**MICROSERVICES**

1.Introduction:

Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are

* Highly maintainable and testable
* Loosely coupled
* Independently deployable
* Organized around business capabilities
* Owned by a small team

The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack.

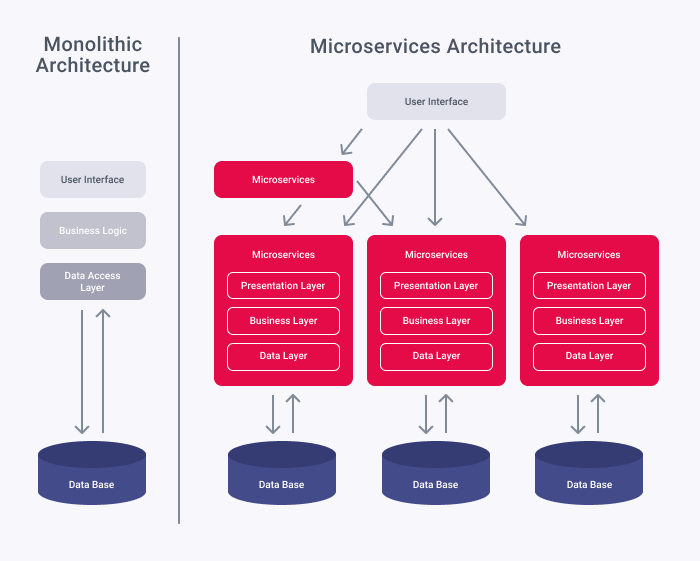
2.Why Microservices..?:

**2.1**. Microservices provide the ideal architecture for continuous delivery. With microservices, each application resides in a separate container along with the environment it needs to run. Because of this, each application can be edited in its container without the risk of interfering with any other application.

**2.2**. Microservice architecture allows you to maximize deployment velocity and application reliability by helping you move at the speed of the market. Since applications each run in their own containerized environment, applications can be moved anywhere without altering the environment. If an application works in development, it will work for the customer. This speeds up time to market and increases product reliability.

**2.3**. Microservices can also help you adapt more quickly to the changing market conditions. Because microservices allow applications to be updated and tested quickly, you can follow market trends and adapt your products faster.

**2.4**. Microservices provide the tools developers need to build higher quality software products. With a microservice architecture, each component of an application can exist in its own container, independently managed and updated. This means that developers can build applications from multiple components and program each component in the language best suited to its function, rather than having to choose a single less-than-ideal language to use for everything. Optimizing software all the way down to the components of the application helps you increase the quality of your products.

3.Monolithic Vs Microservices:

4.Microservices Architecture:



5.Different Frameworks of microservices architecture:

5.1. Spring Boot Framework

Spring Boot gives you Java application to use with your own apps via an embedded server. It uses Tomcat, so you do not have to use Java EE containers.

Spring Boot projects include:

* **IO Platform**: Enterprise-grade distribution for versioned applications.
* **Framework**: For transaction management, dependency injection, data access, messaging, and web apps.
* **Cloud**: For distributed systems and used for building or deploying your microservices.
* **Data**: For microservices that are related to data access, be it map-reduce, relational, or non-relational.
* **Batch**: For high levels of batch operations.
* **Security**: For authorization and authentication support.
* **REST Docs**: For documenting RESTful services.
* **Social**: For connecting to social media APIs.
* **Mobile**: For mobile Web apps.

5.2. Jersey Framework

Jersey RESTful framework is open source, and it is based on JAX-RS specification. Jersey’s applications can extend existing JAX-RS implementations and add features and utilities that would make RESTful services simpler, as well as making client development easier.

The best thing about Jersey is its exceptional documentation. Jersey is also fast and has extremely easy routing.

Jersey is very easy to use with other libraries, such as Netty or Grizzly, and it supports asynchronous connections. It does not need servlet containers. It does, however, have an unpolished dependency injection implementation.

5.3. Play Framework

Play Framework gives you an easier way to build, create, and deploy Web applications using Scala and Java. This framework is ideal for RESTful application that requires you to handle remote calls in parallel. It is also very modular and supports async. Play Framework also has one of the biggest communities out of all microservices frameworks.

5.4. Restlet Framework

Restlet helps developers create fast and scalable Web APIs that adhere to the RESTful architecture pattern. The framework has good routing and filtering. It’s available for Java SE/EE, OSGi, Google’s AppEngine (which is part of Google Compute), Android, and many other Java platforms. It’s a self-sufficient framework that even ships with its own webserver.

