Large and Cloud-based Software-Systems
Lecture Notes

author: **Prof. Dr. René Wörzberger**

contact: rene.woerzberger@th-koeln.de

saved on 14 March 2020 22:16:00

Technology
Arts Sciences
TH Köln

Table of Contents

I. Introduction

1 Introduction

Software systems differ in their complexity. For such systems, complexity comes along in many shapes: the size of the code base, the number of different stakeholders with different interest or concerns about the system, quality attributes that have to be traded against each other and so forth.

1.1 Large Scale Systems

An extremely simply software system is the widely known system "Hello World". The only function of "Hello World" is to print out "Hello World" in a console and then terminate. Normally, the code base consists of less than 10 lines of code, the only stakeholder is the developer with the only "interest" to make some initial steps in a new programming language, and quality attributes, e.g. dependability or maintainability are of diminishing importance.

On the other side of the spectrum, there are complex systems[1] like those facing towards customers of Google, Amazon, or Facebook. Such systems have a huge code base, many stakeholders have different interest or concerns about the respective system and a plethora of quality requirements need to be considered.

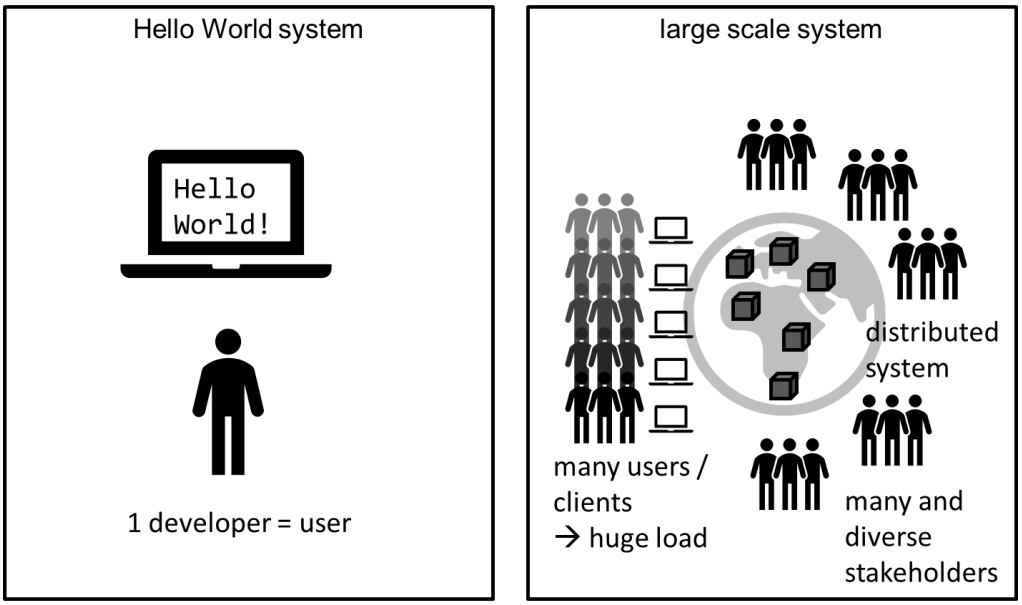


Figure 1: Hello World system vs. large-scale system

These lecture notes for the course "Large and Cloud-based Software Systems" focus on software systems on the complex end of the spectrum, especially with hindsight on operations in the so-called cloud. Despite the fact that our normal altitude is above introductory bachelor courses, we sometimes delve into technical details in order to make insights tangible.

We particularly put ourselves into the role of an IT architect. Caring about a system's architecture is particularly important for complex systems. If you take lines of code as a measure for complexity, Figure 2 shows that the importance of architecture work really increases with the complexity of a system.

[1] For the time being, we do not distinguish between a non-distributed monolithic system and a distributed "system-of-systems".

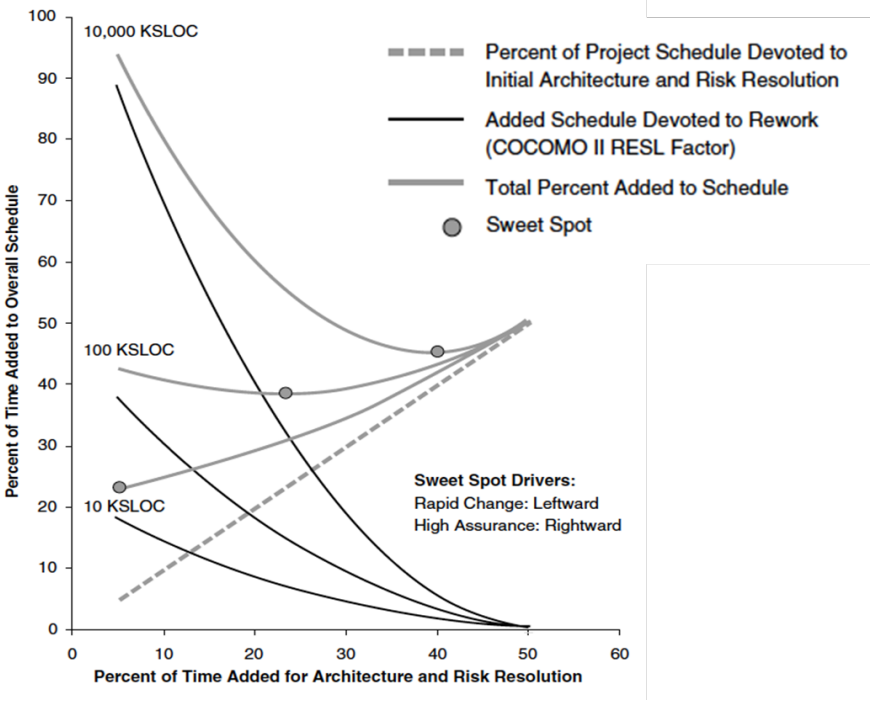


Figure 2: Optimal fraction of architecture work for systems of different sizes [1, p. 180]

1.2 Disclaimer

These lecture notes are in a continuous draft status and have not undergone an official review or lectorate process. Please forgive me mistakes and current inaccuracies, e.g., sloppy uses of references. Every reader is invited to notify the author about typos, obvious mistakes or divergent opinions.

Speaking of "opinions": Since software engineering is still a young profession, there is an ongoing debate about what insights are reliable. Bossavit argues in [2, pp. 3-10] that many alleged facts about software engineering are mere opinions, beliefs, tacit agreements, folklore or exaggeration of scientific evidence that have been ripped out of context and modalities. In contrast, Martin points out in [3, p. 33] that opposing the currently accepted state of the science sometimes lead to progress despite lack of real evidence.

I think both have a point. Software engineering is still in the age of alchemy, not chemistry. Software is an intangible thing and not subject to physical laws. Instead, cognitive and sociological effects play an important role in the creation and maintenance of software. Disputes in the developer community are often driven by the urge of the opponents to upvalue or at least save their learning investments. However, trading the effects of new tools, frameworks, methods, techniques or paradigms without hard evidence is the best we could do right now; yet more rigour[2] surely would improve the state of science and art.

Also due to the lack of scientific rigour, software engineering is susceptible to trends, hypes, cargo cults, fashions which might be recurrent yet with different names[3]. The belief in (positive) effects of some new paradigm, method, or technology often climaxes on a "peak on inflated expectations" early on, fueled by success stories from developer conferences, blogs and the like, which are less real evidence but contribute to a survivorship bias prone view. Later, such beliefs fall in a "trough of

[2] Admittedly, I haven't found the time to thoroughly validate the sources I reference. For the time being, I assume them to tell the truth.

[3] CASE tools in the 1980ies, model driven development around 2000 and low/no code tools in the 2020s.

disillusionment" before reaching a long-lasting "plateau of productivity" as the Gartner Model for Hype Cycles [4] puts it. The one for application architecture and development as of 2019 can be found in [5].

Moreover, some trends oscillate between extremes, like

- whether computation and storage should reside on the client side or the server/cloud side,
- whether a software system is better be broken down into multiple microservices or better be maintained in a structured monolith, or
- whether development projects are better be carried out in a completely agile mode or in a sequence of stages,

just to name a few. Some of the methods and technologies are also subject of these lecture notes. I give my best to focus on facts and give a neutral position on unproven positive or negative effects.

2 Learning Outcome

The learning outcome of the whole course is the following:

Students are capable of

- designing architectures for complex and mission critical enterprise software systems,
- of implementing these systems and
- operate them in the Cloud

by

- knowing and trading conflicting interests and concerns of **stakeholders**,
- knowing **quality attributes** and their trade-offs,
- specifying **architecturally significant requirements** in **quality attribute scenarios,**
- analysing **design decisions** with respect to their effects on quality attributes and stakeholder interests and concerns,
- presenting and documenting architectures by means of suitable **views**, **notations** and **tools**,
- applying **methods** (like RESTful API design) and **tools** in order to **implement design decisions**,
- using **cloud resources** like virtual machines, containers and storages in order to operate a system in the cloud,

in order to

- be able to produce **long-term usable software systems** in subsequent lectures and projects and
- to be able to act as an **IT architect**, e.g. in an IT department of a larger enterprise.

3 Case Study / Story

In order to get a grip on sometimes rather theoretical methods and concepts of this course, we exemplify these methods and concepts in accompanying case study, which starts with the following story:

> You are a chief IT architect in the middle-sized e-commerce company *ReWo-IT*. Up till now, this company focuses on building and operating web shops. Over the years, ReWo-IT carved out a product called *ReWo-Shop*, which is a customizable web shop system. Shop owners (sellers) can either buy a license of ReWo-Shop and operate a customized instance, e.g. with an own product catalogue, corporate identity, and domain name on their own infrastructure or in the cloud or use the software-as-a-service offering for ReWo-Shop.
>
> One day the CEO approaches you with the following idea: "Many of our clients sell identical products. You know, we have this product rating component in ReWo-Shop. A visitor of some shop may rate a product after purchasing it. For the time being, a single rating is visible just within a single shop. Therefore, ratings for the very same product are scattered around shop instances. Buyers, i.e. our customer's customers, do not give much faith in an overall rating of some product if it is backed by just a few actual ratings.
>
> So, let's build a more centralized rating system *ReWo-Rate*, where all ratings are stored or at least those that are triggered by transaction of ReWo-Shop-instances."
>
> Inevitably, a stream of thoughts rushes through your mind: "How do we know which products are the same across different shops? How do we integrate the rating with ReWo-Shop or other web shop systems? We should not harm the experience of buyers; how can we keep response times low and availability high? Is consistency of major importance or can we live with slightly varying ratings? What privacy constraints do we have to adhere to? Should we build for the cloud? Can we make use of AI techniques? NoSQL database, finally?"
>
> You suddenly realize that you already entered the solution space before eliciting any requirements, thought about stakeholders, and so on.

4 Software Systems

4.1 The Nature of Software

Some say programming is some kind of art, some say computer science is a subfield of natural sciences or mathematics. Mostly, software is considered as an engineering product.

Non-software engineering products, like cars, planes, drilling machines, houses, bridges etc are different from software. The following properties show inasmuch software is different from other engineering products:

- Physical laws and limits: Software is immaterial and therefore not affected by physical laws (cannot crush due to its poor design etc.)

- Model / Plan: Software is comparably hard to envision / model

- Modifications: Software is comparably easy to change

- Production / logistics costs: Software is comparably easy and cheap to reproduce and store

4.2 Software Characteristics

Software is ubiquitous. However, a PlayStation game is surely not from the same type like a SAP application. There are plenty of properties that shape the "character" of a software system.

A PlayStation game is an instance of entertainment software as opposed to business software.

There are many ways to characterize different types of software like Facebook, BitTorrent, MS Excel, Fortnine, Bitcoin, Netflix, Linux, DSL router software, weather simulation, or SAP applications. Table 1 provides one way to categorize these kinds of software. In that table, types of software systems are characterised

- whether they are **embedded** or not into a hardware system (e.g. car),
- if and how they are **distributed,**
- if they focus on complex and vast **computation** tasks or **data,**
- if they can be directly used as **applications** by users or serve as bare **middleware** for other applications,
- which **user groups** they target, and
- if or if not their complexity mainly stems from a broad **functional scope** with complex and change-prone business logic, diverse and sometimes incoherent functions, entity types, user roles etc., which target different concerns.

Example	Embedding	Distribution	Computation vs. data	Application / middleware	User group	Functional scope
Facebook	non-embedded	client-server	data	application	private	medium
WhatsApp	non-embedded	client-server + peer-to-peer	data	application	private	narrow
Bittorrent	non-embedded	peer-to-peer	data	application	private	narrow

Example	Embedding	Distribution	Computation vs. data	Application / middleware	User group	Functional scope
MS Excel	non-embedded	local	data	application	business	narrow
Fortnine	non-embedded	local / client-server	-	application	private	narrow
Bitcoin	non-embedded	peer-to-peer	computation	application	private	narrow
Netflix	non-embedded	client-server	data	application	private	medium
Linux	all	local	-	middleware	all	-
DSL router software	embedded	local	data	application	all	narrow
weather simulation	non-embedded	local	computation	application	business	narrow
SAP application	non-embedded	client-server	data	application	business	often broad

Table 1: Categories for software

4.3 Basic Concepts

In the following, we try to use even those terms in a consistent manner that are often used interchangeably.

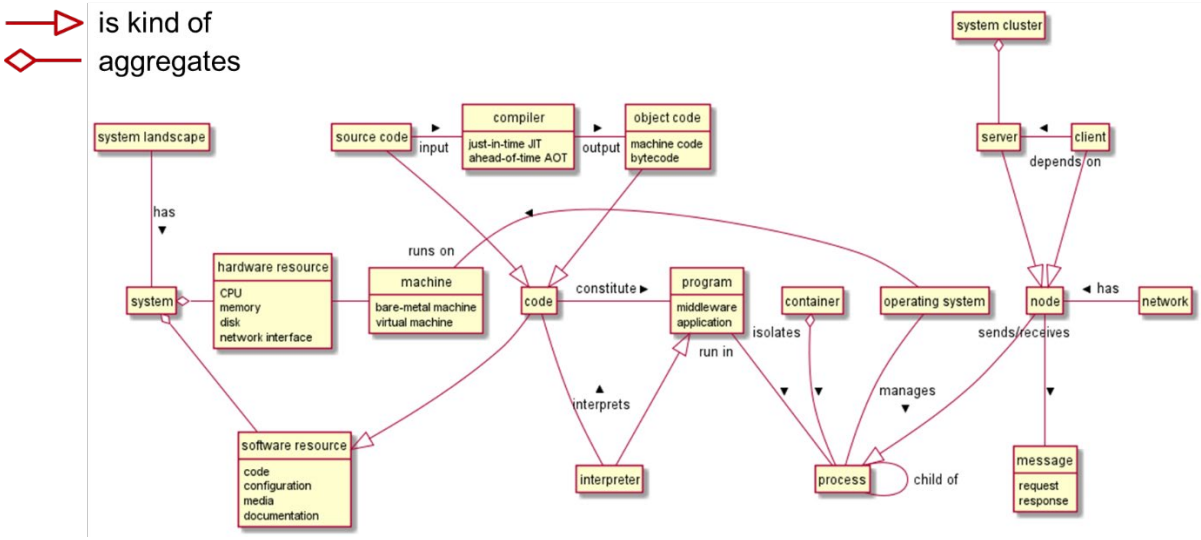


Figure 3: Map of basic concepts

4.3.1 Software

Software comprises **software resources** (a.k.a. **artifacts**). These software resources contain **code,** which constitute **programs**, as well as configuration files, documentation etc.

Human developers edit code in its human-readable **source code** form normally using a **high-level programming language**. A program called **compiler** translates source code into **object code,** which is either machine (architecture) independent **bytecode** or machine (architecture) dependent

machine code. An **ahead-of-time (AOT) compiler** translates all code of a program before its execution. A **just-in-time (JIT) compiler** translates chunks of the code on demand during program execution.

Object code files usually contain references to other object code files from, e.g., external **libraries**. In case of machine code, a program called **linker** cares for resolving these references at build-time and produces an **(executable) object file**[4], also known as program **image**. In case of bytecode, a program called (class) **loader** cares for resolving these references at runtime.

Only machine dependent machine code can run directly on a machine. Source code or—more efficiently—bytecode can only be executed indirectly by a special program called **interpreter** or sometimes **abstract machine**.

4.3.2 Phases

Editing source code, compiling and linking are (iterative) phases of software development. We also speak of **editing time**, **compile time** or **link time** to denote these phases. With **build time**, we subsume all these phases that might vary depending on the programming language[5] but all take place before actually starting a program. With **runtime**, we mean all phases that come after this event.

4.3.3 Processes

At **runtime**, programs are executed in **processes**, which are data structures that manage the state of a running program and which are directly provided by an **operating system kernel**.

Each process allocates its own and isolated **virtual memory** address space and has a **current activity**, represented by the **program counter** which points to the next program instruction and CPU's register values and some other data. The virtual memory address space contains (1) a **text section** with the program (in compiled machine code), (2) a **data section** for global variables, (3) a **heap section** for dynamic data structures, and (4) a **stack section** for stack frames that store local variables and parameters for function calls [6, pp. 104-105].

Processes of a single machine are organized in a **process tree**: A program running in a process can create child processes, which are copies[6] of the parent process' data described above. If such child processes just have their own current activity and stack section but share everything else with their parent process, these “lightweight processes” are more specifically called **threads**[7].

Operating systems kernels run their own **kernel processes** with privileged rights. Such **privileges** include accessing files on disk or devices, creation of processes etc. In contrast, programs such as word or spreadsheet applications or command line tools, which are either shipped as part of the **operating system distribution** or installed afterwards from third-partly websites, run in non-privileged **user space processes.** These user space processes might do a **context switch** to the privileged **kernel mode** just in the course of a **system call** in order to, e.g., obtain access to a file on a disk.

[4] This file then usually conforms to a file format like “Executable and Linking Format (ELF)” [75] for Linux or “Portable Executable (PE)” [76] for Windows.

[5] For example, JavaScript need not be compiled but can be interpreted in its source code form.

[6] Actually, the virtual memory is not really copied on child process creation. Only if a child process writes to virtual memory the respective portion of that memory is copied and then modified. This is also called "**copy on write**".

[7] One could also say that every process has at least one “main” thread since you can consider the stack section and current activity the defining data structures of a thread and each process contains at least one of these.

4.3.4 Machines

Programs run on machines. A **machine** is a collection of resources like CPU, memory, network interfaces, or storage which processes can access by means of the operating system. A machine might be **bare-metal hardware** containing real **hardware resources** or virtualized with **virtual resources**, i.e. being a **virtual machine** or—even more lightweight—a **container**. A machine that runs virtual machines or containers is called also called **host** with the virtual machines or containers being its **guests**.

4.3.5 Networks

In a network, processes can communicate with other processes on other machines, even other bare-metal machines. In the following, we generally call a process that is addressable in a network and communicates with other nodes just by network protocols a **node** in that network. Please note that in the following we are just interested in application layer network protocols [7, p. 32], whose communicating participants, i.e., nodes, are always processes and not some low-level hardware network. Sometimes, we abstract on fine-grained distribution such that “node” could also refer to a set of processes in a network that jointly fulfil a certain role, like a “database node”.

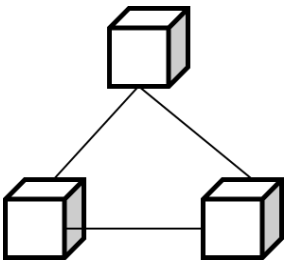


Figure 4: Nodes in a network

4.3.6 Applications and Middleware

A coherent set of functions, which is useful for certain stakeholders, e.g. end users, and which are implemented by programs running in dedicated processes is called **application**. If the programs just support other programs, e.g. by implementing certain technical and crosscutting functions like message queues, these programs are called **middleware**.

If an application is accessible through the World Wide Web and exposes a **web user interface (UI)** for **browsers** or a **web application programming interface (API)** for other external systems in the web, e.g. native smartphone apps, it is called a **web application**.

4.3.7 Systems

An application is served by a system. A **system** is the entirety of its **resources**, which might be software resources or hardware resources like machines/nodes, storage and network infrastructure that realize the respective application. If a system contains software resources, we called it more precisely a **software system** in order to distinguish it from other types of systems. The set of a single company's systems is called **system landscape**.

If a system contains two or more nodes that communicate with each other over a network, we have a **distributed system**. In a network, a node might be a **sender** which sends data in **messages**[8] to a **receiver**.

[8] Messages are considered to be "meaningful and complete" data record for the receiver of arbitrary size, i.e. a program on the receiver is able to process it. Nonetheless, messages could be broken up in fixed-sized **packages** by the sender on the transport layer and reassembled on the receiver side.

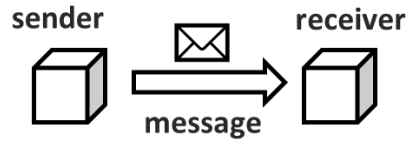


Figure 5: Sender, message, and receiver

4.3.8 Client-server Systems

A **client-server system** is a distributed system, where a **client** node depends on a **server** node in order to work as specified but not the other way around. Web applications are realized by client-server systems, where clients might be browsers[9], smartphone apps or other external systems. If the server-side in a client-server system has two or more nodes, we call these nodes a **system cluster**. Please note, that two nodes in such a server-side cluster might again form a client-server (sub-)system. Moreover, nodes of the same type might be scaled out thus forming a, e.g., **application server cluster** or **database server cluster**.

In a **client-server system** the **client** usually[10] sends a message called **request** to a node[11] called **server**, which replies with a **response**.

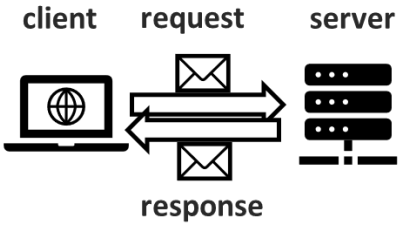


Figure 6: Client, request / response, and server

4.3.9 Messaging

Sometimes, sender and receiver communicate via **message queue**, a.k.a. **message broker** a.k.a. **streaming processor** (cf. Section 14.4). Since messages often contain data related to an **event** that took place in the sender and this event message might be received by many receivers, sender and receiver are usually called **publisher** and **subscriber**, respectively, in such scenarios.

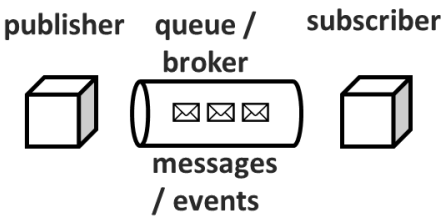


Figure 7: Publisher, queue, and subscriber

[9] In this case the "program" running on the client is the browser itself that interprets and renders web responses from the server or some JavaScript that has been transferred in a response from the server.

[10] The request-response communication pattern is not restricted to client-server systems nor the other way around. Yet, both often come together. Less common client-server communication patterns are discussed in Section 14.3.

[11] Again, it would be more precise to talk about communicating **client processes** and **server processes** and the nodes these processes run on **client node** and **server node**, respectively. However, it is common to just talk about client and server if the distinction between process or enclosing node is either clear from or irrelevant in the respective context.

II. Drivers

5 Stakeholders

Stakeholders in a software development process are people that have "stakes" in the process' execution or outcome. They have **interests** or, conversely, **concerns**. Stakeholder that share the same concerns due to a common responsibility or job profile are considered to be in the same **stakeholder group**.

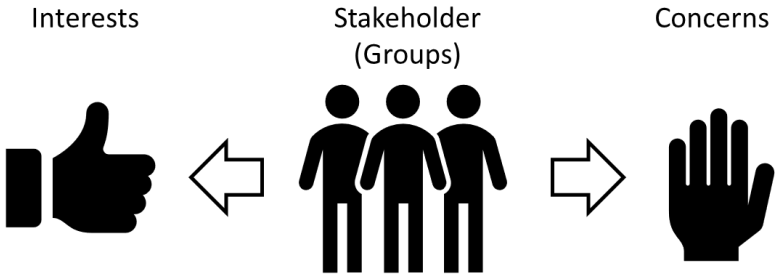


Figure 8: Stakeholders, interests and concerns

5.1 Stakeholder Groups

Developers and users are those stakeholder groups in a software product and development processes which most people have in mind. However, they are by far not the only one.

> Example: The stakeholder group "first level support" is concerned with whether a certain user interaction can easily be retraced in a support case.

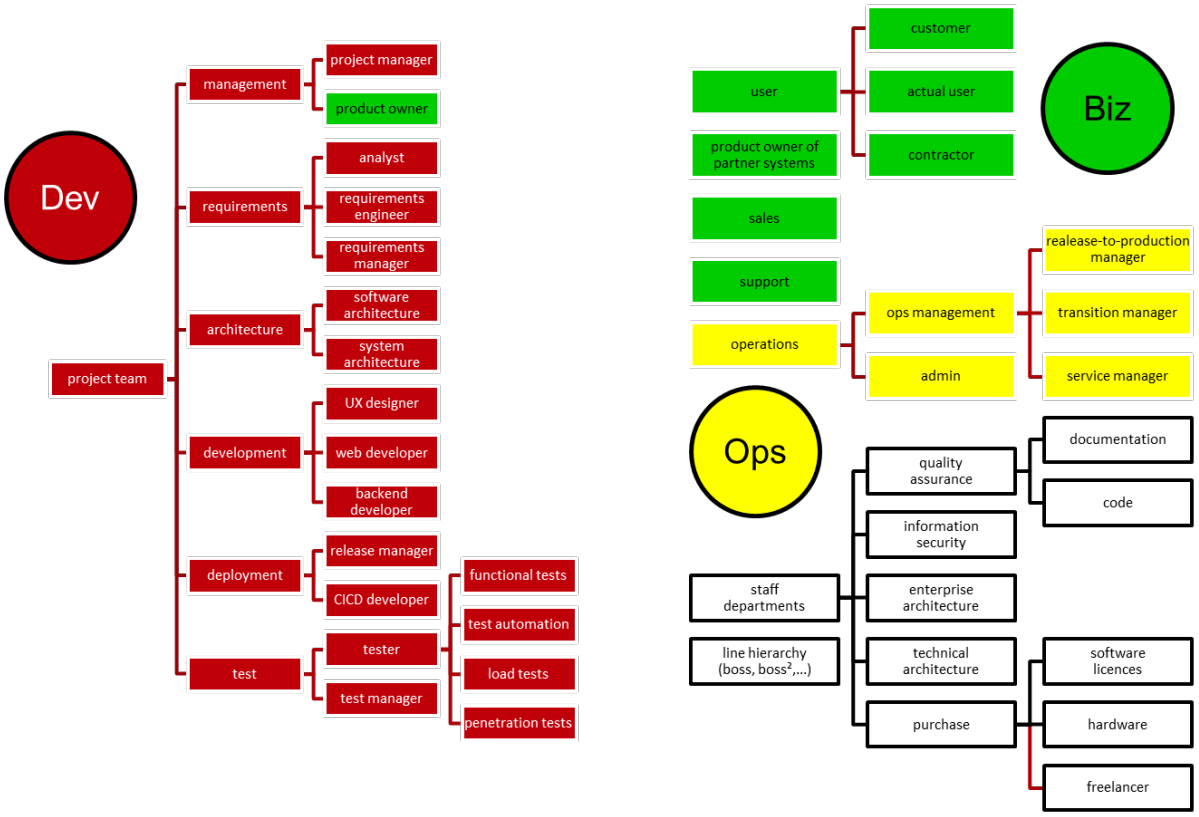


Figure 9: A taxonomy for roles in a development project

Figure 9 depicts stakeholder groups that you may encounter if you participate in a development project in a bigger enterprise. The leaves in the taxonomy are **roles** which are not necessarily been carried out by exactly one person.

