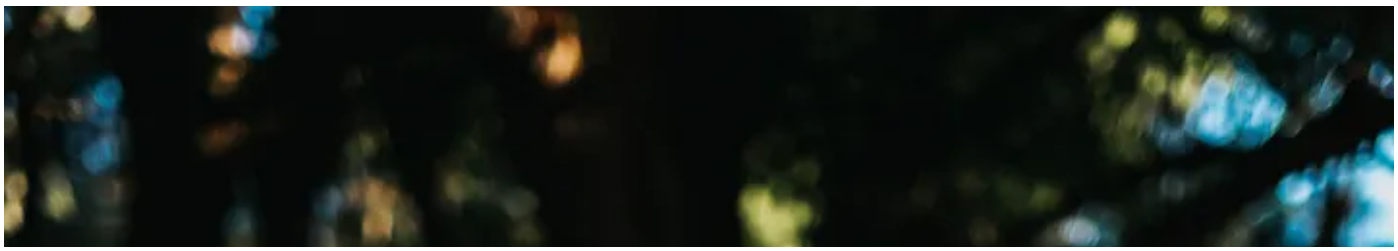
Microservices: Architectural building blocks and trends

Satyajit Panda Follow 13 min read · Feb 6, 2020



Medium Search Write I

Photo by Nathan Dumlao on Unsplash

In this digital age technology based on sound software architecture is a primary foundation for a successful business. The most important value driver of software architecture is the ability to experiment with end users by introducing a new business idea into production quickly.

Technology itself is not a silver bullet for problem-solving, but it provides different approaches for problem-solving. Microservices architecture is one such approach in technology for server-side service design. It has gained popularity due to successful adoption by large internet consumer companies like Amazon, Facebook, Netflix, Twitter, etc. by helping them solve certain problems. They operate on a massive scale and are consistently on the top of their game. A laser-sharp focus on innovation is supported by engineering practices like a short release cycle of features through automation of build-test-deploy pipeline. A/B testing, gamification, and other techniques are used to get quick customer feedback and pivot the feature releases accordingly. Microservices architecture acts as an enabler for the above and also helps in areas like on-demand scaling, effective resource allocation, an independent stream of development and deployment, choice of polyglot runtime environments, etc.

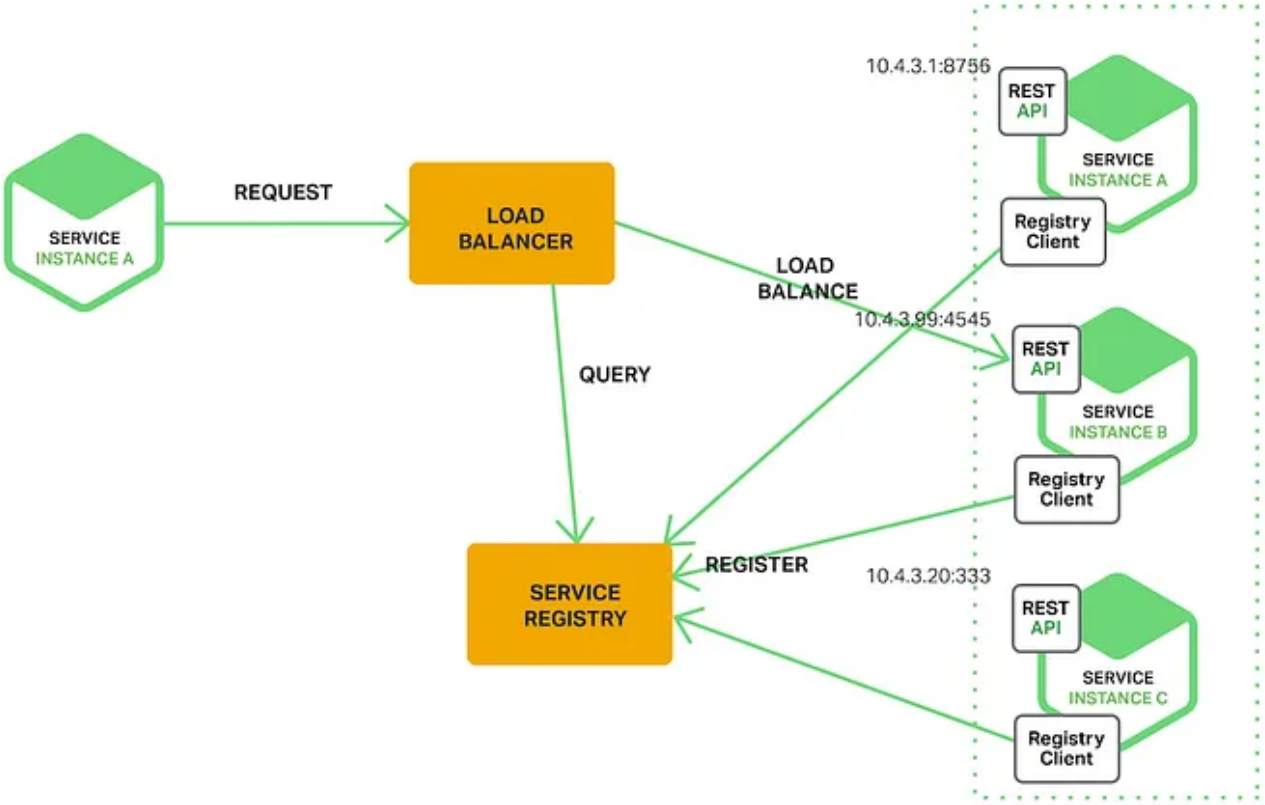
Many excellent articles are detailing the promises of microservice architecture. This article is an attempt to provide a technical overview of core architectural building blocks for successful microservices architecture and the latest trends in that space. It is based on author's and his industry peers' experience of developing microservices-based systems for more than the last half of the decade in both cloud native and cloud agnostic way. There is no single way or standard to develop a

successful microservices architecture and every team's mileage varies in terms of approach and execution.Microservices design patterns, storage, network design are the other critical aspects left out of this article due to lack of space.

Microservices architecture is a distributed server-side architecture based on loosely coupled services that are designed, developed, packaged, build, tested, deployed and scaled independently. The twelve-factor app is a methodology which nicely complements and helps to leverage the value of microservices architecture.

1. Service registration and discovery

Services need to find each other. They also have location transparency in a distributed architecture where services with short life cycles can reside in different containers/virtual machines/bare metals with continuously changing IP addresses. Service registration and discovery component allow services to register themselves by bootstrapping during start-up. They are found by other services and also find other services through the discovery server.

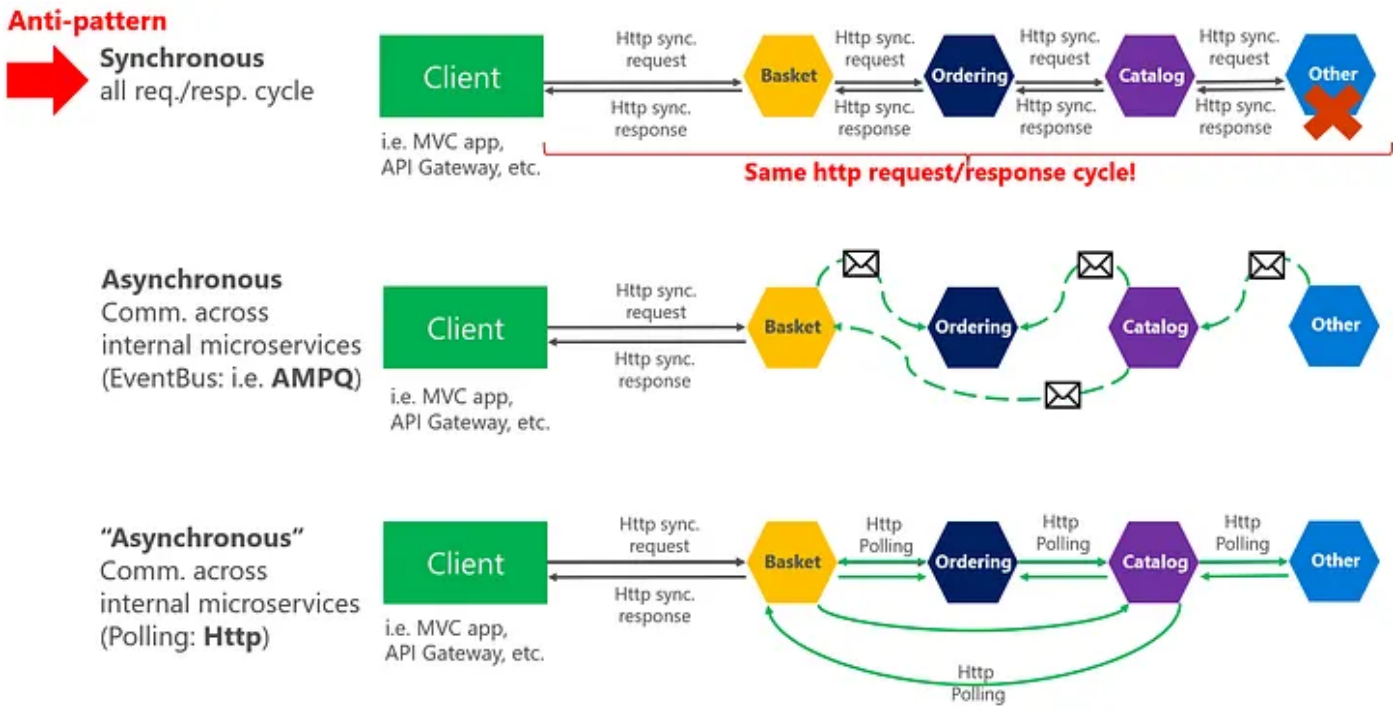


Service registration and discovery (source: nginx.com)

Hashicorp Consul, Netflix Eureka, Etcd are few of the examples of service registration and discovery servers.

2. Service to service communication

A service communicates with other services for the execution of a functionality. There are multiple ways in which service to service communication can happen.



Synchronous vs. asynchronous messaging (source: microsoft.com)

2.1. Asynchronous mode using a message queue or polling

It has two types of approach; one is push based using a message queue and another is poll based using HTTP.A service communicating to other services through message queue keeps the services decoupled, keeps them asynchronous, can scale out as per load, provides message resiliency, etc. The downside to this approach is an extra message queue component to maintain. RabbitMQ, Kafka is leading open source message brokers and there are managed services options for message queues from public cloud providers like AWS, Azure, GCP, etc.

A poll-based approach has the downside of wasting compute cycles while it is waiting for response.

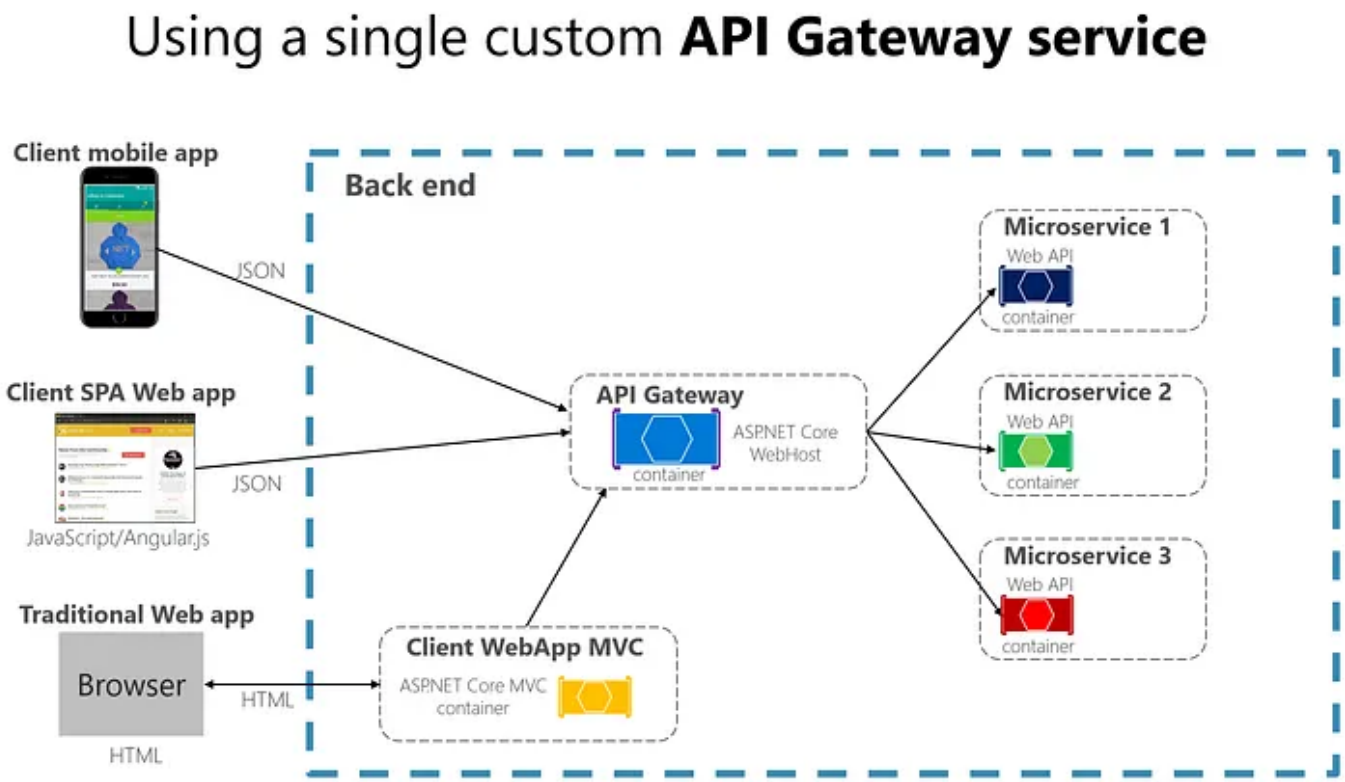
2.2. Synchronous mode of direct service to service communication

It is the simplest approach as it doesn't need any additional components. The downside is the heavy coupling and cascading effect of change

between the services.

2.3. Through a proxy or API gateway

This approach is used when we want to expose services in a centralized and controlled manner. Services that are opened to the outside world to be used by external clients also use this pattern. The downside is another component to maintain. Kong, Zuul and cloud-based options like AWS API gateway, Google cloud apigee, Azure API management are used here.



API gateway(source: Microsoft.com)

No matter which style of the communication we choose, we still need to deal with the resilience issues of distributed systems. We can use resilience handling techniques like request retries with exponential backoff, failure handling, circuit breaking, back pressure, bulkhead, isolation of faults to improve the resilience of the system.

There are components and libraries available which provide these features like spring cloud Netflix, Alibaba, etc. Recently there is a rise in using service mesh components for the same due to their non-intrusive approach of residing as a sidecar along with the microservices component versus being embedded inside the service codebase like the above-mentioned framework and libraries. Leading players in this space are Istio, Linkerd, Envoy.

3. Service configuration

Each service may use some configuration data like file paths, date and time formats, environment-specific values, etc. As the number of services increases; creating, storing, updating, deleting the configurations becomes nontrivial and time-consuming. A configuration server component helps in storing and managing these configurations.

Hashicorp Consul, Spring configuration server are few of the examples of configuration servers. We can also create our configuration servers by using a database.

4. Secrets management

Each service uses some infrastructure and application-level highly sensitive data like credentials, connection strings, API private keys, etc. which are not suitable to store in a version control system like git or a database. A secret manager component helps to store, managing those secrets. In some cases, it also helps in the creation of the secrets.

Lambda based microservice with secret manager provided DB credentials (source: aws.amazon.com)

Hashicorp Vault is a great open-source option for a secret manager.AWS Secrets Manager, AWS Parameter Store, Azure Key Vault, GCP Cloud KMS are software-based options for a secret manager. Hardware security modules (HSM) like GCP Cloud HSM, Azure Key Vault and AWS Cloud HSM is provided by cloud providers to strengthen secret management.

5. Observability

Logs, metrics, traces are called the golden triad of service observability which helps us to understand microservice execution behavior from end to end. An effective way to observe microservices' behavior is to also understand the whole systems' behavior, where we monitor the infrastructure and other components of the whole system along with the microservices. Application/cloud performance monitoring products like Dynatrace, Datadog, Newrelic, AppDynamics can be augmented with the above approaches to collect, monitor and analyze an end to end microservices system.

Call graphs of microservices in several organizations

5.1. Centralized Logging

A software system produces logs like application logs, system logs, or access logs. The log is an ordered, append-only data structure. A logline in its minimalistic form is a minimum is a timestamp and an event typically appended to the end of a file. A software system based on microservices architecture typically produces multiple logs from multiple components.

