

# Self-Healing AI-Enhanced Web Applications: A Runtime Observability and Recovery Framework

Venkata Krishna Teja Kolli

*Independent Researcher, United States*

*Corresponding Author: Venkata Krishna Teja Kolli*

---

## ABSTRACT

*Intelligent web applications increasingly depend on artificial intelligence for personalization, retrieval, natural-language interaction and task automation. These capabilities improve user experience but also introduce new runtime failures, including model uncertainty, stale context, broken user interface states, failed tool calls and inconsistent data flows. This paper proposes a Self-Healing AI-Enhanced Web Application framework that combines runtime telemetry, failure classification, policy-constrained repair planning and human review. The framework treats recovery as a controlled application workflow rather than an unrestricted model action. It identifies failure classes, ranks low-risk repair strategies such as retry, rollback, revalidation and alternate path selection, and escalates high-risk changes for human approval. The contribution is a reference architecture, an operational workflow and an evaluation protocol for measuring recovery quality, user impact, safety and auditability. The paper argues that future intelligent web applications should be judged not only by initial task success but also by their ability to detect, explain and recover from failures without losing user trust.*

---

Date of Submission: 09-10-2026

Date of acceptance: 20-10-2026

---

## I. INTRODUCTION

Web applications are expected to remain available, responsive and correct even as user workflows, data sources and external services change. Intelligent web applications add another layer of complexity because they use models, retrieval systems and automation tools to interpret user intent and produce adaptive behavior. A conventional failure may be a missing API response or a JavaScript exception. An intelligent failure may be a stale retrieved passage, an invalid tool action, a hallucinated instruction, a broken browser-agent path or a confidence mismatch between generated output and application state.

Existing reliability practices emphasize observability, incident response and service-level objectives [1], [2]. These practices remain essential, but they do not fully describe how an application should recover when an AI-enabled feature fails during a user workflow. Simply retrying a model call may reproduce the same bad answer. Automatically changing data or submitting a form may create external consequences. Therefore, self-healing in intelligent web applications must be policy-constrained, observable and reversible.

This paper asks: how can intelligent web applications detect and recover from runtime failures without giving unrestricted authority to an AI agent? The proposed answer is a Self-Healing AI-Enhanced Web Application framework. The framework combines telemetry, failure classification, repair planning, validation and human review in a closed recovery loop.

The paper contributes a reference architecture, a failure taxonomy, an operational workflow and an evaluation protocol. It is a framework paper rather than a report of deployed experiments. This framing avoids unsupported performance claims while giving researchers and practitioners a concrete structure for future implementation and validation.

## II. RELATED WORK

Site reliability engineering provides a foundation for monitoring production services, defining reliability objectives and responding to incidents [1], [2]. Observability standards such as OpenTelemetry and Trace Context support distributed tracing, metrics and logs across modern systems [3], [4]. These practices are necessary for intelligent web applications because recovery decisions require evidence about user actions, network calls, model calls, browser state and service dependencies.

Automatic debugging and software repair research studies how failures can be localized and corrected [5], [6]. Software analytics also shows that operational data can guide engineering decisions [7]. However, web application recovery differs from source-code repair because many failures occur during live sessions and cannot be solved by patching code alone. Runtime recovery may require choosing a different interaction path, revalidating data, asking the user for clarification or rolling back a partially completed workflow.

Web-agent research has shown that autonomous agents can perform tasks in realistic websites, but also that real web environments remain brittle and difficult [8], [9]. Generative AI security guidance identifies prompt injection, excessive agency, insecure output handling and overreliance as major risks [10], [11]. These risks matter for self-healing because a recovery agent that receives too much authority may repair the wrong thing or create a larger incident.

AI risk frameworks emphasize governance, measurement, documentation and human oversight [12], [13]. Retrieval-augmented generation work also shows the value of evidence-grounded output [14], [15]. The proposed framework applies these ideas to runtime recovery by requiring evidence, validation and approval before high-impact repair actions.

### III. PROPOSED FRAMEWORK

The proposed framework is organized as a recovery loop. The loop begins with telemetry collection from the user workflow and application services. A failure classifier labels the problem. A repair planner proposes one or more recovery actions. A recovery policy ranks proposals by safety and expected usefulness. Finally, a gate validates the proposal, executes low-risk repairs or escalates high-risk repairs for human review.

#### A. Architectural Components

The user workflow layer represents the user's task, current page state, entered data and expected outcome. The telemetry collector captures logs, traces, user-interface events, model prompts, retrieval source identifiers, tool calls and validation failures. The failure classifier maps evidence to classes such as interface failure, data failure, model failure, retrieval failure, network failure and policy failure.

The repair planner proposes strategies such as retrying with bounded backoff, rolling back a partial action, refreshing retrieved evidence, selecting an alternate UI path, asking the user for clarification or creating a developer incident. The recovery policy prevents unsafe repair. For example, repeating a read-only request may be low risk, while submitting a corrected payment form should require approval. The recovery gate validates repair results and records an audit event.

