# Context-Aware Predictive Self-Healing Test Automation Using Failure-Causality Analysis for IoT-Enabled Cloud Robotic Systems

Punam Kailas Kale[1], Vaishnavi Jalindar Khalate[2], Pratiksha Ramkisan Jadhav[3], Prof. Pallavi Gholap[4]
[1,2,3,4]*Science and Computer Science Department, MIT Arts, Commerce and Science College, Alandi (D), Pune, Maharashtra, India*
doi.org/10.64643/ijirt.209029-459

***Abstract*—Robots, sensors, edge devices, machine-learning components, communication protocols, APIs, and cloud back-ends all sit inside a modern IoT-enabled cloud robotic system, and that mix is exactly what makes automated testing harder here than for an ordinary web or desktop application. When a test fails, the reason is often nothing to do with a code defect at all a dropped packet, a slow network hop, a jittery sensor reading, a broken MQTT session, a slow API response, or a cloud service that is briefly unreachable can all produce the same red mark on a test report. Treating every one of those red marks as a confirmed bug leads to false alarms, wasted reruns, sluggish CI/CD pipelines, and hours of manual triage that didn't need to happen. To deal with this, the paper puts forward CAPS-TA short for Context-Aware Predictive Self-Healing Test Automation a framework built around one idea: look at the circumstances surrounding a failure before deciding what to do about it. It pulls in live signals from Raspberry Pi or Arduino hardware, IoT communication channels, APIs, cloud endpoints, test logs, and the outcomes of past runs, then runs a lightweight causal model over that evidence to judge what probably caused the failure, whether the problem is likely to pass on its own, and which fix makes sense. From there, a decision layer picks among several options retrying under controlled conditions, re-establishing the communication link, shifting execution to the edge, waiting and rerunning later, or simply flagging a human rather than defaulting to one fixed behaviour. What sets this apart from a plain retry-until-it-passes strategy is that the diagnosis is linked directly to how much the next action would cost and whether it's actually appropriate. The remainder of the paper walks through the architecture, how information flows through it, the recovery algorithm, metrics for judging performance, a plan for injecting faults on purpose, and a CI/CD experiment design meant to be repeatable by others. Because none of this has been run yet, the paper stops short of reporting experimental numbers it stays at the level of a proposed, testable design.***

***Index Terms*—Self-Healing Testing; IoT; Cloud Robotics; Failure Causality; Predictive Testing; Raspberry Pi; Arduino; AI; CI/CD; Fault Injection; Edge Computing; Automated Software Testing.***

## I. INTRODUCTION

Modern robotics rarely runs as a single, self-contained program anymore. A working robotic application typically depends on a chain of connected pieces sensors feeding data, edge hardware doing local processing, middleware moving messages around, cloud services handling heavier computation, and a handful of software interfaces tying it all together. A robot streams telemetry through MQTT; an edge unit cleans up the raw sensor feed before anything else touches it; a REST API relays commands up to the cloud; a dashboard shows whatever the cloud sends back. With that many links in the chain, a single test failure could trace back to almost any one of them.

That's the practical difficulty this work is built around. A dropped packet, network lag, noisy sensor input, a stalled API call, a broken MQTT session, or a cloud hiccup any of these can flip a test to "failed" while the application code underneath stays perfectly correct. CAPS-TA is designed around exactly this distinction. Instead of treating a red test result as an automatic verdict of "defect," it first checks what was actually happening in the system when the failure occurred, and looks back at how similar situations played out before.

This isn't a new direction in isolation self-healing research has spent years on detecting abnormal system states and triggering recovery automatically. Some of that literature frames self-healing purely as behaviour-watching plus fault-adapting; more recent work leans

harder on feedback loops, learned models, and recovery with minimal human input. A parallel thread in software testing, flaky-test research, has already shown that simply rerunning a failing test helps, but doing so without any judgement gets expensive fast which is the argument for pairing reruns with some kind of prediction.

