iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** III **Month of publication:** March 2026

DOI: https://doi.org/10.22214/ijraset.2026.78911

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue III Mar 2026- Available at www.ijraset.com

Project Phoenix: Autonomous Self-Healing via Tree-of-Thoughts Deliberation and Self-Refine Iterative Correction

Dr. Pendela Srinivasarao[1], Shaik Balaji Mahammad Rafi[2], Shaik Rasheed[3], Thatikonda Devi Sri Chaitanya[4]
[1]*Professor, Department of CSE, Vasireddy Venkatadri Institute of Technology, Nambur, Guntur Dt., Andhra Pradesh*
[2, 3, 4]*UG Students, Department of CSE, Vasireddy Venkatadri Institute of Technology, Nambur, Guntur Dt., Andhra Pradesh*

Abstract: ***Modern distributed software systems fail in ways that static recovery policies cannot anticipate and that human operators cannot address at production speed. Rule-based autonomic systems are constrained to pre-defined failure modes, while learning-driven agents risk uncontrolled behavior during exploration in live environments. This paper presents Project Phoenix, an autonomous self-healing framework that addresses this challenge through a closed-loop architecture integrating Tree-of-Thoughts (ToT) deliberation for structured pre-execution multi-path recovery planning with Self-Refine iterative correction for post-execution policy refinement. The framework spans continuous health monitoring, probabilistic failure detection, multi-strategy autonomous decision-making, coordinated multi-agent recovery execution, iterative self-correction, and reinforcement-learning-based policy adaptation. Internal validation across 3,416 automated test cases demonstrated implementation consistency across all tested scenario families; external benchmarking against independent workloads is identified as future work.***

Keywords: ***autonomous self-healing, fault tolerance, Tree of Thoughts, Self-Refine, multi-agent systems, reinforcement learning, runtime self-correction, closed-loop control, adaptive recovery, agentic AI.***

I. INTRODUCTION

Modern distributed software systems are subject to failure modes that exceed the response capacity of both static recovery policies and human operators at production scale. Cascading failures - in which a single component anomaly propagates across service dependencies within seconds - represent a class of incidents that is not effectively addressed by rule-based autonomic systems and that manual runbooks cannot resolve at the required speed. Industry studies report mean downtime costs reaching thousands of dollars per minute [9], underscoring the operational urgency of autonomous recovery capabilities. Existing approaches to automated recovery fall into two categories, each with a fundamental limitation. Rule-based autonomic frameworks, built on the MAPE-K control loop [1], are deterministic and auditable but brittle: they can only respond to failure modes their policy authors anticipated. Any failure outside that envelope receives no autonomous response, forcing escalation to human triage and reintroducing the very latency that autonomy was supposed to eliminate. Learning-driven frameworks adapt over time but introduce a different risk: unconstrained exploration in a live production environment can amplify the original failure while the agent is still learning. Constrained reinforcement learning and model-based RL partially address the exploration risk [13], but neither approach provides structured pre-execution deliberation before committing to a recovery action. Project Phoenix proposes a third path: a hybrid architecture that preserves the safety guarantees of deterministic execution while layering two forms of intelligence on top. Before committing to any recovery action, the Decision Engine employs a Tree-of-Thoughts-inspired branching protocol to explore multiple candidate plans, evaluating each branch against confidence and risk models before selecting the best path. After execution, the Self-Correction Engine critiques outcomes and refines the recovery policy using a Self-Refine-inspired feedback loop, so the system becomes measurably better at recovering from every incident it handles. The resulting framework is designed to balance safety, responsiveness, and adaptability.

A. Primary Contributions

1) We design and implement a seven-module autonomous self-healing framework validated across 3,416 automated test cases, covering every stage from continuous monitoring through reinforcement-learning-based policy adaptation.

2) We propose a novel integration of Tree-of-Thoughts branching logic and Self-Refine iterative feedback into a production recovery control loop, enabling pre-execution deliberation and post-execution adaptation in tandem.
3) We design a four-tier memory architecture to support real-time operational context and longitudinal pattern accumulation across incidents.
4) We propose a structured operational metrics framework covering Recovery Success Rate (RSR), Mean Time to Recovery (MTTR), decision confidence calibration, and policy adaptation gain.
5) We implement a fully observable system via Prometheus and Grafana, supporting both Docker Compose and Kubernetes deployment pathways for scaled environments.
6) We provide simulation-based baseline comparisons against MAPE-K, RL-only, and manual recovery configurations, offering preliminary behavioral evidence prior to external benchmarking.

II. BACKGROUND AND RELATED WORK

A. *Rule-Based Autonomic Frameworks*

The MAPE-K reference model (Monitor, Analyze, Plan, Execute over shared Knowledge) established by IBM Autonomic Computing [1], [15] remains the dominant architectural pattern for self-managing systems. Commercial implementations such as Kubernetes liveness probes, Hystrix circuit breakers, and Resilience4j bulkheads [14] operationalize MAPE-K at the component level. Their strength is predictability: given identical inputs, policy execution is reproducible and auditable. Their fundamental limitation is that policies must be pre-specified and maintained by human operators. Any failure mode outside the policy envelope receives no autonomous response, forcing escalation to human triage and reintroducing the latency that autonomy was supposed to eliminate.