Restlet comes with a steep learning curve that is made worse by a closed community, but you can probably get help from people at StackOverflow.

5.5. RESTful and micro services

There’s a reason why RESTful services and microservices are often associated with each other. It’s because the best microservices architectures treat their services as stateless. REST’s state transfer that pushes state down to the clients means that you can treat your servers as stateless, and run your code as interchangeable parts of a whole. You only need to worry about making sure that there are enough services available to handle the load. And, if one fails, another can pick up the slack.

6.Tools used for Microservices:

6.1. Containers

Containers are a natural fit for microservices .You have to use containerization for microservice architecture, there are no options.Containers give you the ability to create a full automation delivery pipeline — build and test a release in the specific environments, deploy a new release by creating a container instance to replace an old one. It is necessary for rapid deployment and for reducing time to market.

6.2. Infrastructure as Code Conception

It’s not just light-weight containers that make microservices architecture possible. Infrastructure as Code (IaC) stands behind containers and makes them so useful. A container description is **a text file**. This feature brings a list of new interesting things to the old development process. My top list of new things:

1. Put a container description to Git or Hg and commit any changes to DVCS.
2. Look at change history of project’s environment.
3. Compare environments, for example, compare staging and production environments.
4. Easy switch between environments on the local machine.
5. Share containers (environments) in the team.
6. Run the same multi-tiered application on a developer’s laptop, a QA server, or a production cluster of cloud instances, and it behaves exactly the same way.

Therefore, IaC will help you to manage microservice infrastructure.

6.3. Containers Orchestration

Containerization is an indispensable tool for scaling up a service by using scripts automatically. If you have a service under a high load, you have to dynamically increase or decrease the number of containers that run this service, which depends on the load level. Managing containers is a non trivial task so I recommend using orchestration.For instance, Docker has had a built-in orchestration mechanism since 2016. There are a number of cloud and on-premise solutions for container clustering and orchestration.

6.4. Cloud Infrastructure

Microservices demand reliable and scalable infrastructure. Our clients chose clouds most of the time. We usually deploy microservice infrastructure on Amazon EC2 or Azure Service Fabric. All popular cloud platforms have API for Docker and other containers.

6.5. Serverless

I have to highlight Serverless because it has changed the rules.We use Azure Function or AWS Lambda for triggering infrequent operations or events. These tools are helpful for integrating pipelines and message processing. For example, when you process documents, you could send a piece of text to the Function to extract tags and get the result back.

6.6. API Gateway

If you connect a hundred microservices directly to each other, you’ll find how hard it is to control security, analyze traffic, create reliable channels and scale in/out an infrastructure for high-load API calls.

You can use API Gateway to avoid these problems and centralize API management. Azure API Management and Amazon API Gateway offer a cloud gateway for creating, publishing, maintaining, monitoring, and securing APIs at any scale. Also, there is a list of on-premise solutions, for example, Mulesoft API Gateway.

Your microservices should know only about API Gateway and connect to each other only through this gateway, therefore you’ll get full control under API calls.

## 6.6.1. Refactoring Legacy Systems

Specifically, API Gateway is helpful for replacing legacy systems with a new one or changing architecture to microservices. The first step is to change any direct API calls to go through API Gateway. As a result, every system will know only about API Gateway but not about other subsystems. Then you can easily split and change legacy systems one by one.

6.7. Enterprise Service Bus(ESB)

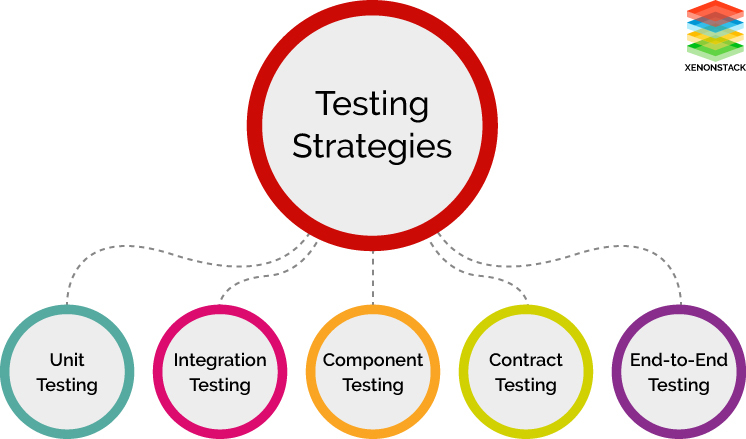
As API Gateway centralizes synchronous calls, an Enterprise Service Bus (ESB) centralizes asynchronous messaging for inter-service communication.