These logs are picked by file uploaders like Logstash and transferred to centralized storage like Elasticsearch or Splunk and then queried and visualized by dashboards like Kibana, Grafana, etc.

Humio is an interesting take in this space that uses an indexless approach during data ingestion so we can search the logs in real-time. It is super-fast and scalable.

5.2. Distributed Tracing

Microservices architecture is a distributed architecture where services are located in different machines, network or even data centers. One of the challenging issues with microservices is to trace the execution of requests through multiple distributed components.

It becomes exponentially difficult when the number of components is large. To create a trace of the flow between components, the application code inside the service component is instrumented along with a correlation id. A correlation id or a trace id stitches all components together which is part of a transaction, workflow or an event chain allowing the end to end visibility of the flow.

Jaeger, which is based on Google's Dapper research paper, Zipkin are popular opensource frameworks for trace collection and monitoring. OpenTracing is an open standard that provides a vendor-neutral instrumentation framework.

5.3. Metrics

To understand the behavior of the services and systems, metrics reporting against time is used. Graphed, time-series metrics helps to get the behavioral snapshot of the system at a glance after new changes are

introduced into the system in terms of code change or system upgradation.

Counter, gauge, meter, histogram, timer is some of the metrics used to determine services and systems behavior. Histogram and timer are majorly used as they measure stuff across time and records percentile values which are the most effective way to measure services and systems behavior. They can be used to track times for the max, min, 95th percentile, nth percentile, etc. for a flow. These can be stored using time series databases like Prometheus, OpenTSDB or InfluxDB and visualized by dashboards like Kibana, Grafana, etc.

6. Continuous integration and continuous deployment (CI/CD)

Continuous integration and continuous deployment are not specific to microservices architecture but are central to the faster idea-to-production cycle. Having a running automated CI/CD pipeline is a prerequisite to realize the benefits of microservices architecture.

Continuous delivery and integration for docker based microservices (source: dzone.com)

Jenkins, TeamCity, Circle CI are the most popular open-source software for continuous integration. Octopus, Concourse, GoCD are some of the popular choices for continuous deployment. GitLab is an interesting take in this space with an end to end offering for CI/CD.

7. Infrastructure

Automation of infrastructure provisioning is another prerequisite of a microservices architecture. Microservices architecture by itself is hardware, operating systems and cloud computing agnostic. Whether to run it in our data center or the cloud is an architectural decision that depends on many trade-offs specific to organization, team and the industry in which it is operating. We may leverage cloud computing to augment capabilities like rapid infrastructure provisioning, process and storage scaling, software-defined network, network automation and innovations like AI/ML, IoT, Data pipelines, etc.

Terraform, Chef/Puppet, Salt, Ansible are popular open space tools in this space. The major cloud providers like AWS have CloudFormation, GCP has Google Cloud Deployment Manager, Azure has Microsoft Azure Automation for infrastructure automation. Pulumi has an interesting no-YAML and languages SDK based approach to infrastructure as a code.

8. Packaging

Application packaging is the result of a build cycle and the package is the deployable unit in a test/stage/production environment. Its format is generally dependent on the language runtime we are using. For example, we package a java service into a fat jar containing the web server responsible to host that jar. Docker has standardised the application packaging where any application can be packaged into a docker image

along with the language runtime and operating system. Building a docker image can be part of the CI/CD pipeline. A docker repository, either self-hosted or hosted in the cloud can be used to store and retrieve these images.

9. Deployment

In non-docker environments, the deployable package containing microservice gets deployed on a virtual machine or a bare metal machine along with the web server.

In the case of docker deployable packages, they are deployed in a docker-compose or Kubernetes based platform which is responsible for docker container runtime orchestration. The orchestration platforms are responsible for container life cycle management, auto-scaling, health checks, etc. The Kubernetes platform can be self-hosted in our own data center or in a managed services environment provided by cloud providers like AWS EKS, Google GKE, AZURE AKS, etc. A step further in that direction is AWS Fargate and Google CloudRun where the abstraction moves upward to containers and we don’t need to deal with Kubernetes directly.