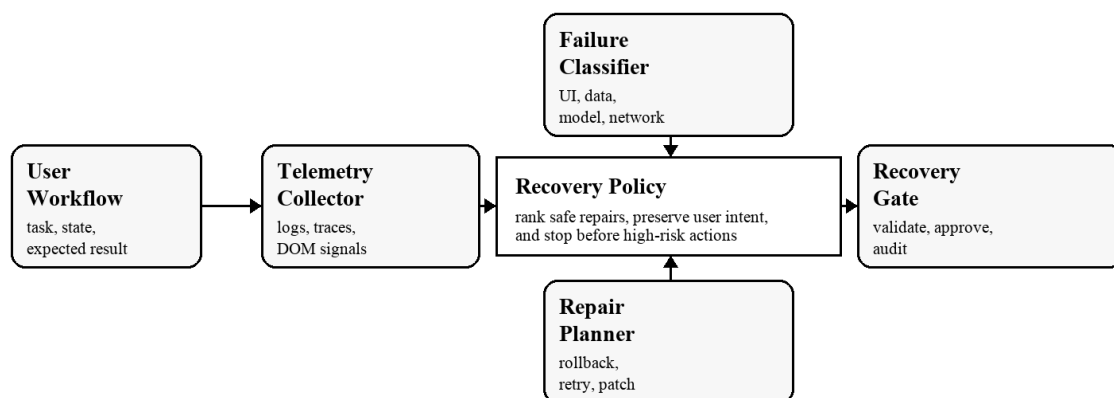


Figure 1: Self-healing runtime loop for AI-enhanced web applications

#### B. Failure Taxonomy

A useful self-healing system needs a shared taxonomy. Interface failures include broken selectors, hidden controls and invalid focus state. Data failures include stale records, schema mismatches and missing required fields. Model failures include low confidence, unsupported claims and inconsistent outputs. Retrieval failures include irrelevant, outdated or untrusted evidence. Tool failures include denied permissions, rate limits and partial execution. Policy failures occur when a requested repair violates safety, privacy or approval rules.

Table 1: Failure classes and recovery strategies

| Failure Class     | Common Signal                                                | Preferred Recovery                                                  |
|-------------------|--------------------------------------------------------------|---------------------------------------------------------------------|
| Interface failure | Element not found, invalid focus order or broken navigation. | Re-observe page, choose alternate path or escalate UI defect.       |
| Data failure      | Schema mismatch, stale field or missing required value.      | Revalidate source, request missing input or rollback partial write. |
| Model failure     | Unsupported claim, low confidence or inconsistent output.    | Regenerate with constrained evidence or ask clarification.          |
| Retrieval failure | Irrelevant, outdated or untrusted source content.            | Refresh index, rerank sources or restrict to approved evidence.     |
| Tool failure      | Permission denied, timeout or partial external action.       | Retry safely, use fallback tool or create incident record.          |
| Policy failure    | Repair violates approval, privacy or agency rule.            | Block automatic repair and route to human review.                   |

#### IV. OPERATIONAL WORKFLOW

The workflow begins when a task fails or when the system detects a high-risk anomaly. Detection can come from client-side events, server logs, distributed traces, model validation rules, retrieval checks or user feedback. The telemetry collector creates a compact failure record that includes the task goal, current state, recent actions, relevant service calls and any generated output that influenced the workflow.

The failure classifier assigns one or more failure classes. The repair planner then proposes candidate repairs. Candidate repairs are deliberately small. Examples include refreshing evidence, re-running a read-only API call, preserving entered form data before page reload, asking the user to choose between ambiguous entities or restoring a previous known-good state. The recovery policy ranks repairs according to user impact, reversibility and required authority.

The recovery gate executes only repairs that pass validation. Read-only and reversible actions may be automatic. State-changing repairs require explicit user confirmation or operator approval. Every repair attempt is logged with its evidence, selected strategy, validation result and final outcome. This record supports debugging, auditability and future improvement of recovery policies.

#### V. EVALUATION PROTOCOL

Self-healing systems should be evaluated by more than whether a workflow eventually succeeds. A recovery action can complete a task while confusing the user, hiding an error or creating an unsafe side effect. The proposed evaluation protocol measures detection quality, repair success, recovery time, user impact, safety and auditability.

A test suite should inject controlled failures into representative web workflows. Examples include missing DOM elements, stale retrieval evidence, invalid API responses, model output that violates schema, network timeouts and tool permission failures. Each injected failure should have an expected safe response. The self-healing framework can then be compared against a baseline that only retries requests and a baseline that escalates every failure to a human.

