WHITE PAPER

Cloud-Native Microservices Testing: De-Risking Application Failures

Contents

Introduction

Cloud-native application development has become mainstream with the convergence of maturing technologies, like infrastructure-as-code (IaC), software-defined cloud computing (SDCC), hyperscale enterprise cloud container platforms, and the widespread adoption of DevOps practices (such as continuous deployment) all of which have led to the agile transformation of information technology (IT) and business.

Distributed cloud-native applications leverage the core principles of cloud computing for resilience, scalability, availability and operational efficiency. By leveraging an architectural paradigm called microservices, application monoliths can be broken down into functional, distributed components that offset development complexity. However, distributed microservices can pose significant challenges to testing and validating applications against stated objectives.

This white paper shares our experience in developing new approaches, methodologies and tooling, including a new microservices testing workbench, for functional testing of modern, cloud-native microservices at scale.

Cloud-Native Software Development

In contrast to traditional models of software, cloud-native applications specifically target and take advantage of cloud features, such as scale and resiliency.

To achieve maximum efficiency in the development cycle, as well as scale and reliability in deployment and operations, modern applications are often broken down into microservices, which typically comprise a single business function. These can be developed, deployed and scaled independently of one another, communicating with each other via application programming interfaces (APIs).

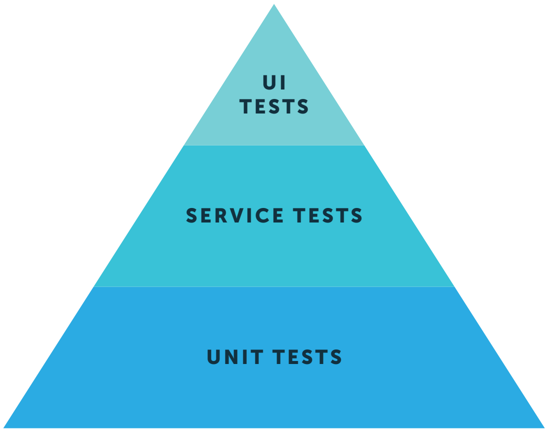
The most common implementation of the microservices pattern is on containers running on the Kubernetes orchestration platform. However, an important and often overlooked aspect of microservices is that this architecture **trades development complexity for operational complexity**, meaning that instead of a single monolithic image, an application may be comprised of many components distributed across many processes and servers.

Container-based, cloud-native applications have become the standard for building distributed applications that can be automated via cloud-native platforms. In this paper, we focus on a particular aspect of cloud-native systems that's often viewed as a lower priority: testing.

Quality engineering and testing of a cloud-native application based on microservices requires a different strategy than those used for monolithic applications. Traditional software testing is often described by a "test pyramid" (first shown by Mike Cohn in the book *Succeeding with Agile*).

> **Microservices can be a means of realizing domain-driven design (DDD), an approach to modeling real-world entities as problem domains. Independent problem areas are referred to as bounded contexts and each bounded context correlates to a single microservice. For example, an eCommerce application can be composed of a shopping cart, catalog, logistics and payment functions, each of which could be a microservice.**

However, testing a cloud-native system requires that we evolve the test pyramid. In a distributed system, functional testing of microservices in isolation must also include API testing to ensure that interfaces function correctly, as well as integration testing to test how the microservices function together.