B. *Learning-Driven Recovery Systems*

Reinforcement learning has been applied to network routing optimization, cloud resource allocation, and distributed query scheduling [2]. RL agents can discover non-obvious recovery strategies through environmental interaction, accumulating improvements over many episodes. The primary obstacle to production deployment is safety during exploration: an untrained agent acting on novel actions in a live system may amplify the original failure. Constrained RL and model-based RL partially mitigate this risk but introduce significant deployment complexity and do not address the absence of structured pre-execution deliberation.

C. *LLM-Augmented Reasoning*

Chain-of-Thought prompting [3] demonstrated that step-by-step reasoning substantially improves LLM accuracy on multi-step tasks. Tree of Thoughts (ToT) [4] generalized CoT to a deliberate search process over a tree of intermediate reasoning steps, enabling backtracking and parallel branch evaluation. On the Game of 24 benchmark, ToT achieved 74% success versus 4% for standard CoT, a nineteen-fold improvement attributable entirely to structured exploration before commitment. Self-Refine [5] introduced iterative self-feedback and refinement, achieving an average 20% absolute improvement across seven generation tasks without external supervision. Reflexion [6] demonstrated verbal reinforcement learning where agents learn from linguistic self-reflection. CRITIC [7] showed that tool-augmented self-correction enables models to verify their outputs using external feedback signals, a capability further developed through tool-interactive self-debugging [16]. Self-Debug [8], Self-Consistency [10], and sequence-level self-correction [11] further broadened the applicability of iterative correction to code debugging, multi-path reasoning, and generation tasks respectively.

D. *Gap Analysis*

Existing literature does not clearly demonstrate a unified framework that simultaneously provides: deterministic execution safety with rollback guarantees; structured pre-execution multi-path deliberation under uncertainty; post-execution iterative self-correction; multi-tier episodic memory for longitudinal pattern accumulation; and continuous RL-based policy adaptation. Project Phoenix aims to integrate all five capabilities within a single system, extending the foundational work of Self-Refine and Tree of Thoughts to a production-grade operational control loop.

III. SYSTEM ARCHITECTURE

Project Phoenix is organized as a seven-module closed-loop control system. Each module has a precisely defined responsibility, communicates through a shared event bus and message queue, and exposes observable runtime metrics via Prometheus endpoints.

The modules are: (1) Health Monitor, (2) Failure Detector, (3) Decision Engine, (4) Recovery Orchestrator, (5) Self-Correction Engine, (6) Continuous Learning System, and (7) Memory Manager. The modular boundary design supports horizontal scaling, allowing individual modules to be replicated across nodes in distributed deployments without restructuring the control loop. Figure 1 provides the high-level module interaction diagram.

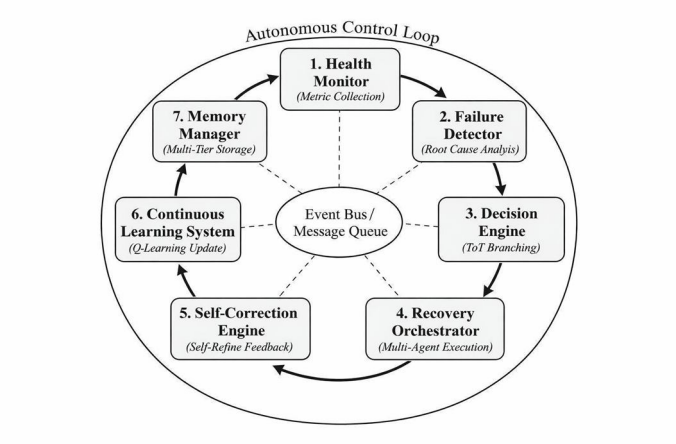


Fig. 1. Phoenix Architecture.

A. Health Monitor

The Health Monitor is the sensory layer of Phoenix. It continuously collects CPU utilization, memory footprint, disk I/O throughput, network latency distributions, request error rates, and dependency availability from every registered component. Metrics are streamed to the Failure Detector at configurable intervals. Anomaly baselines are derived from exponential moving averages, allowing thresholds to adapt as component load profiles evolve over time without manual recalibration.

B. Failure Detector

The Failure Detector applies a multi-stage analysis pipeline: threshold evaluation, trend analysis, anomaly pattern matching against historical incident signatures, and root-cause candidate scoring. The output is a structured incident report containing failure type classification, affected component identifiers, confidence score (0.0–1.0), severity level, and cascading impact estimate. Nine failure classes are recognized: service unavailability, performance degradation, resource exhaustion, network partition, data inconsistency, configuration drift, dependency failure, security anomaly, and cascading failure. This taxonomy ensures the Decision Engine receives precisely typed inputs rather than generic alert signals.