Table 2: Evaluation dimensions for self-healing AI-enhanced web applications

| Dimension         | Metric                                                    | Evaluation Question                                    |
|-------------------|-----------------------------------------------------------|--------------------------------------------------------|
| Detection quality | Precision and recall of failure classification.           | Did the system identify the correct failure class?     |
| Repair success    | Percentage of tasks restored without unsafe side effects. | Did recovery complete or safely preserve the workflow? |
| Recovery time     | Seconds from anomaly detection to stable state.           | Was recovery fast enough for interactive use?          |
| User impact       | Lost input, extra steps and perceived confusion.          | Did recovery respect the user's work and intent?       |
| Safety            | Blocked high-risk repairs and approval compliance.        | Did the system avoid unauthorized state changes?       |
| Auditability      | Completeness of evidence, repair and validation logs.     | Can a reviewer reconstruct the recovery path?          |

#### VI. ILLUSTRATIVE APPLICATION SCENARIOS

The first scenario is an AI support dashboard. A support agent asks the application to summarize a ticket and suggest a reply. The retrieval layer returns outdated documentation. The framework detects source staleness, refreshes approved evidence and regenerates the draft with citations. Because no external message is sent, the repair remains low risk.

The second scenario is a browser-agent checkout assistant. A payment form changes its field labels after a deployment. The agent cannot find the expected field. The framework classifies an interface failure, re-observes the DOM and proposes an alternate path. Because payment submission is consequential, the recovery gate requires user confirmation before any final action.

The third scenario is an enterprise workflow with partial execution. A tool successfully creates a draft record but fails while attaching a document. The framework identifies partial completion, preserves the draft identifier and offers a rollback or retry. The audit log records the tool call, partial state and selected recovery.

The fourth scenario is a learning platform. A generated explanation does not match the source lesson. The framework detects unsupported claims through evidence validation, blocks the response and asks the model to regenerate using approved passages only. The user sees a corrected answer rather than a silent failure.

## VII. DISCUSSION

The proposed framework treats self-healing as an application governance problem. A model may help classify failures and propose repair actions, but it should not receive unrestricted authority to modify application state. The recovery policy and gate are therefore central. They make recovery deliberate, reversible and reviewable.

There are trade-offs. Detailed telemetry can raise privacy concerns if raw user content is stored unnecessarily. Repair planning can increase latency. Conservative approval gates may reduce automation benefits. However, these trade-offs are preferable to opaque autonomous recovery in consequential workflows. The framework encourages teams to define which repairs are automatic, which are confirmable and which are prohibited.

The framework can also support continuous improvement. Aggregated failure classes can reveal weak areas in interface design, retrieval quality or model validation. Repair logs can identify recurring defects and guide permanent fixes. In this sense, self-healing should not hide systemic problems; it should reduce user harm while making systemic problems easier to diagnose.

The limitation of this paper is that it proposes a framework rather than reporting a deployment. Future work should implement the framework in a real intelligent web application and evaluate it under injected failures. A useful study would compare recovery success, user trust and operational cost across different repair policies.

## VIII. CONCLUSION

Intelligent web applications require reliability mechanisms that understand both conventional web failures and AI-specific failure modes. This paper proposed a Self-Healing AI-Enhanced Web Application framework that combines telemetry, failure classification, repair planning, recovery policy, validation and audit logging. The framework allows low-risk recovery while preserving human approval for consequential repairs.

The central conclusion is that self-healing should be safe before it is autonomous. Future web applications should detect failure, explain recovery, preserve user intent and avoid unauthorized state changes. By evaluating detection quality, repair success, recovery time, user impact, safety and auditability, developers can build intelligent web applications that are resilient without becoming unpredictable.

## REFERENCES

- [1] B. Beyer, C. Jones, J. Petoff and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*, O'Reilly Media, 2016.
- [2] B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara and S. Thorne, *The Site Reliability Workbook*, O'Reilly Media, 2018.
- [3] OpenTelemetry Authors, *OpenTelemetry Specification*, Cloud Native Computing Foundation, 2026.
- [4] World Wide Web Consortium, *Trace Context Level 2*, W3C Recommendation, 2024.
- [5] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*, Morgan Kaufmann, 2009.
- [6] M. Monperrus, *Automatic Software Repair: A Bibliography*, ACM Computing Surveys, 2018.
- [7] T. Menzies and T. Zimmermann, *Software Analytics: SoWhat?*, IEEE Software, 2013.
- [8] X. Deng et al., *Mind2Web: Towards a Generalist Agent for the Web*, arXiv:2306.06070, 2023.
- [9] S. Zhou et al., *WebArena: A Realistic Web Environment for Building Autonomous Agents*, arXiv:2307.13854, 2023.
- [10] K. Greshake et al., *Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection*, arXiv:2302.12173, 2023.
- [11] OWASP Foundation, *OWASP Top 10 for Large Language Model Applications*, OWASP GenAI Security Project, 2025.
- [12] National Institute of Standards and Technology, *Artificial Intelligence Risk Management Framework (AI RMF 1.0)*, NIST AI 100-1, 2023.
- [13] National Institute of Standards and Technology, *Artificial Intelligence Risk Management Framework: Generative Artificial Intelligence Profile*, NIST AI 600-1, 2024.
- [14] P. Lewis et al., *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*, *Advances in Neural Information Processing Systems*, 2020.
- [15] Y. Gao et al., *Retrieval-Augmented Generation for Large Language Models: A Survey*, arXiv:2312.10997, 2023.