**Scalability/High Availability**

**In order to build a scalable application, we need to design for concurrency and partitioning: concurrency allows each task to be broken up into smaller pieces, while partitioning is essential for allowing these smaller pieces to be processed in parallel. So, while scalability is related to how we divide and conquer the processing of tasks, performance is the measure of how efficiently the application processes those tasks.**

 In simplest terms, highly available architectures typically involve distribution of compute resources, load balancing, and replication of data

High Availability is feature you provide to your Service API. There are some fundamental principles/pattern which will ensure high availability.

1. **Circuit Breaker**
2. **Retry**
3. **Health Endpoint Monitoring**
4. **Throttling**
5. **Queue-Based Load Leveling pattern**

**OR**

Google Cloud provides tools and services to help you build highly available and resilient apps:

* Google Cloud services are available in [regions and zones](https://cloud.google.com/docs/geography-and-regions#regions_and_zones) across the globe, enabling you to deploy your app to best meet your availability goals.
* Compute Engine instance groups and GKE clusters can be distributed and managed across the available zones in a region.
* Compute Engine [regional persistent disks](https://cloud.google.com/compute/docs/disks#repds) are synchronously replicated across zones in a region.
* Google Cloud provides a range of [load-balancing options](https://cloud.google.com/load-balancing) to manage your app traffic, including global load balancing that can direct traffic to a healthy region closest to your users.
* Google Cloud's [serverless platform](https://cloud.google.com/serverless) includes managed compute and database products that offer built-in redundancy and load balancing.
* Google Cloud [supports CI/CD](https://cloud.google.com/docs/ci-cd) through native tools and integrations with popular open source technologies, to help automate building and deploying your apps.
* Cloud Monitoring provides metrics across your apps and infrastructure, helping you make data-driven decisions about the performance and health of your apps

### Physically distribute resources

### Google Cloud services are available in locations across the globe. These locations are divided into [regions and zones](https://cloud.google.com/docs/geography-and-regions#regions_and_zones). How you deploy your app across these regions and zones affects the availability, latency, and other properties of your app