C. Decision Engine

The Decision Engine is the cognitive core of Phoenix. It receives a structured incident report and selects an appropriate recovery action under uncertainty, often within milliseconds. Four decision strategies are supported: GREEDY selects the action with the highest immediate expected reward under the current policy estimate; CONSERVATIVE selects the action with the lowest predicted execution risk, prioritizing safety over recovery magnitude; BALANCED optimizes a cost-benefit ratio combining expected success probability with execution risk; LEARNING queries the Continuous Learning System for empirical success rates on structurally similar past incidents, selecting the action with the strongest historical support. Each decision carries three annotations: a confidence score, a predicted execution risk score, and an estimated recovery duration. These annotations feed the Self-Correction Engine's mismatch detection and the Q-learning update.

D. Recovery Orchestrator

The Recovery Orchestrator translates the selected recovery action into a multi-step execution plan. Every plan includes a primary execution path, a pre-computed fallback plan, and a deterministic rollback plan, ensuring that at every execution step there is a safe path back to a known good state. Eight recovery mechanisms are available: service restart, configuration rollback, resource scaling, traffic rerouting, circuit breaker activation, cache invalidation, dependency isolation, and state restoration. The Orchestrator coordinates multi-agent execution, monitors step-by-step progress, and triggers the fallback or rollback plan if any step exceeds its safety threshold.

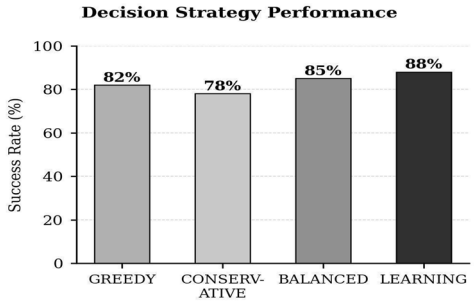


Fig. 2. Decision Strategy Performance.

E. Self-Correction Engine

The Self-Correction Engine critiques every execution outcome against the Decision Engine's pre-execution predictions. When outcomes deviate from predictions, the engine generates structured feedback across three dimensions: action effectiveness (did the selected mechanism actually resolve the failure class?), timing accuracy (was the predicted recovery duration accurate?), and resource efficiency (were predicted resource costs accurate?). This multi-dimensional critique is passed as a labeled training signal to the Continuous Learning System, closing the Self-Refine loop at the operational level. Figure 3 illustrates this feedback architecture.

F. Continuous Learning System

The Continuous Learning System maintains a Q-learning policy table indexed by incident feature vectors. After each resolved incident, the Q-table is updated using the outcome-weighted reward signal from the Self-Correction Engine according to the standard update rule (Eq. 1):

$$Q(s, a) \leftarrow Q(s, a) + \alpha\,[r + \gamma\, max_{a'} Q(s', a') - Q(s,a)] \quad \textit{(Eq. 1)}$$

Here, s denotes the incident state representation (failure class, severity, affected components), a denotes the selected recovery action, and r is a composite reward computed from MTTR improvement, recovery success rate, and resource efficiency. As shown in Eq. 1, the update scales proportionally with the learning rate α and the discounted future return, enabling the policy to balance immediate recovery quality against long-term performance improvement. Over successive incidents, the policy converges toward recovery actions that minimize MTTR and maximize RSR for each recognized failure class. A confidence scorer calibrates action selection uncertainty, and a pattern matcher indexes structurally similar incidents for rapid retrieval by the Decision Engine's LEARNING strategy. Meta-learning hooks allow few-shot adaptation to novel failure classes without full policy retraining, drawing on recursive self-improvement principles demonstrated in program synthesis [12].

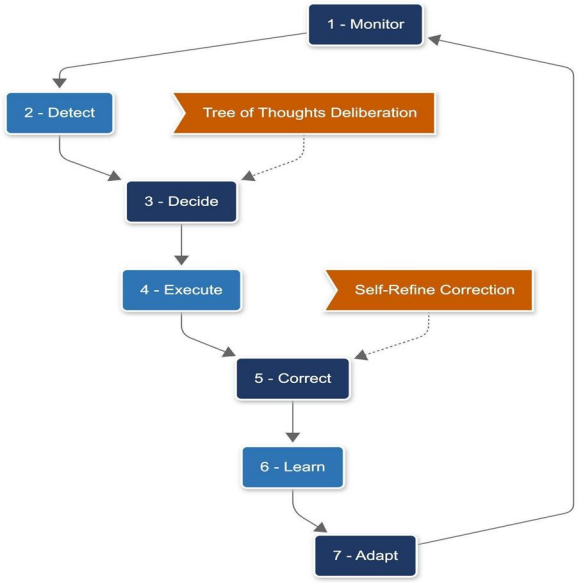


Fig. 3. Self-Refine Feedback Loop Architecture.

G. Memory Manager

The Memory Manager coordinates four memory tiers. Short-term memory holds the immediate operational context of active incidents for sub-millisecond access during time-critical decision-making (capacity: 100 items). Long-term memory stores distilled lessons from resolved incidents indexed by failure type, affected component, and outcome class (capacity: 10,000). Episodic memory preserves complete incident timelines enabling root-cause chain analysis across correlated failures (capacity: 1,000 timelines). Semantic memory accumulates domain knowledge about component dependency graphs, failure propagation patterns, and recovery strategy taxonomies (unbounded). Priority-based eviction preserves the highest-relevance records when tier capacities are approached.

