International Journal of Engineering & Scientific Research
Vol.14 Issue 01, January 2026
ISSN: 2347-6532 Impact Factor: 8.076
Journal Homepage: http://www.ijmra.us, Email: editorijmie@gmail.com
Double-Blind Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gage as well as in Cabell's Directories of Publishing Opportunities, U.S.A

Enhancing End-to-End Test Stability Through AI-Assisted Self-Healing: A Case Study of Playwright Healer Agent Implementation

***Abstract*—End-to-end (E2E) test flakiness remains a persistent challenge in continuous integration/continuous deployment (CI/CD) pipelines, often resulting from brittle selectors, timing dependencies, external service integrations, and dynamic application states. This paper presents a comprehensive case study of implementing Playwright's Healer Agent—an AI-powered autonomous agent that provides runtime resilience and self-repair capabilities for automated UI tests. Over a six-month deployment period, we instrumented our E2E test suite with healing capabilities, including locator healing, action retries with exponential backoff, redirect-aware flows, and stubbing external services. Our results demonstrate a 73% reduction in false-positive test failures, a 45% decrease in mean time to repair (MTTR) for legitimate failures, and a 62% improvement in overall test suite reliability. We present our implementation methodology, provide concrete code patterns, discuss governance considerations for automated test healing, and analyze the implications for test engineering practices. Our findings suggest that AI-assisted self-healing represents a pragmatic approach to managing test fragility while maintaining the diagnostic value of test failures when properly instrumented with transparency and auditability.**

***Index Terms*—end-to-end testing, test automation, flaky tests, self-healing tests, AI agents, Playwright, continuous integration, software quality assurance**

I. INTRODUCTION

A. The Challenge of E2E Test Stability

End-to-end testing serves as a critical quality gate in modern software development, validating complete user workflows across multiple system components [1]. However, E2E tests exhibit significantly higher flakiness rates compared to unit or integration tests, with industry studies reporting that 15-73% of test failures in E2E suites are non-deterministic [2, 3]. This flakiness erodes developer trust, increases CI/CD cycle times, and consumes substantial engineering resources for triage and maintenance [4].

Familiar sources of E2E test instability include: (1) timing-dependent race conditions in asynchronous UI rendering, (2) fragile DOM selectors that break with minor UI changes, (3) integration points with external services exhibiting variable latency or availability, (4) dynamic application routing based on feature flags or user state, and (5) infrastructure instability in staging environments [5, 6].

B. Research Motivation

Traditional approaches to managing test flakiness involve increasing wait times, implementing explicit synchronization, maintaining parallel selector strategies, and manual triage workflows [7]. These methods are labor-intensive, increase test execution time, and often lag behind application evolution. The emergence of Large Language Models (LLMs) and AI agents presents an opportunity to automate test resilience and maintenance through intelligent runtime adaptation [8].

This research investigates whether AI-assisted self-healing capabilities can meaningfully improve E2E test stability without compromising the diagnostic value of test failures or masking genuine regressions. We focus specifically on Playwright's Healer Agent framework, introduced in version 1.56, which leverages LLMs to provide autonomous test repair capabilities.

C. Research Questions

This study addresses the following research questions:

RQ1: To what extent can an AI-powered healer agent reduce false-positive test failures in a production E2E test suite?

RQ2: What categories of test failures are most amenable to automated healing, and what patterns emerge in successful versus unsuccessful healing attempts?

RQ3: What governance and transparency mechanisms are necessary to prevent healer agents from masking legitimate application regressions?

RQ4: What is the impact of healer agent implementation on test maintenance effort and developer workflow?

II. RELATED WORK

A. Test Flakiness Characterization

Luo et al. [2] conducted a comprehensive empirical study of flaky tests across 201 open-source projects, identifying async waits, concurrency issues, and test-order dependencies as the primary root causes. Eck et al. [9] demonstrated that UI-based E2E tests exhibit 4.2× higher flakiness rates than API-based tests, primarily due to DOM instability and rendering timing variations. Our work builds upon these characterizations by targeting the specific failure modes identified in prior research.

B. Automated Test Repair

Several approaches to automated test repair have been proposed. WATER [10] uses DOM tree differencing to repair broken web test locators automatically. Coppola et al. [11] demonstrated that machine learning classifiers can predict brittle test locators before breakage occurs. Leotta et al. [12] developed a technique for automatically healing Selenium tests through multiple locator strategies and visual element matching. These approaches typically focus on single failure modes (primarily locator changes), whereas our healer agent implementation addresses multiple failure categories simultaneously through LLM-powered decision making.

C. AI in Software Testing

Recent work has explored LLM applications across the testing lifecycle. Liu et al. [13] demonstrated that GPT -4 can generate functional test cases from requirements with 67% human-assessed adequacy. Schafer et al. [14] showed that LLMs can adaptively repair test assertions when application behavior changes intentionally. The Playwright Agents framework [15] represents one of the first integrated agent systems spanning test planning, generation, and healing within a single testing framework, which motivated our investigation.

D. Research Gap

While prior work addresses isolated aspects of test stability (locator repair, timing adjustment, or intelligent generation), there is limited empirical evidence on holistic self-healing systems deployed in production E2E test suites. Furthermore, the governance and transparency mechanisms necessary for the production deployment of autonomous test healing agents remain underexplored in the literature.

III. Background: Playwright Agent Architecture

A. Playwright Agents Overview

Playwright Agents, introduced in Playwright v1.56 [15], comprise three specialized AI agents that automate different phases of the test lifecycle:

Planner Agent: Analyzes application requirements and generates structured test plans in Markdown format, outlining test scenarios and user flows.

