10.48047/jocaaa.2025.34.11.01

Serverless Event-Driven Architecture for Enterprise Test Automation

Vamsi Krishna Gattupalli
Independent Researcher, USA

Abstract

Test automation enterprise architectures are confronted with severe challenges in realizing scalability, cost-effectiveness, and operational robustness in certifying distributed cloud platforms using conventional infrastructure-based frameworks. Paradigms of serverless computing integrated with event-driven orchestration patterns radically shift quality assurance capacities by obliterating the need for permanent infrastructure and allowing elastic scalability through ephemeral compute functions. The suggested architecture utilizes message queuing systems, notification services, document databases, and object storage to produce loosely coupled validation environments where decomposition of test suites into independent shards supports enormous parallelism over hundreds of concurrent run contexts. Smart defect classification using machine learning algorithms inspects failure artifacts such as execution logs, stack traces, and interface screenshots to semi-automatically classify issues without human-level triage, while structural similarity algorithms identify visual regressions by perceptual evaluation methods modeled after human visual perception traits. Self-healing orchestration techniques apply smart retry approaches, selective re-execution processes, and quarantine automation methods that remove flaky tests with intermittent failures. Predictive capacity management continuously monitors queue metrics and processing velocity to dynamically adjust computational resources, maintaining optimal performance without manual scaling interventions. Deep learning models trained on historical defect repositories accelerate root cause identification by correlating test failures with recent code modifications and deployment events. The cloud-native architecture sets up autonomous quality governance systems that offer instant validation feedback, reduce infrastructure expenses using pay-per-execution billing schemes, and eliminate human operational overhead in enterprise continuous delivery pipelines.

Keywords: Serverless Test Automation, Event-Driven Architecture, Test Sharding, Intelligent Defect Classification, Self-Healing Orchestration, Predictive Resource Scaling

Introduction

Today's businesses running distributed cloud platforms are confronted with mounting complexity in carrying out end-to-end automated validation at scale, with test suites consistently running into thousands of discrete test cases that have to run across multiple environments, browser setups, and deployment destinations. Legacy test automation tools based on rigid infrastructure and static models of orchestration introduce major bottlenecks to continuous delivery pipelines by consuming dedicated physical or virtual machines that sit idle between non-testing times, taking up computational resources, and generating operational expenses. These traditional systems depend on long-running compute resources like specialized test servers or containerized grid infrastructures that need to be manually provisioned, capacity planned, and otherwise maintained to manage spike testing volumes. Linear execution styles built into legacy frameworks compel test suites to execute in pre-determined batches or restricted parallel threads, proscribing throughput even where further computational capacity is available elsewhere inside the infrastructure.

10.48047/jocaaa.2025.34.11.01

Software structure evolution has seen a vast move closer to distributed, event-driven paradigms that inherently dismantle monolithic models. Architectural trends surfacing from practitioner exercise show microservices-based structures leading the way in enterprise software program design, with event-driven architectures becoming increasingly popular as companies pursue superior system responsiveness and component decoupling. This architectural shift carries over from application development into operational spaces such as quality assurance and test automation, where the shortcomings of centralized, synchronous validation frameworks become more apparent. The use of serverless computing models is a natural evolution in this developmental path, where organizations can break down sophisticated testing processes into self-sustaining, event-reactive units that run independently without constant infrastructure administration. These design patterns prioritize elasticity, fault isolation, and operating efficiency through function-as-a-service paradigms that map computational resource utilization directly with real workload requirements [1].

Serverless computing designs bypass infrastructure management overhead by encapsulating server provisioning, scaling choices, and capacity planning as platform-managed services that allocate computational resources dynamically as per arriving event triggers. The serverless model works on basic principles of stateless execution, event-driven invocations, and auto-scaling, which are especially well-suited for workloads with fluctuating demand patterns and ephemeral computational tasks. In geospatial computing applications, serverless platforms have exhibited the ability to process large datasets with parallel function calls, where each processing unit independently processes individual data partitions before aggregating results via coordinated workflows. This computational model applies directly to test automation situations involving large test suites that must be broken down into separate validation segments executed concurrently over distributed compute resources. Pay-per-execution economics built into serverless environments completely disrupts cost paradigms by removing idle infrastructure capacity fees, only paying for function execution time in millisecond units. Quality engineering organizations are able to take advantage of these architectural attributes to convert test automation from bounded capacity systems, where continuous operations monitoring is needed, to elastic validation platforms that self-scale to fluctuating test loads while reducing both infrastructure cost and human intervention needs [2].