TRADITIONAL TEST PYRAMID

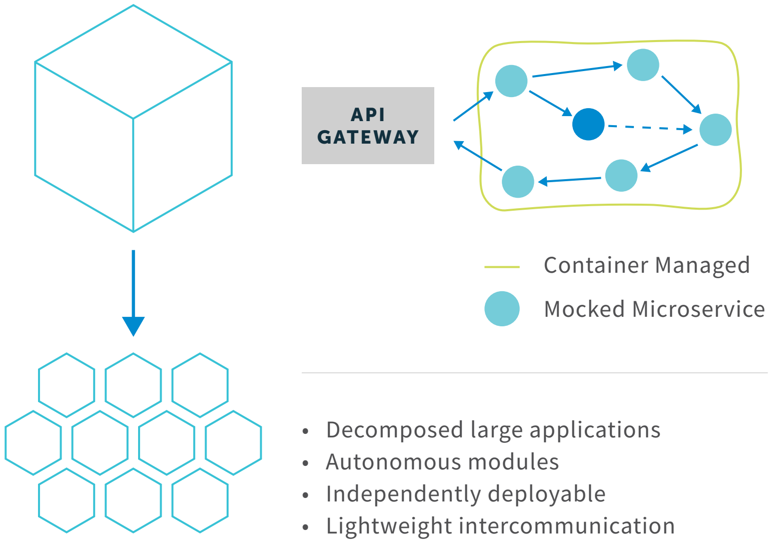
Unlike a monolithic application running as a single process, where components invoke one another using language-level methods or function calls, the biggest challenge in microservices with loose coupling of business logic across different processes is the communication mechanism. A direct migration to microservices through remote procedure calls (RPCs) would make it “very chatty,” making the systems less reliable should any of the processes fail. In this case, it is synchronous communication between the different segregated responsibilities that is not clearly scalable.

Moreover, because transactions may involve many microservices operating across many servers or even regions, traditional transaction management and rollback may not be feasible. Instead, such systems rely on eventual (rather than near real-time) consistency and compensating transactions to restore state in the event of failure.

> **“By 2025, cloud-native platforms will serve as the foundation for more than 95% of new digital initiatives — up from less than 40% in 2021.”**
>
> Source: Gartner®, Top Strategic Technology Trends for 2022: Cloud-Native Platforms, Dennis Smith, Michael Warrilow, Arun Chandrasekaran, Anne Thomas, Sid Nag, David Smith, 18 October 2021. GARTNER is a registered trademark and service mark of Gartner, Inc. and/or its affiliates in the U.S. and internationally and is used herein with permission. All rights reserved.

The following figure shows some common challenges when a monolithic application is broken down into a system of microservices.

MONOLITH TO MICROSERVICES



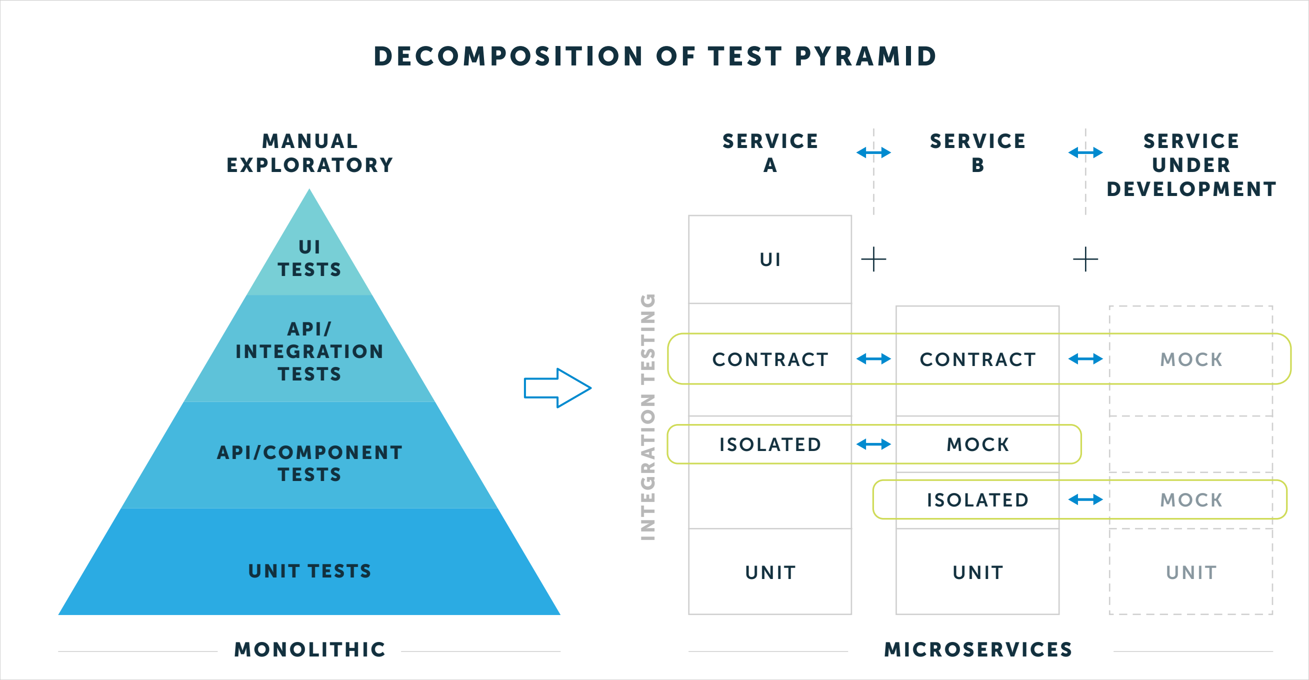
- Decomposed large applications
- Autonomous modules
- Independently deployable
- Lightweight intercommunication