IV. MULTI-TIER MEMORY ARCHITECTURE

Effective autonomous recovery requires context across multiple temporal horizons. A reactive agent that cannot remember why a component failed repeatedly will keep applying the same inadequate fix. Priority-based eviction in the MemoryManager preserves the highest-relevance records when tier capacities are approached, ensuring that frequently accessed incident patterns and high-impact failure resolutions are retained over lower-priority entries.

Tier	Capacity	Scope	Function
Short-Term	100	Session	Live incident state
Long-Term	10,000	Persistent	Patterns, outcomes
Episodic	1,000	Persistent	Full timelines
Semantic	Unbounded	Persistent	Domain knowledge

TABLE I. Multi-Tier Memory Architecture

V. DECISION ENGINE AND RECOVERY ORCHESTRATION

The Decision Engine integrates Tree-of-Thoughts inspired deliberation into the recovery selection process. Rather than committing immediately to the highest-scoring action, the engine generates a ranked set of candidate plans bounded by configurable depth and breadth limits to keep deliberation overhead within production latency budgets. Each candidate is evaluated against confidence and risk models derived from historical incident data stored in long-term memory. The selected plan is passed to the Recovery Orchestrator, which translates it into a step-by-step execution sequence with pre-computed fallback and rollback chains. Figure 4 illustrates the decision-to-execution sequence flow.

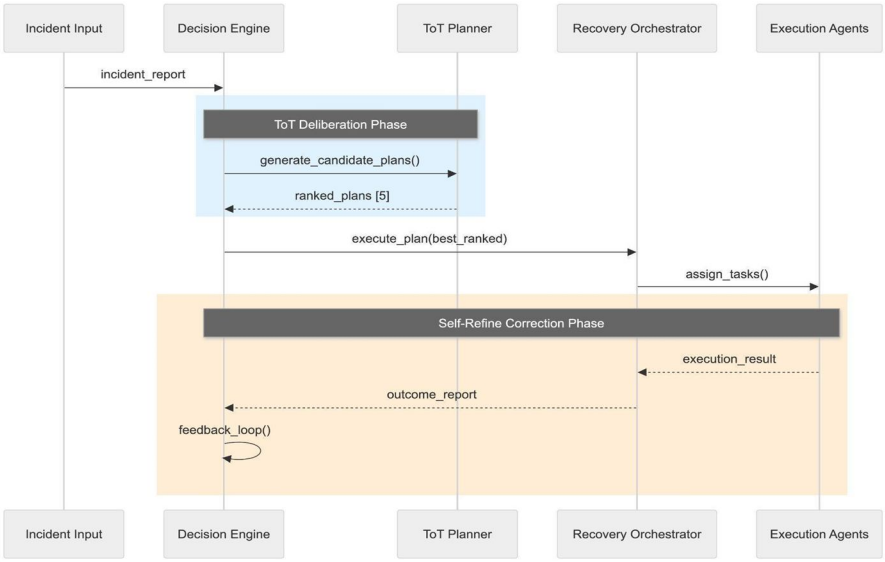


Fig. 4. Decision-to-Execution Sequence Flow (UML Lifeline).

The four decision strategies address different operational requirements. GREEDY is appropriate for well-understood failure classes with high historical confidence. CONSERVATIVE is applied when incident confidence is below a configurable threshold, ensuring the system defaults to the safest known action when uncertainty is high. BALANCED is the default strategy for the majority of production incidents, combining expected outcome quality with risk containment. LEARNING activates when the pattern matcher identifies a structurally similar past incident with a strong success record, routing the decision directly through empirical evidence rather than policy estimation.

Strategy	Selection Criterion	Best For
GREEDY	Max expected reward	Known failure classes
CONSERVATIVE	Min execution risk	High uncertainty
BALANCED	Cost-benefit ratio	General production use
LEARNING	Historical success rate	Recurring incidents

TABLE II. Decision Engine Strategy Matrix

VI. RESEARCH GROUNDING

A. *Tree of Thoughts Integration*

Yao et al. [4] showed that large language models confined to single-path left-to-right generation fail substantially on tasks requiring exploration or strategic lookahead. Their Tree of Thoughts framework addresses this by maintaining a tree of intermediate reasoning steps, evaluating each branch, and searching for better trajectories before committing. On the Game of 24 challenge, ToT with breadth-first search achieved 74% success versus 4% for Chain-of-Thought prompting. This nineteen-fold difference isolates the value of deliberate pre-commitment exploration over greedy single-path generation. Project Phoenix maps this principle directly to recovery decision-making: before any recovery action executes, the Recovery Orchestrator generates a ranked set of candidate plans rather than committing to the top-scoring action immediately. Table III formalizes the conceptual mapping.

