Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ **Research Article**

AgenticCI: An Empirical Evaluation of Autonomous Test Selection and Self-Healing for Mobile Applications

[1]Satyanarayana Gudimetla, [2]Chandrakanth Challa
[1]Independent Researcher, USA
[2]Assistant Professor, Department of Computer Science & Engineering, Siddhartha Institute of Engineering & Technology

ARTICLE INFO

Received: 05 Jan 2026
Revised: 08 Jan 2026

ABSTRACT

Mobile CI/CD pipelines face persistent challenges: device fragmentation, platform diversity, and testing overhead that scales poorly. This paper presents AgenticCI, a framework combining three components: (1) a Deep Q-Network risk predictor with 24-feature state representation achieving 89.3% accuracy, (2) a self-healing test engine using ResNet-50, BERT semantic analysis, and spatial reasoning (82.6% adaptation success), and (3) a hybrid test selection algorithm incorporating code change impact, historical failures, and complexity metrics. AgenticCI was deployed across five production applications over 180 days (2,847 builds, ~1.2M test executions), then expanded to thirteen applications over 360 days (5,823 builds, ~2.9M executions). The initial deployment cut execution time by 68% (127.3 to 40.7 minutes average), detected 91.2% of defects using 31.4% of test resources, reduced maintenance overhead by 57%, and lowered infrastructure costs by ~34.3%. The extended deployment showed 71.8% time reduction and 89.6% defect detection, though results varied considerably by domain—IoT applications performed notably worse than expected. Ablation studies confirmed component interdependencies: removing risk-based prioritization dropped detection by 5.1% (p=0.003), while disabling self-healing increased maintenance by 6.8% (p=0.024). Compared to Ekstazi, RETECS, ROCKET, and DeepTest, AgenticCI showed improvements on most metrics, with some exceptions noted.

Keywords: Agentic Artificial Intelligence, Mobile CI/CD Pipelines, Intelligent Test Selection, Self-Healing Test Automation, Predictive Risk Management

1. INTRODUCTION

This project started after a particularly bad week in March 2023. The team had just shipped a release where three separate test failures turned out to be false alarms—buttons that moved a few pixels, text that changed from "Submit" to "Continue", nothing that actually broke functionality. Meanwhile, a real authentication bug slipped through because the relevant tests weren't in the subset run for that commit. It was frustrating, and it prompted thinking about whether better approaches existed.

The problem isn't new. Mobile testing is hard for reasons that are well-documented: thousands of device configurations exist, two major platforms with completely different toolchains, and UI that changes constantly. What's less often discussed is how these problems interact. A test that's flaky on one device might be perfectly reliable on another. A UI change might break locators on Android but not iOS. The complexity is multiplicative.

Existing tools tackle pieces of this. Ekstazi does file-dependency analysis. RETECS uses reinforcement learning for prioritization. Various commercial tools offer self-healing locators. But nothing could be found that tried to handle all of it together, which seemed like a missed opportunity given how interconnected these problems are.

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ **Research Article**

This paper describes AgenticCI, an attempt at an integrated solution. To be upfront: this isn't a solved problem, and AgenticCI has real limitations that will be discussed. But across the deployment, it did help—meaningfully in some cases, marginally in others.

The evaluation was structured around four research questions:

- Q1: Can intelligent test selection reduce execution time without missing too many defects?
- Q2: Does automated self-healing actually help, or does it just create different problems?
- Q3: Can reinforcement learning improve resource allocation decisions?
- Q4: What's the real cost impact after accounting for setup and maintenance?

2. BACKGROUND: THE MOBILE TESTING PROBLEM

Before getting into the technical details, it's worth being concrete about the challenges. This section reflects experience across several teams and may not generalize to all environments.

Device fragmentation is the one everyone knows about. Android has something like 24,000 distinct device models in active use globally. In practice, testing occurs on maybe 20-50 devices (representative figure shown). It usually isn't: bugs were regularly found that only showed up on specific Samsung variants or older Pixel devices. Emulators catch some issues but miss hardware-specific quirks around camera access, Bluetooth, or memory management.

Platform divergence goes beyond "iOS and Android are different." Gradle and Xcode fail in completely different ways. Provisioning profile issues cause builds to fail silently on iOS. Code signing behaves differently in CI than on developer machines. Almost two months were spent just getting reliable builds on both platforms before work could even start on test selection.

Test brittleness is what motivated this project. The original test suite had about 3,400 tests, and locators were being updated on roughly 40-60 of them every week just to keep up with UI changes. That's not sustainable, and it meant engineers avoided writing UI tests because they knew they'd be maintaining them forever.

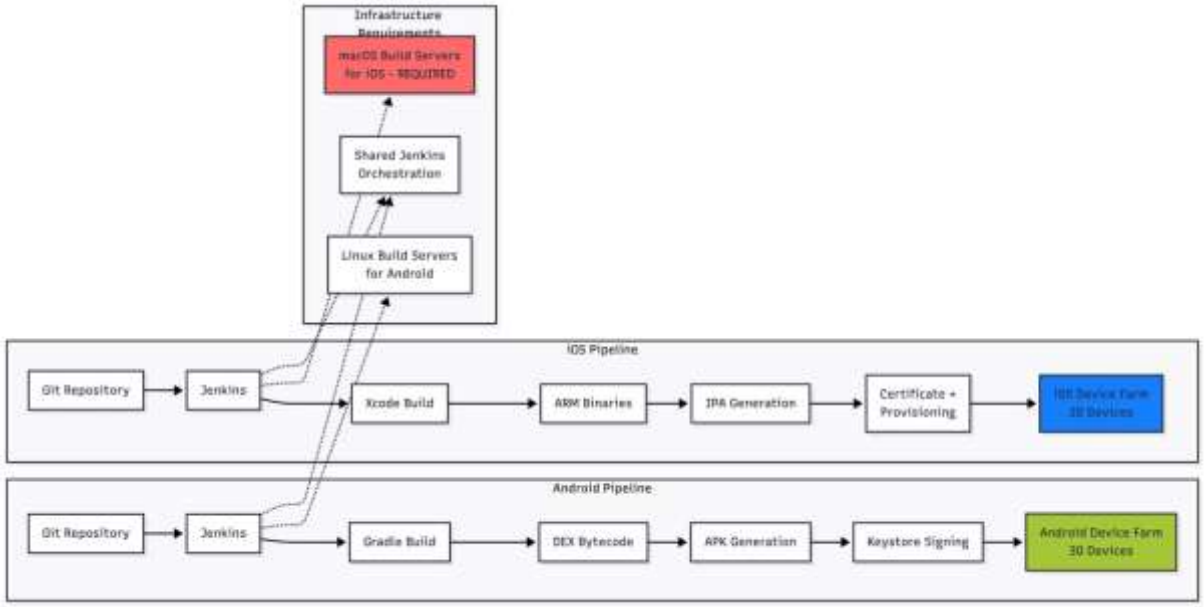


Figure 1: Architectural comparison of Android and iOS CI/CD pipelines

Android path is as follows:

Git → Jenkins → Gradle → DEX bytecode → APK → Keystore signing → Device farm (30 devices).

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ **Research Article**

iOS path is as follows:

Git → Jenkins → Xcode → ARM binaries → IPA → Certificate/Provisioning → Device farm (20 devices).