Narrowing that down to the actual gap this paper targets: a test-automation system has to make a call retry, reconnect, move to the edge, wait, or hand off to a person and making that call well requires linking together the failure's probable cause, the live context, how likely the problem is to resolve on its own, and what each option would cost to attempt. CAPS-TA is the proposed answer to that specific decision problem.

## II. PROBLEM STATEMENT

Multiple interacting components sit underneath any automated test written for an IoT-enabled cloud robotic system, so a single failed assertion rarely has one obvious source. It could follow a network-quality dip, an MQTT session dropping and reconnecting, sensor data arriving late, an API call timing out, or the cloud simply responding slowly. None of that is distinguishable under a fixed retry policy reruns might smooth over the transient cases, but they also burn execution time and, worse, can bury defects that are real and persistent under a pile of "passed on retry" results.

Framed as a design problem, this comes down to five things a context-aware test-automation approach needs to do: gather the runtime and historical evidence that actually matters, work out what most likely caused the failure, judge whether that cause is temporary or here to stay, pick a recovery step that fits the situation rather than a default one, and keep enough of a trail behind for a person to dig into later if needed.

## III. OBJECTIVES

Turning that problem statement into concrete goals, the project sets out to build several things at once. On the data side, it needs a collection layer wide enough to pull in signals from the robot, the edge hardware, IoT channels, APIs, the cloud, and the test-execution process itself none of these alone gives the full picture. On the reasoning side, a lightweight model has to connect whatever symptoms show up to the failure categories they most plausibly belong to, alongside a separate estimate of whether a given failure is likely to pass on its own before anyone commits to an expensive recovery step.

On the decision side, recovery-action selection should weigh predicted cause together with cost, rather than falling back on a flat number of retries regardless of context. Practically, the framework also needs a route into existing CI/CD pipelines and the automation stacks teams already use PyTest, Robot Framework, Selenium so it isn't a standalone tool bolted onto nothing. Finally, none of the above means much without a repeatable fault-injection setup that lets any claimed improvement actually be checked rather than taken on faith.

## IV. REVIEW OF RELATED WORK

The conceptual groundwork for automatic fault detection and recovery comes largely from self-healing-systems research. Ghosh and colleagues frame self-healing systems around using models and live runtime information to bring a system back to normal operation [1], and Schneider, Barker, and Dobson build on that in their survey of self-healing frameworks, with particular emphasis on autonomous detection and recovery inside complex computing environments [2].

A separate strand self-adaptive systems points out a real limitation: reacting only once a condition has already changed leaves a system perpetually one step behind. That observation is part of why runtime-adaptation research has moved toward decisions that are both context-aware and predictive, pulling on current state as well as history [4], [8].

Software testing has its own parallel thread in flaky-test detection, where a test can flip between pass and fail with no actual code change behind it, disrupting CI pipelines and eating into developer time. One recent empirical study pairs test rerun with machine-learning models specifically to bring detection cost down without giving up much accuracy [5].

Machine learning also shows up in existing self-healing test-automation work aimed at spotting and repairing broken tests. Where CAPS-TA departs from that body of work is scope: it's built for distributed IoT-cloud robotic test environments specifically, and it pairs failure causality with environmental context to weigh several recovery options against each other a

claim that still needs to be checked experimentally rather than taken on faith.

| Research direction | Main idea | Limitation for IoT-cloud robotics | CAPS-TA response |
|---|---|---|---|
| Self-healing systems | Watches for faults, diagnoses them, recovers automatically | Usually built for general-purpose use | Adds recovery decisions tailored to testing |
| Runtime adaptation | Changes behaviour based on live context and system state | Often only reacts after the fact | Predicts ahead of time, before acting |
| Flaky-test detection | Spots test outcomes that aren't stable | Tends to lean on reruns or past test history alone | Blends that history with live runtime context |
| AI-based test healing | Fixes or adjusts automated test scripts | Usually centred on UI or script-level issues | Aimed at distributed IoT/robot failure sources |

## V. PROPOSED CAPS-TA FRAMEWORK

Structurally, CAPS-TA is a closed loop rather than a one-shot check (Figure 1).