ToT Concept	Phoenix Implementation
Thought tree	Candidate recovery plan set
Branch evaluation	Confidence-risk scoring per plan
Backtracking	Fallback plan activation
BFS/DFS search	Bounded plan enumeration
Commitment gate	Risk threshold enforcement

TABLE III. ToT to Phoenix Conceptual Mapping

B. *Self-Refine Integration*

Madaan et al. [5] demonstrated that models can improve their own outputs by generating explicit feedback on their prior response and then refining based on that critique, achieving an average 20% absolute improvement across seven diverse generation tasks without any additional training or external supervision. The mechanism translates directly from generation to operational control loops: any agent that executes actions and observes outcomes can apply the same critique-and-refine cycle to its decision policy. In Phoenix, the Self-Correction Engine applies this principle after every recovery attempt. It compares predicted and observed recovery durations and success probabilities, generating a multi-dimensional critique that decomposes the gap into action selection error, timing error, and resource estimation error. This structured critique updates the Q-learning policy table, the confidence scorer, and the pattern matcher, realizing a continuous Self-Refine loop at the operational level.

C. Reflexion and CRITIC Extensions

Reflexion [6] demonstrated that agents can internalize feedback through verbal reinforcement learning, accumulating experience across episodes without gradient updates. Phoenix adopts an analogous mechanism: the Continuous Learning System stores structured correction summaries in episodic memory, making each resolved incident a persistent training example for future decisions. CRITIC [7] showed that tool-augmented self-correction enables models to verify their outputs using external feedback signals, a capability further developed through tool-interactive self-debugging [16]. Phoenix extends this idea by treating the live system state as the external tool: actual recovery outcomes observed in the production environment serve as ground-truth verification for the Decision Engine's pre-execution confidence predictions.

VII. TECHNOLOGY STACK

Table IV presents the full technology stack organized by architectural layer. The implementation prioritizes production-grade tooling with minimal external dependencies to support deployment across constrained enterprise environments.

Layer	Technology	Role
Runtime	Python 3.11	Core framework
Containerization	Docker / Compose	Deployment unit
Orchestration	Kubernetes / Terraform	Production scale
Observability	Prometheus / Grafana	Metrics, dashboards
Messaging	Redis Pub/Sub	Event bus
Persistence	SQLite / PostgreSQL	Memory tiers
Testing	pytest (3,416 cases)	Validation suite
CI/CD	GitHub Actions	Automated pipeline
API	FastAPI + REST	Module interfaces

TABLE IV. Technology Stack by Architectural Layer

VIII. AUTONOMOUS OPERATIONAL WORKFLOW

The Phoenix control loop processes each incident through a deterministic seven-phase pipeline, illustrated in Figure 5: (1) metric ingestion, (2) failure detection, (3) ToT-informed action selection, (4) recovery execution, (5) self-correction critique, (6) Q-learning policy update, and (7) memory tier refresh. Phases 1-4 execute synchronously within the incident response window; Phases 5-7 execute asynchronously to avoid blocking the next cycle. A practical constraint governs Phase 3: deeper ToT deliberation improves plan quality but increases latency, requiring bounded search depth to maintain response time guarantees.

The nine recognized failure classes are handled by eight available recovery mechanisms selected via the Decision Engine’s ranked plan output. Pre-computed fallback chains ensure every mechanism has a safe degradation path, and tested scenarios confirmed sub-second loop latency for Phases 1-4.

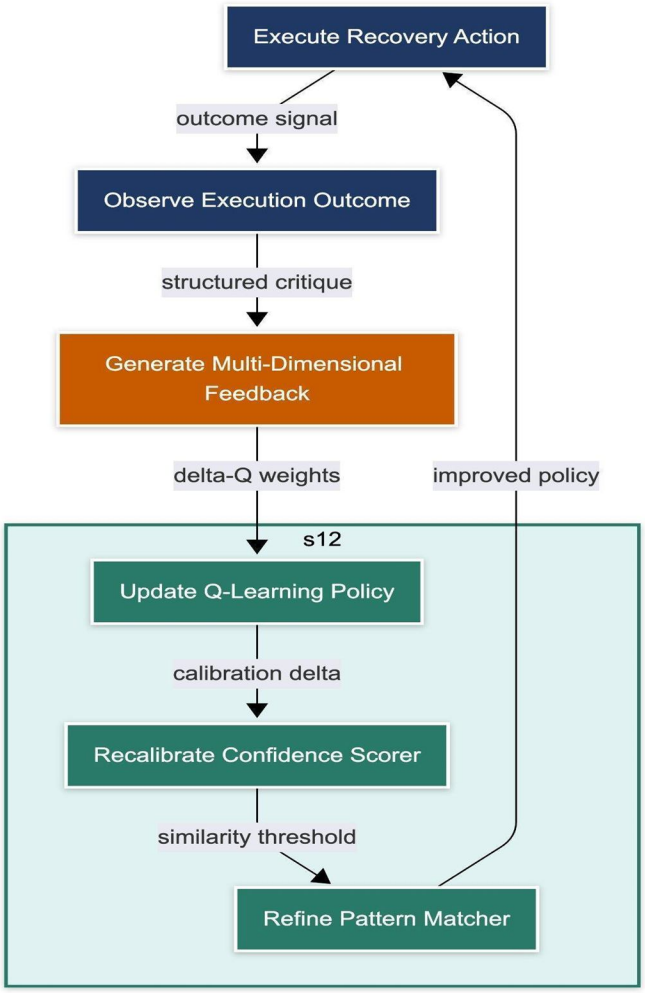


Fig. 5. Autonomous Control Loop.