Docker based packaging and automated deploy in AWS (source: cloudacademy.com)

Recently code only, event-driven, function as a service, serverless platforms like AWS lambda, Google cloud functions, Azure functions have picked up and are at a stage where they can be part of a hybrid architecture and augment microservices architecture to pick some workloads.

10. Security

Microservices security is achieved by following multiple approaches based on solid security principles like zero trust with least privileged access. Authentication mechanisms like Oauth, JWT, SAML are used to secure the microservices with the use of validated tokens. Symmetric encryption techniques like AES and asymmetric encryption techniques like RSA is used with encryption key up to 256 bits to encrypt the data in rest and data in transit. It helps protect the data from the man in the middle attack and data corruption. Clearly defining the microservices boundaries and putting SSL initiation and termination at appropriate

points helps to have a balance between security and performance. Role-based access for each microservices helps to protect it from privilege escalation attacks. In container-based microservices architecture based, security is taken care of by signing the images, scanning the images for vulnerabilities, hardening the base OS of containers, scanning containers for unnecessary port exposure, etc.

Microservice security with JWT tokens (source: Kasun)

Systems security being an end-to-end endeavor the above approaches needs to be well supported by other techniques like OS and network security, auditing, threat monitoring, incident reporting, etc. to achieve a foolproof security in a microservices setup.

11. Testing

The testing needs of microservices are different than the traditional architectures. This is mainly due to the distributed nature of the microservices architecture and a large number of integrations between components.

For distributed load testing tools like Locust.io and flood.io are popular which scales up to millions of requests and also provides geographically segregated load testing. Integration testing is effectively handled through consumer-driven contract (CDC) testing using tools like pact and CDC tests perform better than the end to end(E2E) and mock tests in terms of isolation, feedback time, etc.

The design principles of modularity and isolation in microservices help in canary testing of the system for deployment and migration scenarios. Chaos testing helps to simulate failure scenarios by introducing chaos into the system in a controlled way to make the system robust by finding failure points early.

Deciding the amount of testing is required is always a contention point in software testing. The answer lies in choosing our priority of speed of assembly line (CI/CD) vs the quality of the final product.

12. Protocols

The internet and external-facing microservices use document-centric protocols like HTTP(S)1.1/2.0. The internal microservices communicate between themselves in binary RPC protocols like gRPC and Rsocket for being performant. REST on HTTP(S) is the dominant architectural style for microservices. There is a rise of interest in graph query languages like GraphQL which reduces network chattiness and improved latency by specifying exactly needed information. The emergence of the latest binary and multiplexing HTTP protocol HTTP 2.0 as a dominant protocol of the internet will provide designers of microservices more options.

There are some design decisions like traffic routing, observability, fault

injection which are pushed to either L3-L4(network-transport) or L7(application) layers of the OSI stack depending upon the design context.

13. Data format

The data format of the data being passed around microservices can be textual like plain text/JSON/XML or binary like Google Protocol buffers (ProtoBuf), Apache Avro and Microsoft Bond protocols. The former provides ease of use and readability. The later uses serialization and deserialization and structured as per a schema that provides greater interoperability while being more performant.

14. Workflow/Orchestration engines

Most software systems have varied types of workloads like real-time, synchronous/asynchronous and batch. A microservices architecture needs a workflow manager or orchestrator to coordinate these workloads. The workflows can be authored as directed acyclic graphs (DAGs) of tasks. One of the major use cases for a workflow engine is to handle the SAGA pattern for distributed transactions by providing a mechanism for transactions, checkpoints, failure retry scenario and compensating transactions. They make it easy to visualize task pipelines running in production, monitor progress, pinpoint and troubleshoot issues when needed.

Orchestration-based SAGA (source: microservices.io)

Apache airflow, Netflix Conductor, Uber Cadence are very well designed, and battle-tested workflow managers capable of scaling on demand and handling multiple workloads. Cloud providers like AWS have step functions, Azure has logic apps.

15. What is next?

There are many upcoming and mature cloud-native microservices frameworks like Microsoft DAPR, Quarkus, Netifi, Spring cloud, Lagom, Micronaut, etc. which helps in rapid prototyping with many built-in blocks. Reactive frameworks like Spring WebFlux, Vert.x, Akka, RxJava, etc. helps building a reactive flow where the system resources are used to its maximum. Docker has emerged as a clear winner in the container war and the container orchestrator platform Kubernetes has emerged as a primary base platform for the microservices architecture based on docker.