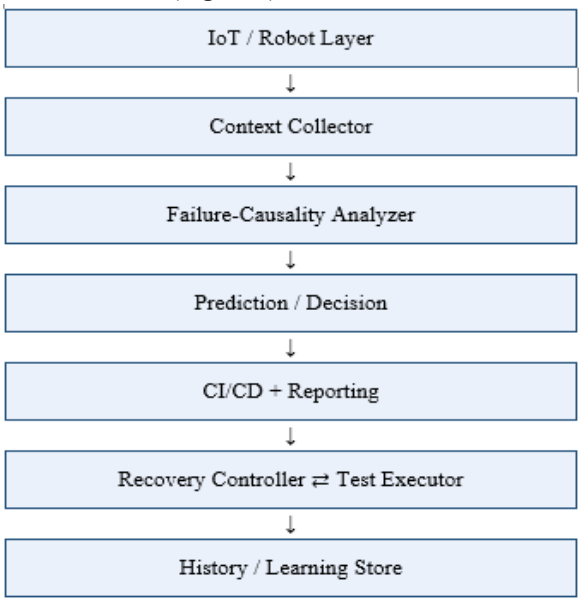


*Fig. 1 CAPS-TA architecture and feedback flow*

While a test executor runs whichever test case is up next, a context collector is quietly logging signals from the robot or edge device, the communication layer, the API layer, the cloud service, and the test environment itself. Should the test fail, a failure-causality analyzer steps in, cross-referencing the failure against everything the collector just logged as well as what happened in prior runs. That feeds a prediction-and-decision module, which picks a recovery action and whatever happens next gets written back into storage, so the next decision has real history to draw on instead of starting from zero.

*The Recovery Controller feeds back into the Test Executor to rerun the required scope, and outcomes from the History / Learning Store loop back into later Recovery Controller and Prediction/Decision cycles, closing the loop.*

## VI. CONTEXT COLLECTION AND FAILURE REPRESENTATION

A CAPS-TA failure record isn't a bare error string it's captured as a structured context vector instead. That means logging things like which test ran, when, the state of the device, current network latency, an estimate of packet loss, whether MQTT was connected, how fresh the sensor data was, how long the API took to respond, whether the cloud service was reachable, what recent runs looked like, and the specific exception or failed assertion.

$C = \{T, D, N, M, S, A, K, H, E\}$

Here T stands for test information, D for device state, N for network indicators, M for messaging state, S for sensor state, A for API/cloud indicators, K for execution context, H for historical evidence, and E for the observed error itself.

Nothing here demands a full digital-twin simulation the whole point of keeping it this lightweight is that it can be captured during an ordinary CI run without extra infrastructure.

## VII. FAILURE-CAUSALITY ANALYSIS

Rather than pinning a single label on a failure, the causality layer scores it against several operational categories at once communication failure, sensor-data instability, API/service timeout, a cloud-availability problem, an environment or resource issue, and a probable application defect. None of these scores is

treated as a confirmed diagnosis; they're just weighted evidence.

$S(f) = \Sigma w_i \times e_i$

$e_i$ is the normalized evidence for a given indicator and $w_i$ is how much weight that indicator carries. A run of MQTT disconnects right before a timeout, for example, would push up the communication-failure score, whereas a deterministic assertion failure occurring alongside otherwise-stable infrastructure would push up the probable-defect score instead. Nothing stops this from starting out as simple rules and later graduating to a trained classifier once there's enough labelled data to work with.

Transience gets its own separate score, deliberately kept apart from the cause score because knowing what caused a failure doesn't automatically tell you whether it will resolve itself. A flaky network condition might clear up on its own; a broken network configuration, by contrast, might just keep failing no matter how many times it's retried, and would need escalation instead.

## VIII. PREDICTIVE RECOVERY DECISION

There's no single retry count baked into CAPS-TA. Instead, it weighs a set of candidate actions against each other a controlled retry, a reconnection attempt, a delayed rerun, shifting execution to the edge, or escalating to a person where each option carries its own estimated odds of working and its own cost to attempt. Whichever option scores best given the current context is the one the decision layer picks.

$U(a) = P(success \mid C, a) - \lambda \cdot Cost(a) - \mu \cdot Risk(a)$

a is the candidate recovery action, C is the context at hand, and $\lambda$ and $\mu$ are tunable penalty weights for cost and risk respectively. Framed this way, recovery becomes an actual decision among competing options grounded in evidence and cost rather than a reflexive "try it again and see."

Algorithm 1: CAPS-TA Failure Recovery

1. Execute test and log runtime context.
2. If pass, archive context and stop.
3. If fail, capture error, logs, timestamps, and component states.
4. Analyze cause, transience, and recovery suitability.
5. Select the highest-utility safe recovery action within limits.
6. Execute recovery and rerun only the affected test portion.
7. If successful, log healing and stop.
8. If failed, update evidence and retry.
9. If thresholds are exceeded, escalate to a human.

## IX. EXPERIMENTAL METHODOLOGY

The evaluation plan calls for a small, reproducible testbed rather than a full production deployment: a Raspberry Pi and/or Arduino board, an MQTT broker, some REST APIs, a cloud service (or a stand-in for one), and a test layer running on PyTest or Robot Framework, with Selenium folded in if a web dashboard is part of the picture. Wrapping all of that in a CI/CD pipeline is what keeps failure logging consistent from run to run.

On top of that testbed, fault injection would deliberately introduce packet loss, extra latency, MQTT disconnections, stale sensor readings, API timeouts, brief cloud outages, and planted application bugs each scenario run more than once under similar conditions. The point isn't to assume one fault always maps neatly to one cause; it's to see whether adding contextual evidence genuinely changes the quality of the recovery decision.

For that comparison to mean anything, there needs to be a baseline: at minimum a conventional fixed-retry strategy, and ideally a context-free classification approach as well. Running CAPS-TA through the identical test cases and fault scenarios as those baselines is what would actually show whether the added context and causal reasoning are worth the extra complexity.

## X. EVALUATION METRICS

| Metric | Definition | Purpose |
|---|---|---|
| False-positive rate | How often a transient or environmental failure gets wrongly labelled a product defect | Diagnostic quality |
| Recovery success rate | Share of failures the chosen action actually recovered from | Self-healing effectiveness |
| Mean recovery time | Time elapsed between detecting a failure and either a successful continuation or escalation | Operational efficiency |

| | | |
|---|---|---|
| Unnecessary rerun rate | Reruns that ended up contributing nothing to recovery | Cost of retry behaviour |
| Causal classification accuracy | How well predicted causes line up with the injected or validated ones | Failure-causality quality |
| Escalation precision | Portion of escalated cases that genuinely needed a human | Decision quality |
| CI/CD disruption | Pipeline delay or blocked runs traceable to failures | Practical impact |

## XI. EXPECTED ANALYSIS AND DISCUSSION

Three hypotheses sit behind the evaluation plan. The first is that causal analysis grounded in context lowers false-positive defect calls compared to blind fixed-retry behaviour. The second is that basing action selection on predicted transience and cost cuts down on wasted reruns. The third is that holding on to execution history sharpens decisions when the same kind of failure keeps recurring. None of these have results attached yet they're framed as hypotheses precisely because they haven't been tested.

Reporting one blended average across all fault types would hide more than it reveals, since a strategy that handles short network blips well might still struggle with sensor faults. Breaking results out by fault category with confusion matrices for the causal categories, recovery-success numbers, and how recovery time is distributed gives a far more honest picture than the aggregate alone.