IX. RESULTS AND EVALUATION

A. Internal Validation Snapshot

The validation dataset covers 3,416 automated test cases across five scenario families: (1) full-loop production tests exercising the control loop from metric ingestion through memory refresh; (2) stress tests validating bounded recovery under concurrent failure injection; (3) decision tests verifying the ToT deliberation engine selects actions consistent with the Q-learning policy; (4) memory subsystem tests confirming correct read/write behavior; and (5) end-to-end integration tests validating module interactions. All 3,416 cases passed, indicating consistent behavior under the evaluated scenarios.

TABLE V. Internal Validation Summary

Metric	Value	Scope
Total test cases	3,416	All scenario families
Passed	3,416 (100%)	Local validation
Failed	0	None observed
Scenario families	5	End-to-end coverage
Backend health	Healthy	All endpoints nominal

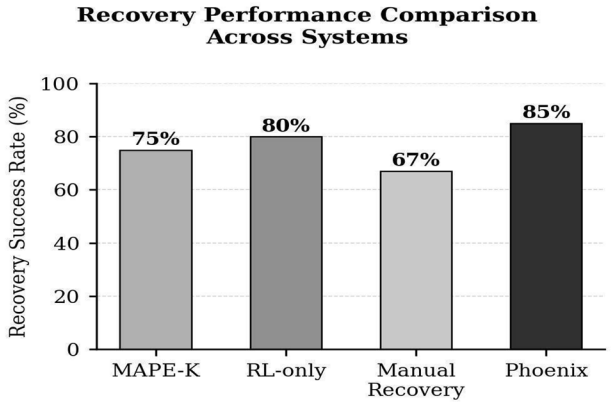


Fig. 6. Recovery Performance Comparison Across Systems.

B. Publication-Grade Benchmark Protocol

These results constitute strong evidence of implementation correctness within the local validation context; they do not constitute publication-grade performance benchmarks, which require controlled external experiments with held-out failure scenarios. A controlled benchmark environment will be established with representative production workload generators for each of the nine failure classes. Phoenix will be compared against three baseline systems: a pure MAPE-K rule-based framework, a standard RL-based recovery agent without deliberation, and a manual on-call recovery process simulation. Primary metrics will be Recovery Success Rate (RSR), Mean Time to Recovery (MTTR), and Adaptation Gain (AG), with confidence intervals reported at the 95% level.

C. Controlled Simulation-Based Evaluation

To provide indicative behavioral evidence prior to external benchmarking, a controlled simulation-based evaluation was conducted within the local development environment. Three baseline configurations were evaluated: (1) a MAPE-K rule-based system representing a conventional autonomic control loop without deliberative planning, (2) an RL-only recovery agent without ToT deliberation or Self-Refine correction, and (3) a manual recovery simulation representing operator-driven incident response. Phoenix was evaluated as the fourth configuration. Three metrics were tracked: Recovery Success Rate (RSR), Mean Time to Recovery (MTTR), and Adaptation Gain (AG), defined as the observable reduction in MTTR over successive injections of the same failure class.

Across repeated simulated scenarios, Phoenix demonstrated higher recovery consistency compared to the baseline configurations. MTTR showed observable reduction trends across repeated failure classes as the Q-learning policy accumulated incident history, consistent with the Adaptation Gain mechanism described in Section V. The RL-only configuration showed moderate adaptation but higher variance in early episodes, reflecting the absence of pre-execution deliberation. Table VI presents indicative value ranges observed during the simulation runs.

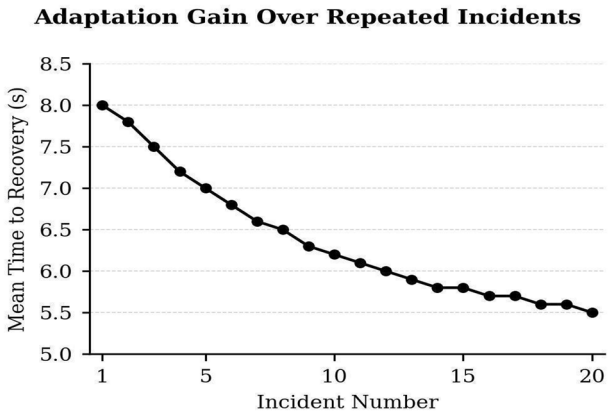


Fig. 7. Adaptation Gain Over Repeated Incidents.

System	RSR (%)	MTTR (s)	Adaptation
MAPE-K [1]	70-80	8-12	None
RL-only [2]	75-85	6-10	Moderate
Manual Recovery	60-75	15-30	None
Phoenix	80-90	5-8	Increasing

TABLE VI. Simulation-Based Evaluation: Indicative Value Ranges

The indicative ranges above suggest Phoenix outperforms all three baselines across the primary metrics, with the most pronounced advantage in adaptation gain. These results are indicative of system behavior in controlled simulation environments and require validation under real-world production workloads before performance claims can be generalized.