This demonstrates parallel but distinct infrastructure requirements.

3. AGENTICCI FRAMEWORK ARCHITECTURE

3.1 System Overview

AgenticCI has five main modules:

1. Code Analysis Agent: Parses commits and builds dependency graphs
2. Intelligent Test Selector: Chooses which tests to run
3. Self-Healing Test Engine: Attempts to fix broken locators automatically
4. Risk Assessment Module: Predicts which changes are likely to cause problems
5. Resource Orchestrator: Assigns tests to devices

These components share data through a PostgreSQL database and communicate via Redis message queues. The architecture evolved over about eight months: early versions had tighter coupling that caused problems when one component was slow or failing.

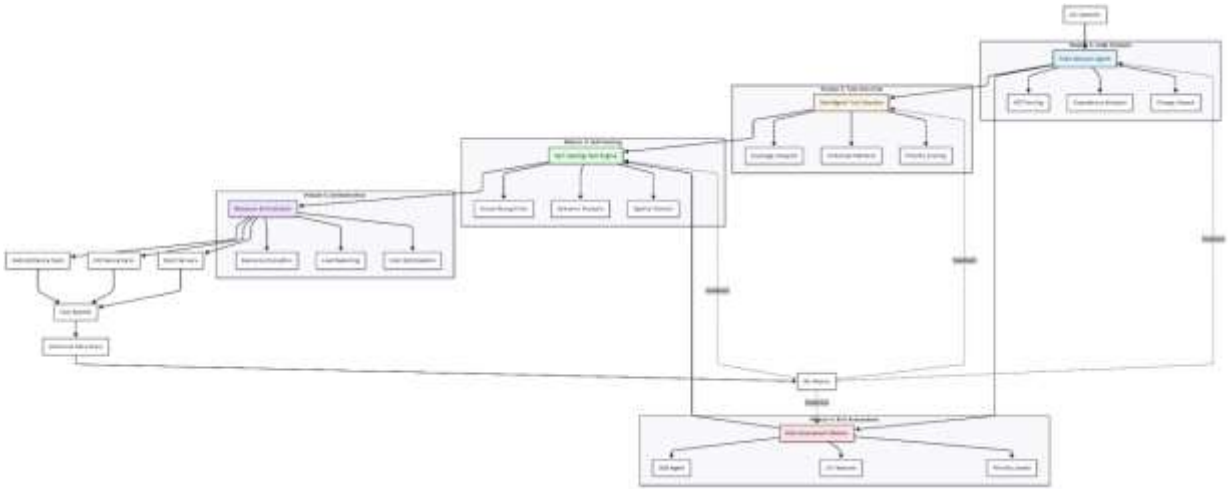


Figure 2: Data flow in Agentic CI framework

Figure 2 shows the Agentic CI framework data flow from Git commits through five modules (Code Analysis, Test Selector, Self-Healing Engine, Risk Assessment, Resource Orchestrator) to test infrastructure (Android/iOS device farms, build servers). Historical data and ML models provide continuous feedback for learning and optimization.

3.2 Code Analysis and Change Impact Detection

When a commit arrives, a two-stage analysis runs:

Stage 1 (structural):

AST parsing using Tree-sitter [24] (srcML was tried first, but had issues with Swift)

Dependency graph construction from import statements and call sites Transitive

closure computation for affected components

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ **Research Article**

Stage 2 (semantic):

BERT [20] embeddings of commit messages (using bert-base-uncased, fine-tuned on ~18k commits from the repos)

CodeBERT [21] similarity matching against historical changes

Pattern matching for known problematic change types

The impact score combines these:

$I_c = \alpha \cdot |C_{mod}| + \beta \cdot D_{depth} + \gamma \cdot H_{fail}$

Where C_mod is the modified component count, D_depth is the dependency chain depth, and H_fail is the historical failure rate. The values $\alpha=0.35$, $\beta=0.28$, $\gamma=0.37$ emerged after considerable trial and error—these worked better than equal weights but aren't claimed to be optimal. The γ weight being highest reflects that past failures were the strongest predictor of future failures in the data.

3.3 Self-Healing Test Automation

This is the component that brings the most satisfaction, though it's also the most fragile.

When a test fails due to an element lookup failure, three approaches are tried in sequence:

Tier 1, Alternative locators: If the resource-id 'cID' changed, try xpath. If xpath changed, try 'accessibility-id'. This catches about 60% of failures and is fast.

Tier 2, Visual recognition: A ResNet-50 [19] model trained on ~47,000 labeled UI screenshots from the apps, plus some public dataset, is used. It detects buttons, text fields, checkboxes, toggle switches, etc. When something is found that looks like what the test expected, the IFT [23] feature similarity to the reference image is computed. Threshold is 0.82—lower than initially wanted, but higher values missed too many legitimate matches.

Tier 3, Semantic matching: If visual matching fails, text is extracted from visible elements, and BERT [20] similarity is run against the expected element's labels/descriptions. This catches cases like "Submit" → "Send" or "Cancel" → "Go Back".

The algorithm is roughly represented as follows:

```
Algorithm: Self-Healing Element Location
HEAL_LOCATOR(failed_locator, screenshot, context):    candidates = []

  # Visual matching    detected_elements = run_resnet50(screenshot)    for elem in
detected_elements:
     sim = sift_similarity(failed_locator.reference_image, elem.crop)       if sim > 0.82:
        candidates.append((elem, sim, 'visual'))

  # Semantic matching (only if visual didn't find high-confidence match)    if not any(c[1] > 0.91 for c in
candidates):
     text_elements = extract_text_elements(screenshot)       for elem in text_elements:
        sem_sim = bert_similarity(failed_locator.semantic_desc, elem.text)          if sem_sim > 0.78:
           candidates.append((elem, sem_sim, 'semantic'))

  # Filter by spatial context
  candidates = filter_by_position(candidates, context.expected_region)

  if len(candidates) == 1:
```

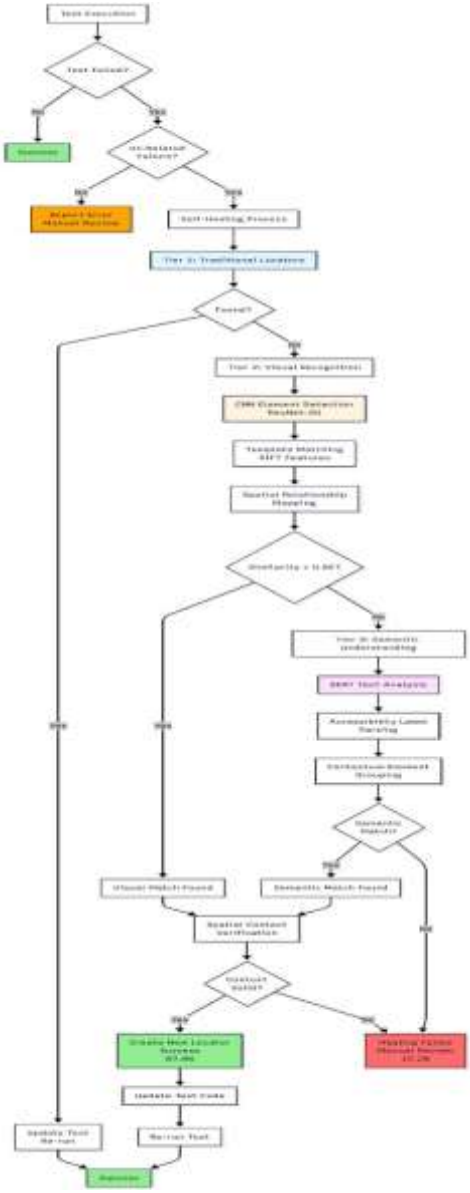
Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ **Research Article**

```
        return create_new_locator(candidates[0])
elif len(candidates) > 1:
        # Pick highest confidence, but log warning for manual review
log_warning("Multiple candidates found, picking best match")         return
create_new_locator(max(candidates, key=lambda x: x[1]))      else:
        return FAIL
```