TESTING CHALLENGES

- **Distributed transactions:** Eventual consistency (compensating transactions, not rollback)
- **Horizontal scaling:** Distributed database per service
- **End-to-end testing (testing that launch multiple services):** Difficult, slow, brittle and expensive
- **Testing in isolation:** Though inevitable, may pass but could fail in production due to resource-constrained scaling by design

The distributed nature of the microservices requires implementing test practices earlier in development. As the complexity of a fully packaged application increases, so does the difficulty and risk profile of the application. Fortunately, the democratization of using cloud infrastructure for development and test gives us the opportunity to test the applications earlier in the development cycle.

The figure below illustrates a reconfiguration of the monolithic test pyramid to a structure organized by service layers for microservices.



TESTING OF MICROSERVICES IN ISOLATION

As mentioned, loosely coupled microservices are an architectural pattern that, correctly implemented, enable scale and resilience in cloud applications. However, the distribution of multiple cooperating code components introduces an additional level of complexity to the system – the use of inter-process communication (IPC) mechanisms to connect the distributed microservices. While there are numerous choices for IPC (REST, WebSocket, etc.), common among them is the presence of a contract between any two services. Contracts, or API documentation, describe endpoints, parameters, and normal and error responses. Adhering to the contracts allows teams to develop microservices independently.

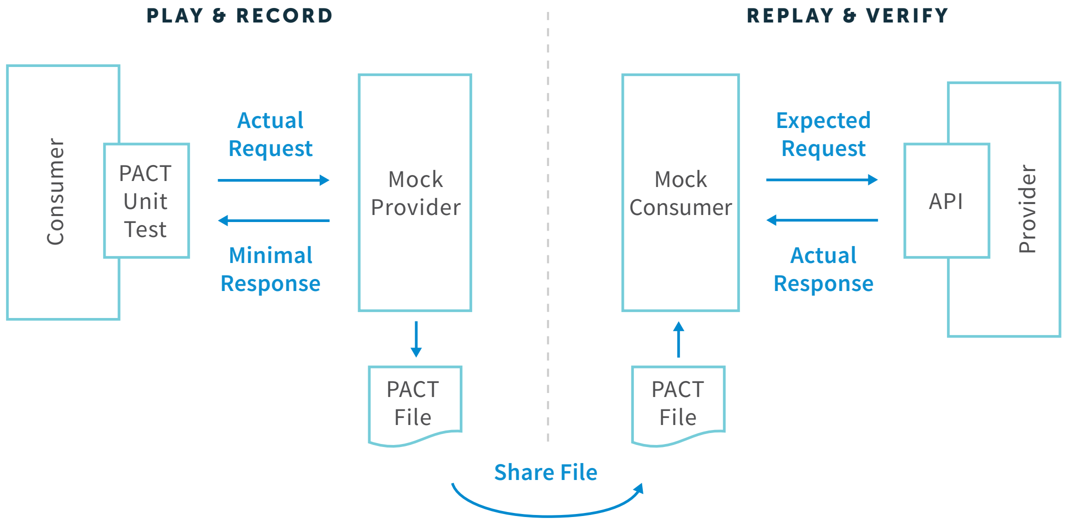
APIs can be used in several different ways, ranging from synchronous request response to publish-and-subscribe metaphors, such as those used in Kafka. Regardless of the design approach, the primary goal when testing microservices is to ensure those contracts are well defined and stable at any point in time. In a test-first, top-down approach, these are the first tests to be covered. A fundamental integration test ensures that the consumer has quick feedback as soon as a client does not match the real state of the producer to whom it is talking. These tests should be part of the regular deployment pipeline. Their failure would inform the consumers that a change on the producer side has occurred, and that responsive changes are required to achieve consistency again. Without the need to write intricate end-to-end tests, "consumer-driven contract testing" would target this use case.

