Introduction To Microservice

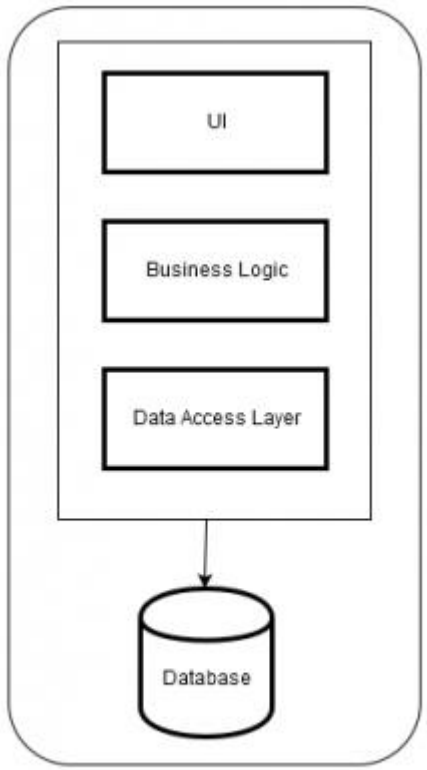
Microservice architecture is a method of developing software applications as a suite of independently deployable, small, modular services in which each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal.

Microservices are a service-oriented architectural pattern as well for defining distributed software architectures.

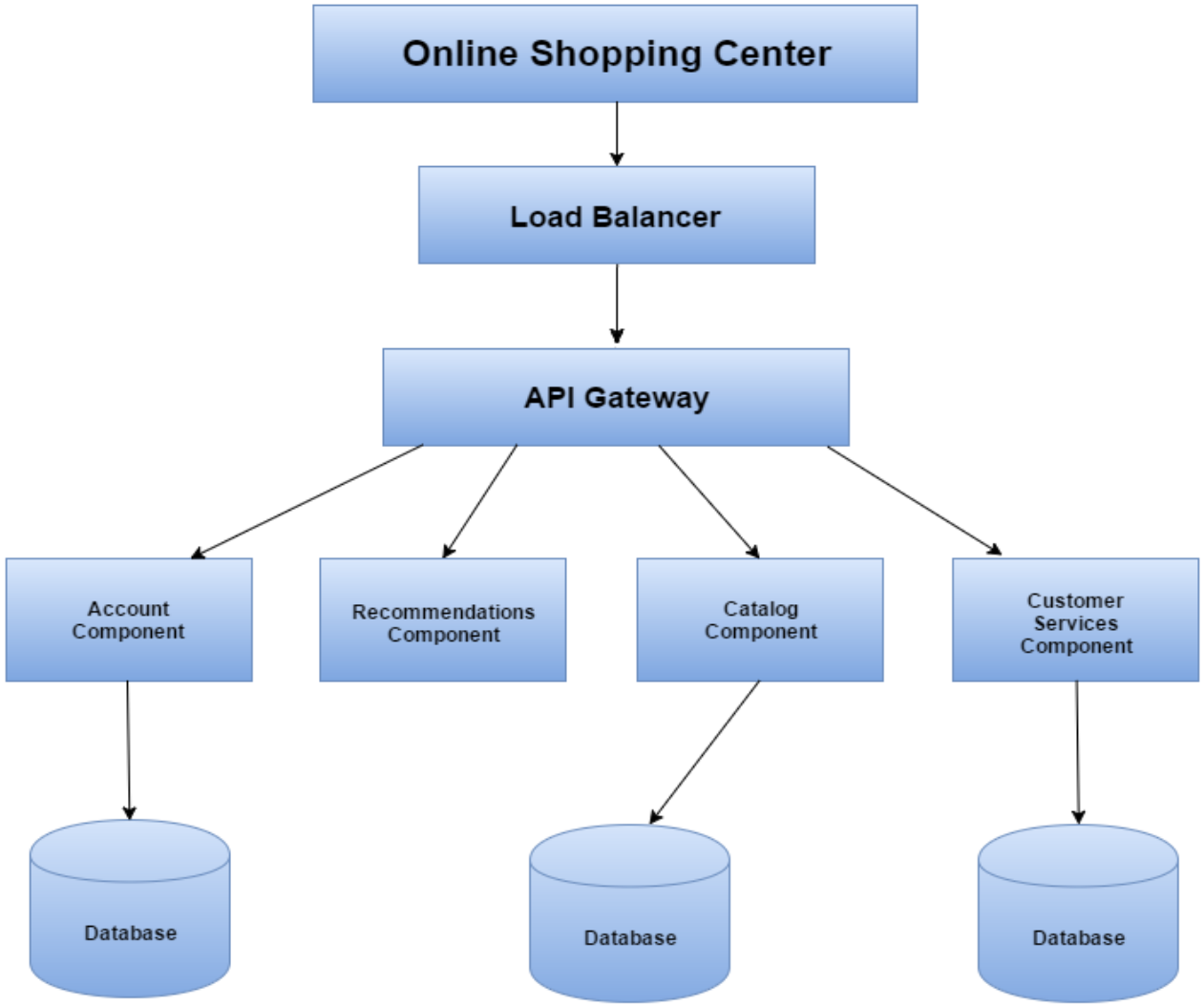
The pattern aims for better scalability, decoupling and control throughout the application development, testing and deployment cycle.

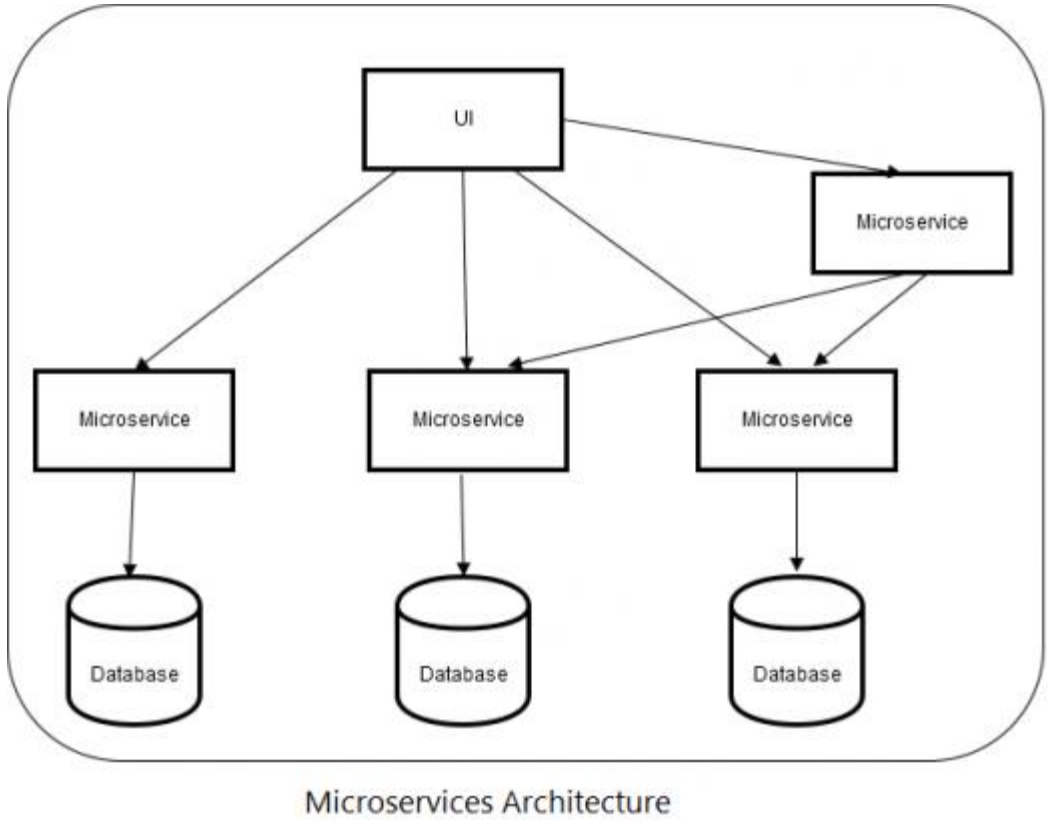
It relies on an inter-service communication protocol, which could be SOAP, REST and other technologies..