Generator Agent: Consumes test plans and generates executable Playwright test files with appropriate locators, assertions, and test structure.

Healer Agent: Monitors test execution, detects failures, and attempts automated remediation through selector adaptation, timing adjustment, and flow modification.

All three agents integrate with LLM providers (OpenAI GPT-4, Anthropic Claude, or local models) through a standardized interface. Agents are initialized via

npx playwright init-agents --loop [vscode | claude | opencode]

, which scaffolds agent definitions and seed files for project-specific configuration.

B. Healer Agent Capabilities

The Healer Agent provides six core capabilities:

1) **Locator Healing:** When a selector fails to match elements, the agent queries the current DOM structure, identifies candidate elements based on semantic similarity to the original intent, and attempts alternative selection strategies (text content, ARIA labels, position-based, visual proximity).
2) **Action Retries:** Transient failures in click, input, or navigation actions trigger exponential backoff retry sequences with progressive timeout adjustments.
3) **Frame and Network Robustness:** Specialized handling for iframe attachment timing and network request/response synchronization, particularly critical for third-party widget integrations.
4) **Redirect-Aware Flows:** Detection of unexpected navigation events (e.g., intermediate authentication, missing profile data, approval workflows) with automated completion or API-based state advancement.
5) **External Integration Stubbing:** Runtime interception of external service calls (payment processors, identity verification, document signing) with configurable stub responses.
6) **Telemetry and Reporting:** Comprehensive logging of healing attempts, success/failure rates, screenshots, and trace artifacts for post-execution review.

IV. Methodology

A. Study Context

This case study was conducted within a financial services technology company's E2E test suite covering a web-based application. The application under test (AUT) features:

- **Technology Stack:** React SPA frontend, Node.js microservices backend, PostgreSQL database
- **External Integrations:** Bank verification service (Plaid), document signing service (DocuSign), identity verification services
- **Deployment:** Kubernetes-based staging environment mirroring production topology
- **Feature Management:** Feature flag system controlling workflow variations

The existing E2E test suite consisted of 287 tests covering critical user journeys from initial application through completion. Tests were implemented in Playwright (TypeScript) and executed in GitHub Actions CI on every pull request and main branch commit.

B. Baseline Metrics Collection

We collected baseline metrics over a four-week pre-implementation period (February 2025):

- **Test Runs:** 1,247 complete suite executions

- **Failure Rate:** 31.4% of runs contained at least one test failure
- **False Positive Rate:** 73.2% of failures were classified as non-deterministic after manual triage (timing issues, external service flakiness, infrastructure problems)
- **MTTR:** Mean time to resolve test failures: 4.2 hours (including triage, investigation, and fix implementation)
- **Triage Effort:** Average 18 minutes per failed test run for initial classification

Failure categories identified during baseline analysis:

Category	Frequency	Description
Timing/Race Conditions	34.2%	Async element rendering, network request timing
External Service Failures	26.8%	Third-party widget timeouts, service availability
Unexpected Redirects	18.5%	Feature-flag-gated flows, state-dependent routing
Selector Fragility	12.3%	DOM structure changes, dynamic class names
Infrastructure Instability	8.2%	Pod restarts, database connection issues

C. Healer Agent Implementation

Implementation proceeded in three phases over 12 weeks (March-May 2025):

Phase 1: Foundation (Weeks 1-4)

- Healer agent initialization and LLM provider configuration (OpenAI GPT-4)
- Implementation of base helper functions for resilient actions
- Telemetry infrastructure for logging healing attempts
- Opt-in flag system for gradual rollout

Phase 2: Core Healing Patterns (Weeks 5-8)

- Redirect-aware URL waiting with automatic flow completion
- Robust iframe handling for third-party integrations
- Selector fallback chains for frequently-changed components
- Retry logic with exponential backoff for transient failures

Phase 3: Advanced Capabilities (Weeks 9-12)

- Admin API integration for state advancement (approval bypass, verification completion)
- Feature flag detection and adaptive test flows
- External service stubbing with intelligent fallback
- Healing attempt dashboard and review workflow

D. Implementation Patterns

We developed four primary healer patterns addressing our most common failure modes:

1) Resilient Click with Selector Fallbacks

```
/**
 * Attempts to click an element using multiple selector strategies
 * with retry logic and comprehensive failure telemetry.
 */
async function healedClick(
  page: Page,
  selectors: SelectorStrategy[],
  opts: HealedClickOptions = {}
): Promise<span><span style="color: rgb(150, 34, 73); font-weight: bold;">&lt;healresult&gt;</span><span style="color: black; font-weight: normal;"> {
  const maxAttempts = opts.attempts ?? 3;
  const telemetry: HealAttempt[] = [];

  for (let attempt = 0; attempt &lt; maxAttempts; attempt++) {
    for (const strategy of selectors) {
      const startTime = Date.now();
      try {
        const locator = page.locator(strategy.selector);
        await locator.waitFor({
          state: 'visible',
          timeout: strategy.timeout ?? 5000
        });
        await locator.click({ timeout: 10000 });

        // Log successful heal
        const healLog: HealAttempt = {
          type: 'locator_fallback',
          timestamp: new Date().toISOString(),
          attempt: attempt + 1,
          strategy: strategy.name,
          selector: strategy.selector,
          duration: Date.now() - startTime,
          success: true
        };
        telemetry.push(healLog);
        await logHealAttempt(healLog);

        return {
          success: true,
          strategy: strategy.name,
          attempts: attempt + 1,
          telemetry
        };
      } catch (error) {
        telemetry.push({
          type: 'locator_fallback',
          timestamp: new Date().toISOString(),
          attempt: attempt + 1,
          strategy: strategy.name,
          selector: strategy.selector,
          duration: Date.now() - startTime,
          success: false,
          error: error.message
        });
```

```
      }
    }

    // Exponential backoff between attempts
    if (attempt &lt; maxAttempts - 1) {
      await page.waitForTimeout(500 * Math.pow(2,
attempt));
    }
  }

  // All strategies failed - capture diagnostic information
  const screenshot = await page.screenshot({
    fullPage: true,
    path: `heal-failure-\${Date.now()}.png`
  });
  await page.context().tracing.stop({
    path: `heal-failure-\${Date.now()}.zip`
  });

  const failureLog: HealAttempt = {
    type: 'locator_fallback',
    timestamp: new Date().toISOString(),
    success: false,
    totalAttempts: maxAttempts * selectors.length,
    strategies: selectors.map(s =&gt; s.name),
    screenshotPath: screenshot,
    telemetry
  };
  await logHealAttempt(failureLog);

  throw new Error(
    `healedClick failed after \${maxAttempts} attempts with ` +
    `\${selectors.length} strategies: \${selectors.map(s =&gt;
s.name).join(', ')}`
  );
}
```

2) Robust Iframe Handling for External Widgets

```
/**
 * Polls for iframe attachment and content availability with
 * resilient contentFrame() handling for race conditions.
 */
async function waitForExternalFrame(
  page: Page,
  frameIdentifier: string,
  targetSelector: string,
  options: FrameWaitOptions = {}
): Promise {
  const timeout = options.timeout ?? 30000;
  const pollInterval = options.pollInterval ?? 500;
  const startTime = Date.now();

  const healLog: HealAttempt = {
    type: 'iframe_polling',
    timestamp: new Date().toISOString(),
    frameIdentifier,
    targetSelector
  };

  while (Date.now() - startTime &lt; timeout) {
    try {
      // Attempt to locate iframe handle
      const frameHandle = await page.\$(frameIdentifier);
      if (!frameHandle) {
        await page.waitForTimeout(pollInterval);
        continue;
      }

      // Attempt to access contentFrame (may be null during
attachment)
      const frame = await frameHandle.contentFrame();
      if (!frame) {
        await page.waitForTimeout(pollInterval);
        continue;
      }

      // Verify target content is available
      try {
        await frame.waitForSelector(targetSelector, { timeout:
3000 });
        healLog.success = true;
        healLog.duration = Date.now() - startTime;
        await logHealAttempt(healLog);
        return frame;
      } catch {
        // Target not yet available, continue polling
        await page.waitForTimeout(pollInterval);
      }
    } catch (error) {
      // Handle errors during frame access
      await page.waitForTimeout(pollInterval);
    }
  }

  // Timeout reached - attempt fallback to stubbing
  if (options.fallbackToStub) {
    healLog.success = true;
    healLog.fallbackUsed = 'stub';
    await logHealAttempt(healLog);
    await stubExternalWidget(page, frameIdentifier);
    return null; // Signal that stub was used
  }

  healLog.success = false;
  healLog.duration = Date.now() - startTime;
  await logHealAttempt(healLog);

  throw new Error(
    `waitForExternalFrame: Frame \${frameIdentifier} or
selector ` +
```

```
    `\${targetSelector} not available within \${timeout}ms`
  );
}
```

3) Redirect-Aware URL Waiting with Automatic Healing

```
/**
 * Waits for expected URL pattern while detecting and
healing
 * unexpected intermediate redirects (missing profile data,
 * approval requirements, verification steps).
 */
async function waitForUrlOrHeal(
  page: Page,
  expectedPattern: RegExp,
  healHandlers: Record,
  options: UrlWaitOptions = {}
): Promise {
  const timeout = options.timeout ?? 120000;
  const pollInterval = options.pollInterval ?? 500;
  const startTime = Date.now();

  const healLog: HealAttempt = {
    type: 'redirect_heal',
    timestamp: new Date().toISOString(),
    expectedPattern: expectedPattern.toString()
  };

  while (Date.now() - startTime &lt; timeout) {
    const currentUrl = page.url();

    // Check if we reached expected destination
    if (expectedPattern.test(currentUrl)) {
      healLog.success = true;
      healLog.duration = Date.now() - startTime;
      healLog.healingRequired = false;
      await logHealAttempt(healLog);
      return { success: true, healed: false };
    }

    // Check for known redirect patterns requiring healing
    for (const [urlPattern, handler] of
Object.entries(healHandlers)) {
      if (currentUrl.includes(urlPattern)) {
        healLog.detectedRedirect = urlPattern;
        healLog.healingRequired = true;

        try {
          // Attempt automated healing
          await handler.heal(page);

          // Wait for navigation after heal
          await page.waitForLoadState('load', { timeout: 10000
});

          // Verify we progressed past the redirect
          if (!page.url().includes(urlPattern)) {
            healLog.success = true;
            healLog.healStrategy = handler.name;
            healLog.duration = Date.now() - startTime;
            await logHealAttempt(healLog);

            // Continue waiting for final expected URL
            break;
          }
        } catch (error) {
          healLog.success = false;
          healLog.error = error.message;
          await logHealAttempt(healLog);
          throw new Error(
            `Healing failed for redirect \${urlPattern}:
\${error.message}`
          );
        }
      }
    }

    await page.waitForTimeout(pollInterval);
  }

  // Timeout - expected URL never reached
  healLog.success = false;
  healLog.duration = Date.now() - startTime;
  healLog.finalUrl = page.url();
  await logHealAttempt(healLog);

  throw new Error(
    `waitForUrlOrHeal: Timeout waiting for
\${expectedPattern}. ` +
    `Final URL: \${page.url()}`
  );
}
```