In effect, it is about establishing what the consumer expects to receive and verifying that the producer is indeed sending the expected responses, both the format and data contents. One of the popular libraries that supports this type of testing is PactJS library, a JavaScript version of the open source PACT microservices testing package.

The flow has two distinct parts – "Play & Record" and "Replay & Verify." As seen in the figure below, by using PACT, the mock provider and mock consumer behaviors are run as a local service. A PACT broker exposes a representational state transfer (REST) API for publishing and retrieving PACTs. If the PACT unit tests are correct, then a JSON PACT file is generated. This is then shared with the team developing the provider.

Once instrumented, these tests can be added to continuous integration automation. As we navigate through the services layers, the focus will shift toward the integration testing of components from other domains and how the external services can be validated.

CONSUMER-DRIVEN CONTRACT TESTING STEPS USING PACT



SCALING MICROSERVICES

The complexity of testing microservices increases with the different patterns of microservice deployment (for example, API management layers or message buses) that are used to scale them across process and server boundaries. While the approach to testing them in isolation helped us to understand the individual service better, debugging and understanding the root cause of a problem faced in a real-world distributed deployment **requires tracing requests across bounded contexts** (process, server, etc.).

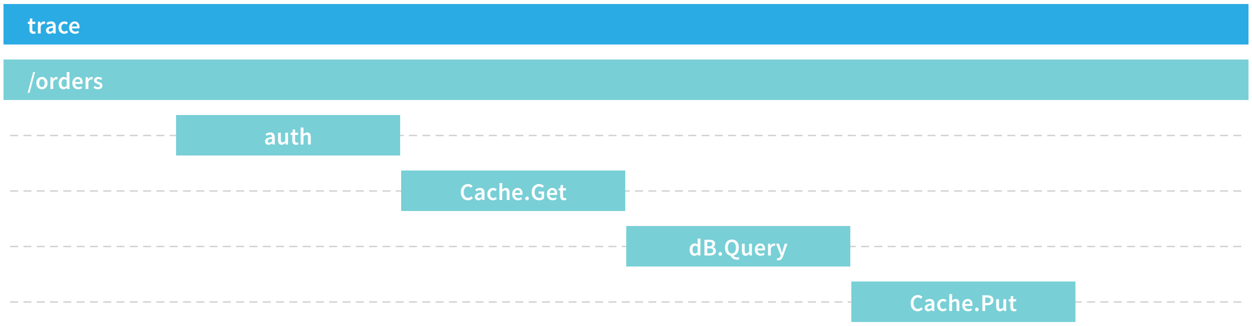
In monolithic architecture, call stacks were used for analyzing the flow of execution (Service A->B->C), as they were running in a single process. But challenges arise in the microservices world as the **service calls are made beyond the process boundary**. Distributed tracing provides the answer for the cloud and microservice architecture.

Debugging and observing distributed transactions in a microservices implementation is harder without a distributed tracing framework, as there are no in-memory calls or stack traces to do so. Distributed tracing, also called distributed request tracing, is a method used to profile and monitor applications, especially those built using a microservices architecture. Distributed tracing helps identify where failures occur and what causes poor performance.

Distributed tracing requires developers to explicitly add instrumentation code into the application to work. One business transaction might involve multiple combinations of protocols and technologies. While the distributed computing lends itself well to the design goal of isolating failure, it adds significant complexity to debug, trace and fix the failures.

