Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Information Engineering

Diploma Thesis

Modern scalable distributed cloud native applications

Bc. Ondřej Svojše

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

DIPLOMA THESIS ASSIGNMENT

Bc. Ondřej Svojše

Systems Engineering and Informatics
Informatics

Thesis title

Modern scalable distributed cloud native applications

Objectives of thesis

Goal of this master thesis is to describe evaluation of available resources and system architecture design to build distributed scallable application that takes full advantage of running in cloud environment. This will include evaluation of advantages and disadvantages of particular cloud providers such as Google Cloud Platform, Amazon Web Services and others. Furthermore there will be examined the cost and complexity of scalling up.

Methodology

In theoretical part, there will be explained different application architecture and their benefits. Also in thesis will be described technologies and reactive programming styles that work hands-on with scalable architectures. The styles will be carefully evaluated and compared with included use-cases. Continuing with exploration of capabilities and services offered in modern public cloud platforms. Then we'll go through principles of developing application that is design to run in cloud environment and explore lifecycle of deployment and additional development and testability. In practical part will be shown cloud native application that serves for gathering big data from user and machine inputs (IoT devices) running in multiple datacentres across the globe with high throughput. This application will be deployed in cloud environment fully integrated with provider's services. At the end of the thesis there will be evaluation of described solution and comparison with alternatives.

The proposed extent of the thesis

80 – 140 pages

Keywords

Big Data, IoT, Reactive streams, WebFlux, Microservice architecture

---

Recommended information sources

Hands-On Microservices with Spring Boot and Spring Cloud, Birmingham, UK PacktSeptember 2019, ISBN 9781789613476.

LEE, J. – WEI, T. – MUKHIYA, S K. *Hands-on big data modeling : effective database design techniques for data architects and business intelligence professionals.* Birmingham, UK: Packt, 2018. ISBN 978-1-78862-090-1.

Orchestrating and Automating Security for the Internet of Things: Delivering Advanced Security Capabilities from Edge to Cloud for IoT, Cisco Press. June 2018, ISBN 9780134756936.

Scalable Architecture for the Internet of Things. O'Reilly Media, Inc., February 2018. ISBN 9781492024125.

Expected date of thesis defence

2020/21 SS – FEM

The Diploma Thesis Supervisor

doc. Ing. Vojtěch Merunka, Ph.D.

Supervising department

Department of Information Engineering

Electronic approval: 19. 11. 2020

Ing. Martin Pelikán, Ph.D.

Head of department

Electronic approval: 19. 11. 2020

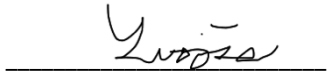
Ing. Martin Pelikán, Ph.D.

Dean

Prague on 21. 03. 2021

Declaration

I declare that I have worked on my diploma thesis titled "Modern scalable distributed cloud native applications" by myself and I have used only the sources mentioned at the end of the thesis. As the author of the diploma thesis, I declare that the thesis does not break copyrights of any their person.

In Prague on 20. 3. 2021 

.

Acknowledgement

I would like to thank associate professor Vojtěch Merunka PhD. for advices with tackling the topic and also to my colleagues from Avast Software for consulting the topic and teaching me.

Modern scalable distributed cloud native applications

Abstract

In the thesis there is examined the state of modern public cloud platforms and the services offered that enable them to operate highly scalable applications with lower costs. First part is focused on technology and cloud services. The way they shape current cloud native application development and the platform they provide for developers to build reliable and resilient solutions. Looking further there is examined elements of how technologies such as containers or orchestrators affect application architecture, exploring the benefits and challenges introduced with this new way of doing things.

In the following part there is quantitative analysis of a developers survey conducted in a large Czech software company on topic. The aim of it is to find out the views developers have on learning new technologies, public cloud and key factors and requirements they view for delivering good SaaS product at scale.

Next step is exploring the platform of a large cloud infrastructure provider. Based on ideas and patterns there is documented building of Kubernetes cluster with many services that are communicating with each other through asynchronous messages bus like Apache Kafka or directly through HTTP protocol and are horizontally scaling instances based on workload. Then there is shown the importance of CI / CD pipeline in delivery to achieve zero downtime deployments of individual services. Another part mentioned in the first part of research is the observability of platforms, system be evaluated according to relevant quality metrics ISO/IEC 25023. The following part is examining ways how to trace the request going through the cluster and important metrics for keeping eyes on day to day operation. In the last part load test is performed and observation of performance are done.

In the end we evaluate the objectives of the thesis and conclude the results.

Keywords: Public Cloud, Containerization, Orchestrators, Virtualization, Distributed Systems, Cloud Native Applications, Microservices, Monitoring, Tracing

Moderní škálovatelné cloud native aplikace

Abstrakt

V diplomové práci je zkoumán stav veřejných cloudových platforem a služeb poskytovaných za účelem spravování vysoce škálovatelných aplikací za nižší provozní cenu. První část se zabývá technologií a cloudovými službami. Způsob, jakým utváří současné trendy vývoje cloud native aplikací a platformy, které jsou poskytovány vývojářům k vybudování spolehlivých a odolných řešení. Dále se práce zabývá výzkumem dopadu technologií jako je kontainerizace nebo orchestrace, na aplikační architekturu, zjišťování výhod a výzev spojených s tímto novým přístupem.

Další část se zabývá kvantitativní analýzou ankety určené pro vývojáře, která byla provedena v české softwarové firmě na dané téma. Cílem ankety je zjištění pohledu vývojářů na témata jako sebevzdělávání v nových technologiích, otázky v oblasti veřejných cloudů a také, jaké vidí klíčové faktory a požadavky pro doručení dobrého a škálovatelného SaaS produktu.

Toto následuje praktický výzkum platformy a služeb velké cloudové společnosti. Na základě poznatků a schémat z předchozí části práce se pustíme do vybudování Kubernetes clusteru, který uvnitř spravuje spoustu aplikací, které spolu komunikují přes asynchronní message bus Apache Kafka nebo napřímo přes http protokol a jsou horintálně škálovatelné v závislosti na zátěži. Dále je ukázána důležitost CI / CD pipeline k dosažení doručení služeb s nulovými výpadky pro individuální služby. Na to celé navazuje, jak již bylo zmíněno v teoretické části, pozorovatelnost platformy a zhodnocení kvality systému podle relevantních metrik ISO/IEC 25023. Dále je na clusteru ukázáno, jaké jsou možnosti trasování požadavků napříč distribuovaným HTTPem a také důležité metriky, které je nutné sledovat pro udržení stabilního chodu. V poslední části jsou oproti systému spuštěny zátěžové testy a pozoruje se chování aplikace. Nakonec se vyhodnotí všechny cíle a výsledky diplomové práce.

Klíčová slova: Veřejný Cloud, Kontajnerizace, Orchestrace, Virtualizace, Distribuované systémy, Cloud Native Aplikace, Microservices, Monitoring, Trasování

Table of content

List of pictures

List of tables

List of abbreviations

IoT – Internet of Things
AWS – Amazon Web Services
MS – Microsoft
CI – Continuous Integration
CD – Continuous Delivery
API – Application Programming Interface
GDPR – General Data Protection Regulation
CPU – Central Processing Unit
RAM – Random Access Memory
GPU – Graphical Processing Unit
SaaS – Software as a Service
IaaS – Infrastructure as a Service
PaaS – Platform as a Service
HTTP – Hypertext Transfer Protocol
mTLS – Mutual Transport Layer Security
VM – Virtual Machine
ISO – International Organization for Standardization
UML – Unified Modeling Language
KYC – Know your customer
UX – User Experience
SLA – Service Level Agreement
REST – Representational state transfer
UDP – User Datagram Protocol

1.1 Introduction

Once upon a time programs ran on the same machine as they were accessed from. This has changed a long time ago. Now almost every application is considered a distributed system and runs on multiple machines that gets accessed by many users from all over the world.

With the invention of the internet, almost everybody is capable of joining a shared network and exchanging data with counterparts all over the world. We can see an increasing number of users joining the internet network from the developing countries as well as many new devices and IoT gadgets that are becoming our extended hands and feed the network with tons of data.

With digitalization happening all around us the classical approach “business as usual” is no longer enough. Today's organizations operating on the internet need to deliver user friendly, reliable and scalable solutions fast. Customer demand is steadily increasing with more people coming online as well as organization opportunities to increase the engagement or revenue. To achieve it, it's necessary to accelerate the delivery of goods and services, responses to potential risks such as security threats or changes in the economy or anticipate regulatory changes and the impact it has on the systems.

Additional events like humanity being plunged by the global pandemic of Covid-19 resulted in most of the business moved from traditional retail space to online selling. As a side effect this also caused growth of the companies offering software solutions and resulted in necessity of delivering fast with constant feedback loops on product.

Software and technology is in this era key differentiator for organizations to deliver value to customers and stakeholders. It allows for rapid growth of products and reach all across the globe. In modern distributed computing it becomes ever more attractive to move infrastructure to public cloud. For companies this brings many benefits like not having to focus on running the whole infrastructure operation. Cloud providers in general are having access to cheaper hardware and are able to solve common problems such as reliability and high availability through their own software solutions.

2 Objectives and Methodology

2.1 Objectives

- Introduce services provided by cloud infrastructure and explain how they enable organizations to achieve higher development speed and lower costs.
- Explore technologies that support running applications in the cloud in a highly secure, observable, reliable and scalable manner with minimal costs for hardware utility.
- Define key metrics for choosing cloud providers and compare the largest on the market.
- Present different architectural options and explain their suitable use cases when building web based systems. Compare their applicability in cloud environments.
- Define best practices for building cloud native applications.
- Describe the process of continuous integration and deployment of applications in Cloud without downtimes.
- Explore suitable testing stacks for applications in the cloud to minimize amounts of error in distributed systems.
- Show ways how to monitor and trace requests running inside distributed systems.
- Gather opinions and feedback on the topic of public clouds and it's benefits from professional software developers
- In practical part develop a distributed system that gathers information from devices or IoT gadgets and load test behavior of the system with artificial traffic.
- Evaluate developed quality system based on relevant selected metrics of ISO/IEC 25023.

2.2 Methodology

In the theoretical part, there will be introduced cloud infrastructure as a service and its benefits for organizations to reduce overhead in their operation. Then follows exploration of services and software that providers offer. Next is defining key metrics for organization and then comparing offers from biggest cloud providers like AWS, MS Azure or Google Cloud Platform. In the next part there will be presented different application architectures, how they relate to cloud and fitting use-cases. Based on research of cloud environment and application architecture deeper research into preferred application structure and properties will be done. We explore principles and patterns necessary to follow in a cloud environment when building applications deployed as containers.

Then the testability of the application and different stages of testing will be explored to achieve stable CI/CD. After that we explore what are the options on how to monitor the whole distributed system and how to trace errors in case they occur. As final part of theoretical research on the topic there will be conducted surveys for professional software developers with the aim to gather feedback on experiences with different cloud providers, unexpected surprises with cloud deployment and developer experience in comparison with traditional self hosted servers.

Goal of the practical part is to demonstrate knowledge gathered in the theoretical part. Based on referenced materials will develop a distributed system of applications deployed in multiple datacenters on cloud that is able to collect data from the IoT devices or users. Whole solution should be fully connected to monitoring and have set up deployment pipelines with running tests for continuous delivery. Final test of the solution will be done through load performance test that is designed to stress test the solution and observe durability of the distributed system under large volumes of requests on application servers.

In the end we evaluate the result of the tested prototype and draw conclusions.

3 Literature Review

3.1 Cloud Infrastructure

Infrastructure in this context is generally all the software and hardware that enables running applications in remote environments. This includes everything necessary to support the lifecycle of the application. Through years of evolution and perfecting practices, many big technological companies like Amazon, Microsoft or Google to name a few, were able to make large business out of leasing computing power and package it as a service.

Nowadays the main reason for adoption and usage of public cloud is for organizations to be able to produce value faster and focus on core business products. Organizations strive to build only what's necessary for creating and delivering products. Using services of other providers, keeps lead time small and agility high.

Some organizations are often hesitating with the choice of public cloud because of scarcity of “vendor lock-in”. It's important to consider also the fact that building your own infrastructure is not having the same effect on organizations. It's also dependent on the choice of how many of the services and in what way you use it.

Moving to cloud native infrastructure certainly doesn't solve every problem and it's necessary to point out that the responsibility to choose the right solution always carries the organization however over the years there are strong indicators of success for many organizations operating in public clouds that have adopted the provided tools and patterns. (Garrison, 2017)

3.1.1 Servers

Compared to cloud infrastructure it's quite difficult to get them and after that correctly set up to be able to run your applications.

Since the beginning of computing the common knowledge about physical servers was they take a certain size, are expensive, make a lot of noise and require a lot of electricity to keep them operational.

Physical servers are powerful and you can configure them in any way imaginable. They have a relatively low failure rate and are engineered to avoid failures with redundant power supplies, fans, and RAID controllers. They also last a long time. However in general owning servers leads to waste due to not fully utilizing them and adding additional maintenance overhead. Also it's not simple to configure them in an optimal way to run

multiple applications, avoid software conflicts, set up correctly network routing and all user access rights.

It's important to point out that in many organizations like the army or other regulated fields it's often not advised to use cloud infrastructures due to security threats connected with just being online.

3.1.2 Virtualization

In (VMWare, 2018) is defined virtualization as the process of running virtual instances of computer systems in layers abstracted from the actual hardware. From the perspective of applications running in a virtualized environment it appears as if they have their own dedicated machine with their own operating system, libraries and other programs unique systems. This computing technique solves decoupling of hardware from software. It allows for more efficient sharing of resources between various workloads running on a server. Technology that enables running multiple guest virtual machines by virtually sharing its resources is called hypervisor.

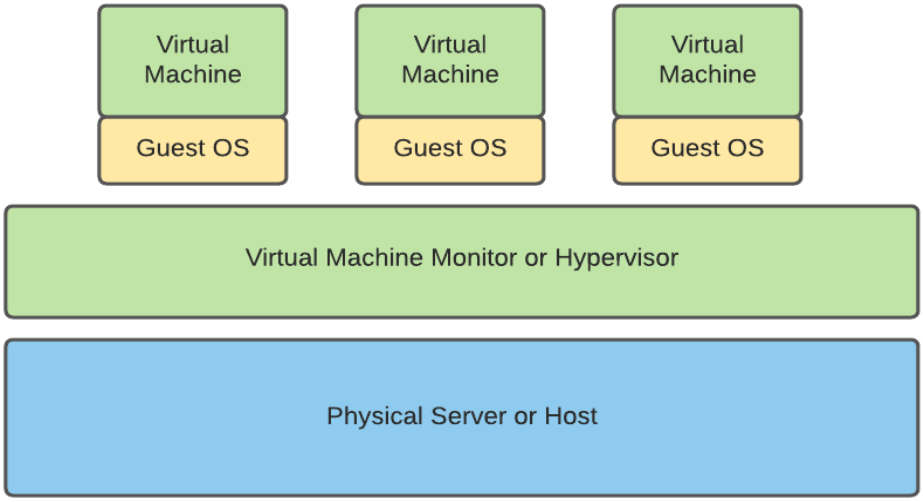


Figure 1: Virtualization layers scheme - source: (author)

Issue with running your own virtualization platform is that you still need to own physical hardware and have people spending time maintaining it and keeping it running. With the increasing scale of IT requirements these costs grow.

3.1.3 Infrastructure as a Service (IaaS)

IaaS is made up of a collection of physical and virtualized resources that provide support for running applications and workloads in the cloud. It's one of many services offered by cloud providers to consumers. IaaS allows organizations to get rid of their own hardware and rent it instead. In this model from an organization's perspective the expense is no longer viewed as capital expenditure but as operational expense of running the business. Organizations can pay for their infrastructure the same as they pay for electricity or human resources.

In addition the hosted infrastructure also comes with consumable application programming interfaces (API) that allows over the HTTP to create and orchestrate virtualized infrastructure. Whole concept is to get rid of the need to purchase servers and setting them up. Instead you can just call the API of the provider and create the on demand infrastructure. This allows organizations to elastically increase or decrease the amount of resources they use based on the demand for their product. (Education, 2019)

3.1.3.1 Data centers

It's standard for large IaaS providers to manage large data centers all around the world. In data centers they store physical machines necessary to power all the layers of abstraction that provide the service through API to customers. In this model customers don't interact with physical infrastructure.

Users can also choose the availability zones for hosting the services. Different zones are logically and physically isolated locations with independent power and network infrastructures. This strengthens fault tolerance by eliminating single points of failure. Also this allows for high bandwidth and low latency within the region. Also having this option of choice helps with obeying data protection laws like GDPR in Europe.

3.1.3.2 Compute

Typically in this platform are used virtualized compute resources in the form of virtual machines. The provider is managing the hypervisors and end users can then programmatically create instances with desired compute power and memory. Most providers offer both CPUs and GPUs for different types of workloads. It's common for cloud providers also to provide supporting services like auto scaling and load balancing to adjust required performance on fly.

3.1.3.3 Network

Networking in cloud is made by networking hardware such as routers and switches that are exposed to be configured programmatically through API.

3.1.3.4 Storage

Storage space is the same as the previous components manageable through API. Cloud vendors provide services to help collect, manage, secure and analyze data at massive scale. Ensuring organization data is secure, safe and available is essential and therefore there are several fundamental requirements when considering storing data.

- Availability - data should be available whenever users request them.

- Durability - data should be stored ideally across multiple independent locations and multiple devices to prevent events like human error, natural disasters or hardware failure that might result in data loss.

- Security - data should be encrypted as well as in storage as in transit. Correct permissions and access control policies should be put in place to prevent unwanted data access. (AWS, 2019)

Another benefit of this solution is waste choice of operating systems, advanced monitoring of the platform and tools built to support and automate running of applications.

3.1.4 Platform as a Service (PaaS)

As IaaS takes away hassle with managing hardware and VMs, PaaS hides the layer of operating systems from the applications. This means for developers that they push application code and define the dependencies. Provider interprets the dependencies and is responsible for handling it according to instructions. In the case of PaaS the infrastructure is fully managed by cloud providers.

From developers perspective this is a big change. Developers no longer remotely connect to VMs to inspect logs from applications or read files on disk. Lifecycle and management is under control of the provider. This might seem like a limitation but it turned out to be a great benefit. In many cases it shaped how cloud native applications are written today.

3.1.5 Software as a Service (SaaS)

SaaS takes this idea of outsourcing a step further. In (Salesforce, 2020) it’s defined as a way to deliver cloud-based service that you can run in your browser from any device. You don't have to take care of updates, patches or maintenance; everything is being done by the service provider for you.

