Microservices: Overview, Misinterpretations and Misuses

Shiju Varghese Follow
Jul 4, 2017 · 12 min read

For the last three years, I have been working as a consultant and trainer on Golang and cloud-native distributed systems architectures including Microservices. In my consulting experience on Microservices, I have had lot of strange experiences from various clients due to the misinterpretations of Microservices. I work in India, at least here, Microservices is the most misinterpreted terminology in IT industry at this moment. Everyone is talking about Microservices. Lot of people are saying that their applications are based on Microservices architecture regardless of its merits to be called as Microservices or even called as distributed system.

Misinterpretations on Microservices

Here're the few misinterpretations, which I have heard about Microservices:

- Building HTTP services, and running it with Docker containers and clustering the containers using Kubernetes, are Microservices.
- Just using an API Gateway with service discovery and service registry, is Microservices.
- Building HTTP services with Spring Boot framework and using Netflix OSS, are Microservices. (This is coming from Java community)
- Building and running applications with Azure Service Fabric, are Microservices. (This is coming from .Net community)
- Running Serverless applications on platforms like AWS Lambda and Google Cloud Functions, are Microservices.
- Building light-weight RESTful APIs, are Microservices.

- There are lot of frameworks, which are claimed as Microservices frameworks. Building applications by using any of these frameworks, are Microservices.

The list of misinterpretations of Microservices are not end here. This is an endless list. You may use some of the technologies from the preceded list, for building your Microservices, but tied-up with some tools and frameworks to be called as Microservices don't make any sense.

Microservices is a distributed systems architecture, but development teams, those who never worked on distributed systems, are trying to learn this like learning a new framework, and building applications based on some misinterpretation, and claims that their applications are based on Microservices. You can see lot of articles titled something like Building Microservices with language X (Go, Node.js, Java, etc), Docker and Kubernetes, which may demonstrate building an HTTP service, running in a Docker container and clustering with Kubernetes, to be called as Microservices. Recently in one of my training on Microservices, an attendee who claims himself as an experienced architect, told me that he is a master on Microservices and know everything on it. During my discussion with him, I realized that he has never heard about things like DDD and Bounded Context, Event Sourcing, CQRS, gRPC, etc, and heard these things first time from me. A man who never heard about at least on domain modelling and DDD Bounded Context claims that he is a super expert on Microservices. Many of the startups to which I had discussions, claim that they're following a Microservices architecture. During the discussions with them, I realized that they call it Microservices because of their products are running with 2–5 apps, and for them each of which app is a Microservice. Their customer facing app claim as one Microservice, an admin app claim as another Microservice, a background worker claim as another Microservice, and a RESTful API claim as another Microservice. People have been coming with their own version of Microservices based on their understanding and what they are comfortable with. The sad part is even people don't realise that Microservices is a distributed systems architecture.

What's Microservices?

First and foremost, Microservices is an architectural style for building distributed systems. Microservices tackles the complexity of building distributed systems. In Microservices architecture, software systems are composed of a suite of independently

deployable, small, modular services. Each of these Microservices are built around a small business capability, which will be managed by an independent team. A Microservice is an independently deployable component of a small business capability, and Microservice architecture is a style of engineering for building highly scalable and evolvable software systems, made up of fine-grained microservices. In a nutshell, **Microservices are small, autonomous services that work together**.

In most of the time, you may start an application as a monolith in which all business capabilities are put into a single application that runs in a single process. This kind of applications are easy to develop, but it would be difficult to scale effectively because every components in the applications are tightly coupled together. In microservices architecture, we build applications by composing a suite of autonomous services where each service runs in its own process. With this approach, we can scale easier — we can give different scalability power to different services, upgrade and replace each service in isolation, and gain lot of other architectural benefits.

Many people think that Microservices is a unified architecture framework and a silver bullet that provide solutions for all problems of software engineering. So people ask questions like what's the security in Microservices, etc. Microservices is not a unified architecture framework that does not represent solutions for all problems in software engineering, but it's a way to build distributed systems where the core idea is modularisation via services. In another way, Microservices is an idea to build distributed systems, where you need to put various architecture approaches to implement this idea. Microservices is an evolution of various architecture approaches which we have been using for long time. So in a real-world Microservices, you're still using various architecture approaches which you have used in the past.

Containerisation is a great companion to build and run your Microservices. If you can package your Microservices in containers, it gives you immensive capabilties to scale and dynamically orchestrate your Microservices without depending on where it runs. Packaging tools like Docker, rkt, and orchestration tools like Kubernetes, are great technologies for containerising Microservices.

DDD Aggregates and Bounded Context for Decomposition of Microservices

When you're moving to Microservices architecture from a monolithic architecture, the big question is how to decompose the Microservices. How to broke up a larger software

system into functional components?. What's the size of a Microservice? The answer is very simple: domain modelling. Sam Newman, on his book titled "Building Microservices" says all roads to Microservices pass through domain modelling. The book titled "Domain-Driven Design: Tackling Complexity in the Heart of Software" by Eric Evans, published in 2003, is a classic book on domain modelling, which provides guideline for building complex software systems based on domain model. Ever since the release of this book, the term Domain-Driven Design (DDD), was widely accepted by the community and have been using DDD as the way to build oftware systems. DDD, introduced various building blocks such as *Entity*, *Value Object*, *Services*, *Repositories*, *Aggregates* and *Bounded Context*.

Aggregates and Bounded Contexts are the building block to build your Microservices, which will decide the size of a single Microservice. Typically business entities like *Order*, *Customer*, *Account* are Aggregates, which represent a graph consisting of a root entity and one or more other entities and value objects, and can be treated as a unit. A better way to perform a transaction in Microservices is to perform persistence against aggregates. Bounded Context is a central pattern in Domain-Driven Design, which provides logical separation of business problem into various sub domains by dividing a large model into different Bounded Contexts. A Bounded Context encapsulates the details of a single domain, a sub domain of large domain model, which can be partitioned into a single Microservices. A common strategy for building Microservices is to build it against each Bounded Context. Each Microservice uses its own database to persist data of domain model of a Bounded Context. A single Bounded Context can include many aggregate roots, or we can organise a single aggregate root into a single Bounded Context. An *Order* aggregate root, consists of entities such as *OrderLine*, *Customer* and *Product,* and value objects such as *ShippingAdress, PaymentMethod etc.* Within the graph of *Order* aggregate root, *Customer* entity can also be treated as a aggregate root because it can organise as a root entity for getting all information for a customer. Aggregates and Bounded Contexts are important concepts in Microservices architecture, which are the foundational block to build your Microservices, which will also decide the size of your Microservices. In short, **Microservices are autonomous services around Bounded Context**.

Challenges in Microservices

Microservices architecture is indeed a great approach for building distributed systems. But keep in mind that Microservices is neither a silver bullet nor an easy approach. It has never been an easy job to build distributed systems, even with Microservices although it tackles the complexity of building distributed systems.

When you move to Microservices architecture from monolithic applications, you need to solve many practical challenges. For an example, a business transaction may span into several Microservices because we broke up a monolithic system into several autonomous services based on Bounded Context. A transaction may need to perform persistence into many Microservices where you need to manage data consistency. And another challenge is querying data from multiple databases. With a monolithic, you can easily perform inner join queries from a single database. Because the monolithic database has been moved into several databases as part of the decomposition of functional components, you can't simply execute inner joins, thus you must get data from multiple databases. Here you don't have any centralized database.

Building Scalable Microservices with Event Sourcing and CQRS

DDD Aggregates and Bounded Contexts are the foundational building blocks of Microservices, but when you're going to choose a concrete architecture for building your distributed system to solve practical challenges of Microservices, an event-driven reactive system on DDD aggregates would be a great approach. For this, I highly recommend to use Event Sourcing, which is an event-centric architecture to construct the state of an application by composing various events. Event Sourcing deals with an event store of immutable log of events, in which each log (a state change made to an object) represents an application state. Because every state change in the application, is treated as an immutable log, you can easily troubleshoot the application and can also going back to a particular version of application state at any time. An event store is like a version control system. In a Microservices architecture, we can persist aggregates as a sequence of event. Events are facts, which represent some actions happened in the system. These are immutable, which can't be changed or be retracted. If you would like to make a change in the system, do persist logs into the event store to represent an another set of events. The example of events are *OrderCreated, OrderApproved, OrderShipped, OrderDelivered, etc.* In your Event Sourcing architecture, when you publish one event from one Microservice, other Microservices can be reactive to those events and publish another set of events. Sometimes, the sequence of events can be

compared with Unix pipes. A single transaction in a Microservices system may span into multiple Microservices where we can perform a transaction as a sequence of events by building reactive Microservices. Each state change of an aggregate can be treated as an event, which is an immutable fact about your system. In order to publish events to let other Microservices know about something has happened in our system, we can use messaging systems like Apcera NATS, Kafka, RabbitMQ, etc. My personal choices are Apcera NATS and Google Cloud Pub/Sub. An event-driven, reactive architecture is a great choice of architecture approach for building massively scalable Microservices.

When you make persistence as a sequence of events by using an Event Sourcing, you may need an architecture approach to make queries for your Microservices. An architecture pattern, Command Query Responsibility Segregation (CQRS) is an ideal pattern for implementing queries for your Microservices. As the name implies, CQRS segregates an application into two parts: Commands to perform an action to change the state of aggregates, and Queries to provide a query model for the views of aggregates. We may also use different databases for both write operations and query operations. This will also allows you to make highly performant query model by loading denormalised data sets into data stores of read models. NoSQL/NewSQL databases are great options to store the data of read models.

Although Event Sourcing and CQRS, are great patterns for implementing a distributed system using Microservices architecture, it's not a silver bullet and it has its own limitations. You may need different architecture style for building some kind of Microservices systems. But generally, I feel that Event Sourcing, paired with CQRS, is a great approach for implementing Microservices systems.