4) Feature-Flag-Aware Healing

```
/**
 * Queries feature flag state and adapts test flow or uses
 * admin API bypass for flag-gated requirements.
 */
async function handleFeatureFlagGatedFlow(
  page: Page,
  flagKey: string,
  flowHandler: FlagFlowHandler
): Promise {
  const healLog: HealAttempt = {
    type: 'feature_flag_adaptation',
    timestamp: new Date().toISOString(),
    flagKey
  };

  try {
```

International Journal of Engineering & Scientific Research
Vol.14 Issue 01, January 2026
ISSN: 2347-6532 Impact Factor: 8.076
Journal Homepage: http://www.ijmra.us, Email: editorijmie@gmail.com
Double-Blind Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gage as well as in Cabell's Directories of Publishing Opportunities, U.S.A

```
    // Query feature flag state from feature management API
    const flagValue = await queryFeatureFlag(flagKey,
TEST_USER_CONTEXT);
    healLog.flagValue = flagValue;

    if (flagValue) {
      // Flag enabled - determine if we should complete flow or
bypass
      if (flowHandler.strategy === 'complete') {
        await flowHandler.completeFlow(page);
        healLog.healStrategy = 'flow_completion';
      } else if (flowHandler.strategy === 'admin_bypass') {
        await flowHandler.adminBypass();
        healLog.healStrategy = 'admin_api_bypass';
      }

      healLog.success = true;
      await logHealAttempt(healLog);
      return { success: true, healed: true };
    } else {
      // Flag disabled - no action needed
      healLog.success = true;
      healLog.healingRequired = false;
      await logHealAttempt(healLog);
      return { success: true, healed: false };
    }
  } catch (error) {
    healLog.success = false;
    healLog.error = error.message;
    await logHealAttempt(healLog);
    throw error;
  }
}
```

E. Telemetry and Governance Implementation

To ensure transparency and prevent masking of legitimate regressions, we implemented a comprehensive telemetry system:

```
interface HealAttempt {
  type: 'locator_fallback' | 'iframe_polling' | 'redirect_heal' |
        'feature_flag_adaptation' | 'retry_with_backoff';
  timestamp: string;
  testName?: string;
  success: boolean;
  duration?: number;
  strategy?: string;
  selector?: string;
  error?: string;
  screenshotPath?: string;
  tracePath?: string;
  healingRequired?: boolean;
  fallbackUsed?: string;
}

async function logHealAttempt(attempt: HealAttempt):
Promise {
  // Log to structured JSON for analysis
  await fs.appendFile(
    'heal-attempts.jsonl',
    JSON.stringify(attempt) + '\n'
  );

  // Report to monitoring system (DataDog, Grafana, etc.)
  await metrics.increment('test.heal.attempt', {
    type: attempt.type,
    success: attempt.success.toString()
  });

  // For successful heals, mark test result as "healed"
  if (attempt.success &amp;amp;&amp;amp; attempt.healingRequired)
{
    await test.info().annotations.push({
      type: 'healed',
      description: `Test healed via \${attempt.type}:
\${attempt.strategy}`
    });
  }
}
```

We established governance policies:

1. **Heal Visibility:** All tests that benefited from healing are marked with a

   @healed

   annotation in CI results
2. **Heal Budget:** CI builds fail if >10% of tests require healing, indicating systemic issues
3. **Admin API Audit Trail:** All state-mutating admin API calls are logged with test context
4. **Weekly Review:** The Engineering team reviews the heal attempt dashboard to identify trends and prioritize root cause fixes
5. **Opt-Out Mechanism:** Critical tests can be marked

   @no-heal

   to ensure they fail on any deviation

F. Evaluation Metrics

We measured the following metrics during a 12-week post-implementation period (June-August 2025):

Primary Metrics:

- False positive failure rate (percentage of non-deterministic failures)
- Test suite reliability (percentage of runs with zero failures)
- Mean time to repair (MTTR) for test failures
- Triage effort (time spent classifying failures)

Secondary Metrics:

- Healing success rate by category

International Journal of Engineering & Scientific Research
Vol.14 Issue 01, January 2026
ISSN: 2347-6532 Impact Factor: 8.076
Journal Homepage: http://www.ijmra.us, Email: editorijmie@gmail.com
Double-Blind Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gage as well as in Cabell's Directories of Publishing Opportunities, U.S.A

- Regression masking incidents (false negatives)
- Test execution time impact
- Maintenance effort for healer logic

Qualitative Data:

- Developer survey on test trust and debugging experience
- Code review feedback on healing patterns
- Incident reports of masked regressions

V. RESULTS

A. Overall Impact on Test Stability (RQ1)

The implementation of healer agent capabilities produced significant improvements across all primary stability metrics:

Metric	Baseline (Pre)	Post-Implementation	Change
False Positive Rate	73.2%	19.7%	-73.1%
Suite Reliability (Zero Failures)	68.6%	91.3%	+33.1%
MTTR (hours)	4.2	2.3	-45.2%
Triage Effort (min/failure)	18.0	6.5	-63.9%
Test Runs Analyzed	1,247	1,389	-

Statistical Significance: Chi-square tests confirmed that the reduction in false positive rate was statistically significant ($\chi^2 = 247.3$, $p < 0.001$). Mann-Whitney U tests showed significant reductions in MTTR ($U = 89234$, $p < 0.001$) and triage effort ($U = 67891$, $p < 0.001$).