Cloud Native Compliance Foundation (CNCF) is an ongoing effort to provide a bunch of software components for each of the aspects of the microservices architecture.

Application services architecture is on an evolutionary path from the age of modular monoliths to microservices to function as a service/serverless providing lots of design choices to architects. Also due to the availability of robust cloud-based managed services and SAAS options some features can be realized through them. Serverless is picking up and new design patterns are emerging to do the heavy lifting for complex architecture.

Artificial intelligence-based DevOps platform will continue to provide more insights and data on these platforms to improve themselves. Low code and no-code platforms will help developers to reduce their development time so that they can focus on the more important aspects of application.5G with its supersonic speed is expected to bring changes to the architectural styles. Edge computing with its decentralized structure will continue pushing the architectural boundaries resulting in new microservices design patterns that can take advantage of the decentralized edge. Microservices is a developer-centric architecture vs UX i.e. end-users don't care about the microservices, an increasing design thinking approach will be applied to make it easier for developer adoption.

16. Conclusion

Microservices architecture is not a panacea for every architectural problem. It is meant to develop and maintain highly scalable, available, resilient, secure, automated, self-healed distributed systems.

When starting to define a software architecture we should target a clean modular monolith-based design as a rule of thumb. It gives us the advantages of speed and the ability to gain quick feedback from users. Also, a well-designed modular monolith can handle the load for up to

10000 users in most cases. As the concept gains acceptance amongst the users and the number of users increases the monolith can be decomposed into microservices and each of the above microservices architectural building blocks can be added gradually. With the increased cloud computing adoption, the microservices architecture is becoming central to cloud-native application development.

. . .

“A successful microservices journey requires serious software engineering rigour and commitment to engineering excellence, buy-in and support from a valid ROI driven business case, and having lots of fun! “

Microservice Architecture | Scalability | Ci Cd Pipeline | Cloud Computing | AWS

Written by Satyajit Panda

6 followers · 20 following

Follow

No responses yet

I Ishu

What are your thoughts?

More from Satyajit Panda

Satyajit Panda

Multi-cloud adoption in enterprises:An introduction

Introduction Cloud computing is mainstream now and it is the major driver...

Oct 11, 2019 3

Satyajit Panda

Smart process and human in the loop

Why Human in the loop is important to create Smart Processes?

Apr 2, 2018 1

See all from Satyajit Panda

Recommended from Medium

In Data, AI and Beyo... by Julius Nyerere Nyam...

Generate AWS architectural diagrams using this simple...

A combination of LLMs and MCP Servers.

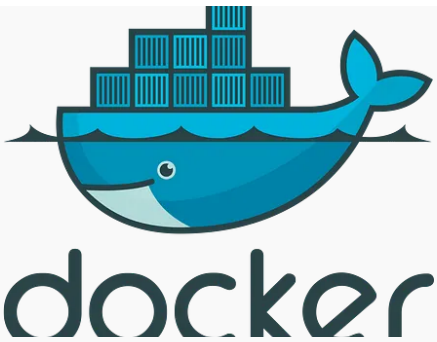
Oct 14 100 2

In Netflix TechBlog by Netflix Technology Blog

Post-Training Generative Recommenders with Advantage...

Author: Keertana Chidambaram, Qiuling Xu, Ko-Jen Hsiao, Moumita Bhattacharya

4d ago 55 2



Abhinav

Docker Is Dead—And It's About Time

Docker changed the game when it launched in 2013, making containers...

Jun 9 7.1K 200

The Latency Gambler

I Interviewed 20+ Engineers. Here's Why Most Can't Code

Over the past year as a Senior Software Engineer at a B2B SaaS company, I've...

Sep 9 2.9K 95

In Level Up Coding by Fareed Khan

Building an Agentic Deep-Thinking RAG Pipeline to Solve...

Planning, Retrieval, Reflection, Critique, Synthesis and more

Oct 20 1.4K 16

Nivetha Thangaraj

🪦 Kubernetes Is Dead: Why Tech Giants Are Secretly Moving to...

By Nivetha Thangaraj

Jul 1 763 100

See more recommendations