Architectural Foundations of Serverless Test Systems

Core Component Integration

A serverless test automation architecture gets rid of persistent infrastructure via utilizing ephemeral compute features that execute on-demand, with every feature instance existing only for the duration required to finish its certain validation assignment before terminating and liberating all allocated assets. The system incorporates several integrated components working in concert through well-defined interfaces and communication protocols. Compute functions are the main execution engines, executing individual tests by loading test scripts, communicating with application endpoints, capturing validation outcomes, and creating execution artifacts in isolated runtime environments offering consistent execution contexts irrespective of underlying physical infrastructure. Message queues provide asynchronous job distribution models that detach test orchestration from execution, enabling orchestrator elements to quickly add thousands of test execution requests to an enqueued state without waiting for processing completion, and worker functions consuming such requests in numbers depending on computational availability and set concurrency thresholds. The union of message queue metrics with predictive

10.48047/jocaaa.2025.34.11.01

autoscaling capabilities supports advanced capacity management in which machine learning algorithms model past queue depth trends, message arrival rates, and processing rates to predict future computational loads and adjust concurrency limits ahead of time before queue backlogs form. These forecasting models analyze temporal patterns in testing execution workflows, recognizing repeated daily or weekly cycles of testing that correspond to continuous integration pipeline activity, and pre-scale function concurrency to support expected workload spikes instead of scaling reactively after performance deterioration has set in. Application of time-series forecasting methodologies to queue statistics enables the system to separate transient traffic bursts that demand short-term capacity boosts from persistent workload increases that demand concurrency adjustments over longer periods, optimizing both response latency and cost efficiency of computation [3].

Document stores keep important system metadata such as test configuration options, execution history logs, defect routing rules, and aggregated result summaries in adaptive schema configurations that can handle changing data needs without needing database migrations or schema changes. Object storage systems retain execution artifacts like comprehensive test logs, exception stack traces, network traffic captures, and visual evidence like screenshots or video recordings of test sessions, storing these artifacts in hierarchical namespaces that support efficient retrieval during failure analysis and debugging investigations. This design builds a loosely coupled system in which every element works independently based on its specific responsibilities, only communicating through asynchronous events and data persistence mechanisms instead of synchronous dependencies that would lead to blocking operations and decrease total system throughput [3].

Event-Driven Design Principles

The event-driven approach reworks conventional request-response interactions into reactive processes where system elements react independently to state changes spread via event notifications instead of performing pre-defined sequential routines. Smart orchestration platforms route incoming events through rule-based routing logic and state machine specifications that synchronize intricate multi-step processes comprising conditional branching, concurrent execution branches, and error handling processes. Event routing infrastructure assesses incoming messages against pattern match rules that analyze event attributes such as source identifiers, event types, and payload content in order to identify suitable downstream subscribers and processing actions. Workflow state is kept by the orchestration layer across distributed execution steps, monitoring completion status of parallel tasks, aggregating results from multiple concurrent operations, and enforcing advanced retry logic with exponential backoff strategies for temporary failures. Real-time processing pipelines take advantage of these orchestration features to host smart business logic that reacts to system events within milliseconds, allowing automated decision-making flows that examine event patterns, correlate similar events across temporal windows, and invoke suitable actions based on threshold conditions configured or pattern-based anomalies identified [4].