In agile development, teams act mainly autonomously. In order to fill out the roles, each team member typically assumes multiple roles of Figure 9, particularly those form Dev and Ops. This makes them **generalists** who are committed to fulfil the promise "You build it, you run it", but in turn can focus just on one development project. In classic development, there are **specialists**, who do not do anything but activities from a certain role, but for different development projects in parallel.

Anyway, roles are typically bound to certain interests and concerns and are therefore treated as equivalent with stakeholder groups in the following.

Stakeholder group	**Responsibilities, interests and concerns**
project manager	The project manager is responsible for keeping a project in time, budget and quality, which are his major concerns.
product owner	The product owner represents the system under development towards the business. He or she is also responsible for maintaining the products requirements. His or her main interests and concerns are about the final product and its success. He or she is often less concerned about time and budget.
application owner	A product owner is responsible for managing processes that revolve around a single application, which is normally implemented in a single system. These are, e.g., managing maintenance processes (like library updates). If a certain project mainly concerns a single application, he or she also has the role of the project manager. The interest and concerns of an application owner are in a way all of those of Chapter 6 pertaining his or her application.
systems analyst	The analyst is mostly responsible for eliciting requirements and upholding the quality of requirements in terms of unambiguity, completeness, precision etc. Systems analysts are primarily interested in having all stakeholders finally agree on the contents in requirements even if their particular view on a business domain is not entirely reflected in these.
requirements engineer	Same as systems analyst.
requirements manager	Requirements in a development project are a moving target. They are in- or out-scoped, reprioritized, elaborated, rephrased etc. A requirements manager keeps track on the current state of requirements, e.g. with the help of tools like JIRA or HP QC, but does not participate in actual requirements gathering nor elicitation (as opposed to a requirements engineer). Requirements managers are mainly interested that all stakeholders agree on a current set of requirements, while they are a little less interested in the actual contents of requirements.
software architect	A software architect focuses on the architecture of software which is mainly reflected in the structure of the source code and other software artifacts. He or she decides about the use of architectural styles and patterns like a 3-layered-architecture and governs the implementations of the design decisions. Software architects are concerned about non-functional requirements that directly target their system, maintainability in particular. As opposed to system architects they are less concerned about the overall, e.g., performance of systems that contribute to a certain use case.
systems architect	A system architect mostly focuses on designing system clusters with resources of different kinds (cf. Chapter IV), and with integrating different systems, particularly negotiating interfaces between systems He or she is concerned with whether these systems properly work together but a little less concerned about the inner architecture of software artifacts (as opposed to a software architect).

Stakeholder group	Responsibilities, interests and concerns
UX designer	A user experience (UX) designer is mostly concerned about the usability and look & feel of an application, e.g. if labels of input fields a better placed left to or above the input field. He or she usually do not code but use mocking tools to design the user interface of a system.
frontend developer	A frontend developer focuses the implementation of the user interface of a system. In case of a web application, the frontend developer has advanced skills in technologies like HTML, CSS, Javascript and associated tools and frameworks.
backend developer	A backend developer focuses on the server side in case of a web application. He or she normally has advanced skills in Java and SQL.
release manager	The release manager is part of the development team. He or she is responsible for the proper delivery of a software release, e.g. that exactly those changes are included in a release that have been ordered. Particularly, a release manager keeps an eye on the release state if last-minute changes have to be included in a release.
CICD developer	A CICD developer is responsible for the creation and maintenance of a continuous integration and delivery (CICD) pipeline. The pipelines are supposed to automate the repetitive tasks of software build, integration, test and delivery.
tester for functional tests	A tester for functional tests defines test plans based on requirements and manually executes them while recording if a software release behaves as expected or not. Nowadays, re-execution of test cases for functions introduced on prior releases, i.e. regression tests, is often automated. Therefore, tester for functional tests focus on progression tests, i.e. test for functions that were newly introduced in the current release.
test automation developer	Some specialists particularly focus on automation of test, mostly regression tests. This can be done with unit test tools like Junit but also on the UI with tools with Selenium.
load test tester	Load test tester focus on the non-functional quality attribute "performance". The use tools like Apache JMeter in order to generate workload for a system and measuring, e.g. response times.
penetration test tester	Penetration tests (or short: pentest) simulate attacks on a system thus testing its security.
test manager	Like a requirements manager, a test manager keeps track of existing and relevant test cases and their execution w.r.t. to a specific release. He or she is responsible for reporting testing progress to the project manager and endorses or advices against deploying a release to production.
customers	A customer might be a real person or a legal entity that pays for a certain software or services provided by a certain software system.
actual users	An actual user is a real person that uses a certain application. In a business setting actual users might be employees of a "customer", which is then a legal entity (e.g., some company).
other product owners	In a development project of software system x, product owners of those systems are important stakeholder, which interact with x. Sometimes these systems have to implement minor changes in order to interact with x, so these product owners have at least an interest of getting a share of system x' budget.
sales department	The sales department is responsible for selling a software product or the services that are accessible through a software system. Bigger customers are assigned to dedicated members of the sales department, also known as key account managers. These stakeholders are mainly concerned about customer satisfaction though not necessarily user satisfaction.

Stakeholder group	Responsibilities, interests and concerns
support department	The support department solves problems that arise on the customer's side. These stakeholders are interested in logging and tracing functions that allow for retracing user interactions with the system.
release-to-production manager	A release to production manager manages the transition of a release into production. He or she normally is employed in the company that is responsible for the operations of system and keeps track of availability of key resources (admins) during a release. His or her primary concerns are whether a release can smoothly be deployed into production.
transition manager	Same as release-to-production manager, yet he or she is usually employed at the company that owns the system.
administrator	An admin monitors the health of systems, installs new release and sets up environments.
documentation quality assurance	In bigger enterprises there is a dedicated department that reviews system and project documentation. They mainly focus on the apparent completeness of documents and a little less on their inner quality as judging about the correctness of a document's contents for each system would require too much insight into each system.
code quality assurance specialists	Another department in large companies cares for the quality of the source. These stakeholder use tools like Sonarqube in order to measure, record, report and archive performance of development teams in this regard.
information security department	Members of an information security department usually have a high standing in bigger enterprises which cannot afford reputation losses due to security incidents. These stakeholder demand extensive penetration test of systems before each release.
technical architecture group	Technical architects are concerned about which third-party resources (tools, libraries, and frameworks) are used in the overall system landscape. They make prescriptions or at least endorsements here and keep track of the support of software versions, e.g. Java 8.
enterprise architecture	Enterprise architects deal with the architecture of the overall system landscape. They are concerned about which business capabilities are supposed to be supported by which (new) system but are not interested in technical details about each system's implementation.
purchase	The purchase department is responsible for acquiring resources, i.e. software licences, hardware or temporary human resources (freelancers). Their main concern is about the cost of these resources.
boss(es)	Almost every stakeholder has a boss. They often come into play if severe conflicts arise between stakeholder. Their major concern is whether their subordinates perform well and -- if something goes wrong -- blame is put to somebody from another department.

Table 2: Stakeholder groups

Particularly in agile requirements engineering, one often talks about personas instead of stakeholder groups. **Personas** are concrete yet fictive and stereotypical representatives of some stakeholder group. Such personas are mostly used to represent subgroups of the users stakeholder group, i.e. represent different kinds of users.

A persona relevant for ReWo-Rate could look like this: "Bob is a 30 year old single and tech enthusiast. He is an early adopter of new tech gadgets and eager in reading and writing his opinions about his latest purchases."

5.2 Stakeholder Analysis

Typically, actual stakeholders are classified according to the classification of Figure 10. Each classification is highly project specific, i.e. you cannot generally peg a certain stakeholder group as "apathetics" in every project.

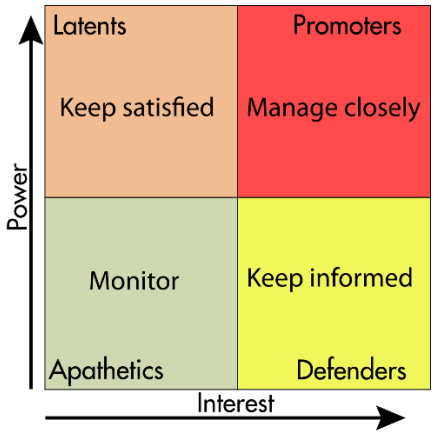


Figure 10: Stakeholder classification (By Zirguezi - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=30927010)

Table 3 depicts a (highly subjective) stakeholder analysis for ReWo-Rate.

Stakeholder group	**Classification**
project manager	Promoter. However, he or she is in charge manage other stakeholders
product owner	Promoter. Same as project manager
systems analyst	Defender. Often high identification with project but rather low power.
requirements engineer	Defender. Often high identification with project but rather low power.
requirements manager	Defender. Often high identification with project but rather low power.
software architect	Defender. Often high identification with project but rather low power.
systems architect	Defender. Often high identification with project but rather low power.
UX designer	Defender. Often high identification with project but rather low power.
frontend developer	Defender. Often high identification with project but rather low power.
backend developer	Defender. Often high identification with project but rather low power.
release manager	Apathetics. Often manages multiple projects, yet with low power.
CICD developer	Apathetics. Often manages multiple projects, yet with low power.
functional tests	Apathetics. Often manages multiple projects, yet with low power.
test automation	Apathetics. Often manages multiple projects, yet with low power.
load tests	Apathetics. Often manages multiple projects, yet with low power.
penetration tests	Apathetics. Often manages multiple projects, yet with low power.
test manager	Apathetics. Often manages multiple projects, yet with low power.

Stakeholder group	Classification
customers	Latents. Their combined power is high but interest has to be nurtured.
actual users	Latents. Their combined power is high but interest has to be nurtured.
other product owners	Latents. Often somewhat reluctant to adapt their systems (interest) but with the power to block a rollout if their system is vital to the system under development.
sales	Promoters. Usually with high interest regarding the marketability of some software system (interest) but also putting high pressure on the product owner to suit customer's needs (power).
support	Apathetics. Often manages multiple projects, yet with low power.
release-to-production manager	Apathetics. Often manages multiple projects, yet with low power.
transition manager	Apathetics. Often manages multiple projects, yet with low power.
admin	Apathetics. Often manages multiple projects, yet with low power.
documentation quality assurance	Apathetics. Often manages multiple projects, yet with low power.
code quality assurance	Apathetics. Often manages multiple projects, yet with low power.
information security	Latents. Often manage multiple projects, yet with the power to block a rollout are even shutdown a system in case of a security incident.
technical architecture	Apathetics. Often detached from actual development projects. Provide mere guidelines.
enterprise architecture	Apathetics/Latents. Often detached from actual development projects. Might obstruct or stop a project if enterprise architecture goals are not met.
purchase	Latents. Often detached from actual development projects. Might defer project start due to difficulties in negotiations with subcontractors.
boss(es)	Latents/Promoters. Bosses of stakeholders have to power to overrule the stakeholders. Their interests range from outright opposition to high interest. It is vital for every project to have a promoter with high interest and power in the organisation.

Table 3: Stakeholder analysis for ReWo-Rate

Note that the classification of stakeholders is dependent on the culture and politics of the organization and the particular development project as well. Generally, the interest in a particular development project or running system is low if stakeholders are specialists that deal with a lot of projects or systems. Moreover, the power of stakeholders again depends on the company's culture and their standing within the company as well as their position in the organization's hierarchy.

Assessments in a stakeholder analysis often collide with the self-esteem of certain stakeholders. Thus, stakeholder analyses are better kept confidential and accessible to just a few members of a development project team.

6 Quality Attributes

In the preceding chapter we have learnt that different stakeholders have different interest and concerns. Together with system analysts it is the job of an architect to transform the rather vague interest and concerns into proper requirements[12] and identify constraints.

6.1 Requirements

Junior architects tend to narrow architecturally relevant requirements to something like "Keep the system evolvable by enforcing a 3-layered architecture". Though modifiability is key to a long-lasting system there are many other types of requirements that are important for architects. This chapter is meant to broaden the view for these types of requirements.

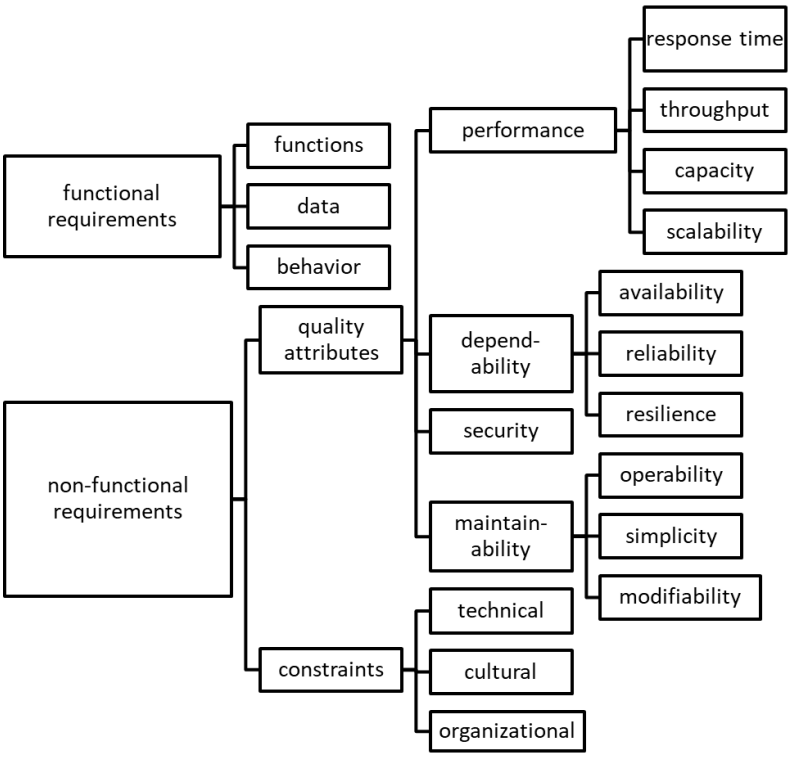


Figure 11: A partial taxonomy for requirements

Requirements come along in very different flavours. Figure 11 depicts a taxonomy[13] for requirements. **Functional requirements (FR)** are predominant for many stakeholders yet are just the tip of the requirement iceberg. Each requirement, which is no functional requirement, is called a **non-functional requirement (NFR).**

Functional requirements have an influence on a system's architecture just inasmuch that their respective implementation need to be assigned to components in a reasonable, i.e. maintainable manner. How to do this is taught, e.g. in the "Software-Praktikum (SWP)" in "Technische Informatik (Bachelor)", where functional requirements are explicitly mapped on components, each of which is

[12] According to [51] a proper requirement is complete, unambiguous, consistent, modifiable, verifiable, and traceable.

[13] This taxonomy is partial as it just includes those quality attributes that are relevant for this lecture. For example, dealing with "usability" would be a lecture on its own and is therefore excluded. Moreover, there is no de facto and de jure standard for classifying requirements. There are many taxonomies yet each with just subtle differences, e.g., from [49] or [50].

manifested in a JAR-file. The mapping of functional requirements to components is a virtue on its own and gave rise for methods like **Domain-driven design** [8] for larger systems. Apart this mapping, functional requirements are not supposed to have a greater impact on a system's architecture.

Non-functional requirements often contain so-called **architecturally significant requirements (ASR)**, which are those requirements that have an impact on the system's architecture. An architect rather focusses on these requirements while system analysts deal with sometime tedious details of how certain functions should look like or how a system should behave in certain situations.

> Counter example: "The rate should be 200px * 100px in size" is not an architecturally significant requirement.

6.2 Quality Attributes

Furthermore, there are non-functional requirements which can be negotiated and thus leave room for multiple design decisions. These are called **quality attributes** or sometimes **ilities** due to the fact that many of them end with -ility.

Table 4 exemplifies each quality requirement category.

Category	Requirement
function	Sellers should trigger an email, which is sent to buyers and asks for a rating after a purchase has been completed.
data	Quantitative ratings, e.g. 1 to 5 stars, and a textual description are supposed to be persisted along with their associated transaction.
behaviour	A rating can only be created by a seller. A buyer can edit, save, delete or commit it. After it has been deleted or committed, it cannot be further edited.
response time	When a user opens a pending rating, the response time should be below 1000ms in the 98th percentile.
throughput	ReWo-Rate must cope with a workload of 300 rating commits per second.
capacity	ReWo-Rate must store a maximum of about 100.000.000.000 ratings.
scalability	ReWo-Rate resources are supposed to be utilized at an average of 60% while the workload peaks vary between 50 ratings/s (3 am on a summer vacation day) a 500 ratings/s (6 p.m. Christmas holidays).
availability	ReWo-IT must be available for 99.99%, i.e. may be down for at most 52 minutes per year.
resilience	The core use cases, i.e. store and retrieve ratings must still work even if all partner systems are unavailable.
security	URLs to ratings should be practically impossible to guess.
maintainability	The technical debt should be below 100 days for repair according a given SonarQube ruleset.
portability	The web UI should be usable in a web view within a hypothetical future ReWo-Rate Android or iOS app.

Table 4: Quality requirements for ReWo-Rate

6.3 Constraints

On the other side, **constraints** are not negotiable. There imply design decisions made externally. Constraints can be technical in nature, organizational, or imposed by legislation.

Constraint	Category	Explanation and rationale
Google Cloud	technical	Due to a partnership between ReWo-IT and Google, it’s a matter of fact that ReWo-Rate will be operated in GCP. Therefore, GCP's strengths and limits must be considered right from the start.
CICD	organizational	Development processes at ReWo-IT follow a continuous delivery approach. So, the build of ReWo-Rate has to be automated and development processes to be set-up appropriately.
Enterprise Service Bus	technical	The ReWo-IT's reference architecture requires that ReWo-Rates uses an existing enterprise service bus for message-based communication with other systems of ReWo-IT.
DSGVO	legislative	Users of ReWo-Rate have to consent to ReWo-IT's common cookie policy. Therefore, a respective pop up should be displayed to unknown users.
SSL only	technical	ReWo-Rate is exposed to the internet via https only. This has to be considered by external systems that are integrated with ReWo-Rate.
storage encryption	technical	Every data that is stored by ReWo-Rate must be encrypted. So, if an attacker gets hold of raw data files, these must be useless for him or her.
rating scale	Cultural	The rating scale must adapt to the customs of a buyer's country. A rating scale of five stars is the default.

Table 5: Exemplary constraints for ReWo-Rate

7 Performance

Performance is the quality attribute that is most noticeable by users. Requirements regarding workload can be immense for large scale systems as Figure 12 demonstrates.

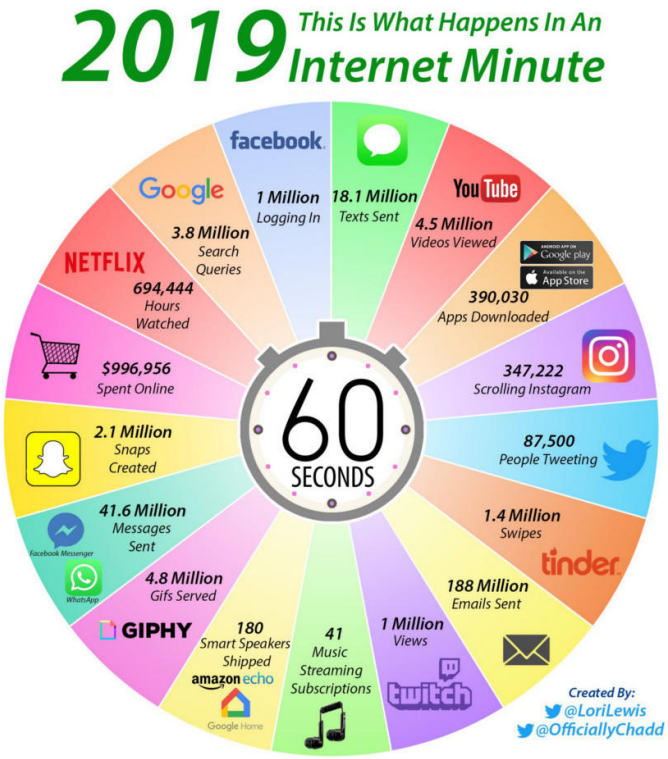


Figure 12: Workload for large scale systems in 2019

7.1 Performance Limiting Resources

A system's performance is of course finite. It is limited by the following hardware resources in particular:

Hardware resource	Description
processing units	Processing units like the **central processing unit (CPU)** or the **graphics processing unit (GPU)** can just execute a limited number of machine instructions in a given time. This number is limited by the frequency of the CPU **clock signal**, which typically ranges between 1.7 GHz and 4.0 GHz, the CPU **cache** sizes and speeds and the number of CPU **cores**, which can execute machine instructions in parallel.
random access memory (RAM)	RAM of different types differ in throughput, i.e. the amount of data that can be transferred read and written in a given time span, and latency.
storage	Like RAM, storage has a finite read-write throughput and non-zero latency. Often, storage performance is expressed by means of **input/output operations per second (IOPS)**. Single HDDs range from 20 IOPS to 200 IOPS and SSDs to 10,000,000 according to [9] depending on the size of the blocks read or written in each operation, the interface type (fibre channel, serial attached SCSI (SAS), serial ATA (SATA)), the rotational speed in case of HDDs (from 5400 rpm to 15000 rpms), the distribution of blocks and thus the need for rotation and repositioning read/write heads in case of HDDs.
network	Networks have a limited throughput and non-zero latency, too. Maximum throughput is typically limited by the link speed of network interfaces, which nowadays ranges from kilobytes (poor mobile connection) to gigabytes per second (fast LAN). Latency is inherent to networks as described in Section 7.3.

7.2 Workload Categories

Every request contributes to the **workload** a system has to cope with. The more requests per second hit the server the higher the workload. However, incoming requests that demand for a response are just one type of workload.

We can roughly distinguish between two types of workload:

- There is workload which is **computationally intensive** (a.k.a. **CPU-bound**) like inputs for scientific applications, simulations etc. These workloads are subject of **high-performance computing**, which is done with a very large machine that is upscaled to thousands of CPUs.
- In these lecture notes, we refer to **data-intensive** (a.k.a. **I/O-bound**) workloads, i.e. workloads that are limited by the speed of disk and network I/O.

Table 6 provides an overview about common data-intensive workload categories for systems that are operated in cloud environments.

Workload category	Description	Limiting resources
transactional requests	Typical workload for web applications, i.e. many requests form many clients. Request mostly require modest computational resource due to execution of certain business logic. Moreover, they cause a few comparably simple database reads and writes, e.g. straight read or write of a record in a table. Database literature often refers to this kind of workload as **online transactional processing (OLTP)**.	database storage performance, network latency
analytical requests	Business analytics applications tend to have a rather small user space containing just a few domain experts that are interested in analytical data (like the number of purchases of a certain product in a certain time span). Therefore, the number of requests per second is comparably small. However, database reads are comparably expensive: Analytical requests tend to be much more complex than transactional requests, e.g. by joining many tables in the database per request. The database community calls this kind of workload **online analytical processing (OLAP)**	database storage latency, database server CPU, network latency
batch jobs	Transactional or analytical requests and responses usually occur in an interactive application. Response time for each request-response-cycle is important for the user's satisfaction. Conversely, batch jobs often run in the background without any user interaction. Classic examples for batch jobs are full filesystem scans of an anti-virus software, the mass-generation of thumbnails for a big photo gallery or video reencoding. Inputs as well as outputs of batch jobs are often files in a (distributed) file system or dedicated "staging tables" in a persistent database as opposed to requests and responses which are transient by nature.	CPU, storage throughput
streams	Streams are considered unbounded, i.e. without a definite end. Therefore, receivers start processing and producing output without receiving the whole stream beforehand. Receivers might also start receiving somewhere in the middle of a stream without catching up what was on the stream before. This makes streams different from request and batch jobs. **Media streaming** providers like Netflix or YouTube stream video and audio data. Though the end of a stream is known beforehand in this case, streams are still so big that they treated as describe above. After all, **message brokers** for system integration (cf. Section 14.4) provide a kind of streaming, where streams mostly contain event data that needs to be forwarded, e.g. records with timestamps, key, old value and new value. **Stream processing systems** [10, pp. 439-487] go beyond pure messages forwarding. Again, network throughput is an issue for all stream workloads even for pure messaging. At least for media streaming, storage throughput at the sender side is vital. If processing comes into play, CPUs can become a bottleneck.	network throughput, storage throughput, CPU

Workload category	Description	Limiting resources
voice/video over IP	Voice or video over IP is similar to media streaming as it requires a certain network throughput. However, network latency is more important here since big latencies hamper synchronous communication between users.	network throughput, network latency

Table 6: Workload categories

Since database storage often turns out to be a limiting resource and RAM got cheaper in the last two decades, in-memory databases like Redis, SAP Hana or Oracle In-Memory DB gained popularity. Batch jobs and event streaming gain from horizontal scaling using algorithms like Map-Reduce or by partitioning and distributing data across multiple nodes.

In the following, we mainly focus on transactional request workloads, i.e. this kind of workload is talked about unless specified otherwise.

7.3 Response Time

7.3.1 Network Metrics

For a client-server system, it is not easy to tell which two events delimit the relevant time span that should be further specified in requirements. Server, client (browser) and even the user work in parallel. Moreover, servers usually also have several parallel processes that receive and process requests. Figure 13 shows common time measures with respect to a single http request.