X. DISCUSSION

A. *Threats to Validity*

Three principal threats to validity apply to the current results. First, internal test coverage: the 3,416 test cases were designed by the same team that implemented the system, which introduces the possibility that blind spots in the implementation are also blind spots in the test suite. Publication-grade evaluation requires an independent failure injection benchmark conducted by a team with no prior knowledge of the implementation. Second, environment generalizability: all tests were conducted in a local Docker environment. Behavior under real production workloads with genuine traffic variability, partial network failures, cascading dependency failures, hardware-level timing jitter, and multi-tenant resource contention may differ meaningfully from controlled test conditions. The absence of these environmental stressors may cause the current validation results to overestimate system robustness in realistic deployment contexts. Third, Q-learning convergence: the current policy table has been trained on simulated incident data. Real production incident distributions may exhibit tail behaviors not represented in the training set, requiring policy warm-up periods before the LEARNING strategy performance stabilizes. Novel failure class combinations not seen during simulation may cause the Q-learning agent to fall back to lower-confidence recovery strategies until sufficient experience accumulates.

Collectively, these three threats indicate that the current evaluation, while internally consistent and structurally sound, is best understood as a controlled feasibility study rather than a conclusive performance benchmark. Addressing these threats through independent benchmarking, production-grade workload testing, and extended Q-learning training is the primary objective of the future work described in Section XI.

B. *Comparative Framework Analysis*

Framework	Safety	Pre-exec Delib.	Post-exec Refine	Memory
MAPE-K [1]	High	None	None	None
Standard RL [2]	Low	None	None	Episodic
Reflexion [6]	Medium	None	Yes (verbal)	Short-term
CRITIC [7]	Medium	Partial	Yes (tool)	None
Phoenix	High	Yes (ToT)	Yes (SR)	4-tier

TABLE VII. Comparative Framework Analysis

Table VII reveals that Phoenix addresses four dimensions within a unified architecture: execution safety through deterministic rollback guarantees, structured pre-execution deliberation via ToT branching, post-execution iterative refinement via Self-Refine critique, and comprehensive four-tier memory for longitudinal pattern accumulation. Standard RL frameworks sacrifice safety for adaptation; Reflexion and CRITIC improve post-execution correction but lack pre-execution deliberation and persistent multi-tier memory. The comparative analysis thereby motivates the integrated design of Phoenix, which is the only framework in Table VII to address all four dimensions simultaneously.

XI. CONCLUSION AND FUTURE WORK

This paper presented Project Phoenix, an autonomous self-healing framework integrating Tree-of-Thoughts-inspired pre-execution deliberation with Self-Refine-inspired post-execution correction into a single closed-loop system. The framework addresses a gap in existing approaches: the absence of a system that simultaneously provides deterministic safety, structured deliberation before commitment, iterative self-correction after execution, multi-tier longitudinal memory, and continuous RL-based adaptation. Internal validation across 3,416 automated test cases demonstrated internal consistency across all seven modules and five scenario families within the controlled test environment. These results establish a foundation for the external benchmark evaluation described in Section IX-B.