When continuous integration infrastructures catch new code commits and trigger build operations, pipeline completion events cause orchestration functions to be called that receive the build artifacts, inspect the test suite structure, and break down the entire validation inventory into individual execution units by employing test categorization, previous execution duration information, and specified parallelism parameters. As each test segment finishes its validation process, worker functions fire structured completion events with execution status, duration statistics, assertion outcomes, and artifact pointers to the event routing infrastructure that notifies several interested parties, such as result aggregation services, metric gathering systems, and real-time monitoring dashboards. The orchestration architecture coordinates these distributed workflows by using state machine definitions that define task dependencies,

10.48047/jocaaa.2025.34.11.01

timeouts, and failure recovery strategies, allowing reliable progression of execution even in the event of sporadic failures or performance decline by individual components. Validation failures are detected, producing defect events that contain contextual data, which are then processed by intelligent routing algorithms to decide suitable notification channels, ownership based on defined business rules, and automated triage workflows that trigger within seconds of failure detection [4].

Component	Primary Function	Key Characteristics	Operational Benefits
Compute Functions	Execute individual test shards	Ephemeral instances, stateless execution, automatic scaling	Eliminate idle costs, enable massive parallelism
Message Queues	Decouple test distribution from processing	Asynchronous distribution, predictive autoscaling	Rapid enqueuing prevents blocking operations
Notification Services	Broadcast system events	Pub-sub messaging, real-time propagation	Simultaneous delivery to multiple subscribers
Document Databases	Maintain metadata and execution history	Flexible schemas, configuration storage	No schema migrations, efficient queries
Object Storage	Preserve execution artifacts	Hierarchical organization, logs, screenshots	Efficient retrieval for debugging

Table 1. Architectural components enabling distributed validation workflows [3, 4].

Test Sharding and Distributed Execution

Test sharding is the inherent technique allowing enormous parallelism in serverless architectures, breaking down monolithic test suite execution into highly distributed validation processes that take advantage of the elastic scaling traits inherent in function-as-a-service architectures. Instead of running tests one by one or in a few parallel threads limited by static infrastructure capacity, the entire validation suite is subjected to systematic partitioning into many independent subsets referred to as shards, each being a logically partitioned set of test cases partitioned based on execution traits, functional domains, or resource demands. The sharding mechanism is based on smart partitioning algorithms that examine test suite metadata such as past execution times, resource utilization patterns, and interdependency relationships in order to generate shard distributions balanced for optimal overall execution time while adhering to computational limits like function timeout quotas and memory allocations. Parallel processing architectures utilize spatial partitioning techniques by means of uniform grid structures that divide computational workspaces into regular cells, allowing parallel workers to process discrete regions independently without global coordination or synchronization barriers. These grid-based partitioning strategies structure data into hierarchical forms in which coarse-grained partitions at higher levels allow for quick identification of appropriate processing areas, with fine-grained subdivisions at lower levels allowing for effective workload allocation to available computational resources. Two-level partitioning approaches strike a balance between preprocessing overhead and runtime performance by spending computational resources in early decomposition stages to produce optimized work tasks to reduce redundant processing and maximize parallel execution performance during actual computation stages [5].

Each shard holds a manageable subset of overall test cases, usually in dozens to hundreds of discrete validations based on test complexity and anticipated execution time, and runs in complete isolation from other shards in dedicated function instances with no shared state or communication channels with concurrent execution contexts. Such decomposition presents several strategic benefits that radically

10.48047/jocaaa.2025.34.11.01

change test automation scalability traits and operational efficiency metrics. Individual compute tasks execute isolated workloads with no inter-process dependencies or coordination needs, excluding synchronization overhead and contention for shared resources that beset conventional parallel testing frameworks based on thread-based concurrency models. Failures are kept strictly localized within particular shards without influencing concurrent execution streams since every shard is an independent fault domain in which exceptions, timeouts, or assertion failures invoke localized error handling procedures without having failure states propagated to neighboring shards or interfering with ongoing validation operations in concurrent execution contexts [5].