Wrong calls matter too, and the study needs to track them. A self-healing system that keeps retrying an actual software bug, or that quietly overwrites evidence a human investigator would have needed, isn't harmless just because it's automated. That's precisely why safety limits, retry budgets, evidence retention, and escalation thresholds aren't nice-to-haves here they're load-bearing parts of the design.

## XII. NOVELTY AND RESEARCH CONTRIBUTION

What CAPS-TA actually contributes is the link it draws between failure causality and the specific recovery action chosen, inside a distributed IoT-cloud robotic context. A typical retry mechanism only ever answers one question run it again or not? CAPS-TA pushes that further, into what probably caused the failure, whether it's likely temporary, which recovery option fits, and what that option would cost to run.

Its second contribution is treating context as something that spans layers rather than living in one log file: robot state, sensor freshness, MQTT connectivity, API behaviour, cloud indicators, and past test outcomes all get weighed together. The third is a fault-injection design built to be reproducible, so self-healing decisions can be tested under conditions someone else could set up and check, instead of resting on a handful of anecdotes.

## XIII. LIMITATIONS

Nothing here has been measured yet, so no claim is made about accuracy, recovery rate, or time saved this is a design proposal, not a completed study. Any predictive model built on top of it would need a reasonably representative set of labelled failures to train on, and thin or lopsided data would drag classification quality down. Collecting all this context isn't free either it adds runtime overhead and opens up privacy and security questions that would need addressing. And a recovery action considered safe on one robot or cloud setup might be a bad idea on another, so any real deployment would need its own policies and limits rather than a one-size-fits-all rule.

## XIV. CONCLUSION

Testing gets genuinely harder once software behaviour is entangled with communication, sensing, edge computing, APIs, and cloud services which is exactly the situation IoT-enabled cloud robotic systems create. That's the reasoning behind treating a failed test as something to interpret in context first, rather than flag as a defect on sight. CAPS-TA is built around that idea, combining context collection, causal analysis, transience prediction, and cost-aware recovery selection into one framework.

What's delivered here is an architecture that could actually be built, a decision model, an algorithm, and a plan for controlled evaluation deliberately stopping short of claiming results that don't exist yet. The logical next step is building this on a real Raspberry Pi/Arduino-MQTT-cloud testbed, injecting faults on

purpose, capturing execution traces, and running CAPS-TA head-to-head against fixed-retry and context-free baselines. Only that kind of empirical run would settle whether the approach genuinely cuts false positives and makes testing more reliable in real IoT-cloud robotic deployments.

## XV. THEORETICAL FOUNDATION OF CAPS-TA

Underlying CAPS-TA is a fairly simple premise: you can't judge a failed test purely by the failure itself you also need to know what the system was doing at that moment. Runtime context, causal analysis of the failure, the history of past runs, and the cost of recovering all feed into that judgement together. Laid out as a chain, the reasoning goes: look at the system's context → work out what probably caused this → judge whether it's likely temporary → pick a recovery action that fits → log the outcome so the next decision benefits from it. A plain fixed-retry approach skips every one of those steps and just repeats the same action regardless of why the test failed which is precisely the gap this chain is meant to close.

### 15.1 Theory of Context-Aware Testing

The starting assumption here is that a software event doesn't mean the same thing in every situation its meaning depends on what else was going on in the environment at the time. Two identical test errors in a distributed IoT-cloud robotic system could have completely different roots depending on network conditions, sensor behaviour, device state, or cloud status at that moment. So, reading an error message in isolation isn't enough; it needs to sit alongside variables like network latency, packet loss, device availability, sensor freshness, MQTT connection status, API response time, cloud health, and what happened in earlier runs. Within CAPS-TA, all of that context becomes evidence for telling a real application bug apart from a failure that's really just infrastructure being temporarily unreliable.