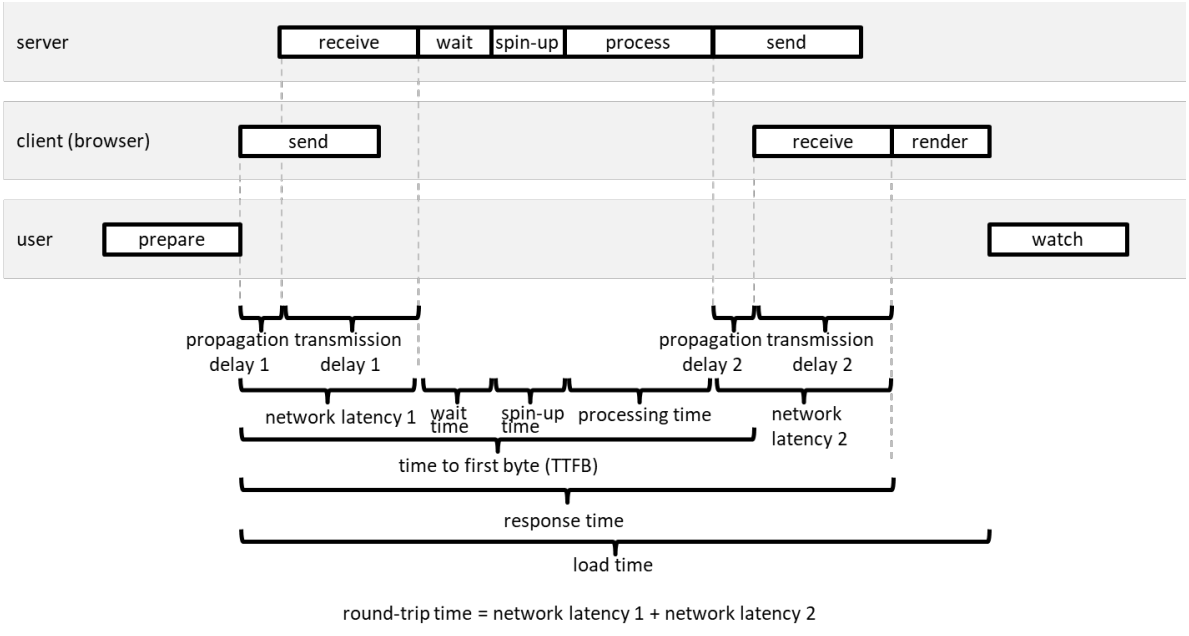


Figure 13: Common time measures

In network theory, **network latency** is the time a message needs to travel from a sender to a receiver. In a round-trip, the server replies to a request with a response and the **round-trip time (RTT)** the sum of both network latencies.

System architects are mostly concerned with **response times** with basically is round-trip time + **wait time** on server + **spinning-up** time[14] for creating an appropriate process + actual **processing time**, i.e., the time the process needs to process the request and produce a response.

Response times and particularly network latency can be kept small by locating copies of frequent responses in private local or shared remote caches that are (a) close to the expected locations of the clients and (b) do just a lookup of stored responses based on the request instead of actual computing a response (cf. web caching in Section 15.12). Processing time can be kept small by, e.g. parallelizing even the processing of a single request where possible.

Figure 13 simplifies things particularly inasmuch a typical web page consists of multiple responses. Moreover, rendering in a browser is a complicated process that yields meaningful results that a user can interact with before every response has been received and completely rendered. If you are interested in browser networking: An excellent resource is https://hpbn.co/ by [11]

7.3.2 Network Latency

A normal web application, where the client (browser) and the server are located anywhere on the globe, can suffer a lot from network latency. The ultimate threshold is the speed of light, which limits the **propagation delay** of a single bit of data.

> Example: A roundtrip between Cologne and New York, which are 6,000 km apart from each other, cannot be faster than $\frac{2*6{,}000\ km}{300{,}000\frac{km}{s}} = 40\ ms$.

Of course, in reality there are additional factors that slow down the **roundtrip time** (**RTT**) [11, p. 7]:

- The propagation delay is typically lower: The speed of light in fiber is just about 200,000 km/s.
- The cables on the network route are in sum longer than the bee-line.
- If the request or response are bigger, then **transmission delay** comes into play, i.e. the time it takes to put a message on a link. This delay depends on the data rate of the respective link, e.g. 100 Mbit/s. Transmission delays are part of the length of the sending and receiving bars in Figure 13.
- Packets of a message might also wait in queues before being routed in a router thus contributing to **queuing delays** or being **processed** in a router.

> Even if the messages are small and transmission delay is neglectable, the RTT between Cologne and New York is about 150 ms in reality (instead of 40 ms).

7.3.3 Impact of Response Time

Particularly in web applications response time is of major importance. With higher response times **conversion rate**, i.e. the probability that a visitor becomes a paying customer, drops significantly.

Response time	User perception
0 – 100 ms	Instant
100 – 300 ms	Small perceptible delay

[14] If corresponding processes have been created before and are just pooled, then this is neglectable. However, with the advent of serverless computing / Function-as-a-Service this time span again gains relevance.

Response time	User perception
300 – 1000 ms	Machine is working
1,000+ ms	Likely mental context switch
10,000+ ms	Task is abandoned

Table 7: User perceptions of response times (after [11])

7.3.4 Response Time Variance

Real-time systems guarantee for certain fixed response times. However, in usual web applications we do not have such guarantees. Instead, response time vary for diverse reasons [10, p. 14]:

- **context switches** between processes consume processing time,
- **page faults** require reads from disk, which is why machines are usually equipped with much RAM in order to prevent page faults or even swapping at all,
- a **garbage collector** run could stop all Java threads in a JVM for up to 60s,
- **retransmission** of TCP packets, which have been dropped on their route, or
- **mechanical faults** like vibrations that slow down disk IO drastically.

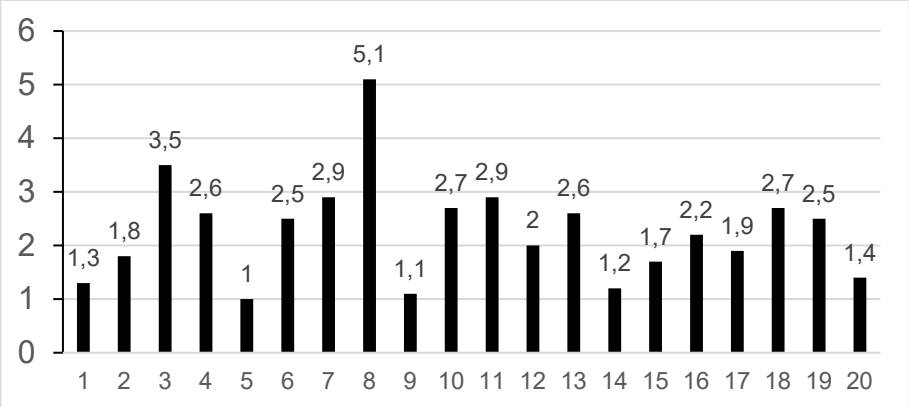
7.3.5 Measuring Response Time

Due to the variation of response times, it would be pointless to require an upper bound for response time. Instead, response time requirements are often made with respect to the 0,99-percentile (p99) of response times. If the requirement says "The p99 of response times must be at most 1,000 ms", then 99% of all response times must be below 1,000 ms and just 1% may be above 1,000 ms.

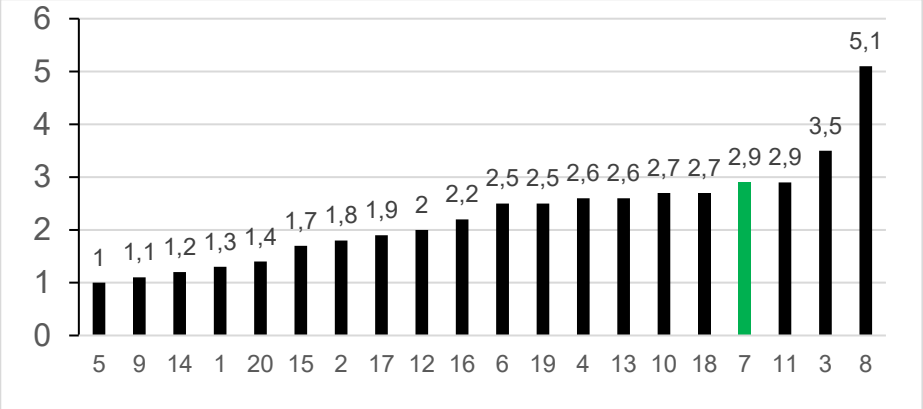
Given a sequence $(x_1, x_2, \dots, x_n)$ of (measured) response times, which is sorted by size, i.e. $x_1 \leq x_2 \leq \cdots \leq x_n$ the p-percentile x_p is defined by

$$x_p = \begin{cases} \frac{1}{2}\left(x_{n \cdot p} + x_{n \cdot p+1}\right), & if\ (n \cdot p) \in \mathbb{Z} \\ x_{\lfloor n \cdot p+1 \rfloor} & otherwise \end{cases}$$

Example: Consider the sample response times from below.



We want to compute the 0.83-percentile (p83). First, we have to order the response times by size:



For n, we have 20 values in total. With $n \cdot p = 20 \cdot 0.83 = 16.6 \notin \mathbb{Z}$ we have

$$x_{0.83} = x_{\lfloor 20 \cdot 0.83 + 1 \rfloor} = x_{\lfloor 17.6 \rfloor} = x_{17} = 2.9$$

If we want to compute the 0.95-percentile ($p95$), we have $n \cdot p = 20 \cdot 0.95 = 19 \in \mathbb{Z}$ and therefore

$$x_{0.95} = \frac{1}{2}(x_{20 \cdot 0.95} + x_{20 \cdot 0.95+1}) = \frac{1}{2}(x_{19} + x_{20}) = \frac{1}{2}(3.5 + 5.1) = 4.3$$

7.4 Throughput

7.4.1 System Throughput

Response time might not be mixed up with **throughput** of a system. Consider a system that processes 100,000,000 requests simultaneously with a processing time of 10s for each request. This yields a high throughput of 10,000,000 requests per second. However, the response time of at least 10s would be considered slow from a single client's perspective.

Throughput might not just refer to incoming requests per second. Instead, throughput[15] generally refers to "jobs" per second that have to be processed. This also includes queries to a database or reads and writes on a disk.

In queuing theory, a simple model that defines throughput is the following:

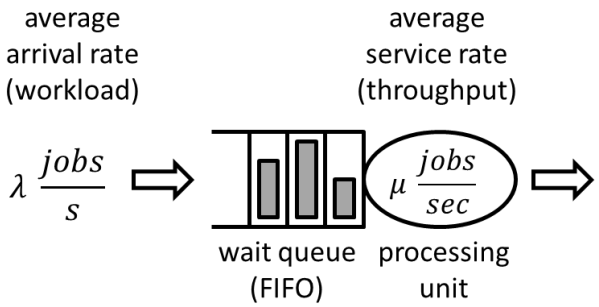


Figure 14: Simple queuing model

Jobs enter a system with a certain **arrival rate** and can be processed with a certain **service rate** (a.k.a. departure rate), which is the reciprocal value of **service time**. The **job size** is the required

[15] In an undersaturated system, one has to distinguish between **actual throughput** and the **maximum throughput** -- also known as the **bandwidth** -- that would saturate the system. However, we use "throughput" homonymously for both actual and maximum throughput.

work of the **processing unit**, e.g. a CPU core, to process the job, e.g. generate a response for a request. The job size can be expressed in, e.g., CPU cycles or seconds of overall processing time spent for a single job. Since jobs might arrive randomly and job sizes S might vary, we are just interested in averages of both rates. Here, **workload** is another word for the **average arrival rate** λ. **Throughput** is synonymous to the **average service rate** μ if no jobs are dropped from the wait queue.

Despite throughput and response time are different concepts, they correlate: Lowering network latency is still out of reach but using a faster processing unit increases throughput, thus decreases the processing time for each job but also time a job waits in the queue. Parallelizing processing units with dedicated work queue multiplies the throughput but also decreases the average number of jobs in a queue and therefore the waiting time in a queue for each job.

7.4.2 Network Throughput

The term throughput is also applicable to single hardware resources like network devices or storage controllers. In network devices, processing basically boils down to manipulating and forwarding packets. Here, throughput is synonym to **transmission rate** (e.g., 1000 MB/s) and reciprocal to transmission delay.

Please note that in case of TCP, network latency (a.k.a. roundtrip time (RTT)) limits network throughput depending on the size of the **TCP receive window** (**RWin**), which is the amount of data a sender might send without getting an acknowledgement from the receiver.

$$max.network\ throughput \leq \frac{RWin}{RTT}$$

With an RWin of 64kB and an RTT of 150ms the maximal network throughput is 427kB/s!

7.5 Utilization

The **utilization**[16] of a system is $\rho := \frac{\lambda}{\mu}$. If $\rho = 1$, a system is **saturated**. If $\rho > 1$ a system is **oversaturated,** which inevitably leads to a congestion of the wait queue and therefore to dropped jobs[17]. If $\rho < 1$, the system is **undersaturated**.

Of course, this model is a little simplistic as it omits the actual statistical distribution of job arrivals and job sizes as well as the possibility to have several parallel processing units each with a dedicated wait queue. [12] delves into these kinds of scenarios and their mathematical modelling.

7.6 Capacity

Capacity[18] refers to the amount of entities that are (transiently or persistently) stored in limited space. These include

- records in a database

[16] Sometimes the term "**load**" is used synonymously with "utilization". In order to avoid confusion with the external "workload" of a system, we avoid this term.

[17] Before that, response times typically exceed the timeout set by clients, e.g. 60 seconds in case of a browser, so that even if the job is finally processed, the response is of no use.

[18] Sometimes capacity is also used synonymously with maximum throughput. However, we differentiate between both concepts in this document.

- bytes on a disk
- server-side sessions in an application server's memory
- concurrent sessions in a load balancer
- opened TCP connections in a web server

7.7 Scalability

A way of dealing with performance requirements in the face of growing workload could be to allocated much resources to the system right from the start. Having, e.g., Googles whole infrastructure for your own system surely would put you on the safe side for a very long time regarding growing workload. Though this would be an effective strategy, it would not be efficient because unused resources would cause costs anyway.

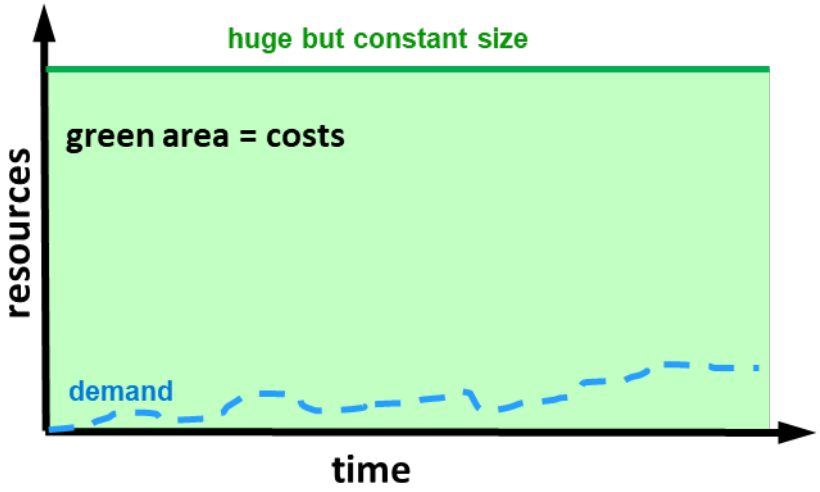


Figure 15: Huge but constant system size

If (a) system's resources can easily be allocated and deallocated and (b) performance grows proportionally with allocated resources, a system is said to be **scalable**.

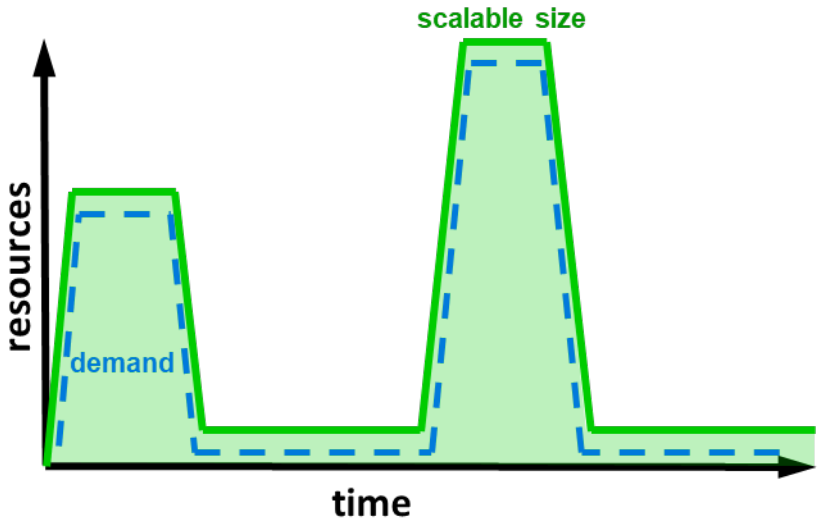


Figure 16: Scalable system

Since allocated resources drive costs, scalability particularly contributes to the **efficiency**[19] of a system.

7.7.1 Transactional Requests Workload Profiles

In an ideal world, the workload for a system would be constant. A system could be sized once and the utilization of CPU and RAM resources would be constant for the entire system's lifecycle. However, in reality, workload is far from being evenly distributed at least for transactional request workload. In e-commerce applications like ReWo-Rate, the workload often follows certain patterns:

- **long-term growth**: An e-commerce start-up that rapidly gains new customers / users has to cope with a growing workload in the long run. Moreover, the workload caused by a single web page (re-)load also tends to grow due to technological advances.
- **seasonal peaks**: Christmas, Eastern or Black Friday come along with increased workload for e-commerce application segment. Parcel logistic systems face about twice as much load, web shops with a focus on gifts may have a seven times higher workload.
  (Even more **exceptional peaks** occur, e.g., in web shops that sell sparse products like concert tickets from a given point in time in a first-come-first-serve manner. These systems have to cope with a burst of requests from clients of users who want to be among these first to serve.)
- **weekly cycles**: The workload follows a certain pattern throughout a week. For business-to-business e-commerce applications, weekends have much fewer workloads than workdays. Within the workdays, the workload decreases from Mondays to Fridays.
- **daily cycles**: For business-to-business e-commerce applications, the workload often peaks at 11 a.m. and drops after 6 a.m.

Example: Figure 18 and Figure 17 show a workload profile that is typical for a parcel logistics system. They contain all the patterns described above.

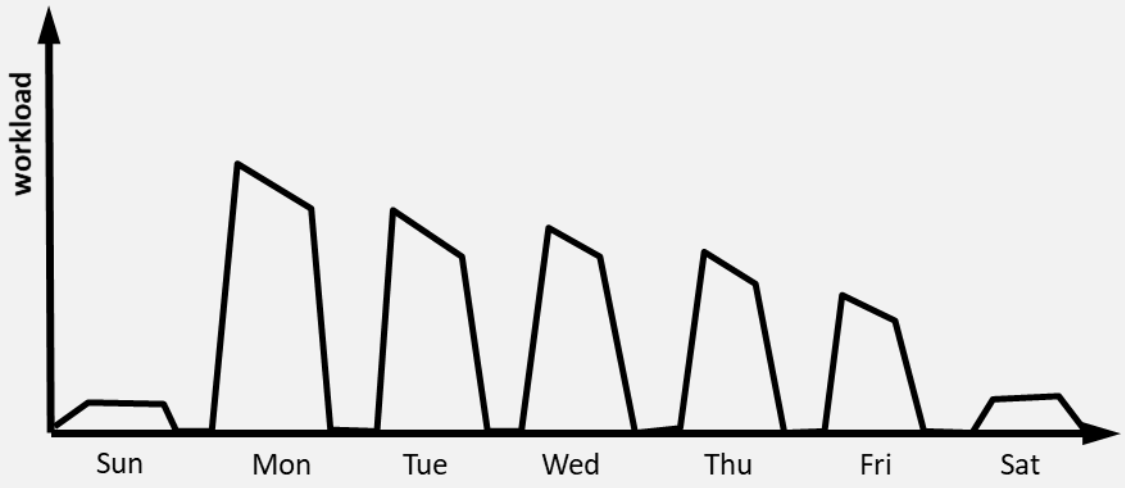


Figure 17: Workload profile for a single week

[19] Scaling a system's resources in order to fit the workload is not the only contributor to efficiency. Profiling your system and removing bottlenecks in the code base is often a better way.

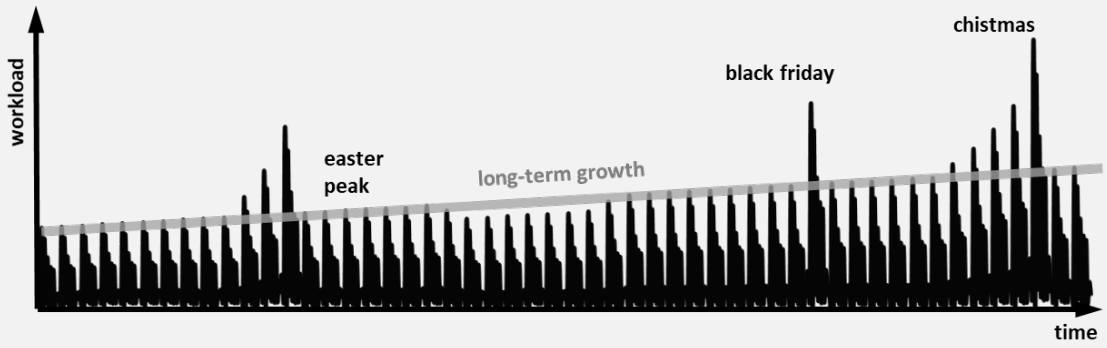


Figure 18: Workload profile for a whole year

7.7.2 Vertical and Horizontal Scaling

Basically, there are two options for scaling a system:

- One can add additional resources to existing (virtual) machines of a system, i.e. more CPU, RAM, disk space or network bandwidth. This is known as **vertical scaling** of a system or **scaling up/down** a system. Advantages of this sort of scaling include
  - that no additional complexities and problems arise, which come along with distributed systems, particularly with distributed databases, and
  - it works for non-distributed systems, too.
- One can also distribute the workload over more (virtual) machines of the same type, i.e. several machines each of which runs the same programs, while leaving the resources equipment of each machine as-is. This is known as **horizontal scaling** or **scaling out/in** a system. Advantages of this sort of scaling include that
  - there are no natural boundaries like max. CPUs per machine that limit this kind of scaling and
  - machines can be geographically distributed over different data centers in order to reduce latency or to deal with complete data center outages.

The term **elasticity** refers to the capability of a system to scale in an automated fashion, i.e. to react automatically to changes in the current load.

Also note that performance and scalability are not synonyms. A system might have a very good performance for a certain workload but cannot be scaled at all if the workload exceeds a certain threshold.

8 Dependability

Dependability is about correct and timely responses of a system from the perspective of a client or user. There are many levels of how this quality can be violated. A system might always or sometimes not respond not at all or not in a reasonable time span or with just an error message[20]. If it responds, the responses might be plain wrong, wrong by pure chance, or served outdated (stale) from a cache and thus inconsistent. In this Chapter we focus on fault states where systems do not respond or with stale data.

8.1 Faults and Failures

Many stakeholders of web applications are concerned about if a system is operational and produces correct responses all the time[21]. However, **faults** in a system occur and lead to a complete or partial system **failure**. Note that a failure of a single system resource might be just a fault from the perspective of the entire system.

Hardware faults are reason for 10% - 25% of data center outages [13]. These faults include faulty hard drives, RAM or network resources.

Software faults are another type of faults. These faults include excessive consumption of resources that are shared among processes, e.g.

- CPU if a process is stuck in an endless loop,
- RAM if there are memory leaks, or
- network or disk I/O.

Human errors[22] constitute another class of faults and are mostly configuration errors. According to [13] humans are the most likely source of faults.

8.2 Availability

8.2.1 Expected Availability

For sake of simplicity, we assume that a system is either up or down, i.e. it either produces (a) correct responses (b) within required response times or it does not respond at all. We call these states **uptime** and **downtime**, respectively. A transition from uptime to downtime is a **failure** and from downtime to uptime a **recovery**.

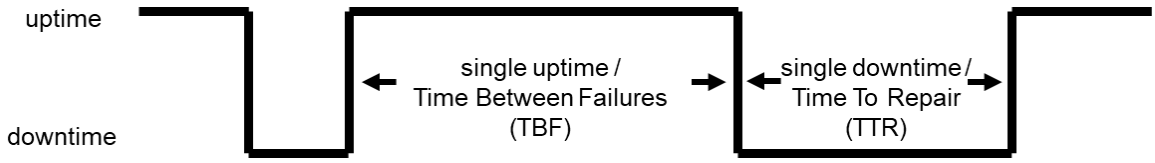


Figure 19: Up- and downtimes

[20] For web applications with a fault behind the load balancer the requested response is normally replaced by a response with a HTTP status code 5xx.

[21] Conversely, in high performance computing a fault is escalated to a failure such that the ongoing computation stops. After repair, it just resumes from the last checkpoint [5].Complaints from stakeholders in HPC applications are unlikely nonetheless

[22] Of course, one can consider software fault a subclass of human errors as they are made by human developers.

The **availability** of a system is the probability that—for a given *point in time*—the system produces correct responses.

Mostly, one refers to the **expected availability** A of a system in a given time span, which is

$$A = \frac{MTBF}{MTBF + MTTR}$$

Equation 1: Expected availability

where **MTBF** is the **mean time between failure** in that time span

$$MTBF = \frac{overall\ uptime}{number\ of\ failures}$$

and **MTTR** is the **mean time to recovery**:

$$MTTR = \frac{overall\ downtime}{number\ of\ failures}$$

We consider MTBF to be equal to **mean uptime** and MTTR to be equal to **mean downtime** though actual recovery is just a part of a downtime besides, e.g., detection and analysis of the failure, time logistics take for getting a replacement part. Please note that the formulae above are not meaningful for edge cases like if the number of failures is zero. They are meant to converge to a meaningful value for long time spans and thus high number of failures.

8.2.2 Nines

Usually, required availability is expressed as percentage values just containing nines, e.g. 99% or 99.99%.

8.2.3 Combined Availability

Often, a system is composed of resources, e.g. nodes, each of which has a certain availability a_i on its own. It is crucial how these resources are combined. If they work in **parallel** inasmuch the workload is distributed over the resources and each resource can serve as a failover for another one, then the combined availability is higher than a_i for each i.