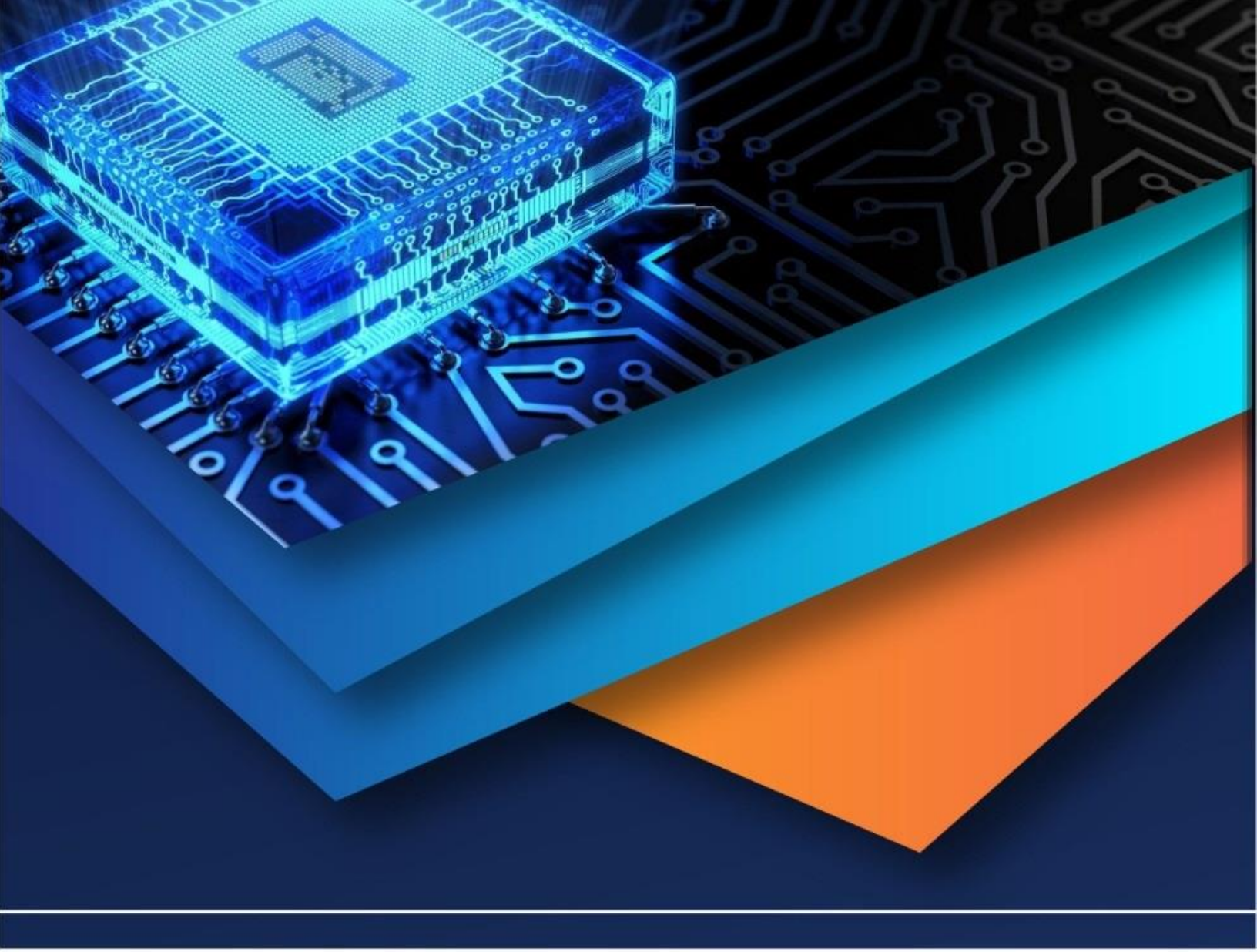
A rigorous publication-grade benchmark protocol has been specified to establish external performance evidence through controlled experiments against three baseline systems, with primary metrics of RSR, MTTR, and Adaptation Gain reported at the 95% confidence level. If validated under real production workloads, a framework of this kind could meaningfully reduce the operational burden on engineering teams managing complex distributed systems. Future work will focus on three directions: (1) live production deployment to build the external benchmark dataset; (2) integration of LLM inference into the Decision Engine's plan evaluation step to augment structured metric signals; and (3) extension to federated multi-cluster environments where recovery coordination spans heterogeneous infrastructure stacks.

XII. ACKNOWLEDGMENT

The authors express sincere gratitude to Dr. Pendela Srinivasarao, Professor, Department of Computer Science and Engineering, Vasireddy Venkatadri Institute of Technology, Nambur, Guntur, Andhra Pradesh, for his invaluable guidance, continuous encouragement, and expert mentorship throughout the course of this project. His insights significantly shaped the research direction and quality of this work.

REFERENCES

[1] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," Computer, vol. 36, no. 1, pp. 41-50, Jan. 2003.
[2] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, 2nd ed. Cambridge, MA: MIT Press, 2018.
[3] J. Wei, X. Wang, D. Schuurmans et al., "Chain of thought prompting elicits reasoning in large language models," arXiv:2201.11903, 2022.
[4] S. Yao, D. Yu, J. Zhao et al., "Tree of thoughts: Deliberate problem solving with large language models," in Proc. NeurIPS, vol. 36, 2023.
[5] A. Madaan, N. Tandon, P. Gupta et al., "Self-Refine: Iterative refinement with self-feedback," arXiv:2303.17651, NeurIPS 2023.
[6] N. Shinn, F. Cassano, E. Berman et al., "Reflexion: Language agents with verbal reinforcement learning," in Proc. NeurIPS, 2023.
[7] Z. Gou, Z. Shao, Y. Gong et al., "CRITIC: Large language models can self-correct with tool-interactive critiquing," in Proc. ICLR, 2024.
[8] X. Chen, M. Lin, N. Schaerfer et al., "Self-Debug: Teaching large language models to debug their predicted program," in Proc. ICLR, 2024.
[9] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do Internet services fail, and what can be done about it?" in Proc. USENIX USITS, 2003, pp. 1-16.
[10] X. Wang, J. Wei, D. Schuurmans et al., "Self-consistency improves chain of thought reasoning in language models," in Proc. ICLR, 2023.
[11] S. Welleck, X. Lu, P. West et al., "Generating sequences by learning to self-correct," in Proc. ICLR, 2023.
[12] E. Zelikman, Y. Wu, J. Mu, and N. Goodman, "Self-Taught Optimizer (STOP): Recursively self-improving code generation," in Proc. ICLR, 2024.
[13] Y. Bai, S. Jones, K. Ndousse et al., "Constitutional AI: Harmlessness from AI feedback," Anthropic Technical Report, arXiv:2212.08073, 2022.
[14] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," ACM Queue, vol. 14, no. 1, pp. 70-93, 2016.
[15] IBM Research, "An architectural blueprint for autonomic computing," IBM White Paper, 4th ed., 2006.
[16] Z. Gou, Z. Shao, Y. Gong et al., "Teaching language models to self-debug," in Proc. EMNLP, 2023.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY