of the ***harness***, the external software layer that wraps a general agent or model and adapts it to a specific task by supplying the tools, domain knowledge,

and control logic it operates with [22]–[24]. As models mature, a growing body of work finds that it is increasingly this surrounding layer, rather than the model or general ability alone, that decides how well an agent performs on a given task. The design of this surrounding layer has come to be known as ***harness engineering*** [22], [24].

Building on this observation, we argue that, in the current era of LLM agents, the key to effective LLM-based RCA lies in a well-built external harness (Fig. 1), which should possess two properties. **First**, rather than rebuilding an RCA agent from scratch, the harness should focus on the diagnosis-specific infrastructure that sits outside the general capabilities, thereby compatible with rapidly evolving general agents and reuses their strong general capabilities, while making them better at RCA. **Second, and more importantly**, this harness should continuously ***self-evolve***. As shown in our industrial study in Sec. III-B, what distinguishes an expert SRE from a capable newcomer is deep, accumulated experience of the target system, earned incident by incident (e.g., which metric pattern always means which fault here, how faults propagate between components) Such experience is hard to specify in advance and usually has to be distilled from diagnosis over time. An effective harness should therefore be able to mine reusable best practices, knowledge, and tools from past diagnoses and apply them to later ones, so that, even in unfamiliar systems, it can gradually accumulate experience during diagnosis, self-evolve, and continuously improve.

To this end, we propose ***OpsHarness***, the first self-evolving external agent harness for RCA. Delivered as an external harness compatible with a general-purpose agent, it equips the agent with diagnosis-related skills, tools, and control loops to better handle RCA tasks. OpsHarness is organized into a *data plane* and a *control plane*. The data plane comprises a layered operational knowledge store and an idea-card tool library. The control plane drives the harness lifecycle through four workflows, namely *setup*, *diagnose*, *evolve* and *verify*. The evolve and verify workflows form a self-evolution loop. The evolve stage mines runtime diagnosis trajectories and distills patterns from both correct and incorrect runs into atomic evolution proposals. The verification stage then applies a dual-gate sandbox check to each proposal and only promotes those that pass, enabling OpsHarness to continuously improve while preventing overfitting.

We conduct comprehensive experiments on OpsHarness across two public benchmarks (i.e., OpenRCA [21] and RCAEval [25]), an industrial production deployment. Averaged over the four backbones, OpsHarness reaches 59.0% top-1 accuracy, a 63.4% relative gain over the bare general agent and 4.02× above the specialized agents. We further show that OpsHarness continuously improves through self-evolution during the diagnosis process, with each design component contributing to the overall performance gains. Its overhead is comparable to a bare general-purpose agent, while significantly outperforming baseline specialized RCA agents.

This paper makes the following contributions.

- We propose *OpsHarness*, the first self-evolving external agent harness for RCA. Unlike prior RCA agents built from scratch, OpsHarness focuses on the external layer around diagnosis. It stays compatible with rapidly evolving general agents and reuses their strong general capabilities, while making them better at RCA.
- We design a self-evolution process that learns from both positive and negative diagnosis trajectories under a dual-gate verification control. It takes the harness from being unfamiliar with a target system to gradually distilling that system's best practices while resisting overfitting, so diagnosis improves the more it is used.
- We conduct comprehensive experiments on OpsHarness across two public benchmarks and an industrial production deployment. It reaches 59.0% top-1 accuracy, a 63.4% relative improvement over a bare general agent and a 4.02× improvement over baseline RCA agents, and each component contributes to this gain at a reasonable cost.

II. Background and Related Work

LLM-based RCA. Classical RCA localizes faults with dependency graphs, causal inference, or spectrum analysis, as in MicroRCA [6], Sage [7], Eadro [8], and BARO [9]. With LLMs, recent work builds agents that plan, write code, read telemetry, and reason about the cause. RCAgent [14] and the RCA-Agent [21] act over raw multi-modal telemetry, mABC [15] and Flow-of-Action [16] coordinate role-specialized agents under SOPs, and others assist incident management from tickets and logs, such as RCACopilot [13], Xpert [26], and in-context approaches [27]–[30]. Almost all approaches design a specialized agent from scratch and fit it to a single system, which introduces two key limitations: they cannot leverage rapidly evolving general-purpose agents, and they remain fixed frameworks that do not learn or generalize well to unseen scenarios. OpsHarness, as the first self-evolving external RCA harness, addresses both limitations by being compatible with modern general agents while enabling continual self-evolution during deployment.

Harness engineering and general coding agents. An agent is increasingly understood as a model plus a harness, the external layer that supplies its tools, context, memory, control loop, and evaluation, now studied directly as harness engineering [22]. CoALA [31] named its components, and recent surveys formalize the *agent harness* and recast agent building as *externalizing* capability into memory, skills, and tools [32], [33]. Each component is its own research target, including the agent-computer interface for tools [23], operating-system-style context paging for memory [34], and skill libraries that accumulate reusable procedures [35]–[37]. In terms of general capability, general coding agents such as Claude Code [18], Codex [19], and OpenCode [20] now provide mature tool use, code execution, sandboxing, and long-horizon planning out of the box, and they improve quickly. Their general ability is strong, yet on a specific task they still benefit from a harness layered on top. Prior work typically takes the harness to be everything outside the model; we additionally define ***external harness***, the task-specific layer that wraps a whole

TABLE I
TOP-1 ACCURACY OF GENERAL AGENTS VS. SPECIALIZED RCA AGENTS.

Model	Agent framework	OpenRCA	RCAEval	Overall
GPT-5.5 [40]	Codex	44.9	57.4	**51.2**
	RCA-Agent†	31.8	1.9	16.8
	mABC†	7.6	11.1	9.4
Claude Sonnet 4.6 [41]	Claude Code	25.9	27.8	**26.8**
	RCA-Agent†	26.4	5.6	16.0
	mABC†	2.2	9.3	5.8
GLM-5.2 [42]	Codex	46.1	46.3	**46.2**
	RCA-Agent†	36.8	11.1	24.0
	mABC†	2.2	7.4	4.8
DeepSeek-V4 [43]	Codex	20.2	20.4	**20.3**
	RCA-Agent†	28.0	1.9	14.9
	mABC†	1.2	3.7	2.5

general agent on top of its general capabilities. For software development, external harnesses such as Superpowers [38], an agentic skills framework, and OMX [39], a workflow layer for Codex, already add such an external layer of reusable skills, commands, and workflows in the Dev domain. However, in the Ops domain, such external harnesses are still largely missing. OpsHarness, as the first self-evolving RCA external harness, fills this gap and enables general-purpose agents to achieve stronger capability on RCA tasks through the harness.

III. MOTIVATION

We ground OpsHarness in two observations, drawn from controlled quantitative experiments and from industrial evidence. To obtain solid evidence from a real production setting, we conduct case studies at Company A, a large commercial cloud service provider, using data from production systems.

A. External Harness is Now the Key to LLM-based RCA

General-purpose coding agents already show strong potential on RCA with no RCA-specific design. We run a quantitative experiment across four backbone models (i.e., GPT-5.5, Claude Sonnet 4.6, GLM-5.2, and DeepSeek-V4) and two public benchmarks (i.e., OpenRCA [21] and RCAEval [25]). For each backbone we compare two settings. In the first, we directly connect the base model to a general agent framework (i.e., Codex or Claude Code). In the second, we connect the same model to a specialized RCA agent designed for the task (i.e., RCA-Agent or mABC). Measured by overall top-1 accuracy across both benchmarks, the general agent is the stronger of the two on every backbone, reaching 36.1% on average against 17.9% for RCA-Agent and 5.6% for mABC (Table I). The gap is widest on RCAEval, where the specialized RCA-Agent, designed for OpenRCA, fails to generalize and drop to 5.1%, while the general agent holds at 38.0%.

The reason is that a good RCA agent also depends on the general abilities that flagship agent frameworks have spent years maturing (e.g., context management over long-horizon tasks, multi-round planning, and reasoning). OpenRCA's own analysis of RCA-Agent is telling. Many of its failures stem from gaps in general agentic competence. It is brittle at error recovery and reasons "lazily" with short chains [21]. These are precisely the competencies (i.e., robust tool-use loops, long-horizon planning, and context management) that general agent frameworks now provide out of the box and document as standard practice [22], [23]. Building an RCA agent from scratch means re-implementing, and usually under-implementing, this machinery.

Yet a bare general agent still falls short of production use. It averages only 36.1% overall top-1 accuracy across backbones, and even its strongest configuration (GPT-5.5) reaches 51.2%. The shortfall is not one of reasoning. A general agent behaves like a broadly capable engineer who is new to the team, in that it brings reasoning ability but lacks diagnostic expertise.

A failure from our general-agent runs makes this concrete. On a Telecom query about a latency spike around 02:15, the agent correctly localizes the anomalous window but incorrectly attributes the root cause to a network-error counter on host `os_022`, which shows the largest absolute jump. However, this counter is chronically noisy under load, while the true cause is CPU saturation on container `docker_003`, which deviates only modestly in absolute terms from a near-zero baseline. Misled by raw magnitude and lacking awareness of system-specific noise profiles and operational conventions (e.g., near-zero baseline shifts matter more than volatile swings), the agent localizes a downstream symptom rather than the root cause. This reflects missing system-specific diagnostic knowledge rather than a model limitation; such knowledge is exactly what an external RCA-specific harness is designed to provide, aligning general agents with the target system and enabling effective diagnosis.

Finding 1. Fast-improving general agents now often outperform previous RCA agents. They already bring strong general diagnostic ability to RCA, but their accuracy is bounded by the external supporting layer.

Implication 1. The key to LLM-based RCA is to build an external harness around the agent's general capabilities, one that both reuses the flagship agent's mature general capabilities and makes it perform better on RCA.

B. The Ability to Self-Evolve is Decisive for RCA

For a given system, diagnosis is rarely a one-shot exercise but an ongoing process, as similar incidents often recur over a system's operational lifetime (e.g., 36.2% of incidents with recorded troubleshooting guides occur at least twice [44]). What distinguishes expert SREs from capable newcomers is the ability to continuously summarize best practices and mine experience from each diagnosis, i.e., to self-evolve. This self-evolving capability enables an agent to transition from unfamiliarity with a newly onboarded system to progressively extracting expertise and improving its performance over time, achieving true generalization across systems.

We illustrate this with a real-world production case from Company A (Figure 2). **(1)** On June 15, an on-call SRE received a change-induced alert for service α, where front-end latency spiked and front-end containers showed resource saturation. The SRE observed that database active sessions

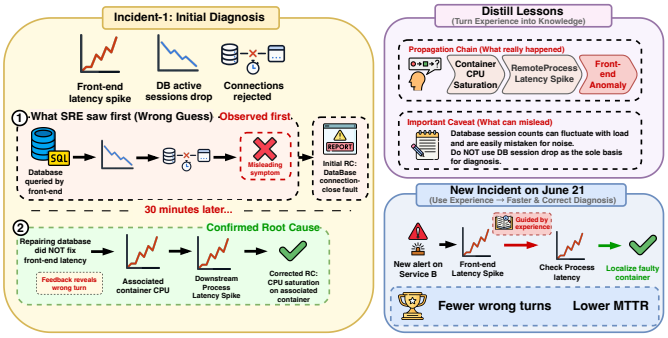

Fig. 2. A continuous diagnosis process on one system, in which an expert SRE accumulates and reuses best practices across incidents.

dropped sharply from 391 to 52, with connections rejected within about 40 ms, and attributed the root cause to a database connection-close fault. However, 30 minutes later, downstream engineer reported that fixing the database did not resolve the issue. Further investigation showed a different associated container where CPU utilization surged from 1.6% to 98%, triggering a downstream RemoteProcess latency spike; the true root cause was CPU saturation on that container. After restart, all services recovered, and the database session drop was identified as incidental noise. **(2)** From this diagnosis, the SRE recorded two lessons: the propagation chain from container CPU saturation to RemoteProcess latency spike and front-end anomaly, and the caveat that database session counts can fluctuate under load and are not reliable standalone signals. **(3)** On June 21, the same SRE handled another alert on service β. Guided by prior experience, the SRE directly checked RemoteProcess latency and container CPU, quickly localized the faulty container, and resolved the incident via restart. This case demonstrates that continuously distilling and reusing diagnostic experience is critical in production, and that expert SRE accuracy stems from such accumulated knowledge, which a strong RCA agent should also be able to self-evolve.

Most operational knowledge and its evolution are still created manually. Engineering teams document expert experience and recurring fixes as Troubleshooting Guides (TSGs). At Microsoft, over 4,000 TSGs are linked to thousands of incidents, and attaching the right TSG reduces mean time to mitigate a severity-2 incident from about 18 to 13 hours, yet TSGs are still written and executed manually [45]. A study of 152 high-severity incidents shows that over 90% are mitigated using such procedures [4], while deeper root-cause knowledge remains in unstructured postmortems that are "not directly reusable" [28], [46]. Overall, the artifacts defining SRE expertise—TSGs, diagnostic skills, and fault-propagation patterns—are learned from past incidents but largely maintained manually or in practitioners' heads.

This motivates an evolving harness that automatically mines each diagnosis for reusable best practices, knowledge, and tools, and applies them to future diagnoses on the same system.

Finding 2. Both SREs and agents need self-evolving to mine expert experience from diagnostic cases and reuse it in subsequent diagnoses. However, this evolution process and the resulting expertise are still largely manual today.

Implication 2. Self-evolution is a core capability of an RCA harness, which enables an agent to generalize to unfamiliar systems, progressively extract system-specific expertise during diagnosis, and continuously improve its performance to achieve effective analysis.

IV. OpsHarness Design

A. Overview

We organize OpsHarness into a **data plane** and a **control plane**, as shown in Fig. 3. The data plane is the per-system state that OpsHarness adapts and evolves. We write it as

$$h = (\mathcal{K}, \mathcal{T}), \tag{1}$$

a layered operational knowledge store $\mathcal{K}$ (Section IV-B) and an idea-card tool library $\mathcal{T}$ (Section IV-C). The control plane is a set of lifecycle workflows over h, namely *setup*, *diagnose*, *evolve*, and *verify*, together with an observability subsystem that records each run. Setup and diagnose adapt and apply the harness (Section IV-D), while evolve and verify form the self-evolution loop that advances it from one version to the next, $h_i \to h_{i+1}$ (Section V).

B. Layered Operational Knowledge

Operational knowledge is not flat. Some of it is experience about how a diagnosis should flow, while some is best practice for one concrete operation step. OpsHarness therefore organizes knowledge into four tiers,

$$\mathcal{K} = (\mathcal{K}_0, \mathcal{K}_1, \mathcal{K}_2, \mathcal{K}_3), \tag{2}$$

each with its own disclosure and update policy.

$\mathcal{K}_0$ stores **general knowledge and background** about the RCA process. At a high level, it defines what a diagnosis should do and what it should produce. $\mathcal{K}_0$ ships with the harness as text in its root documents (e.g., `AGENTS.md` or `CLAUDE.md`) and is loaded on every diagnosis.

$\mathcal{K}_1$ stores **a profile of the target system** that captures its metadata (e.g., its schema, on-disk layout, and component inventory). $\mathcal{K}_1$ is produced automatically once *setup* completes, and it is read in full at the start of each diagnosis. It replaces a per-dataset adapter, in that it tells the agent how to read this system without any hard-coded loader.

$\mathcal{K}_2$ and $\mathcal{K}_3$ are both mined from diagnosis trajectories during evolution (Section V). Together they form the target system's diagnostic knowledge network, the automated counterpart of the expertise an SRE accumulates over a system. We organize this network as a directed graph $G = (\mathcal{O}, E, \mathcal{R})$. The operations $\mathcal{O}$ are the nodes. A directed edge $(o, o') \in E$ records that o' is a sensible next move after o on this system. Each rule in $\mathcal{R}$ is a typed annotation on a diagnostic state. A workflow skeleton is a path in G. The two mined tiers slice

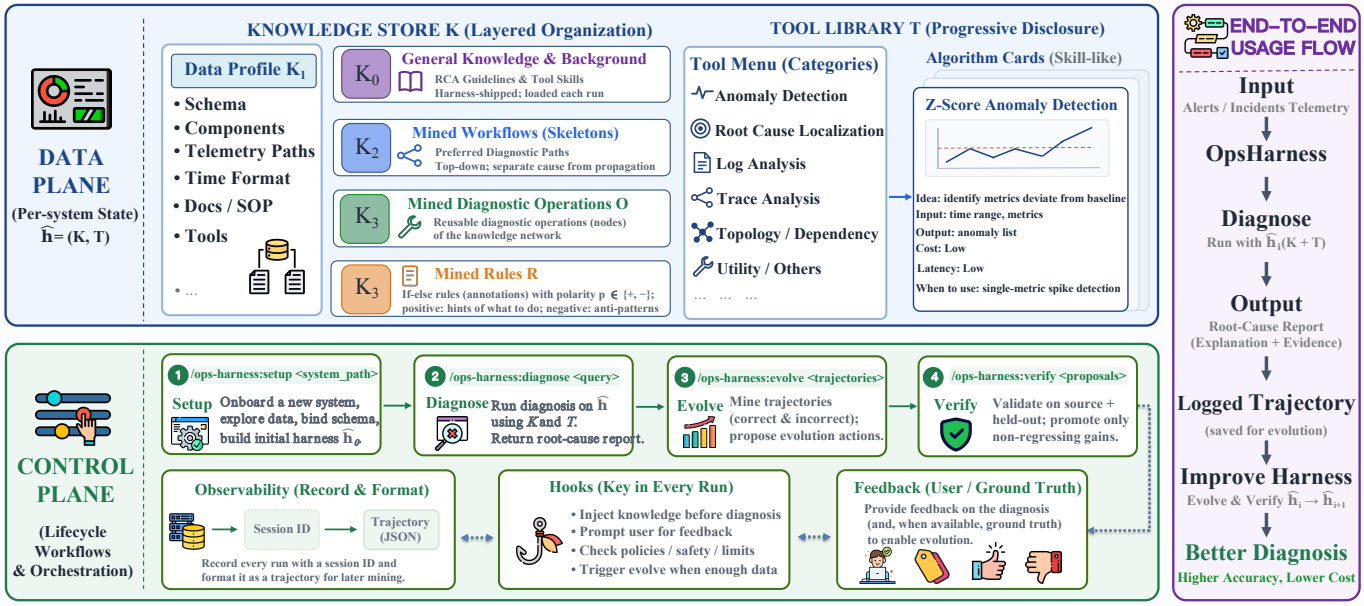

Fig. 3. An overview of OpsHarness design.

this graph by granularity. The coarse workflow paths form $\mathcal{K}_2$, and the fine-grained nodes and annotations form $\mathcal{K}_3$,

$$\mathcal{K}_2 = \Pi \subseteq \mathrm{Paths}(G), \qquad \mathcal{K}_3 = (\mathcal{O}, \mathcal{R}). \tag{3}$$

$\mathcal{K}_2$, mined workflows. A skeleton $\pi = (o_1, \ldots, o_m)$ is an ordered workflow over operations, i.e., a path through the network that represents one of the system's preferred diagnosis workflows (e.g., confirm the front-end KPI window, localize top-down across layers, then separate cause from propagation).

$\mathcal{K}_3$, mined operations and rules. The fine-grained tier, $\mathcal{K}_3 = (\mathcal{O}, \mathcal{R})$. Operations $\mathcal{O}$ are reusable diagnostic moves, the nodes of the network (e.g., isolate container from node CPU, or pivot on the caller of a failed trace). Rules $\mathcal{R}$ are fine-grained if-else diagnostic rules, the annotations of the network. Each rule

$$r = (\phi, \psi, p), \qquad p \in \{+, -\}, \tag{4}$$

pairs a condition ϕ with an action or caveat ψ and a polarity p. A positive rule is a hint of what to do (e.g., a container CPU surge propagates to a RemoteProcess latency spike, then to the front end). A negative rule is an anti-pattern, a thing not to do (e.g., database session counts fluctuate with load and should not be the sole basis for a diagnosis).

Progressive disclosure. OpsHarness stores all knowledge as text under the harness root. Loading every tier on each diagnosis would crowd out the agent's context and degrade its reasoning, so disclosure is progressive. The small, always-relevant $\mathcal{K}_0$ and $\mathcal{K}_1$ are loaded on every diagnosis. Each $\mathcal{K}_2$ workflow carries a short summary; a diagnosis first loads these summaries, then loads in full only the workflow it selects. That workflow in turn drives recall in $\mathcal{K}_3$, pulling in the relevant operations and rules on demand as the diagnosis reaches a state to which they apply.

C. The Idea-Card Tool Library

A diagnosis usually needs analytical tools (e.g., anomaly detection, feature scoring, and suspect ranking). OpsHarness organizes them as idea cards rather than pre-written scripts. An *idea card* is a skill that gives a natural-language specification of one algorithm, stating its core idea, its procedure as light pseudocode, its input and output contract, when to use it, and its key parameters. At diagnosis time the agent reads the card and writes the few lines of code that implement it against the active schema. We make this choice because operational data layouts differ across systems and even across windows, so a once-generated hard-coded script tends to break on a schema it did not anticipate, whereas code the agent writes and iterates against the live context is more likely correct for the diagnosis at hand. An operation in $\mathcal{K}_3$ (Section IV-B) often maps to a tool idea card here, pairing a diagnostic move with the algorithm that realizes it.

The library is disclosed in the same staged way: the agent first reads a one-page menu of algorithms with a short "use when", then opens only the cards it needs, never the whole library. At cold start it holds only scaffolding placeholders and no system-specific tools; as diagnoses accumulate, evolve distills the fixed code logic that recurs across correct trajectories into new tool cards, so the library gradually fills with diagnostic tools tuned to the target system.

D. The Diagnosis Lifecycle

We detail *setup* and *diagnose* here, since *evolve* and *verify* form the self-evolving loop of Section V.

Setup. Setup is a skill that profiles the target system, exposed as a slash command (i.e., `/opsharness:setup`) and run when OpsHarness first onboards a system or when its structure changes. The user only points OpsHarness at the telemetry (e.g., a path on disk, or a database endpoint). OpsHarness then

queries and samples the data, detects the semantic columns from repeated samples (e.g., entity, metric name, and metric value), builds the entity inventory and metric categories, and writes $\mathcal{K}_1$ together with a profile YAML, all with no per-dataset code. If the system already ships with knowledge or tools, the user can pass their access paths, and setup folds what it finds into $\mathcal{K}_1$.

Diagnose. Diagnose is likewise a slash command (i.e., `/opsharness:diagnose <query>`) that the user invokes or that an alert triggers. Given a diagnosis query or a symptom alert, a diagnosis proceeds top-down. The agent loads $\mathcal{K}_0$ and $\mathcal{K}_1$, consults $\mathcal{K}_2$ and $\mathcal{K}_3$ where relevant, and narrows *when* (the anomaly window), *where* (the responsible layer and component), and *why* (the fault type), cross-validating across modalities (e.g., metrics confirm, logs explain, and traces show the impact path). Along the way it implements the idea cards it needs in code against the current diagnostic context and runs them. The output is a ranked root-cause report (i.e., a component, a fault type, and an onset time), and the observability subsystem captures the run as a trajectory. A diagnosis thus uses the agent's general capabilities as connective tissue, assembling the diagnosis-specific knowledge and tools that setup and evolve prepared to carry it through.

V. Self-Evolving in OpsHarness

The self-evolving process runs in three stages (Fig. 4). *Trajectory mining* turns raw sessions into labeled diagnosis trajectories (Section V-A). *Evidence mining and proposal synthesis* summarizes the successful patterns in the positive trajectories and the actionable fixes in the negative ones, emitting a set of evolution actions (Section V-B). *Staged verification* runs each proposal through a dual gate in a sandbox and applies only the ones that pass (Section V-C).

A. Trajectory Mining

Diagnosis trajectory. A session is a time-ordered sequence of events, and slash-command markers delimit its activity. A *diagnosis block* is a maximal span that opens at a *diagnose* command and closes at its end marker. Each tracked block becomes one trajectory τ, while the spans outside OpsHarness logic are discarded as noise.

After a diagnosis is resolved, a feedback hook fires and asks the user for lightweight feedback. The user gives a thumbs-up or thumbs-down and the actual root cause for that diagnosis from downstream. Trajectory mining then reads the feedback for each diagnosis and attaches a label to its trajectory,

$$\ell(\tau) = (a,\, c), \tag{5}$$

the accuracy $a \in \{\text{correct}, \text{partial}, \text{wrong}\}$ derived from the feedback, and the cost c (i.e., tokens and time) measured by observability. The labels split the batch into a positive and a negative set,

$$\mathcal{D}^+ = \{\tau : a(\tau) = \text{correct}\},\ \mathcal{D}^- = \{\tau : a(\tau) = \text{wrong}\}. \tag{6}$$

B. Evidence Mining and Proposal Synthesis

The evolver mines $\mathcal{D}^+$ and $\mathcal{D}^-$ into a proposal. We define a proposal as a set of atomic, typed operators over the data-plane state. Each operator is one element of

$$\omega \in \{\text{add}, \text{update}, \text{delete}\} \times \{\textsc{skel}, \textsc{op}, \textsc{rule}, \textsc{tool}\}, \tag{7}$$

a proposal is a set $P = \{\omega_1, \ldots, \omega_k\}$, and applying it gives a candidate $h' = \text{Apply}(h_i, P)$. An operator edits exactly one element, namely a skeleton path π in $\mathcal{K}_2$, an operation or a rule in $\mathcal{K}_3$, or a tool card in $\mathcal{T}$, and thus advances the full data-plane state $h = (\mathcal{K}, \mathcal{T})$.

Learning from correct runs (commonality). Correct trajectories share structure worth making explicit and reusable. The evolver extracts this commonality at several granularities. A recurring whole-diagnosis order becomes or refines the skeleton π. A move that recurs across correct runs becomes an operation node (e.g., isolating container from node CPU appeared in every correct CPU diagnosis), and a block of fixed code that recurs across them is distilled into a tool card in $\mathcal{T}$ (Section IV-C). A recurring observation becomes a positive rule. Conceptually, an element o is promoted when it recurs across a sufficient fraction of correct trajectories,

$$\text{add } o \quad \text{when} \quad \text{freq}(o,\, \mathcal{D}^+) \geq \rho, \tag{8}$$

a criterion the evolver approximates by reasoning over the labeled batch. The recurring order is distilled into a $\mathcal{K}_2$ skeleton, pruned to its decisive steps so that later diagnoses replay the efficient path rather than rediscover it, dropping the exploratory detours that made early runs expensive.

Learning from incorrect runs (contrastive correction). A wrong trajectory reveals where reasoning diverges from the truth. The evolver identifies the *divergence point*—the first step inconsistent with the confirmed root cause—by comparing every step's intent of the trajectory against the ground truth, and converts the resulting corrective difference into an edit. When a correct and an incorrect diagnosis of similar incidents are available, the skipped moves after their divergence point become the fix. Misleaders recurring across wrong runs are distilled into negative rules, while failures caused by outdated evolved knowledge trigger updates or deletions.

C. Staged Dual-Gate Verification

A mined proposal is a hypothesis, not yet an improvement. It may overfit its source cases, or help locally while hurting elsewhere. OpsHarness admits a change only after it passes two gates in an isolated sandbox.

Sandbox. The verifier creates a *stage*, a pristine copy of the live harness, and applies P there to obtain $h' = \text{Apply}(h_i, P)$. The live harness is untouched until promotion.

Inner gate: strictly better on the source cases. The evolved harness must first beat the original harness on the cases it was mined from. For a harness h and a case set D, let $A_h(D)$ be the accuracy rate and $C_h(D)$ the mean cost (in tokens). On the source batch $\mathcal{D}_i$ we measure h_i and h' and compare them.

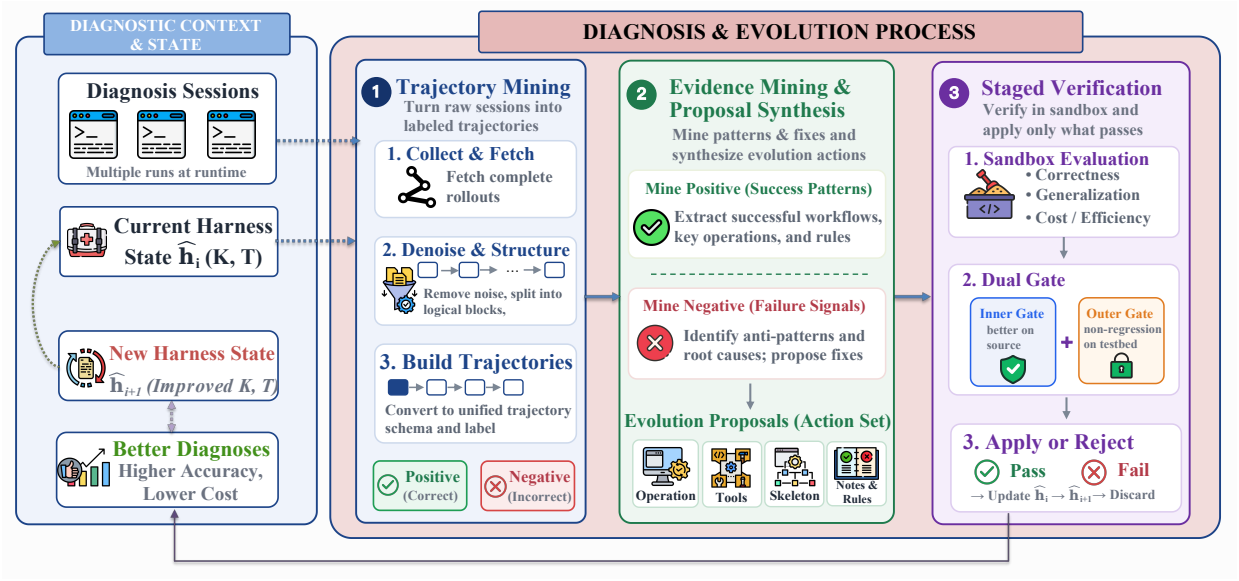

Fig. 4. An overview of the self-evolving loop in OpsHarness.

With deltas $\Delta A = A_{h'}(\mathcal{D}_i) - A_{h_i}(\mathcal{D}_i)$ and ΔC on $\mathcal{D}_i$, and a cost tolerance $\beta = 0.05$,

$$\text{accOK} \equiv \Delta A \geq 0, \text{costOK} \equiv \Delta C \leq \max(\beta\, C_{h_i},\, 1), \quad (9)$$

$$g_{\text{in}} \equiv \text{accOK} \wedge \text{costOK} \wedge (\Delta A > 0 \vee \Delta C < 0). \quad (10)$$

The inner gate requires no regression in accuracy, no material increase in cost, and a strict improvement in at least one objective. The strict clause keeps out no-op changes, and the cost band tolerates a small token increase but rejects accuracy bought with runaway exploration.

Outer gate: non-regression on a held-out testbed. Passing on the source cases does not show that a change generalizes. The verifier maintains a separate testbed $\mathcal{B}$, kept diverse over time and over fault family (e.g., CPU, memory, disk, network, and database). Rather than being fixed upfront, $\mathcal{B}$ is maintained at run time: before every evolution, we cluster the accumulated cases in parallel by fault family and by occurrence time and draw a stratified sample, sized by default to the evolve window. Resampling ahead of each evolve step keeps $\mathcal{B}$ current and disjoint from the mined cases, so a proposal cannot be fit to a static held-out set across iterations. Each testbed case is stored as a truth-free task prompt with a privately held label, and the baselines are refreshed whenever the live harness changes, so the bar is always current. On $\mathcal{B}$ we require only non-regression,

$$g_{\text{out}} \equiv \text{accOK} \wedge \text{costOK} \quad \text{(measured on } \mathcal{B}\text{)}. \quad (11)$$

Acceptance and refinement. A change is promoted if and only if both gates pass,

$$\text{accept} \equiv g_{\text{in}}(h', h_i;\, \mathcal{D}_i) \wedge g_{\text{out}}(h', h_i;\, \mathcal{B}), \quad (12)$$

in which case $h_{i+1} \leftarrow h'$ atomically. If a gate fails, the proposal returns to the evolver, which refines it with the failing cases as fresh negatives and retries, up to three rounds. A proposal that still does not pass is rejected and discarded, and its stage environment is torn down to reclaim space.

VI. Evaluation

- **RQ1:** How effective is OpsHarness, especially after self-evolution?
- **RQ2:** How much does each design of *OpsHarness* contribute?
- **RQ3:** What is the cost of OpsHarness?
- **RQ4:** How does OpsHarness perform on an industrial deployment?

A. Experimental Setup

1) Datasets: We evaluate on two public RCA benchmarks and one industrial dataset. The two public benchmarks together cover a broad range of systems, fault types, and noise profiles. **OpenRCA** [21] contains 335 real-world cases drawn from 3 enterprise software systems, namely a telecom system (Telecom), a banking system (Bank), and a market system (Market). The dataset involves 46 nodes, 68 containers, and 176 service meshes, while covering 28 distinct root-cause categories, providing comprehensive evaluation across systems of different scales and diverse fault types.

RCAEval [25] contains 270 labeled cases over 3 widely used open-source microservice benchmarks, namely Online Boutique, Sock Shop, and Train Ticket. Faults are injected with chaos engineering and cover CPU, memory, network, disk, and code-level anomalies. RCAEval involves 97 monitored entities and 87 microservices, also spanning all 3 telemetry modalities.

Industrial dataset. We additionally use a production change-anomaly dataset from Company A. The telemetry is reported through Prometheus and persisted to a distributed log store. The dataset covers 773,340 data points and 88 confirmed anomalies that occurred during change windows. Labels come from on-call SREs feedback combined with the pulled-back raw logs.

Train/test split. For every (sub)dataset we hold out the last 20% of cases in time order as the test set and use the earlier 80% for warm-up (i.e., self-evolution for OpsHarness and

TABLE II

OVERALL EFFECTIVENESS OF 24 INSTANCES (FOUR BACKBONES × SIX FRAMEWORKS) ON THE SIX SUB-DATASETS OF OPENRCA AND RCAEVAL. Specialized RCA agents are marked †. Within each backbone block, **OpsHarness** (our method) is in **bold** and the best value in each column is underlined.

Framework	OpenRCA [21]									RCAEval [25]									Final A@1
	Telecom			Bank			Market			Online Boutique			Sock Shop			Train Ticket			
	A@1	A@3	Avg	A@1	A@3	Avg	A@1	A@3	Avg	A@1	A@3	Avg	A@1	A@3	Avg	A@1	A@3	Avg	
GPT-5.5																			
RCA-Agent†	36.4	45.5	55.0	35.7	64.3	65.0	23.2	53.6	55.0	0.0	11.1	46.0	5.6	5.6	29.0	0.0	0.0	24.0	16.8
mABC†	9.1	18.2	14.0	10.7	21.4	35.0	3.1	9.8	32.0	16.7	33.3	76.0	11.1	22.2	61.0	5.6	11.1	61.0	9.4
Codex (Direct)	54.5	54.5	64.0	57.1	64.3	72.0	23.2	46.4	59.0	66.7	83.3	95.0	61.1	77.8	89.0	44.4	72.2	83.0	51.2
Codex (ICL)	27.3	36.4	50.0	46.4	53.6	65.0	20.0	40.0	49.0	55.6	77.8	93.0	61.1	61.1	85.0	61.1	88.9	94.0	45.3
OpsHarness (no-evolve)	54.5	54.5	64.0	46.4	57.1	62.0	26.8	43.3	58.0	66.7	72.2	91.0	72.2	72.2	91.0	50.0	77.8	89.0	52.8
OpsHarness	**72.7**	**72.7**	**77.0**	**64.2**	**71.4**	**78.0**	**37.1**	**66.5**	**72.0**	**72.2**	**88.9**	**96.0**	**77.8**	**88.9**	**93.0**	**72.2**	**94.4**	**96.0**	**66.0**
Claude Sonnet 4.6																			
RCA-Agent†	9.1	18.2	32.0	53.6	67.9	79.0	16.5	33.5	55.0	11.1	22.2	59.0	5.6	5.6	29.0	0.0	5.6	44.0	16.0
mABC†	0.0	0.0	0.0	3.6	25.0	40.0	3.1	6.7	30.0	11.1	33.3	76.0	11.1	11.1	59.0	5.6	22.2	69.0	5.8
Claude Code (Direct)	36.4	45.5	55.0	24.2	36.3	61.0	17.0	26.3	38.0	38.9	61.1	85.0	27.8	44.4	76.0	16.7	22.2	76.0	26.8
Claude Code (ICL)	45.5	54.5	64.0	21.4	39.3	58.0	26.7	36.7	49.5	44.4	72.2	89.0	27.8	44.4	72.0	38.9	55.6	76.0	34.1
OpsHarness (no-evolve)	36.4	63.6	73.0	39.3	46.4	61.0	28.6	35.7	50.0	38.9	77.8	89.0	33.3	50.0	79.0	50.0	72.2	89.0	37.8
OpsHarness	**63.6**	**63.6**	**73.0**	**57.1**	**63.7**	**81.9**	**35.7**	**64.2**	**72.0**	**61.1**	**83.3**	**95.0**	**55.6**	**77.8**	**89.0**	**61.1**	**77.8**	**93.0**	**55.7**
GLM-5.2																			
RCA-Agent†	36.4	54.5	59.0	60.7	64.3	77.0	13.4	23.7	40.5	16.7	22.2	56.0	16.7	16.7	33.0	0.0	0.0	28.0	24.0
mABC†	0.0	9.1	14.0	0.0	17.9	34.0	6.7	6.7	20.5	11.1	33.3	74.0	11.1	11.1	52.0	0.0	11.1	59.0	4.8
Codex (Direct)	54.5	63.6	68.0	57.1	57.1	67.0	26.8	50.0	65.0	38.9	77.8	93.0	44.4	72.2	89.0	55.6	72.2	87.0	46.2
Codex (ICL)	36.4	36.4	45.0	46.4	46.4	55.0	16.7	23.3	36.4	61.1	83.3	95.0	50.0	77.8	91.0	50.0	77.8	87.0	43.4
OpsHarness (no-evolve)	54.5	63.6	80.0	57.1	64.3	74.0	28.6	42.9	52.0	66.7	88.9	89.0	44.4	55.6	74.0	55.6	88.9	94.0	51.2
OpsHarness	**72.7**	**81.8**	**87.0**	**57.1**	**64.3**	**74.0**	**42.9**	**50.0**	**61.0**	**77.8**	**88.9**	**89.0**	**55.6**	**66.7**	**76.0**	**88.9**	**94.4**	**98.0**	**65.8**
DeepSeek-V4																			
RCA-Agent†	45.5	45.5	50.0	28.6	35.7	47.0	9.8	9.8	26.0	5.6	5.6	35.0	0.0	5.6	26.0	0.0	0.0	22.0	14.9
mABC†	0.0	0.0	5.0	3.6	14.3	32.0	0.0	3.1	16.0	11.1	33.3	72.0	0.0	5.6	43.0	0.0	0.0	48.0	2.5
Codex (Direct)	18.2	27.3	41.0	32.1	39.3	52.0	10.3	23.2	39.0	27.8	44.4	76.0	22.2	27.8	63.0	11.1	22.2	59.0	20.3
Codex (ICL)	36.4	36.4	41.0	28.6	39.3	50.0	20.0	30.0	42.8	61.1	72.2	91.0	22.2	33.3	61.0	16.7	38.9	67.0	30.8
OpsHarness (no-evolve)	27.2	45.5	46.0	35.7	46.4	61.0	21.4	28.6	50.0	44.4	53.3	76.0	27.8	50.0	80.0	33.3	66.7	72.0	31.6
OpsHarness	**45.5**	**63.6**	**68.0**	**46.4**	**50.0**	**64.0**	**28.6**	**42.9**	**60.0**	**53.3**	**73.3**	**80.0**	**50.0**	**72.2**	**90.0**	**66.7**	**77.8**	**89.0**	**48.4**

demonstration retrieval for in-context-learning baselines). A temporal split mimics a real deployment, in which a system is diagnosed going forward in time, and it prevents any method from seeing future cases. All numbers below are on the held-out test set unless stated otherwise.

2) Baselines and Implementation: We evaluate every method as a (backbone model × agent framework) pair.

Backbone models. We use four recent models spanning open and closed weights and a range of cost and capability, namely GPT-5.5 [40], Claude Sonnet 4.6 [41], GLM-5.2 [42], and DeepSeek-V4 [43], the latest releases at the time of experiments. We exclude very expensive models (e.g., Claude Opus), as they are hard to run at volume a production RCA pipeline requires, whereas our goal is a setting operators can deploy.

Agent frameworks. We compare six frameworks, two specialized and four built on a general agent. **RCA-Agent** [21], the specialized agent proposed with OpenRCA, plans, writes and runs code over telemetry, and reasons about the cause, while **mABC** [15] is a multi-agent framework whose role-specialized agents collaborate to localize the cause. **Direct** connects the backbone to a general agent framework (i.e., Claude Code or Codex) with no RCA harness, and **ICL** adds few-shot in-context learning [47] to it, retrieving three correct rollouts of the most similar past queries into the prompt. **OpsHarness (no-evolve)** is OpsHarness with self-evolution disabled (i.e., the Day-0 substrate only), measuring the harness's cold-start ability, and **OpsHarness** is our full method with self-evolution enabled. We pair Claude Sonnet 4.6 with Claude Code and the other three models with Codex, following each vendor's supported agent. For OpsHarness, we integrate it into the corresponding general-purpose agent. We run every method with the same environment and available information.

3) Evaluation Metrics: We report effectiveness and efficiency metrics standard in RCA evaluation [9], [21]. A diagnosis is scored against the labeled root cause, i.e., a component and, where applicable, a fault type and onset time. **A@**k (top-k exact accuracy, $k \in \{1, 3\}$) is the fraction of cases whose complete root-cause tuple appears among the top-k candidates. **Avg Score** is the mean per-case score giving partial credit across root-cause elements (e.g., the component is correct but the fault type differs). For efficiency we report the average tokens and time per case.

B. RQ1: Effectiveness

To answer RQ1, we run all 24 instances on the two public benchmarks and report A@1, A@3, and Avg Score on each of the six sub-datasets, plus a Final A@1, defined as the mean of the A@1 values (Table II). Three results stand out.

OpsHarness is the strongest framework on every backbone. Averaged over the four backbones, full OpsHarness reaches

59.0% Final A@1, against 41.4% for OpsHarness (no-evolve), 38.4% for ICL, and 36.1% for Direct, while the specialized RCA-Agent and mABC reach only 17.9% and 5.6%. The two gains compound. The cold-start substrate lifts a bare agent by 5.3 points at cold start (Direct 36.1% to no-evolve 41.4%), and self-evolution adds a further 17.6 points (to 59.0%), for a total of 22.9 points over the bare agent.

The cold-start substrate helps most on weaker backbones, and ICL is not enough. On average the cold-start substrate beats both Direct (41.4% vs 36.1%) and ICL (38.4%), but the picture varies by backbone. On the three weaker backbones (GLM-5.2, DeepSeek-V4, and Claude Sonnet 4.6) it clearly beats Direct, with the largest lift on the weakest model (DeepSeek-V4, 20.3% to 31.6%). On the strongest backbone (GPT-5.5) the bare agent is already strong (Direct 51.2%), so the generic substrate adds little at cold start (no-evolve 52.8%) and the harness's edge there comes almost entirely from self-evolution. ICL helps only marginally on average (38.4% vs 36.1% for Direct) and does not improve overall Avg Score (67.0% vs 68.8% for Direct), because retrieving similar rollouts injects raw examples but not distilled, reusable best practices. This matches Finding 2, that what compounds is mined experience, not raw recall of past cases.

Specialized agents trail the general agents. On the same backbone, the from-scratch RCA-Agent and mABC sit well below even Direct (17.9% and 5.6% vs 36.1% Final A@1), echoing the motivation study (Section III-A).

C. RQ2: Contribution of Each Design

To isolate the contribution of self-evolution and of its verification gate, we run three variants of OpsHarness in continuous-diagnosis mode over the training split of both benchmarks: the full system, **OpsHarness (no-evolve)** (the cold-start substrate with self-evolution disabled), and **OpsHarness (no-verify)**(self-evolution on but the verification gate removed, promoting any proposal that helps its source cases). Each variant diagnoses the cases in order; we split the stream into 12 windows, trigger one evolution per window, and report the mean A@1 per window. We repeat this for all four backbones (Fig. 5).

1) Self-Evolution Improves with Use: On every backbone, full OpsHarness climbs steadily across the 12 windows, roughly doubling its A@1 from the first window to the last (e.g., 0.4 to 0.9 on GPT-5.5, 0.5 to 0.9 on GLM-5.2), and ends as the most accurate variant in all four panels. OpsHarness (no-evolve) stays essentially flat (final-window A@1 around 0.2 to 0.6), since a static harness cannot learn the target system. Averaged over the four backbones, the final-window A@1 is 0.83 for full OpsHarness against 0.43 for no-evolve. This gain reflects the system expertise the harness accumulates with use.

2) The Verification Gate Prevents Overfitting: The no-verify variant shows why the gate matters. With self-evolution on but unverified, accuracy first rises as early proposals help, but then stalls or regresses: averaged over the four backbones its final-window A@1 is 0.33, below even the non-evolving

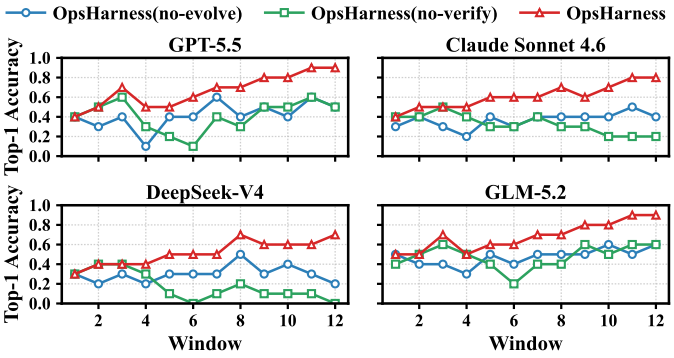

Fig. 5. Per-window A@1 over 12 continuous-diagnosis windows for three OpsHarness variants on four backbones.

TABLE III
RQ2: EVOLVED KNOWLEDGE AND TOOLS ON ONE SYSTEM.

Artifact family	Count	Recall	Precision
Workflow skeleton	6	91.7	73.5
Operation skill	19	82.5	84.3
Note-rule	21	64.7	85.1
Persisted tool	11	61.0	77.2

harness (0.43). The damage concentrates on the weaker backbones, whose proposals overfit most. On DeepSeek-V4 no-verify collapses from a peak of 0.4 to 0.0. A weaker model is more likely to mine a spurious pattern from its source cases. For example, in one cycle the evolver saw repeated failures on network-type cases and proposed up-weighting network anomalies during reasoning; this helped the source cases but, in the next window, mislabeled CPU- and memory-type cases that were previously correct, because the no-verify evolution now surfaced network noise as the cause. The outer gate rejects exactly such proposals: it promotes a change only if it does not regress on a disjoint held-out testbed. With the gate on, every promoted change holds or improves out of sample, which is why full OpsHarness keeps climbing while no-verify decays. On average, 37% of evolution proposals are rejected by the verification gate, and accepted proposals require 1.7 attempts.

3) Knowledge and Tool Mechanisms: To report how the harness learns, we characterize the quality of artifacts it evolves and how often they are used (Table III). For each family we report the count, the fraction of test cases that recall it (Recall), and the fraction of correct cases in which it was recalled (Precision). We can see higher-granularity knowledge is recalled broadly, while fine-grained note-rules and tools are recalled selectively but with high effectiveness when they fire.

D. RQ3: Overhead

1) Diagnosis Cost: We compare the per-case Avg Token and Avg Time of OpsHarness against Direct, ICL, and the two specialized agents (Fig. 6). Although OpsHarness loads skills and knowledge into context, it cuts trial-and-error during diagnosis, so its cost stays in line with the lightweight baselines. Averaged over both benchmarks, it spends about 106k tokens and 325s per case, on par with Direct (112k, 317s) and ICL (106k, 308s). The specialized agents cost far more, about

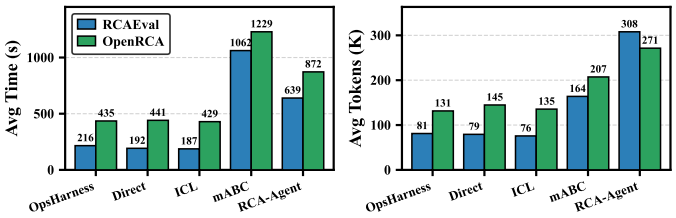

Fig. 6. Per-case diagnosis cost on the two benchmarks.

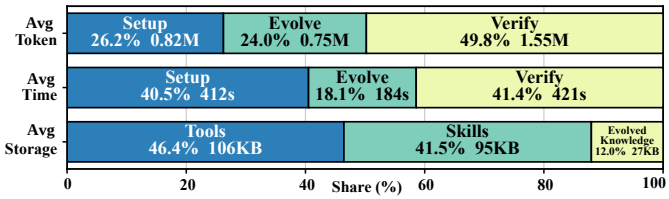

Fig. 7. OpsHarness per-stage cost, and on-disk footprint by component.

1.7× to 2.7× the tokens and 2.3× to 3.5× the wall-clock of OpsHarness , mostly because they retry and re-plan more.

2) Setup, Evolution, and Verification Cost: Beyond per-case diagnosis, OpsHarness pays cost in three non-diagnosis stages: setup (once per dataset), evolution, and verification (once per cycle). On average, one pass costs 0.82M tokens and 412s for setup, 0.75M and 184s for evolution, and 1.55M and 421s for verification (Fig. 7). Verification dominates the token bill (49.8%) because it runs held-out diagnoses in parallel, while evolution is the lightest stage. These stages run rarely, executed in parallel as sidecar, and amortize over many later diagnoses, so the recurring cost is the per-case diagnosis cost above.