B. Healing Success by Failure Category (RQ2)

Analysis of 1,847 healing attempts over the evaluation period revealed differential success rates across failure categories:

Failure Category	Attempts	Success Rate	MTTR Before	MTTR After	Notes
Timing/Race Conditions	632	91.3%	2.1h	0.3h	Action retries, frame polling
External Service Failures	495	87.5%	6.8h	1.2h	Stubbing fallback effective
Unexpected Redirects	342	94.4%	3.4h	0.5h	Admin API bypass is highly successful
Selector Fragility	227	78.2%	4.1h	1.8h	Multiple fallback strategies needed
Infrastructure Instability	151	42.1%	8.2h	7.1h	Limited healer effectiveness

Key Findings:

1) **Redirect healing proved most effective:** The combination of URL pattern detection and admin API state advancement resolved 94.4% of unexpected redirect scenarios without manual intervention. This addressed our most time-consuming failure category (previously requiring cross-team coordination to understand feature flag interactions).
2) **External service integration showed high variance:** Third-party iframe handling achieved 89% success with robust polling, but document signing integration remained problematic (68% success) due to complex multi-step flows and webhook timing dependencies. Fallback to stubbing improved overall success to 87.5%.
3) **Infrastructure failures require different approaches:** Only 42.1% of infrastructure-related failures could be healed (primarily through retries), confirming that healer agents cannot compensate for systemic environment instability. This finding validated our governance decision to fail fast on infrastructure issues rather than masking them.
4) **Selector healing effectiveness varies with DOM complexity:** Simple selector fallbacks (text content, ARIA labels) succeeded in 92% of cases, but complex dynamic SPAs with generated class names required LLM-powered DOM analysis, reducing success to 64%.

C. Temporal Patterns in Healing Attempts

Analysis of healing frequency over time revealed important patterns:

![Healing attempts over 12 weeks showing initial spike followed by steady decline]

Week 1-2: High healing frequency (avg 34 heals/day) as existing flakiness was addressed. **Week 3-6:** Declining frequency (avg 18 heals/day) as teams fixed root causes identified in healing logs. **Week 7-12:** Stabilized at baseline (avg 8 heals/day), representing new failures from ongoing development

This temporal pattern suggests that healer agents serve a dual purpose: (1) immediate stability improvement, and (2) a diagnostic tool highlighting areas requiring architectural improvement.

D. Governance and Regression Masking (RQ3)

Over the evaluation period, our governance mechanisms detected three instances of potential regression masking:

Incident 1 (Week 3): Healer agent automatically completed identity verification flow for 47 consecutive test runs, masking a regression where validation was failing for edge-case data formats. Detected via "heal budget" alert (>30% of tests healing for the same pattern). Root cause fixed within 4 hours of detection.

Incident 2 (Week 7): Admin API approval bypass masked a UI regression in the status display component. User-facing bug reached production. Detected via customer support ticket.

Post-mortem led to the implementation of visual regression checks independent of healer flow.

Incident 3 (Week 10): Excessive retry logic on a transaction submission endpoint masked a performance regression, causing 8-second latencies. Detected via APM alerts rather than test failures. Led to policy change: retries now fail if individual attempt latency exceeds baseline by >2x.

Governance Effectiveness:

The "heal budget" mechanism (failing CI if >10% of tests healed) successfully prevented 12 instances of systemic issues from being masked. Weekly heal review meetings resulted in 34 root cause fixes and 8 test architecture improvements. However, the three masking incidents revealed limitations:

1) **Silent UI regressions:** Admin API bypasses can mask user-facing UI bugs
2) **Performance degradation:** Retry logic can mask latency regressions
3) **State-dependent bugs:** Automated state advancement may not exercise all code paths

These findings informed our revised governance policy (Section 6.3).

E. Developer Experience Impact (RQ4)

Developer survey responses (n=23, 88% response rate) indicated positive reception with notable concerns:

Positive Impacts:

- 91% reported increased trust in test results
- 87% reported reduced context-switching from CI triage
- 78% reported faster PR merge times due to reduced false positives
- 74% found heal attempt logs helpful for debugging

Concerns:

- 61% expressed concern about potential regression masking
- 48% found heal logs occasionally noisy or difficult to interpret
- 35% reported instances where healer logic itself needed debugging

Qualitative Feedback Examples:

> "The redirect healing has been transformative - I no longer dread feature flag changes breaking every test." - Senior Engineer

> "I appreciate the stability, but I'm uncomfortable with admin API bypasses. We found a real bug that was masked for a week." - QA Engineer

> "The heal logs are great when they work, but sometimes I still need to dig through traces to understand what actually happened." - Staff Engineer

F. Maintenance and Operational Costs

Implementation and maintenance effort:

Activity	Engineer Hours
Initial implementation (Weeks 1-12)	240
Ongoing maintenance (per week)	4-6
Weekly heal review meetings	1-2
Healer logic debugging	2-3

Cost-benefit analysis:

- **Pre-healer:** Avg 8.2 hours/week on test triage and maintenance
- **Post-healer:** Avg 6.8 hours/week (4-6 maintenance + 1-2 review + 2-3 debugging - 8.2 saved)
- **Net savings:** Avg 1.4 hours/week (17%) after initial implementation ROI period

Additional benefits not captured in time savings:

- Reduced developer frustration and context-switching
- Faster feedback loops enabling more frequent deployments
- Improved confidence in test results, leading to better debugging

VI. Discussion

A. Interpretation of Results

Our results provide strong evidence that AI-assisted self-healing can meaningfully improve E2E test stability (RQ1), with a 73% reduction in false positives representing a substantial improvement over the baseline. However, the effectiveness varies dramatically by failure category (RQ2), with redirect handling and timing issues showing >90% success rates while infrastructure failures show limited amenability to healing.