The system horizontally scales by merely increasing the shard count without provisioning more computational resources, based on the inherent serverless principle that platforms automatically provision more instances of functions to handle greater invocation concurrency without operations teams being involved in capacity planning for infrastructure or resources. Comparative evaluations of containerized microservice-based vs. serverless function-based implementations show different performance aspects and operational trade-offs that guide architectural choices for distributed systems. Legacy microservice deployments utilizing long-running container orchestration systems have instances of services running at all times that consume resources independent of actual request traffic, offering consistent response latencies via pre-warmed runtimes but with constant operational expenses even when idle. Serverless deployments avoid idle resource usage by creating function execution contexts only when requests are received, sharply decreasing costs for workloads with sporadic or variable traffic patterns while adding cold start latencies on platforms that have to initialize new function instances to fulfill incoming requests. Performance tests show that serverless architectures are best suited for workloads that feature infrequent invocation patterns, brief execution times, and tolerance for variability in response latencies, while classic container-based deployments are designed for long-standing high-throughput operations with millisecond-scale response consistency needs and steady connection processing [6].

The sharding method removes historical queue bottlenecks under which tests would build up in pending states waiting for the availability of execution slots on throughput-constrained testing environments, instead allowing for instantaneous parallel processing over hundreds or potentially thousands of concurrent execution contexts constrained by platform concurrency quotas and organizational budget limits rather than physical hardware availability. The distributed execution model supports advanced result aggregation patterns with shard completions causing incremental updates to aggregated test reports, enabling stakeholders to see validation progress in real time as results trickle in from concurrent execution contexts instead of waiting until full suite completion to retrieve any feedback [6].

Sharding Aspect	Implementation Approach	Scalability Impact
Partitioning Algorithms	Analyze historical durations, resource patterns, and dependencies	Optimize execution time within function limits
Shard Composition	Dozens to hundreds of tests per shard	Enable independent processing without dependencies
Spatial Partitioning	Two-level uniform grid structures	Rapid identification and efficient distribution
Fault Isolation	Independent fault domains per shard	Failures are contained without affecting parallel streams
Result Aggregation	Incremental updates as shards complete	Real-time visibility before suite completion

10.48047/jocaaa.2025.34.11.01

Table 2. Sharding strategies enabling elastic scalability in serverless frameworks [5, 6].

Smart Defect Analysis and Categorization

Intelligent Failure Pattern Detection

Integration of artificial intelligence transforms test automation from mere pass-fail reporting to knowledge-intensive quality analysis systems that use advanced machine learning algorithms to derive meaningful insights from test run data. Machine learning algorithms review failure artifacts such as execution logs, error stack traces, and captured screenshots to apply supervised learning-based algorithms to automatically classify defect categories from large historical failure datasets comprised of thousands of labeled examples of mixed failure modes and system behaviors. These classifiers utilize multi-class categorization models that differentiate between environmental issues like network connectivity issues or resource exhaustion states, locator brittleness brought about by dynamic document object model changes or asynchronous rendering behaviors, network timeouts brought on by service degradation or geographic latency fluctuations, application errors like unhandled exceptions or erroneous business logic implementations, and data discrepancies caused by test data corruption or database synchronization failures, all without human interpretation or manual triage effort. The classification schemes use natural language processing methods to scrutinize text content in error messages and stack traces, extracting semantic features that are indicative of certain failure categories based on word patterns, syntactic structure, and contextual relationships between error descriptions and related code execution paths.

Visual analysis algorithms evaluate captured interface screenshots with baseline expectations based on structural similarity evaluation methods that measure image quality and resemblance through computational models that are formulated based on human visual perception traits instead of basic mathematical distance measures. Classic image quality estimation methods involving mean squared error calculations are shown to have weak correlation with subjective human quality opinions since they are indifferent to all pixel errors and are oblivious to the perceptual importance of various distortion modes. Structural similarity models break down image comparison into three discrete components, investigating luminance features, contrast relations, and structural pattern congruence, acknowledging that human vision systems derive structural information from the field of view and that perceived image quality is based primarily on magnitude of structural distortion as opposed to absolute differences in pixel values. The structural similarity index measures degradation of structural information content between reference and distorted images by calculating local statistics in spatial neighborhoods, integrating luminance comparison, evaluating overall brightness consistency, contrast comparison, assessing local variance relationships, and structure comparison, and examining correlation coefficients between normalized pixel patterns after eliminating luminance and contrast effects. This multi-factor evaluation approach allows strong detection of significant visual flaws such as blur distortions, compression artifacts, and geometric distortions while ensuring tolerance for uniform luminance changes or contrast manipulation that do not degrade structural content integrity and have less effect on perceived quality [7].