3) Harness Footprint: Finally, we report the harness's on-disk footprint after warm-up (Fig. 7, Avg Storage). It totals about 228KB: reusable tools (106KB, 46.4%), the skill library (95KB, 41.5%), and the evolved knowledge learned for the target system (27KB, 12.0%). The whole footprint is negligible relative to the per-case telemetry.

E. RQ4: Industrial Deployment

1) Results on the Industrial Dataset: We deploy OpsHarness on the Company A change-anomaly dataset and compare it to Direct under Codex and open-source Open Code with 3 models, reporting A@1 (Fig. 8). OpsHarness improves over Direct in all six configurations. Averaged over them it reaches 0.74 A@1 against 0.24 for Direct, roughly a 3× lift. The gains do not depend on a proprietary stack: on the open-source Open Code agent, OpsHarness still reaches 0.73 A@1 with the open-source GLM-5.2 and 0.57 with DeepSeek-V4 (vs. 0.23 and 0.12 for Direct), so a fully open agent-and-model stack stays effective on a real production system.

2) Case Study: We trace one anonymized production incident across three consecutive days to show the self-evolution loop concretely. **(1)** On day one, `/opsharness:setup` automatically samples the system's telemetry store and infers its data schema and layout, writing the system profile and $\mathcal{K}_1$ (the initial harness h_0). **(2)** At 12:07 the next day, an upstream business service α alerts that entity-lookup queries keep failing, which triggers `/opsharness:diagnose`. Guided

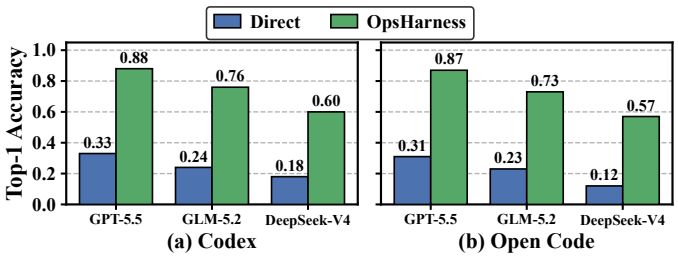

Fig. 8. Industrial effectiveness on the Company A dataset.

by $\mathcal{K}_0$, $\mathcal{K}_1$, and the profile, OpsHarness queries the telemetry and returns a ranked Top-3 root-cause list. An on-call SRE checks them in order: the first two are wrong, but the third is correct, an upstream database synchronization fault that leaves lookups repeatedly hitting empty records. The SRE confirms it with one line of feedback. **(3)** Within the hour, at 13:00, `/opsharness:evolve`. mines the trajectory and distills the causal chain (DB-sync fault → ... → entity-query failure) and its supporting rules into a proposal. It passes the dual-gate verification and is committed as new $\mathcal{K}_2$/$\mathcal{K}_3$ knowledge ($h_i \rightarrow h_{i+1}$), and an SRE confirms the evolved items match the real failure pattern. **(4)** On day three the same failure recurs, and OpsHarness now ranks the database-synchronization fault Top-1 on the first attempt, localizes the faulty component, and finishes in under 2 minutes, reusing a best practice mined automatically rather than written by hand.

VII. Discussion

Performance of OpsHarness under system changes. Such change is itself an argument for an evolving harness over a static one, since OpsHarness degrades gracefully rather than breaking. In the worst case a change invalidates the profile $\mathcal{K}_1$ and part of the mined knowledge, and OpsHarness falls back toward its cold-start state, which already matches or exceeds a bare agent (41.4% vs. 36.1% Final A@1) because $\mathcal{K}_0$ stays valid. Re-running *setup* then rebuilds $\mathcal{K}_1$ with no code, and the harness re-accumulates from post-change diagnoses (Fig. 5). Stale knowledge is retired rather than trusted, as the evolution action includes *update* and *delete* (Section V-B) that correct a rule once later trajectories contradict it.

Failure case analysis. We inspected the cases OpsHarness gets wrong and find two groups. Most are first encounters with a fault mode the harness has not seen before, so the agent reasons without the relevant experience and misses; these failures are progressively mitigated as self-evolution accumulates the corresponding best practices over later diagnoses (Fig. 5). A second group is cases whose telemetry is too tangled for any reusable pattern to surface. We sampled these and find that over 75% are also undiagnosable by human operators, which suggests they are close to unanswerable in the first place rather than a shortcoming specific to OpsHarness.

VIII. Threats to Validity

On *internal* validity, LLM agents are non-deterministic, so we report aggregate accuracy over four backbones, two agent frameworks, and six sub-datasets rather than single runs. On *data leakage*, backbone memorization cannot explain

our gains, since a bare agent on the same model reaches only 36.1% Final A@1 (*Direct*, Section VI-B). We also keep training and test disjoint by a temporal split (Section VI-A), and a guard script blocks any access to ground-truth labels during diagnosis, so self-evolution never sees the answers it is later evaluated on.

IX. Conclusion

With capable general agents, the leverage in LLM-based RCA moves from building an agent to building the harness around one. We present OpsHarness, the first self-evolving external RCA harness, pairing a data plane of layered knowledge and idea-card tools with a control plane. Its self-evolution mines best practices from correct and incorrect diagnoses and gates each change on a dual-gate verification. Across three benchmarks, four backbones, two agent frameworks, and an industrial deployment, OpsHarness achieves a 63.4% improvement in average accuracy over the two bare agents, and a 4.02× improvement over baseline RCA agents on average.

References

[1] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *ICSOC 2018*. Springer, 2018, pp. 3–20.

[2] B. Yu, J. Yao, Q. Fu, Z. Zhong, H. Xie, Y. Wu, Y. Ma, and P. He, "Deep learning or classical machine learning? an empirical study on log-based anomaly detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623308

[3] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," 2010.

[4] S. Ghosh, M. Shetty, C. Bansal, and S. Nath, "How to fight production incidents? an empirical study on a large-scale cloud service," in *Proc. 13th ACM Symp. Cloud Computing (SoCC)*, 2022, pp. 126–141.

[5] S. Zhang, S. Xia, W. Fan, B. Shi, X. Xiong, Z. Zhong, M. Ma, Y. Sun, and D. Pei, "Failure diagnosis in microservice systems: A comprehensive survey and analysis," *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 2025, arXiv:2407.01710.

[6] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root cause localization of performance issues in microservices," in *Proc. IEEE/IFIP Network Operations and Management Symp. (NOMS)*, 2020, pp. 1–9.

[7] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: Practical and scalable ML-driven performance debugging in microservices," in *Proc. 26th ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, pp. 135–151.

[8] C. Lee, T. Yang, Z. Chen, Y. Su, and M. R. Lyu, "Eadro: An end-to-end troubleshooting framework for microservices on multi-source data," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 1750–1762.

[9] L. Pham, H. Ha, and H. Zhang, "BARO: Robust root cause analysis for microservices via multivariate bayesian online change point detection," *Proc. ACM Softw. Eng. (PACMSE), FSE*, vol. 1, no. FSE, pp. 2214–2237, 2024.

[10] G. Yu, P. Chen, Y. Li, H. Chen, X. Li, and Z. Zheng, "Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 553–565. [Online]. Available: https://doi.org/10.1145/3611643.3616249

[11] P. Wang, J. Xu, M. Ma, W. Lin, D. Pan, Y. Wang, and P. Chen, "Cloudranger: Root cause identification for cloud native systems," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018, pp. 492–502.

[12] L. Wu, J. Tordsson, J. Bogatinovski, E. Elmroth, and O. Kao, "MicroDiag: Fine-grained Performance Diagnosis for Microservice Systems," in *ICSE21 Workshop on Cloud Intelligence*, Madrid, Spain, May 2021. [Online]. Available: https://inria.hal.science/hal-03155797

[13] Y. Chen, H. Xie, M. Ma, Y. Kang, X. Gao, L. Shi, Y. Cao, X. Gao, H. Fan, M. Wen, J. Zeng, S. Ghosh, X. Zhang, C. Zhang, Q. Lin, S. Rajmohan, D. Zhang, and T. Xu, "Automatic root cause analysis via large language models for cloud incidents," in *Proc. 19th European Conf. Computer Systems (EuroSys)*, 2024, pp. 674–688, arXiv:2305.15778.

[14] Z. Wang, Z. Liu, Y. Zhang, A. Zhong, J. Wang, F. Yin, L. Fan, L. Wu, and Q. Wen, "RCAgent: Cloud root cause analysis by autonomous agents with tool-augmented large language models," in *Proc. 33rd ACM Int. Conf. Information and Knowledge Management (CIKM)*, 2024, pp. 4966–4974, arXiv:2310.16340.