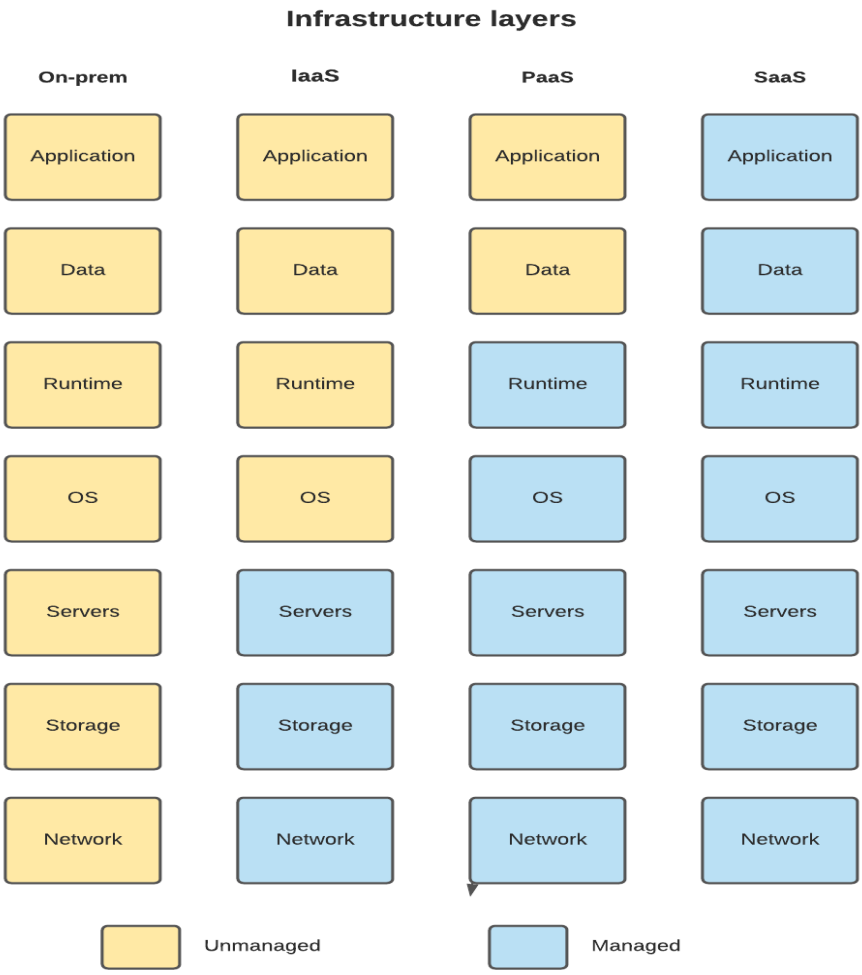


Figure 2: Infrastracture layers overview - source: (author)

3.1.6 Key Success Factors

3.1.6.1 Cost of Infrastructure

In a research paper by (KyungWoon Cho, 2020), that explores key variables and impact on total cost of running infrastructure in the cloud.

- Ingress / Egress (per cluster)
- CPU power used
- RAM amount used
- Storage amount used
- Managed services

In the case of CPU the price of a model is determined according to the computing power that it provides and is charged based on the unit price of the model and service time. With Storage there it depends on combinations of storage volumes and number of I/O operations.

Egress and Ingress cost is the cost for the data that moves in and out of the cloud. Often providers offer input of data in the network for free but charge large network fees to move them out of the cloud elsewhere.

With services it's individual it depends on how much of the software offered by 3rd parties is the organization gonna decide to use.

Total Estimation Equation

$$Cost_{inst}(t_{act}, t_{inact}, T, f_{sto}, f_{net}) = Cost_{inst}^{act}(t_{act}, T, f_{sto}, f_{net}) + Cost_{inst}^{inact}(t_{inact}, T)$$

This equation can be used to estimate cost of the infrastructure but usually there are other things like services used along with it so it has to be také in consideration.

3.1.6.2 People Expertese

In order for organizations to sucessfully execute cloud computing strategies, they need the right people with the right skills. According to (Burger, 2021) at the beginning of adoption of third-party providers services, there is a critical need for people who know what services to pick, who can negotiate service level agreements, and can integrate those off-site offerings with on-site data and operations.

In (Humble, 2018) is recommended to estimate the people's expertise, organizations can use simple tools as surveys. Data from well-designed and well-tested psychometrics surveys can give quick feedback on identifying its own resources and talent. It allows to quickly and easily analyze data in a reliable manner.

Descriptive research

In (SVMK Inc., 2020) says that descriptive research is considered conclusive in nature due to its quantitative nature. Unlike exploratory research, descriptive research is preplanned and structured in design so the collected information can be statistically used on population.

This type of research aims to better define an opinion, attitude, or behaviour held by a specific group of people on given subject. This allows to measure the signifikance of

results on the overall population that's subjected to study, as well as the changes of respondent opinions, attitudes and behaviours over time.

3.2 Containerization

Containers have been used for quite some time now, especially by large tech companies like Google, Microsoft or Amazon. It's a piece of technology that's supposed to address shortcomings of the VM model.

In some way VMs and containers are quite similar. Major difference between those two is that containers don't require a full operation system. All containers on a single host share the one OS that runs on it. This results in huge freeing up of system resources such as CPU, RAM and storage. Also reduces potential licensing costs and overhead with maintenance. Containers are designed to be fast to start and portable.

In the last few years there were several advancements that enabled massive growth of the containerization trend. Particularly kernel namespaces, control groups, union filesystems and Docker.
It's a software that runs on most common platforms like Windows, Mac and Linux. Docker made linux containers more usable and simple to take advantage of.

The whole solution consists of 3 main components:

1. Runtime
2. Engine / daemon
3. Orchestrator

Purpose of runtime is to start and stop containers. It operates on the lowest level. Runtime architecture splits on lower-level and higher-level.

Lower level is called "runc" and it's job is to interface with the OS and manage the container state. Higher-level runtime is called "containerd". It manages the entire lifecycle of containers. It means creating network interfaces, pulling images and managing lower-level runc instances.

Engine sits on top of containerd and performs tasks as exposing Docker API, managing images and managing volumes. (Poulton, 2019)

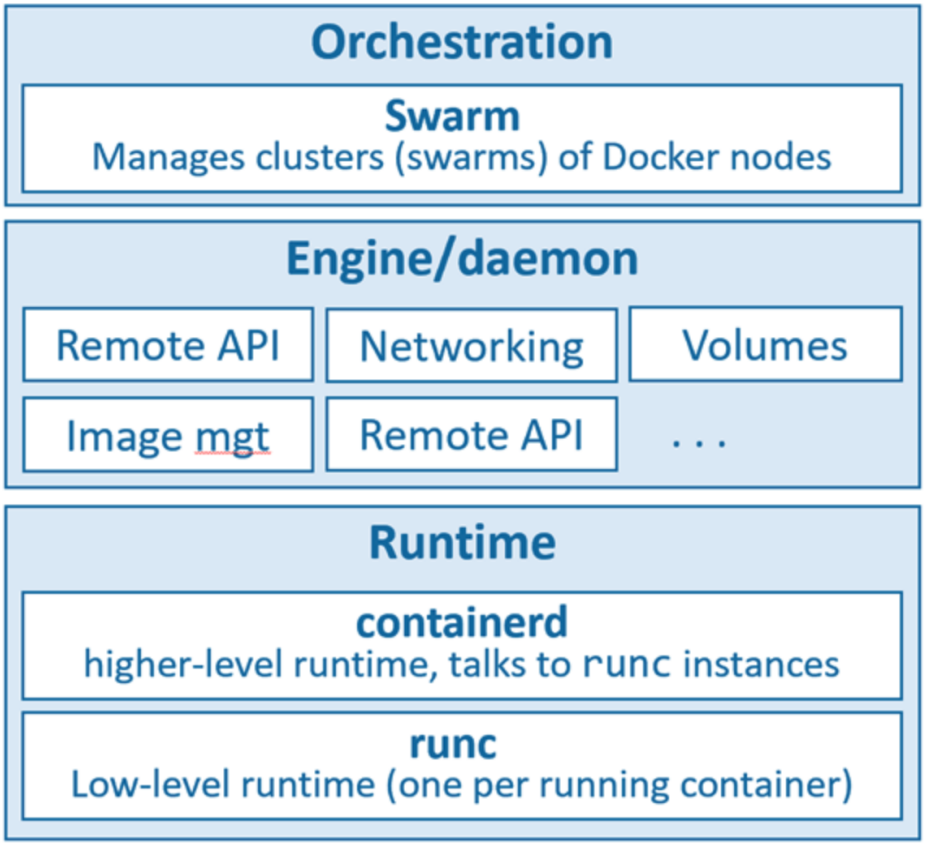


Figure 3: Docker architecture - source: *(Poulton, 2019)*

3.3 Orchestrators

When you learn how to containerize distributed applications you start to face the same problems and challenges that non-containerized applications face. It can be service discovery, load balancing, scaling and so on. To solve these issues named above there were developed software solutions that we call orchestrators.

Main purpose of orchestration is to achieve expected outcome, mostly making sure that all containers in the cluster communicate with each other how they‘re supposed to.

3.3.1 State control

Typically use of an orchestrator includes specifying declarative ways how application services should run. We want to declare what should be run and how to run it. This is usually related to container images, opened ports and properties of application services.

As a result when we run orchestrator it creates all the application services we requested with ports we specified and makes sure they‘re deployed to the cluster where they‘re supposed to run.

3.3.2 Service Discovery

It‘s not recommended to use deterministic placement rules, this should be left to the orchestrator itself. Knowledge of for example service A shouldn‘t be where to discover service B that it relies on. In distributed and scalable applications it should rather rely on orchestrators to give these indices where to discover services.

3.3.3 Routing

In distributed systems are typically many services interacting with each other through HTTP/s protocol. It‘s application developer‘s expectation that orchestrators HTTP over this task of routing from source to desired destination. When routing messages between services running in containers, there can happen 3 situations. First is that source and target containers are in the same cluster node, then the second would be if source container is located in a different cluster node than target container and the third when data comes outside the cluster and has to be routed to the target container running inside the cluster. All those scenarios have to be handled by the orchestrator.

3.3.4 Load Balancing

Precondition to run a highly available distributed application is to have redundancy between all components. Every application is run in multiple instances and if single instances fail, the service is still operational. To distribute the load to all instances of a service and not letting them sit idle it‘s necessary to make sure that requests are evenly distributed among instances. This proces is called load-balancing. There are multiple algorithms how to distribute the workload but the most standard and process called “round robin algorithm”.

Round robin load balancing is one of the simplest methods for distributing client requests across a group of servers. Going down the list of servers in the group, the round-robin load balancer forwards a client request to each server in turn. When it reaches the end of the list, the load balancer loops back and goes down the list again (sends the next request to the first listed server, the one after that to the second server, and so on). (Nginx, 2019)

3.3.5 Scaling

After containerizing our application we can let them run and be managed by the orchestrator. Not only that but we also need an easy way to handle unexpected increases of requests they have to process. In that case the orchestrator usually schedules more instances to spawn according to declarative policy. Load balancers will then automatically distribute the load over new instances.

This happens fairly regularly in the real world. Typically we run as few instances as possible to reduce cost of the operation. It‘s normal that for example shopping applications have peaks during the day when there are more visitors than the other time. When this scenario is spotted the orchestrator simply schedules more instances of the service and scales horizontally.

Horizontal scaling – modify compute resources of an existing cluster.
Vertical scaling – modify the attributed resources (CPU, RAM) of node in the cluster.

Furthermore critical services should be evenly distributed across all data centers to avoid outages. All these decisions and many more are responsibility of orchestrator.

3.3.6 Self-healing

One of the functions of an orchestrator is also to monitor the health of all containers running in the cluster and automatically replace those who appear faulty with newly spawned instances. Naturally there are different conditions upon which each we evaluate health of the services and dependant how the author of the application exposes them.

Orchestrator defines seams or probes, over which application can communicate with it and share in what state it currently is. Fundamentally we have probes of 2 categories.
Readiness probe states

- The ACCEPTING_TRAFFIC state represents that the application is ready to accept traffic.

- The REFUSING_TRAFFIC state means that the application is not willing to accept any requests yet.

Liveness probe states

- The *CORRECT* value means the application is running and its internal state is correct.

1. On the other hand, the *BROKEN* value means the application is running with some fatal failures.

The orchestrator task in this case is only to define how it‘s going to ask. It means for example through HTTP GET request and expecting predefined answers from the service like UP or DOWN. (Mohamandinia, 2021)

3.3.7 Deployments without outage

For organizations it‘s not acceptable to have downtimes when business critical applications need to be updated. Outage usually leads to bad customer experience and in damaging the reputation of the organization. Furthermore with applying methodologies like agile development, the release cycles are getting shorter and shorter. It no longer releases a couple releases each year, it has changed to multiple updates per week.

The role of orchestrator in this solution is the zero downtime update strategy it implements to ensure smooth deployments. Application services are updated in batches and we call it “rolling updates”. At any given time few instances are taken down and replaced with a new version of the application. When there are no issues the orchestrator switches the rest of the instances while the HTTP is running. In case of failure of update, orchestrator automatically rollbacks updated instances back to their previous version.

3.3.8 Canary releases and blue-green deployments

Those two options make new versions of the service possible that are installed in parallel with the currently active version. Initially the new version is accessible only internally so it can be properly tested and run smoke tests against. After seeing everything runs smoothly, the operation team can switch to exposing it on production. In case there is an issue spotted it‘s possible to switch router settings to point back on the old version to achieve complete rollback of deployment.

With canary releases router is configured in a way that it routes a certain small percentage of total traffic to the new version of the application, while rest of the traffic still is routed to the old version. This enables developers to closely observe how the new version behaves in production on a small number of users and reduce impact of the error.

Most of the orchestrators support by default rolling update with zero downtime. With blue-green and canary releases there typically is extra configuration required.

3.3.9 Security

Security of the solutions is critical for organizations to protect data privacy of the customers and own reputation. Cyber crime rate is at all-time high with more data being transmitted and accessed through the internet people or organizations become targets of hacker groups attacks.

To protect the organization against outside threats there has to be an established secure software supply chain and enforced 28roces security rules.

It‘s necessary that the cluster managed by the orchestrator is secure. Only for trusted nodes should be possible to join and each node should incorporate cryptographic identity. Also the communication between the nodes itself has to be encrypted. This is usually done by secure transport protocol mTLS. For authentication of nodes among each other there are used certificates.

Communication inside the cluster can be separated into three types often referred to as planes.

Management plane – is used to schedule service instances, create or modify volumes inside the cluster, calling health checks, manage secret configuration and network.

Control plane – is used for exchanging information between all nodes of the cluster. This can be for example to update the local IP tables in the cluster for routing purposes.

Data plane – is in between application services communication.

Orchestrators usually HTTP care of securing management and controlling planes and to secure the data plane is up to application developers.

3.3.10 Managing secrets

In applications we have to use secrets for example certificates to authenticate our application services with some external service or vice versa or we need a token to authenticate and authorize our service when calling API. As mentioned in (Larsson, 2019) traditionally all the configuration secrets were stored in an external configuration file that application loaded them from. This may contain sensitive data and make it accessible to a broader audience then is desired.

This is also one area that‘s being solved by the orchestrator. Orchestrator offers a way to deal with sensitive data in a highly secure way. Secrets can be added only by trusted and authorized personnel. Values itselfs are encrypted and stored in a cluster database. Authorized application then requests the secret and it‘s forwarded to the cluster nodes that run an instance of that particular service.

Secret is never stored on a node, it gets stored into RAM-based volume in a *tmpfs* folder of the container, so it‘s accessible only from the container itself. Information transferred

between services are transmitted as plain text but the data packets are encrypted by mTLS.

Content Trust Security

another vulnerable part of the cluster are images. Often we can for added security configure that the orchestrator can run only signed images. This is the way to make sure that the author of the image is the one we expect it to be and it was not tampered with by some malicious attacker.

Signed images at the source and their validation at the target guarantees that the image that orchestrator is going to run not compromised.

3.3.11 Introspection

In earlier chapters it was discussed how many tasks an orchestrator does autonomously. But there is still a necessity for operators to monitor and analyze if the HTTP works inside the cluster as intended. Therefore it‘s important to be able to introspect.

Orchestrators in general collect a lot of HTTP metrics from cluster nodes. Those metrics include disk usage, CPU, memory usage and network bandwidth. It should be available in aggregate form across all nodes to enable analysis whether there is mistake in the application or per node form that enables for example to examine issues in specific datacenter.

In distributed applications it‘s also necessary to be able to trace requests that go through various services inside the cluster. This support i salso one of the parameters to look out for when choosing a good orchestrator.

For humans it‘s necessary to have all that data also accessible visually in 29roces29 graphs so the decisions can be made based on properly understanded data. Most of the orchestrators also allow to set up alerts on selected metrics to give an operator heads up in case something is out of regular.

3.4 Application Architecture Style

Nothing is more contextual to a number of factors within an organization and what software it builds. Choosing an architecture style represents the culmination of analysis and thought about trade-offs for architecture characteristics, domain considerations, strategic goals, and a host of other things. However contextual the decision is, some

general advice exists around choosing an appropriate architecture style (Marks Richards, 2020)

It‘s important to acknowledge that style salso shift over time and it‘s driven by numerous factors.

3.4.1 Factors:

- Observations from the past – every now and then new architecture styles arise based on observations of shortcomings of the previous one.

- Changes in the ecosystem – software is constantly being developed and it‘s difficult to predict where it‘s going to move in future. Often new technologies shift the view on architecture style completely.

- New Technologies – When significant new technologies appear the old architecture may not be merely replaced but rather shifted to an entirely new paradigm. For example with popularizing container technology there was a big shift in how to design the architecture of systems.

- Acceleration – new tools lead to new engineering practices which have an impact on design and capabilities of software. Architects must constantly observe thost changes.

- External factors – sometimes development is happy with infrastructure they run on or tool they use but the licensing cost or operational cost grow disproportionate to alternatives.

3.4.2 Decision Criteria

It‘s necessary to HTTP in account all the various factors when selecting proper architectural style. Fundamentally architect designs whatever specified domain and all other structural elements required to make the HTTP running smoothly.

- The domain - one of the most important aspects, it affects operational architecture characteristics. Architects don‘t have to be necessarily subject matter experts, but

should have at least a good understanding of major aspects of the domain under design. Discover and elucidate the architecture characteristics needed to support the domain and other external factors.

- Data architecture – architects have to collaborate on database, schemas and other data design related concerns. It should be understood what impact the data might have on their design. Particularly it‘s important when having more to work also with older systems to design proper interaction in data architecture models.

- Organizational constraints – many external constraints may affect final design. It can be for example the cost of features from cloud vendors that would be ideal to use or company plans of growth into a certain direction in future.

- Knowledge of team structures and information streams – project specific requirements often affect architecture as well, the architecture should be well structured to ease up operational processes and quality assurance processes. For example if agile engineering teams lack maturity in the organization, it‘s very hard for certain architecture styles that rely on those practices for success, which will present many difficulties.

- Distributed or Monolithic architecture choice – architect must determine if the single set of architecture characteristics will be sufficient for the design, or do different parts of the HTTP need differing architecture characteristics ? Single set implies that a monolith is suitable, in other cases it implies a distributed architecture.

- Type of communication between services – before doing any decision around communication we have to know whether the communication will be synchronous or asynchronous. Synchronous communication is more convenient but it can lead to scalability and reliability issues. On the other hand asynchronous communication provides benefit in terms of performance and scaling possibilities but also makes things more complicated. In asynchronous communication often many issues like data synchronization, deadlocks or race condition arise. Therefore it makes sense to use synchronous communication wherever you can predict that there shouldn‘t be any scalability issues.

3.4.3 Monolithic architecture

Monolithic architecture is the traditional unified model for designing a service. Monolithic means that the whole solution is composed all in one piece. It‘s tightly interconnected and interdependent rather than decoupled as in case of more modular architectures like microservices. Tightly-coupled architecture means that each component and the associated components must be present in order for code to compile and execute.

This also means that if any component must be updated, you have to update the whole application whereas in modular architectures it‘s enough to update only components that‘s changed.