Root Cause Correlation

More sophisticated systems use recent code updates, deployment activities, and prior failure histories to connect test failures with likely root causes through temporal analysis methods involving examination of chronological relationships between system changes and recorded validation failure occurrences. Advanced deep learning automated bug triaging uses neural network models learned from large historical defect repositories to master intricate patterns relating bug report attributes to suitable developer assignments or defect classifications. These extensive triage systems analyze textual bug descriptions,

10.48047/jocaaa.2025.34.11.01

stack trace content, and related metadata using recurrent neural networks or convolutional models that learn semantic representations that capture important defect features relevant to assignment decisions. Training data for deep triage models generally consist of hundreds of thousands of fixed bug reports across a variety of software versions and development iterations, allowing networks to learn complex relationships between defect symptoms and root causes that are beyond human pattern recognition across such large historical contexts. The deep learning models achieve multi-class classification goals in predicting developer assignments from organizational rosters, with performance measures showing accuracy levels closely approaching human expert triage decisions when compared to held-out test sets of unseen defect reports [8].

Through examination of temporal associations between commit activity and test breakage over several runs, such correlation engines build probabilistic links between individual code changes and resulting test failures through a combination of co-occurrence frequency counts and temporal proximity metrics. The models apply attention mechanisms that concentrate model capacity on informative sections of long stack traces or bug reports, learning to recognize diagnostic words or error patterns most indicative of correct triage choices and discounting irrelevant contextual information. Feature extraction pipelines convert raw text content into compact vector representations of semantic meanings and term-term relationships, allowing similarity-driven reasoning that generalizes across reports describing similar defects using varying words or phrasing conventions. This correlation capability significantly speeds up debugging cycles by focusing engineering effort on relevant parts of code and probable offending developers instead of forcing exhaustive manual analysis, with empirical studies showing significant mean time to assignment and resolution reductions for validation failures analyzed by automated deep triage systems over conventional manual triage processes [8].

10.48047/jocaaa.2025.34.11.01

Analysis Component	Technical Methodology	Diagnostic Outcomes
Failure Classification	Machine learning on labeled historical datasets	Automated categorization of failure types
Natural Language Processing	Extract semantic features from error messages	Correlate failure categories through linguistic patterns
Structural Similarity	Compare luminance, contrast, and structural patterns	Detect visual defects aligned with human perception
Visual Validation	Perceptual hashing via frequency transformations	Identify meaningful changes, ignore rendering variations
Temporal Correlation	Construct causal graphs linking code changes to failures	Prioritize investigation toward relevant modifications
Deep Learning Triage	Neural networks process bug descriptions and traces	Predict developer assignments with expert-level accuracy

Table 3. Artificial intelligence techniques for automated defect analysis [7, 8].

Self-Healing Orchestration Mechanisms

Self-repairing features redefine test automation from error-reporting responses to active quality management systems that automatically identify, diagnose, and repair validation failures with no human intervention necessary on the part of quality engineering teams. The system automatically applies smart retry mechanisms to re-run failed test shards with exponential backoff durations, differentiating between transient environmental problems like temporary network outages, service availability windows, or resource contention states that will improve spontaneously with time and real defects as reproducible functional regressions or design faults that need code changes to fix. The retry mechanisms include advanced failure classification algorithms that process error signatures, execution context metadata, and past failure history to make retry success probability estimates before launching re-execution attempts, avoiding unnecessary consumption of computational resources for chronic failures with a low chance of resolution from mere repetition. Flaky tests are a common problem in software testing, which exhibit nondeterministic behavior where the same test runs under seemingly similar conditions result in inconsistent results, switching between passing and failing states without commensurate changes to the tested code. Qualitative research into flaky test phenomena shows several underlying causes, such as asynchronous wait problems where the tests do not adequately synchronize with application timing behaviors, concurrency issues due to race conditions in multi-threaded execution environments, test ordering dependencies where execution order impacts individual test results, and resource leaks where incomplete cleanup between tests causes environmental pollution for subsequent runs. The consequences of flakiness during testing go far beyond direct inconvenience to developers, actually eroding trust in automated testing infrastructure as teams discover ways to ignore legitimate failures as likely false positives, slowing down defect detection and reducing overall product quality [9].