6.8. API Gateway Vs ESB

Sometimes it is really hard to choose between ESB and API Gateway in any particular case. The reason for this obstacle is that ESB can be used for a synchronous call simulated by implementing two queues, one for an outgoing request and another for an incoming response. On the other hand, API Gateway can send an asynchronous call by creating a waiting thread on both sides. Start exploring this topic in the article API platforms are different from “just another ESB”.

Any mature cloud platform offers a reliable and scalable message bus. For example, AWS has Simple Queue Service (SQS) and Azure has Service Bus. My favorite on-premise solutions are Apache Kafka and RabbitMQ.As a result, your architecture will look like some sort of combination of API Gateway and ESB:

7.Testing of microservices:



A microservices architecture consists of focused, small services that together create a complete application or task. Every instance of a microservice represents a single responsibility within your application. The real advantage is that, these services are independent of one another, which makes them independently deployable and testable.

7.1. Unit Testing  
The scope of unit testing is internal to the service. In terms of volume of tests, they are the largest in number. Unit tests should ideally be automated, depending on the development language and the framework within the service.

7.2. Contract Testing  
Contract testing should treat each service as a black box and all the services must be called independently and their responses must be verified. Any dependencies of the service must be stubs that allow the service to function but do not interact with any other services. This helps avoid any complicated behavior that may be caused by external calls and turn the focus on performing the tests on a single service.

A “contract” is how a service call (where a specific result or output is expected for certain inputs) is referred to by the consumer-contract testing. Every consumer must receive the same results from a service over time, even if the service changes. There should be the flexibility to add more functionality as required to the Responses later on. However, these additions must not break the service functionality. If the service is designed in this manner, it will stay robust over longer durations and the consumers will not be required to modify their code to take into account the changes made later on.

7.3. Integration Testing  
Verification of the services that have been individually tested must be performed. This critical part of microservice architecture testing relies on the proper functioning of inter-service communications. Service calls must be made with integration to external services, including error and success cases. Integration testing thus validates that the system is working together seamlessly and that the dependencies between the services are present as expected.

7.4. End-To-End Testing  
End-to-end testing verifies that the entire process flows work correctly, including all service and DB integration. Thorough testing of operations that affect multiple services ensures that the system works together as a whole and satisfies all requirements. Frameworks like JBehave help automate Functional testing by taking user stories and verifying that the system behaves as expected.

7.5. UI/Functional Testing  
User interface testing is the testing of the highest order as it tests the system as an end-user would use it. Testing of this level must feel like a user trying to interact with the system. All the databases, interfaces, internal and third-party services must work together seamlessly to produce the expected results

7.6. Microsoft Testing Tools

**7.6.1**. Goreplay is an open-source network monitoring tool that records your live traffic. This tool can be used for capturing and replaying live HTTP traffic into your microservices test environment.

**7.6.2**. Mountebank is an open-source tool that provides cross-platform, multi-platform test doubles over the wire. You can simply replace real dependencies with Mountebank, and test like you’d do with traditional stubs and mocks.

**7.6.3**. VCR helps you in recording your test suite’s HTTP interactions that can be played later during future tests for fast, accurate and reliable tests.

**7.6.4**. Wilma is a service virtualization tool with the combines capability of Service Stub and a HTTP/HTTPS Transparent Proxy. It is easily expandable via plug-ins and can be configured on-the-fly.

**7.6.5.** Hikaku is a library that helps you in making sure that the implementation of REST-API meets its specifications.

**7.6.6.** Mitmproxy is a free and open-source interactive HTTPS proxy that can be used for debugging, testing, privacy measurements, and penetration testing.

**7.6.7.** Wiremock is a simulator for HTTP-based APIs. Unlike general purpose mocking tools, it works by creating an actual HTTP server that your code under test can connect to as it would a real web service.

8. Microservices and Data management:

8.1. Database per service

In this pattern, each microservice manages its own data. What this implies is that no other microservice can access that data directly. Communication or exchange of data can only happen using a set of well-defined APIs.

It sounds easier than it actually is to implement this pattern. Applications usually are not so well demarcated. Usually, microservices need data from each other for implementing their logic. This leads to spaghetti-like interactions between various services in your application.