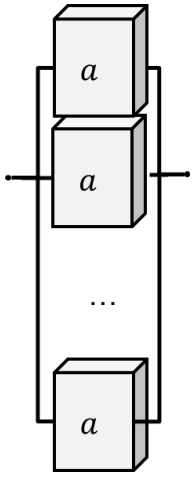


Figure 20: k parallel resources

In the following formula, we assume that each of the k resources have the same availability a. The converse probability $1 - a$ is the probability that a resource is not available, $(1 - a)^k$ is therefore the probability that all k are not available.

$$A = 1 - (1 - a)^k$$

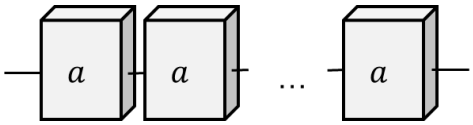
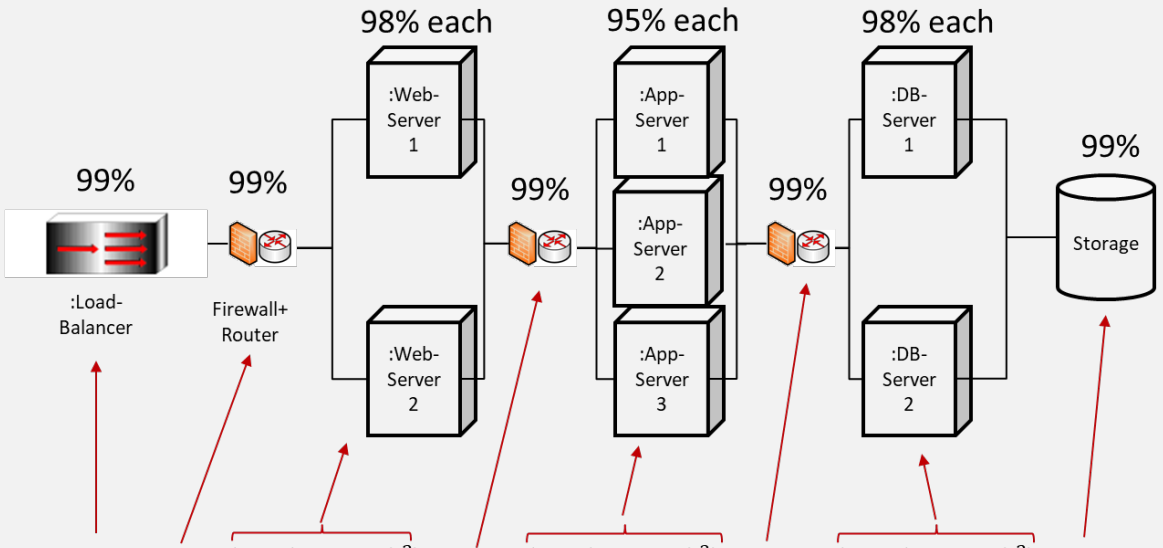


Figure 21: k serial resources

If the connection of k resources is **serial**, each of the resources must be available if the system is available.

$$A = a^k$$

A typical cluster consists of nodes which are arranged both in parallel and in sequence. Thus, combined availability of the cluster is a mixture of the above formulae.



8.2.4 Planned Downtimes

Planned downtimes are less bad than unplanned downtimes. Planned downtimes of a system X can be announced such that systems, people or processes that depend on X can be prepared appropriately. Such announcements are particularly important if the depending systems or users are paying clients of a company that runs X.

Planned downtimes normally cannot be avoided entirely as certain maintenance actions like updates cause downtimes. However, in a distributed system updates can often be executed in a way that upholds the availability of the system. For example, **rolling updates** for k parallel components

8.3 Reliability

The **reliability** of a system is the probability that—for the *uninterrupted* duration of a given *time span*—the system produces correct responses. Thus, reliability has a subtle different meaning than availability.

In systems where an incorrect response or downtime immediately leads to catastrophic results, reliability is more important than availability. Regarding downtimes, low frequency of downtime event is important here more important here than the average duration of a downtime, i.e. MTTR.

> The power supply of hardware is supposed to be reliable as even short outages lead to uncontrolled reboots. Only high availability is not helpful in this case.

Conversely, in systems where the impact corresponds to the duration of a downtime, i.e. MTTR, availability is more important.

> A client-server system (without transient server-side states) that has a downtime of one second after each 99999 seconds (i.e. a little more than a day), has an availability of A = 99.999%. So, on average, just 0.001% of requests won't get a response.

8.4 Resilience

The term **resilience** or **fault-tolerance** of a system is closely related to the terms availability and reliability. Basically, resilience means that

- certain internal faults within a system can be compensated or
- if system X depends on an external system Y, X can compensate for a failure of Y.

These compensations keep MTTR low and/or lead to a slightly degraded system operational state instead of a complete failure. Compensation methods include the following:

8.4.1 Exception Handling

Certain faults can be treated by additional branches of execution in programs. Many programming languages provide the concept of exception handling that is meant for dealing with such faults. The "degradation" in this case usually is that—though the program still runs normally—a certain function cannot be executed.

> A file might be opened by some process for read. If a Java process tries to write to that file, a `java.io.IOException` is "thrown", which could be "catched" elsewhere in the Java program and, e.g., inform the user by a message dialog that he or she should close the other program.

8.4.2 Failover for Parallel Resources

If a resource like a node fails, then there might be redundant parallel resources that serve as **failover** for each other. Here, one might distinguish two cases:

- **Active-Active**: Resources that are also active in normal uptimes take over the workload of a failed resource. This is usually the case for parallel web servers, application servers or database nodes with **multi-leader replication** in a system cluster. Failover usually works in a completely automated manner, i.e. a load balancer detects the failure of a web or application server and removes this particular server from load balancing. Then, we have a partial failure of the system. It affects just some of the requests between the failure of the server and the reaction of the load balancer, which is typically about 5 seconds. After the reaction of the load balancer, there is a performance degradation since fewer servers must handle the same load.
- **Active-Passive**: In this case, the failover resources do not get workload in normal uptimes, i.e. they are passive. This is the case for database clusters with **single-leader replication**. in which just one leader node accepts writes and the replica nodes only reads. Again, failover might happen in an automated fashion. Between the failure of a resource and the completion of the failover, there might be a system downtime. In case of databases, the MTTR is typically in the two-digit seconds.

Container orchestration middleware like Kubernetes can circumvent the performance degradation for an active-active failover by moving processes (contained in so-called pods) running on unhealthy machines to healthy ones. If processes themselves go wild, e.g. by consuming more memory than specified, Kubernetes can get them straight again by just restarting the enclosing pod on the same machine.

8.4.3 Fall-back for Serial Resources

Sometimes, a system X depends on another system Y in order to fulfil all or some of its functions. So, these systems are serial resources in the sense of Figure 21. If Y has a downtime, failover to X is pointless. Instead X might implement a so-called **fall-back** which surrogates Y.

> Example: Cache
>
> For cases where X **pulls** data from Y, responses of Y might be cached in X in normal uptimes. Cached responses can then be used in case Y fails. Here, the quality degradation of X is the risk of hitting stale responses in that cache.
>
> For cases where X **pushed** data to Y, both systems might be decoupled by replacing a synchronous call by a message broker (cf. Section 14.4)**Fehler! Verweisquelle konnte nicht gefunden werden.**). A quality degradation is that Y might reject messages without letting X know in a synchronous response.

8.5 Consistency

Consistency is the absence of contradictions. It is particularly broken if clients get different, i.e. contradictory responses for the very same request even if no changes to any data has happened in the meantime. Reasons for inconsistencies are

- **caches** that serve **stale data** or
- replicated database whose **replicas** have diverged due to network partitions.

In classical, mostly **relational database** systems, consistency is a sacred property. It is enforced by database transactions, which leverage the so-called **ACID properties** of a database system. ACID stands for

- **atomicity**, i.e. a transaction either succeeds or fails but nothing in between,
- **consistency**, i.e. every transaction enters and leaves a database in a consistent state,
- **isolation**, i.e. it does not matter whether several transactions are executed concurrently or sequentially,
- **durability**, i.e. once committed, a transaction remains committed even in the event of a database failure.

In many **NoSQL databases**, consistency is partially sacrificed in favour of availability. These databases merely adhere to the so-called **BASE** properties, which stands for

- **basically available**, i.e. a database is highly available,
- **soft-state**, i.e. the state of a database might change even if there are no writes to it (due to ongoing synchronizations between database replicas) and,
- **eventually consistent**, i.e. the database finally becomes consistent, i.e. replicas get synchronized, if there are no additional writes.

8.5.1 CAP Theorem

The **CAP theorem** summarizes the trade-off between availability and consistency: **Network partitions** that prevent communication between nodes cannot be entirely prevented in distributed systems. For example, In the event of a network partition in a multi-leader database cluster, one has to decide between

- **consistency**, i.e. the node detects the partition and reject all requests or at least write requests (thus sacrificing availability) or
- **availability**, i.e. each node still accepts even write requests knowing that its replica cannot be kept in sync with the other (thus sacrificing consistency).

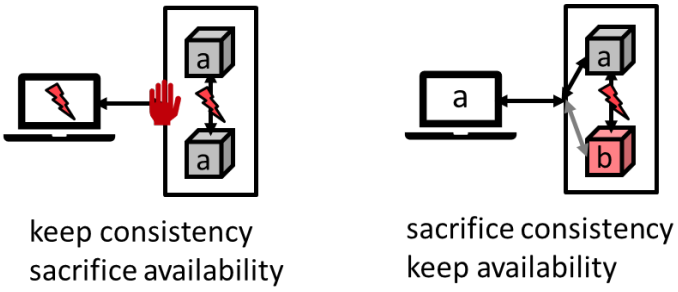


Figure 22: CAP theorem

This dilemma is stated by the CAP theorem in its original form in a somewhat awkward way: One can have at most two of the three: (1) availability, (2) consistency, or (3) partition tolerance [10].

8.5.2 Performance and Availability vs. Consistency

Scaling out a database cluster with replicas yields better performance since workload can be better distributed and higher availability since there are more servers available for failover. On the other hand, having multiple replicas might lead to inconsistency as stated before. So, performance and availability go hand in hand but both compete with consistency.

9 Maintainability

In general, maintainability is the effort necessary to operate a system at runtime or to change it at edit time. It is an umbrella term for more specific terms.

9.1 Operability

Operability is the effort necessary to run a system. **Monitoring** ensures to keep relevant stakeholders (e.g. admins or support) informed about the current system state and its event history. Monitoring can be subdivided into functions that are about metrics, logging and tracing, which overlap to a certain extend (cf. Figure 23).

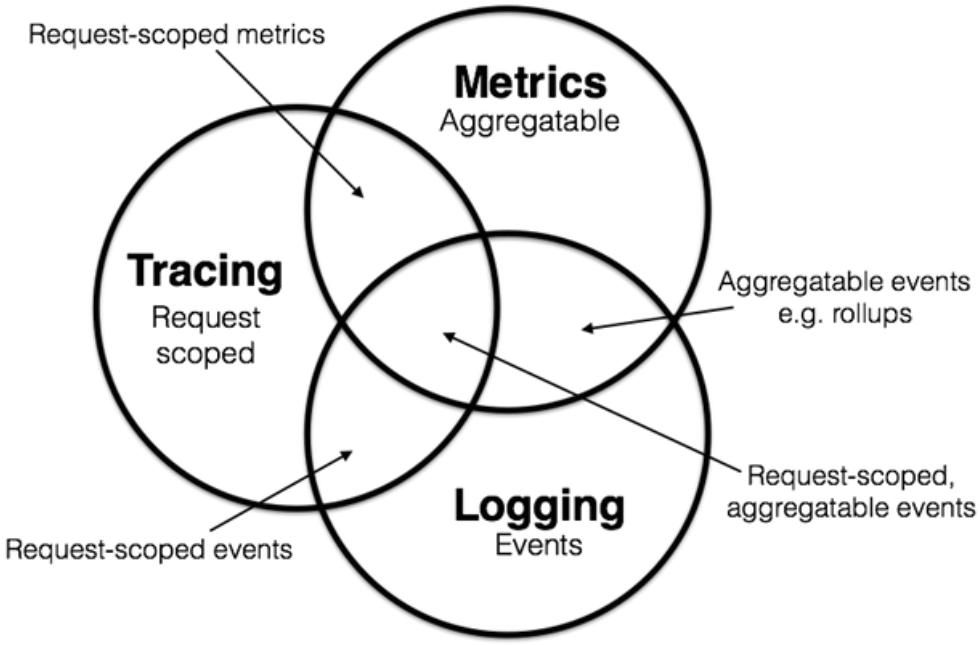


Figure 23: Metrics, tracing, and logging [14]

9.1.1 Metrics

Metrics focuses on the current or recent state of processes or machines, presented in an aggregated manner. Common metrics collect and present technical values, like CPU utilization at some point in time, the number of certain events like requests in a certain time span, utilization of some queue at some point in time, (mean) duration of a certain function call etc. **Business metrics** measure, e.g., the overall response times of a certain use case's page flow.

A monitoring tool notifies admins if certain thresholds of capacity or utilization are exceeded. Relevant metrics should be displayed in an aggregated view. Figure 24 shows some metrics as presented by the popular analytics and monitoring tool Grafana (https://grafana.com/).

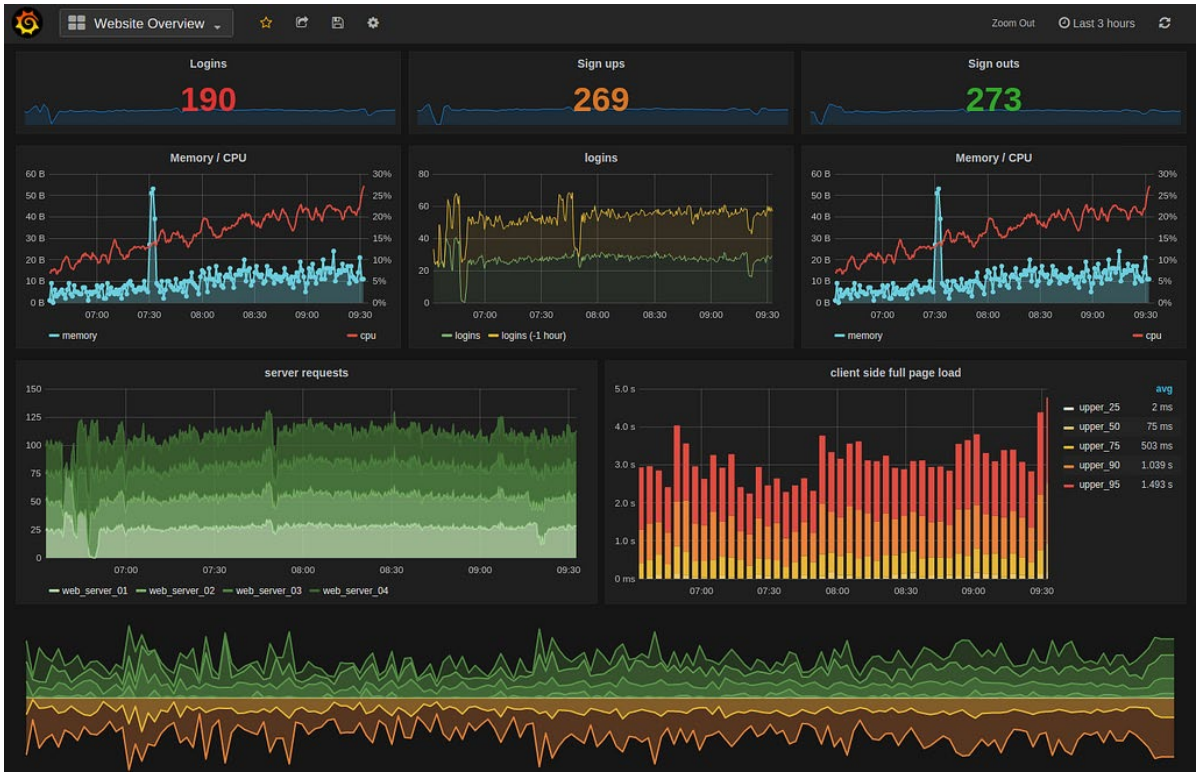


Figure 24: Grafana dashboard by Linux Screenshots (CC BY 2.0)

9.1.2 Tracing

A **trace** is the sequence of calls together with certain detailed metrics, e.g. duration of single call, input and output parameters, which are caused by a single request or during execution of one or more system's use case, a.k.a. **user's journey**. In a system landscape, a single trace might include processes of several systems, which call each other.

9.1.3 Logging

Logging focuses on the long-term **event history** of a single system containing, e.g., incoming requests, database accesses, outgoing requests to partner systems, completions of internal jobs etc. The aggregation of certain events again might serve as a metric. Logging tools usually provide functions for drilling down into certain **log entries** (i.e. events). Moreover, they allow for searching and filtering according to event attributes like severity or request id. This makes them even suitable for tracing. Figure 25 is a screenshot of the popular logging framework Logstash.

Figure 25: Logstash view

9.2 Simplicity

So far, we have dealt with quality attributes that are observable at a system's runtime. Simplicity and modifiability (cf. Section 9.3) are among those quality attributes that pertain a system's edit time.

If a new developer for some system X, needs comparably much time to understand how a certain change (bugfix, new requirement) can be implemented, we consider X **complex**, otherwise **simple**. Sometimes, the complexity of a system is quantified in **technical debt**, often expressed in developer working hours or days necessary to remove that debt from the system. Particularly systems with a broad functional scope (cf. Section 4.2) tend to become complex.

There are some basic mechanisms of human cognition that contribute to simplicity:

9.2.1 Chunking

A developer **cognition** works best if information, e.g. source code, is suitable for **chunking**. This is the case if an unexperienced developer can iteratively group (source code) elements to reasonable abstractions in a bottom-up manner or if an experienced developer can top-down apply memorized chunks to a new situation [15, p. 70].

> Example: When pupils learn to read, the first recognize characters, then whole words and even sentences. A natural language support chunking inasmuch there is a distinction between characters, words and sentences. Something like THISSENTENCEISNOTREALLYREADABLE is much harder to grasp than "This sentence is readable.".

9.2.2 Schema Building

In cognitive science, a **schema** is a combination of abstract and concrete knowledge [15, p. 78].

Example: Architectural patterns like the "model-view-controller pattern" are such schemas. Developers normally memorize the abstract nature of a certain schema, e.g. the roles of a model, a controller and a view. They also memorize at least one concrete example, e.g. a concrete application they have developed in which the MVC pattern has been applied.

9.2.3 Hierarchization

Information like source code is easier to grasp and memorize if it is presented in **hierarchical structures** [15, pp. 84-88].

Example: A programming language's block structure, i.e. the possibility to nest blocks of statements into each other, is an example for hierarchization.

Example: Programming languages, principles in software design, software architectures, data structures etc. provide additional examples, where these cognitive mechanisms are supported:

* Principles of Modularity, high cohesion and low coupling support chunking
* Using a control structure like a for-loop in a proper manner is the application of a schema
* Inheritance hierarchies or aggregation hierarchies are examples for hierarchization. Accidental cycles in these structures

9.3 Modifiability

Modifiability (a.k.a. evolvability) is about how easy it is to actually implement a change in a system. To anticipate future requirements and to build a system that could easily evolve towards these future requirements maybe is the hardest part in architecture work.

In some situations, simplicity and modifiability go hand-in-hand.

Example: If complex data types are properly encapsulated in classes, this normally leads to simplicity and modifiability as changes to the data types (e.g. an additional attribute) have to be made just in a central place.

In other situations, simplicity and modifiability are opposing forces.

Example: Internationalization (i18n), i.e. supporting several natural languages, is a common requirement for web applications. Having string literals in switch-statements with cases for each language all over the source code might be design decision that promotes simplicity. However, it would certainly hamper modifiability, e.g. the adoption of an additional language. Fortunately, there are lots of frameworks that support modifiability w.r.t. internationalization for the price of an additional layer of indirection and thus a slightly higher complexity.

Conversely, intentionally adding complexity, e.g. by introducing unnecessary indirections, in order to cope with very hypothetical future requirements is a common antipattern in software engineering, also called **over-engineering**.

10 Security

10.1 Basic Terms

Security is the resilience of a system against potential harm deliberately caused by (criminal) others. In contrast, **safety** is a system's resilience from accidental harm. The German term "Sicherheit" is ambiguous since it translates to both security and safety. Those system parts that support information-related activities and are valuable resources for criminals are called **assets**. These assets could be confidential data records in database, files in a file system, or critical soft- or hardware. They are at risk in the presence of a system weakness, also called **vulnerability**. A vulnerability would be harmless if there were no associated threats. A **threat** is way to exploit a vulnerability. A mere threat might become an actual **attack** or **incident**, that is an actual exploitation of a certain system's vulnerability.

Let X be a common web application where different users store confidential information, e.g. credit card numbers.

Asset: Data of other users.

Vulnerability: The source code line
`stmt.executeQuery("SELECT * FROM USERDATA WHERE userId = " + userId + " AND category = " + category)`

Threat: SQLInjection - If values are copied unsanitized, i.e. cleansed from SQL keywords, from a web input fields into SQL statements, additional confidential data might be exposed.

Attack: Actually submitting the value `'"whatever" OR 1=1'` for `category` in a web input field of the respective system.

Security is a broad field that comprises several overlapping subdisciplines. **Information security** mainly focuses on the asset "data", i.e. the protection of "information or information systems from unauthorized access, use disclosure, disruption, modification, or destruction" [16]. Its purpose is to uphold confidentiality, integrity as well as availability of information also known as the **CIA triad**. **Application security** focuses on the asset "application" with its vulnerabilities in each stage of its lifecycle (design, development, deployment, maintenance or retirement). In **network security** assets are network related.

10.2 Common Threats

The example in Section 10.1 is just one of common threats to typical web applications. The following table lists the "OWASP Top 10 - 2017" threats for web applications [17]:

Threat	Description
Injection	Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker’s hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
Broken Authentication	Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users’ identities temporarily or permanently.

Threat	Description
Sensitive Data Exposure	Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.
XML External Entities (XXE)	Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
Broken Access Control	Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.
Security Misconfiguration	Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched and upgraded in a timely fashion.
Insecure Deserialization	Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.
Using Components with Known Vulnerabilities	Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
Insufficient Logging & Monitoring	Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

Table 8: OWASP Top 10 Application Security Risks [17]

III. Design

11 Specifying Requirements

11.1 Quality Attribute Scenarios

In this chapter we describe a best practise for specifying architecturally significant requirements. **Quality attribute scenarios (QAS)** as defined by [18, p. 61 ff.] is a common text scheme that particularly suits quality attributes. (Conversely, **user stories** [19] rather suit functional requirements.)

A QAS is a sentence or short paragraph containing the following parts:

- **Stimulus**: An event on which the system (or its developers) must react somehow. Note that "event" is meant in a very generic way. It can be a mouse-click but also a new data protection law which requires modifications in backup processes of the system.
- **Stimulus source**: The emitter of that event. Again, this could be anything or anyone, e.g. some piece of hardware or a product owner or the legislation.
- **Response**: How the system should react. If the stimulus is a HTTP-GET-request for some web resource, the response should contain that respective web resource. If the stimulus is the modification request of a product catalogue in some shop system, the response is the correct implementation of this modification.
- **Response measure**: A observable criteria or event that determines if a response is satisfactory. For latency this might be a value for the 0.99-percentile, for a modification request this might be a test case which must succeed after the modification.
- **Environment**: Additional constraints that apply for the scenario. For latency this might be the current workload on the system.
- **Artifact**: The system or the system's resource to which the requirement applies. If in doubt, the whole system can be referenced here.

The downside of this generic pattern is that each part must be interpreted depending on the addressed quality attribute. In [18, p. 61 ff.] the authors present a **generic quality attribute scenario** for each quality attribute with valid keywords for each of the parts above.

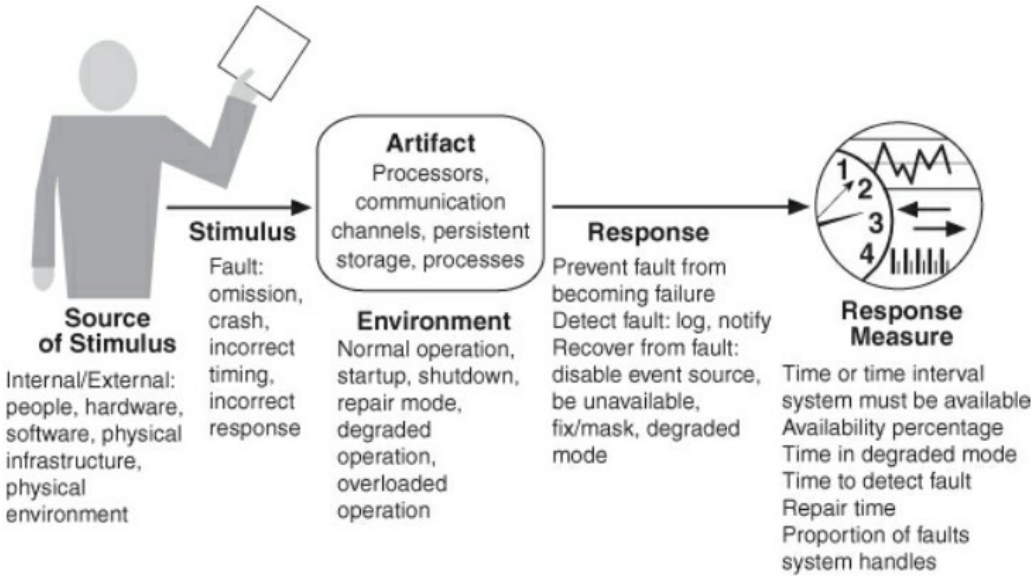


Figure 26: Generic quality attribute scenario for "availability" [18, p. 86]

For the quality attributes of Section 6, the generic QAS would look like this

Quality attribute	**Portion of scenario**	**Possible values**
performance	source of stimulus	internal (batch process) or external (clients)
	stimulus	arrival of periodic, sporadic, or stochastic events (like incoming requests)
	artifact	system (or system components)
	environment	operational mode: normal, degraded, peak load, overload
	response	process events, auto scale, notify for manual scaling
	response measure	response time, response time variation (jitter), throughput, dropped events (miss rate)
dependability	source of stimulus	internal/external: people, hardware (servers and network), software, physical infrastructure (e.g. electricity), physical environment (e.g. data center)
	stimulus	Fault: omission, crash, incorrect timing, incorrect response
	artifact	processors, communication channels, persistent storage, processes
	environment	normal operation, start-up, shut-down, repair mode, degraded operation, overload operation
	response	resilience: prevent fault from becoming failure detect fault: log, notify recover from fault: disable event source, be unavailable, fis/mask, de-graded mode
	response measure	acceptable availability or reliability mean time to detect fault, mean repair time
maintainability	source of stimulus	end user, product owner, developer, administrator, external system
	stimulus	request for adding/changing/deleting functionality, changing a quality attribute, adapting to new constraints (e.g. new data center)
	artifact	code, data, configurations, misc. resources, logs, processes, hardware
	environment	runtime (normal operation, non-normal operation), edit time
	response	implement, test, deploy change monitor, log, notify
	response measure	estimates (including time and money). how are artifacts affected
security	source of stimulus	internal/external human attacker or system
	stimulus	arbitrary attack vectors, e.g. brute-force password-guessing
	artifact	system functions or data (persisted or transient in RAM, network buffers)
	environment	operation mode: normal or non-normal connectivity of system: online or offline accessibility: accessible from internet, behind firewall

Quality attribute	Portion of scenario	Possible values
	response	protect assets from unauthorized access, track/log activities, block clients, notify administrators
	response measure	duration of attack, number of (resisted) attacks, amount of compromised data, duration of counter measure (e.g. client blacklisting)

A **specific quality attribute scenario** might take the suggestions of the generic QAS in order to formulate a requirement for a specific system.

In the following, we exemplify one specific quality attribute scenario each for performance, dependability, maintainability and security that applies for ReWo-Rate.

Performance

(Stimulus source) If a client
(Stimulus) sends a request for the average rating of a particular product
(Artifact) to ReWo-Rate's RESTful web API
(Environment) while ReWo-Rate is in normal operation mode with average workload
(Response) then a http-response containing the average rating and the number of ratings is returned
(Response Measure) with a processing time of less than 50ms in the 0.99-percentile.

Dependability (Resilience fall-back)

(Stimulus source) If the SMTP-server for outgoing emails
(Stimulus) is not available for ReWo-Rate
(Artifact) then ReWo-Rate
(Environment) while being in normal operation mode with average workload
(Response) buffers outgoing message and
(Response Measure) and resends them within 10 minutes after the SMTP-server is available again.

Maintainability (Operability)

(Stimulus source) If the SMTP-server for outgoing emails
(Stimulus) is not available for ReWo-Rate
(Artifact) then the logging of ReWo-Rate
(Environment) while ReWo-Rate is in normal operation
(Response) generates a log entry
(Response Measure) instantly with log level ERROR.

Security

(Stimulus source) If an external client
(Stimulus) repeatedly sends invalid login passwords for different users
(Artifact) to ReWo-Rate
(Environment) while ReWo-Rate is in normal operation
(Response) the current IP of the client is blacklisted
(Response Measure) for 1 day and the incident is logged in ReWo-Rate's logging.

11.2 Risks

Risks are pivotal in every development process. Budgets might be cut, key members might get sick, a competitor might scoop on a product under construction and so on. Yet, some of these risks are particularly connected to the system's architecture.

Example:

Risk: (condition) Given that ReWo-Rate synchronously calls an external system to verify the validity of an associated transaction / purchase during a rating and this external system is either slow or down
(consequence) then committing a rating might also be appear to be slow or even unavailable thus having a negative impact on the reputation of ReWo-Rate for the particular buyer.

Probability: high, since partner systems are accessed via internet which increases the risk for a partner system to appear down or slow

Impact: medium, just those ratings are affected that are associated with transactions in a slow / unavailable partner system

Countermeasure: Defer validation after the committing of the rating. If the transaction turns out to be invalid so will be the rating.

Note the distinction between condition and consequence in the example above. Do not mix them up. In order to quantify a risk, estimate the probability that the condition actually comes true and the impact of the consequence. Risk is usually defined as:

$$Risk := Probability * Impact$$

As long as we talk about mere subjective estimates, **probability** and **impact** normally have values from {low, medium, high} instead of numbers.

Countermeasures can be classified be

- **mitigating** the risk like the proposed one,
- **transfer** the risk, e.g. by covenanting a service level agreement (SLA) with each partner systems, which is enforced by penalties that would compensate a reputation loss, or
- just **accepting** the risk as-is.

The opposite of risks are **chances**: given a certain condition something good might happen.

Example: If implemented in Java, ReWo-Rate can be backported in ReWo-Rate, therefore avoiding redundant code bases.

So, while thinking about risks also think about chances.

12 Architectural Goals

In Table 4 we exemplified actual requirements per category. Section 11.1 (Quality Attribute Scenarios) showed us a way to formulate requirements which is particularly suitable for quality attributes.

In early discussions with stakeholders, it is not helpful to delve into requirement details, e.g. negotiating four nines availability. Instead, one should better outline which **goals**, i.e. stakeholder interests, have to be met in general. Unfortunately, goals sometime conflict and have to be traded with each other. Section 12.1 deals with such trade-offs. With such trade-offs in mind, it is sometimes useful to rate the importance of certain goals in order to set a focus early on.

In some situations, the most important goal is to be faster than the competition, while other situations require a software system of very high quality.

12.1 Trading Goals

12.1.1 Interest Trade-offs

Certain interests tend to oppose each other. First of all, it is commonly known in product development, that speed of development, quality, scope[23], and inexpensiveness oppose each other to some degree. The grey area in Figure 27 reflects the general performance of a development team or whole company: If you increase, e.g. the scope, while retaining the area of the quadrilateral, at least one of the other interests suffers.

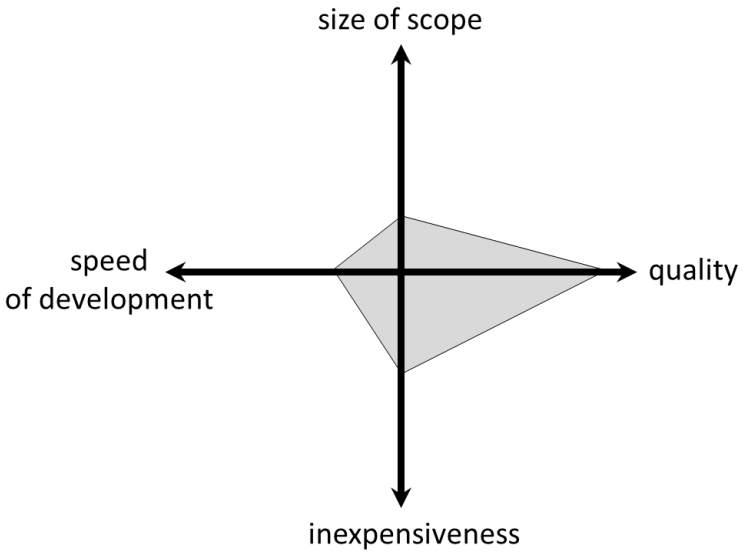


Figure 27: Opposing goals

12.1.2 Quality Attribute Trade-offs

The quality of a system is a complex goal that can be broken down to a hierarchy of quality attributes (cf. Figure 11) each of which constitutes a goal a stakeholder group might be interested in. Even within quality attributes there are opposing forces. In Section 8.5.2 we have seen how performance

[23] "Scope" refers to the number of implemented functional requirements, which is later on reflected by the number of supported functions in the navigation bar of an application, entity types in the database schema, supported user role etc.

and consistency do not go hand-in-hand. Finding the best compromise for each project and system is one of the most important tasks of an architect. Table 9 depicts some of the conflicts.

Conflicting pair	Rationale or example
performance vs. consistency	cf. Section 8.5.2
usability vs. security	A two-factor login increases the security of a system but lowers its usability as it requires a higher effort for each login.
performance vs. security	A TLS handshake increases the overall response time.
maintainability vs. speed of development	Often, maintainability of a software product is sacrificed in order to decrease the time-to-market of the first version, disregarding the fact, that the speed of development for later versions is severely hampered.
availability/performance vs. inexpensiveness	scaling out and replicate across different data centers increase availability and even performance but is also a costly thing to do.

Table 9: Conflicting quality attributes

With respect to ReWo-Rate, one could trade these conflicting requirements as follows:

performance vs. consistency: It is not that bad, if not each and every rating is considered during a request for a product's overall rating.

usability vs. security: Ease of use is surely predominant for ReWo-Rate. The actual ratings are not that highly confidential. A hard-to-guess URL satisfies most security requirements; two-factor-logins etc. are not necessary here.

maintainability vs. speed of development: This is something to discuss with the management. The case study suggests that ReWo-Rate is not a product that has to be pushed on the market with high time pressure.

availability/performance vs. inexpensiveness: One should spend some money in making ReWo-Rate highly available and performing as ReWo-Rate is a kind of product that delights less through neat functions but through high performance and availability.

In stakeholder discussions, it is often helpful point out such conflicts. Making the downside of a certain requirement explicit ensures that every stakeholder is aware of it.

12.2 Rating Goals

Before delving into requirement details it can be helpful to characterize a system by rating, i.e. prioritizing, common goals. Table 10 does so for ReWo-Rate.

Goal	Rating	Rationale
speed of development	low	ReWo-Rate faces no direct competitors nor has its release been publicly announced. The time-to-market-pressure is therefore below average.
inexpensiveness	medium	Building ReWo-Rate: Due to the lack of time pressure, ReWo-Rate can be developed by a small team with just a small communication and management overhead. Also, there are no special requirements w.r.t. developer skills which would increase costs. Operating ReWo-Rate: In order to achieve performance goals, enough money should be spent in ReWo-Rates cloud-based production infrastructure.

Goal	Rating	Rationale
size of scope	low	ReWo-Rate is deliberately restricted to a very limited set of functions, entity types, roles etc.
quality		
performance		
response time	very high	Response time directly affect the user experience of system that are integrated with ReWo-Rate.
throughput	high	In the long run, we expect a high workload (rating requests) that have to be processed (stored, aggregated etc.)
capacity	high	Since some products have a long lifecycle, a lot of ratings have to be stored.
scalability	high	We expect big variances in the workload (cf. Figure 18) and particularly a step long term growth. Therefore, ReWo-Rate should be highly scalable.
dependability		
availability	very high	Downtimes directly affect the user experience of system that are integrated with ReWo-Rate.
reliability	medium	Frequent downtimes are acceptable as long as they are very short so they do not affect availability.
resilience	high	ReWo-Rates is integrated with partner systems in the internet whose availability cannot be controlled. Failures of such partner systems should not lead to a complete failure of ReWo-Rate but at worst to a partial failure or stale data.
consistency	low	At least for transactional data, i.e. ratings, ReWo-Rate may be temporarily inconsistent. For example, it is okay if most recent ratings of a product are not displayed or considered in the computation of an overall rating.
maintainability		The maintainability of ReWo-Rate should be that of comparable systems. There is no particular deviation from a "medium" rating in each of the subcategories.
operability	medium	
simplicity	medium	
modifiability	medium	
security	low	The security requirements for ReWo-Rate can be low (for sake of usability) since stored data are not very confidential except user master data.
usability	high	Since users give ratings mostly as a courtesy, the usability of ReWo-Rate is crucial. A suboptimal usability would drastically lower the conversion rate, i.e. the transformation of a mere visitor to someone who actually does a rating.

Table 10: Rating goals

Admittedly, such ratings simplify things since, of course, different system parts could have different goals.

For master data including user passwords, consistency is crucial, while for certain transactional data like ratings in ReWo-Rate performance prevails over consistency.

Of course, such ratings are susceptible to be highly subjective. The best ratings are done by architects who have enough experience to benchmark a new system against comparable systems. For example, a "high" rating should be interpreted as "high in comparison with similar systems".

13 Architectural Principles

Architectural principles, though mostly not directly applicable, lay foundations for good design decisions. They are the rational for most architectural patterns and styles. The following architectural principles primarily aim at improving simplicity (cf. Section 9.2) and modifiability (cf. Section 9.3).

13.1 Abstraction

The **principle of abstraction** is ubiquitous in computer science: In order to reason about a system from a particular viewpoint, it is often helpful to leave out irrelevant details.

> A class in programming language like Java abstracts from concrete objects and their numbers at runtime as well as from concrete runtime values in the attributes defined in that class.

The best abstractions allow for forgetting the abstracted details not just temporarily but permanently without losing any control because of some automated mechanism that cares about these details.

> A variable "abstracts away" the actual bitwise representation of some value in memory as well as its actual location (RAM address). The compiler and/or runtime environment cares about accommodating variable values in the right form and place in the memory.
>
> The ISO/OSI stack allows for thinking of communicating layers despite the fact that actual communication takes place between the bottommost layers. Nonetheless, one can reason about application layer protocols without considering what happens in the layers beneath.

Abstraction is also in the heart of any kind of **modelling**. Here, those attributes that are existent in the modelled original but left out in the model are called "**disregarded attributes**" (in German: präterierte Attribute), those which are only in the model are called "**abundant attributes**".

13.2 Structuring

In information technology, everything is put in some kind of structure. We often encounter **relational structures** which consist of elements and relationships. **Graphs** are special relational structures, which have just binary relationships, called **edges**, among elements, called **nodes** (also: **vertices**).

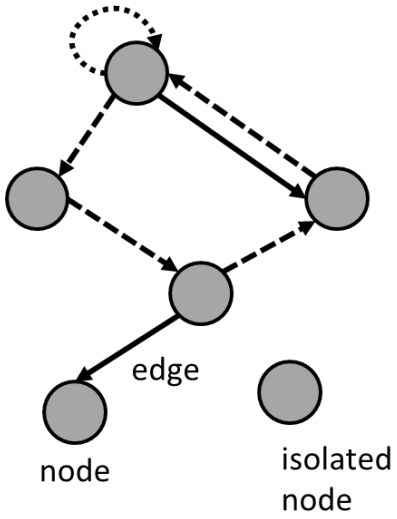


Figure 28: Graph with nodes (vertices) and edges

Acyclic graphs are graphs, which are free of **paths** of edges, which form a cycle (like the dotted ones in Figure 28). A node with just outgoing edges is a source and a node with just incoming edges a sink.

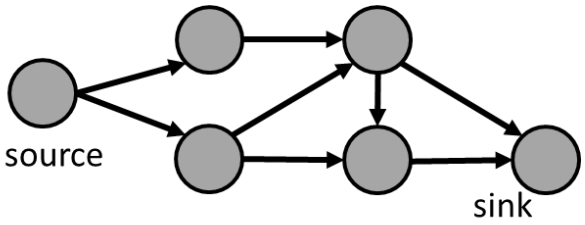


Figure 29: Acyclic graph

A subtype of an acyclic graph is a **tree** where each node has exactly one incoming edge except the only source (**root)** which has none. A node with an incoming and outgoing edges is called **inner node** and a node with just one incoming edge is a **leaf**.

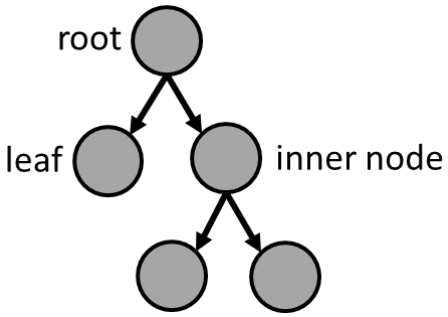


Figure 30: Tree

A **linear list** is a tree with just one leaf called **tail**. (The root is usually called **head**. Depending on the definition both might switch positions.)

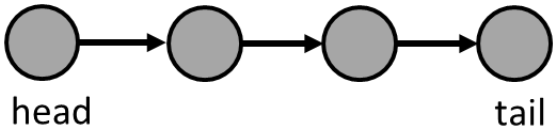


Figure 31: Linear list

A Gantt chart in project planning is an acyclic graph. A file system with directories and files is a tree or an acyclic graph if symbolic links are considered as edges as well. The class `ArrayList` in Java implements a linear list.

13.3 Locality

The **principle of locality** demands that coherent parts of a system are close to each other. Here, "parts" also includes documentation, lines of code etc. "Close to each" other means, that the parts are not scattered around different resources (files) or machines.

In a program C, a self-developed method might rely on another one of ready-to-use framework B, which again depends on another framework A. The developer has to jump between three different programs and frameworks each with their own documentation on their own sites.

Violations of the principle of locality due to using many frameworks which introduce many indirections are a commonly accepted downside of contemporary (open source) programming ecosystems. Back-tracing indirections is a common task of a developer.

13.4 Strong Cohesion

Cohesion is a measure for the "common ground" of a program's part (module, component, subsystem etc.). Generally, strong cohesion is desirable.

In a Java program, one could organize attributes and methods by alphabet in classes, e.g. a class A for all attributes and methods that begin with an A. However, these methods and attributes are likely to have nothing in common. For example, it is unlikely that the methods access the same attributes. Therefore, this is an example for weak cohesion.

```
class MethodsWithA {
  void add(int i) {…}
  void alterName(String newName) {…}
  void acceptCreditCard(String id) {…}
}
```

A class that exhibits strong cohesion would rather look like this:

```
class Set<T> {
  Set unionWith(Set<T> other) {…}
  boolean hasElement(T element) {…}
  …
}
```

The principle of **separation-of-concern** and the **single-responsibility** principle highly correlate with strong cohesion.

13.5 Weak Coupling

If the probability is high that program part B must be modified if part A is modified (or vice versa) then the coupling between A and B is said to be strong.

Consider Figure 32 to be a model of a Java program which consists of six public methods (white boxes), 10 package protected boxes (grey boxes) and four attributes (black boxes). The outer boxes represent classes and the arrows represent (call or access) dependencies. Structuring a program like this leads to a strong coupling between classes. Moreover, the cohesion is weak as the parts within the classes apparently have no commonalities.

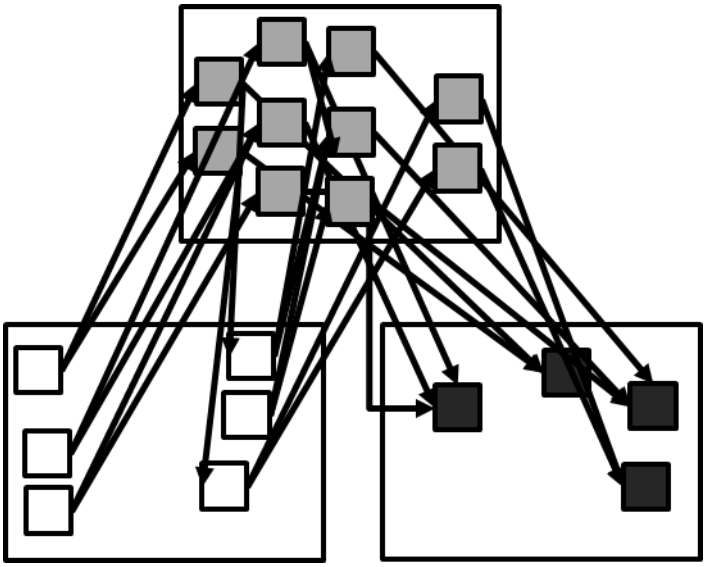


Figure 32: Badly modularized system with weak cohesion and high coupling

In Figure 33 we have the very same program with the same dependencies just with a different assignment of the methods and attributes to classes. Here, most dependencies are contained within classes and just a few are between classes so in comparison, we have a strong cohesion and weak coupling here.

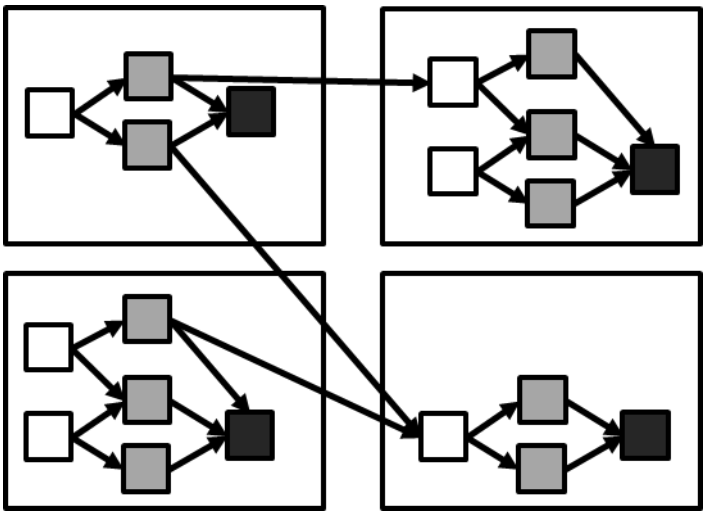


Figure 33: Well modularized system with strong cohesion and weak coupling

13.5.1 Dependencies

Weak coupling corresponds to

1. a low number of **dependencies** between program parts,
2. a high **locality** of the dependencies (this is what makes Figure 33 from Figure 32=
3. how **explicit** such dependencies are modelled which eases the detection of violations particularly
4. if violations are **automatically checked** (by a compiler, linter, runtime environment) and
5. if they are **checked early**, e.g., during editing time

These properties 2 to 5 make a dependency somewhat benign, their absence malign.

A bening dependency in that sense would the static invocation of a method by another one:

```
class A {
  static List<Animal> search(String criteria) {
    List<Animal> result = db.lookup(criteria);
    return result == null ? Collection.emptySet() : result;
 }
}

class B {
  void printMammals() {
    for (Animal animal : A.search("mammal", 23)) {
      System.out.println(animal.toString());
    }
  }
}
```

Here, the Java compiler can detect the mismatch between the declaration of the search method and its invocation with one more parameter. Since modern IDEs have built-in programming language support or make use of IDE agnostic language servers [20], developers are notified even earlier about this mismatch that is during editing time.

A less benign dependency stems from the assumption that search always returns a (potentially empty) List-object, so the caller can skip null-checks here. This dependency could be violated by changing the body of the search method, i.e., by actually dispatching the null value. This would be typically detected during tests of the respective program where the Java runtime environment would throw a NullPointerException (NPE).

The most malign dependencies are those whose violations are left undetected for a long time even if they made their way to the production system. Mutual assumptions about data schemas and formats are among these dependencies

Consider a database column for sums of money in dollars where the values are strings (VARCHARs in SQL speech). Readers and writers to that column have particularly to agree on what character is a decimal separator and what is for grouping digits (comma or decimal point). Undetected disagreements can lead to financially disastrous consequences in this case.

One way to make such dependencies less malign is to make such mutual assumptions explicit. This can be done by **explicit database schemas** and **strong programming languages types** whose violations by readers or writers can be detected early, e.g., compile time.

Please note that the “runtime usage” of a dependency does not account for stronger coupling. For example, if a method call is wrapped in a loop with one billion iterations, this coupling does not get stronger. So, the maintainability of a system is not affected. However, the performance would be affected particularly if the call is a remote method call.

13.5.2 Connascence

A more sophisticated way to categorize dependencies and dependent program parts is the notion of **connascence** [21]**.** In [22], the authors identify three properties that contribute to connascence:

1. **Strength of connascence**, where strongly connascent dependencies are harder to detect. Moreover, w.r.t. mere programming languages, the strength can be classified be the following

categories, where, e.g., name connascence is a weaker and meaning a stronger form of connascence

- Static connascences
  - Name, e.g. if the name of a method is changed, callers must be adapted as well
  - Type, e.g. a caller and a callee must agree on the type of a passed parameter or return value
  - Meaning, e.g. the meaning of status codes
  - Position, e.g. callers of the adhere to the positions of the formal parameters when calling a method
  - Algorithm, e.g. a caller and a callee have to agree on, e.g., a certain encoding like utf-8 or encryption, hashing or compression algorithm for certain exchanged data
- Dynamic connascences
  - Execution, e.g. if the invocation order of different methods of a callee is important
  - Timing, e.g. if two program parts concurrently write to a single value and have to be synchronized in order to avoid race conditions
  - Value, i.e. if several values have to be changed together
  - Identity, i.e. if several program parts have to reference the same entity.

2. **Degree of connascence**, which is a metric for the amount of dependencies a program part has.
3. **Locality of connascent program parts**, where dependent parts that are close to each other (e.g. methods in the same class) are easier to refactor.

13.6 Information Hiding

Modularizing a system like in Figure 33 not only keeps coupling low, but also follows the principle of information hiding. A class may only depend on public methods (white boxes) of other classes. The other members are hidden and if changed, might only break members of the same class. This is also been known as **encapsulation**.

14 Architectural Patterns

14.1 Patterns on Different Levels

Principles like those of Chapter 13 lead to certain recurring patterns, which are applied when software is built. Patterns are often classified according to the level of granularity on which they could be found:

- **Architectural patterns** and **styles** are about how a system or system landscape is distributed and how the distributed processes interact with each other, e.g. in a peer-to-peer manner via messaging. Sometimes they refer to a profound design decision in a single system which is hard to change afterwards, e.g. organizing the code base in a three-layered software architecture and implementing the frontend in a model-view-controller variant.
- **Design patterns** are about proven solutions for detailed software design. For example, leveraging information hiding by using the design pattern "façade" is common practice [23, p. 185 ff.].
- **Idioms** are on the level of code lines. They provide solution for common programming tasks like filtering a list according to a certain criterium and include programming best practises like yielding an empty list instead of null.

Table 11 further characterizes these levels.

	Architectural patterns and styles	**Design patterns**	**Idioms**
characteristic	fundamental organisation	proven solutions for recurring design tasks	proven solutions for recurring implementation tasks
subject	- distributed system - system of systems	- single-process system - component in a system	- single classes - bodies of methods
specific for	information technology	programming paradigm (e.g. object oriented)	programming language
examples	- 3-layered architecture - client/server model - monolith or microservice - messaging - MVC pattern	- observer pattern - factory pattern - singleton pattern	- "Hello World" - iteration through map - reverse string (http://www.programming-idioms.org)

Table 11: Patterns on different levels

14.2 Distribution

14.2.1 Reasons for distribution

There are different reasons that lead to distribution of a system:

1. Technical reasons:
    1. As stated before, each web application is distributed by nature just by the fact that **clients (browsers) and servers** a separate and normally remote processes.

2. On the server side, certain **middleware** like web servers, application servers and database servers run in their own nodes by design, which are also called **tiers** if arranged in a typical sequence[24].
3. **Horizontal scaling** (cf. Section 7.7.2) requires to run identical programs in different processes.

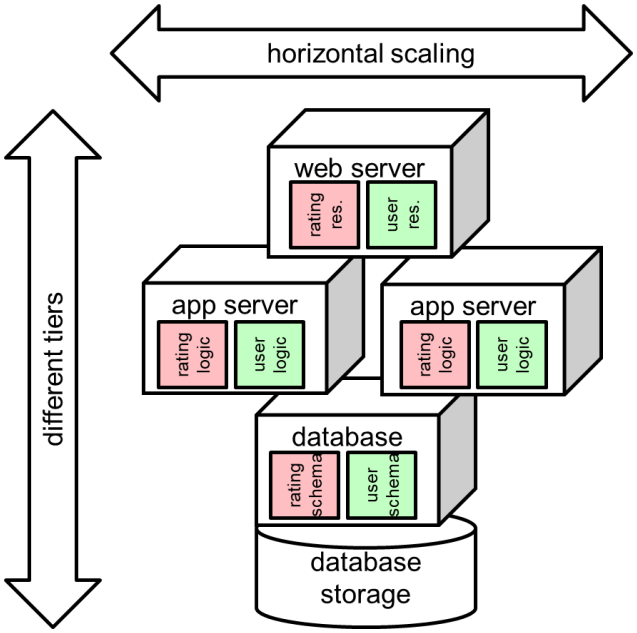


Figure 34: Distribution due to horizontal scaling and tiers

2. Organizational reasons:
    1. Systems of enterprise system landscapes are frequently affected by **organizational changes** like restructuring of departments as well as mergers and acquisitions. So, systems that have once been developed separately are integrated in a post hoc manner. In most cases, it is out of question to merge their code bases and to make one program out of two. Instead, they are left distributed.
    2. Systems that grow or could **grow big** in their deployment size can be distributed.

[24] Do not mix that up with **layers** (presentation, business logic, database access) which structure the program that is deployed to the application server.

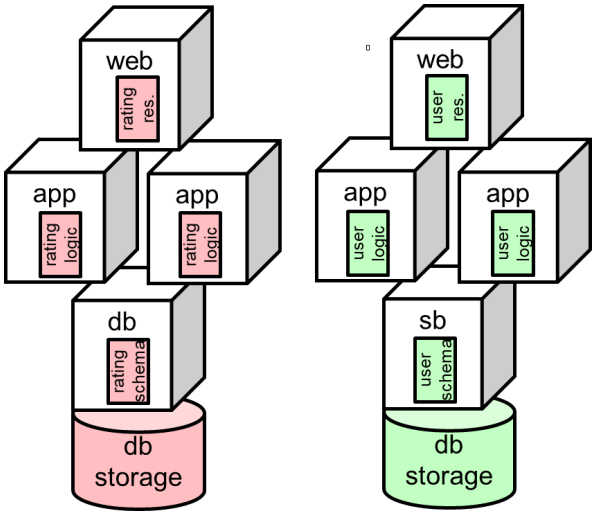


Figure 35: Distribution due to size

Except for the last one, the reasons for distribution are compelling. For the last one, there is an ongoing discussion about the up- and downsides of rather big systems (monoliths) and their distribution into smaller systems (microservices).

In the example of Figure 35, ReWo-Rate has been split into two microservices: One that provides function around rating and one for user management.

Since there is no real dichotomy between monoliths and microservices, we discuss both approaches in their respective extremes. In reality, there are also architectures that exhibit properties of both architectures. For example, Google's code base is stored in a monolithic repository (**mono repo**) but deployed and run in a microservice fashion [24].

14.2.2 Characteristics

a) Monoliths

In a **monolithic architecture,** distribution of a system just takes place for technical reasons (cf. Section 14.2.1). The code is being developed by **a big team** using a **single technology stack** (Java, Ruby, Golang, .NET …). A monolithic program is typically and versioned in a **single big repository**. Each tier is being deployed by a **single deployment artifact**, e.g., a big "Java archive" file (jar) for the application server, which runs in a **single (JVM) process**, except for identical replication due to a horizontal scaling.

b) Microservices

In a **microservice architecture**, a system that realizes an application is distributed also for organizational reasons. A **microservice** is a (sub-)system

- that implements just a portion of what would subjectively[25] considered an application
- that again might be distributed but just for technical reasons,
- has its **own runtime environments**,

[25] This subjectivity makes it sometimes hard to tell if a system is a (sub-)system / microservice or realizes an application on its own and therefore is a monolith.

  - i.e., runs in its own processes and typically in containers, which is an isolated tree of processes (cf. Chapter 19)
  - therefore, may have its **own technology stack**,
  - therefore, can technically be **(re-)deployed in independently** from other microservices,
  - usually has its **own data base / storage** or at least an own replica[26] of a shared database,
- communicates with other microservices only **via HTTP** and its extension
  - therefore, exposes HTTP-based application programming interface (**API**) and
  - sometimes a web user interface (**UI**) [27],
- has its own code base in its **own source code repository**,
- is **incrementally and iteratively developed as a product** as well as operated by a stable "**two-pizza team**", i.e., a team of about 6 people.

We can distinguish two microservices by what business functions they support. **Domain-Driven design** [8] helps to design a system of microservices in such a way.
Example: If you have attended the "Software-Praktikum" in "Technische Informatik (Bachelor)" you might remember the WaWi's component architecture of Figure 36:

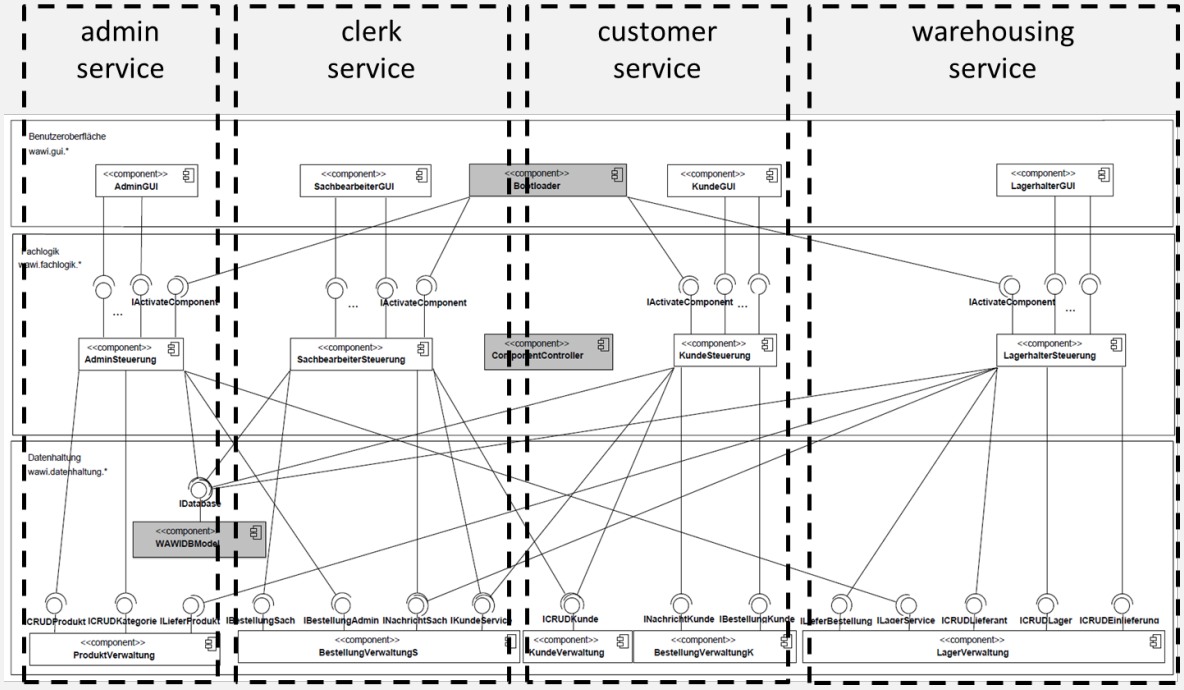


Figure 36: Component architecture of a "Warenwirtschaftssystem" (source: Software-Praktikum, Grobspezifikation WaWi)

[26] This makes a microservice independent of the availability of a shared database. An architecture that focuses on the independence of its parts (microservices) is also called a **shared-nothing architecture**. Counter examples are architectures with different microservices that access the same database or a scaled-out microservices with server-side sessions states which has to be synchronized among replicas (**session replication**).

[27] If a microservice serves its own web UI, this UI is also called **micro frontend** [73]. A microservices that serves its own micro frontend and has a database on its own like the ones on the right side of Figure 33 is also called **self-contained system** [74].

It is better to split this WaWi "vertically", e.g. having "admin service", one "clerk service", one for interaction with customers, and one for the warehousing would be better than to do it horizontally, i.e. having one microservice for GUI rendering, one for the business logic and one for persistence. Please note that the figure just reflects the application tier, which runs a program that is structured in three layers. The database schema within the database tier is not depicted but is supposed to be split as well.

There are certain **enabling factors** that partially gained traction in the last years and facilitated the breakthrough of microservice architectures:

- Virtualization, containerization and cloud computing makes rapid **provisioning** of resources (machines, storage, network, etc.) easy. Before that, provisioning efforts would have stifled microservice architectures.
- Speed and **time-to-market** gained importance, while the diversity of technology stacks became larger. It is faster and more secure to let a team implement a working minimal viable product or function in a new microservice with their favourite technology stack than to extend an old and big monolith.
- **Conway's law**, which states that the architecture of a system or of a system landscape tends to resemble organizational structures anyway. According to [25], such alignments actually help improving the quality of a software system and speeding up development. Microservice architectures are better suitable to reconcile with organizational structures than one or few monoliths could do.

14.2.3 Downsides of Monoliths

The **downsides of these monoliths** are the following:

a) Organizational Issues

- Big teams tend to work inefficient as **communications paths** grow quadratically with each new team member.
- Team members might not be familiar with the particular, **single technology stack**.
- Large teams working on a large monolith have a higher risk for committing erroneous source code changes that break the build or tests of the monolith and hamper other team member's **productivity**.

b) Architectural and Technical Issues

- Common programming languages offer constructs for code structuring, information hiding and modularization. However, (except for distribution due to technical reasons) a monolith is developed as a single program running in a single process in which all heap data (i.e. objects) and text data (i.e. functions) are technically reachable from everywhere else. Therefore, maintaining loose coupling of internal parts of a monolith highly depends on the discipline of its developers. This discipline tends to fall victim to **technical shortcuts** in face of deadlines. These shortcuts turn a once **structured monolith** into a so-called **big ball of mud** with a high **technical debt**.
- Technology stacks (languages, libraries, frameworks) become ultimately out of fashion, which makes it harder to acquire developers which are familiar with that stack. It also bears the risk that parts of the stack get out of maintenance (even for security updates). Normally, a program's **migration to another technology stack** can only be done a single step. This is harder

for a monolith due to its larger program size. Conversely, a system of microservices (i.e. multiple rather small programs) can be migrated incrementally microservice by microservice.

- **Build-times** (including compilation and test automation) and **start-up times** might grow large for monoliths (up to hours) therefore hampering rapid development.
- The larger a program, the higher the probability for bugs that risk the running process' **reliability**. At runtime, such bugs might cause a failure of the whole system and render the respective application inaccessible for users. Such bugs might be, e.g.,
  - too much or frequent allocation of heap **memory** that could cause
    - **garbage collector stops** for the whole process or,
    - even worse, get the respective **process killed** by, e.g., the Linux' Out Of Memory Manager (OOM) [26, p. 685ff]
  - overconsumption of **CPU** power due to, e.g., infinite loops or accidental complex computations like evaluations of certain regular expressions [27]
  - depletion of **thread pools**[28] since, e.g., all threads got stuck at the same call (to, e.g., some external service) that is relevant just for a certain functionality.

  Conversely, in microservice architectures there is a change for keeping such bugs isolated in the respective microservice and graceful degrading the application by just switching off the affected functionality.

In ReWo-Rate the non-availability of a microservice for the user management -- implementing use cases like "registration" -- might be "gracefully degraded" by redirecting the respective registration link to a static and informative http 500 web page.

- Even if just small parts of a monolith need to be updated, **redeployment** means that the monolithic process has to be restarted. Since a monolith normally exposes a lot of functions (invokable via different routes, API REST resources, web pages etc.), a lot of functions are potentially unavailable during the redeployment. This problem can be mitigated if the monolith is scaled horizontally and the redeployment is conducted in a **rolling update** manner.
- Moreover, the larger a monolith the higher the risk of an **erroneous redeployment** due to wrong configurations that would prevent the monolith from starting and again affecting a whole big application's reliability. Even worse, the risk is higher for **bad database (schema) modifications** thus risking the existing data of the whole application instead of just a subset.
- Processes running monoliths utilize a lot of resources like memory that makes them **unsuitable for binpacking** (cf. Section 16.4) or **fast migration** or even **fast rescheduling** particularly in cloud environments.

14.2.4 Downsides of Microservices

The downsides of a microservice architecture should not be neglected. In a way, they resemble those of distributed systems in a typical enterprise system landscape, except that, ideally, microservice are deliberately distributed each with thoughtful boundaries that once might have been parts of a monolith whereas systems in a system landscape have typically been developed mostly

[28] Using Apache Tomcat embedded in in Spring Boot is part of many server side technology stacks. By default, the Java thread pool is limited to 200 threads (cf. property `server.tomcat.max-threads` in [85]). So, a single JVM process can process at most 200 requests in parallel for all implemented functions.

independent from each other with (at least if the system landscape has been undergone mergers and acquisitions).

a) Increase of Complexity

Although each microservice has a lower complexity, the entirety of all microservices of a system has a **higher complexity** than a well-structured monolith of equal functionality.

- The **diversity** of technology stacks makes it harder to keep track of deprecations, necessary security patches and to reassign developers.
- **Integrating microservices** require additional efforts, e.g. glue code for network communication. This is particularly true if different microservices serve parts of the web UI, i.e., have micro frontends, or parts of a web API of a single web application, which is supposed to have a **coherent user experience** with a **single sign-on**.
- **Operations** faces more metrics as well as distributed and disparate log files to follow and correlate.
- What is a benign dependency in a monolith, like a normal in-process function call or dereferencing of a heap data record, might become more malign in a functionally equivalent microservice architecture if these dependencies overarch microservices boundaries. In this case, **broken dependencies** like deviating formal and actual function parameter signatures, are harder to detect. This makes it harder to avoid new bugs during independent evolution of these microservices and even to avoid regression in the course of a refactoring of the overall code base. So, a decent **design up front**, which particularly avoids overarching dependencies is even more vital of microservice architectures but nonetheless hard to achieve. These downsides can be mitigated if coherent microservices are still built together and versioned in a single so-called **mono-repo** and just deployed and run distributedly.
- The upside of independent deployability of different microservices comes with the cost that an overarching dependency just harmfully breaks at the event of the deployment (and not before). Keeping **backward compatibility** of overarching dependencies is therefore a bigger problem in microservice architectures and must be traded with maintainability.

b) Distribution of Systems

Some **problems are inherent distributed systems** and grow bigger in a more distributed system. These problems include the following:

- Communication over a network is inherently **unreliable**, e.g. due to congested network buffers. Conversely, in-process calls between different components within a monolith will never fail. Therefore, it is particularly hard to guarantee ACID properties for **distributed transactions**, particularly if performance is important.
- **Sharing in-memory data** between separate processes is infeasible if they are really distributed among different machines. Consequently, passing just memory addresses (references or pointers) is out of question. Therefore, processing data in a distributed fashion requires **frequent data transfers** and thus **replication** of that data along with **additional processing times** for (de-)serialization. From an isolation perspective, this might be considered an upside of microservices, from a performance perspective it is a downside.
- Communication over network induces **latencies** in the order of milliseconds, whereas latencies of in-process calls are in the order of nanoseconds and thus neglectable. In particular, whereas in-process calls might just pass small a reference to a common but very large data object,

transmission time (cf. Figure 13) gets recognizable when this large data object (e.g. a video) needs to be passed between microservices (which are, e.g., forming a kind of video processing pipeline). Moreover, **propagation delays** get significant if they add up since communications between two microservices (1) occur frequent and (2) are synchronous or (3) the microservices are quite far apart (e.g. not in the same data center). At least (1) and (2) can be mitigated to some degree: If one microservice B processes an array of data elements for another microservice A, then A should better put the whole array in a single request to B instead of looping over the elements and waiting for a response in every iteration or at least do the request-responses asynchronously. These problems have observed even before the advent of microservices, as Martin Fowler points out in [28].

- Due to the unreliability and latency of networks, it is hard to reach a **system-wide consensus** even about basic fact such as the exact current local time. For example, this turns out to be annoying when one tries to trace the exact chronology of events based on log files from different machines.
- Communication over network must be **secured** in order to avoid sniffing or tampering of messages. In-process calls do not need to be secured.

c) Increase of Overall Footprint

Early application servers were designed to run multiple (monolithic) applications. This was also due to reducing the overall memory footprint. In order to achieve enough isolation, running just a single application (packaged in a WAR or EAR file) per application server became best practise.

With microservices, this approach became even more extreme: Although, full-blown application servers got out of fashion, each microservice still comes along with a significant **memory footprint**. Splitting a monolith vertically (cf. Figure 36) yields microservices each with a memory footprint right after start-up[29] comparable to that of the monolith instead of just a proportional fraction of it. The same holds true for the mere start-up time. This is because the dependencies (libraries) of the monolith are mostly inherited to all microservices. However, build-time is significantly reduced mostly since just a proportional fraction of automated tests have to be executed. It is just the application specific code with accompanying tests that is split into small amounts of code per microservice. Together with its dependencies, a microservice is not really that "micro".

Such memory footprint is **expensive particularly in cloud environments**, where each gigabyte costs about 25$ per year (as of 2019) and vertical scaling large-scale systems quickly demands for memory in the magnitude of terabytes.

14.3 Communication Patterns

14.3.1 Communication Directions

A communication act between a nodes A and B might be **unidirectional**. In this case, w.l.o.g. node A is a sender and node B a receiver, which send and receive a **message**, respectively. If A and B both act as senders as well as receivers of messages, we have a **bidirectional** communication. The Transmission Control Protocol (TCP) and the WebSocket Protocol [29] are protocols for two-way network communication although on different layers of the ISO/OSI stack.

[29] Right after start up the size text section containing the executable code (cf. Section 4.3.2) dominates over the still unpopulated heap and stack section for common processes. For JVM processes the same holds true for the metaspace vs. the heap.

If a message has follow-up in the other direction, we speak of a **request** and a **response** messages in a bidirectional communication instead of just messages. The Hypertext Transfer Protocol (HTTP) [30] is a request/response protocol.

14.3.2 Synchronization Patterns

In a **synchronous** communication, a client process pauses execution by stopping the program counter, i.e. has a **blocking wait**, right after sending a request until it receives a response (or loses patience if the response time is greater than a certain timeout). The program model is easy in this case since the programmer can assume that the response is there (and normally accessible through a variable) right after the program statement that caused the request. However, such blocking waits are fatal if the client is single threaded since the client is completely irresponsive during response time.

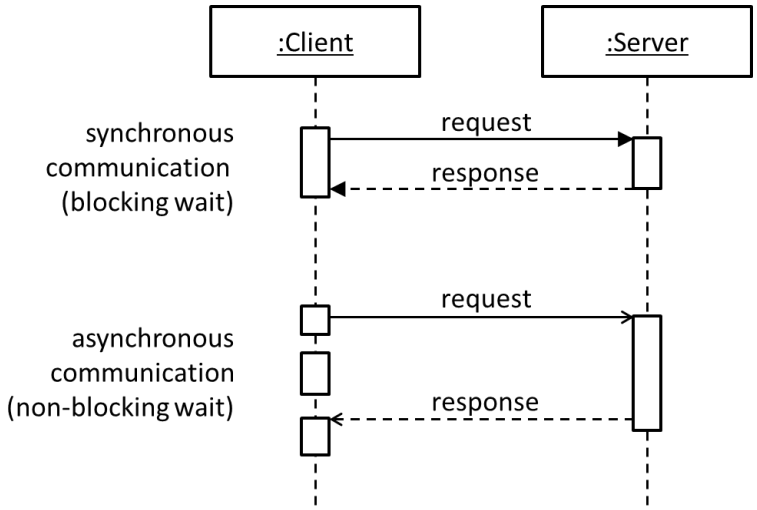


Figure 37: Synchronous and asynchronous communication

In an **asynchronous** communication as depicted in the lower part of the UML sequence diagram in Figure 37, a client process sends a message but does not block, e.g., the client process proceeds executing. This does not matter, if the communication is unidirectional. However, if the message is a request that causes a response from the server, the programming model is a little more complicated compared to synchronous communication. Usually, a handler function is invoked at the client side that processes the response.

> JavaScript (more precisely: ECMAScript) browser runtimes are single threaded. Therefore, http requests made by means of, e.g., the JavaScript Fetch API, do not block this single thread. JavaScript offers different concepts like Promises in order to keep the statement that causes the request and those that process the response textually close (local) to each other.

14.3.3 Communication Contents

A request might convey a rather small **query** with selection criteria for data that node B can look up, e.g. in a database or a file system. A request could also contain **textual or binary data** that is supposed to be stored or forwarded by node B. Alternatively, it is input for some kind of computation on node B, e.g. video reencoding or prime factorization, whose output is placed in the response from B.

14.3.4 Communication Origin

If node A sends a request to node B that contains a query and the response from B contains the answer for that query, we speak about a **pull** communication. If B proactively sends data in a message

to A, this is called **push** communication. If both nodes form a client-server system, there are four combinations of pull and push from a client or server. Table 12 summarizes these.

Name	(Request) message originates from	(Request) Message contains	Description or example
Client Pull	client	query	example: HTTP GET request that queries some web resource from the server
Client Push	client	data	example: HTTP POST / PUT request that is used to upload a web resource to the server
Server Pull	server	query	- rarely used - example: server sends keep-alive queries to clients in order to determine, which clients are still out there.
Server Push	server	data	- example: callback from server to client - example: notification about server-side data changes that a client is supposed to cache (cf. Section 8.4.3)

Table 12: Pull and push in client-server systems

By definition, a server is independent from its client(s). Consequently, a server typically does not know its clients nor their network addresses by itself and even if so. Therefore, server pushes are not that easy to implement. One could make a client known to the server by adding its network address to a configuration record on the server side. This works for stable client-server systems, which might be two nodes in the IT landscape of some enterprise. However, in the web, clients are often user agents like browsers that come and go and have dynamic addresses. Even if a server could be kept up to date regarding its current clients, ingress network traffic to clients is typically blocked by some firewall (on the DSL router or the operating system).

So, how can data be pushed from the server to the client anyway and particularly push **notifications** about data change **events** to the client?

a) Polling

When using **polling**, there is no server push in the sense of Table 12. Instead, there are frequent client pulls that query for latest events. The server might defer each response until there at least one event has occurred that could be put in the response. This saves at least some network traffic and is also called **long polling**. If there are no new events for a certain time, the client might run into a timeout by default since it must also assume that the server does not respond for other reasons. Then, the client sends just another request directly after that timeout.

b) Callbacks

When using network **callbacks,** the first request still originates from a client, in which the client includes a network address to himself. The server uses this network address later on to push its arbitrarily many replies to the client.

Even if there is just a single reply, this pattern can be used instead of a common request-response communication in order to circumvent timeouts.

Please note, that if callbacks are implemented on top of HTTP (i.e. the network address of the client is a URL), there are always responses (grey in Figure 38). The subsequent replies of the server, which technically are HTTP requests, go over another TCP connection than the initial request form the client and thus are likely to be blocked. For example, the WebSocket Protocol [29], to which an existing HTTP connection can be upgraded, is meant to solve this problem.)

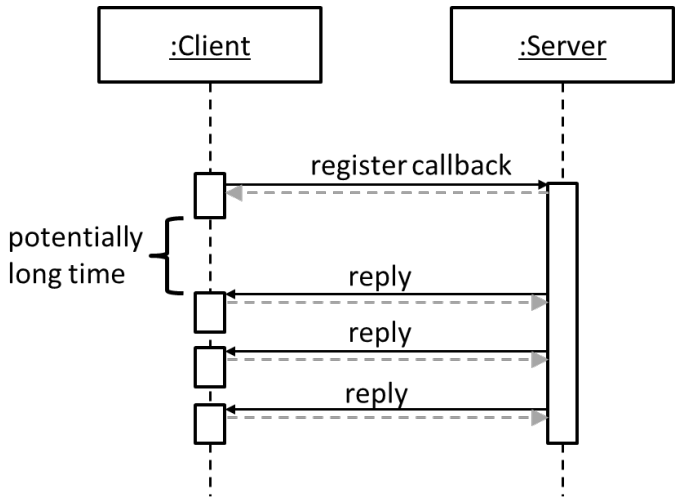


Figure 38: Callbacks

14.4 Messaging

Unidirectional messages between two nodes are the simplest form of messaging. **Message queues** add a layer of indirection between a sender and a receiver. Senders send messages not directly to receivers but to a **message queue** where they are (temporarily) stored and forwarded to receivers. This decouples senders and receivers in so far that all can be kept agnostic of each other's addresses.

Normally, messages are processed and forwarded in a "first in first out" (FIFO) manner (hence the name queue). If the message becomes the oldest, it is forwarded. Normally, if the first attempt fails, the delivery is retried until success. This decouples senders and receivers in so far as temporary failures of senders or receivers do not lead to system failures but just to message queues that temporarily have a lower or higher rate of unforwarded messages, respectively.

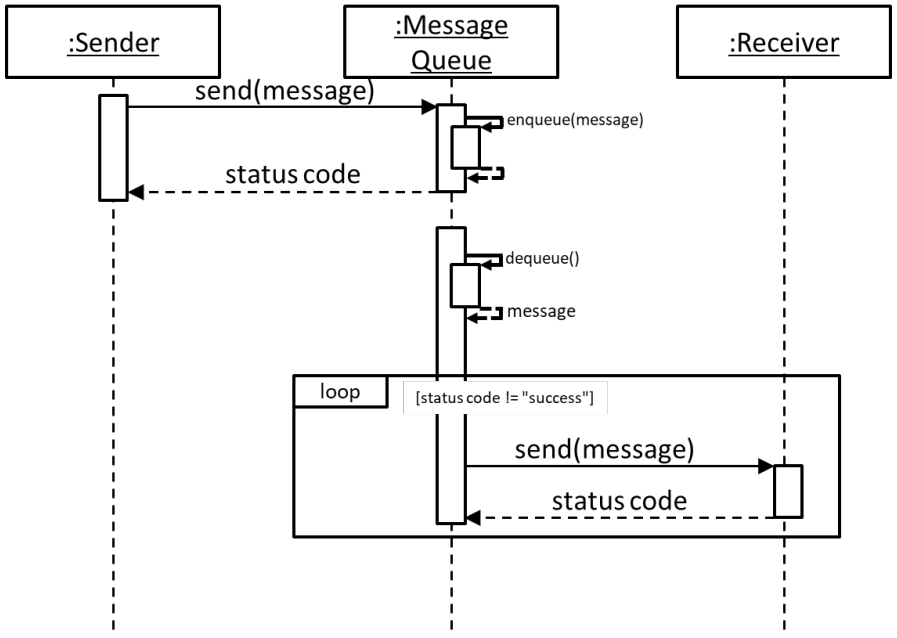


Figure 39: Sequence for forwarding a message with a message queue

A **message broker** normally handles multiple queues which can be addressed separately by senders and receivers. Depending on the context, these sub-queues are also called **channel**, **topic** or

stream. Normally, each channel, topic or stream contains messages (data records) of the same type and thus with a common structure.

14.4.1 Message Broker Characteristics

Different message brokers differ in

1. if they allow for **multicasting** a single message to multiple receivers or just forward a single message to a **single receiver**,
2. if they **delete** forwarded messages or **retain** them at least for a while, which is particularly if receivers are added to a multicast group which need to catch up,
3. if forwarding means that messages are **pushed** to the receivers or **pulled** from the receivers,
4. if they store messages in **memory** or on **disk,**
5. if they accumulate received messages for the duration of a certain time-slice, e.g., one day, into **chunks (batches)** or if they process or forward messages as soon as possible, i.e., keeping the batch size equal to a single message and thus keeping a **stream** of messages,
6. if they just **forward** messages or allow for **processing** them, e.g., validating, transforming, augmenting them with additional data or joining messages from different streams,
7. if they allow for **horizontal scaling** via partitioning and replication of streams in order to increase throughput and availability.

14.4.2 Messaging Use Cases

(The following mostly summarises [10, pp. 439-487].)

Even without processing capabilities (cf. list item 5 in Section 14.4.1), message brokers allow for a wide range of uses cases.

a) Multiple Receivers

It is not unusual that a single stream has multiple receivers, a.k.a. consumers (cf. (cf. list item 1 in Section 14.4.1), According to [10, pp. 444-445] one can identify two main patterns that determine which messages are forwarded to which receivers (cf. Figure 40).

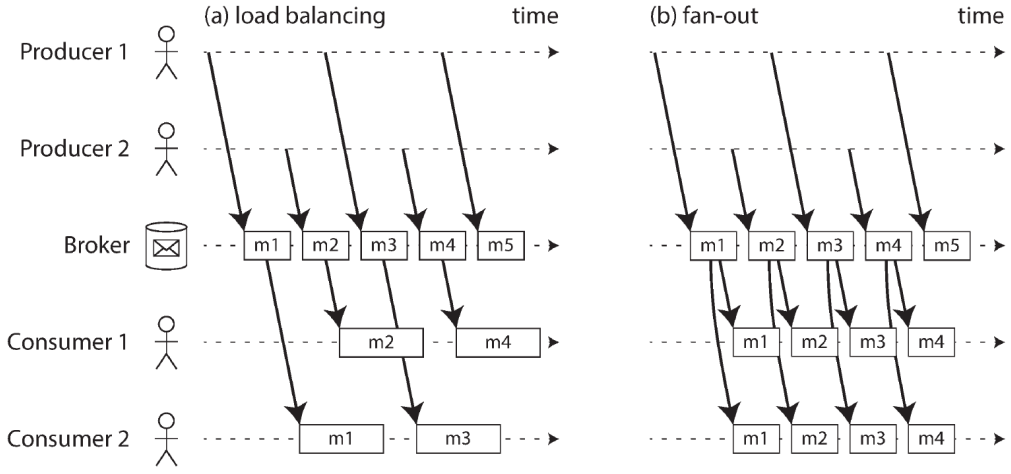


Figure 40: Load balancing vs. fan-out [10, p. 445]

A message broker can be used as some kind of **load balancer** (cf. Section 15.3) with additional capabilities. If processing takes place within the consumers and requires a lot of resources, each message could be forwarded just to one consumer.

If the intend to distribute data, then it is possible to use a **message broker** to **fan out** messages, i.e. send identical copies of a message to multiple consumers.

b) Redelivery

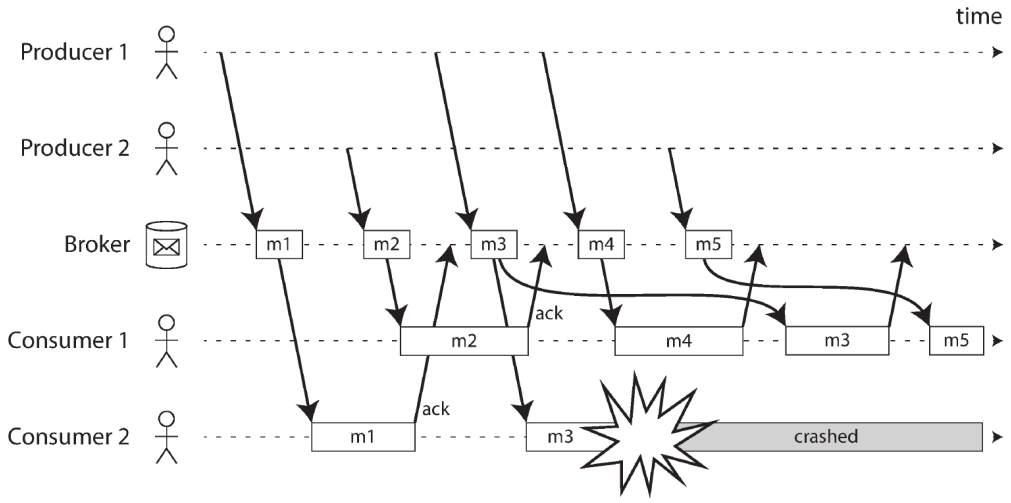


Figure 41: Re-ordering of messages due to consumer failure [10, p. 446]

Problems arise in load balancing if ordering of messages is of importance. In the example of Figure 41 messages are sent before the acknowledgments of a preceding message arrives at the broker due to performance reasons. Consequently, message `m4` is sent to `Consumer 1` before the broker notices that `m3` could not be delivered due to a failure of `Consumer 1`. The broker re-delivers `m3` to the still healthy `Consumer 1` at the price that the ordering of messages on `Consumer 1` now is `m2`

c) Log-based Messaging

One way to implement a message broker is to use **log-based messaging**, which uses **log files** that are distributed. Produced messages are sequentially appended to these log files persistently. Consumers can consume messages by sequentially[30] reading[31] from these files. In particular, new consumers or consumers that just have recovered from a failure can still catch up by reading not just the most recent messages but a certain **offset** before the most recent one.

Although consumers read from the same log files, they do not interfere with each other as data is not destroyed by consumption[32]. This makes log-based messaging is particularly suitable for fan-out of messages. Eventually, even log files are purged regularly, i.e. messages that are smaller than the minimal offset of all partition's consumers can be deleted safely. In rare cases, consumers cannot keep pace with producers, e.g. if they suffer from frequent downtimes. Since messages are appended to files on disk in log-based messaging in a ring buffer, disk space might run out in these cases and data gets lost.

[30] Please note that sequential disk IO has a much higher throughput than random disk IO with many disk seeks and is comparable to random memory access [72, p. 6].

[31] Actually, as consumers are distributed and remote to the log files, they do not access these files directly but consume network messages sent to them which contain log file records.

[32] Eventually, even log files are purged regularly according to some retention strategy. But this is merely an optional housekeeping job and normally affects only those messages that have been consumed long before.

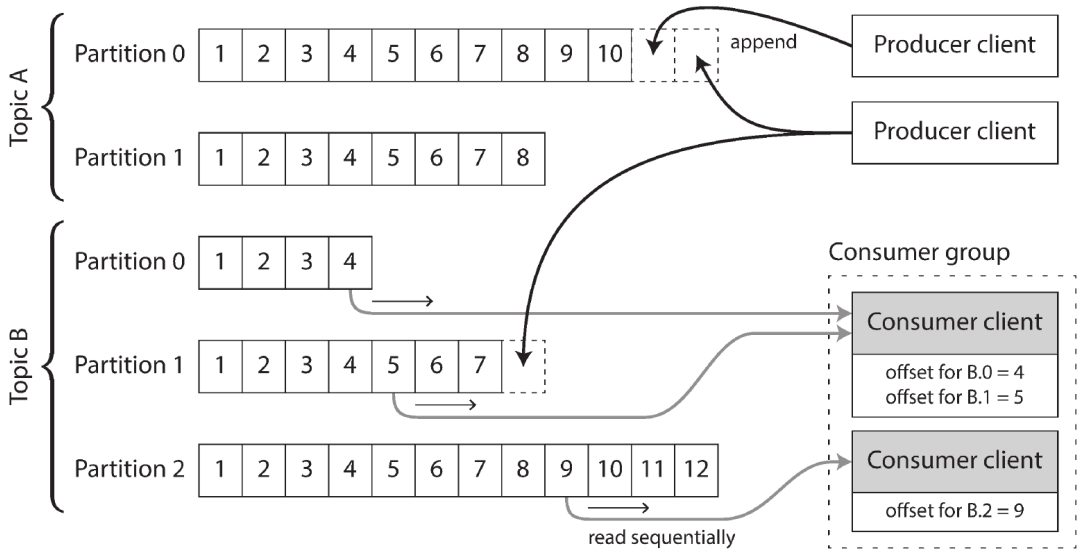


Figure 42: Partitioned logs [10, p. 448]

Load balancing is leveraged by splitting up a single stream (topic) into several **partitions** (cf. Figure 42). These are again just files and are at best distributed among several message brokes. Each partition can be read by several consumers and each consumer can read several partitions. Consumer store the last read message in the current offset.

d) Even Sourcing

Often, messages basically contain **event data**. E.g., if a relevant datum in the sender changes, the sender emits a message that reflects the event (timestamp, original value, actual value, etc.)

ReWo-Rate might emit a message each time a rating is created or updated.

Architectures that extensively use messages containing event data for communication between systems or microservices are also called **event-driven architectures**.

Event sourcing is an architectural pattern for **Domain-driven design** and **Event-driven architectures**. It promotes storing not just the current state of an application but also its entire change history in an **event log**, A single change is reflected i.e. the events that led to changes in the stored entities (data records). Basically, an event is a data record containing timestamps of the event's occurrence and recording, the "before" data value that is supposed to be changed and the "after" data value after the change.

In ReWo-Rate, ratings pose change prone entities. Using event sourcing, changes to ratings can be recorded, audited and replayed if needed.

Of course, a raw stream is unsuitable for certain use cases like complex queries concerning the current application state. However, streams can be used as input for database, e.g. a consumer's job could be transforming events into data manipulation queries to some database. After all, a database binary log (cf. Section 17.9) is some kind of log in a log-based messaging system with the restriction to database (replicas) as producers and consumers.

14.4.3 Streaming Platforms

Streaming platforms like Apache Kafka [31] are modern message brokers with can be characterized according to Section 14.4.1 as follows: They (1) allow for multicasting messages, (2) retain

messages, (3) lets consumers pull messages, (4) store messages on disk, (5) are stream-oriented, (6) allow for processing (called “**stream processing**”), and (7) allow for horizontal scaling. Particularly (7) gave rise to the new term streaming platform that now somewhat supersedes the term message broker.

Java provides foundation classes for handling streams since its inception. Since version 8 streams became a first-class citizen in Java as they suit lambda functions quite well. The following outputs squared values of values that come from a pseudo stream ("pseudo" because of its obvious finiteness).

```
int[] pseudoStream = {1,2,3,4,5};

Stream.of(pseudoStream).forEach(n -> System.out.println(n*n));
```

Instead of being constructed from a finite array, a Java stream object can also wrap inputs from a I/O device (e.g. network).

Due to their possible infinite size, handling streams comes along with some limitations. In general, there is no random access on the entirety of a stream. At least a consumer can read the latest data record on a stream and at best a finite history of recent data records.

14.4.4 Stream Processing Use Cases

If we add processing capabilities (cf. 6 in Section 14.4.1) to stream-oriented messaging (cf. 5 in Section 14.4.1) new use cases emerge.

a) Complex Event Processing

Complex event processing systems (CEP) have persistent queries in place that match on event streams and emit complex events in case of a match. CEPs are used for, e.g. fraud detection in credit card usage event streams, detecting trends in stock markets event streams or detecting (upcoming) machine malfunction of mechanical machines from sensor data.

b) Event Stream Processing

Event stream processors (ESP) enrich, transform, filter and aggregate event data. For example, one could consider incoming emails as a stream. Such emails could be enriched by contact data of the sender thereby making the stream of emails more "valuable". "Real-time" analytics is another common use case: For example, user interactions (clicks) with a website could be considered a stream. An ESP could count the clicks on certain buttons like the checkout button. If this number goes down (due to a buggy release) this can be timely detected by help of an ESP. Post-hoc analytic requests to a data warehouse on day after the event would not be as helpful.

IV. Technology

15 System Resources

Figure 43 depicts a typical architecture view (using a non-standard, non-formal notation) of a typical cloud-based system that realizes a web application. The architecture viewpoint focuses on the deployment. It depicts typical **system resources** of different types: nodes, queues, storages and in-between dataflows. Here, the nodes are captioned by the middleware processes and thus their role in the cluster which they are dedicated for.

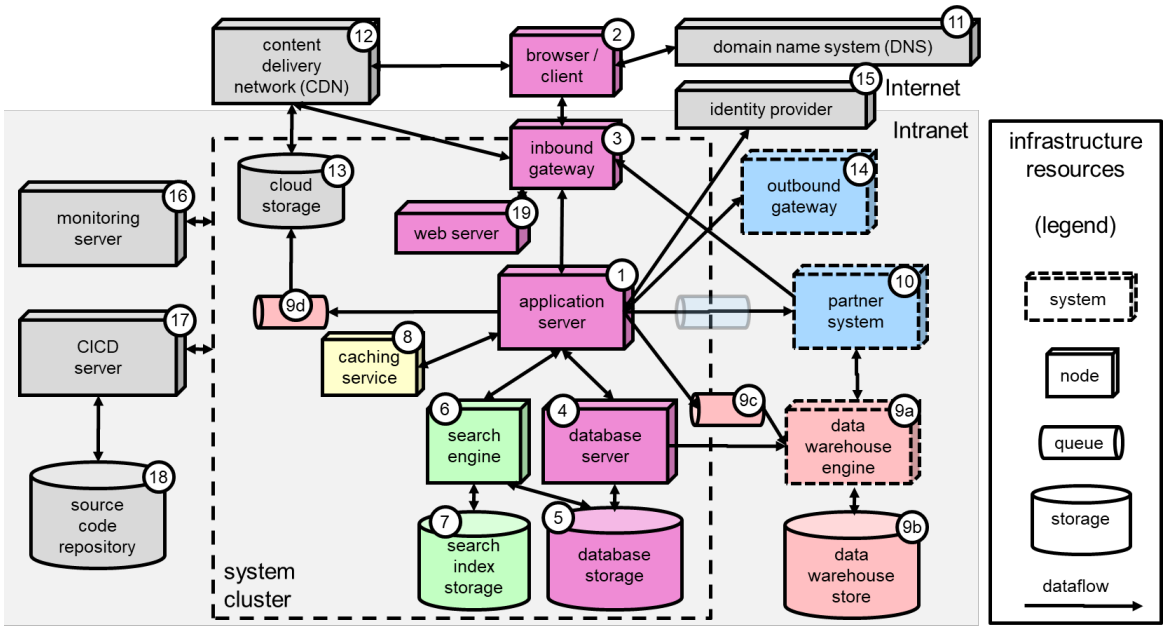


Figure 43: Typical web-based system architecture (after [32])

15.1 Application Server

At the very heart of the system is an **application server**. Typically, the application server receives http requests and computes http responses. These responses typically contain dynamically generated **web resources**, e.g. represented as HTML or JSON documents. So, application servers might serve **web UIs** as well as **web API**. Software artifacts deployed to application servers usually consist of framework artifacts which implement technical boilerplate functionality, e.g. for parsing http requests, accessing databases, rendering web UI by means of HTML etc. and artifacts which have been individually developed and which implement the unique functionality of the respective web application.

15.2 Browser / Client

In a way, the **browser**[33] is the counterpart to the application server. It emits requests to the application server, e.g. by GETting a certain web page or by POSTing form data and renders respective responses. The client might also be another **user agent**, i.e. a client that directly interacts with a user, like a smartphone application. In this case, responses rather contain "raw data" in form of JSON documents than HTML which could be rendered in a browser.

[33] Specifications for the internet and the web (called "request for commons (RFCs)) often talk about **user agents** instead of browsers which is an abstract concept of some client which just makes server-side services accessible to a human user.

15.3 Inbound Gateway

Inbound gateway is an umbrella term for the following system resources that deal with routing and distributing ingress traffic. These are sometimes implemented by a single middleware program or sometimes by multiple programs that run in different processes (maybe on different machines) which are arranged in a network cascade.

Often, inbound gateways are exposed to the internet as opposed to, e.g. application servers which cannot be directly accessed from the internet. In this case, they care for terminating **transport layer security (TLS)** encryption, i.e. for those requests that are directed to URLs with a **https schema**.

15.3.1 Reverse Proxy

Reverse proxies analyse ingress traffic and route it to the destination (load balancer, application server), where it can be further forwarded or processed.

Oftentimes, the ingress traffic consists HTTP requests and the path segment of the request URL (cf. Figure 78) determines where to forward the respective request. This can be used to integrate different web applications each implemented within a dedicated system cluster within a single web site, e.g., behind a single domain name. By doing so, the user experiences a single cohesive web user interface (UI). For example, www.dhl.de works this way.

15.3.2 API Gateways

API gateways are reverse proxies that particularly focus on HTTP requests for web APIs. Besides routing, they typically support

- transformations of requests and response in order to provide a consistent developer experience,
- API oriented, granular authorization
- usage monitoring and
- request rate limiting.

15.3.3 Load Balancer

Figure 43 is a simplification particularly inasmuch as it omits horizontal scaling. Often, there is not just one single but multiple application servers that are identical w.r.t. their deployed software artifacts. A load balancer cares for distributing workload such that all application servers are equally utilized, e.g. by using a round-robin strategy.

If the web application relies on server-side sessions, the load balancer cares for routing requests of a single session to the respective server (**sticky sessions**).

In Figure 44, there are three client-side sessions each on a different machine. The server-side counterparts are identified via a browser-cookie named JSESSIONID which is transferred in every request. The load balancer stores a mapping of JSESSIONID values to destinations, web server 1 to web server N in the example. So

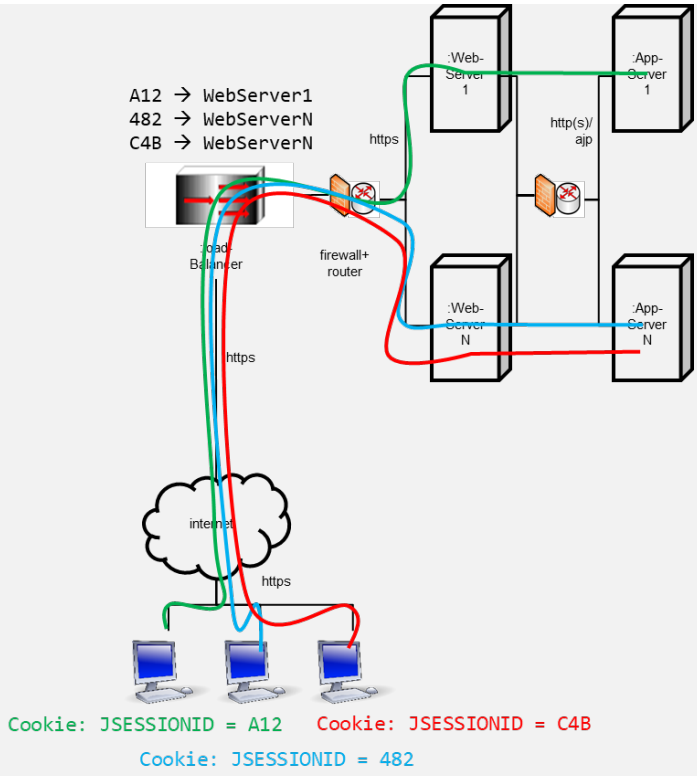


Figure 44: Sticky session example

15.4 Database Server

A **database server** for a database cares for the proper access to that database. A database server usually provides an API that allows for remotely accessing data in a database. Usually, it supports some kind of query and manipulation language, e.g. the **standard query language (SQL)** for relational databases. Statement of these language are parsed by a database server and transformed to low level access operations to the database. The database server also cares for keeping database consistency properties like the ACID properties (cf. Section 8.5) mostly by encapsulating database accesses in so-called transactions. Moreover, a database server does numerous "house-keeping jobs", e.g. it keeps data files and indexes up to date as well as undo and redo logs.

Like application servers, database servers are subject for horizontal scaling which is omitted in Figure 43.

Sometimes, it makes sense to use different types of databases in a single system. For example, **master data** which is not susceptible to changes but need to be consistent is better accommodated in a relational database with ACID properties. Conversely, highly **transactional data** with non-strict consistency constraints should be stored in a NoSQL database with BASE properties.

In ReWo-Rate, master data includes user properties like login names, passwords, contact data etc. but also catalogs of products that are rated. Transactional data mainly consists of ratings but also logs in the sense of Section 9.1.3 which might be stored in a database as well.

15.5 Database Storage

In production environments the actual database data is not stored in the local filesystem of a database server. Instead the **storage**[34] that contains the actual database data is realized by **hard drive disks (HDD)** or **solid-state disks (SDD)** (or just **disks**) which are attached to a database server using low-latency-high-throughput networks of the following types:

- A **network attached storage (NAS)** is a kind of server on its own and exposes a file system which can be mounted as-is by other servers. A NAS uses common ethernet LAN networks and thus is comparably slow.
- A **storage area network (SAN)** connects a disk to a machine such that it can almost be used like an internal disk, i.e. it can and must be formatted with a certain file system and provides a high-performance communication between the machine and the disk.
- An **iSCSI NAS** is somewhat in-between: It uses normal ethernet LAN but represents data on disks like SAN, i.e. in a raw and unformatted way.

	Network Attached Storage (NAS)	**iSCSI Network Attached Storage (iSCSI NAS)**	**Storage Area Network (SAN)**
Data representation	file system	raw block data	raw block data
Network interface	ethernet LAN	ethernet LAN	fibre channel
Typical machines	workstations	servers	servers

Table 13: Technologies for attaching storage to machines

15.6 Search Engines

Full-text search is not among the strength of classical databases: In general, records either match values or value ranges in queries or not.

Therefore, a search engine like Elasticsearch [33] is deployed aside to the database. **Search engines** are special purpose NoSQL databases, which particularly include results that syntactically deviate from the query due to misspelling, inflection or being syntactically totally different but synonym. On the other side, it is oftentimes acceptable if a search index lags behind the indexed data corpus.

So, in modern web applications it is quite common to use a relational database with strong ACID properties along with a NoSQL database with just BASE properties.

15.7 Search Index Storage

The **index** of a search engine is a set of documents that are instances of the same entity type, e.g. "rating" or "customer" and therefore have the same (implicit) schema. Since these sets tend to grow large, search engines and indexes are often distributed to several nodes and storages. A partial index (or in general partial database data) in a certain storage is also called a **shard**. If an index or one of its shards has multiple copies in a cluster, such a copy is also called a **replica**.

[34] Storage is a generic term for resources that allow for persisting data. Besides HDDs and SSDs, storage could also be realized by magnetic tapes, which are cheap but have a huge latency in random accesses due to long seek times, SD cards, USB sticks, DVD RW disks etc. However, these devices are not suitable for accommodating database files.

15.8 Caching Service

A **caching service** is used to cache intermediate results of a web applications in order to reduce processing time. These intermediate results are typically result sets of a database query or partial HTML renderings. Such caching services are typically implemented using a key-value in-memory store, which is some kind of NoSQL database.

Another typical use case is to cache session data in a caching service in order to keep such session data out of the memory of single server-side processes (cf. Section 21.6.2).

Common key-value in-memory stores for implementing a caching service are Memcached [34] and Redis [35].

15.9 Data Warehouse

Data warehouses consolidate mostly transactional data from different databases in order to provide a holistic view on business data for analytical purposes. This consolidation takes place in a repeated (e.g. once per night) so-called **ETL process**, which stands for extraction of data from different databases, transformation of that data to the uniform schema of the data warehouse and loading that data into the data warehouse database storage.

With the rise of big data, it is got fashionable to not just analyse transactional data but to track events at a very detailed level, e.g. to track the navigation (clicks or even mouse movements) of the users on a website. These event data are pushed to streaming platform servers (cf. Section 14.4) and stored in data warehouses.

15.10 Partner Systems

In system landscapes, systems are integrated with each other. Integration mostly works by APIs that are exposed by the application servers of the respective systems (via a load balancer). Technically, integration is implemented by synchronous communication (cf. Section 14.3) or asynchronous messaging (cf. Section 14.4) between the systems using a classical message broker like Apache ActiveMQ [36].

15.11 Domain Name System (DNS)

The internet's **domain name system (DNS)** provides a dynamic mapping of mnemonic addresses to IP addresses. So, a domain name is easier to memorize for a human than the IP address it maps to. Moreover, the mapping to IP addresses is dynamic, i.e. domain names for web applications can be kept stable while the IP addresses might change.

A **domain name** is a sequence of **labels** separated by periods. A **domain** is a set of domain names that share a certain suffix. The **top-level domain** `de` contains all domain names that end with `de`; within that domain, the **second-level domain** `swmgmt.de` contains all domain names, which end with `swmgmt.de` and the **third-level domain** `parcer.swmgmt.de` contains all domain names ending with `parcer.swmgmt.de` and so forth. `swmgmt.de` is also called a **subdomain** of `de` and `parcer.swmgmt.de` a subdomain of `swmgmt.de`.

Domain names that map to IP addresses are also called **hostnames**. To be exact, the leftmost label in such domain names is the hostname; the complete domain name is called **fully-qualified domain name (FQDN)**.

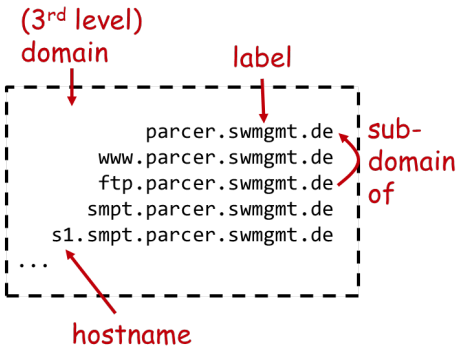


Figure 45: Domain names

The **domain name system (DNS)** is a hierarchical network whose nodes are **name servers**. Each name server is equipped with a **zone file** which contains a number of **resource records**. Each resource record is a mapping of a domain name to

- another name server (NS record),
- an IP address (A record for IPV4 or AAA record for IPV6),
- another (alias) domain name (CNAME record)

just to name the most important among all resource record types [37]. A **zone** is the set of all resource records with domain names of a certain domain.

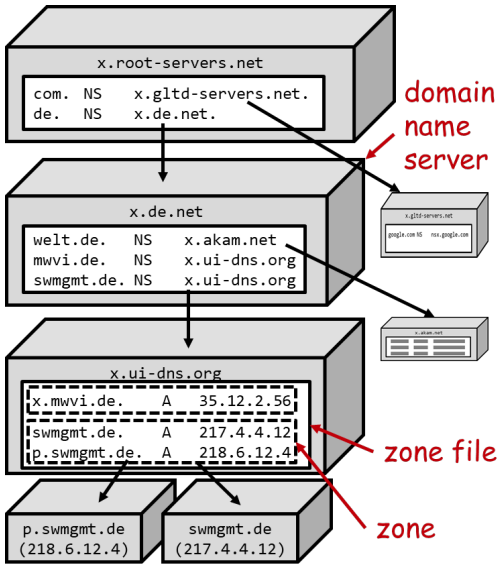


Figure 46: Hierarchy of domain servers

The hierarchy of domain servers is established by NS records.