Figure 1: Conventional Approach



Monolithic Architecture

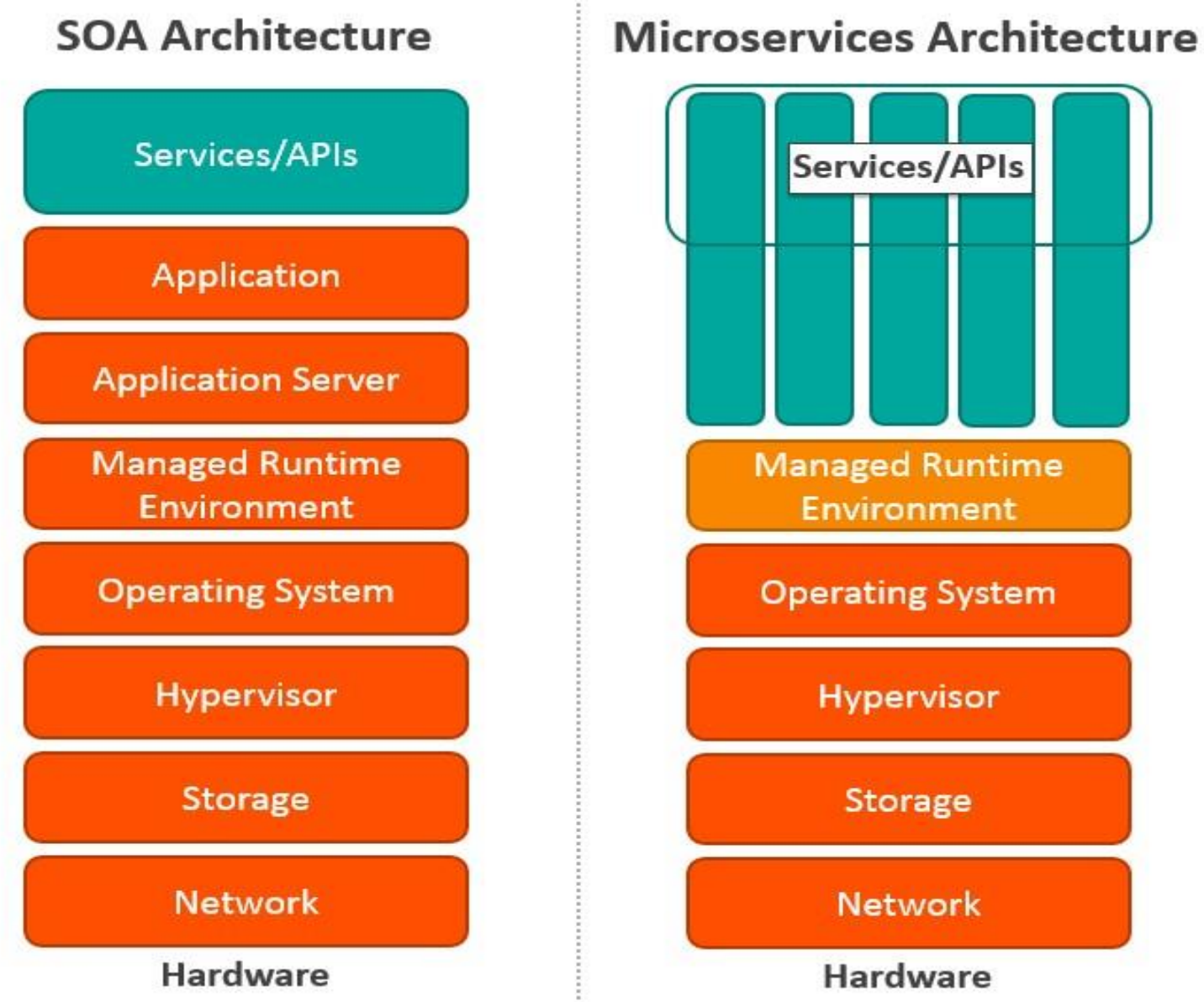




Microservices Architecture

Figure 2: Micro Services Approach

SOA Vs Microservices



SOA Vs Microservices

SOA commonly relies on a shared data model with multiple hierarchies and complex relationships between dozens or hundreds of data structures and services.