The success of this pattern hinges on effectively defining the bounded contexts in your application. For a new application or system, it is easier to do so. But for large and existing monolithic systems, it is troublesome.

Other challenges include implementing business transactions that span several microservices. Another challenge could be implementing queries that want to expose data from two or three different bounded contexts.

However, if done properly, the major advantages of this pattern are loose coupling between microservices. You can save your application from impact-analysis hell.

Also, you could scale up microservices individually. It can provide freedom to the developers to choose a specific database solution for a given microservice.

8.2. Shared Database

A shared database could be a viable option if the challenges surrounding Database Per Service become too tough to handle for your team.

This approach tries to solve the same problems. But it does so by adopting a much more lenient approach by using a shared database accessed by multiple microservices.

Mostly, this is a safer pattern for developers as they are able to work in existing ways. Familiar ACID transactions are used to enforce consistency.

However, this approach takes away most of the benefits of microservices. Developers across teams need to coordinate for schema changes to tables. There could also be run-time conflicts when multiple services are trying to access the same database resources.

Overall, this approach can do more harm than good in the long run.

8.3. Saga Pattern

The Saga pattern is the solution to implementing business transactions spanning multiple microservices.

A **Saga** is basically a sequence of local transactions. For every transaction performed within a Saga, the service performing the transaction publishes an event. The subsequent transaction is triggered based on the output of the previous transaction. And if one of the transactions in this chain fails, the Saga executes a series of compensating transactions to undo the impact of all the previous transactions.

As you can see, this is drastically different from the usual point-to-point call approach. This approach adds complexity. However, in my view, Sagas are a very powerful tool to solve some tricky challenges. But they should be used sparingly.

8.4. API Composition

This pattern is a direct solution to the problem of implementing complex queries in a microservices architecture.

In this pattern, an API Composer invokes other microservices in the required order. And after fetching the results it performs an in-memory join of the data before providing it to the consumer.

As evident, the downside to this pattern is the use of inefficient in-memory joins on potentially large datasets.

8.5. CQRS

CQRS, or Command Query Responsibility Segregation, is an attempt to get around the issues with API Composition pattern.

An application listens to domain events from other microservices and updates the view or query database. You can serve complex aggregation queries from this database. You could optimize the performance and scale up the query microservices accordingly.

The downside to this is an increase in complexity. All of a sudden, your microservice should be handling events. This can cause latency issues where the view database is eventually consistent rather than always consistent. It can also increase code duplication.

8.6. Event Sourcing

Event sourcing mainly tries to solve the problem of atomically updating the database and publishing an event.

In event sourcing, you store the state of the entity or the aggregate as a sequence of state changing events. A new event is created whenever there is an update or an insert. The event store is used to store the events.

You can use this pattern in conjunction with CQRS. By doing so, a lot of challenges around event handling and maintaining query data can be solved.

However, as a drawback, this pattern imposes an unfamiliar programming style. Also, the data is eventually consistent which may not be suitable for some use cases.

9.Deployment of microservices:

Making the switch to microservices from a monolith is far easier when you have the right tools. The biggest concern for microservice developers is in finding tools that both fit their application needs and offer long-term stability and support. The list below gives a brief overview of a few popular Java Microservices deployment tools that fulfill those needs.

9.1.Docker

Docker is a top choice for many developers transitioning their applications to microservices. It relies on containers, or isolated bundles of software, databases and config files. It’s available in both free and premium tiers. According to our recent microservices survey, 61% of developers are using Docker containers for their microservices applications.

9.2.Kubernetes

Originally developed by Google, Kubernetes is now an open-source system that helps to scale, manage and automate deployment for container-based systems.

9.3.Prometheus

Prometheus is an open-source system monitoring toolkit designed to accommodate multi-container environments — making it perfect for microservices-based applications.

9.4.Micronaut

Micronaut is an open-source framework used for building modular microservices and serverless applications..

10.Benefits of microservices:

* Software built as microservices can be broken down into multiple component services. So that each of these services can be deployed and then redeployed independently without compromising the integrity of an application. That means that microservice architecture gives developers the freedom to independently develop and deploy services.
* Better fault isolation. If one microservice fails, the other will continue to work
* Code for different services can be written in different languages
* Easy integration and automatic deployment; using open-source continuous integration tools such as Jenkins, etc.
* The microservice architecture enables continuous delivery
* Easy to understand since they represent the small piece of functionality and easy to modify for developers, thus can help a new team member become productive quickly
* The code is organized around business capabilities
* Scalability and reusability, as well as efficiency. Easy to scale and integrate with third-party services
* Components can be spread around multiple servers or even multiple data centers
* Work very well with containers, such as Docker
* Complement cloud activities
* Microservices simplifies security monitoring because the various parts of an app are isolated. A security problem could happen in one section without affecting other areas of the project
* Increase the autonomy of individual development teams within an organization, as ideas can be implemented and deployed without having to coordinate with a wider IT delivery function

11.Successful case studies and examples of microservices architecture implementation:

Netflix, eBay, Amazon, the UK Government Digital Service, Twitter, PayPal, The Guardian, and many other large-scale websites and applications have all evolved from monolithic to microservices architecture. Let’s look at some of the success stories to see the result.

**11.1.Walmart successfully revitalized its failing architecture with microservices**

This is a good example of what should be done when aging architecture begins to negatively affect business. This is the multi-million dollar question which the IT Department of Walmart Canada had to address after they were failing on Black Fridays for two years in a row

The problem was that It couldn’t handle 6 million pageviews per minute and made it impossible to keep any kind of positive user experience anymore. Before embracing microservices, Walmart had an architecture for the internet of 2005, designed around desktops, laptops and monoliths. The company decided to replatform its old legacy system in 2012 since it was unable to scale for 6 million pageviews per minute and was down for most of the day during peak events. They wanted to prepare for the world by 2020, with 4 billion people connected, 25+ million apps available. So Walmart replatformed to a microservices architecture with the intention of achieving close to 100% availability with reasonable costs.

Migrating to microservices actually brought notable results:

* Conversions were up by 20% literally overnight
* Mobile orders were up by 98% instantly
* No downtime on Black Friday or Boxing Day
* The operational savings were significant as well since the company moved off of its expensive hardware onto commodity hardware
* They saved 40% of the computing power and experienced 20-50% cost savings overall

 11.2. **Spotify builds outstanding user experience with microservices**

Kevin Goldsmith, VP of Engineering at Spotify knows from his experience that an enterprise which intends to move fast and stay innovative in a highly competitive market requires an architecture that can scale.

Spotify serves more than 75 million active users per month, with an average session length of 23 minutes, while running incredibly complex business roles behind the scenes.

So, Spotify came to the conclusion that if you’re worried about scaling to hundreds of millions of users, you build your system in a way that you scale components independently. And Spotify built a microservice architecture with autonomous full-stack teams in charge in order to avoid synchronization hell within the organization. These teams are autonomous, and their mission does not overlap with other teams mission.

Now, Spotify has around 90 teams, 600 developers, and 5 development offices on 2 continents building the same product, so they needed to minimize these dependencies as much as possible.

What Spotify really likes about microservices is that they don’t have large failures; big services fail big, small services fail small. Building a microservices architecture allows Spotify to have a large number of services down at the same time without the users even noticing it. They’ve built their system assuming that services can fail all the time, so individual services that could be failing are not doing too much, so they can’t ruin the experience of using Spotify.

**11.3.Amazon embraced the DevOps philosophy with microservices**

 Amazon has also migrated to microservices.They get countless calls from a variety of applications, including applications that manage the web service API as well as the website itself, which would have been simply impossible for their old, two-tiered architecture to handle.

In 2001, the Amazon.com retail website was a large architectural monolith. It was architected in multiple tiers, and those tiers had many components in them, but they were coupled together very tightly, and behaved like one big monolith. They had a large number of developers working on one big monolithic website, and even though each one of these developers only worked on a very small piece of that application, they still needed to deal with the overhead of coordinating their changes with everyone else who was also working on the same project. When they were adding a new feature or making a bugfix, they needed to make sure that the change is not going to break something else on that project. If they wanted to update a shared library to take advantage of a new feature, they needed to convince everyone else on that project to upgrade to the new shared library at the same time. If they wanted to make a quick fix – to push out to their customers quickly – they couldn’t just do it on their own schedule, they had to coordinate that with all the other developers who have been processed changes at the same time. In the early 2000’s Amazon even had an engineering group whose sole job was to take new versions of the application and manually push it across Amazon’s production environment.  
It was literally a very long and complex process, it was hell. It was frustrating for the software engineers, and most importantly, it was slowing down the software development lifecycle, the ability to innovate, so they made architectural and organizational changes.