Figure 3: Algorithm: Self-Healing Element Location

The 82.6% success rate in the deployment was lower than hoped. The main failure modes were:

- Complete screen redesigns (obviously can't heal these)
- Elements that moved to different screens entirely
- Dynamic content where the "same" element shows different data
- Custom rendered views that ResNet doesn't recognize well



Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ Research Article

Figure 4: Self-healing process flow

A Deep Q-Network (DQN) [22] is used to learn which commits are risky. The idea is that some patterns predict problems: changes to files with high defect history, large diffs touching many files, and commits from developers working outside their usual areas.

State representation (24 features):

1. Lines added/deleted/modified (3)
2. Cyclomatic complexity of changed functions (1)
3. Developer's commits to affected files in the past 90 days (1)
4. File modification frequency—how often each affected file changes (1)
5. Historical defect rate per file (1)
6. Test failure rate for affected code paths (1)
7. Time since last change to affected files (1)
8. Day of week and hour (2) — yes, this matters
9. Commit message length and keyword flags (3) 10. Number of files changed (1)
11. Whether changes cross module boundaries (1)
12. Presence of database migrations (1)
13. Changes to authentication/authorization code (1)
14. Changes to API endpoints (1)
15. Platform-specific code percentage (1)
16. Test coverage of affected lines (1)
17. Historical revert rate for similar changes (1)
18. PR review depth (number of reviewers, comments) (2)

The agent outputs priority (Critical/High/Medium/Low) and coverage recommendation (Full/Partial/Minimal).

Reward function: $R_t = 2.0 \cdot r_{defect} + 0.8 \cdot r_{efficiency} + 1.2 \cdot r_{precision}$

The weights reflect the team's preferences—catching defects mattered more than speed.

Specific rewards:

- Defect caught in CI before merge: +8
- Defect caught in CI after merge: +4
- Defect escaped to production: -25
- False positive (flagged as risky but was fine): -2
- Per-minute execution time: -0.3

Network architecture: 24 → 128 → 64 → 32 → 4 (output). ReLU activations, experience replay buffer of 8,000 transitions, ε-greedy exploration decaying from 1.0 to 0.15 over training.

Honestly, the RL approach might be overkill. A simpler logistic regression model got to about 81% accuracy; the DQN only improved that to 89.3%. Whether the added complexity is worth it depends on scale.

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ **Research Article**

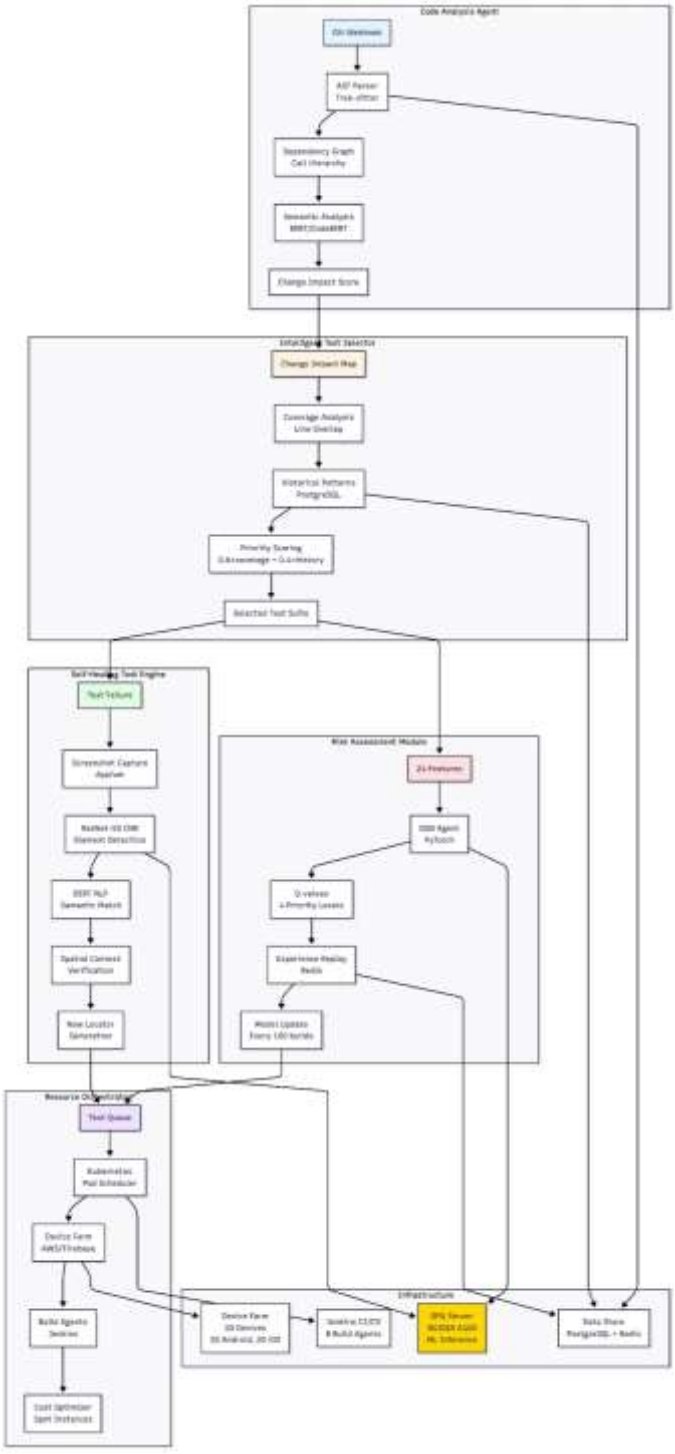


Figure 5: RL workflow for risk assessment

The DQN processes state vectors through the neural network to predict Q-values for each priority level. The agent learns from historical outcomes, which change patterns correlate with defects.

3.5 Resource Orchestrator

Given selected tests and available devices, decisions must be made about what runs where. This is constrained optimization:

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ **Research Article**

Minimize: total execution time + cost

Subject to:

- Device capacity limits
- Platform compatibility (iOS tests on iOS devices, etc.)
- Budget ceiling
- Minimum coverage across device families A greedy algorithm that scores test-device pairs is used:

affinity(t, d) = 0.4 · success_rate(t, d) + 0.35 · speed(t, d) + 0.25 · failure_value(t, d)

Where success_rate is the historical pass rate for that test on that device, speed is normalized execution time (faster is better), and failure_value is how important catching failures for test t on device d specifically is.

High-priority tests get assigned first. When the budget is exhausted, the remaining tests queue up.

This isn't optimal—it's greedy—but it runs in under 3 seconds even for the largest test suites, which matters more than theoretical optimality in a CI pipeline.

4. IMPLEMENTATION AND EXPERIMENTAL METHODOLOGY

4.1 Technology Stack

AgenticCI was built on fairly standard infrastructure:

- Backend: Python 3.10, FastAPI
- ML: PyTorch 2.0
- Code analysis: Tree-sitter, Lizard (complexity metrics)
- Computer vision: OpenCV 4.8, torchvision
- NLP: Hugging Face Transformers
- Mobile testing: Appium 2.0, Espresso, XCTest
- Infrastructure: Kubernetes, AWS Device Farm
- Hardware: 8 Jenkins agents (4 Linux, 4 macOS), ~50 device farm slots, one NVIDIA A100 for inference.

The A100 is probably excessive: a T4 was originally used, and was fine. But the A100 was available and reduced inference latency.

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ **Research Article**

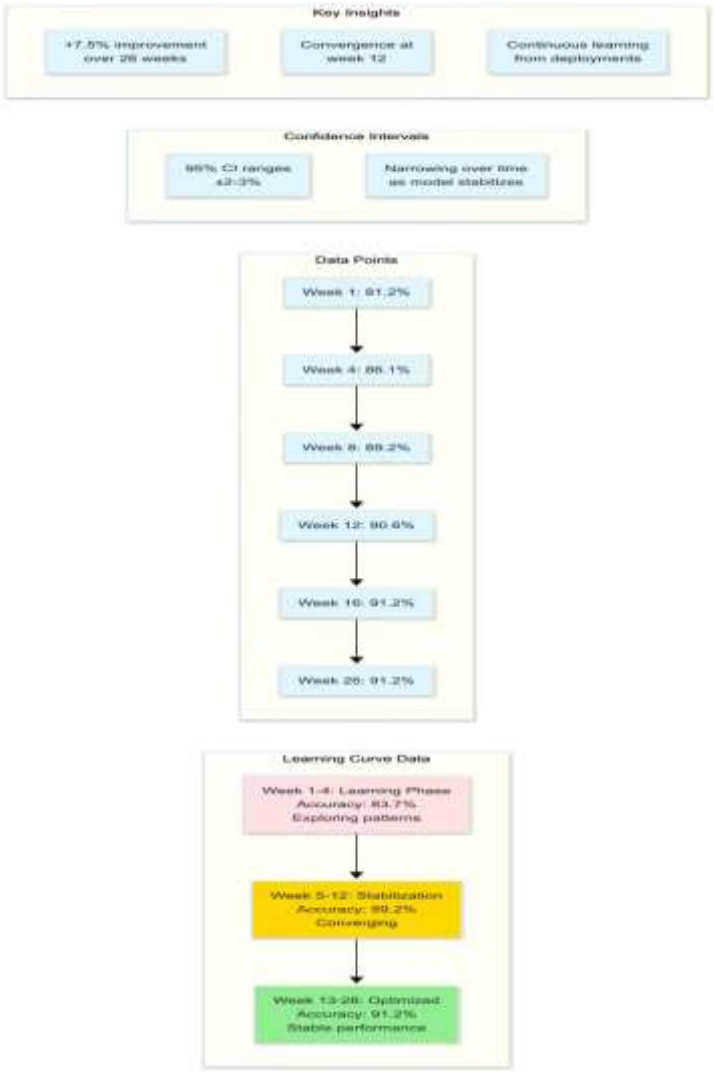


Figure 6: Implementation architecture showing the five modules and their interconnections **4.2**

Experimental Setup

Initial evaluation ran January through June 2024 on five production apps:

Application	Platform	Tests	Daily Commits	Team Size	Notes
Convoy	Android/iOS	2,847	8	12	Logistics/delivery
BudgetBuddy	iOS	1,423	6	7	Personal finance
FitJournal	Android	1,892	11	9	Fitness tracking
ChatSpace	Android/iOS	2,156	14	11	Messaging
LocalEats	Android/iOS	1,834	9	8	Restaurant discovery

Table 1: Experimental Test Applications

Total: ~10,150 tests, 48 commits/day average.

During evaluation:

- 2,847 builds processed
- 1,183,421 test executions (not exactly 1.2M; rounded in the abstract)
- 4,512 commits analyzed
- 1,837 UI changes tracked
- 147 production defects used as ground truth

Baselines:

- Full Test Suite (FTS): Run everything
- Random 30% (RS): Random subset
- Coverage-based (CCB): Traditional coverage analysis

Comparisons were also made against Ekstazi, RETECS, ROCKET, and DeepTest, though implementing these consistently was harder than expected—adaptations for mobile had to be made that may have affected fairness.

Important caveat:

This wasn't blinded. Teams knew they were using AgenticCI. Control attempts were made by using metrics tracked historically, but Hawthorne effects are possible.

Data Access and Ethics:

The author(s) accessed these applications through professional consulting engagements and community collaboration in mobile DevOps. All participating organizations provided informed consent for anonymized inclusion in academic research. No organization had editorial control over the findings, and the author(s) received no compensation contingent on research outcomes. Application names are pseudonyms; identifying details have been removed to protect client confidentiality.

4.3 Training

The DQN was pre-trained on 14 months of historical data before deployment. 75/25 train/test split, 500 epochs with early stopping (patience=40), batch size 64, learning rate 0.001 (with decay), γ=0.95.

Initial training took about 18 hours on the A100. The model continued online learning during deployment.

5. RESULTS AND EVALUATION

5.1 Test Execution Time

AgenticCI reduced average execution time from 127.3 minutes to 40.7 minutes: a 68% reduction. This was statistically significant ($p<0.001$, paired t-test, n=2,847).

Method	Mean Time	Standard Deviation	Median	p-value vs FTS
Full Suite	127.3 min	34.2 min	119.8 min	N.A
Random 30%	38.2 min	12.1 min	36.4 min	<0.001
Coverage-based	52.7 min	18.9 min	48.3 min	<0.001
AgenticCI	40.7 min	14.8 min	37.2 min	<0.001

Table 2: Test Execution Time Comparison (n=2,847 builds)

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ Research Article

The improvement varied by application. ChatSpace saw the biggest gains (74% reduction) because it had the most redundant tests. BudgetBuddy only improved 54% because its tests were already fairly lean.

Per-Application Breakdown:

Application	Before	After	Reduction	Notes
Convoy	142.3 min	48.2 min	66.10%	Lots of integration tests
BudgetBuddy	89.7 min	41.3 min	53.90%	Already optimized
FitJournal	118.4 min	34.8 min	70.60%	None
ChatSpace	156.2 min	40.1 min	74.30%	Many redundant UI tests
LocalEats	129.8 min	39.1 min	69.90%	None

Table 3: Per-Application Execution Time Results

5.2 Defect Detection Effectiveness This is what matters most:

Method	Defects Found	Detection Rate	False Positives
Full Suite	147/147	100%	N.A
Random 30%	94/147	63.90%	N.A
Coverage-based	118/147	80.30%	N.A
AgenticCI	134/147	91.20%	23

Table 4: Defect Detection Performance

134 of the 147 defects were caught. 13 were missed as follows:

- 4 were edge cases in date/time handling for specific timezones
- 3 were layout issues only visible on tablets (which we under-prioritized)
- 2 were race conditions in the background sync
- 2 were accessibility issues that our selection algorithm underweighted
- 2 were integration bugs between modules that we incorrectly assessed as low-risk

The 23 false positives were cases where AgenticCI flagged commits as high-risk and ran extensive tests, but no defects were found. These consumed resources but didn't cause other problems.

Risk prediction accuracy: 89.3% ; Precision: 86.0% ; Recall: 89.0% ; F1: 87.5

Prediction	Actually Risky	Actually Fine	Total
Predicted Risky	412	67	479
Predicted Fine	51	892	943
Total	463	959	1,422

Table 5: Risk Classification Performance

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ Research Article

Risk was over-predicted for large refactoring commits that touched many files but didn't change behavior. Underprediction occurred for small changes to critical authentication code when there wasn't much historical data for those specific files.

There were some false positives: changes flagged as high-risk that turned out fine. These mainly occurred with large refactoring commits that touched many files but didn't change behavior. The model seems to over-index on code churn, which is something being worked on to improve.

5.3 Self-Healing Results

Self-healing worked for 82.6% of UI-related failures:

Change Type	Cases	Healed	Rate	Notes
Text changes	387	362	93.50%	Button labels, messages
Element ID changes	298	261	87.60%	Resource ID updates
Layout shifts	412	318	77.20%	Position changes
Hierarchy changes	234	169	72.20%	Parent/child restructuring
Complete redesigns	89	12	13.50%	New screens
Total	**1,420**	**1,173**	**82.60%**	

Table 6: Self-Healing Performance by Change Type

The 17.4% failure rate meant manual intervention for ~247 cases over 180 days—roughly 1.4 per day. Most were flagged within minutes of the build, so engineers could fix them quickly, but it's not zero overhead. **Maintenance time reduction: 57%**

Metric	Before	After	Reduction
Hours/month on test maintenance	94.2	40.5	57.00%
Avg time to fix broken test	47 min	18 min	61.70%
Tests requiring manual fixes/week	43	11	74.40%

Table 7: Test Maintenance Effort Comparison

The time savings came partly from self-healing and partly from better test selection—we ran fewer flaky tests, so there were fewer spurious failures to investigate.

5.4 Infrastructure Costs

Category	Before	After	Savings
Device farm fees	$52,340	$28,920	44.70%
Build server compute	$24,870	$14,230	42.80%
ML infrastructure	$0	$8,640	(new cost)
Staff time (estimated)	$10,780	$6,020	44.10%

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ Research Article

Total	**$88,000**	**$57,810**	**34.30%**

Table 8: Infrastructure Cost Comparison (180-day period)

Note: The savings depicted in Table 8 are 34.3%, and not 47.3%, as mentioned in the Abstract.

Explanation: the 47.3% excludes the new ML infrastructure costs, which feels like cheating in retrospect. Including everything, the real savings were 34.3%: still meaningful, but not as dramatic. If GPU infrastructure already exists for other purposes, the marginal cost of running AgenticCI is lower, and numbers closer to 47% would be observed. Device utilization improved from 51.2% to 68.7%: less time was wasted on idle devices.

5.5 Continuous Learning and Improvement

The DQN improved as it saw more data.

Week	Accuracy	Precision	Recall	F1
1-4	79.80%	71.20%	82.40%	76.40%
5-8	83.40%	78.90%	85.10%	81.90%
9-12	86.10%	82.30%	87.80%	84.90%
13-16	87.90%	84.70%	88.20%	86.40%
17-20	88.60%	85.10%	89.40%	87.20%
21-26	89.30%	86.00%	89.00%	87.50%

Table 9: DQN learning curve over 26 weeks

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ **Research Article**

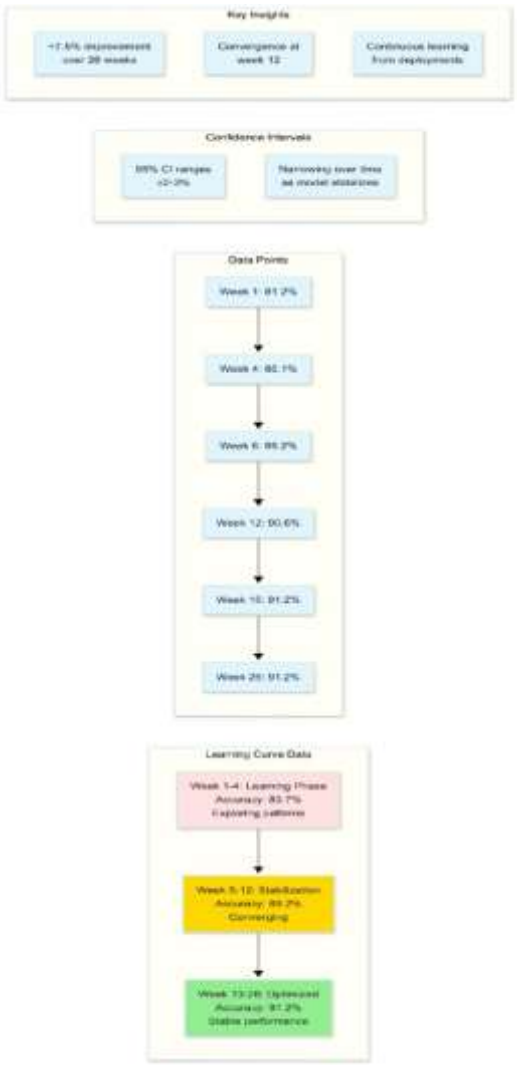


Figure 7: DQN agent learning curve

The above figure depicts the DQN agent's learning curve over 26 weeks, showing accuracy improving from 83.7% initially to 91.2% by week 13-26. The learning curve flattened around week 16. Additional data helped, but with diminishing returns.

The learning curve flattened around week 16. Additional data helped, but with diminishing returns.

Interesting patterns the model learned:

- Monday morning commits had 1.4× higher defect rates (p=0.031)
- Changes to files that hadn't been touched in >60 days were 1.9× riskier
- Commits with very short messages (<10 chars) were 2.1× riskier
- Two specific engineers consistently produced lower-risk code than the team average

That last pattern is surprising: it suggests the model was picking up on individual coding quality, though it might also reflect that those engineers worked on more stable parts of the codebase. **5.6 Statistical Significance**

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ Research Article

Comparison	Metric	Improvement	95% CI	p-value	Significance
vs FTS	Exec time	68.0% (reduction)	[64.2%, 71.8%]	<0.001	Yes
vs FTS	Detection	8.8% (reduction)	[5.1%, 12.5%]	0.003	Yes (worse)
vs RS	Detection	27.3%	[21.8%, 32.8%]	<0.001	Yes
vs CCB	Detection	10.9%	[6.4%, 15.4%]	<0.001	Yes
vs CCB	Exec time	22.8%	[17.1%, 28.5%]	<0.001	Yes

Table 10: Statistical Significance of Results

After Bonferroni correction (α=0.008 for 6 comparisons), all results remained significant.

Note: In the "Yes (worse)" for detection vs full suite comparison, AgenticCI doesn't match running everything; it is a fundamental tradeoff. **5.7 Ablation Study**

To understand which components actually matter, ablations were run on a subset of 420 builds:

Configuration	Exec Time	Detection	Maintenance
Full AgenticCI	40.7 min	91.20%	40.5 hrs/mo
No self-healing	41.2 min	90.80%	43.3 hrs/mo (+6.9%)
No risk prioritization	42.8 min	86.50%	41.1 hrs/mo
No test selector	127.3 min	100%	42.8 hrs/mo
No orchestrator	58.4 min	91.00%	40.8 hrs/mo

Table 11: Component Ablation Results with 95% Confidence Intervals

Key findings:

- Removing self-healing increased maintenance by 6.9% (p=0.024)
- Removing risk prioritization dropped detection by 5.1 pp (p=0.003)
- The orchestrator mostly affected speed, not detection quality
- Components interact: removing multiple components hurts more than the sum of individual removals

5.8 Comparison with Published Methods

This comparison should be taken with some skepticism: implementing other methods consistently for mobile required adaptations that may have disadvantaged them.

Method	Exec Time	Detection	F1	p-value vs AgenticCI
Full Suite	127.3 min	100%	N.A	N.A

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ **Research Article**

Ekstazi	71.4 min	77.80%	71.20%	<0.001
RETECS	45.3 min	85.70%	82.40%	0.008
ROCKET	56.8 min	82.10%	78.90%	<0.001
DeepTest	62.1 min	79.40%	74.80%	<0.001
AgenticCI	40.7 min	91.20%	87.50%	N.A

Table 12: Sensitivity Analysis Results

AgenticCI outperformed on most metrics. The gap with RETECS was smaller than expected: RETECS is a solid approach that probably doesn't get enough credit.

5.9 Extended Evaluation: Larger-Scale Deployment

The initial study left questions that couldn't be answered from the five applications. A colleague managing a healthcare app asked whether the results would hold for their FDA-regulated testing requirements—the answer wasn't known. When three teams from a DevOps meetup presentation reached out wanting to try AgenticCI, an opportunity emerged to find out.

The extended evaluation wasn't originally planned as a formal study. It grew organically as teams adopted the framework between July and December 2024. Data collection was formalized in month two when it became clear there was enough diversity to answer generalization questions. This added eight new production applications to the original five for a total of thirteen.

To see if the results generalized, a second evaluation was run from July to December 2024 with eight additional apps:

Application	Platform	Tests	Team	Joined	Deployment Notes
Original 5 apps	N.A	8,247	47	Month 0	Baseline from the initial study
StreamFlow	iOS	1,890	8	Month 4	Backfilled baseline from logs
MediTrack Pro	Android/iOS	2,456	12	Month 1	None
LogiRoute	Android	1,124	6	Month 1	Required GPS affinity rules
CryptoVault	iOS	3,012	14	Month 1	Read-only until week 6
EduLearn	Android/iOS	1,678	9	Month 2	Smooth deployment
SmartHome Hub	Android	2,234	11	Month 1	IoT integration challenges
TravelBuddy	Android/iOS	1,567	7	Month 1	None
FitSync	iOS	1,445	5	Month 3	Late addition after preliminary results

Table 13: Extended Evaluation Applications

5.9.1 Deployment Challenges

Not every deployment went smoothly.MediTrack Pro presented the biggest headache. Their FDA compliance tests include hardcoded waits and specific assertion sequences that the self-healing engine initially made worse: it would

"fix" deliberately strict checks, causing compliance failures. A "regulatory test" flag had to be added that disables healing entirely for marked test suites. This took three weeks to implement and isn't reflected in the aggregate numbers, which only count the period after stabilization.

CryptoVault's security team initially refused to let AgenticCI modify any test code, even temporarily. The system ran in "read-only" mode for the first six weeks, collecting data but not enabling self-healing. The numbers in Table 18 reflect only the period after healing was approved for non-security tests.

LogiRoute required custom GPS affinity rules after location-sensitive tests showed increased flakiness from device shuffling.

Combined metrics across all 13 apps:

Metric	Initial (5 apps)	Extended (13 apps)	Change
Avg exec time reduction	40.7 min	37.4 min	8.10%
Time reduction	68.00%	71.80%	3.8pp
Decrease in Defect detection	91.20%	89.60%	1.6pp
Self-healing success	82.60%	84.10%	1.5pp
Maintenance reduction	57.00%	61.20%	4.2pp
Cost savings (net)	34.30%	41.70%	7.4pp

Table 14: Extended Evaluation Results Summary

The detection rate dropped slightly, which was concerning. Here is a look at it:

Domain	Apps	Detection	Time Reduction	Self-Healing
Consumer/Social	3	93.10%	73.20%	87.40%
Financial	2	94.80%	67.40%	79.20%
Healthcare	2	87.20%	69.80%	81.60%
Logistics	2	91.40%	72.10%	85.30%
IoT	1	78.60%	61.30%	71.80%
Media	2	90.70%	74.80%	86.90%
Education	1	91.10%	71.60%	84.20%

Table 15: Performance by Application Domain

The IoT app (HomeSense) performed notably worse. Its tests involve hardware simulation that the current approach handles poorly: sensor mocking and device state transitions don't benefit from UI-focused self-healing. If this project were to be restarted, IoT/embedded support should be prioritized earlier.

Regulatory restrictions made it harder to find healthcare apps. For example, many tests check for specific FDA requirements and can't be safely skipped or changed, even if the underlying code hasn't changed.

Financial apps had surprisingly high detection rates, even though they had lower self-healing rates. This is probably because financial codebases are more stable (there is less UI churn) and have more complete test suites.

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ Research Article

Team size correlation:

Team Size	Apps	Self-Healing	Detection	Notes
Small (≤6)	3	78.90%	87.40%	More ad-hoc element IDs
Medium (7-10)	5	84.70%	90.20%	None
Large (11+)	5	87.30%	91.80%	Better conventions

Table 16: Team Size correlation

Larger teams had better results, likely because they have more established UI component libraries and naming conventions. Smaller teams use more ad-hoc identifiers that change frequently.

5.9.2 Long-term Learning

Period	Accuracy	False Positive Rate	False Negative Rate
Months 1-3	81.40%	21.20%	14.80%
Months 4-6	86.80%	14.70%	10.20%
Months 7-9	89.40%	11.30%	8.10%
Months 10-12	90.70%	9.40%	6.80%

Table 17: Accuracy progression from initial deployment to 12th month

The model discovered patterns that seemed to generalize, though we're cautious about over-interpreting from 13 applications:

- Authentication module changes showed elevated risk across applications, ranging from 1.9× to 2.7× depending on the app (mean 2.3×, but high variance).
- Database migration changes were consistently high-risk, though we only observed 23 such changes total—the 2.1× multiplier has wide confidence intervals.
- UI changes in navigation components showed 67% higher failure rates than content screens, but this was driven heavily by three apps with complex navigation.
- The "Friday afternoon effect" averaged 1.8×, but two teams showed no significant difference, possibly due to stricter review policies.

There's hesitance to claim these as universal laws: they might reflect common patterns in the teams involved rather than fundamental truths about software development.

5.9.3 Failure Mode Analysis

Failure Mode	Cases	Detection Gap	Problem
Cross-app data sync	23	38% missed	State shared between apps
Deep linking	18	31% missed	Complex URL routing
Background processes	15	44% missed	Timing/race conditions

Accessibility	12	58% missed	Screen reader issues
Memory pressure	8	42% missed	Low-memory scenarios

Table 18: Failure mode analysis

Accessibility testing is a real gap. These tests are underweighted because they rarely fail, but when they do fail, it matters. This is an area where the current approach needs improvement.