It uses a tiered organizational architecture that contains a centralized messaging middleware for service coordination plus additional messaging functionality.

Microservices are a service-oriented architectural pattern as well for defining distributed software architectures.

The pattern aims for better scalability, decoupling and control throughout the application development, testing and deployment cycle.

It relies on an inter-service communication protocol, which could be SOAP, REST and other technologies..

Microservices Benefits

- ❑ Smaller code base is easy to maintain.
- ❑ Easy to scale as individual component.
- ❑ Technology diversity i.e. we can mix libraries, databases, frameworks etc.
- ❑ Fault isolation i.e. a process failure should not bring whole system down.
- ❑ Better support for smaller and parallel team.
- ❑ Independent deployment
- ❑ Deployment time reduce

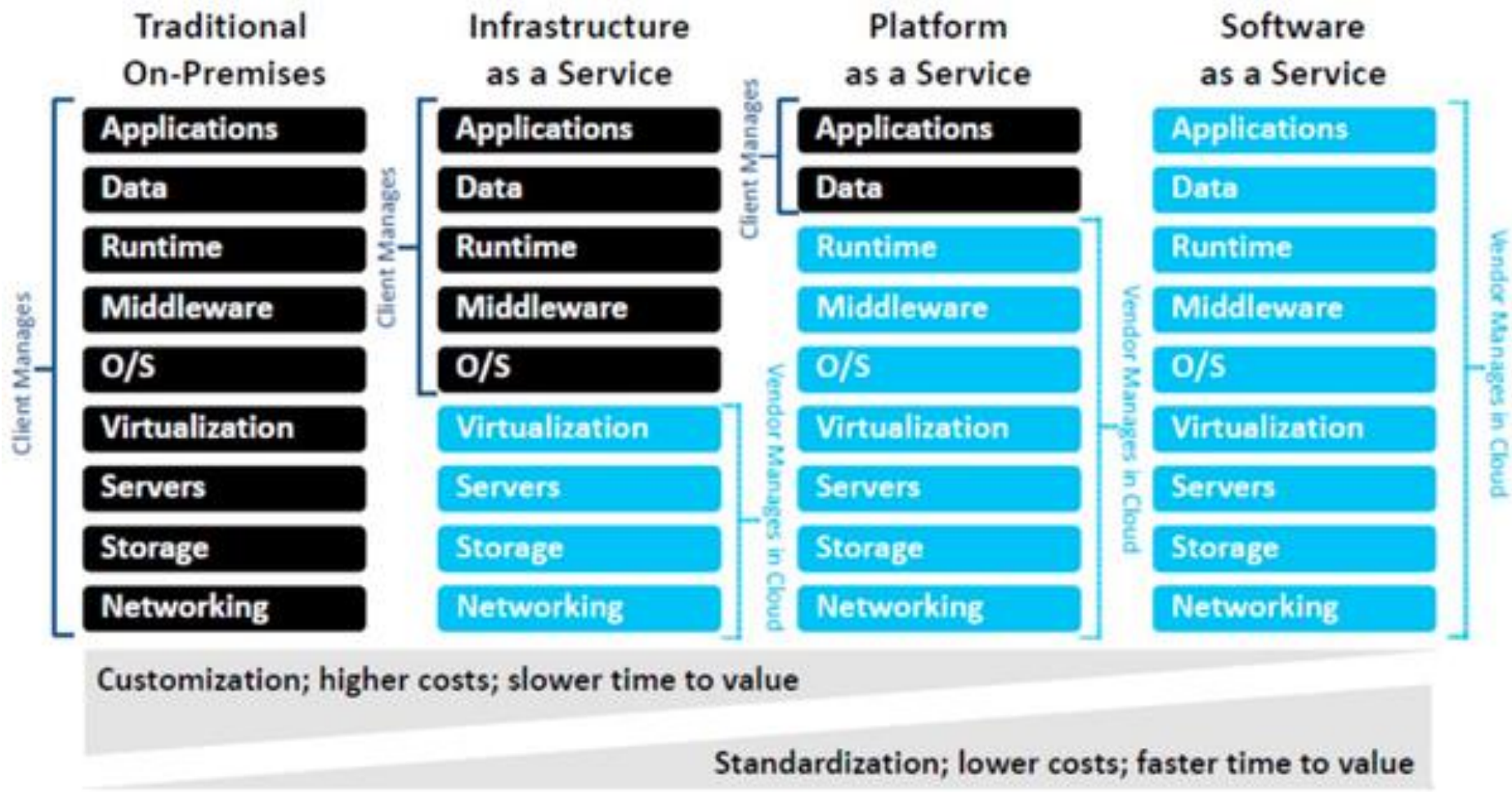
Microservices Challenges

- ❑ Difficult to achieve strong consistency across services
- ❑ ACID transactions do not span multiple processes.
- ❑ Distributed System so hard to debug and trace the issues
- ❑ Greater need for end to end testing
- ❑ Required cultural changes in across teams like Dev and Ops working together even in same team.