The three regression masking incidents (RQ3) highlight a critical tension: healer agents that are too aggressive in bypassing test steps risk masking legitimate bugs. At the same time, overly conservative healing fails to provide stability benefits. Our findings suggest that governance mechanisms—particularly heal visibility, budget limits, and regular review—are essential to navigate this tension successfully.

The modest net time savings of 17% (RQ4) may seem disappointing given the substantial reduction in false positives. However, this figure understates the value by not accounting for (1) reduced cognitive load from context-switching, (2) improved CI pipeline throughput enabling more frequent deployments, and (3) compounding benefits as test suite size grows.

B. When Healer Agents Are Most Effective

Based on our results, healer agents provide maximum value for:

1. **Integration points with external services:** Third-party widgets (payment processors, identity verification, document signing) introduce timing variability and availability issues that are tedious to manually accommodate but straightforward for automated healing.
2. **Feature-flag-gated flows:** Applications with frequent A/B testing or gradual rollouts benefit

substantially from automated detection and adaptation to flag-dependent routing.

3. **Microservice architectures with async state propagation:** Distributed systems with eventual consistency can benefit from intelligent retry and polling logic.
4. **Rapidly evolving UIs:** Frequent design changes that break selectors are amenable to fallback strategies and LLM-powered element identification.

Conversely, healer agents provide limited value for:

1. **Infrastructure instability:** Environmental issues (pod crashes, network partitions, database deadlocks) require operational solutions, not test-level healing.
2. **Complex business logic bugs:** Healer agents can mask regressions in multi-step workflows if not carefully constrained.
3. **Performance regressions:** Retry logic and extended timeouts can hide latency degradation.

C. Revised Governance Recommendations

Based on lessons learned from masking incidents, we recommend the following governance framework:

1. Stratified Healing Policies:

- **Low-risk heals (no state mutation):** Selector fallbacks, timing adjustments, iframe polling—allow unconditionally.
- **Medium-risk heals (reversible state changes):** Feature flag adaptation with flow completion—allow with logging.
- **High-risk heals (irreversible state changes):** Admin API bypasses that skip UI steps—require explicit opt-in per test.

2. Enhanced Transparency:

- Mark healed tests distinctly in CI with expandable heal detail
- Generate daily heal digest reports for team review
- Implement "heal debt" metric tracking cumulative heals requiring root cause fixes

3. Parallel Validation:

- For admin API bypasses, run lightweight UI assertions even when bypassed to catch rendering bugs
- Implement visual regression testing independent of the functional test path
- Monitor APM metrics during healed tests to detect performance impacts

4. Fail-Fast on Patterns:

- If the same heal pattern occurs in >3 consecutive runs, fail and require manual intervention
- If the heal attempt duration exceeds 2x original timeout, fail rather than succeed
- If a heal required an admin API call that failed, always fail the test

5. Controlled Rollout:

- Begin with read-only healing (log opportunities but do not modify behavior)
- Graduate to low-risk heals, then medium-risk after a monitoring period
- Require a security review for any heals calling admin/internal APIs

D. Architectural Insights

Implementation revealed several architectural patterns that facilitate effective healing:

1. Healer-Friendly Test Design:

Tests structured as explicit state machines with observable stage transitions enable more reliable healing:

```
const testStages = {
  APPLICATION_START: { url: /application\/start/, heals: []
},
  IDENTITY_VERIFICATION: {
    url: /identity-verification/,
    heals: ['enhanced_verification', 'document_upload']
  },
  BANK_CONNECTION: {
    url: /connect-bank/,
    heals: ['bank_iframe', 'manual_entry_fallback']
  },
  // ... etc
};
```

2. Declarative Heal Registration:

Rather than embedding heal logic throughout tests, centralized heal handler registration improves maintainability:

```
registerHealHandlers({
  'approval_required': {
    detect: (page) =&gt; page.url().includes('/approval'),
    heal: async (page) =&gt; await
adminAPI.approveRequest(testContext.requestId),
    risk: 'high',
    requiresOptIn: true
  },
  'verification_prompt': {
    detect: (page) =&gt; page.locator('[data-
testid="verification-code"]').isVisible(),
    heal: async (page) =&gt; await page.fill('[data-
testid="verification-code"]', '123456'),
    risk: 'low',
    requiresOptIn: false
  }
});
```

3. Separation of Concerns:

Healer logic should be cleanly separated from test logic, ideally as Playwright fixtures:

```
const test = base.extend({
  healedPage: async ({ page }, use) =&gt; {
    const healer = new PageHealer(page, healConfig);
```

```
  await healer.initialize();
  await use(healer.page);
  await healer.reportAttempts();
 }
});
```

E. Limitations of AI-Powered Healing

Our experience revealed several fundamental limitations:

1. LLM Reasoning Opacity:
While GPT-4 successfully identified alternative selectors in many cases, failures were difficult to debug. The model occasionally proposed plausible-seeming but incorrect element matches, requiring explicit validation logic.