Inter-Process Communication of Microservices over APIs using gRPC

In Microservices architecture, you may need to make lot of inter-process communication between Microservices. Here're the two options you may use to make inter-process communication between Microservices:

- Asynchronous event-driven architecture with a messaging system
- Build high performance APIs at scale for millions of API calls per second

By using an Event Sourcing architecture with a messaging system, you can implement an asynchronous event-driven architecture to manage the state of aggregates. You may also need to make APIs to communicate between Microservices. When you make APIs to perform inter-process communication between Microservices, performance and scalability are very important things. You should not feel that there are lot of communications are happening over networks by building high performance APIs. When you build massively scalable systems, JSON based RESTful APIs are not a good option because of the performance challenges and the lack of capability to expose domain-specific operations as APIs as RESTful systems are working over the concept of resources. Here, gRPC, a high performance, open-source remote procedure call (RPC) framework, can be used for building massively scalable APIs for the inter-process communication between Microservices. By default, gRPC uses Protocol Buffers as the Interface Definition Language (IDL) and as its underlying message interchange format. gRPC is an open source version of Google's internal framework, Stubby, which is used to scale 10 billion API calls per second. gRPC can be called as a protocol for inter-process communication in Microservices architecture over APIs.

Misinterpretations and Misuses of Microservices

As I mentioned in the beginning of this post, there're lot of misinterpretations on Microservices, are coming from various developer communities. I have been getting lot of requests for conducting a workshop on Microservices from various clients, and most of the time, clients are coming with a course agenda, which is really shocking as it's not about anything on Microservices. Lot of developer communities are tied-up Microservices with a some frameworks and tools. Lot of Java people believe that building RESTful APIs with Spring Boot and leveraging some tools of Netflix OSS is Microservices while some people from .Net community say that running applications with Azure Service Fabric is Microservices. Even I have been asked questions like what's the difference between a RESTful service and a Microservice, as many people think that building some light-weight RESTful APIs are Microservices. For some other people, building those APIs using Spring Boot is Microservices. I don't know from where the Spring Boot came to the picture as a misinterpretation to be called as Microservices.

I feel that people just heard the hype and associating it with some tools and frameworks without understanding the soul and purpose of this evolutionary architecture. The situation is very similar like architecture patterns like SOA and some engineering

practices like Agile. I know lot of companies are claiming that they're following agile engineering because they're just following Scrum process even though they're building applications with a highly conventional manner. For the sake of TDD, lot of organisations are writing unit tests after writing the production code, and then just trying for getting a target test coverage, to be called as engineering applications with TDD. Even though it is a minor community, but lot of people are adopting technologies, patterns and practices just for the sake of technologies and to be called as something.

Microservices is indeed great architectural style for building massively scalable applications. Microservices is also well suited for building internet-scale applications like Netflix, Uber, Amazon, eBay, etc. But this is not for everyone and it's not a silver bullet. Although Microservices is a right architectural approach for building massively scalable applications including public face internet-scale applications, it might be a wrong choice for most of other kind of applications, especially for building enterprise applications, which has a complex domain model. Personally, I don't recommend Microservices for a complex business applications, which has very complex domain model. While Microservices might be an ideal for a massively scalable application in which domain model is not much complex, and scalability is the most challenging factor to be called as a successful application. For some kind of applications, a Microservices architectural style of implementation may create lot of performance problems and add complexities to the sytem and finally there is a chance to failure. Adopting Microservices for the sake of its hype may make lot of side effects to your system.

Improving Monolithic Applications

In my consulting experience, I realised that Microservices architecture is not required for most of the use cases, and a middle-grounded solution is better approach to engineering most of the applications. You can adopt a hybrid approach to solve your problems. Your company may not have the muscle power of engineering capabilities as organisations like Google, Amazon, Netflix, Uber, eBay, Square, etc. Your problem is unique within your organisation. An architecture approach should solve your problem and use it for solving your own unique problems instead adopting something for the sake of that.

You can improve your existing monolithic applications in different ways: splitting your monolithic application into multiple systems without blindly following the Microservices guidelines, but with a hybrid architecture approach and with a DevOps

culture and a modern CI/CD pipeline. The great benefit of Microservices is modularisation. When you build new applications, you can architect systems by applying modular system design principles without a Microservices architecture in front of you. For example, the package ecosystem of Go programming language lets you design your applications with better modularity. This approach will also enable you easier migration to Microservices if you would like to move to Microservices later on for scaling modules and teams independently. You don't need to start a new application with Microservices. It is better to start as a monolith application by applying modular system design principles, and when your system and teams are evolving, you can move to Microservices if it is really essential.

Keep in mind that your company and products are unique. Instead of becoming a poster child of technologies, patterns and practices, solve your own unique problems by using middle-ground solutions. Adopting technologies for the sake of technologies don't make any sense.

You can follow me on twitter at @shijucv. I do provide training and consulting on Go programming language (Golang) and distributed systems architectures, in India.

Microservices Domain Driven Design Distributed Systems