There are also benefits of this, in general it‘s easier to test and debug, when all elements of a solution are in one place and also keeping correct dependencies. This type of architecture is more recommended for simpler solutions. (Wigmore, 2016)

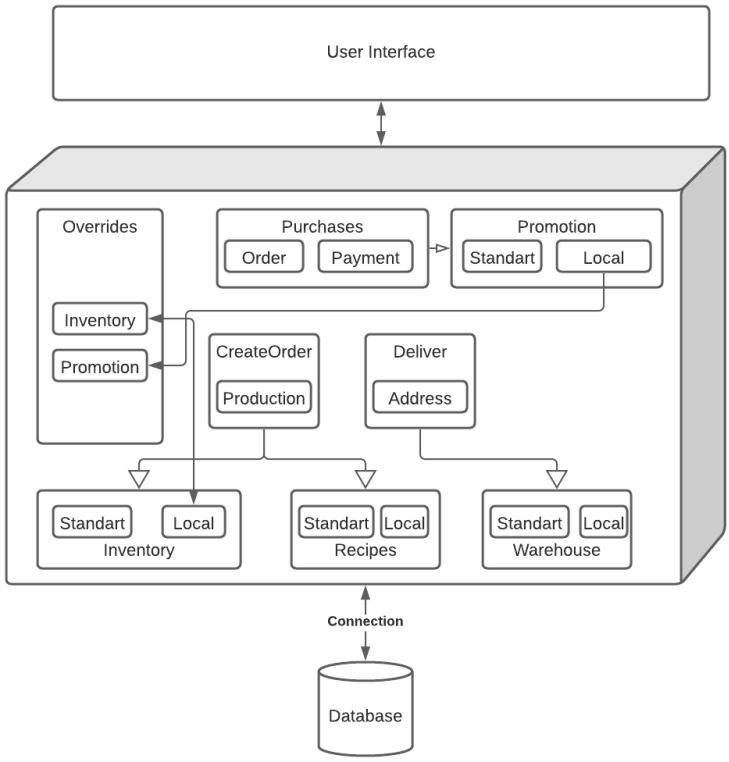


Figure 4: Monolithic architecture example - source: (author)

3.4.3.1 Modular Monolith

Modularization is a technique in programming and software design that emphasizes separating functionality of a program into independent, interchangeable modules, such

that each contains everything necessary to execute only one aspect of the desired functionality. A module interface expresses the elements that are provided and required by the module. The elements defined in the interface are detectable by other modules. The implementation contains the working code that corresponds to the elements declared in the interface. (Grzybek, 2019)

One way how to loosely decouple independent parts of application in one monolith is to split the independent business domains into several modules that are:

- independent and interchangeable
- have everything necessary to provide functionality
- have defined interfaces

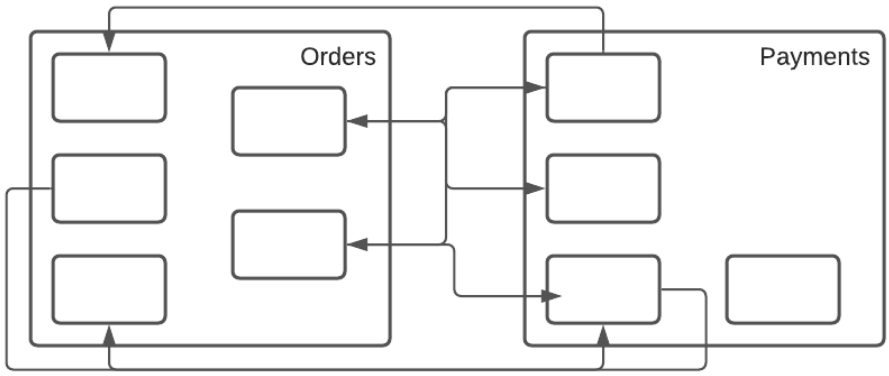


Figure 5: Modules without encapsulation example - source: (author)

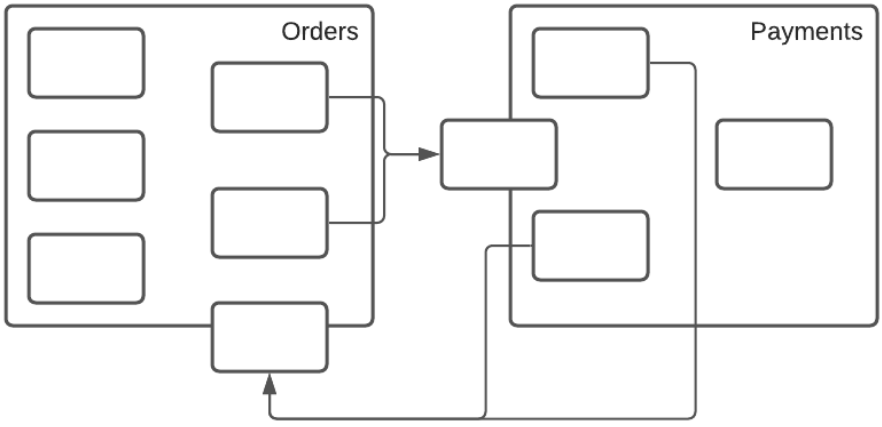


Figure 5: Modules with encapsulation example - source: (author)

3.4.4 Microservice architercute

Service oriented architecture in which applications are decomposed into loosely coupled, small services based on area of functionality. It‘s important to keep the service small and decomposed just around business capability.

It‘s often contrasted with monolithic architectures. In case of microservices instead of managing a single codebase and sharing data schema and database as in monolithic applications, in microservices architecture cod eis decomposed into smaller codebase that is managed independently. All the services in this case contribute to achieve a single well-defined task. Services run in separate processes and communicate synchronously through either HTTP API or asynchronously through message contracts. (Newman, 2021)

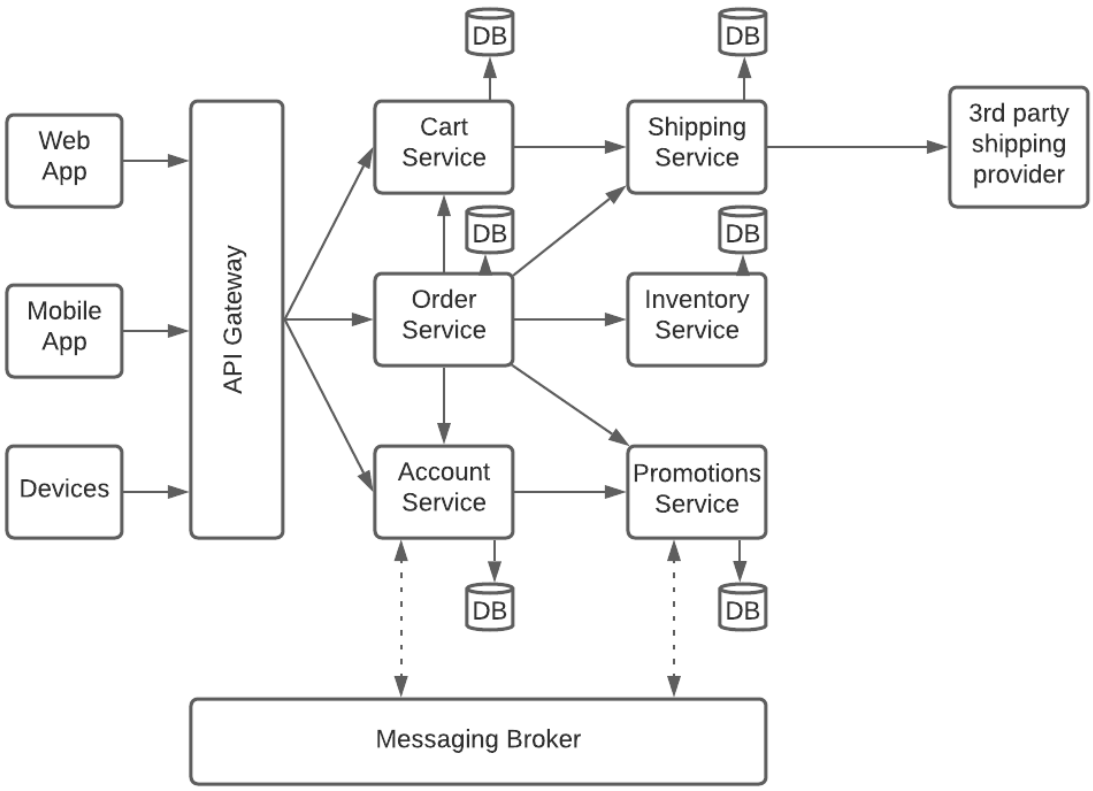


Figure 6: Microservice architecture example - source: (author)

3.4.4.1 Benefits of Microservices Architecture

Well implemented microservices architecture increases the release velocity of large applications and enables business to deliver value faster and more reliably.

Agility

Compare to monolithic applications where fast and reliable deployments can be challenging. Having small change in one feature area can be held up by a change to another feature. In large applications, time spent on testing will increase and it can take up significant time to deliver new change. By decomposing an application into smaller services, the time necessary

Fault Isolation

In monolithic applications, a single library fallacy or module can cause large problems for the whole application. This can be as little as memory leak imported through mistakes in external libraries and can affect stability and performance of applications. When the services are isolated and operate independently, defects can be contained in single service.

Scaling and resource utility

In general applications scale horizontally or vertically. It means that one option of scaling is increasing resources on a given machine that applications run on or scale out by increasing the number of deployed instances and route users across all of them. Usually applications have many features and many of them could have different resource requirements. Some features scale easily, whereas others might require a lot of memory and therefore limit the possibility of vertical scaling. Decoupling those features into separate services, teams can help to meet individual resource and scale requirements.

Team Responsibilities

Having many small services greatly helps with splitting responsibilities among engineering teams. Making people responsible for designing, running and operating improves the motivation of the team to keep things clean whereas in monolithic application all components are intertwined and it‘s hard when making changes to not affect other areas in application.

Observability

Decomposing application into many services enables us to gain deeper insights into individual parts,behavior of the features and their interaction. Also it‘s easier to identify and optimize the services based on HTTP metrics such as proces utilization or memory usage. It‘s easier to tie them back to the feature when they run in a separate process or container.

3.4.4.2 Challenges of Microservice architecture

As mentioned in chapter 3.4.3, there are trade-offs when choosing one or the other. Despite gaining more popularity in modern stacks, microservices architecture has its own challenges that need to be addressed. The fact that it‘s popular to use in new applications doesn‘t have to neccessary mean that it‘s suitable for every scenario (Sabella, 2018).

Complexity

Distributed systems are complex by their nature. When application is decoupled into many services, it‘s necessary to use network calls for data exchange between them. This adds a bit of latency and experience transient failures, and the operations can run on different machines with a different time. It‘s not assumed that even in the cloud the latency is zero, infinite bandwidth and the network is always secured. In fact there are many false assumptions that developers commonly have like that the topology of the network will not change, there is one administrator, transport is for free. Many developers that are not familiar with distributed systems often make false assumptions when entering this world.

Consistency and Data Integrity

Having decentralized data means that data will often exist in multiple places with relationships spanning different systems. (Lee, 2018) says it‘s not sufficient to perform transactions across all those systems. There is a necessity to deploy different operational models to data management. To do it new patterns for dealing with this have arised.

Versioning integration

In microservices architecture it‘s important to pay attention to forward and backward compatibility between services that rely on mutual communication. To ensure services can communicate there are usually contract tests between services. This guarantees that API interfaces are compatible with each other. This layer certainly adds additional

overhead compared to monolithic applications when the whole solution compiles in one place.

Service dependency management

In monolithic applications the advantage is that all libraries and dependencies get to be compiled into one package and tested as a whole. With microservices, dependencies are managed differently, requiring environment-specific routing and discovery.

Availability

Despite the isolation of faults in microservices design, It‘s still an issue if one service fails to serve the functionality that it should provide. Therefore services should implement resilient patterns, or possibility of functionality downgraded in the event of outage. Implementing health and readiness checks is vital part for detecting flaws in application communication in time and preventing degraded experience for users. (Boris Scholl, 2019)

3.5 Cloud Native Applications

When building an application that‘s supposed to be deployed and run in a cloud environment it‘s necessary to keep in mind that all major cloud providers such as Google, Microsoft or Amazon offer various guides on how to integrate with their platform specific tools and patterns to follow.

In the modern era, software is commonly delivered as a service in front of SaaS. Therefore many developers come together and put experiences during development, operation and scaling of hundreds of apps into a document called “The Twelve-Factor App”.

3.5.1 The Twelve-factor App

This document was published to raise awareness about recurring problems in modern application development. It offers a set of broad conceptual solutions to those problems with accompanying terminology.

3.5.1.1 Codebase

Applications should always be tracked in a version control system like Git, Subversion or Mercurial.

Only one codebase per application but many deployments. This is typically multiple instances of the application running like production, staging and development environment.

3.5.1.2 Isolated Dependencies

The full and explicit dependency specification is applied uniformly to both production and development.

3.5.1.3 Config

Storing config into the environment variables. Environment variables are easy to change between deploys without changing any code and extra rebuild.

Config file shouldn‘t contain any credentials.

3.5.1.4 Backing Services

Like the database are traditionally managed by the same systems administrators who deploy the app’s runtime. In addition to these locally-managed services, the app may also have services provided and managed by third parties.

Code for a twelve-factor app makes no distinction between local and third party services. To the app, both are attached resources, accessed via a URL or other locator / credentials stored in the config.

3.5.1.5 Build, Release, Run

Build stage converts code from the source repository into an executable bundle called build. It fetches dependencies and compiled binaries and assets.

Release stage takes build from the previous stage and combines it with specified configuration files and is ready for execution on selected environments.

Run stage runs application in the execution environment by execution run commands.

3.5.1.6 Processes

Twelve-factor processes are stateless and share-nothing. Any data that needs to persist must be stored in a stateful backing service, typically a database.

The memory space or filesystem of the process can be used as a brief, single-transaction cache. For example, downloading a large file, operating on it, and storing the results of the operation in the database. The twelve-factor app never assumes that anything cached in memory or on disk will be available on a future request or job – with many processes of each type running, chances are high that a future request will be served by a different process. Even when running only one process, a restart (triggered by code deploy, config change, or the execution environment relocating the process to a different physical location) will usually wipe out all local (e.g., memory and filesystem) state.

Some web systems rely on "sticky sessions" – that is, caching user session data in memory of the app's process and expecting future requests from the same visitor to be routed to the same process. Sticky sessions are a violation of twelve-factor and should never be used or relied upon. Session state data is a good candidate for a datastore that offers time-expiration, such as Memcached or Redis.

3.5.1.7 Binding of Ports

Application is self contained and doesn't rely on runtime injections of web server into execution environment to create web-facing service

The web app exports HTTP as a service by binding to a port, and listening to requests coming in on that port.

3.5.1.8 Concurrency

Processes in the twelve-factor app HTTP strong cues from the unix process model for running service daemons. Using this model, the developer can architect their app to handle diverse workloads by assigning each type of work to a *process type*. For example, HTTP requests may be handled by a web process, and long-running background tasks handled by a worker process.

3.5.1.9 Disposibility

Processes are *disposable*, meaning they can be started or stopped at a moment's notice. This facilitates fast elastic scaling, rapid deployment of code or config changes, and robustness of production deploys.

Processes should aim to minimize startup time.

Processes should also be robust against sudden death, in the case of a failure in the underlying hardware.

3.5.1.10 Parity of Environments

The twelve-factor app is designed for continuous deployment by keeping the gap between development and production small.

- Make the time gap small: a developer may write code and have it deployed hours or even just minutes later.

- Make the personnel gap small: developers who wrote code are closely involved in deploying it and watching its behavior in production.

- Make the tools gap small: keep development and production as similar as possible.

3.5.1.11 Logging

It should not attempt to write to or manage log files. Instead, each running process writes its event stream, unbuffered, to *stdout*. During local development, the developer will view this stream in the foreground of their terminal to observe the app's behavior.

3.5.1.12 Admin Processes

One-off admin processes should be run in an identical environment as the regular long-running processes of the app. They run against a release, using the same codebase and config as any process run against that release. Admin code must ship with application code to avoid synchronization issues. (Wiggins, 2017)

3.5.2 API Design and Versioning

API is a prime interface for communication of services and therefore it is important to have proper documentation in place and versioning of API. Versioning is not that easy, especially given that there are many different approaches that can be implemented in a project.

Generally there are 3 main approaches. One is the "The knot" where consumers of your API are tied to a single version and when changes in the API are made, all consumers need

to change as well. This approach is disadvantageous for consumers because they‘re forced to upgrade when each new API version is released.

Next approach is called “point-to-point” where all API versions keep running and each consumer uses the version they need to. Consumers can migrate to the new versions when they decide to. Compared to the first approach, this strategy is kinder to consumers but on the other hand creates difficulty with the maintenance of older versions.

Last approach is called “compatible versioning”. All consumers talk to the same API version. Old versions are deprecated and no longer exist because the latest version is backward compatible.

3.5.2.1 Semantic Versioning

It‘s the most standard in the industry. The semantic versioning of the API can look following:

Table 1: Example of versioning (source: author)

Type	Major	Minor	Patch
Example	1.	2.	1.

- Major version is increased when you make API-incompatible changes.
- Minor version is increased when you add backward-compatible features.
- Patch version is increased when you make backward-compatible bug fixes.

This type of versioning can be applied to the level of API that communicates to consumers for signalizing changes that were made.

3.5.2.2 The OpenAPI Specification

(Miller, 2017) defines it as specification defines standard, programming language-agnostic interface description for REST APIs, which allows both humans and computers to discover and understand capabilities of a service without requiring access to the source code, external documentation, or inspection of network requests. When properly defined via OpenAPI, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Similar to what interface descriptions have

done for application level programming. The goal of specification is to remove guesswork in integration external service.

3.5.3 Service Communication

Services communication through the network are essential topics in distributed systems because of the impact it has on performance of systems. Therefore it‘s necessary to understand all options available during the design phase and implementation of cloud native application. Communication can be separated to the external and internal. External means the requests exchanged between services within our cluster and external are typically integrations on 3rd party services or end user requests.

3.5.3.1 Standard Protocols

The most common protocol for communication between server and client is HTTP, however it‘s not the most efficient one. Large distributed systems can be composed out of hundreds of services and chosen protocol for data exchange is an essential factor that affects performance.

Websockets

Websockets represent standard bidirectional real-time communication between server and clients. They allow us to open a long single TCP socket connection that‘s bidirectional. Full duplex messages can be instantly distributed with minimal latency. Connection of websockets is initiated by standart handshake through HTTP request to the server after that it‘s replaced with Websocket connection. It allows to transfer large volumes of data with relatively low latency.

gRPC

This protocol is fairly new and is gaining popularity through friendliness to developer and high performance. It uses protocol buffers which represent the way of defining serializing structured data into binary format. Binary format is very efficient and enables for small payloads and quick transmission.

HTTP 2

This protocol is designed for low latency and multiplexing requests over a single TCP connection by using streams. Another change over the HTTP 1.x is that it‘s no longer textual protocol but it changed into binary format of messages which are more efficient to transmit.

3.5.3.2 Messaging Protocols

Cloud native applications embrace event.-driven and message-based communication. There are many messaging protocols like STOMP, WAMP, AMQP, MQTT to name a few. Probably the 2 most popular are the following.

Message Queue Telemetry Transport

Binary protocol primarily used in the domain of IoT and machine-to-machine communication. It‘s designed to work in conditions with low bandwidth connection and unpredictable network. Often is used for communication between sensors and gateways. The protocol is based on publisher / subscriber messaging mechanism. It transmits messages through sending compact binary payload.

Advanced Message Queueing Protocol

Same as the previously mentioned protocol AMQP is also a binary protocol. It‘s designed around reliable queuing feature rich transactions. Compared to MQTT it‘s not as lightweight or fast, but is often preferred by various vendors due to reliability and interoperability.

Generally when choosing between these 2 protocols it depends on what the use case requires. MQTT is more designed for fast simple operations on the other hand if there is necessity for interoperability and advanced functionality that goes beyond simple messaging AMQP would be more suitable.

3.5.3.3 Asynchronous Messaging

Asynchronous messaging stands for message-based communication that enables loosely and light coupling between microservices and interactions through passing standardized messages. When using asynchronous communication, microservice publishes events when something happens and another microservice subscribes when it needs to be aware of it.