When defect tracking systems report issue resolution through status changes indicating bugs as fixed, verified, or closed, the architecture automatically initiates selective re-execution of only those test segments initially impacted by the resolved defect, eschewing gratuitous reprocessing of the entire validation suite and saving computational resources for useful validation efforts rather than redundant verification of unaffected functionality. Unstable tests that show intermittent failures with inconsistent

10.48047/jocaaa.2025.34.11.01

pass-fail results during repeated runs under the same conditions are subjected to automatic quarantine processes that temporarily remove them from regular validation streams until restored reliability is assured by consecutive successful execution sequences that pass statistical confidence thresholds as confirmed by stability validation processes. Mitigation measures for flaky tests include preventive actions taken during development stages of the tests and reactive measures applied to existing flaky tests detected via execution monitoring. Prevention methods focus on robust synchronization mechanisms with explicit waits based on well-defined conditions instead of arbitrary sleep statements, thorough test isolation with each test setting up necessary preconditions instead of relying on previous executions, and deterministic test data management not depending on shared mutable state or external dependencies with variable availability. Reactive mitigation methods encompass automated detection of flaky tests via statistical analysis of patterns in execution history, concentrated debugging efforts to detect and remove sources of nondeterminism, and deliberate quarantine policies that quarantine flaky tests while maintaining signal quality of the test suite [9].

Dynamic allocation of resources guarantees test shards waiting in the queue receive computational power instantly without human-scale-up decisions for underlying infrastructure by constantly observing queue depth measurements, processing speed metrics, and platform concurrency utilization levels to adjust function concurrency limits proactively and ensure target service level goals for test run latency. Security testing practices show the benefit of using multiple kinds of analysis techniques to obtain thorough vulnerability coverage for different threat types. Static analysis tools analyze source code or compiled artifacts without running programs, detecting likely security vulnerabilities by matching patterns against known vulnerability signatures and dataflow analysis monitoring potentially malicious information flows from suspicious inputs to sensitive operations. Dynamic analysis techniques run applications in isolated environments and observe the runtime activities, identifying vulnerabilities present only at runtime, such as authentication bypasses, session management issues, and input-dependent injection vulnerabilities. The synergy among static analysis, with its comprehensive coverage of code paths at no execution cost, dynamic analysis, with confirmation of actual runtime behaviors under real-world conditions, and interactive testing methods scanning application reactions to malicious input patterns, provides detection capabilities beyond individual technique shortcomings. Empirical analysis shows that testing strategies that integrate multiple analysis methods identify far wider vulnerability profiles than single-technique methods, and that combined toolchains identify security issues that stand-alone tools miss [10].

10.48047/jocaaa.2025.34.11.01

Mechanism	Operational Strategy	Quality Impact
Intelligent Retry	Exponential backoff distinguishing transient vs. persistent failures	Prevent wasteful resource consumption
Selective Re-Execution	Trigger only affected tests after defect resolution	Conserve resources, avoid full suite reprocessing
Flaky Test Detection	Statistical analysis of execution history patterns	Isolate noise, preserve signal quality
Automated Quarantine	Exclude unstable tests until stability is confirmed	Prevent false positives, maintain confidence
Dynamic Resource Allocation	Monitor queue metrics and adjust capacity continuously	Maintain latency targets, minimize costs
Predictive Capacity Planning	Forecast demands from historical patterns	Prevent degradation through preemptive scaling

Table 4. Autonomous quality governance mechanisms in serverless architectures [9, 10].

