ZUCI SYSTEMS

A Complete Guide

MICROSERVICES ARCHITECTURE & TEST STRATEGIES

www.zucisystems.com

ZUCI SYSTEMS

INDEX

Chapter 1	**Microservices – The basics**	03
Chapter 2	**Why microservices?**	04
Chapter 3	**How does microservices differ from monolithic?**	05
Chapter 4	**Challenges in a microservices set up**	08
Chapter 5	**Approaches to automated microservices testing**	10
Chapter 6	**Microservices Testing - Best Practices**	12

01

MICROSERVICES - THE BASICS

What is Microservices?

Microservices are an alternate approach to application development in which a large application is built in isolation, as a suite of granular components or services. Each component carries a specific functionality or a task or business goals and uses precise APIs as a means of communication to talk to other services.

Martin Fowler, a software developer and a thought leader in his book, stated, "A microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery."

Why the buzz?

Though microservices are a relatively new concept, it gained its momentum when Amazon, Netflix, Uber became trailblazers of the architecture. They were the first to adopt this architecture and have led others with an example by mastering the areas of DevOps and Continuous Delivery, thereby driving their organizations to greatness.

Operating on a microservices architecture, Netflix and Amazon engineers today are able to deploy code thousand times a day and latter every 11.6 sec, allowing them to make a successful transition to continuous deployment.

DID YOU KNOW?

By 2022, 90% of all new apps will feature microservices architectures that improve the ability to design, debug, update and leverage third-party code, according to IDC research.

02
WHY MICROSERVICES?

Much of the migration to the world of microservices begins with developers wanting for more application stability during recurrent code updates or as an easy way out of dependencies or the ability to scale without compromising resources.

A monolithic application becomes an obstacle to managing codebase when the teams want to deploy continuously or when they want to add new features as needed. It can scale only in one dimension, running multiple versions of the application as the volume of transaction increases.

Microservices, unlike a monolithic application, enable you to build services using diverse technologies, and the developers in microservices need not manage code in one place as they do with monolithic apps. Instead deploying a whole application update, they can quickly release code to the container.

The architecture also makes much sense in situations where the teams are working on a complex application and are spread across geographically, where it becomes cost-effective and easy to manage and scale the app seamlessly.

Making the right choice!

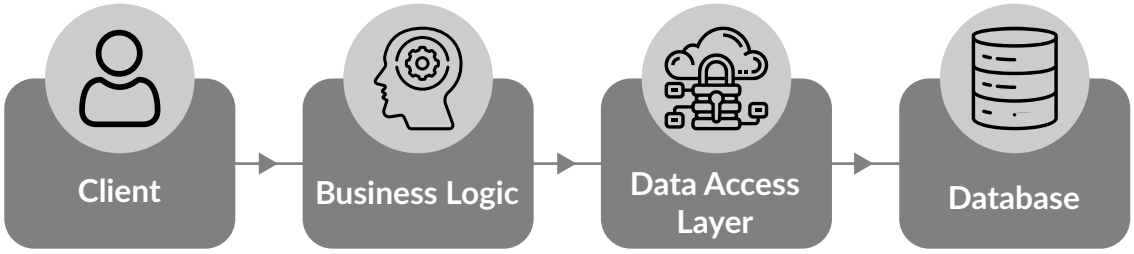
Despite the charm of microservices, the monolithic application still serves quite well for a lot of non-complex applications. Thus, making the right choice of architecture depends on various factors like complexity, developer expertise, and management capabilities. The next chapter of the guide will throw some light on how microservices differ from a monolithic application.

> *The golden rule: can you make a change to a service and deploy it by itself without changing anything else?*
> **— Sam Newman, Building Microservices**

03

HOW DOES MICROSERVICES AND MONOLITHIC DIFFER FROM EACH OTHER?