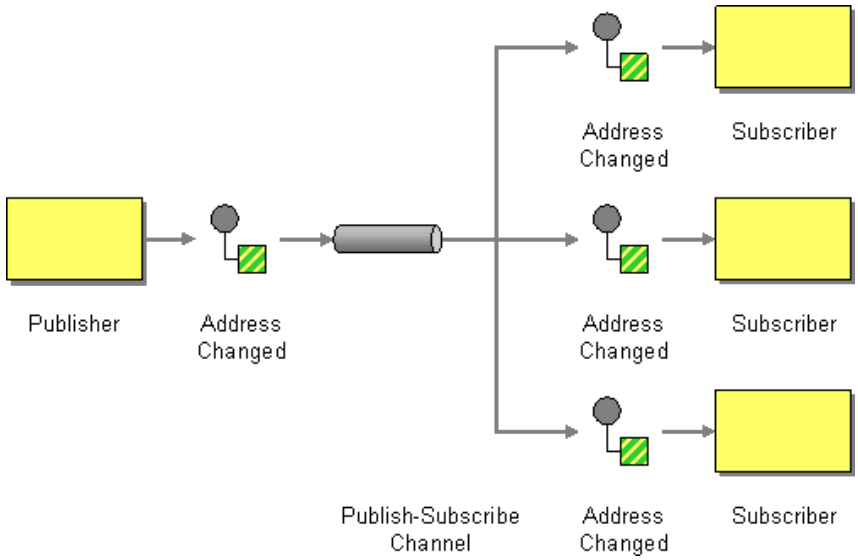


Figure 7: Publisher / Subscriber pattern of asynchronous communication - source: *(Hohpe, 2003)*

3.5.3.4 IoT Communication with Backends

In previous chapters 3.5.3.1 – 3.5.3.3 Were described communication types that support machine-to-machine communication. Knowing when to use them is essential to achieve proper results in terms of performance, response and robustness.

In (Mijic, 2018) are describe two possible patterns to communicate with the IoTs. The request/response mechanism requires an established channel between parties. It may be used for individualized information exchanges. The request/response mode isn't plausible when some condition (event) must trigger many of devices to execute an action. In this case, there are two general choices.

- Initiating loop sending matching command to all IoT's from matching device list.

- Send event to centralized service that notifies devices via message.

The publish/subscribe pattern is inherently decentralized. Figure 8 shows a sequence of actions that accompany this mechanism. The publish/subscribe mode creates fast, local action loops that don't swamp the central node. Moreover, a triggering device doesn't need to maintain a separate device list for each event. The protocol layer is optimized to manage the device network; thus it can efficiently handle registrations and notifications.

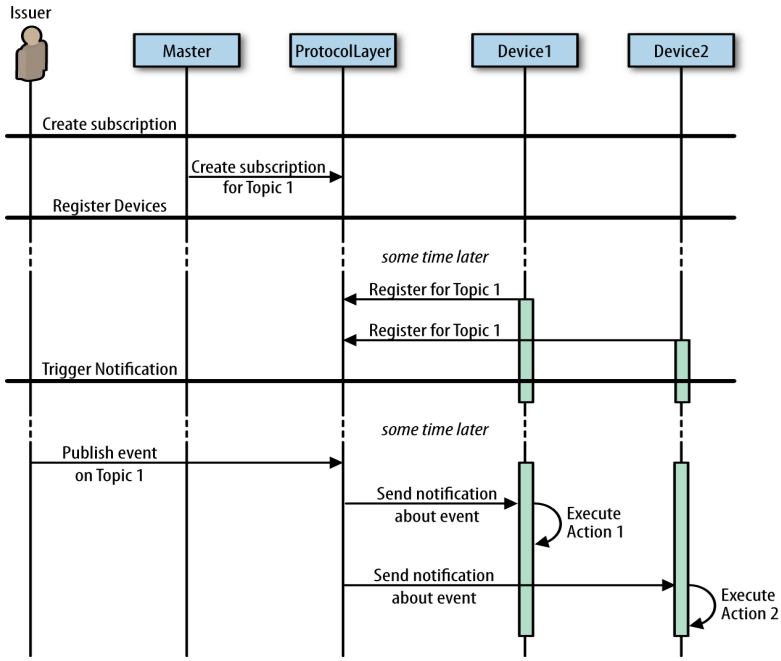


Figure 8: UML sequence diagram illustrating backends and device communication – source: (Mijic, 2018)

Apache Kafka

Kafka is an event. Streaming platform that combines messaging, storage and stream processing to secure, reliable and highly scalable infrastructure pipeline. Kafka enables integration with any source or sink. Kafka allows continuous streaming and processing. The main benefits according to (WAEHNER, 2019) are stream processing, high throughput and good integration with the rest of the enterprise.

As mentioned at the beginning there are two options that can be used for communication.

MQTT Proxy for data ingestion

This proxy allows organizations to eliminate the additional cost in comparison with MQTT broker. MQTT Proxy accesses, combines, and guarantees that IoT data flows into the business without adding additional layers of complexity.

It's horizontally scalable, consumes and pushes data from IoT devices and forwards it to Kafka Broker. The Kafka broker is the source of truth responsible for persistence, high availability, and reliability of the IoT data.

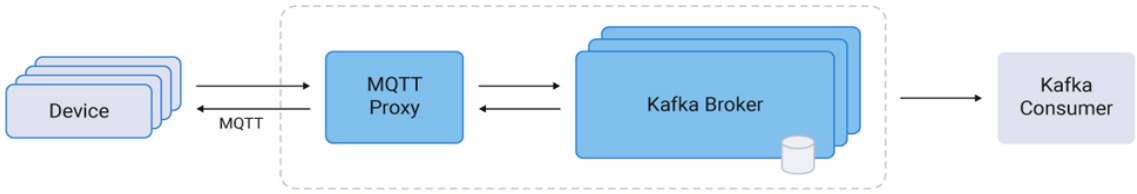


Figure 9: MQTT Proxy integration with broker - source: (WAEHNER, 2019)

REST Proxy for IoT Integration

Implementation with IoT through HTTPs is usually much faster due to easier deployment and technology complexity. HTTP is push based so it makes security much simpler. Scalability is controlled in a standart manner through load balancer and it supports up to 1000 requests / second.

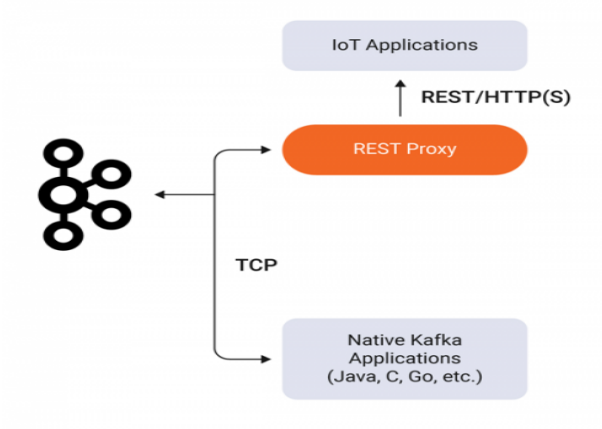


Figure 10: REST proxy with Kafka Broker - source: (WAEHNER, 2019)

3.5.4 Reactive Microservices

The term “reactive” refers to programming models that are built around reacting to change, usually network components react to I / O Events, UI controllers react to user inputs and others. Main purpose of this paradigm is to be able to react to events as the operations complete or data becomes available.

The key expected benefit is ability to scale with a small, fixed number of threads and less memory. That makes applications more resilient under load, because they scale in a more predictable way. In order to observe it you need to have the right use case for this

scenario. This typically refers to a scenario when service A waits for result from service B and after that calls service C.

In such microservices it‘s assumed that applications do not block, therefore non-blocking servers use a small, fixed-size thread pool event loop workers) to handle requests. (Spring Foundation, 2020)

3.5.5 Quality Metrics – ISO/IEC 25023

Standard ISO / IEC 25023 defines metrics of quality for quantitative measuring of software product from the view of individual characteristics described in ISO / IEC 25010.

(ISO/IEC, 2016) Software product quality can be evaluated by measuring internal properties, or by measuring external properties (typically by measuring the behaviour of the code when executed), or by measuring quality in use properties (when the product is in real or simulated use). Appropriate internal properties of the software are a prerequisite for achieving the required external behaviour and appropriaty external behaviour is a prerequisite for achieving quality in use.

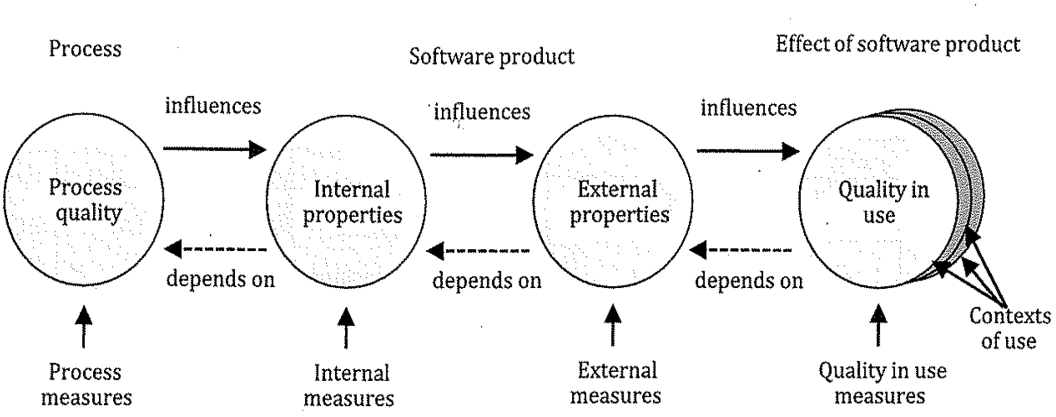


Figure 11: Relatinonship between types of quality measures - source: (ISO/IEC, 2016)

From the point of view of topic scalable distributed cloud native applications, it makes sense to focus on relevant metrics from the area of performance efficiency and reliability.

3.5.5.1 Performance Efficiency measures

Performance efficiency measures are used to assess the performance relative to the amount of resources used under stated conditions. Resources can include other software products, the software and hardware configuration of the systém, and materials (e. g. Storage media)

Table 2: Performance efficiency measure metrics ISO / EIC 25023 - source: (ISO/IEC, 2016)

ID	**Name**	**Description**	**Measurement function**
PTb-1-G	Mean response time	How long is the mean time taken by the system to respond.	$X = \sum_{i=1\ to\ n} (A\,i) \,/\, n$ Ai = time taken by system to respond. n = number of responses measured
PTb-2-G	Response time adequacy	How well does system response time meet the specified target ?	$X = A \,/\, B$ A = Mean response time. B = Target response time specified

3.5.5.2 Availability measures

Availability measures are used to assess the degree to which a system, product or component is operational and accessible when required use.

Table 3: Availability measures metrics ISO/EIC 25023 - source: (ISO/IEC, 2016)

ID	**Name**	**Description**	**Measurement function**
RAv-1-G	System availability	For what proportion of the scheduled system operational time is	$X = A \,/\, B$

		the system available ?	A = System operation time actually provided. B = System operation time specified in the operation schedule.

3.5.5.3 Fault tolerance measures

Fault tolerance measures are used to assess the degree to which a systém operates as intended despite the presence of hardware or software faults.

Table 4: Fault tolerance measures metrics ISO/EIC 25023 - source: (ISO/IEC, 2016)

ID	Name	Description	Measurement function
RFt-1-G	Failure avoidance	What proportion of fault patterns has been brought under control to avoid critical and serious failures?	$X = A / B$ A = Number of avoided critical and serious failure occurences (base on test cases) B = Number of executed test cases of fault pattern, during testing
RFt-3-G	Mean down time	How long does the system stay unavailable when a failury occurs	$X = \sum_{i=1\ to\ n} (A\,i - Bi) / n$ Ai = Time at which the faul tis reported by the systém Bi = Time at which faul tis detected n = number of faults detected

3.5.6 Continuous Delivery

Continuous delivery is a set of practices and disciplines in which software delivery teams produce valuable and robust software in short cycles. Functionality is added incrementally and the software can be reliably released at any time. This maximizes the opportunity for rapid feedback and learning, both from a business and technical perspective.

Automation of Releases

The build pipeline must provide rapid feedback for the developers in order to support daily work cycles related to code changes. Operation of the pipeline must be highly repeatable and reliable. The goal is to achieve 100% of automation or as close you can get to it. Here is list of steps that should be always automated:

- Compilation and static analysis of code
- Testing (chapter 3.5.7)
- Provisioning of all environments
- Monitoring and alerting systems in place (chapter 3.5.8)
- Data store migrations
- HTTP testing, performance and security testing
- Tracking and auditing history of changes

3.5.6.1 Development Pipeline

In the Figure 13 is shown an example of a typical build pipeline. It starts with a process of continuous integration. Code that developers write is continuously committed to a shared repository where it‘s managed by version control. This gets automatically built and packaged into an artifact. After This stage is finished it‘s submitted to a series of automated tests and HTTP quality attribute verification stages, before going through manual testing. Then build is promoted to the stage environment where the configuration should ideally mirror the production environment. Finally then it‘s promoted to production and delivered between customers. (Daniel Bryant, 2018)

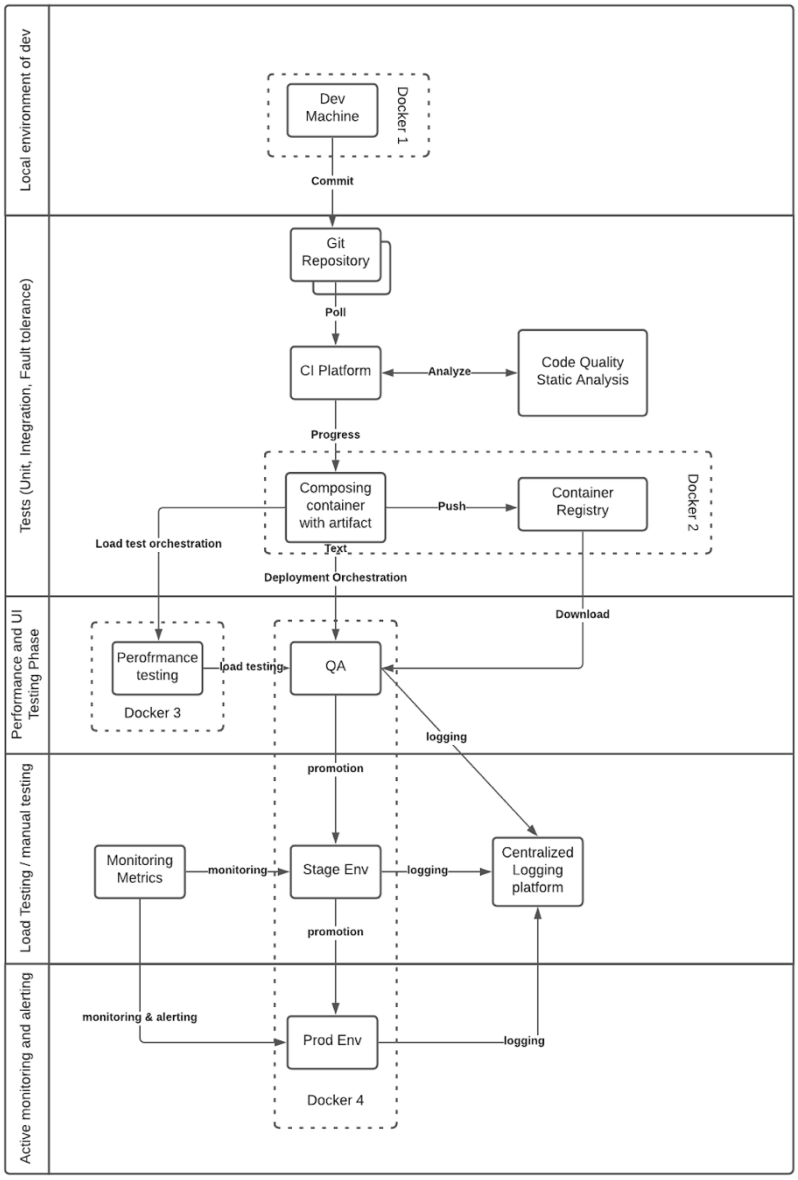


Figure 12: CI / CD Pipeline concept - source: (author)

3.5.7 Testing

Every piece of code that is deployed to the production environment among customers needs to be thoroughly tested. Velocity of deployments and releases for cloud native solutions tends to be high and you can no longer rely just on doing manual tests. In order to increase release velocity with confidence that changes will not break anything, you need to have solutions properly covered with automated tests.

Most of the functionality should be tested locally through unit tests and integration tests. It can ensure that all components work well together and a smaller portion of test coverage should be on the UI layer. Point of this effort is to reduce long running and manual tests.

Test Doubles

For most of the testing are used mocked objects. Mocked objects are something that simulates functionality of real objects. For example using a mocked credit card or authorization token so you don‘t have to use real instances of those for your testing. The common three of those types are mocks, stubs and fakes.

With Mocks you define specific expectations of what should be returned from the object. Main purpose of using mock is to test interactions between objects. For example setting up a mock of a dependent payment service that on invoking returns certain expected values.

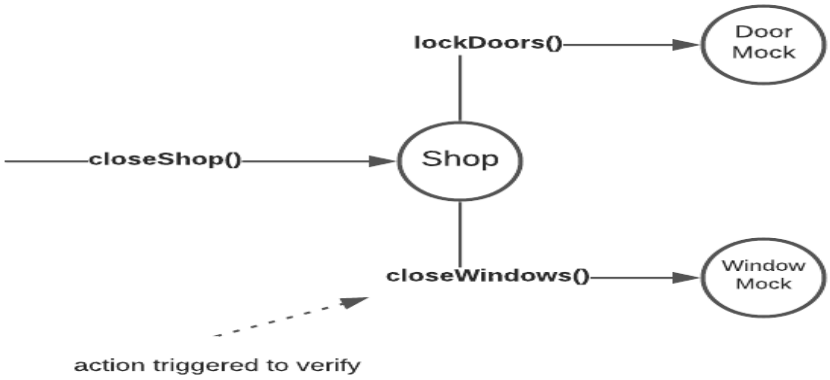


Figure 13: Mock example (source: author)

Stubs don‘t contain any logic, they simply return values defined by the test author. They‘re useful when you need to simulate the specific state of the object.

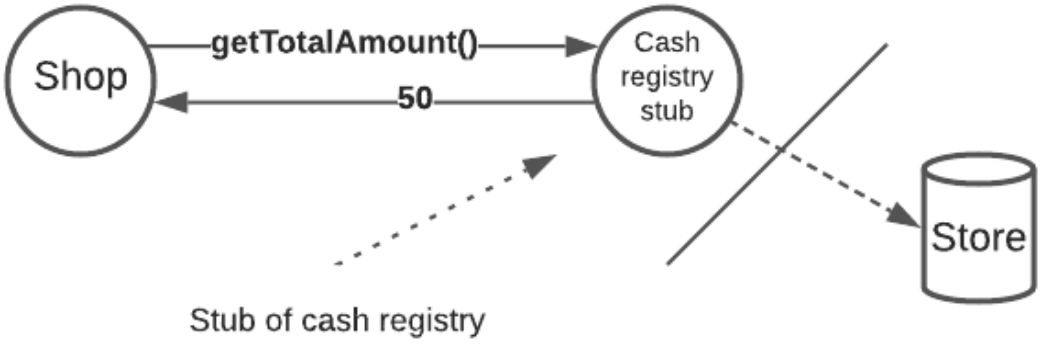


Figure 14: Stub example - source: (author)

Last of those three types is fake. It‘s typically a simple implementation of API that behaves like a real thing. This can be very helpful especially for stabilization of the tests when services you deal with are difficult to automate testing on or they‘re not very stable and therefore without using such a thing as a fake, it would cause instability of tests.