2. Context Window Constraints:
For complex multi-page flows, providing sufficient context to the LLM for intelligent healing decisions approached token limits. We resorted to hierarchical summarization of earlier test steps.

3. Latency Costs:
LLM API calls during test execution added 200-800ms latency per healing attempt. For tests requiring multiple heals, this became noticeable. Local model deployment (Llama-3) reduced latency but decreased healing success rates.

4. Non-Determinism:
LLM responses exhibited occasional nondeterminism even with temperature=0, leading to flaky healing (healing succeeded on retry but failed initially with the same inputs). This ironically introduced a new source of test flakiness.

5. Cost:
OpenAI API costs for LLM-powered healing averaged $0.08-0.15 per test run (including both successful and failed healing attempts). At our scale (1,400 runs/month), this represented $112-210/month—modest but non-trivial.

F. Comparison with Traditional Approaches

Our healer agent implementation can be compared against traditional stability techniques:

Approach	False Positive Reduction	Maintenance Burden	Risk of Masking
Increased timeouts	20-30%	Low	Low
Multiple selector strategies	35-45%	High (brittle over time)	Low
Manual triage & fix	100% (eventually)	Very high	None
Retry the entire test on failure	40-50%	Low	Medium (flaky passes)
Healer agent (this work)	**73%**	**Medium**	**Medium (with governance)**
Human-in-loop AI assist	85-95%	Medium-high	Very low

The healer agent approach occupies a middle ground: more effective than simple timeouts or retries, less labor-intensive than manual fixes, but requiring careful governance to avoid masking regressions. For teams with sufficient engineering maturity to implement robust telemetry and review processes, healer agents represent a pragmatic compromise.

G. Generalizability

Several factors affect generalizability to other contexts:

Favorable Conditions:

- Applications with external service integrations (high timing variability)
- Microservice architectures (distributed state management complexity)
- High feature flag usage (dynamic routing)
- Frequent UI changes (selector brittleness)
- Mature CI/CD with good observability

Unfavorable Conditions:

- Monolithic applications with deterministic flows
- Highly regulated domains with strict audit requirements (healthcare, finance)
- Small test suites (<50 tests) where manual maintenance is manageable
- Resource-constrained teams are unable to implement governance overhead
- Applications with complex multi-user interaction patterns

Our context—a financial services microservice application with extensive third-party integrations—represents a favorable case for healer agents. Teams in different contexts should carefully assess whether their failure modes align with the healer agent's strengths.

VII. THREATS TO VALIDITY

A. Internal Validity

Confounding factors: During the evaluation period, the development team implemented several other stability improvements (upgraded the Playwright version, improved the staging infrastructure, and refactored flaky tests). While we attempted to attribute improvements to healer implementation through detailed telemetry, complete isolation was not possible. The 73% reduction likely includes contributions from these concurrent improvements.

Hawthorne effect: Developers' awareness of the study may have influenced their attention to test maintenance, potentially improving baseline stability independent of healer implementation.

Regression masking detection: We identified three masking incidents through explicit reports and alerts. Additional masked regressions may have reached production undetected, particularly for non-critical UI bugs or subtle performance degradation.

B. External Validity

Single-organization study: Results are based on one company's test suite and may not generalize to other application domains, team sizes, or engineering cultures.

Specific technology stack: Our implementation targeted Playwright with TypeScript. Adaptation to other frameworks (Selenium, Cypress, Puppeteer) may face different challenges and yield different results.

Application characteristics: The application under test featured extensive third-party integrations and microservice complexity. Simpler applications may see less dramatic improvements.

Team maturity: Our engineering team has strong CI/CD practices and observability infrastructure. Teams with less mature testing practices may struggle to implement governance.

C. Construct Validity

False-positive classification: The baseline false-positive rate of 73.2% was determined through manual triage by engineers. This classification may have been subjective or inconsistent, affecting the validity of improvement measurements.

Developer survey bias: Survey responses came from volunteers, potentially introducing selection bias toward more engaged or positive participants.

MTTR measurement: MTTR calculations included time from failure detection to fix deployment, but excluded time for fix verification and potential regression. This may understate actual resolution time.

VIII. Future Work

A. Enhanced LLM Reasoning

Current healer implementations use LLMs primarily for selector identification and element matching. Future work could explore:

- **Multi-step reasoning:** Enabling LLMs to reason about multi-page flows and complex state dependencies
- **Visual understanding:** Integrating vision models to identify UI elements through screenshots rather than DOM inspection
- **Causal reasoning:** Training models to distinguish correlation from causation in failure analysis

B. Predictive Healing

Rather than reactively healing failures, agents could proactively identify likely failure modes:

- **Pre-execution analysis:** Analyzing test code and current application state to predict which tests are likely to fail and why
- **Adaptive timeouts:** Dynamically adjusting timeouts based on historical execution patterns and current system load
- **Preventive stubbing:** Automatically stubbing external services when availability monitoring indicates potential issues

C. Collaborative Human-Agent Healing

Hybrid approaches combining agent automation with human judgment:

- **Human-in-loop healing:** Agent proposes healing strategies but requires human approval before execution
- **Interactive debugging:** Agent assists the engineer in real-time debugging sessions by suggesting hypotheses and gathering evidence
- **Learning from human fixes:** Agent observes how engineers manually fix tests and learns to apply similar patterns

D. Cross-Suite Learning

Extending healer capabilities across multiple test suites and organizations:

- **Transfer learning:** Training models on healing patterns from multiple codebases to improve generalization
- **Shared heal pattern libraries:** Open-source repositories of proven healing patterns for common frameworks and services
- **Federated learning:** Collaborative model training across organizations while preserving proprietary test details

E. Formal Verification Integration

Combining self-healing with formal methods to ensure correctness:

- **Heal correctness proofs:** Formally verifying that healing transformations preserve test intent
- **Invariant checking:** Ensuring healed tests still validate critical application invariants
- **Regression detection guarantees:** Providing formal bounds on regression masking probability

IX. Conclusion

This research demonstrates that AI-assisted self-healing via Playwright's Healer Agent can substantially improve E2E test stability, reducing false-positive failures by 73% in a production test suite. However, effectiveness varies significantly by failure category, with redirect handling and timing issues showing >90% success rates while infrastructure failures show limited amenability to healing.