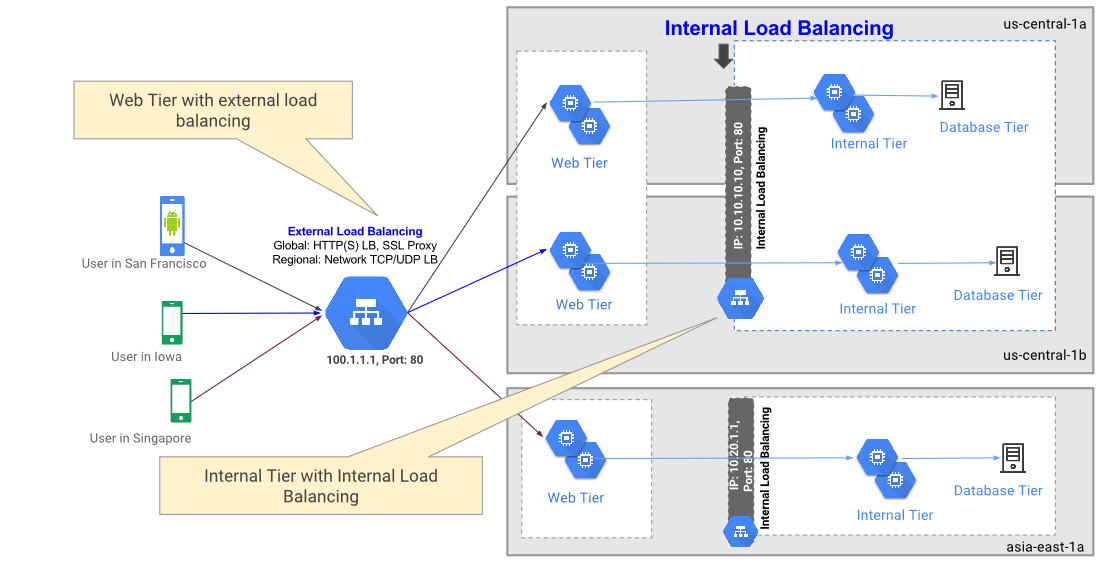
### 2. Favor managed services

### Rather than independently installing, supporting, and operating all parts of your application stack, you can use managed services to consume parts of your application stack as services. For example, rather than installing and managing a MySQL database on virtual machines (VMs), you can instead use a MySQL database provided by [Cloud SQL](https://cloud.google.com/sql). You then get an availability [Service Level Agreement (SLA)](https://cloud.google.com/sql/sla) and can rely on Google Cloud to manage data replication, backups, and the underlying infrastructure. By using managed s

### Load-balance at each tier

Rather than independently installing, supporting, and operating all parts of your application stack, you can use managed services to consume parts of your application stack as services. For example, rather than installing and managing a MySQL database on virtual machines (VMs), you can instead use a MySQL database provided by [Cloud SQL](https://cloud.google.com/sql). You then get an availability [Service Level Agreement (SLA)](https://cloud.google.com/sql/sla) and can rely on Google Cloud to manage data replication, backups, and the underlying infrastructure. By using managed services, you can spend less time managing infrastructure, and more time on improving the reliability of your app.

Many of Google Cloud's managed compute, database, and storage services offer built-in redundancy, which can help you meet your availability goals. Many of these services offer a *regional* model, which means the infrastructure that runs your app is located in a specific region and is managed by Google to be redundantly available across all the zones within that region. If a zone becomes unavailable, your app or data automatically serves from another zone in the region



## **Monitor your infrastructure and apps**

[Cloud Monitoring](https://cloud.google.com/monitoring) is Google Cloud's integrated monitoring tool. Cloud Monitoring ingests events, metrics, and metadata, and provides insights through dashboards and alerts. Most Google Cloud services automatically send [metrics](https://cloud.google.com/monitoring/api/metrics) to Cloud Monitoring, and Google Cloud also supports many third-party sources. Cloud Monitoring can also be used as a backend for popular open source monitoring tools, providing a "single pane of glass" with which to observe your app.

### Monitor at all level

### - Infrastructure monitoring

### - App monitoring.

#### -Service monitoring.

#### **-End-to-end monitoring**

Ref - <https://konghq.com/blog/microservices-high-availability/>

<https://cloud.google.com/solutions/scalable-and-resilient-apps>

# **Best Practices for Microservice Performance**

## 1. **Provide batch APIs**

In addition to CRUD-style APIs, you can still provide good microservice performance for groups of entities by providing batch APIs. For example, rather than only exposing a GET API method that retrieves a single user, provide an API that takes a set of user IDs and returns a dictionary of corresponding users:

**Request:**

/user-service/v1/?**userId=**ABC123&**userId=**DEF456&**userId=**GHI789

**Response:**

{

"ABC123": {

"userId": "ABC123",

"firstName": "Jake",

… },

"DEF456": {

"userId": "DEF456",

"firstName": "Sue",

… },

"GHI789": {

"userId": "GHI789",

"firstName": "Ted",

… }

}

The App Engine SDK supports many batch APIs, such as the ability to fetch many entities from Cloud Datastore through a single RPC, so servicing these types of batch APIs can be very efficient.

## 2 **Use asynchronous requests**

Often, you will need to interact with many microservices to compose a response. For example, you might need to fetch the logged-in user's preferences as well as their company details. Frequently, these pieces of information are not dependent on one another and you could fetch them in parallel. The Urlfetc

App Engine SDK supports asynchronous requests, allowing you to call microservices in parallel.

## 3. **Use the shortest route**

Depending on how you invoke Urlfetch, you can cause different infrastructure and routes to be used. In order to use the best-performing route, consider the following recommendations:

**Use**[***REGION\_ID***](https://cloud.google.com/appengine/docs/standard/java/microservice-performance#appengine-urls).r.appspot.com**, not a custom domain**

A custom domain causes a different route to be used when routing through the Google infrastructure. Since your microservice calls are internal, it's easy to do and performs better if you use https://***PROJECT\_ID***.[***REGION\_ID***](https://cloud.google.com/appengine/docs/standard/java/microservice-performance#appengine-urls).r.appspot.com.