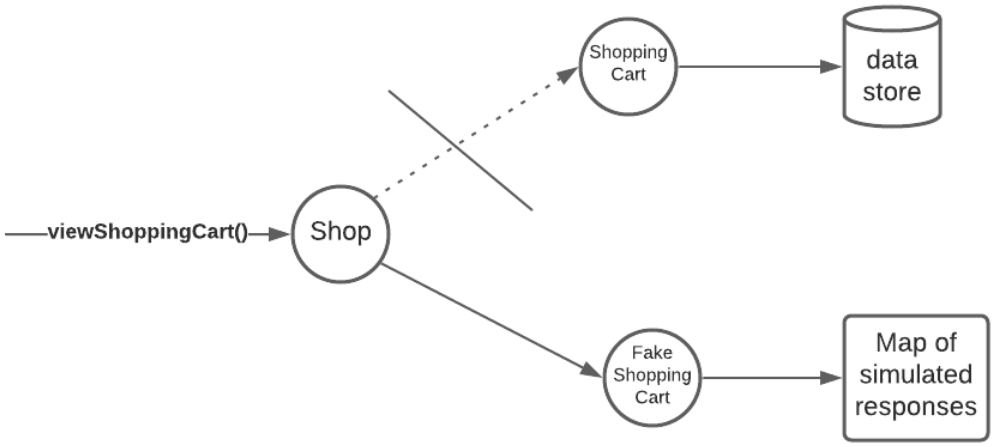


Figure 15: Fake example (source: author)

3.5.7.1 Test Automation Pyramid

Testing Pyramid is a framework that helps developers and quality assurance engineers to ensure quality of software keeps very high as the codebase grows. It reduces time

required for developers to identify if newly introduced code could cause any breaking changes. It‘s a helpful framework to build more reliable test suites.

Pyramid consists of 3 layers that should be included in an automated test suite. It also outlines the sequence and frequency of these tests. Point of this is to provide immediate feedback to ensure code changes do not disrupt service. (Bose, 2020)

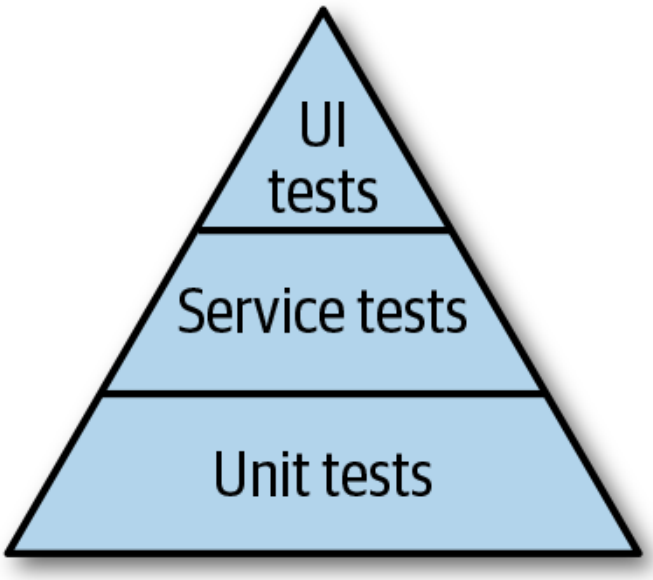


Figure 16: Test Automatition Pyramid – source: (Bose, 2020)

3.5.7.2 Unit Testing

In the context of microservice, unit test covers only the given service that it‘s designed to test. In general there are many unit tests in each service.

A unit consists of a line of code, method or a class. Unit testing refers to testing a particular unit for a piece of code. Best practices say that unit tests should be kept small and very specific in what it tests. Aim of developers is to have a large percentage of coverage. Unit tests should be designed to verify if the methods or classes behave as developers expect. During the writing or modifying of code it‘s very useful for developers to constantly verify if the design does what it‘s supposed to.

Unit testing alone doesn‘t determine the behavior of the HTTP. Unit tests have good coverage of each of the core modules of the HTTP in isolation. To verify that each module correctly interacts with its collaborators, more coarse-grained testing is required.

3.5.7.3 Service Testing (Integration)

Distributed systems are typically made out of multiple services. Integration is defined as the set of interactions among those services. Testing interactions between services with external systems is called integration testing. Integration testing starts when two of the components are available and ends when all component interfaces have been tested. The final round of integration involving all components is called HTTP integration.

Architecture and design gives details of interactions within the systems however, testing the interactions between one HTTP and another requires detailed understanding how they work together. This introduces complexity in procedures and in what needs to be done. Recognizing this complexity, a phase in testing is dedicated to test these interactions, resulting in the evolution of a process. This ensuing phase is called the integration testing phase. (Srinivasan Desikan, 2007)

3.5.7.4 UI Tests

At the top of the pyramid are UI tests. Among these 3 test types that are essential in testing of the HTTP, UI tests should represent the fewest number of tests of all types. In general UI tests are costly to write and difficult to maintain however, they are useful during testing usability and accessibility. Example of UI tests would be the scenario of simulating the end user process of achieving a certain business use case. It could start like opening the browser, then go to a certain section and trigger the buy process in the HTTP. In this example we can very well test not only if the functionalities work but also if the functionalities are accessible in all browsers and their response times.

3.5.7.5 Performance Tests

Performance tests are used to test how the application or HTTP performs and measure it. This could mean for example how long it took to execute a certain scenario. You can write performance tests on multiple levels. It can be on the function or unit level to measure how long a function or request takes to respond. Performance tests often help to pinpoint bottlenecks and allow us to investigate why functions HTTP a certain amount of time to execute. It‘s an excellent way to track performance of the HTTP and establish baselines to which it compares to. This allows also for measuring the difference between versions of the HTTP as the developments continue and identify mistakes or newly created bottlenecks that might block successful execution of other functionalities. Depending on how critical performance to the HTTP, It can be specified what should be expected and measure outcomes on different environments.

3.5.7.6 Load Tests

Those are types of performance tests that have purpose to determine the behavior of the HTTP under certain conditions. These conditions can for example simulate typical load on the HTTP that it‘s under most of the time or exceptional peaks that it goes through during certain periods of the day. This is a good way to determine what is the maximal load that the HTTP can handle and identify the breaking point. Based on those results it‘s common to define alerting thresholds for monitoring systems.

3.5.7.7 Apache JMeter

To perform load and performance test there can be use tools such as JMeter (Apache Software Foundation, 2011) describes it as tool for test performance both on static and dynamic resources, Web dynamic applications.

It can be used to simulate a heavy load on a server, group of servers, network or object to test its strength or to analyze overall performance under different load types.

Functionalities:

Ability to load and performance test many different applications/server/protocol types:

- HTTP, HTTPS (Java, NodeJS, PHP, ASP.NET, …)
- SOAP / REST Webservices
- Database via JDBC
- Message-oriented middleware (MOM) via JMS
- Mail - SMTP(S), POP3(S) and IMAP(S)
- Native commands or shell scripts
- TCP
- Java Objects

3.5.8 Monitoring

Monitoring remains a critical part of managing the distributed HTTP. Especially microservices-based applications have different, more intensive, monitoring requirements. Necessity of monitoring is evident because sooner or later the applications are going to fail. In general systems are not just binary either up or down but also can operate in degraded state that impacts performance. These degraded states have a direct negative impact on the end user. Monitoring the behavior of systems can alert operators to make adjustments in time and prevent fatal failure of the HTTP.

3.5.8.1 Service Metrics

All systems generate a large number of metrics for us to track. This can be found immediately when metrics collecting software is installed. It gives useful information like response times of processes or cache hit rates and other.

In general most of the frameworks for writing webservices have built in libraries that expose basic metrics themselves. As a bare minimum is to track error rates and response times of web servers or better of individual endpoints. These metrics inform us of how our systems behave and also about communication with external services and users.

Metrics should also be visualized into human-readable form like graphs, so they can be properly analyzed. For this there are many open source platforms like for example Grafana.

3.5.8.2 Basic Metrics

Error rate

This metric should indicate rate failing requests (4xx, 5xx HTTP responses).

Incoming request rate

Usually measured in HTTP requests per second (or reads/writes/transactions per time unit if this is a database), it indicates how much traffic is coming into your HTTP.

Latency

Latency is the time it took for your service to process a request. The latency is usually broken down to successful and unsuccessful requests.

Utilization

Utilization gives you information about the usage of different pieces of your system. For example, you would monitor utilization of the nodes in the Kubernetes cluster making sure memory, disk, and CPU usage are in normal ranges.

Recommendations:

- Track response times and error rates for each service.
- Aggregate host-level metrics like CPU together with application-level metrics.
- Standardize format of collecting metrics.

- Track health of all downstream responses.
- Have a single queryable tool for aggregating and storing logs.
- Ensure your metric storage tool allows you to maintain data long enough to understand trends in your HTTP.
- Have data visualized in graphs in a way that makes possible to read and understand them properly.

Prime importance is to collect relevant data and ability to anylize. Applications should be instrumented by developers to report application-specific events. In (Swersky, 2018) article is stated that operations teams must gather data not just from applications, but from the supporting platforms and deployment systems. Open source and paid solutions are available to support both publication and storage of monitoring events. This data is critical to support a distributed system that is resilient, reliable and highly available.

3.5.8.3 Alerting

Core functionality of an alarm is to trigger detection of abnormality in time series. Alerting is essential functionality in most monitoring platforms. It should have a flexible, feature-rich plotting engine that supports for graphing multiple time series and include an alerting engine that supports sophisticated alarm configurations like aggregation and suppression.

Process of alerting is full of unstable variables of a qualitative nature, and it presumes an element of responsibility. Priorities are open to interpretation, but the level of severity usually depends on what's at stake. The extent of pressure involved in incident response varies from organization to organization, but the overal process has a common pattern. (Ligus, 2012)

Goal of alerting is to notify operators when following conditions occur:

- Increase response times
- Loss of availability
- Surpassing ordinary error rate

Operator who receives an alert has the task to isolate and identify the source of the problem and mitigate the impact in the shortest time possible. Challenge in alerting is defining the right levels of sensitivity this may often lead to false alarms.

Recoverability and impact into three levels each, as depicted on picture below, to describe the severity of undesired events. The matrix illustrates a classification of events into nine separate bins, from most to least severe.

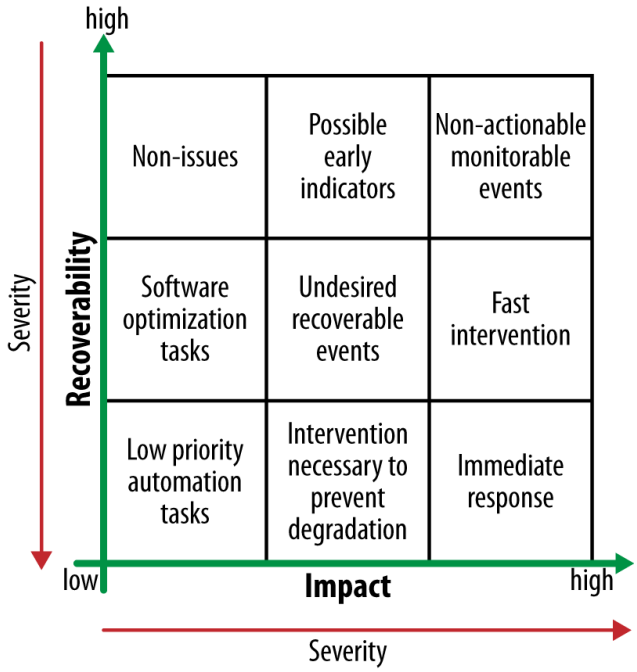


Figure 17: Recoverability / Impact matrix- source: (Ligus, 2012)

3.5.9 Distributed Tracing

Distributed request tracing is a method used to profile and monitor applications. It helps to identify failures in the HTTP and causes of degraded performance. It‘s designed to debug and monitor modern distributed software architecture such as microservices.

Systems behave differently under load and at scale. The specification of the HTTP‘s behavior often diverge from the actual behavior of the HTTP. It‘s important to contextualize requests as they transit through the HTTP. Tracing is a very simple concept where requests flow from one service to another in the HTTP, through ingress and egress points, tracers add logic where it‘s possible to perpetuate a unique identifier that‘s generated when the entry request is made. As the request transits it gains a new span assigned from another component that gets added into the trace. Trace represents the whole journey of the requests throughout the HTTP. Span represents individual hop along the way and contains certain tags and metadata used for contextualizing requests. (Long, 2016)

3.5.10 Open Tracing

Main goal is to describe the semantics of transactions in distributed systems. Describing those transactions should not be influenced by any particular backend way like to 60roces or represent data.

Traces are defined explicitly by their “Spans”. Trace can be thought of as a directed acyclic graph of Spans, where the edges between Spans are called “References”.

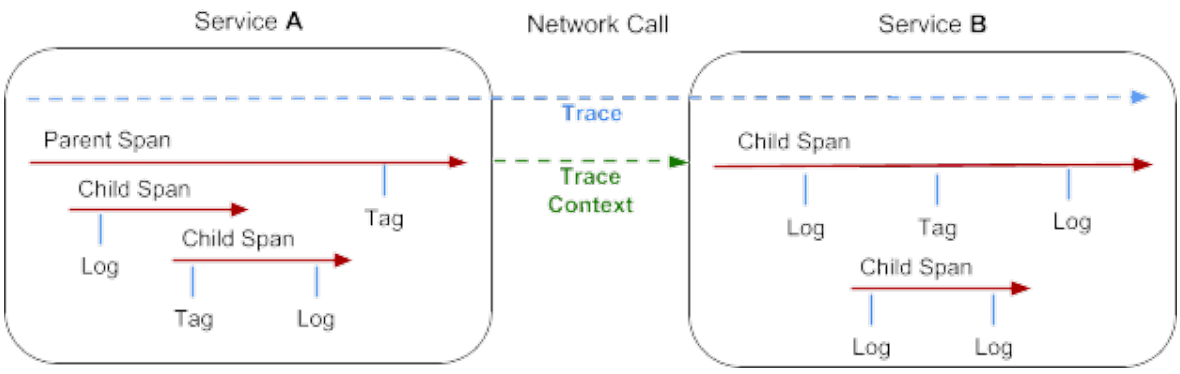


Figure 18: Tracing data model - source: (Quan, 2019)

Span

Each span as a unit inside the acyclic graph encapsulates following metadata:

- An operation name
- A start timestamp
- A finish timestamp
- Key / pair values of Span tags (custom metadata)
- Span Context

A Span may reference zero or more other contexts that are causally related. OpenTracing defines two types of references. One of them is “ChildOf” that points at the previous Span and second is called “FollowsFrom”. (Cloud Native Computing Foundation, 2016)

Zipkin

Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data. (OpenZipkin, 2017)

4 Practical Part

4.1 Data Collection & Analysis

To address questions about potential for speed of adoption of cloud native applications among developers, a descriptive research survey was used. Method for data collection was an anonymous questionnaire with colleagues from Avast Software.

4.1.1 Survey

All the research questions were contained in the survey. There were three reasons for choosing anonymous self-completed surveys outlined in chapter 3.1.6.2 to collect qualitative and quantitative information.

Tool used for gathering data was Google Forms. Google forms allows us to set up and manage surveys that are quickly shareable through links. The survey had 45 attendees. Every participant had the opportunity to self identify his profession at the beginning of the survey.

Main goal is to access willingness to learn, general expertise and opinions around trying new technologies product development, cloud technologies and practices.

4.1.1.1 Research questions

Table 5: List of survey questions

	Question
1.	How much of your work time do you dedicate to learning new things ?
2.	What is the source of acquiring new skills for you ?
3.	Have you in the last few years encountered some technology that would significantly impact your work ? If yes, please write the name and impact it had.
4.	What do you see as key aspects for delivering good SaaS software ?
5.	Rate impact of having automatised CI on your project.
6.	What are the key factors / requirements that you look for when choosing a solution on which to deploy an application ?
7.	Have you ever used a public cloud provider platform for building any application ?

8.	If the answer is yes, which one ?
9.	Name key benefits you've identified using some of the cloud provider platforms over self managed infrastructure.

4.1.1.2 Survey results

In the graph from Figure 19 we can see that largest groups of participant's profession was backend developers, fullstack developer and quality engineers.

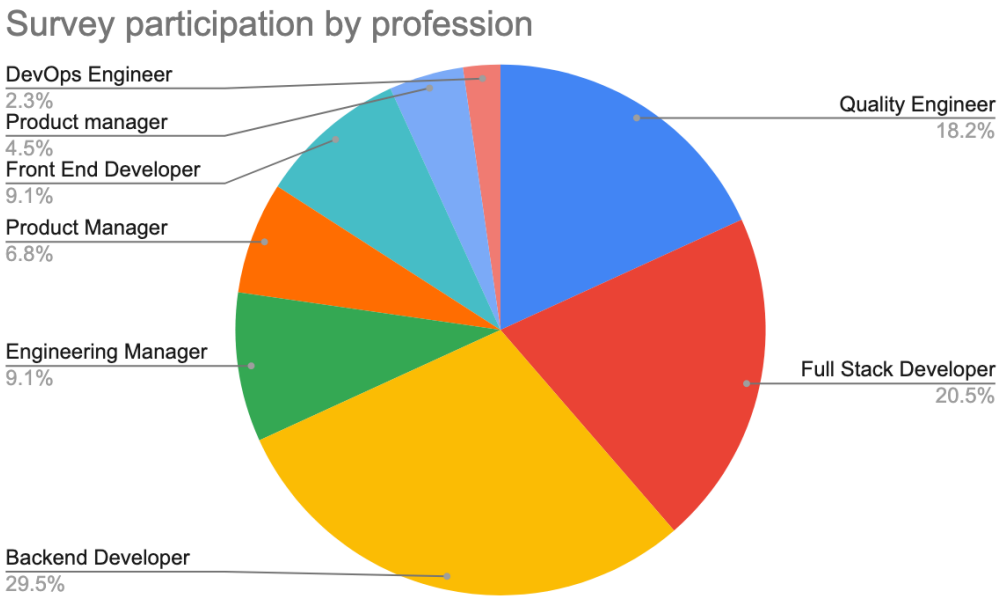


Figure 19: Participation statistic by profession – source: (author)

This section explores explanatory analysis of professionals to learn new technologies.

Result of the questions, how much time do people involved in product development spend by studying new technologies and acquiring new knowledge? that largest group of respondents which consists of 45,5 % of all responses spends 10-20%. off their working time on learning new things. Average time for all participants is 14,28 % of their work.

Intervals [%]	Frequency	Midpoint	Midpoint * Frequency	Midpoint - Mean	(Midpoint - Mean)^2	(Midpoint - Mean)^2 * Frequency
0—5	5	2,5	12,5	-11,77777778	138,7160494	693,5802469
5—10	12	7,5	90	-6,777777778	45,9382716	551,2592593
10—20	20	15	300	0,722222222	0,521604938	10,43209877
20—40	8	30	240	15,72222222	247,1882716	1977,506173
N	45	55	642,5			
Mean	14,27777778					
Sample Variance	73,47222222					
Sample Standart Deviation	8,571593914					

Figure 20: basic statistical calculation table – source: (author)

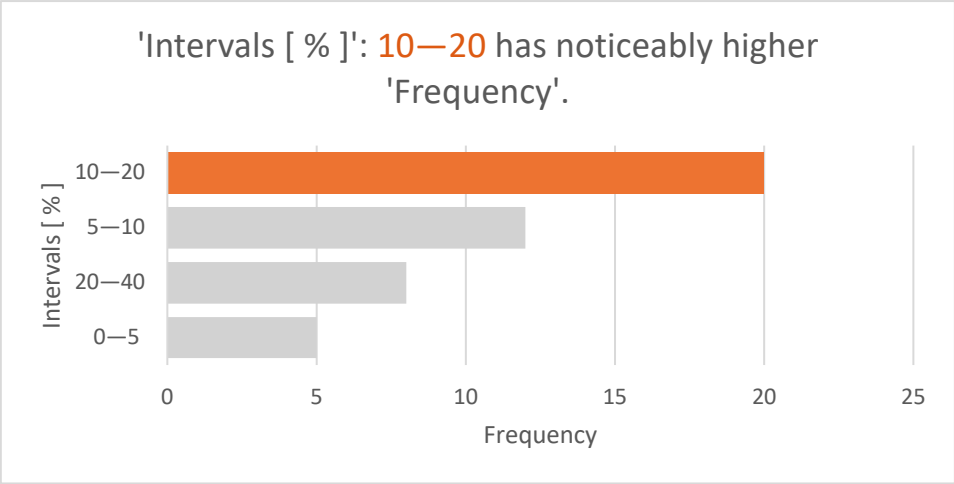


Figure 21: Graphical representations distributed by groups – source: (author)

Next question focuses on what are the sources of acquired knowledge. Assumption is that for most of the people it might be combination of conferences, reading books, experimenting with emerging technology and knowledge transfer among colleagues.

We can also assume that for most of the people it's usually combination of multiple of multiple sources, therefore the question has multiple answers.

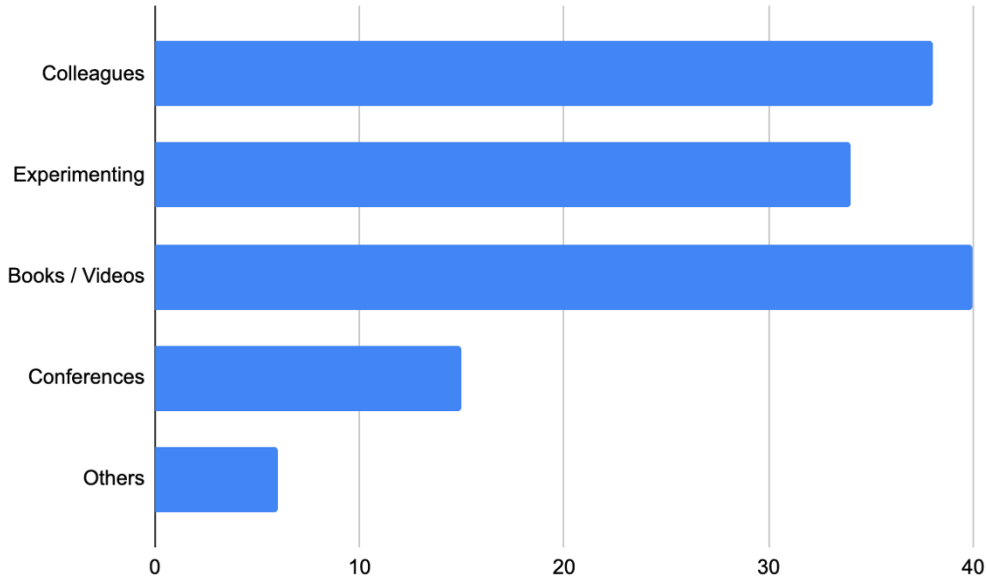


Figure 22: Sources of learning for professionals – source: (author)

As we can see in Figure 22, the most common sources of new knowledge are ideas shared between colleagues, experimenting and books or videos. Most common combination of answers were options Colleagues, experimenting and books.

In following question when respondets are asked about encountering technology that meaningfully impacted their day to day work life. First due to nature of the answers, data were labeled into categories.

Table 6: Question 3 - Categorical answers results

Answer category	**Total respondents who answered**	**% of respondents who answered**
Docker	15	29 %
Packaging tools	4	8 %
Orchestrators	7	13 %
Public Clouds	12	23 %
Virtualization	4	8 %
Deployment Tools	4	8 %
Web Frameworks	4	8 %
Integration Frameworks	2	4 %

Based on the answers we can say that containerization that is described in chapter 3.2. respondents sense as impactful. Second the most meaningful technology are based on responses orchestrators (chapter 3.3). Third is the opportunity of using public clouds.

From the answers we can conclude that the most significant difference in product development in the last couple years had development of containerization and orchestration technologies. This is very likely connected to the impact they have on integration of product changes and testability of software.

This question is exploring opinions regarding what's important in SaaS product delivery. In Table 7 results show that most of the respondents seem as key to success product delivery reliability of the services, quality and testability.

Table 7: Question 4 - Categorical answers results

Answer category	**Total respondents who answered**	**% of respondents who answered**
Scalability	5	7 %
Availability & Reliability	24	34 %
Quality & Testability	18	25 %
KYC, UX	8	11 %
Security	9	13 %
Cost	7	10 %

Following question is connected to automation. As stated in chapter 3.5.6, automation pipeline is mission critical for most of the projects. This confirms also responses from survey where average of importance is 4,2 out of 5 from 44 responses as we can see in Table 8.

Table 8: Q5 - descriptive analysis of responses

IMPORTANCE OF CI PIPELINE IN PROJECT	
Mean	4,272727273
Standard Error	0,139326515
Standard Deviation	0,924187547
Sample Variance	0,854122622
Minimum	2
Maximum	5
Sum	188
Count	44
Confidence Level(95,0%)	0,280978696

Next questions were more focused on finding out whether respondes have any experience with cloud platforms and what they are looking for when choosing provider for deployment their solutions.

Table 9: Q7 - result

Answer	**Count**
Yes	29
No	15
Grand Total	**44**

The answers indicate that almost 2/3 of respondents have at some point used public cloud for deployment of software. Next question is exploring preferred providers among the respondes.

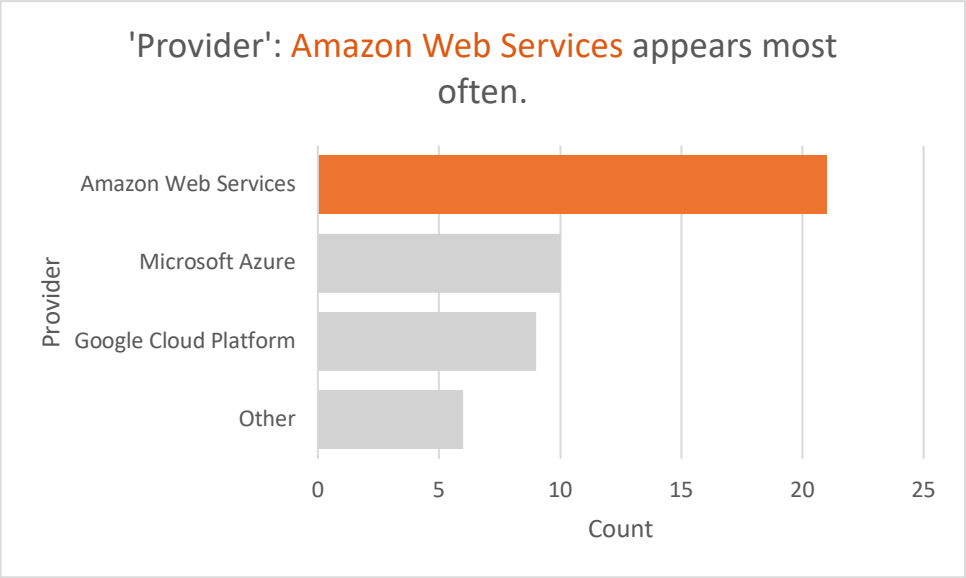


Figure 23: Most popular provider among respondes – source: (author)

Most common choice was AWS, according to comments it provides all requested services for a fair price. According to (Stalcup, 2021), AWS dominates the market with 31 % share among the cloud providers.

Last question is focused on what are the key factors when choosing a public cloud provider over self managed deployments. This question is open so the answers were labeled in categories.

Table 10: Question 9 - answers evaluation

Answer category	**Total respondents who answered**	**% of respondents who answered**
Observability	8	17 %
Scalability & Performance	11	24 %
Reliability & SLA	19	41 %
Faster Deployments	8	17 %

Based on results, the main benefit of using public cloud is reliability and the service-level agreement that clearly states quality, availability and responsibilities of the service that customer is using.

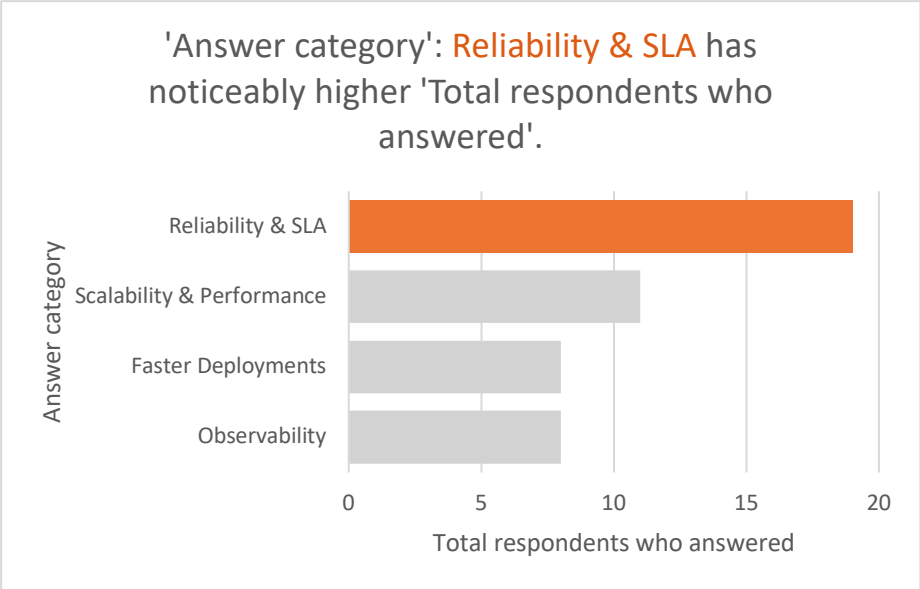


Figure 24: Preffered benefits of Cloud – source: (author)

Overall result indicate that professionals in field of software product developments are positively accepting the shift that's happening with delegating more responsibilities on cloud providers and being able to focus more on their „core business".

4.2 Cloud Provider Comparison

This part of the practical thesis focuses on taking in count all the key variables mentioned in chapter 3.1.6, affecting choice of cloud provider. Goal of this is to estimate the best offering based on publicly available pricing tables.

In the cloud provider market the largest 3 service providers are according to (Stalcup, 2021), Amazon with 33 % of market share, second Microsoft with 20 % of market share and third Google with 7 %.

The thing that's not evaluated in this analysis is the fact that most of the large organizations that use IaaS, SaaS or PaaS commonly are charged based on individual pricing model.

4.2.1 Managed Services

First metric to mention are managed services offered on platforms. As we can see in Table 11, all of the major players on the market offer the same set of essential features with having them named and using different implementations.

Table 11: Essential services terminology

Service	AWS	Azure	GCP
Compute	Elastic Cloud Compute	Virtual Machines	Compute Engine
App Hosting	Elastic Beanstalk	Cloud Services	App Engine
Serverless Computing	AWS Lambda	Azure Functions	Cloud Functions
Container Support	ECS/EKS Containers	AKS Container Service	Kubernetes Engine
File Storage	S3 Storage Service	Azure Storage	Cloud Storage
Block Storage	Elastic Block Storage	Azure Blob	Persistent Disc
Backup Options	AWS Glacier	Azure Backup	Cloud Storage
Data Orchestration	Data Pipeline	Data Factory	Cloud DataFlow
Data Management	AWS Redshift	SQL Data Warehouse	Google BigQuery
NoSQL Database	DynamoDB	Cosmos DB	Cloud DataStore

Prices of these services are various, although they serve the same purpose the implementation and choice of technology may favour one over the other.

When it comes to the pricing in these areas it's very specific, for example some providers calculate writes / reads / updates in databases and so on. For those purposes each

provider offers a calculator that based on the services you add in computes the estimated price.

4.2.2 Computation Power & Memory

Table 12 shows converted prices into the same metrics. It focuses on 2 of the most saw upon aspect of virtual machines and their computation power and memory. The prices show the cost of running various virtual machine sizes per hour.

Table 12: vCPU & RAM / per hour - pricing comparison

Type	vCPU	RAM	AWS	Azure	GCP
General Purpose	2	8GB	$0.0928	**$0.0850**	$0.1070
	4	16GB	$0.1856	**$0.1700**	$0.2140
	8	32GB	$0.3712	**$0.3390**	$0.4280
Compute Optimized	2	4GB	$0.0850	$0.0850	**$0.0813**
	4	8GB	$0.1700	$0.1690	**$0.1626**
	8	16GB	$0.3400	$0.3380	**$0.3253**
Memory Optimized	2	16GB	**$0.1330**	**$0.1330**	$0.1348
	4	32GB	**$0.2660**	**$0.2660**	$0.2696
	8	64GB	**$0.5320**	**$0.5320**	$0.5393

Despite the assumption of AWS having the biggest market share, therefore due to the economy of scale should have lowest prices, it appears that the lowest cost for virtual machines of various sizes has Microsoft Azure.

When it comes to various instances of virtual machines, commonly cloud providers offer to pay upfront for a certain size and price with large discounts. This might be tricky though because of the advancements that are constantly happening in computing. In general the standard billing model is „pay as you go“ where you pay for the resources that you use.

4.2.3 Storage

Table 13 shows comparison of storage pricing for all the providers per gigabyte of space. The price may slightly vary based on the location of the storage. In this category the cheapest offer has Microsoft Azure.

Table 13: Storage pricing per GB - comparison

Platform	Product	Price
Azure	Standard (GPv2) storage	**$0.0183**
AWS	Amazon S3	$0.024
GCP	Cloud Storage	$0.023

4.2.4 Network (Ingress & Egress)

Following costs are based on two billing elements. First is outbound data transfer and second port hours. The cost might slighly vary based on the location of the trafic. All the providers have specified zones and the tarifs that apply on certain territories. In Table 14 we can see average price for the 1 GB connection.

Table 14: Ingress & Egress Port-hour / 1 GB connection price - comparison

Type	AWS	Azure	GCP
Egress	$0.30	**$0.224**	0.2778
Ingress	free	free	free

Based on the metrics we defined in chapter 3.1.6.1, Microsoft Azure cloud offers the most favourable prices. Needless to say that the standard prices can drastically vary from upfront billing or individual deals that large organizations get. In general there is a rule that the more you consume the less you pay per unit. When it comes to software support there is a large offering for all of them but specific cases may prefer different implementations.

4.3 Prototype Cluster

Aim of this chapter is to use all the best practices describe around building cloud native applications that are data intensive and scale horizontally, described in chapter 3.5 in the theoretical part and construct cluster of containerized microservices written in programming language Java, running in Kubernetes orchestrated cluster deployed in Google Cloud Platform.

Prototype application is a solution of warehouse management software. In warehouses there are typically many information exchanges between machines. Therefore the prototype aim is to test how to enable durability and scaling but at the same time being able to increase testability and observability.

4.3.1 Solution Architecture Specification

In chapter 3.4.2 were discuss decision criteria of choosing proper architectural structure for the software solution. In this case data intensive solution is being design. Taking all the criterias in count there was proposed following architecture (
Figure 25).

This software solution consists of 5 microservices described in Table 12.

Table 15: Services list

Service	**Function**
Sensor Service	Recieves and stores data from the sensors placed in the warehouse.
Device Service	Controls and remotely manages devices in warehouse.
Report Service	Aggregates infrormations from devices and sensors into actionable reports.
Warehouse Service	Represents management of quantities and items in warehouse.
User Service	Holds system user details with roles and permissions.

To achive higher throughput of informations services communicate with each other through combination of synchronous and asynchronous channels that's closer described in chapter 3.5.3.

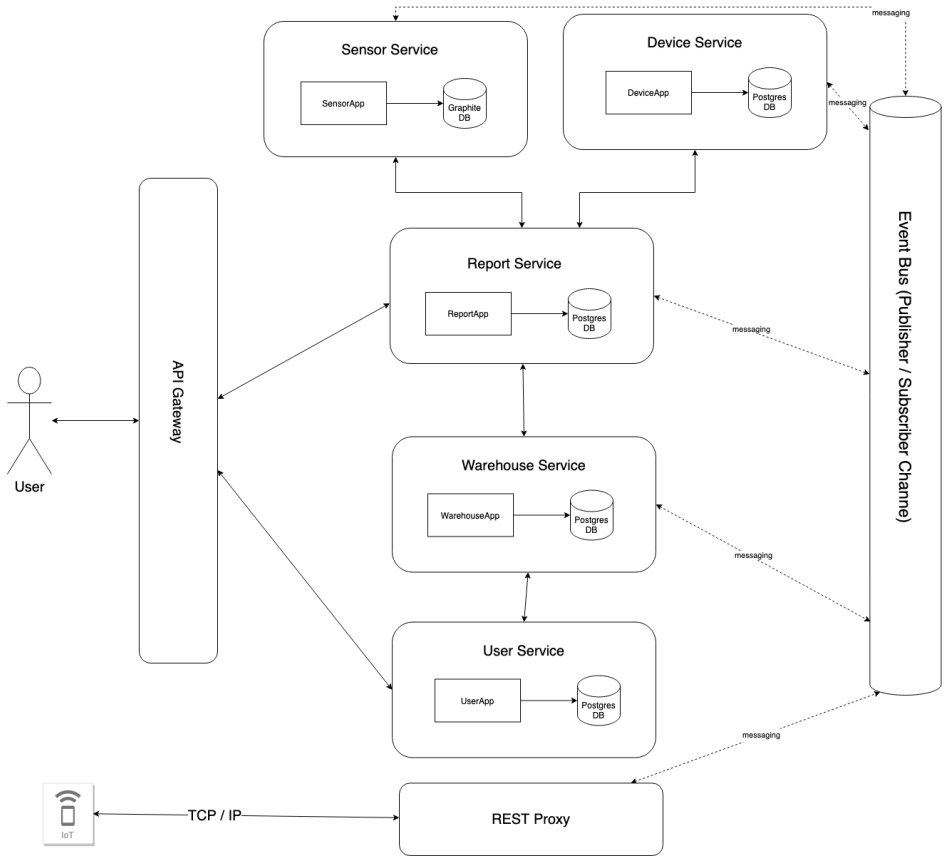


Figure 25: Development Environment Architecture – source: (author)

4.3.1.1 Implementation of Service Communication

To connect with IoT devices we using pattern of REST proxy. Advantages of this decision are shown in chapter 3.5.3.4. There are many ways and protocols that can be implemented but it's usually complicated to set up and it takes more time to think through correctly. In this pattern device communicates to REST proxy and then asynchronous message is published to Kafka Broker. Then service that is suppose to subscribe to this message does so.

OpenAPI Definitions

Definitions of API is documented through the OpenAPI 3.0 that's further described in chapter 3.5.2.2. This way of documenting allows easy integration for other services calling the API and removes guesswork.

Going further based on this standart definitions in Java through plugin can be generated web client that speeds up proces of integration and reduces amount of code programmer has to write do implement communication between services.

Using OpenAPI 3.0 also provides with having very clearly defined API contracts in human readable manner.

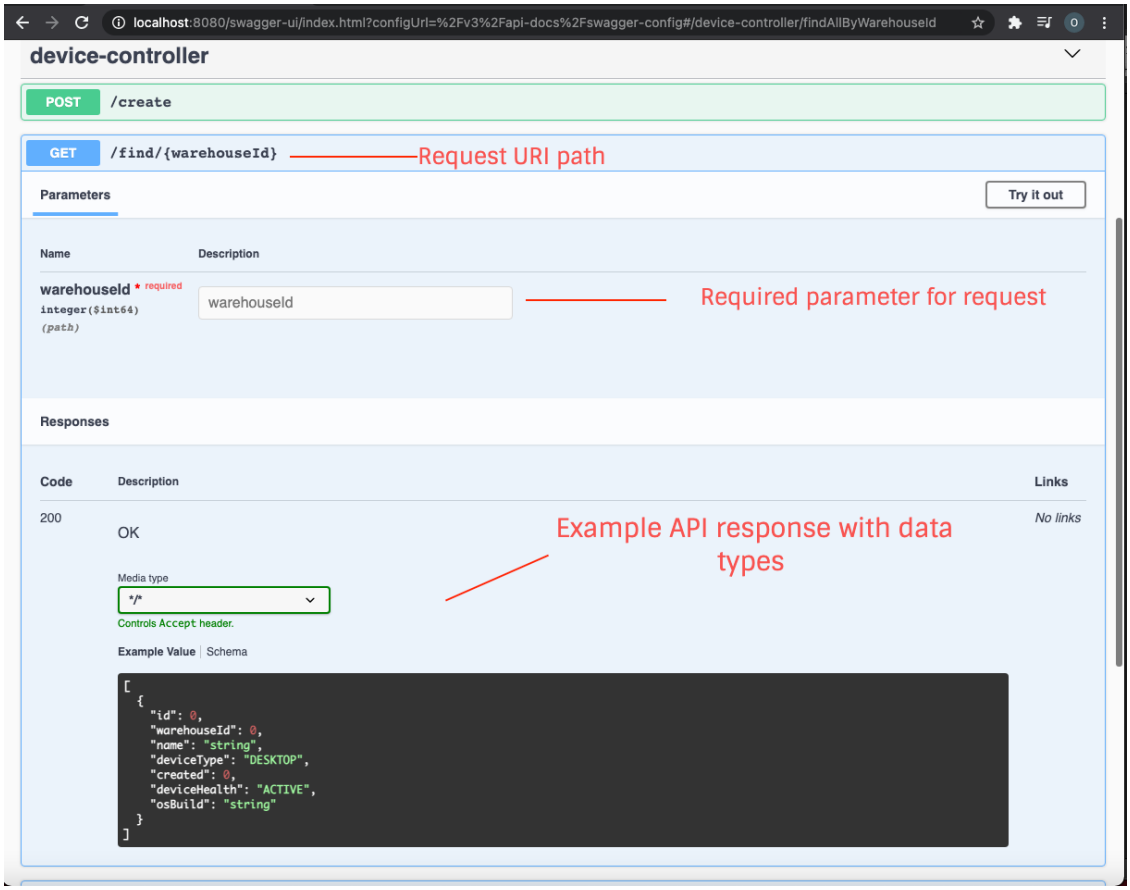


Figure 26: Device Service API definitions - source: (author)

From this definition it's clear what is the URI path, required parameters for the request and response type the API returns. Also it's possible to automatically generate CLI cURL command that calls the API and try out whether it works.