### 15.2 Theory of Self-Healing Software

The job of self-healing software, broadly, is to notice when something's off, work out a probable cause, and act on it with as little human input as possible. That usually plays out across five stages watching system behaviour, catching the abnormal event, diagnosing why it happened, planning a response, carrying that response out, and finally feeding the outcome back in for next time. CAPS-TA takes that same loop and points it at test automation specifically the target isn't just keeping the application environment healthy, but keeping the testing process itself moving, by picking whatever action makes sense right after a failure.

### 15.3 Theory of Failure Causality

What failure causality is really asking is: given what we're seeing, what actually produced it? In a distributed setup, the two can be far apart a test might report an API timeout, for instance, when the real culprit was packet loss between an edge device and an MQTT broker several steps upstream. Untangling that requires looking at timing, the state different components were in, which events lined up together, and what's happened historically. CAPS-TA keeps this lightweight through a scoring approach, where evidence pulled from different layers builds up support for candidate causes rather than pinning down one definitive root cause. Getting the ranking of plausible causes right is the actual goal not certainty, just enough confidence to make a safer call about recovery.

### 15.4 Theory of Predictive Test Recovery

Where this differs from purely reactive testing is timing: instead of waiting to see what happens, it tries to estimate the likely outcome before committing to an action. That only pays off when the expected benefit of a recovery step is actually bigger than its cost and risk an immediate retry is probably fine after a brief network hiccup, but retrying a deterministic assertion failure over and over accomplishes almost nothing. CAPS-TA weighs both transience and how well-suited each recovery option is before acting, and gets sharper at that estimate over time as historical patterns for the same kind of failure accumulate.

### 15.5 Theory of Flaky and Transient Failures

Flaky and transient failures are worth treating as their own category because a test can start failing intermittently with zero change to the underlying application logic timing quirks, asynchronous communication, resource contention, environmental instability, or an external service acting up can all be behind it. Just rerunning blindly does cut down on some false alarms, but it never actually explains why the failure happened in the first place. CAPS-TA takes that rerun instinct and adds judgement to it using

contextual evidence to decide whether a rerun is actually warranted, or whether something else, like reconnecting a channel or simply waiting before trying again, is the better move.

15.6 Theory of Edge-Cloud Cooperation

Cloud robotics doesn't put all its computation in one place it's split between the robot itself, edge hardware, and the cloud. The cloud brings scale and centralised services; the edge responds faster locally and can keep functioning through short cloud or network outages. That split creates a genuine option for test recovery: if a cloud-dependent step is briefly unreachable, routing through a compatible edge path can stand in as a controlled fallback. CAPS-TA treats edge execution as just one item on the menu of recovery options, rather than assuming every failed operation has to loop back through the cloud.

15.7 Theory of Cost-Aware Recovery

Recovery actions aren't interchangeable in terms of cost. A retry might cost a few seconds; reconnecting an MQTT session takes real setup time; a delayed rerun stretches out the whole pipeline; and escalating to a human can eat up meaningful engineering hours. Any recovery strategy worth using has to weigh probability of success against that cost and risk together, not separately. CAPS-TA builds this directly into a utility function, giving it a principled basis for choosing actions on the fly instead of falling back on a fixed retry count.

## XVI. MATHEMATICAL MODEL OF CAPS-TA

16.1 Context Vector

Putting the runtime context of a failed test into a single expression gives:

$C = \{T, D, N, M, S, A, H, E\}$

T here is test information, D device state, N network information, M messaging state, S sensor information, A API/cloud information, H historical execution information, and E the observed error. Before any of this reaches the causal-analysis layer, each element gets normalised first.

16.2 Causal Score

Scoring a candidate cause f against the current context looks like:

$S(f|C) = \Sigma\, w_i \cdot e_i$

with $e_i$ as the normalised evidence for indicator i and $w_i$ as the weight assigned to it. The higher this score climbs, the stronger the case the current context makes for that particular cause. Weighting can begin as something hand-set from domain knowledge, then shift toward values learned from labelled fault-injection data as that data accumulates.