Microservices Infrastructure

- Platform as a Service like Pivotal Cloud Foundry help to deployment, easily run, scale, monitor etc.

- It support for continuous deployment, rolling upgrades fo new versions of code, running multiple versions of same service at same time


Traditional On-Premises
Infrastructure as a Service
Platform as a Service
Software as a Service
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
Client Manages
Client Manages
Client Manages
Vendor Manages in Cloud
Vendor Manages in Cloud
Customization; higher costs; slower time to value
Standardization; lower costs; faster time to value

Microservices Tooling Supports

1. Using Spring for creating Microservices

- ❑ Setup new service by using Spring Boot
- ❑ Expose resources via a RestController
- ❑ Consume remote services using RestTemplate/Feign

2. Adding Spring Cloud and Discovery server

- It is building blocks for Cloud and Microservices
- It provides microservices infrastructure like provide use services such as Service Discovery, Configuration server and Monitoring.
- It provides several other open source projects like Netflix OSS

.

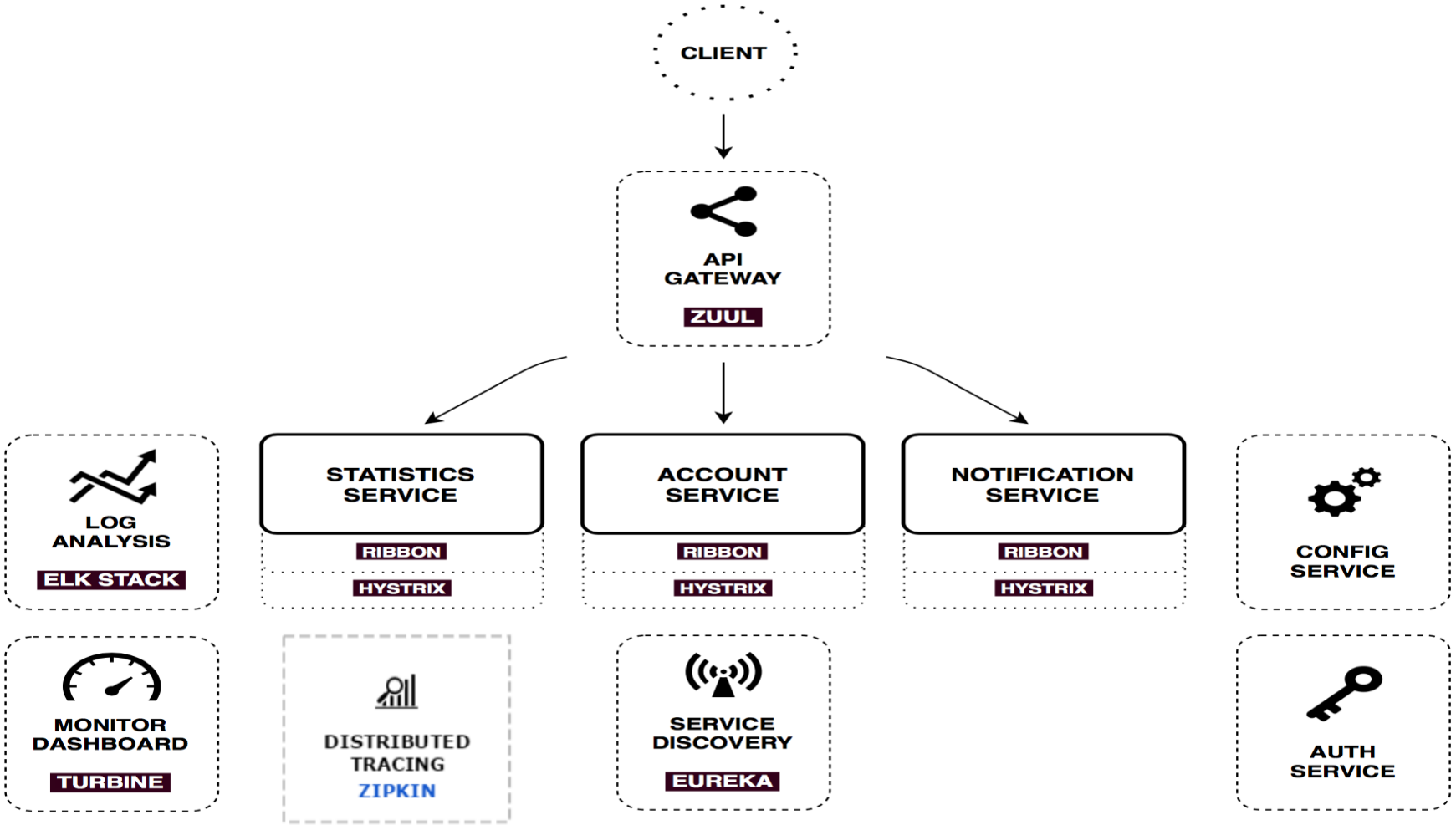
- It provides PaaS like Cloud Foundry, AWS etc.,
- It uses Spring Boot style starters