4.3.2 Cluster Resources

First to be able to run our microservices, we need to make sure that in the cloud exist required resources to run it on. Most of them were covered in theoretical part of thesis in chapter 3.1.

Based on our application architecture requirements (chapter 4.3.1) is specified table of resources neccessary to run the solution on.

Table 16: List of deployed resources

Resource	Description	Amount
API Gateway	API Gateway Network http Load Balancing Egress to Load Balancer	1
Apache Kafka Confluent Cloud	Software bus using stream-processing.	1
E2-standard-2	Compute Engine (vCPU: 2, RAM 8GB)	5
Postgresql	Relational database management system emphasizing extensibility and SQL compliance	4
Graphite	Graphite collects, stores, and displays time-series data in real time.	1

In a first part it's needed to create project that encapsulates all the resources that it's being lunched under.

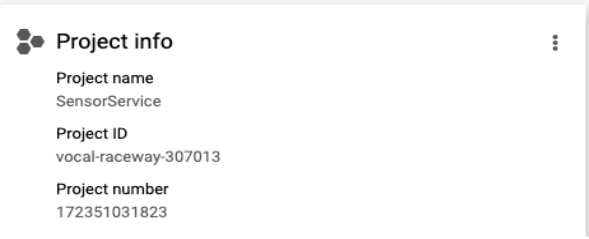


Figure 27: Project creation - source (author)

After project initialization, in GCP we can create the virtual machines that microservices will be deployed to. As described in chapter 3.4.4, to keep cost for the virtual machines low we use instances with less computing power. In case of unexpected workload more instances will be initiated by Kubernetes orchestrator. This is another example of benefit because for example in monolithic applications (chapter 3.4.3). In monolithic application in order to scale up one part of the system you need to deploy it as a whole on stronger instances and therefore it leads to higher cost for usage.

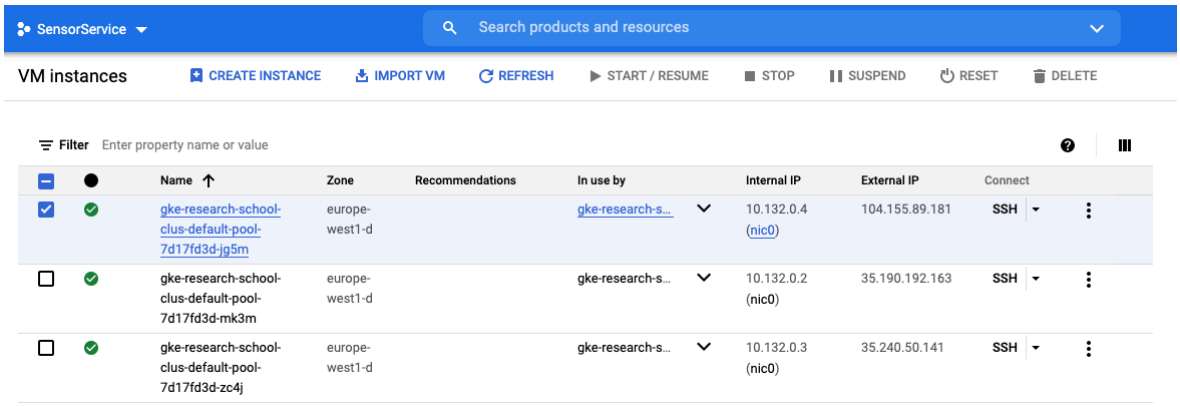


Figure 28: VMs deployment in Cloud - source: (author)

4.3.3 Build and Deployment

This flow is designed based on chapter 3.5.6. To get the application deployed in our cluster on GCP we first need to create the infrastructure, this step builds and executes infrastructure changes towards the cloud provider. It ensures to create resources necessary for running the application like databases, queues and others.

In the meantime application is being built by package building technology Maven which results in a docker image that is pushed to the vendor container registry.

During this proces also different there are set up different stages and run the test layers. (chapter 3.5.7)

Snippet bellow shows the configuration of deployment new docker image to development environment.

```
dev_deploy:
dev: deploy
image: $CI_REGISTRY_IMAGE:latest
only:
- develop
script:
- npm ci
- cd / && config credentials --provider gcp --key $GCP_ACCESS_KEY_ID_DEV --secret
$GCP_SECRET_ACCESS_KEY_DEV && cd -
- SLS_DEBUG=* --dev dev
- for r in $DEPLOY_REGIONS; do SLS_DEBUG=* deploy --verbose --force –dev dev --region $r;
done
```

When an image is pushed in a container registry, then deployment can be executed. It can be a simple "kubectl set image" command (not recommended) up to more sophisticated deployment using a deployment manager. This step ensures that all nodes for the app get the new docker image previously pushed to registry.

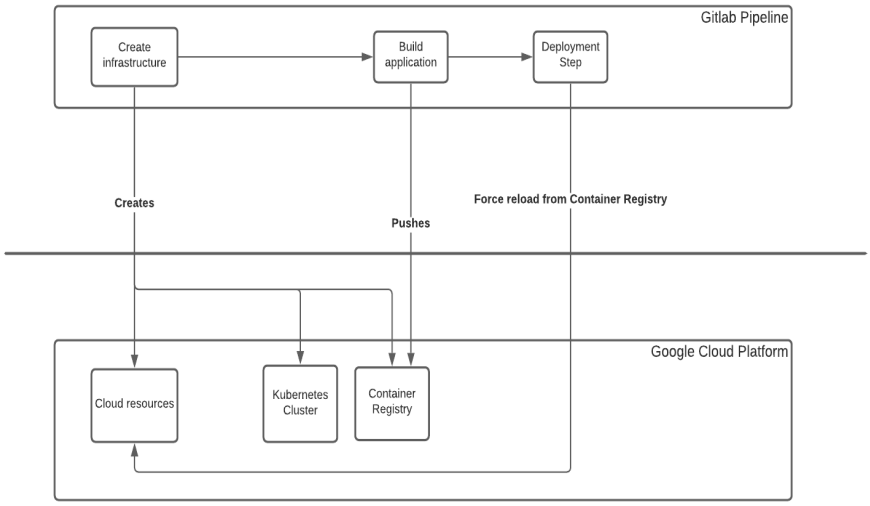


Figure 29: Flow for building new versions – source: (author)

4.3.4 Monitoring and Alerting

In this solution all microservices are pushing application metrics that are collected on application actuator through UDP protocol on port 8125.

The metrics that we're interested in are described in chapter 3.5.8. Project applications are set up in a way to be able to reliably observe what's going on in the distributed system and provide information about the system and server communication.

Error! Reference source not found. contains the most important selected metrics in S ensor Service and description of their meaning.

Table 17: List of selected monitored application metrics in Sensor Service

Metric	Group	Description
Records Lag	Kafka Consumer	Difference between a consumer's current log offset and producer's log offset.

Consumed Bytes Rate	Kafka Consumer	Monitors network throughput. Sudden drop in rate of records consumed might indicate that some consumers are failing.
Fetch Rate	Kafka Consumer	Fetch rate of consumers is a good indicator of overall health. Fetch rate approaching zero could mean issue on consumers.
Aggregate of All HTTP Metrics	HTTP Server	Shows individual endpoints success / error rates and latency of calls over time.
Aggregate of Client Calls	HTTP Client	Describes the amount of calls with particular service through the client and monitors the amount of errors and response latencies.
Healthcheck probes	Application Health / Readiness	Important part of self-healing strategy in services – described further in chapter 3.3.6.
VM properties	System	Reports load on CPU, Heap size and other system metrics.
Transactions	Database	Monitors rate of error and latency in DB transactions.

Emittating of metrics can be displayed through *netcat* command on UDP port 8125 on the VMs. Then we see a stream of metrics that are in realtime eagerly pushed (shown in Figure 30).

Used Command: *nc -ukvl 8125*

Figure 30: Sensor service application metrics output after startup – source: (author)

As we can see this form is not very readable and based on information from chapter 3.5.8.1. To be able make the actionable decisions the output metrics of applications are visualized into the graphs in monitoring platform Grafana.

Following examples in Figure 31 and Figure 32 show visualizing of application kafka consumer metrics in human-readable form.

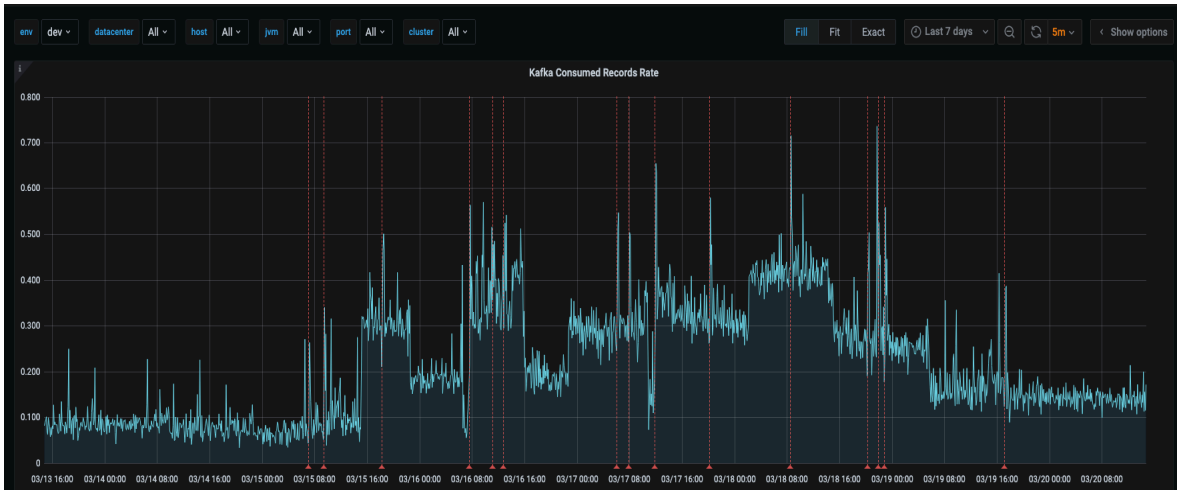


Figure 31: SensorService - Kafka Consumed Records Rate – source: (author)

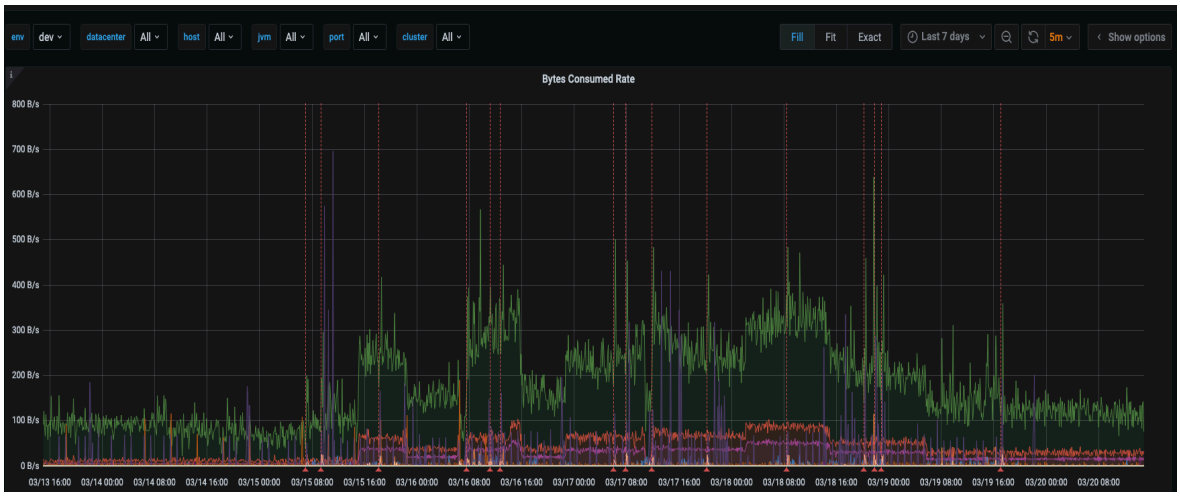


Figure 32: SensorService - Kafka Consumed Bytes Rate – source: (author)

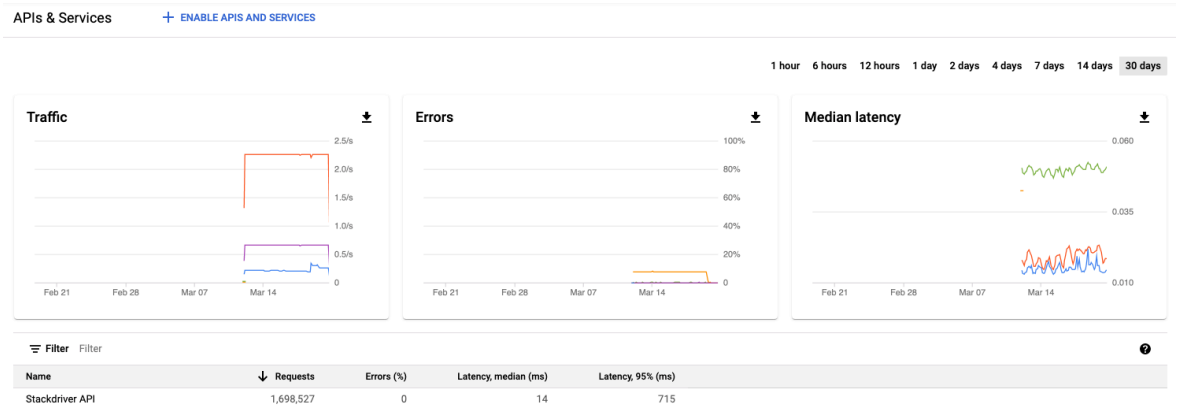


Figure 33: Cluster API metrics – source: (author)

As described in chapter 3.5.8.3 based on long term observation of the application's behaviour is determined critical and warning levels of service degradation. These level are individual for each metric. When the level is exceeded it automatically alerts the responsible person.

4.3.4.1 Distributed Tracing

In the chapter 3.5.9 of the theoretical part were described concepts of request tracing in distributed applications. Ability to pin-point errors and analyse request flow in distributed systems is crucial. In the thesis prototype solution to address this need was a used tool following standard open tracing protocol Zipkin.

To enable this tool all the applications have to include dependency on Zipkin library implementation. Zipkin monitoring platform is then treated just as another containerized service. Once it's deployed to the cloud, applications can register the address and report the spans and traces from the requests.

Examples of this interaction and tracing can be found in Figure 34.

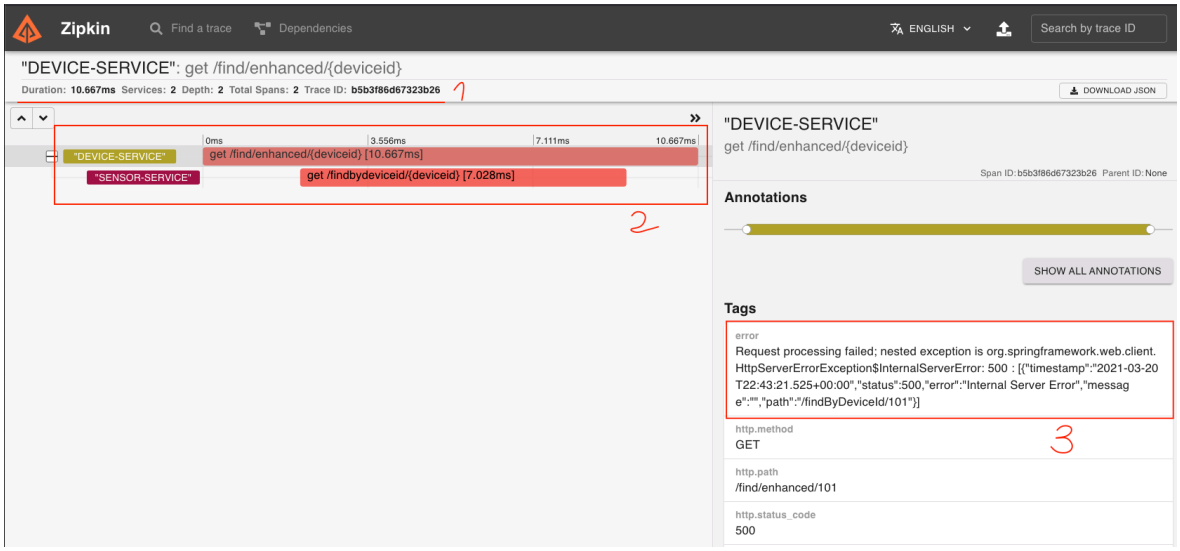


Figure 34: Tracing interaction between device and sensor service - source: (author)

1. Shows details about total duration of request and depth of requests.
2. Visualized interaction with breakdown of individual requests.
3. Pin-pointing where the error occurred during multiple service calls.

4.3.5 Quality Evaluation

In chapter 3.5.7 were explained test practices and selected relevant quality metrics to this solution.

Based on assumptions and goals of the thesis in this chapter of the practical part, test case scenarios were designed to verify the quality of the prototype system deployed in Google Cloud Platform.

For evaluation of the quality based on tests results will be used ISO/IEC 25023 product quality metrics.

4.3.5.1 Performance Tests Scenarios

Performance testing is done to provide a good picture on how the system will behave under real load of requests. As the topic of the thesis is reliability and scalability cloud applications, the goal is to design test cases that will verify stability and availability of the system with large input from the outside traffic.

As a testing tool will be used Apache JMeter that is designed to load test functional behavior and measure performance. Location for the test of calling services was selected Prague, Czech Republic.

During the results were filtered 99th percentile to remove outliers that might cause inaccuracy in observations.

Table 18: Definition of test cases

Test number	Description	HTTP method	Amount of Requests	Amount of Users	Services
1	Adding new devices into platform	POST	50 000	300	2
2	Deleting specific devices by deviceId from the platform	DELETE	50 000	300	2
3	Getting reports of all devices in the warehouse with sensors they use.	GET	50 000	150	3
4	Finding all placed devices in the warehouse.	GET	50 000	300	2
5	Find device by Id.	GET	50 000	300	1

After executing test cases through Apache JMeter the tests results are shown in Table 19. Average response value was on average very low which indicates good responsibility of the solution. Also the error rate on requests was 0.00 % which indicates excellent fault tolerance. During the test it was recorded that service capacity started to be fully utilized. Based on policy in orchestrator were auto-scalled more instances to handle the load. This process was also further described in chapter 3.3.