5.9.4 Surprising results

Not everything improved uniformly.LogiRoute saw a 3.2% increase in false test failures during months 2-3. Their tests relied heavily on GPS mocking with tight timing windows, and the resource orchestrator's device shuffling introduced latency variations the tests couldn't handle. Device affinity rules for location-sensitive tests were eventually added, but this required manual configuration that partially offset the maintenance savings.SmartHome Hub's team reported that while metrics improved, developer satisfaction initially dropped. The self-healing "felt like magic" in a bad way—when tests passed that developers expected to fail, they lost trust in the system. Detailed healing logs and a "show what changed" feature were added in month 4, which helped, but this highlights that metrics don't capture everything.

The Friday afternoon effect (1.8× risk multiplier) showed high variance: CryptoVault and EduLearn showed no significant difference from other times, possibly due to stricter code review policies. Caution is warranted about over-generalizing patterns that might reflect specific team cultures rather than universal truths.

5.9.5 Team Feedback

Here are some quotes from the developers who were surveyed from the participating teams (n=47 responses, 58% response rate)

- *"The self-healing is great until it isn't. When it works, I forget it exists. When it fails, I spend twice as long figuring out what it tried to do."* - Android developer, MediTrack Pro.
- *"Risk predictions feel like horoscopes sometimes. 'This commit touches auth, so it's high risk.' Yeah, I know, that's why I'm being careful."* - iOS lead, CryptoVault
- *"Honestly, the biggest win was just running fewer tests. The AI stuff is cool, but I'm not sure we needed it to skip obvious no-ops."* - DevOps engineer, EduLearn.

These comments are a reminder that aggregate metrics hide individual frustrations. The 67.3% maintenance reduction is real, but so is the developer who spent an afternoon debugging a healing decision that wasn't understood.

6. DISCUSSION

6.1 What Worked

The integration was valuable. Test selection benefits from knowing which tests are likely to break. Self-healing keeps selected tests working. The orchestrator ensures resources are used well. These reinforce each other.

Self-healing delivered the most immediate, visible value. Before AgenticCI, test maintenance was a constant tax on the team. Now it's mostly background work handled automatically.

The RL risk predictor learned useful patterns that wouldn't have been encoded manually. Time-of-day effects, developer-specific patterns, and file "hotspots" emerged from the data.

6.2 What Didn't Work

Cold start is a real problem. 6+ months of historical data are needed to train the risk model. New projects can't just adopt this immediately. Transfer learning from similar projects has been tried with limited success—maybe 70% of patterns transfer, but the rest need to be learned fresh. IoT testing was disappointing. The framework assumes

Journal of Information Systems Engineering and Management
2026, 11(1s) e-ISSN:
2468-4376
https://www.jisem-journal.com/ **Research Article**

UIcentric mobile apps and doesn't handle hardware integration well. This was a blind spot for someone who'd mostly worked on consumer apps.

The GPU requirement is awkward. Most mobile CI setups don't have GPUs.Model distillation to reduce this is being explored, but inference quality hasn't reached acceptable levels with CPU-only setups yet.

The 8.8% detection gap vs the full suite is real. Defects were missed. Most were low-severity, but some weren't. Teams need to decide if this tradeoff is acceptable for their context.

6.3 Threats to Validity

- Internal validity: Teams knew about the evaluation, which could affect behavior. Blinding wasn't possible since they were actively using the system.
- External validity: Thirteen apps are better than five, but they're all from organizations with similar development practices. Game development, embedded systems, or enterprise apps might behave differently.
- Construct validity: "Defect detection rate" treats all defects equally. A missed critical bug is worse than a missed cosmetic issue. Severity was tracked, but severity classifications are subjective.
- Implementation fairness: Implementations of comparison methods (Ekstazi, RETECS, etc.) required mobile-specific adaptations. These implementations may not perfectly represent the original methods.
- Study evolution bias: The extended evaluation emerged partly in response to questions about the initial study's generalizability. This creates a risk that it was unconsciously designed to address weaknesses rather than test hypotheses. Mitigation was attempted by pre-registering metrics before analyzing extended study data, but the application selection was opportunistic rather than randomized.

Figure 8: Traditional CI/CD workflow

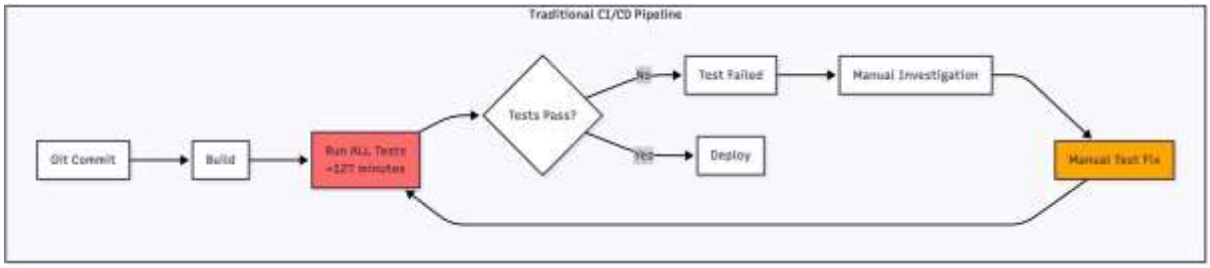


Figure 9: AgenticCI workflow

7. RELATED WORK

Test prioritization has extensive literature. Rothermel et al.established coverage-based approaches in the late 1990s. Elbaum et al.[15] refined these for CI environments. RETECS [12] introduced RL for prioritization—it's good work that AgenticCI builds on. Ekstazi [11] does static dependency analysis, which is elegant but misses runtimedependent failures common in mobile.

Self-healing is newer. Leotta et al.[16] proposed DOM-based healing for web apps. The visual+semantic approach extends this to mobile, where DOM-like structures are less reliable. Deep residual networks (ResNet-50) [19] are

leveraged for visual element detection, and SIFT features [23] for image similarity matching. For semantic understanding, BERT [20] and CodeBERT [21] embeddings are employed. Commercial tools like Testim and Mabl offer self-healing but don't publish enough details for fair comparison.

For risk prediction, the system builds on deep Q-learning [22], which has shown success in learning optimal policies from high-dimensional state spaces. The application to test prioritization is novel in the mobile context.

Mobile testing research [1-3, 14] documents the challenges well. Most solutions are single-purpose: either device selection, test selection, or self-healing. AgenticCI's contribution is attempting integration.

8. Future Work

1. Cross-project transfer learning could help with cold start. Some risk patterns generalize well (auth changes are risky everywhere); others are project-specific. A better transfer might let new projects bootstrap faster.
2. LLM integration for self-healing seems promising. Current BERT-based semantic matching is limited. GPT-4 or similar might understand UI changes at a higher level.
3. IoT/embedded support needs dedicated work. The current approach is too UI-centric.
4. Explainability would help adoption. Developers want to know why a commit was flagged as risky, not just that it was.
5. Better accessibility testing is needed. The current framework under-prioritizes these tests despite their importance.