Conclusion

Serverless architecture fused with event-driven patterns of orchestration is are paradigm-shifting improvement in test automation for the enterprise, solving inherent limitations in infrastructure-based validation frameworks. The breakdown of monolithic test suites into autonomous shards run across transient compute functions provides unprecedented elasticity to scale parallelism from zero to thousands of concurrent validation contexts without human-operated capacity planning or infrastructure provisioning decisions. Message queuing systems separate test distribution from processing execution, whereas notification services send critical events to multiple subscribers in parallel, building reactive workflows in which system elements react independently to state changes instead of implementing predefined sequential steps. Smart defect categorization using machine learning algorithms and structural similarity evaluation methods turns raw failure artifacts into actionable classifications that separate environmental problems from actual functional regressions, speeding triage cycles and focusing engineering resources on likely root causes. Self-healing processes using smart retry policies, differential re-execution processes, and automatic quarantine methods create quality governance models that independently ensure optimal levels of operation free from human intervention, keeping false positives from undermining confidence in automation outcomes while providing comprehensive regression testing coverage for fixed defects. Predictive capacity allocation continually tracks runtime metrics and utilizes principles of control theory to dynamically reallocate computational power in advance, preserving desired latency targets while reducing provisioning expenses via advanced forecasting mechanisms. The convergence of serverless execution patterns, AI functionality, and event-driven coordination provides cornerstones for autonomous validation systems that run invisibly in continuous deployment pipelines and offer immediate feedback at development speed without fundamentally increasing operational overhead and infrastructure costs in contemporary software delivery environments.

10.48047/jocaaa.2025.34.11.01

References

[1] Ruoyu Su et al., "Emerging Trends in Software Architecture from the Practitioner's Perspective: A Five-Year Review," arXiv, 2025. [Online]. Available: https://arxiv.org/pdf/2507.14554

[2] Sujit Bebortta et al., "Geospatial Serverless Computing: Architectures, Tools and Future Directions," MDPI, 2020. [Online]. Available: https://www.mdpi.com/2220-9964/9/5/311

[3] Oleksandr Kyrychenko et al., "Predictive autoscaling in AWS Serverless by means of machine learning and SQS metrics," The 6th International Workshop on Intelligent Information Technologies & Systems of Information Security, 2025. [Online]. Available: https://ceur-ws.org/Vol-3963/paper7.pdf

[4] Srikanth Jonnakuti, "Real-Time AI with EventBridge and Step Functions: Intelligent Orchestration for Business Pipelines," International Journal of Leading Research Publication (IJLRP), 2025. [Online]. Available: https://www.ijlrp.com/papers/2025/1/1559.pdf

[5] Salles V. G. Magalhães et al., "Fast exact parallel map overlay using a two-level uniform grid," ACM, 2015. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/2835185.2835188

[6] Juan Mera Menendez et al., "A COMPARISON BETWEEN TRADITIONAL AND SERVERLESS TECHNOLOGIES IN A MICROSERVICES SETTING," arXiv, 2023. [Online]. Available: https://arxiv.org/pdf/2305.13933

[7] Zhou Wang et al., "Structural Similarity Based Image Quality Assessment," ResearchGate. [Online]. Available: https://www.researchgate.net/profile/Zhou-Wang-19/publication/265182836_Structural_Similarity_Based_Image_Quality_Assessment/links/5a08bf0f0f7e9b68229cd05d/Structural-Similarity-Based-Image-Quality-Assessment.pdf

[8] Senthil Mani et al., "DeepTriage: Exploring the Effectiveness of Deep Learning for Bug Triaging," arXiv, 2018. [Online]. Available: https://arxiv.org/pdf/1801.01275

[9] Sarra Habchi et al., "A Qualitative Study on the Sources, Impacts, and Mitigation Strategies of Flaky Tests," arXiv, 2021. [Online]. Available: https://arxiv.org/pdf/2112.04919

[10] Francesc Mateo Tudela et al., "On Combining Static, Dynamic, and Interactive Analysis Security Testing Tools to Improve OWASP Top Ten Security Vulnerability Detection in Web Applications," MDPI, 2020. [Online]. Available: https://www.mdpi.com/2076-3417/10/24/9119