These big changes began on an architectural level: Amazon went through its monolithic application into a Service Oriented Architecture. Amazon also implemented changes in how their organization operated. They broke down their one, central, hierarchical product development team into small, “two-pizza teams”. They wanted teams so small that they could feed them with just two pizzas. In reality, it’s 6-8 developers per team right now.”  
Each of these teams were given full ownership of one or a few microservices. So they were defining their own feature roadmap, designing their features, implementing their features, then test them, deploy them and operate them.

After all these changes were made, Amazon dramatically improved its front-end development lifecycle. Now the product teams can quickly make decisions and crank out new features for their microservices. Now the company makes 50 million deployments a year, thanks to the microservice architecture and their continuous delivery processes.

12.Disadvantages of microservices:

**12.1**.Why not use them all the time, then? As it turns out, what works for large and complex programs does not always work at a smaller scale, and what makes sense when designing a new application does not always make sense when maintaining or updating an existing application.

When it comes to microservice architecture, complexity is perhaps the key factor. As Martin Fowler, one of the most influential thinkers in modern software development, states, "...don't even consider microservices unless you have a system that's too complex to manage as a monolith..." In other words, complexity, more than anything else, is the problem for which microservices are a solution. **If complexity isn't your problem, microservices aren't the solution.**

Microservice architecture also brings with it significant overhead in terms of design, interoperability of services, management, and use of system resources. It has a price, and for applications which cannot make sufficient use of its advantages, the price may be too high.

**12.2**.Consider, for example, a moderately large, moderately complex application being maintained by a relatively small development and operations team. If it's monolithic, the interactions between individual services may be very direct, and may be optimized for specific tasks as required. For a small development team familiar with the code, maintenance may be relatively simple. Deployment (as is often the case with monolithic applications) may be a bit awkward at times, but for the most part, it is manageable.

If the same team has to manage a microservice-based version of the same application, however, their overhead in terms of time and effort may increase significantly. Communication between microservices is more likely to be generic and protocol-based, requiring more coding time to implement even a minor change in data being handed from one service to another—and small design changes may require changes to the microservice orchestration and management system. This can put a strain on the resources of the operations team, as well as the developers.

**12.3**.Not all applications are large enough to break down into microservices. Even high-end, high-ticket desktop applications, for example, tend to be at least an order of magnitude too small for microservice architecture. A suite of applications consisting of small to medium-sized discrete services is probably already broken down as much as it needs to be, even if those discrete services contain multiple subordinate services.

Do your inventory module and your accounts payable module really need to be split up into microservices, or are they doing quite well as they are? Chances are very good that the scale at which they are currently operating is appropriate for your application. Decomposition into microservices would have the effect of adding rather than reducing complexity.

**12.4**.For most software developers, legacy code isn't just a fact of life — it is their basic, day-to-day reality. If you are maintaining an application consisting largely of legacy code, no matter how haphazard its original design may have been, no matter how ugly it may be now, you need to think long and hard before refactoring it into microservices. At what point is it in its lifecycle? Does it serve a mission-critical function, such as maintaining an irreplaceable legacy database? Are you likely to be able to replace it entirely in a few years? Or, conversely, will the process of updating or replacing it require a well thought-out, long-term strategy?

Microservice architecture may very well play a major role in updating or replacing legacy software, but that process may be long, and in the meantime, a poorly conceived attempt at converting it to microservices with no overall strategy may turn out to be an avoidable disaster.

**12.5**.Some applications by their nature require tight integration between individual components and services. This is often true, for example, of applications that must process rapid streams of real-time data. Any added layers of communication between services may slow real-time processing down. When the system needs to respond quickly to data contained in that stream (for example, input from the sensors of a self-driving vehicle), delays could be catastrophic.

In fact, embedded applications in general operate under often very tight constraints, both in terms of response time and available resources, making them unlikely candidates for microservice architecture. It is typically important to design embedded applications for simplicity of operation and optimum use of resources right from the start. Microservices, on the other hand, are largely a way of compensating for unavoidable complexity in a system where resources are not a major constraint.

13. Conclusion:

Using microservices has many benefits, but that doesn’t mean it’s the right choice for everyone. While microservices in Java simplify development, other elements like your infrastructure and team become far more complicated. Assess your existing application and development process to determine whether the transition makes sense. If you decide to move forward, do so slowly but surely. A “big bang” approach starting from scratch will only end in a major (and costly) headache; build confidence internally with a step-by-step approach via the Strangler Pattern.