> **"As Twitter has moved from a monolithic to a distributed architecture, our scalability has increased dramatically. Because of this, the overall complexity of systems and their interactions has also escalated. This decomposition has led to Twitter managing hundreds of services across our datacenters. Visibility into the health and performance of our diverse service topology has become an important driver for quickly determining the root cause of issues, as well as increasing Twitter's overall reliability and efficiency. Debugging a complex program might involve instrumenting certain code paths or running special utilities; similarly, Twitter needs a way to perform this sort of debugging for its distributed systems."**
>
> Source: https://tinyurl.com/2p9fpp4f

EXAMPLE OF A TRACE



Often, these request tracing capabilities of a service mesh are integrated with instrumentation libraries, like OpenCensus or OpenTracing. Testing the services requires the ability to tie together the entire trace, which requires the application to propagate request-specific tracking information in the headers between services using the span. A span is each unit of work in a trace, the different spans represent internal requests that are made to process the top-level request for "orders," for example. The authorization check is followed by the availability of the information in the cache and a database query (in this event there is a miss in the cache and must be reloaded).

Open instrumentation libraries, such as OpenTracing, enable microservices to be instrumented using API calls from within the application, helping testers describe and analyze cross-process transactions. These enable anomaly detection, help diagnose steady state problems, and provide distributed profiling, resource attribution and workload modeling of microservices. When these implementations report to backend systems, like Elasticsearch, visualization of request traces using Kibana, or similar visualization and exploration tools, provide contextual information to diagnose issues within the system.

TAP TESTING IN BLUE/GREEN OR CANARY DEPLOYMENTS

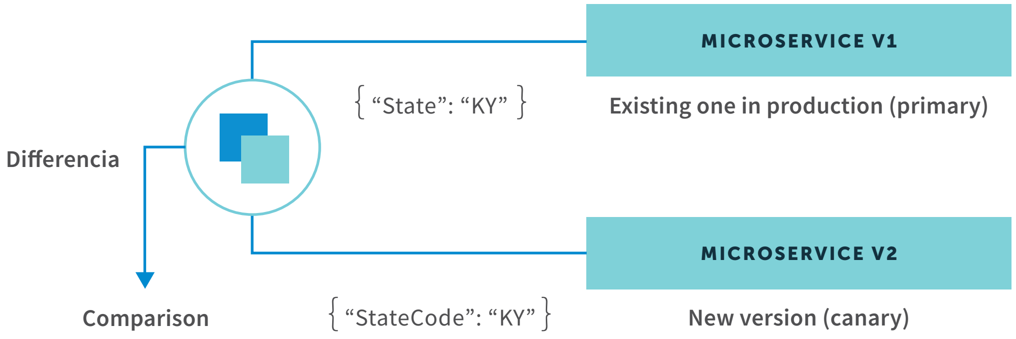
In a microservices-based architecture, many services might be evolving independently at the same time, and often rapidly. Done correctly, microservices thus allow development teams to build and deploy independently of one another, making it possible to respond to business needs more quickly.

However, they must be independently releasable and in an isolated way, meaning that the release is not orchestrated between services.

While providing new levels of agility, this approach raises another concern: can we verify that a new service (or a new version of the service) does not break anything in the current application?

In addition to the ability to trace requests, monitor logs and checking the network configuration, such as routing tables, network address translation (NAT) and proxy settings, testing of different versions of microservices that are deployed can be based on the contract changes of the microservices.

COMPARE SERVICES IN PRODUCTION (CANARY)



This type of testing is very useful in blue/green deployment and can be applied to testing in production. In fact, tap compare testing assesses an environment without developing test scripts. It requires an automated design to fetch production requests. Then, it is ready to sample real production requests by sending the fetched requests to a canary or blue/green production environment, not servicing production traffic.