The implementation revealed a critical tension: aggressive healing improves stability but risks masking legitimate regressions. Our experience with three masking incidents underscores the necessity of robust governance mechanisms—particularly heal visibility, budget limits, regular review, and stratified risk policies based on heal invasiveness.

Key contributions of this work include:

1. **Empirical validation** of healer agent effectiveness in a production context with statistical significance
2. **Concrete implementation patterns** for standard failure modes (redirect handling, iframe polling, selector fallbacks)
3. **Governance framework** balancing stability benefits against regression masking risks
4. **Characterization of failure modes** most amenable to automated healing
5. **Limitations analysis** identifying scenarios where healing is ineffective or counterproductive

For teams considering healer agent adoption, we offer the following guidance:

Adopt healer agents when:

- Test suite exhibits >30% false positive rate due to timing, external services, or UI changes
- Application features extensive third-party integrations or feature flags
- The team has the capacity to implement telemetry and governance infrastructure
- Rapid feedback loops are critical to development velocity

Avoid or delay adoption when:

- The test suite is small (<50 tests) and manually maintainable
- Strict audit requirements preclude automated state mutation
- Team lacks CI/CD maturity for proper observability
- Application is largely deterministic without external dependencies

The emergence of AI-powered testing agents represents an important evolution in test automation, shifting from static scripts to adaptive systems capable of self-repair. While not a panacea—infrastructure issues and complex bugs still require human intervention—healer agents provide pragmatic improvements in test stability when deployed with appropriate governance. As LLM capabilities advance and the testing community develops better patterns and practices, we anticipate self-healing tests will become a standard component of mature CI/CD pipelines.

The fundamental insight from this work is that **test stability and diagnostic value need not be in opposition**. With careful design, healer agents can improve stability while maintaining—indeed enhancing—the diagnostic information available to engineers through comprehensive telemetry. The key is transparency: making healing visible, auditable, and governable rather than silently masking problems.

X. Acknowledgments

We thank the engineering team members who participated in the healer agent implementation, provided survey responses, and contributed to weekly heal review meetings. Special thanks to the QA engineering team for their diligent tracking of regression masking incidents and their contributions to the governance framework. We also acknowledge the Playwright team at Microsoft for their work on the Agent framework that enabled this research.

References

[1] Leotta, M., Clerissi, D., Ricca, F., & Tonella, P. (2013). Approaches and tools for automated end-to-end web testing. *Advances in Computers*, 101, 193-280.

[2] Luo, Q., Hariri, F., Eloussi, L., & Marinov, D. (2014). An empirical analysis of flaky tests. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 643-653.

[3] Eck, M., Palomba, F., Castelluccio, M., & Bacchelli, A. (2019). Understanding flaky tests: The developer's perspective. *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 830-840.

[4] Thorve, S., Sreshtha, C., & Meng, N. (2018). An empirical study of flaky tests in Android apps. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 534-538.

[5] Parry, O., Kapfhammer, G. M., Hilton, M., & McMinn, P. (2021). Surveying the developer experience of flaky tests. *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*, 253-262.

[6] Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., & Marinov, D. (2018). DeFlaker: Automatically detecting flaky tests. *Proceedings of the 40th International Conference on Software Engineering*, 433-444.

[7] Micco, J. (2016). Flaky tests at Google and how we mitigate them. *Google Testing Blog*. Retrieved from https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html

[8] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

[9] Eck, M., Palomba, F., Castelluccio, M., & Bacchelli, A. (2020). A large-scale empirical study on the effects of code coverage on bug detection in continuous integration. *Proceedings of the 17th International Conference on Mining Software Repositories*, 213-224.

[10] Stocco, A., Leotta, M., Ricca, F., & Tonella, P. (2018). WATER: Web Application Test Repair. *Proceedings of the 1st International Workshop on Machine Learning Techniques for Software Quality Evaluation*, 11-16.

[11] Coppola, R., Morisio, M., & Torchiano, M. (2019). Mobile GUI testing fragility: A study on open-source Android applications. *IEEE Transactions on Reliability*, 68(1), 67-90.

[12] Leotta, M., Stocco, A., Ricca, F., & Tonella, P. (2020). Automated generation of visual web tests from DOM-based web tests. *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 1221-1232.

[13] Liu, J., Lin, Y., Liu, Z., & Sun, M. (2023). GPT-3-driven test case generation: An empirical study. *Proceedings of the 2023 IEEE/ACM International Conference on Software Engineering*, 892-903.

[14] Schafer, M., Nadi, S., Gousios, G., & Chauduri, S. (2023). Adaptive test repair with large language models. *arXiv preprint arXiv:2309.12345*.

[15] Microsoft. (2024). Playwright Agents: AI-powered test automation. *Playwright Documentation*. Retrieved from https://playwright.dev/docs/agents