spring-cloud-netflix

This project provides Netflix OSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms.

With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components.

The patterns provided include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul) and Client Side Load Balancing (Ribbon


CLIENT
API GATEWAY
ZUUL
LOG ANALYSIS
ELK STACK
STATISTICS SERVICE
RIBBON
HYSTRIX
ACCOUNT SERVICE
RIBBON
HYSTRIX
NOTIFICATION SERVICE
RIBBON
HYSTRIX
CONFIG SERVICE
MONITOR DASHBOARD
TURBINE
DISTRIBUTED TRACING
ZIPKIN
SERVICE DISCOVERY
EUREKA
AUTH SERVICE

Service Discovery:

Eureka instances can be registered and clients can discover the instances using Spring-managed beans

Circuit Breaker:

Hystrix clients can be built with a simple annotation-driven method decorator
Embedded Hystrix dashboard with declarative Java configuration

Declarative REST Client:

Feign creates a dynamic implementation of an interface decorated with JAX-RS or Spring MVC annotations

Client Side Load Balancer:

Ribbon

External Configuration:

Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system. With the Config Server you have a central place to manage external properties for applications across all environments.

Router and Filter:

Automatic registration of **Zuul** filters, and a simple convention over configuration approach to reverse proxy creation

Hystrix stream

Netflix Hystrix has a neat feature called the Hystrix stream that provides real-time metrics on the state of the Hystrix commands in an application. This data tends to be very raw though, a very cool interface called the Hystrix Dashboard consumes this raw data and presents a graphical information based on the underlying raw data.

Turbine stream

Hystrix stream provides information on a single application, Turbine provides a way to aggregate this information across all installations of an application in a cluster. Integrating turbine into a Spring-Cloud based application is straightforward, all it requires is information on which clusters to expose information on and how to aggregate information about the specific clusters.

ELK – Elasticsearch, Logstash, Kibana:

three different tools usually used together.

They are used for searching, analyzing, and visualizing log data in a real time.

Zipkin is a distributed tracing system

It helps gather timing data needed to troubleshoot latency problems in microservice architectures.

It manages both the collection and lookup of this data.