Diferencia, an open source application, is one such tool that can be used as shown above. Diferencia acts as a proxy, with each request being multi-casted to multiple versions of running services. When the response from each of the services is returned, it then compares the responses and checks if they are "similar." If after repeating this operation with a representative amount of different requests, and all (or most) of them are "similar," then a new service can be considered regression-free.

When using Kubernetes, a canary deployment can tap the ingress controller configuration to assign specified percentages of traffic requests to the stable and canary deployments. Combined with a solution like flagr, that supports dark launching, it can route traffic upon inspection of the request packet to identify which flag variant to apply for the request context.

Other tools like **flagger** (not to be confused with flagr) provide additional capabilities around the closed control-loop, that give dynamic routing capabilities based on key performance measurements, such as request success rate, request average duration and health of the pods, posing further challenges.

Adapting the test strategy – based on the implementation pattern and deployment configuration – needs a test instrumentation set that can validate the configurations and routing rules. The adoption of a tester's workbench, which models the traffic patterns, usage and deployment configurations to manage testing of roll-out flags in a distributed architecture for system validation is essential, as the traffic is now dynamic and policy-based.

EVENT-DRIVEN REACTIVE MICROSERVICES

A critical element for scaling out microservices is the adoption of an asynchronous messaging queue to implement publish-and-subscribe semantics, such as Kafka, in a distributed system architecture involving microservices.

A pattern that has found wide adoption is the smart endpoints dumb pipes paradigm,[1] in which the message bus is responsible for delivering the message, but the endpoint (API) must intelligently interpret it. Typically, in reactive systems, events are a record of what happens to an entity and microservices push them into a durable storage, such as an event log or event queue for other microservices to consume. This then becomes the history that a service can rely on to replay in order to recover like a state machine. One challenge is accepting the fact that when a state changes, it can take an inordinate amount of time before entities see the change.

[1] https://medium.com/@nathankpeck/microservice-principles-smart-endpoints-and-dumb-pipes-5691d410700f

Testing Strategy

The complexity of testing these architectures for resilience, fault-tolerance and consistency increases with this implementation. Testing microservices in these instances can leverage the loosely coupled services' integration approach by injecting messages that are pertinent (subscribed to) for the microservice under test.

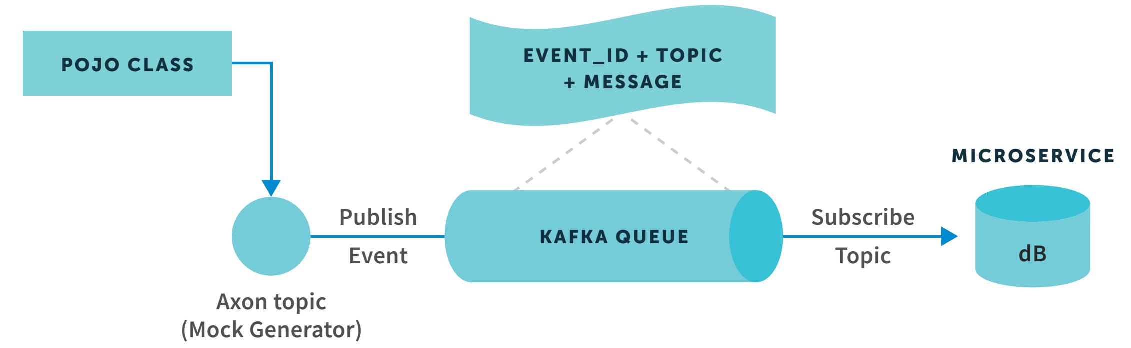
An interesting set of cases involves the way exceptions are handled when a business logic violation is encountered. Because of the distributed and asynchronous nature of the microservices, a set of compensatory transactions must be triggered to revert the ongoing transaction that was being processed but failed. A verification of the eventual consistency of the transactional unit must be confirmed via the query API of each microservices chain involved in the transaction. This is particularly important because a compensating transaction doesn't necessarily return the data in the system to the state it was in at the start of the original operation. Instead, it compensates for the work performed by the steps that were completed successfully before the operation failed. Validating this requires visibility into the sequence of calls through distributed tracing – and reconstruction of the event logs for processing the application state – before the transaction was initiated.