**Set**follow\_redirects**to**False

Explicitly set follow\_redirects=False when calling Urlfetch, as it avoids a heavier-weight service designed to follow redirects. Your API endpoints should not need to redirect the clients, because they are your own microservices, and endpoints should only return HTTP 200-, 400-, and 500-series responses.

**4.Prefer services within a project over multiple projects**

There are good reasons to use multiple projects when building a microservices-based application, but if performance is your primary goal, use services within a single project. Services of a project are hosted in the same datacenter and even though throughput on Google's inter-datacenter network is excellent, local calls are faster.

# practices for microservices security

## 1.Use OAuth for user identity and access control

## 2. Use 'defence in depth' to prioritize key services.

 the Defense in Depth mechanism is basically a technique through which you can apply layers of security countermeasures to protect the sensitive services. So, as a developer, you just have to identify the services with the most sensitive information and then apply a number of security layers to protect them. In this way, you can make sure that any potential attacker cannot crack the security on a single go, and has to go forward and try to crack the defense mechanism of all the layers.

3. . Use automatic security updates

4. Use a distributed firewall with centralized control

## 5. Get your containers out of the public network.

6. Use security scanners for your containers

Ref- <https://techbeacon.com/app-dev-testing/8-best-practices-microservices-app-sec>

<https://www.edureka.co/blog/microservices-security>

#### Maintainability

Maintainability is all about how easy it is to fix a bug found in the application.

we call maintainability is typically two things

* Maintainability
* Extensibility

#### Maintainability

Maintainability is all about how easy it is to fix a bug found in the application.

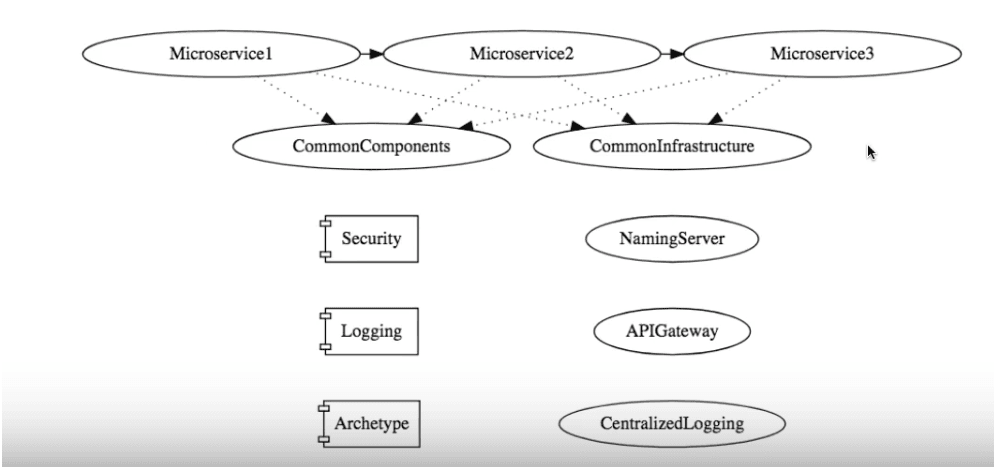
#### Extensibility

Extensibility of a system describes how easy it is to incorporate new features into it.

Both these concepts are quite deeply related, and are clubbed together under a single umbrella - maintainability. That’s how we address them as well

Adhering to 4 Principles of Simple Design is a great starting point.

#### Ensure Modular Design



#### 2. Ensure Great Automation Tests

#### 3. Automate The Deployment Pipeline

With an automated deployment pipeline, the build moves through different stages such as development, QA, staging and production in a systematic and efficient manner.

#### 4. Greater Visibility

Greater visibility of code across the modules in the system makes it easier to detect bugs, and shorten the time line in addressing them.

**Monitoring**

Monitoring systems over time produces valuable data that can be used to improve service performance. Failure and performance data can be analyzed to look for patterns in system failures, which can be correlated with events. For example, consider a case where data indicates 25 percent of total system failures occur within an hour of a new deployment. This would thus be a strong indicator that deployment processes need attention and improvement.

Monitoring technology and tools come in two broad categories: libraries and platforms. Some tools include both, providing a platform for collection and a library to instrument code.

Examples include open-source libraries like [AppMetrics](https://www.app-metrics.io/) for .NET and [SPF4J](http://www.spf4j.org/) for Java.

* [Zipkin](https://zipkin.io/): Zipkin is an open-source tracing system designed specifically to trace calls between microservices. It is especially useful for analyzing latency problems. Zipkin includes both instrumentation libraries and the collector processes that gather and store tracing data.
* [Apache Kafka](https://kafka.apache.org/): Kafka is a streams-processing system. It uses a “publish/subscribe” methodology for reading and writing data to a logical “stream,” which is similar in concept to a messaging system such as RabbitMQ. Kafka can be combined with other tools such as Zipkin to provide a robust solution for transmitting and storing metrics data.

## Conclusion

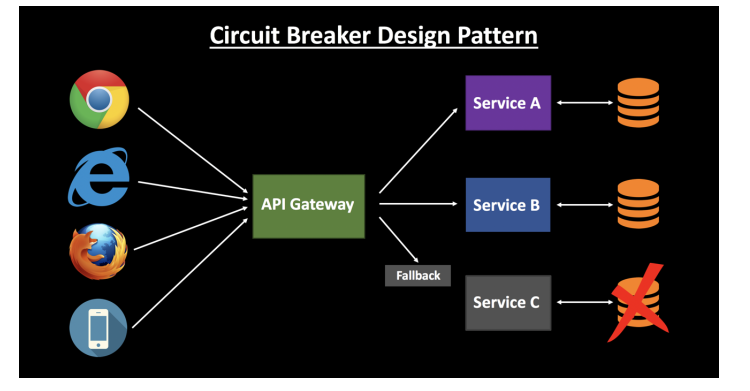
Monitoring requirements should be considered from the very beginning of an application’s lifecycle. Systems monitoring requires contributions from both development and operations.

## Fault Tolerance

## **Design Patterns to Ensure Service Resiliency**

### ****Circuit Breaker Pattern****

If there are failures in your microservices ecosystem, then you need to fail fast by opening the circuit. This ensures that no additional calls are made to the failing service, once the circuit breaker is open. So we return an exception immediately. This pattern also monitor the system for failures and once things are back to normal, the circuit is closed to allow normal functionality.



This is a very common pattern to avoid cascading failure in your microservice ecosystem.

You can use some popular third-party libraries to implement circuit breaking in your application, such as ***Polly*** and **Hystrix**.

### ****Retry Design Pattern****

This pattern states that you can retry a connection automatically which has failed earlier due to an exception. This is very handy in case of temporary issues with one of your services. A lot of times a simple retry might fix the issue. The load balancer might point you to a different healthy server on the retry, and your call might be a success.