[15] W. Zhang, H. Guo, J. Yang, Y. Zhang, C. Yan, Z. Tian, H. Ji, Z. Li, T. Li, T. Zheng, C. Chen, Y. Liang, X. Shi, L. Zheng, and B. Zhang, "mABC: Multi-agent blockchain-inspired collaboration for root cause analysis in micro-services architecture," in *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024, pp. 4017–4033.

[16] C. Pei, Z. Wang, F. Liu, Z. Li, Y. Liu, X. He, R. Kang, T. Zhang, J. Chen, J. Li, G. Xie, and D. Pei, "Flow-of-action: SOP enhanced LLM-based multi-agent system for root cause analysis," in *Companion Proc. ACM Web Conf. 2025 (WWW Companion)*, 2025, arXiv:2502.08224.

[17] Y. Han, Q. Du, Y. Huang, J. Wu, F. Tian, and C. He, "The potential of one-shot failure root cause analysis: Collaboration of the large language model and small classifier," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 931–943. [Online]. Available: https://doi.org/10.1145/3691620.3695475

[18] Anthropic, "Claude code," https://www.anthropic.com/claude-code, 2025, accessed: 2026-06.

[19] OpenAI, "Introducing codex," https://openai.com/index/introducing-codex/, 2025, accessed: 2026-06.

[20] SST, "OpenCode: The open source AI coding agent," https://github.com/sst/opencode, 2025, accessed: 2026-06.

[21] J. Xu, Q. Zhang, Z. Zhong, S. He, C. Zhang, Q. Lin, D. Pei, P. He, D. Zhang, and Q. Zhang, "OpenRCA: Can large language models locate the root cause of software failures?" in *Proc. Int. Conf. Learning Representations (ICLR)*, 2025, openReview: M4qNIzQYpd.

[22] V. Trivedy, "The anatomy of an agent harness," LangChain Blog. https://www.langchain.com/blog/the-anatomy-of-an-agent-harness, 2026, accessed: 2026-06.

[23] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2024, arXiv:2405.15793.

[24] Anthropic, "Effective context engineering for AI agents," Anthropic Engineering. https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents, 2025, accessed: 2026-06.

[25] L. Pham, H. Zhang, and H. Ha, "RCAEval: A benchmark for root cause analysis of microservice systems with telemetry data," in *Companion Proc. ACM Web Conf. 2025 (WWW Companion)*, 2025, arXiv:2412.17015.

[26] Y. Jiang, C. Zhang, S. He, Z. Yang, M. Ma, S. Qin, Y. Kang, Y. Dang, S. Rajmohan, Q. Lin, and D. Zhang, "Xpert: Empowering incident management with query recommendations via large language models," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng. (ICSE)*, 2024, arXiv:2312.11988.

[27] T. Ahmed, S. Ghosh, C. Bansal, T. Zimmermann, X. Zhang, and S. Rajmohan, "Recommending root-cause and mitigation steps for cloud incidents using large language models," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, 2023, arXiv:2301.03797.

[28] P. Jin, S. Zhang, M. Ma, H. Li, Y. Kang, L. Li, Y. Liu, B. Qiao, C. Zhang, P. Zhao, S. He, F. Sarro, Y. Dang, S. Rajmohan, Q. Lin, and D. Zhang, "Assess and summarize: Improve outage understanding with large language models," in *Proc. 31st ACM Joint European Softw. Eng. Conf. and Symp. Foundations of Softw. Eng. (ESEC/FSE), Industry Track*, 2023, arXiv:2305.18084.

[29] X. Zhang, S. Ghosh, C. Bansal, R. Wang, M. Ma, Y. Kang, and S. Rajmohan, "Automated root causing of cloud incidents using in-context learning with GPT-4," in *Companion Proc. 32nd ACM Int. Conf. Foundations of Softw. Eng. (FSE Companion)*, 2024, arXiv:2401.13810.

[30] D. Roy, X. Zhang, R. Bhave, C. Bansal, P. Las-Casas, R. Fonseca, and S. Rajmohan, "Exploring LLM-based agents for root cause analysis," in *Companion Proc. ACM Int. Conf. Foundations of Softw. Eng. (FSE Companion)*, 2024, pp. 208–219.

[31] T. R. Sumers, S. Yao, K. Narasimhan, and T. L. Griffiths, "Cognitive architectures for language agents," *Trans. Mach. Learn. Res. (TMLR)*, 2024, arXiv:2309.02427.

[32] Q. Meng, Y. Wang, L. Chen, W. Wu, Y. Li, W. Jiang, Q. Wang, C. Lu, Y. Gao, Y. Wu, and Y. Hu, "Agent harness for large language model agents: A survey," Preprints.org, 2026.

[33] C. Zhou, H. Chai, W. Chen, Z. Guo, R. Shan, Y. Song, T. Xu, Y. Yang, A. Yu, W. Zhang, C. Zheng, J. Zhu, Z. Zheng, Z. Zhang, X. Lou, C. Zhang, Z. Fu, J. Wang, W. Liu, J. Lin, and W. Zhang, "Externalization in LLM agents: A unified review of memory, skills, protocols and harness engineering," arXiv preprint arXiv:2604.08224, 2026.

[34] C. Packer, S. Wooders, K. Lin, V. Fang, S. G. Patil, I. Stoica, and J. E. Gonzalez, "MemGPT: Towards LLMs as operating systems," arXiv preprint arXiv:2310.08560, 2023.

[35] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar, "Voyager: An open-ended embodied agent with large language models," *Trans. Mach. Learn. Res. (TMLR)*, 2024, arXiv:2305.16291.

[36] A. Zhao, D. Huang, Q. Xu, M. Lin, Y.-J. Liu, and G. Huang, "ExpeL: LLM agents are experiential learners," in *Proc. AAAI Conf. Artificial Intelligence (AAAI)*, vol. 38, no. 17, 2024, pp. 19 632–19 642, arXiv:2308.10144.

[37] Anthropic, "Equipping agents for the real world with agent skills," Anthropic Engineering. https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills, 2025, accessed: 2026-06.

[38] J. Vincent, "Superpowers: An agentic skills framework for coding agents," https://github.com/obra/superpowers, 2025, accessed: 2026-06.

[39] Y. Heo, "oh-my-codex: A workflow layer for the OpenAI codex CLI," https://github.com/Yeachan-Heo/oh-my-codex, 2025, accessed: 2026-06.

[40] OpenAI, "Introducing GPT-5.5," https://openai.com/index/introducing-gpt-5-5/, 2026, accessed: 2026-07-01.

[41] Anthropic, "Introducing Claude Sonnet 4.6," https://www.anthropic.com/news/claude-sonnet-4-6, 2026, accessed: 2026-07-01.

[42] GLM-5 Team, "GLM-5: From vibe coding to agentic engineering," 2026. [Online]. Available: https://arxiv.org/abs/2602.15763

[43] DeepSeek-AI, "DeepSeek-V4: Towards highly efficient million-token context intelligence," 2026. [Online]. Available: https://arxiv.org/abs/2606.19348

[44] J. Jiang, W. Lu, J. Chen, Q. Lin, P. Zhao, Y. Kang, H. Zhang, Y. Xiong, F. Gao, Z. Xu, Y. Dang, and D. Zhang, "How to mitigate the incident? an effective troubleshooting guide recommendation technique for online service systems," in *Proc. 28th ACM Joint Meeting European Softw. Eng. Conf. and Symp. Foundations of Softw. Eng. (ESEC/FSE), Industry Track*, 2020.

[45] M. Shetty, C. Bansal, S. P. Upadhyayula, A. Radhakrishna, and A. Gupta, "AutoTSG: Learning and synthesis for incident troubleshooting," in *Proc. 30th ACM Joint European Softw. Eng. Conf. and Symp. Foundations of Softw. Eng. (ESEC/FSE), Industry Track*, 2022, arXiv:2205.13457.

[46] A. Saha and S. C. H. Hoi, "Mining root cause knowledge from cloud service incident investigations for AIOps," in *Proc. 44th Int. Conf. Softw. Eng.: Softw. Eng. in Practice (ICSE-SEIP)*, 2022, arXiv:2204.11598.

[47] Q. Dong, L. Li, D. Dai, C. Zheng, J. Ma, R. Li, H. Xia, J. Xu, Z. Wu, T. Liu, B. Chang, X. Sun, L. Li, and Z. Sui, "A survey on in-context learning," 2024. [Online]. Available: https://arxiv.org/abs/2301.00234