Caselet

In one use case, EPAM engineers used about 300 tests, which took about four to five working days of effort to assess a microservice architected implementation.

Through the automation of event-layer orchestration and using mock producer and consumer objects to test the microservices in isolation, the testing coverage improved significantly.

TESTING MICROSERVICES IN ISOLATION



This enabled contract-based testing of the microservices while reducing the testing effort to just a few hours.

The consumer-driven contract testing provided the higher coverage needed to increase confidence about the development process. In this engagement, Axon, an open source microservices building framework, was used in conjunction with Kafka event bus.

AXON PROVIDES THE FOUNDATION FOR BUILDING ASYNCHRONOUS, MESSAGE-DRIVEN SYSTEMS BASED ON THE CONCEPTS OF:

- **MICROSERVICES**
- **COMMAND QUERY RESPONSIBILITY SEGREGATION (CQRS)**
- **DOMAIN-DRIVEN DESIGN (DDD), EVENT-DRIVEN ARCHITECTURE**
- **EVENT SOURCING**

Caselet

The framework uses an architecture that advocates the design and development of applications by treating it as a "system of events," rather than as a "system of state."

One of the benefits of CQRS, and especially that of event sourcing, is that it is possible to express tests for events and commands.

The fixtures provided by an Axon framework work with any testing framework, such as JUnit or TestNG. By using a fluent interface, optimal expression and use of the stages of configuration, execution and validation can be achieved using the "given-when-then" construct.

Sagas can dispatch commands using a callback to be notified of command processing results.

Often, sagas will interact with resources. These resources are not part of the saga or its state but are injected after a saga is loaded or created. The Axon test fixtures use this capability to allow for the registration of resources that need to be injected into the saga. A useful approach is to inject mock objects, using tools such as Mockito or EasyMock, into a saga. Mock objects allow for verification that the saga interacts correctly with external resources.

Command gateways provide sagas with an easier way to dispatch commands. Using a custom command gateway also makes it easier to create a mock or stub to define its behavior in tests. However, when providing a mock or stub, the actual command might not be dispatched, making it impossible to verify the sent commands in the test fixture. The fixture was used to provide two methods that allow for the registration of command gateways and, optionally, a mock defining its behavior.

The test fixture also can timestamp events from the moment the fixture was created because, by default, the test fixture tries to eliminate elapsing system time, where possible, to make predicting what time events are scheduled for publication easier. If a test case verifies that an event is scheduled for publication in 30 seconds, it will remain 30 seconds, regardless of the time taken between actual scheduling and test execution. Finally, the Axon framework provides capabilities to tune the performance for Axon applications. This includes capabilities to take event snapshots/segment tuning. We then use these capabilities to customize the test strategy, one that leverages the architectural patterns for efficiency, speed and one that is fit-for-purpose.

Continuous Testing of Microservices with Our Test Workbench

Continuous testing is an extremely important part of a cloud-native application pipeline, as it helps to ensure the product is of an expected quality at every stage of the pipeline. Our test workbench can accelerate instrumentation, orchestration and testing of cloud-native microservices for availability and resiliency by inducing network faults and pod disruptions, to provide clear actionable insights and design considerations.

Continuously testing microservices entails engaging tasks and activities at each step in the development and deployment cycle. The test workbench provides an ecosystem of tools to evaluate independent architectural components for resiliency, elasticity, functional correctness and performance in isolation.

The platform can also provide monitoring and trace information using pre-configured tools, such as Prometheus, Grafana and OpenTracing. The developer can also use the platform to shorten the feedback loop with real-time insights for quick and early remediation. The advantage of a built-in, cloud-native test instrumentation is the ability to automate test vectors that can be configured to process logs, trace requests and monitor containers using best-fit tools directly at the source. This also addresses any data residency and privacy considerations.

OUR TEST WORKBENCH ENHANCES THE TEST EFFORT VIA THE FOLLOWING CAPABILITIES:

1 **INTEGRATES SEAMLESSLY WITH A DISTRIBUTED APPLICATION ARCHITECTURE** TO INJECT, TRACE AND REPORT ON DATA THROUGH THE DIFFERENT PARTS OF THE SYSTEM.