MONOLITHIC ARCHITECTURE DIAGRAM



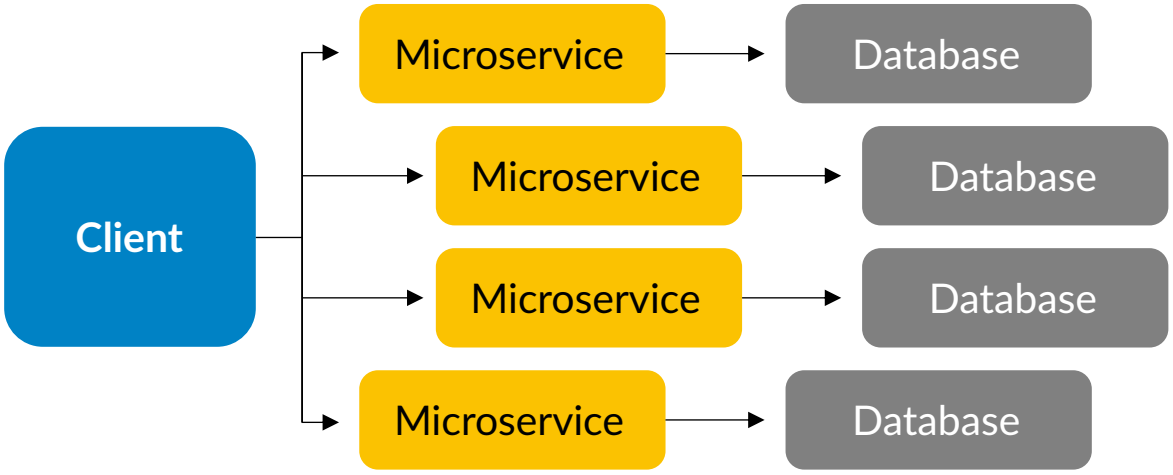
As you can see in the image, A monolith is built as a large system with a single code base and deployed as a single unit; the components in monolithic applications are tightly coupled. This architecture makes itself a tough candidate for continuous deployment, scalability, and management of the applications. The most common drawbacks of following a monolithic architecture include,

Reliability: An error in any one of the modules in the application can shut down the rest of the system

Frequent update: Due to a single giant code base and tight coupling of the components, the entire application would have to undergo deployments for each update

Homogeneous Technology: A monolithic application will have to follow the same tech stack throughout. Hence, changes in the tech stack will be an expensive task.

MICROSERVICES ARCHITECTURE DIAGRAM



On the other hand, microservices are made of loosely coupled, independent services. Elementally, the architecture consists of small autonomous components that can be scaled and deployed independently. This built-up structure gives rise to many benefits, one being the ability to test these independent services individually, and the capability to adapt to any changes made in the application on its journey towards continuous deployments is another.

However, the shift to microservices comes with a price, particularly when it comes to the monitoring of the application and the toll it can take on the development team undergoing the shift. Despite its fault tolerance capabilities, the ability to identify failures in microservices-based applications via manual monitoring and testing methods remains a difficulty. This hardship is primarily due to the architecture's decoupled nature and undefined boundaries

> *When services are loosely coupled, a change to one service should not require a change to another.*
>
> **— Sam Newman, Building Microservices**

How to decide?

Migrants to the world of microservices will have to consider factors such as the team expertise and testing and monitoring tools.

Your team will need the right expertise to build, deploy, and manage microservices-based applications. If your developers are used to monolithic architecture and do not have the right skills, working with microservices can become counterintuitive.

And, once the application is made into loosely coupled components, you'll have more moving parts to track and fix. Without the right monitoring and testing tools in place, adoption can become a nightmare.

Generally, monolithic architectures are the right choice for small, simple app development. Alternatively, a microservices architecture is a right choice when you're developing for complex systems.

Ask yourself these questions before making a choice:

- Do you have well-defined boundaries for application?
- Does your team have microservice expertise?
- Do you have the right infrastructure for independent services?
- Have you evaluated the business risks involved?

> *If you are working in an organization that places lots of restrictions on how developers can do their work, then microservices may not be for you.*
>
> **— Sam Newman, Building Microservices**

04

CHALLENGES IN A MICROSERVICES SETUP

Microservices can provide better flexibility and performances, but the downside of the application is that it can complicate a few basic tasks like testing, monitoring, etc. As an organization transitioning to this distributed architecture, you need to understand the compelling challenges that come along with it.

Let's delve into a few challenges with microservices architecture.

Communication hiccups:

> *The biggest single challenge arises from the fact that, with microservices, the elements of business logic are connected by some sort of communication mechanism ... rather than direct links within program code.*
>
> **- Randy Heffner, Principal Analyst at Forrester**

In this distributed architecture set up, a microservice may need to talk to another microservice or require access to other microservice for a service or in need of data. These scenarios could become favorable for errors in-network and container configurations, errors in request or response, network blips, and errors in security configurations, configs, and much more.

This communication hiccups could pose some challenges when multiple streams of testing are active in an integration testing environment.

Multiple security gates: Components in a microservice architecture operate and communicate in both internal and external environments, which can make monitoring and security a more significant challenge.

With many moving parts in place -- e.g., services, components, APIs, etc., it merely increases the potential attack surface. And, unlike the well-defined security boundaries that a firewall provides a monolithic app, there is no such definitive boundary with cloud-based microservices apps.

Version compatibility: Retaining the older versions of service becomes tricky when teams design, develop and deploy microservices independent of one another. This is problematic because when it comes to versioning, service compatibility may be lost. Teams have to keep in mind to design a microservice that supports version compatibility.