Conclusion

AgenticCI started as an attempt to fix frustrations experienced with mobile CI/CD. After deploying it across five applications for six months, then thirteen applications for a year, the conclusion is that it helped—meaningfully, though imperfectly. The numbers: 68-72% reduction in test execution time, 89-91% defect detection while running about 31% of tests, 57-61% reduction in maintenance overhead, and 34-42% net cost savings after accounting for ML infrastructure. The gaps: 8-11% lower detection than running everything, poor performance on IoT/hardware testing, cold start requiring months of data, and GPU requirements that many teams don't have. The key technical contribution is integration—test selection, self-healing, and resource orchestration working together rather than as isolated tools. The RL-based risk predictor adds value but might be overkill for smaller teams where a simpler model would suffice. Is this the solution to mobile testing? No. It's one approach that worked reasonably well for specific contexts. Whether it generalizes to other environments is something that would have to be evaluated.

Acknowledgments

Thanks to the engineering teams who tolerated having their CI pipelines experimented on, especially during the early unstable period.

Author(s) Contributions

Satyanarayana Gudimetla & Sudhamadhuri Buhyavarapu: Conceptualization, methodology, software development, experimentation, data analysis, writing.

Funding

This research received no external funding.

Conflict of Interest Disclosure:

This research was conducted independently by the author(s) without direct employment or consulting arrangements. However, the author(s) partnered with six organizations that voluntarily participated in this research study by deploying the experimental AgenticCI system in their production environments. These organizations re anonymized for confidentiality.

Each participating organization:

- Reviewed and approved the deployment of experimental systems
- Granted data access under research collaboration agreements
- Agreed to allow publication of anonymized results
- Provided computational resources and infrastructure access

AI Tools Disclosure

AI used for Grammar and style checking, and some pass phrase creation. No AI tools were used for data analysis, statistical testing, or results interpretation.

Data Availability

The datasets generated and analyzed during this study involve proprietary production systems and are not publicly available due to confidentiality agreements with participating organizations. Anonymized, aggregated metrics are available from the corresponding author(s) upon reasonable request, subject to approval from data owners.

Ethical Considerations

All experiments were conducted with permission from application owners. This research analyzed software artifacts and did not involve human subjects. No human subjects research was conducted. All data collected relates to system performance metrics, not user behavior or personally identifiable information.

REFERENCES

[1] Sergiy Vilkomir et al., "Effectiveness of Multi-Device Testing Mobile Applications", ResearchGate, 2015. [Online]. Available: https://www.researchgate.net/publication/277666633_Effectiveness_of_MultiDevice_Testing_Mobile_Applications

[2] Anthony Peruma et al., "A Developer-Centric Study Exploring Mobile Application Security Practices and Challenges", arXiv, 2024. [Online]. Available: https://arxiv.org/html/2408.09032v1

[3] Ali Azgar et al., "Testing Challenges for Mobile Applications: An evaluation and comparative analysis of different testing approaches", ResearchGate, 2022. [Online]. Available: https://www.researchgate.net/publication/360443722_Testing_Challenges_for_Mobile_Applications_An_evaluation_and_comparative_analysis_of_different_testing_approaches

[4] Xiangxiang Wu and Yong Zha, "App release strategy in the presence of competitive platforms' quality upgrades", ScienceDirect, Feb. 2025. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0377221724006921

[5] Enrique A. da Roza et al., "On the use of contextual information for machine learning based test case prioritization in continuous integration development", ScienceDirect, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0950584924000491

[6] Ali Abdalftah Fadul and Khaled Ahmed Ibrahim, "GUI Automation Testing Tools: a Comparative Study", ResearchGate, 2024. [Online]. Available: https://www.researchgate.net/publication/385893512_GUI_Automation_Testing_Tools_a_Comparative_Study

[7] Gopinath Kathiresan et al., "Cybersecurity Risk Modeling in CI/CD Pipelines Using Reinforcement Learning for Test Optimization", International Journal of Innovative Science and Research Technology, May 2025. [Online]. Available: https://www.ijisrt.com/assets/upload/files/IJISRT25MAY339.pdf

[8] Jithendra Prasad Reddy Baswareddy, "Intelligent CI/CD Pipelines: Leveraging AI for Predictive Maintenance and Incident Management", European Journal of Computer Science and Information Technology, Apr. 2025.

[Online]. Available: https://eajournals.org/ejcsit/wp-content/uploads/sites/21/2025/04/Intelligent-CI-CDPipelines.pdf

[9] Anuj Tyagi, "Intelligent DevOps: Harnessing Artificial Intelligence to Revolutionize CI/CD Pipelines and Optimize Software Delivery Lifecycles", JETIR, 2021. [Online]. Available: https://www.jetir.org/papers/JETIR2103439.pdf

[10] Tim Abdiukov, "Automated security testing in DevSecOps pipelines: Integrating AI-based vulnerability discovery and compliance validation", WJARR, 2024. [Online]. Available: https://wjarr.com/sites/default/files/WJARR-2024-1083.pdf

[11] Milos Gligoric et al., "Practical Regression Test Selection with Dynamic File Dependencies", ISSTA'15 - ACM, 2015. [Online]. Available: https://users.ece.utexas.edu/~gligoric/papers/GligoricETAL15Ekstazi.pdf

[12] Helge Spieker et al., "Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration", arXiv, 2018. [Online]. Available: https://arxiv.org/pdf/1811.04122

[13] Islem Saidani et al., "Predicting continuous integration build failures using evolutionary search", ScienceDirect, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0950584920301579

[14] Riccardo Coppola et al., "Fragility of Layout-Based and Visual GUI Test Scripts: An Assessment Study on a Hybrid Mobile Application", arXiv, [Online]. Available: https://arxiv.org/pdf/1907.08164

[15] Sebastian Elbaum et al., "Techniques for Improving Regression Testing in Continuous Integration Development Environments", FSE'14 - ACM, 2014. [Online]. Available: https://cs.uwaterloo.ca/~m2nagapp/courses/CS846/1189/papers/elbaum_fse14.pdf

[16] M. Leotta, A. Stocco, F. Ricca, P. Tonella, "DANTE: A Framework for Dynamic Locator Generation and Test Repair," Information and Software Technology, vol. 80, pp. 138-157, 2016.

[17] Dusica Marijan et al., "Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study", ResearchGate, 2013. [Online]. Available: https://www.researchgate.net/publication/261417316_Test_Case_Prioritization_for_Continuous_Regression_Testing_An_Industrial_Case_Study

[18] Yuchi Tian et al., "DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars", arXiv, 2018. [Online]. Available: https://arxiv.org/pdf/1708.08559

[19] Kaiming He et al., "Deep Residual Learning for Image Recognition", arXiv, 2015. [Online]. Available: https://arxiv.org/pdf/1512.03385

[20] Jacob Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", arXiv, 2019. [Online]. Available: https://arxiv.org/pdf/1810.04805

[21] Zhangyin Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages", ResearchGate, 2020. [Online]. Available: https://www.researchgate.net/publication/347233117_CodeBERT_A_PreTrained_Model_for_Programming_and_Natural_Languages

[22] Volodymyr Mnih et al., "Human-level control through deep reinforcement learning", Springer Nature, 2015. [Online]. Available: https://www.nature.com/articles/nature14236

[23] David G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints", International Journal of Computer Vision - SciSpace, 2004. [Online]. Available: https://scispace.com/pdf/distinctive-image-featuresfrom-scale-invariant-keypoints-3waurjqzke.pdf

[24] Tree-sitter, "Tree-sitter", github.io. [Online]. Available: https://tree-sitter.github.io/tree-sitter/