# History of Microservices

* The microservices style of architecture develops complex application software from small, individual applications that communicate with each other using language-independent interfaces (APIs).
* Companies run into trouble if they are unable to scale monolithic architecture that has developed over time, if their architecture is difficult to upgrade or maintenance becomes too complex.
* Microservices can be the answer to this problem, as they break down complex tasks into smaller processes that work independently of each other.
* **Central services**—Handle business data persistence and apply business rules and other logic.
* **Composite services**—Organize either a number of central services in order to fulfill a common task, or aggregate information from several central services.

# Problems with Monolith & SOA

* Service-oriented architecture (SOA) is an enterprise-wide approach to software development of application components that takes advantage of reusable software components, or services.
* In SOA software architecture, each service is comprised of the code and data integrations required to execute a specific business function — for example, checking a customer’s credit, signing into a website or processing a mortgage application.
* The service interfaces provide loose coupling, which means that they can be called with little or no knowledge of how the integration is implemented underneath.
* Because of this loose coupling and the way the services are published, development teams can save time by reusing components in other applications across the enterprise.

# Microservices Architecture

* Microservices architecture (often shortened to microservices) refers to an architectural style for developing applications.
* Microservices allow a large application to be separated into smaller independent parts, with each part having its own realm of responsibility.
* To serve a single user request, a microservices-based application can call on many internal microservices to compose its response.
* Containers are a well-suited microservices architecture example, since they let you focus on developing the services without worrying about the dependencies.
* Modern cloud-native applications are usually built as microservices using containers.

# Problems Solved by Microservices

* Before going any further, let me offer a specific definition.
* After all, relying on vague, hand-waving definitions is the main culprit in buzzword fatigue.
* I certainly don’t want to contribute to that.
* Imagine an organization with a large, monolithic website.
* This sprawling entity serves as the company’s entire public interface on the internet.

# Designing Microservices Architecture

* Imagine an organization with a large, monolithic website.
* This sprawling entity serves as the company’s entire public interface on the internet.
* I'm a Software Architect at Software Mill and I've been designing microservices implementations for the past few years. They are not a perfect solution to every problem, sometimes a well structured monolith is better.
* How to investigate if your system would benefit from microservices without jumping right away into a binary approach? Here are my lessons learned and thoughts on microservices.

# Testing Microservices

* In this article, I will share my experience in **microservices testing**.
* Most of the companies are implementing the microservices architecture to have the capability to develop, test, and deploy the services independently and faster.
* Of course, these architectures come with a lot of complexities and in order to test these systems effectively, we need to be aware of the system architecture very well.
* Let’s start with a simple architectural view of microservices.
* Generally, we have clients/channels/consumers such as Web, Mobile Web, Mobile Apps (iOS and Android), and Desktop.
* We may have some downstream or external services which do business-critical operations such as loyalty operations, customers’ data-related operations, and they may hold some critical data of the business.
* These operations and data depend on the company’s sector.

# Logging and Monitoring

* Logging and monitoring are both valuable components to maintaining optimal application performance.
* Using a combination of logging tools and real-time monitoring systems helps improve observability and reduces the time spent sifting through log files to determine the root cause of performance problems.
* Logging is used as both a verb and a noun, referring either to the practice of logging errors and changes or to the application logs that are collected.
* The purpose of logging is to create an ongoing record of application events

# When not to Use Microservices

* Microservices are solutions to complex concerns and if your business doesn’t have complex issues, understand that you don’t have a system in place to handle the complexities of microservices.
* Using microservices can prove to offer contrary consequences if you don’t have a team size that cannot handle the tasks involved. This will only result in the delay of delivery.
* Implementing microservices for the sake of it can be hampering as well. If your application does not require to be broken down into microservices, you don’t need this.
* There is no absolute necessity that all applications should be broken down to microservices. There are those that are simple by their nature and functionality

# Microservices and the Organization

* Microservices are often advertised as an architecture style/pattern. What is hardly ever mentioned is that the organization needs to be supportive of this style as well.
* The main takeaway is that Microservices is not just an architecture style, but also an organizational structure.
* Adopting the one without the other is like buying new running shoes and wearing them just to watch Netflix.
* Microservices do offer a solution to this caveat and they do so by organizing around business functionalities rather than technologies.
* The old Unix philosophy ‘do one thing and do it well’ certainly applies to Microservices where the aim is to build smaller and clearer bits of functionality that are functioning very well.

# Anti-Patterns and Common Mistakes

* An **anti-pattern** is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive.
* The term, coined in 1995 by computer programmer Andrew Koenig was inspired by the book *Design Patterns*, which highlights a number of design patterns in software development that its authors considered to be highly reliable and effective.
* An **anti-pattern** is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive.
* The term, coined in 1995 by computer programmer was inspired by the book, which highlights a number of in software development that its authors considered to be highly reliable and effective.
* The term was popularized three years later by the book  which extended its use beyond the field of software design to refer informally to any commonly reinvented but bad solution to a problem.
* Examples include analysis paralysis, cargo cult programming, death march, groupthink and vendor lock - in

## **Common Microservices Anti-Patterns**

Some of the common anti-patterns include *Break the Piggy Bank*, *Cohesion Chaos*, *Versioning Avoidance*, *Gateway for each service*, *Everything Micro*, and so forth. In the sections that follow, I will walk you through the most common anti-patterns and pitfalls when working with microservices-based applications, and solutions for avoiding them.

### **Distributed Monolith**

This is one of the biggest concerns when you split an existing monolithic application into microservices. If a service needs all other microservices or libraries to be available to execute, that’s a distributed monolith. The choice between a microservices architecture and a distributed monolith depends on several factors. That’s the million-dollar question. To answer this question, we will need to be able to answer a few questions that follow.

First off, will a change in a microservices in your application require redeployment of other microservices of the application as well? If so, your microservices are not decoupled from each other. This breaks the basic purpose of adopting microservices architecture – a major benefit of microservices architecture is decoupling such that services can be independently developed, tested, deployed, and maintained. Finally, you have a closely connected architecture, which results in delivery entropy and cohesion chaos, as well as other problems.

# Breaking Monolith to Microservices

* Follow these tips from two experts, both of whom presented at the 2020 O'Reilly Software Architecture Conference in New York, to break up messy monoliths into tidy microservices -- or discover where to leave the monolith in place.

### Step 1: Establish an application architecture baseline

* To break apart a monolith, you first have to know what's in it. At the start of any microservices project, segment the existing monolith's architecture by logical components, such as the namespaces.
* Components, in this context, are any sets of classes that do one thing, said Mark Richards, an independent consultant. "You don't have to move individual classes," he said.

### Step 2: Refactor into a domain-driven design

* To progress toward microservices, assess the size of the namespaces identified as future services.
* If any single component in the architecture contains more than 10% of the overall application, break it down into smaller pieces, Richards said.
* He bases this on statements, rather than lines of code, since code volume varies from one developer to another.

### Step 3: Build microservices

* Microservices are pieces of single-purpose code that do one thing well. From tens of business or functional services, an application architecture can split into hundreds or thousands of microservices.
* Start your work at points where the application will benefit the most from microservices -- ideally, where automation, DevOps and containerization are already in place.
* If possible, tackle a high-change but low-risk component, Richards said, such as nonfinancial, customer-facing functionality.
* Using the strategies described above, like dark launches, teams increase the chance that they can refactor without negative consequences on the user base.