Manual monitoring: While microservices make sure that application is up and running even when there is a network outrage or a system failure, the decoupled nature of the architecture turns particularly complex to identify failures through manual monitoring and testing methods.

The answer to this is incorporating automated monitoring and testing tools. Manual monitoring or testing of too many services simply won't cut. These are some of the common challenges in a microservices architecture; the next chapter of the guide will talk about microservices test strategies

05

APPROACHES TO AUTOMATED MICROSERVICES TESTING

Testing of the microservices-based application is complicated as the services work independently and extend communication through API calls. The test team assigned to test these services must well be aware of the given service, its dependencies, and roll out an effective test strategy accordingly.

There are four conventional approaches to test microservices: unit tests, integration tests, contract tests, and component tests.

Unit testing: The objective of unit testing is to isolate written code to test and validate if the code of each application component is in line with the necessary business logic. In terms of volume, they represent the largest number of tests.

To test microservices flawlessly, you need to keep your test units limited. Large units of distributed services will produce variable complex tests, as those services scale their resource utilization over time. Unit tests can be automated depending upon the language and the framework used in the service.

Integration testing: Multiple microservices interact and execute together to achieve a business goal. This critical part of microservice architecture testing relies on the proper functioning of inter-service communications.

Integration testing validates the connectivity and flow of data between two or more components and its dependencies and identifies bugs within the interaction. Testers must ensure requests and communication channels that flow through the services work seamlessly, and the dependencies between the services are present as expected.

> *Getting integration right is the single most important aspect of the technology associated with microservices in my opinion. Do it well, and your microservices retain their autonomy, allowing you to change and release them independent of the whole. Get it wrong, and disaster awaits.*
>
> **- Sam Newman, Designing Fine-Grained Systems**

Contract testing: Contract-driven testing, often called as contract testing is different from the functionality tests done so far. The contract under tests establishes a relationship between a client seeking data, who is the consumer, and the API on a server that provides the data, the provider.

This contract-driven testing process uses tests from consumers to validate the provider, or API publisher. The consumers of API publish the expected behaviors to a computer program, which stands up a mock server that can verify the tests pass.

To pass the test, the calls and responses (provider) must produce the same results every time, even if the service is altered or upgraded. Albeit, there is a flexibility to add more functionality to the responses as required; these additions must not break the service functionality.

For contract testing on microservices, both the service producer and consumer must have the most up-to-date version of the contract.

Component testing: Component tests are used to test the components or services of an application in isolation. It checks and validates the microservices in isolation by creating mock services that mirror the dependent system's responses and behavior, the process is also called as service virtualization/stubbing/mocking.

Virtualized services are replicas of systems that new applications depend on. They are created to test how well the application and systems integrate. This process removes a significant development bottleneck by ensuring testers don't have to wait to begin testing. It allows teams to release products with fewer defects, on-time without disrupting service.

The next chapter will give some tips on how to go about addressing microservices testing.

06

MICROSERVICES TESTING – BEST PRACTICES

Strengthen your system integration test environment: In the pursuit of continuous delivery, teams want to deploy new features as often as possible and, they run automated E2E tests to check GUIs before deploying to production thoroughly. This way, the team can simply deploy a change to the system integration test (SIT) environment, run the tests, and wait for the results.

But imagine someone rolls out a new GUI element during the test run, the chances are that it can cause a test failure. And after some point in time during the re-run, the odds are even high that different sets of sub-tests could fail again. This happens because repetitive tests act as change detection making the tests to report those changes as a failure.

The remedy to this problem is strengthening the SIT environment, and this can be done by reducing the number of unique but similar microservices that developers create and investing in better update deployments.

80/20 rule: Traditional end-to-end testing, covering all possible workflows that a real user might perform will not be as effective as it is with other software architectures. A better approach would be making a contract with your consumers and apply 80/20 rule: invest 20% of your testing efforts on 80% of the areas your users usually consume.

Test in production: Microservices are made of fluid shifting relationships, where it's almost uncertain on the ways how these microservices are going to be consumed and how they are going to behave. The best practice is to shift-right into testing in production.

Monitoring tools: With the monitoring tools in place, you can trace the issues in production at runtime and can react quickly. Tracing even helps to go back to the last known good version of the service before it slips to the hands of users.

ZUCI SYSTEMS

Final thoughts:

Some engineering teams may feel uncomfortable in their initial stages to take microservices route to their DevOps journey. However, depending on the team's expertise, listening carefully to the customer feedbacks and having the right test strategies in place, they can soft-pedal into the DevOps world of ambiguity and uncertainty.

Headquarters – Chicago, U.S.
+1 (331) 903 – 5007

Office – Chennai, India
+91 (44) 49525020

www.zucisystems.com

sales@zucisystems.com