2 **PROVIDES ISOLATED SERVICES FOR TESTING-SPECIFIC CONTRACTS** (FORMAT AND DATA TYPES) FOR FUNCTIONAL VALIDATION.

3 **TESTS THE RESILIENCY OF THE SYSTEM** BY INJECTING FAULT TO SIMULATE AVAILABILITY ISSUES; DETECTS TOLERANCE THRESHOLDS AND RECOVERY THROUGH EVENTUAL TRANSACTIONAL CONSISTENCY.

4 **LOCALIZES AND IDENTIFIES PERFORMANCE BOTTLENECKS** AND LATENCY ISSUES CAUSED BY DISTRIBUTED AND ASYMMETRIC SCALING OF PARTS OF THE SYSTEM.

Conclusion

Testing cloud-native applications involving microservices require bespoke solutions to address specific architectural patterns and implementations. To quickly address the specific challenges posed by the implementation, testers need to have the relevant skills, knowledge and cloud-native tools to help facilitate the appropriate tests.

There are risks and challenges associated with cloud-native microservices architectures. The test workbench seamlessly integrates with the various cloud providers to quickly configure the required tools, reducing the time to design, test and validate business objectives. **The cloud-native test strategy includes:**

1. UNDERSTANDING THE DOMAIN DECOMPOSITION AND ASSOCIATED BUSINESS WORKFLOWS.

2. ASSEMBLING THE RIGHT TOOLS TO INSTRUMENT FOR OBSERVABILITY, TRACING, CONTRACTS TESTING AND VALIDATION IN THE CLOUD.

3. A TESTER WORKBENCH THAT PROVIDES OUT-OF-THE-BOX CAPABILITIES FOR END-TO-END TEST MANAGEMENT.

4. BLUEPRINTS AND SOLUTIONS TO ONBOARD APPLICATION PAYLOAD USING DIVERSE CLOUD SERVICES FOR VALIDATION.

5. AUTOMATION RECIPES FOR ACCELERATED, MULTI-LAYER TESTING AND ANALYSIS OF APPLICATION ASSESSMENT.

About The Authors

Venkat Moncompu, Director, Software Testing, EPAM, is passionate about software quality engineering and is an agile evangelist. He has full software lifecycle experience, including product development, customer experience, user experience design, software engineering and quality management spanning 24 years. He is a frequent industry conference speaker and an avid follower of disruptive trends, such as cloud and digital technologies.

venkataraman_moncompu@epam.com

Srikanth Mohan, Quality Architect, EPAM, is an Azure cloud solution architect with 18 years of experience in cloud product validation. In previous roles, he has worked as a lead in presales and solution development activity for quality assurance for evolving cloud products. He worked on various digital transformation projects, IoT and connected devices initiatives, and the cloudification of products. He has special interest in reliability and scalability of cloud applications. He has authored multiple articles in open source forums and actively participates in community meetups.

srikanth_mohan@epam.com

ABOUT EPAM SYSTEMS

Since 1993, EPAM Systems, Inc. (NYSE: EPAM) has leveraged its software engineering expertise to become a leading global product development, digital platform engineering, and top digital and product design agency. Through its 'Engineering DNA' and innovative strategy, consulting, and design capabilities, EPAM works in collaboration with its customers to deliver next-gen solutions that turn complex business challenges into real business outcomes. EPAM's global teams serve customers in over 25 countries across North America, Europe, Asia and Australia. EPAM is a recognized market leader in multiple categories among top global independent research agencies and was one of only four technology companies to appear on Forbes 25 Fastest Growing Public Tech Companies list every year of publication since 2013. Learn more at **https://www.epam.com/** and follow us on Twitter **@EPAMSYSTEMS** and **LinkedIn**.

CONTACT US

GLOBAL
41 University Drive, Suite 202
Newtown, PA 18940, USA

P: +1-267-759-9000

F: +1-267-759-8989