Also important to notice is that services that have a chain of more remote synchronous calls start to have increasing latency that is caused due to I / O blocking of the thread during synchronous processing of the tasks. This could be solved by modification of design by delegating those actions to asynchronous processing through Kafka message bus (chapter 3.5.3.3) if possible or leveraging reactive microservices that were described in chapter 3.5.4.

Table 19: Cross DC Performance test results

Test number	Average Response Time [ms]		Error rate [%]
	Belgium DC	Netherlands DC	
1	92.13	101.30	0.00
2	72.33	80.29	0.00
3	153.66	158.13	0.00
4	123.20	135.65	0.00
5	36.45	43.93	0.00

4.3.6 ISO/IEC 25023 Product Quality Evaluation

Evaluation of solution quality is done by using product relevant metrics from ISO / EIC 25023 that were further described in chapter 3.5.5. Based on research in the theoretical part of the thesis were chosen metrics focusing on responsiveness, performance, fault tolerance and availability of solution.

Following part shows evaluation of those metrics based on results presented in previous chapter 4.3.5.1.

4.3.6.1 Time Behaviour Measures

Time behaviour measures are used to evaluate the degree to which responsiveness and processing times of the solution performs its functions meet the requirements of stakeholders.

PTb-1-G Mean Response Time

Indicates how long it takes to the system to respond to user or system tasks. To evaluate this metrics data from performance tests in Table 19 were used. Total number of observations from all endpoints were 250 000 requests but for the statistic were sums of averages for each test result.Cross DC Performance test results

Calculation:

$$X = \sum_{i=1\ to\ n} (A\,i)\ /\ n.$$

Ai = Time taken by system to respond.

n = Number of responses measured.

Result :

Ai = 996.79 ms n = 10

X = 99,679 ms

In this case the smaller result value is better by the standard evaluation number that is less or equal to 1 second is still considered a good result. Therefore it can be concluded that the system is very responsive when it comes to speed of response to user or system tasks.

PTb-2-G Response time adequacy

Calculation:

$X = A / B$

A = Mean response time.

B = Target response time specified.

Result :

A = 99.679 ms B = 500 ms

X = 0,199

As in the previous result the smaller result value is better by the standard evaluation number that is less or equal to targeted response time specified can be evaluated as system fulfilling expectation when it comes to response time.

4.3.6.2 Availability Measures

Availability measures are used to evaluate the degree to which a solution is operational and accessible when required for use. One of the goal of the solution examined in this practical part was to achieve high availability which is expressed through the percentage of uptime in a given year.

These metrics are important to measure as the service uptime is often mentioned in SLA.

RAv-1-G System availability

Describes what proportion of the scheduled system operational time is the system available. In this case the system was available for 3 weeks and needless to say that as for the prototype the load was very inconsistent. As system time operation schedule we use the total time since the system was deployed and system operation time actually provided is the time that services have been operational during the scheduled period.

Calculation:

$X = A / B$

A = System operation time actually provided.

B = System operation time specified in the operation schedule.

Result:

A = 504 hours B = 504 hours

X = 1

System uptime was 100 % out of the scheduled time. This signalizes that there were no issues with infrastructure or application itself. This statistic might be little biased due to the unstable load of requests on the service. Most of the traffic was recorded in time of running performance tests. Results such as this are great because it meets all the availability SLA requirements.

4.3.6.3 Fault Tolerance Measures

Fault tolerance measures are used to evaluate the degree to which a solution despite the presence of software or hardware faults is still able to operate.

Those faults can be also in the external systems that solutions rely on.

RFt-1-G Failure avoidance

Describes what proportion of fault patterns has been brought under control in order to avoid critical failures in solution and prevent outage of service.

To explore fault tolerance of the system there was artificaly during tests brought down several machines in the cluster. Expected result based on architecture was that instances would self heal (described in chapter 3.3.6) and be able to load balance the traffic among healthy instances.

Calculation:

$X = A / B$

A = Number of avoided critical and serious failure occurrences (base on test cases).

B = Number of executed test cases of fault pattern, during testing,

Result:

A = 50

B = 50 test cases

X = 1

During the test orchestrator was able based on the architecture of the cluster to cover requests coming inside through load balancing on other instances that were running in parallel, this is one of the features of the orchestrator. In the end the result is that the system was able to handle failures introduced by the test with 100% success rate.

It's necessary to say that there are test cases that would definitely brought down the service. For example outage in cloud provider infrastructure. This would inevitably lead to downtime in whole cluster. Therefore even these unlikely scenarios have to be accounted for and have back up plans in occurence of the event.

RFt-3-G - Mean Down Time

This metric describes how long does the system stay unavailable when a failure occurs. As mentioned in chapter 4.3.6.2, during the observed period of testing the system had 100 % uptime which could make this statistic redundant.

Important test case emulated and very relevant to this statistic was the fault pattern of turning off one of the services. Base on alerting setup in case of fault occurrence in any microservice (more described in chapter 3.5.8.3), the examined part was how quickly alerting platforms would notify operators since the occurrence of error. The refresh period of service status check is an interval of 30 seconds.

The test scenario is trying to uncover how long it takes from the service removal to the alerting system to send notification to the operator.

Calculation:

$$X = \sum_{i=1\ to\ n} (A\,i\ -\ Bi)\ /\ n$$

Ai = Time at which the fault is reported by the system.

Bi = Time at which fault is detected / alert received.

n = number of faults detected.

Result:

Ai: 17:06:33

Bi: 17:06:52

N: 1

X = 19 seconds

The expected result value can vary from 0 to infinite. In this case it was 19 seconds to alert since the error has occurred in the service. As mentioned the default checks that are supposed to catch unavailability in the service run every 30 seconds. This result is considered as very good. It enables the operation team responsible for smooth run of the service to react quickly and prevent damages that might be caused by outage.

5 Results and Discussion

In the theoretical part was described the technology enabling cloud computing and continuing further the layers of technology built on top of it, that enables organizations to outsource their infrastructure from the public cloud providers in order to focus on their core business and be able to deliver faster with lower costs.

Continuing further in the theoretical part were explored tools to manage cloud operations that optimize cost, performance and availability of hosted solutions. Also different models of using cloud infrastructure and their scope were introduced. Then important key success factors for adopting and running tight ship in cloud environments were described. These key metrics were further explored In practical part by market research of cloud providers with largest market share.

Cost of key cloud resources was compared between Google, Amazon and Microsoft cloud platforms and then evaluated according to predefined metrics. From was found out that even though Amazon Web Services has largest market share, it's not necessary cheapest as many people might assume. Surprisingly Microsoft had the best prices for leasing most of the cloud resources but Google also offered some lucrative features.

As mentioned in the theoretical part, working in a cloud environment with managed services might be a big mindset shift for many developers, therefore in practical part was a research survey conducted to explore opinions and relationship to this topic of public clouds. Survey had 44 participants from company Avast Software that are professionally involved in product development. After evaluation all questions were concluded and the overall result indicates that participants are positively accepting the shift that's happening with delegating more responsibilities to cloud providers and being able to focus more on core business.

Next goal was defining best practices for writing cloud native applications. This started in the theoretical part by comparison the benefits and challenges of monolithically applications with microservices. To summarize it, the outcome was that monolithic applications are easier to operate from the beginning but as the solution grows and more people need to work together, it adds additional overhead to development and is not very suitable to run in containers. This also makes horizontal scaling difficult.

Then best practices in writing microservices called "The Twelve Factor App" which is an industry standard guide on how to avoid common pitfalls that come with microservice development. This was followed up by describing service communication using different synchronous protocols or asynchronous messaging. Next was establishing API contracts between services based on the Open API 3 standard that's widely used in industry for API integration.

Another part of lifecycle was CI / CD patterns used for cloud native microservices and their automation with testing stack in order to be able not just flexibly deploy newer versions to the production at any time with low risk of breaking system but also deploy different versions gradually for limited amounts of users traffic called canary releases.

With distributed systems due to the increased complexity it's necessary to be always able to identify what's happening inside the cluster. Practices for cloud monitoring, alerting and distributed tracing of requests coming in and out of the platform were extensively covered and defined application and system metrics important to keep closer eye on.

Practical work also covers the prototype solution of distributed warehouse. This solution consists of 5 microservices connected by message bus Apache Kafka as well as communicating through API integration established based on an exposed Open API contract. These services run inside the Kubernetes cluster deployed on Google Cloud Platform. On the deployed cluster is set up monitoring of selected metrics that applications push through the UDP port to Grafana monitoring platform where the outcomes of applications and message queue are visualized on graphs. The most critical metrics determining the status of the services are closer monitored and have set up alerting conditions. Next chapter then shows the implementation of distributed tracing and ability to trace interaction of requests with all the microservices and based on context sharing report to observability solution Zipkin. It enables to break down the behaviour of request and visualize the order of calls with latencies inside the cluster.

To evaluate durability, scalability and availability of this cluster. Next chapter breaks down the test cases that were run against the service to determine it's stability and performance quality. The test consisted of a total of 250 000 HTTP requests that run at a rate of 300 concurrent requests. Goal of this was to performance test all the endpoints exposed on the services and evaluate the response latencies and error rates of services.
Results of these tests were that the average latency for all endpoints was 99.679 ms and no errors occurred.

Quality of service was further examined by using relevant selected metrics from ISO/IEC 25023 that were described in the theoretical part of the thesis. Evaluation of these metrics have shown that the system is highly available and very responsive. Also it's necessary to add that it would be helpful to simulate a higher load of requests to explore the tipping point of break.

6 Conclusions

To sum everything up it's visible that organizations in the IT industry are more inclined to migrating infrastructure to the public cloud. It offers great infrastructural support, availability for hosting, tooling and scaling flexibility. That results in faster delivery of products, potentially more customers and better costs.

In order to successfully execute cloud computing strategy, it's necessary to have required people expertise on-board. In order to estimate experiences with cloud solutions and the overall positivity around adoption a survey was concluded in Czech software company Avast Software. Based on result of participants, the impression was that most of the people involved in software product development see positively the changes that are happening around cloudification.

Based on computing resources and services costs, offerings of largest cloud services providers analyzed and compared. Microsoft Azure cloud seemed to have the most favourable prices. Needless to say that the standard prices can drastically vary from upfront billing or individual deals that large organizations are able to get. When it comes to software support there is a large offering for all of them but specific cases may prefer different implementations.

When running applications in the cloud it's important to keep in mind organization pay for every resource it uses. This led to shift from large overdimensioned instances hosting monolithical application that often required running standby machines in order to ensure high availability.

In the public cloud the trend is to build compact services also known as microservices packaged in containers that can be operated through orchestrator components such as Kubernetes or EKS to name a few. Orchestrators ensure management of service containers and enable for self-healing of services and elastic horizontal auto-scaling of instances that's evenly load balanced.

In the thesis the goal was to explore what it takes to deploy a cluster of services and the tooling required for operation such solutions. As a main part prototype cluster with 5 microservices connected to Kafka message queue managed by Kubernetes orchestration was bult.

In the work was shown the ease of deployment of new services and setting up monitoring and tracing platforms connecting them together inside the Google Cloud Platform. This example was sucessfully shown how monitoring of applications and managed services in the cloud works as well as distributed tracing of remote calls between the services. The

microservices were integrated through Open API 3.0 specification standard and asynchronous messaging.

To test if the solution meets the requirements set in the work, performance test cases were prepared in Apache JMeter and run against the deployd distributed system. The results were recorded and evaluated based on Time measure, fault tolerance and availability metrics according to ISO/IEC 25023. Results were really good and met the expectation of the software product in this category.

7 References

AWS. 2019. Cloud Storage. *https://aws.amazon.com/.* [Online] Amazon, 1 12, 2019. [Cited: 12 29, 2020.] https://aws.amazon.com/what-is-cloud-storage/.

Boris Scholl, Trent Swanson, Peter Jausovec. 2019. *Cloud Native.* Massachusetts, USA : O'Reilly Media, Inc., 2019. 9781492053828.

Bose, Shreya. 2020. Testing Pyramid : How to jumpstart Test Automation. *browserstack.com.* [Online] 1 21, 2020. [Cited: 11 12, 2020.] https://www.browserstack.com/guide/testing-pyramid-for-test-automation.

Cloud Native Computing Foundation. 2016. The OpenTracing Semantic Specification. *opentracing.io.* [Online] 11 29, 2016. [Cited: 12 2, 2020.] https://github.com/opentracing/specification/blob/master/specification.md.

Daniel Bryant, Abraham Marín-Pérez. 2018. *Continuous Delivery in Java.* Massachusetts, USA : O'Reilly Media, Inc., 2018. 9781491986028.

Education, IBM Cloud. 2019. What is IaaS (Infrastructure-as-a-Service). *IBM Cloud Learn Hub.* [Online] IBM, 7 12, 2019. [Cited: 12 15, 2020.] https://www.ibm.com/cloud/learn/iaas.

Grzybek, Kamil. 2019. Modular Monolith: A Primer. *kamilgrzybek.com.* [Online] 12 3, 2019. [Cited: 12 27, 2020.] http://www.kamilgrzybek.com/design/modular-monolith-primer/.
Ligus, Slawek. 2012. *Effective Monitoring and Alerting.* Massachusetts, USA : O'Reilly Media, Inc., 2012. 9781449333522.

Long, Josh. 2016. Distributed Tracing with Spring Cloud Sleuth and Zipkin. *spring.io.* [Online] Spring Foundation, 2 15, 2016. [Cited: 11 28, 2020.] https://spring.io/blog/2016/02/15/distributed-tracing-with-spring-cloud-sleuth-and-spring-cloud-zipkin.

Marks Richards, Neal Ford. 2020. *Fundamentals of Software Architecture.* Massachusetts, USA : O'Reilly Media, Inc., 2020. 9781492043454.

Mohamandinia, Mona. 2021. Liveness and Readiness Probes in Spring Boot. *baeldung.com.* [Online] Tarnum Java SRL, 2 8, 2021. [Cited: 2 19, 2021.] https://www.baeldung.com/spring-liveness-readiness-probes.

Newman, Sam. 2021. *Building Microservices, 2nd Edition.* Massachusetts, USA : O'Reilly Media, Inc., 2021. 9781492034025.

Nginx. 2019. What Is Round-Robin Load Balancing? *nginx.com.* [Online] Nginx, Inc., 8 12, 2019. [Cited: 11 19, 2020.] https://www.nginx.com/resources/glossary/round-robin-load-balancing/.
Poulton, Nigel. 2019. *Docker Deep Dive.* Birmingham : Packt Publishing, 2019. 9781800565135.

Salesforce. 2020. saas. *salesforce.com.* [Online] Salesforce.com, inc, 2 3, 2020. [Cited: 12 29, 2020.] https://www.salesforce.com/saas/.

Spring Foundation. 2020. Web on Reactive Stack. *docs.spring.io.* [Online] 8 12, 2020. [Cited: 12 5, 2020.] https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html.

Srinivasan Desikan, Gopalaswamy Ramesh. 2007. *Software Testing: Principles and Practices.* Chennai : Pearson India, 2007. 9788177581218.

Swersky, Dave. 2018. The Hows, Whys and Whats of Monitoring Microservices. *thenewstack.io.* [Online] 6 21, 2018. [Cited: 11 26, 2020.] https://thenewstack.io/the-hows-whys-and-whats-of-monitoring-microservices/.

VMWare. 2018. VMWare.com. [Online] VMware, Inc., 8 12, 2018. [Cited: 12 26, 2020.] https://www.vmware.com/topics/glossary/content/hypervisor.

Wiggins, Adam. 2017. Manifesto. *12factor.net.* [Online] 1 1, 2017. [Cited: 12 19, 2020.] https://12factor.net/.

Wigmore, Ivy. 2016. Monolithic architecture. *whatis.techtarget.com.* [Online] 5 1, 2016. [Cited: 12 26, 2020.] https://whatis.techtarget.com/definition/monolithic-architecture.

Quan, Adam. 2019. Distributed Tracing, OpenTracing and Elastic APM . *www.elastic.co.* [Online] 19. 2 2019. [Citace: 28. 12 2020.] https://www.elastic.co/blog/distributed-tracing-opentracing-and-elastic-apm.

Garrison, Justin. 2017. *Cloud Native Infrastructure.* Newton, Massachusetts, USA : O'Reilly Media, Inc., 2017. 9781491984307.

Hohpe, Gregor. 2003. *Enterprise Integration Patterns.* Amsterdam : Addison-Wesley, 2003. 0321200683.

Larsson, Magnus. 2019. *Hands-On Microservices with Spring Boot and Spring Cloud* . místo neznámé : Packt Publishing, 2019. 9781789613476.

Mijic, Dejan. 2018. *Scalable Architecture for the Internet of Things* . místo neznámé : O'Reilly Media, Inc., 2018. 9781492024125.

Sabella, Anthony. 2018. *Orchestrating and Automating Security for the Internet of Things: Delivering Advanced Security Capabilities from Edge to Cloud for IoT.* místo neznámé : Cisco Press, 2018. 9780134756936.

WAEHNER, KAI. 2019. Internet of Things (IoT) and Event Streaming at Scale with Apache Kafka and MQTT. *www.confluent.io.* [Online] 10. 10 2019. [Citace: 3. 1 2021.] https://www.confluent.io/blog/iot-with-kafka-connect-mqtt-and-rest-proxy/.

Laporte, Claude Y. 2018. místo neznámé : Wiley-IEEE Computer Society Press, 2018. 9781118501825.

Miller, Darrel. 2017. OpenAPI Specification. *openapis.org.* [Online] OpenApi Initiative, 26. 7 2017. [Citace: 10. 1 2021.] http://spec.openapis.org/oas/v3.0.0.

KyungWoon Cho, Hyokyung Bahn. 2020. A Cost Estimation Model for Cloud Services and Applying to PC Laboratory Platforms. *www.researchgate.net.* [Online] 7. 1 2020. [Citace: 5. 1 2021.]

https://www.researchgate.net/publication/338464515_A_Cost_Estimation_Model_for_Cloud_Services_and_Applying_to_PC_Laboratory_Platforms.

Burger, Loraine. 2021. Why Cloud Computing Is Essential to Your Organization . *https://www.simplilearn.com/.* [Online] 2. 2 2021. [Citace: 15. 2 2021.] https://www.simplilearn.com/why-cloud-computing-is-essential-to-organization-article.

Humble, Jez. 2018. *Accelerate.* místo neznámé : IT Revolution Press, 2018. 9781942788331.

Stalcup, Katy. 2021. AWS vs Azure vs Google Cloud Market Share 2021: What the Latest Data Shows. *parkmycloud.com.* [Online] 10. 2 2021. [Citace: 16. 2 2021.] https://www.parkmycloud.com/blog/aws-vs-azure-vs-google-cloud-market-share/.
—. 2021. AWS vs Azure vs Google Cloud Market Share 2021: What the Latest Data Shows . *https://www.parkmycloud.com/.* [Online] 21. 2 2021. [Citace: 26. 2 2021.] https://www.parkmycloud.com/blog/aws-vs-azure-vs-google-cloud-market-share/#:~:text=As%20of%20February%202021%2C%20Canalys,%25%2C%20Alibaba%20Cloud%20close%20behind..

ISO/IEC. 2016. *System and software Quality Requirements and Evaluation (SQuaRE) - Measurement of system and software product.* Geneva : INTERNATIONAL STANDARD ISO / IEC, 2016. ISO / IEC 25023.

SVMK Inc. 2020. The 3 types of survey research and when to use them . *www.surveymonkey.com.* [Online] 20. 12 2020. [Citace: 15. 1 2021.] https://www.surveymonkey.com/mp/3-types-survey-research/.

Apache Software Foundation. 2011. Apache JMeter. *https://jmeter.apache.org/.* [Online] 2. 11 2011. [Citace: 15. 1 2021.] https://jmeter.apache.org/.

OpenZipkin. 2017. https://zipkin.io/. *https://zipkin.io/.* [Online] 12. 1 2017. [Citace: 1. 15 2021.] https://zipkin.io/.

Lee, James. 2018. *Hands-On Big Data Modeling.* místo neznámé : Packt Publishing, 2018. 9781788620901.