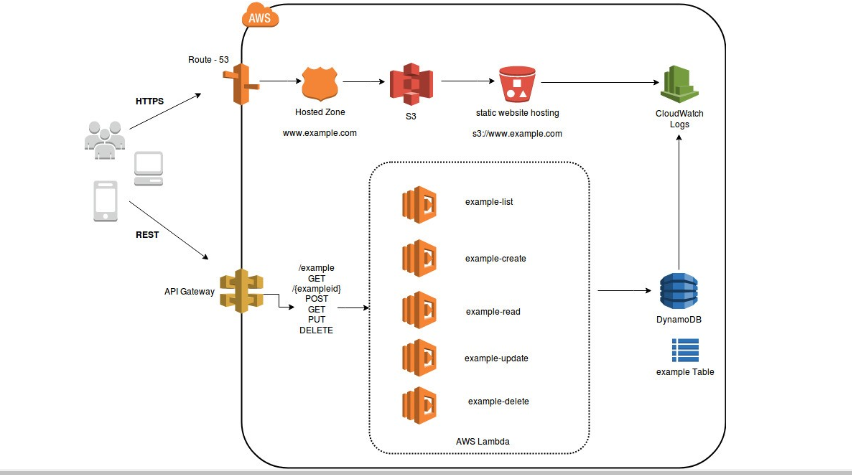
### ****Timeout Design Pattern****

This pattern states that you should not wait for a service response for an indefinite amount of time — throw an exception instead of waiting too long. This will ensure that you are not stuck in a state of limbo, continuing to consume application resources. Once the timeout period is met, the thread is freed up.

Serverless

The diagram below shows the serverless microservices architecture where the complete solution is built without managing any server. This also eliminates the operational efforts of running and monitoring the servers.

Lambda will handle everything required to run and scale the execution to meet actual demand with high availability. Lambda supports several programming languages and it can be called directly from any web or mobile applications.



In the architecture diagram, Lambda is integrated with API Gateway. Synchronous calls from API gateway to AWS Lambda enables the application to operate as serverless. AWS Lambda will store all the data in a fully managed NoSQL database called DynamoDB and all the static data will be stored in S3 Bucket.

It can be said that microservices architecture is designed to overcome the challenges of traditional monolithic architectures seen in enterprise applications. It allows collaboration between operations and development teams of any organization leading to DevOps and is a preferred choice nowadays.

AWS offers multiple managed services that can help engineers build microservice architectures and minimize architectural and operational complexity.

## **Challenges of Microservices Architecture**

Quick Setup needed : You cannot spend a month setting up each microservice. You should be able to create microservices quickly.

Automation : Because there are a number of smaller components instead of a monolith, you need to automate everything - Builds, Deployment, Monitoring etc.

Visibility : You now have a number of smaller components to deploy and maintain. Maybe 100 or maybe 1000 components. You should be able to monitor and identify problems automatically. You need great visibility around all the components.

Bounded Context : Deciding the boundaries of a microservice is not an easy task. Bounded Contexts from Domain Driven Design is a good starting point. Your understanding of the domain evolves over a period of time. You need to ensure that the microservice boundaries evolve.

Configuration Management : You need to maintain configurations for hundreds of components across environments. You would need a Configuration Management solution

Dynamic Scale Up and Scale Down : The advantages of microservices will only be realized if your applications can scaled up and down easily in the cloud.

Pack of Cards : If a microservice at the bottom of the call chain fails, it can have knock on effects on all other microservices. Microservices should be fault tolerant by Design.

Debugging : When there is a problem that needs investigation, you might need to look into multiple services across different components. Centralized Logging and Dashboards are essential to make it easy to debug problems.

Consistency : You cannot have a wide range of tools solving the same problem. While it is important to foster innovation, it is also important to have some decentralized governance around the languages, platforms, technology and tools used for implementing/deploying/monitoring microservices.

**Solutions to Challenges with Microservice Architectures**

**Spring Boot**

Provide non-functional features

* embedded servers (easy deployment with containers)
* metrics (monitoring)
* health checks (monitoring)
* externalized configuration

**Spring Cloud**

**Spring Cloud provides solutions to cloud enable your microservices. It leverages and builds on top of some of the Cloud solutions opensourced by Netflix (Netflix OSS)**

**Important Spring Cloud Modules**

**Dynamic Scale Up and Down. Using a combination of**

**Naming Server (Eureka)**

**Ribbon (Client Side Load Balancing)**

**Feign (Easier REST Clients)**

**Visibility and Monitoring with**

**Zipkin Distributed Tracing**

**Netflix API Gateway**

**Configuration Management with**

**Spring Cloud Config Server**

**Fault Tolerance with**

**Hystrix**