16.3 Transience Probability

Estimating how likely a failure is to be transient takes the form:

$P_{transient} = g(C, H)$

g stands in for either a rule-based function or a trained model, working off the current context C and historical evidence H. A high output means this failure looks like conditions that resolved themselves before; a low one means repeating the test probably won't accomplish much.

16.4 Recovery Utility

For any given recovery action, a, the utility CAPS-TA assigns it is:

$U(a|C) = P(success|C,a) - \lambda \cdot Cost(a) - \mu \cdot Risk(a)$

Whichever action clears the acceptable-utility bar by the widest margin gets chosen, though always within safety limits, retry budgets, and whatever escalation policy applies. Put together, this equation is really what ties diagnosis, prediction, and the final action-selection step into one coherent decision.

## XVII. THEORETICAL RESEARCH HYPOTHESES

H1: Analysing causes with full context in view should cut down on transient failures being wrongly flagged as defects, compared with a strategy that just retries blindly.

H2: Choosing recovery actions based on predicted transience and cost, rather than habit, should bring unnecessary reruns and overall recovery time down.

H3: Keeping a record of past executions should sharpen how well recurring failure patterns get classified and which recovery action gets picked for them.

H4: Pulling context from across IoT, edge, API, cloud, and test-execution layers together should give more

useful recovery evidence than relying on test logs by themselves.

Underneath all of this is a feedback loop: whatever gets learned during one execution doesn't just disappear it's stored and fed into the executions that follow. That's what ties failure analysis to recovery and ongoing learning in CAPS-TA, instead of letting each test failure sit as a one-off, disconnected event.

## REFERENCES


[1] D. Ghosh, R. Sharman, H. R. Rao, and S. Upadhyaya, "Self-healing systems survey and synthesis," *Decision Support Systems*, vol. 42, no. 4, pp. 2164–2185, 2007, doi: 10.1016/j.dss.2006.06.011.

[2] C. Schneider, A. Barker, and S. Dobson, "A survey of self-healing systems frameworks," *Software: Practice and Experience*, vol. 45, pp. 1375–1398, 2015, doi: 10.1002/spe.2250.

[3] H. Chinni, Y. K. Sharma, S. Shevkari, J. L. Vydehi, Saurabh, and M. Kavitha, "Comparative evaluation of custom CNN architectures and transfer learning models for image classification in PyTorch," in *2026 13th International Conference on Computing for Sustainable Global Development (INDIACom)*, New Delhi, India, 2026, pp. 1–6, doi: 10.23919/INDIACom70271.2026.11525435.

[4] F. D. Macías-Escrivá, R. Haber, R. del Toro, and V. Hernández, "Self-adaptive systems: A survey of current approaches, research challenges and applications," *Expert Systems with Applications*, vol. 40, no. 18, pp. 7267–7279, 2013, doi: 10.1016/j.eswa.2013.07.033.

[5] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Empirically evaluating flaky test detection techniques combining test case rerunning and machine learning models," *Empirical Software Engineering*, vol. 28, 2023, Art. no. 72, doi: 10.1007/s10664-023-10307-w.

[6] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Evaluating features for machine learning detection of order- and non-order-dependent flaky tests," in *Proc. IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2022, pp. 93–104.

[7] Y. Aruna, M. Kavitha, M. Patil, S. Shevkari, A. Aggarwal, and Y. K. Sharma, "Deep learning based clinical decision-support web framework for heart disease prediction," in *2026 13th International Conference on Computing for Sustainable Global Development (INDIACom)*, New Delhi, India, 2026, pp. 1–6, doi: 10.23919/INDIACom70271.2026.11526620.

[8] M. W. Mutanu *et al.*, "State of runtime adaptation in service-oriented systems: What, where, when, how and right," *IET Software*, 2019, doi: 10.1049/iet-sen.2018.5028.

[9] "Evaluation of self-healing systems: An analysis of the state-of-the-art and required improvements," *Computers*, vol. 9, no. 1, 2020, Art. no. 16.