> In the example of Figure 46, the domain server with the domain name `x.de.net` is authoritative for the zone `de`. However, its zone file itself mainly contains NS records for second level domains. For example, the server `x.ui-dns.org` is authoritative for the second-level domain `swmgmt.de`. The zone file finally contains an A resource record for the host `p.swmgmt.de`.

Though **authoritative name servers** are considered to be the single source of truth regarding the resource records of their zones, DNS makes extensive use of caching. Each client normally has an internal DNS cache with resource records of former queries from that client. In case of a cache miss,

the client ask a nearby domain name server. If the resource record still cannot be found, root domain servers get queried which delegate the query to the appropriate authoritative domain server of the respective top-level domain. (There are just 13 hostnames for root domain name servers starting with `a.root-servers.net` and ending with `m.root-servers.net`, each of which with single IPV4 and IPV6 address. However, the query workload is actually distributed to many more servers by use of **IP anycast**.)

15.12 Content Delivery Network (CDN)

Modern web pages consist of numerous **web resources** like HTML, images, videos, cascading style sheets and Javascripts. While HTML is often dynamically generated per request, the other resources are rather static, i.e. change just in the course of a deployment. **Web caching** [38] is a crucial concept in the architecture of the web. Of course, latency is smaller if a cache is close to the respective client. **Private local caches** (in browsers) with low latencies contain just those responses, i.e. web resources, that have already been requested by the particular client. On the other side, **origin servers**, which are often application servers (cf. Section 15.1), are far away from a client and can respond just with high latencies. A compromise are **shared caches,** which are servers that receive requests before origin servers and serve matching responses in case of a cache hit.

A **content delivery network (CDN)** mainly consist of shared caches. These shared caches are distributed over the world. Requests from clients (browsers) for static web resources are routed to the closest shared cache in a content delivery network mainly in order to keep latency low. CDNs are an example of **edge computing,** which is a paradigm for building distributed systems where certain system resources are moved closer to the clients.

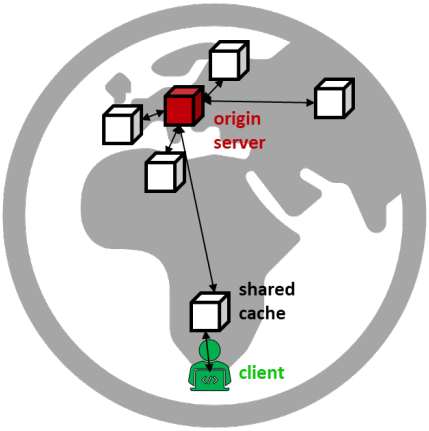


Figure 47: Content delivery network

Before the rise of content delivery networks, it was common to have web servers running Apache HTTPD or Nginx in every systems cluster. These web servers relieved application servers from serving static web resources. A typical request from a client went through three different kinds of servers: web server, app server and database server, hence the name **3-tier architecture**.

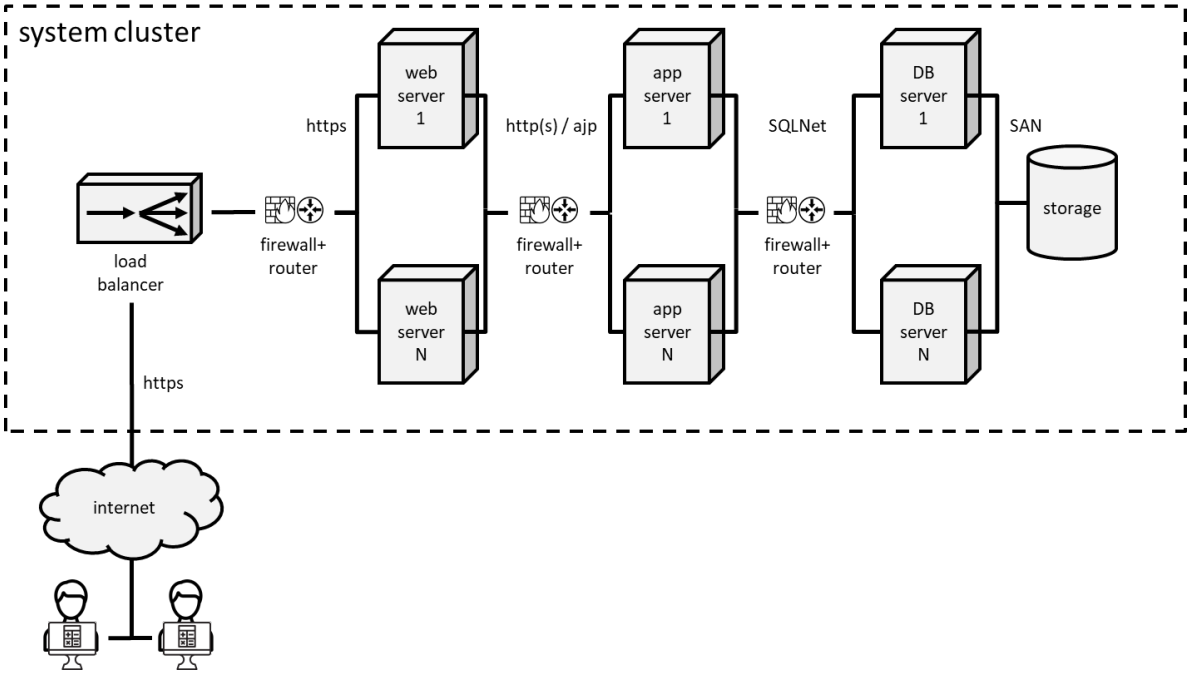


Figure 48: Classical 3-tier architecture

15.13 Cloud Storage

Data can be persisted in many ways, i.e. stored in some kind of **storage**. The first question to ask is about the data model from a fundamental point of view:

- Certain data is stored best as **raw block data**, i.e. with no data model at all. Specific queries to these data are of course quite hard if not impossible.
- Data with no or no common structure is best stored in a **file system** accessible through functions of an operation system. Specific queries end at the file level as data models of file contents might vary from file to file.
- Structured data with a common explicit or implicit schema should be accommodated in a **database** and is accessible by means of the respective database management system. Specific queries can target even single records or values in such data.

In cloud environments, raw block data or file systems are stored on unformatted or formatted persistent **disks**, respectively. Disks are in a way local to the machine that mount them, i.e. are in the same data center.

Cloud storage is a kind of storage that is exclusively offered by cloud providers, whereas disks and databases are commodity hard- and software that need not to be operated in a cloud at all. Cloud storage is a compromise of a file system on a disk and a database. It offers a simple **data model** that resembles that of a file system. Data are organized in **objects** similar to files that have unique names similar to paths + filenames in a filesystem and live in **buckets** similar to a single file system.

Buckets cannot be directly mounted into the filesystem of a machine. Instead cloud storages expose web APIs by which objects can be created, read, updated and deleted (CRUD operations) in a bucket.

Each bucket gets a storage class on creation. The **cloud storage class** mainly determines

- the guaranteed availability of the bucket (ranging from 99,9% to more than 99,99%),

- if data is replicated multi-regionally,
- the cost for storing data for a certain time, which is in the order of magnitude of 0.01$ per gigabyte and month, and
- the cost for transferring the data from and to the bucket.

Storage classes that provide high availability, multi-regional replication and low transfer costs are suitable for data, which are frequently accessed. Buckets of that storage class might back a CDN. A storage class with low storing costs is suitable for uses cases like backups or archives, since these data are not accessed frequently.

15.14 Outbound Gateway

Without an outbound gateway, a system can just react to incoming requests, i.e. communication from a user to the system. However, certain use cases require proactive communication from the system to the user. Here, an **outbound gateway** comes into play, which is just an umbrella term for diverse types of gateways.

The most common outbound gateways are **SMTP servers,** which can send emails generated by the system to the user, e.g. in order to verify an email address. Other types of outbound gateways include **SMS**, **WhatsApp** (for, e.g., **two-factor logins**) or servers for smartphone or browser **push notifications**.

15.15 Identity Provider

From a security perspective, a software system is a system of **entities** which might be end users or clients. These entities might possess (digital) **identities**, which are basically sets of properties (key-value pairs) with elements like username, email address, etc.

In order to protect confidential data, each system backing a certain web application must perform

- **identification**, i.e., recognizing an identity by some criterium,
- **authentication**, i.e., implement a process by which an entity can proof to possess an identity as well as
- **authorization**, i.e., determining which entity is permitted for which actions, e.g., accessing which web resource.

In software systems, the common criterium for identification is non-confidential string, e.g., a username, which might be identical to an email address of that user.

Authentication can be implemented in various ways. It is still common that, in order to authenticate with a system, the user provides a secret string called password that is hashed and then compared by that system.

Oftentimes, authentication is just a means for authorization. As soon as the user is authenticated, he or she is authorized for actions that are, e.g., configured for the particular identity. Please note that authentication is not necessarily a precursor for authorization. For example, a bus ticket authorizes a passenger to enter a bus but does not identify let alone authenticate him or her. So, generally, one cannot use authorization as a precursor for authentication.

In the past, it was common that every user had a different identity for each web application and this identity (after authentication) was merely used to authorize the user for certain actions. Nowadays in order to avoid user password fatigue and enforcing compliance requirements (password strength,

renewal periods, highly secure credential storage, etc.) authentication often is delegated to and centralized in an identity provider. An **identity provider** is a system that performs **authentication** of a user on behalf of another system. For web applications this is typically done by a temporary redirect of the browser from the requesting web application to the identity provider, which in turn issues some kind of **token**, to the requesting web application. The token contains data that identifies as well as authenticates the user. A single identity provider usually performs authentication for multiple systems which then are called to be members of a **federated identity management**.

Besides authentication, identity providers can be used to **authorized** some web application A to access non-public web resources by some user stored in (the database of) some web application B. This is particularly useful for a user's master data, e.g., contact information (address, email address, phone number) which is otherwise cumbersome to keep current and consistent over multiple web application.

In recent years, several standards have emerged in the field of identity management. **OAuth 2.0** [39] is an authorization framework that often relies on exchanging (browser cookie) tokens following the **JSON Web Token (JWT)** [40] standard. **OpenID Connect (OIDC)** [41] standardizes the way authentication should be done using OAuth 2.0 and JWT.

Identity providers can be part of a single sign-on implementation in a company's intranet but also be used publicly in the internet. Big players like Facebook, Microsoft, Google, and GitHub are the most common identity providers in the internet surfacing by the well-known "Login with X" button.

15.16 Monitoring Server

A monitoring server gathers, consolidates and visualizes metrics, traces and logs of one or more systems clusters.

15.17 CICD Server

A CICD server builds deployable software artifacts, tests them using automated tests and deploys them to the respective machines.

15.18 Source Code Repository

A source code repository contains different versions of file trees, where the files are typically the constituents of a certain program including auxiliary resource files (configuration files, media files etc.)

15.19 Web Server

As stated in Section 15.12, before the advent of edge computing and CDNs, static web resources, e.g., media files, used to be deployed to web servers that were exclusively assigned to a certain system.

Since modern web applications heavily rely on JavaScript and frameworks like ReactJS [42] web servers more and more fulfil other roles like transpiling TypeScript [43] or Sass, bundling or generating (server-side rendering) web resources (modularized JavaScript, CSS, HTML etc.) normally based on NodeJS [44].

16 Cloud Computing

In Chapter IV, we have described system resources that are common for a cloud-based web application. In this chapter, we explain what cloud computing is all about and how to actually build a system cluster with (some of) these infrastructure components in the Google Cloud.

16.1 Cloud Characteristics

[45, p. 2] gives the most commonly accepted definition of cloud computing.

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.

Essential Characteristics:

On-demand self-service. *A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.*

Broad network access. *Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).*

Resource pooling. *The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth.*

Rapid elasticity. *Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.*

Measured service. *Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service."*

16.2 Service Models

Services offered by cloud providers can be categorized according to the degree of prefabrication and specialization for certain customer needs. There is a whole spectrum of service models bounded by **Software-as-a-service (SaaS)** at the upper end, which provides read-to-use, multi-tenant web applications with certain functionality. At the lower end of the spectrum, **Infrastructure-as-a-service (IaaS)** essentially provides storage, network, and virtual machines with installed operating systems and leaves much flexibility for the customer to operate arbitrary software on them. The other models are somewhat in-between as they provide preinstalled runtime environment, middleware or container services which allow for a faster and more convenient deployment than IaaS.

A classic, commonly accepted definition can be found at [45, pp. 2-3]:

*"***Software as a Service (SaaS)***. The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.*

Platform as a Service (PaaS)*. The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.*

Infrastructure as a Service (IaaS)*. The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls)."*

Please note that there are cloud providers like Heroku [46] that are "second-level providers" as they do not possess own locations and hardware but provide PaaS or particularly SaaS offerings based on IaaS capabilities of "first-level providers" like AWS (in the case of Heroku).

In the last years, even more nuanced service models than those above have emerged. Figure 49 attempts to delineate these from a technological point of view, where upper filled boxes represent resources that run and/or depend on the lower ones. The bars on the right side indicate, what is managed by the cloud provider and what is left to manage by the customer.

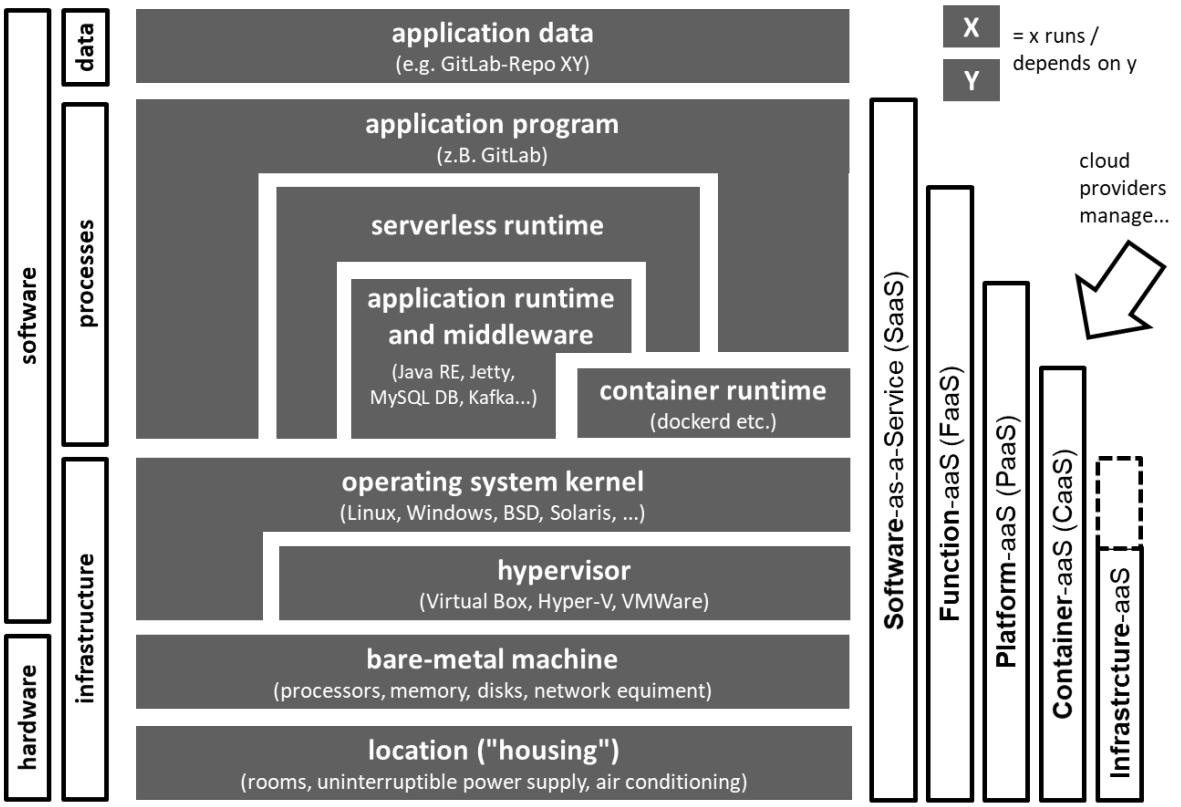


Figure 49: Cloud service models

Some cloud providers—particularly the big ones like Amazon, Microsoft, and Google—offer more than one service model. Since these offerings are of course integrated, it is sometimes hard to tell the service models apart.

The Google App Engine [47] falls in the category of a Function-as-a-Service while the Google Compute Engine [48] in its core is an Infrastructure-as-a-Service service model.

16.2.1 Infrastructure-as-a-Service (IaaS)

The bottom-most resource is the mere **locations**, i.e., **housing** for hardware. There are providers which provide just this, e.g., computer-friendly space that can be rented in order to accommodate own hardware there.

Bare metal machine denotes the actual physical hardware computers are made of. Such machines either run an operating system kernel or a **type-1-hypervisor** (cf. Section 18.1.3).

Sometimes, Infrastructure-as-a-Service (IaaS) even includes preinstalled operating systems. Nowadays, operating system distributions largely consist of optional programs that run in user space like editors, video players etc. The upper resources mainly rely on the core of the operating system, i.e., the **operating system kernel**.

16.2.2 Container-as-a-Service (CaaS)

A **Container-as-a-Service** service model offers preinstalled **container (orchestration) runtimes** like Docker (cf. Section 19.3) and/or Kubernetes (cf. Chapter 20). They can run arbitrary applications as long as they are containerized, i.e., are packed in a (set of) container image(s) and can run within containers.

16.2.3 Platform-as-a-Service (PaaS)

Platform-as-a-Service even more mostly relieves the customer from tedious, repetitive and error-prone installations and configuration of common middleware resources like Java runtime environments, message queues, databases and the like. Instead, such middleware resources can typically be provisioned by just a single (or just a few) clicks or commands.

16.2.4 Function-as-a-Service (FaaS)

In PaaS, the customer is still in charge to build, package, deploy, start and partly out-scaling application programs in a suitable way (by means of containers or without them).

Function-as-a-Service (synonymously: **serverless computing**[35]) automates these steps and allows for updating application programs right from the source code. Process recreation, i.e. stopping and restarting application programs on the same or different machine due to failover or horizontal scaling, is completely controlled by a **serverless runtime**. Therefore, developers have to cope with two important architectural constraints in FaaS:

8. During restart, the process virtual memory is lost (i.e, at least the data, heap and stack sections). Thus, server-side sessions (cf. Section 21.6.2) are effectively unusable in FaaS.
9. Processes may be stopped and not restarted until there is demand, e.g., a certain request arrives which needs to be processed by the respective process. Therefore, spin-up time (cf. Section 7.3.1) may increase the overall response time significantly.

These constraints render FaaS useful for workloads where (1) no state at all has needs to be stored between requests or can be efficiently offloaded to the client or a backing database and (2) processing time dominates spin-up time anyway.

> A function that takes an video input stream as input and produces a, e.g., transcoded one as output would be a perfect match for FaaS. In this case, no state needs to be stored since transcoding one video does not affect another one and processing time would be in the minutes or hours and therefore, say, two seconds spin-up time would not matter much.

16.2.5 Software-as-a-Service

In SaaS, customers get a fully functional application. The only thing that customers still have to do, is using this application and producing (transaction) application data.

> GitLab.com is a perfect example for a Software-as-a-Service.

16.3 Deployment Models

Cloud **deployment models** define where the cloud "resides" relative to a customer and a cloud provider. [45, p. 3] define the following delivery models:

"**Private cloud**. *The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises.*

Community cloud. *The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of*

[35] Serverless computing is a misnomer since it suggests that there are no servers in place.

the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.

Public cloud. *The cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider.*

Hybrid cloud. *The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds).*"

Beyond these definitions, the **multi-clouds** or **inter-clouds** became popular in the last years. These clouds distribute systems of a customer to public clouds of different cloud providers, mainly in order to avoid dependencies from and lock-ins to a single cloud provider.

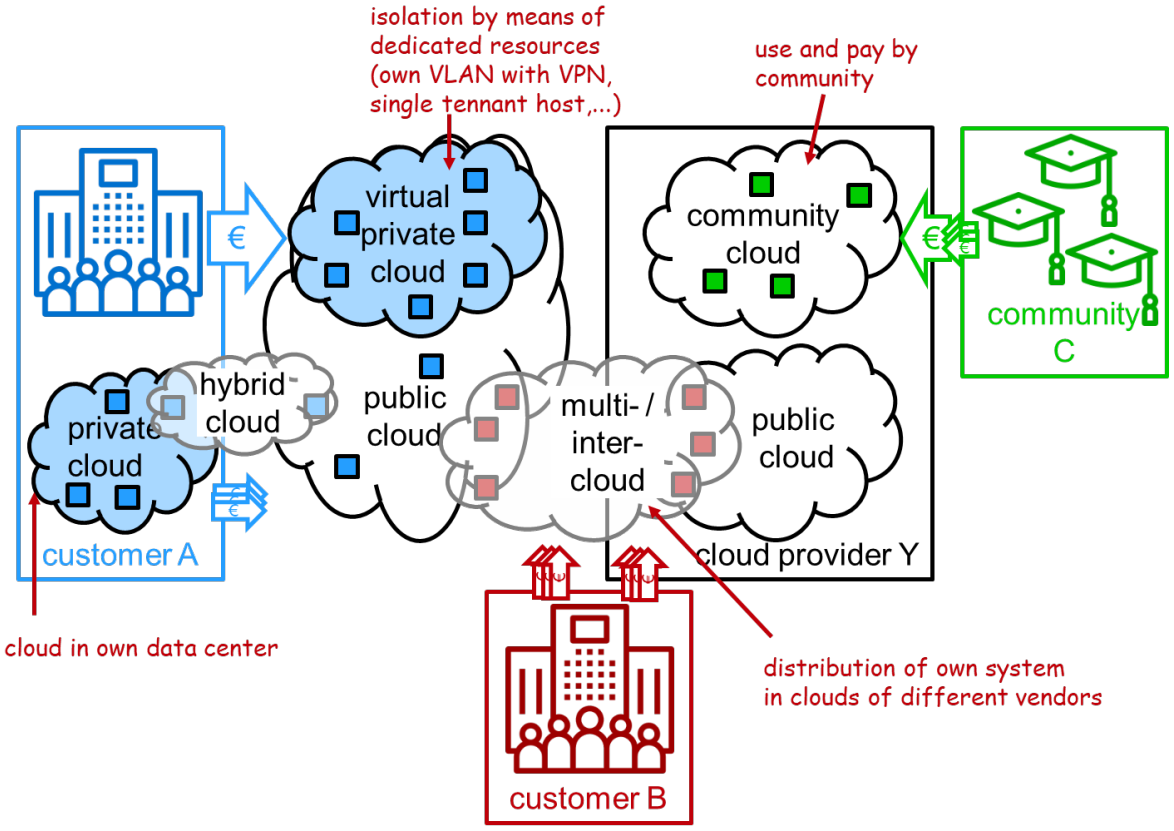


Figure 50: Cloud deployment models

16.4 Utilization and Costs

Resource pooling on large scale hardware resources leads to a better utilization of these resources. The assignment of demands for resources to actual resources often follows some bin packing algorithm [49]. Normally, assigning comparably large resource demands to comparably small resources leaves much unused "daylight" in these resources, i.e. undersaturates them or does not use their full capacity. In the visualization of Figure 51, the resource on the left side is just used by $5/8$. Therefore,

a request for $1/2$ of that resource cannot be satisfied any more. The fraction of unused resources is lower if demands are comparably small as the right side of Figure 51 shows.

From a cost perspective, it is in the very interest of cloud providers to keep the utilization of hardware resources close to 100% in order avoid waste. This can only be achieved by serving demands of multiple customers because of the binpacking effects and since variations in demands of customers tend to compensate each other.

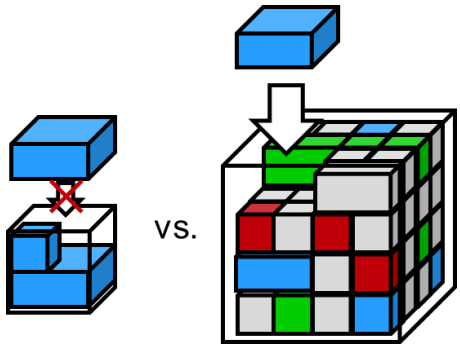


Figure 51: Binpacking in small and large resources

From a customer's point of view, rapid elasticity is attractive because it contributes to the ideal cost-effective scalable system of Figure 16. Self-hosting leads to over- and underutilized systems which in turn lead to poor performance thus customer's customer dissatisfaction and waste of money, respectively. Figure 52 depicts both cloud-hosting and self-hosting from a cost perspective.

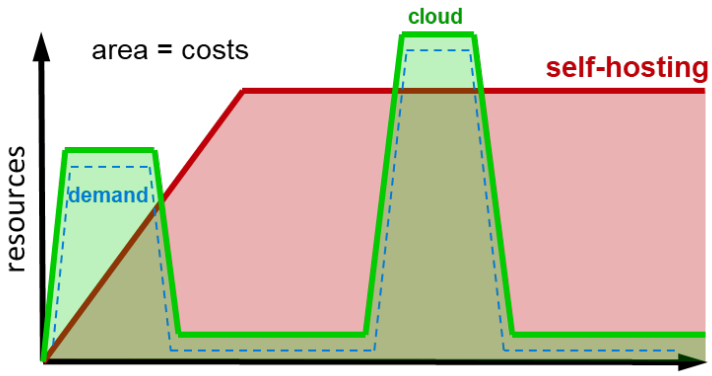


Figure 52: Cloud-hosting vs. self-hosting

Sometimes, cloud-hosting is also attractive for stakeholders of the financial department. Expenditures for self-hosting are **capital expenditures (CAPEX)**, those for cloud-hosting are **operational expenditures (OPEX)**.

16.5 Cloud Services

The scope of functionality, which cloud providers offer, is overwhelming. Therefore, it is not trivial to derive a common categorization for this functionality that suits offerings of every cloud provider. At the bottom level, we group coherent functionality into **cloud services**, a.k.a. **managed services**.

"Virtual Server" is a common cloud service that every cloud provider offers yet with its own branding. In the Google Cloud the respective cloud service is called "VM instances".

Since big cloud providers even offer more than hundred different cloud services, we can further group them into **cloud service categories**. One ambitious attempt to map the provider-specific cloud services to a common cloud service name and to further group them into cloud service categories can be found in [50]. In Table 14, we just provide a provider-agnostic and condensed overview of cloud service categories.

Cloud service category	Description
compute	Compute services provide provisioning of dedicated, single tenant, bare-metal machines, as well as virtual machines, or containers together with their orchestration. Even higher on the PaaS spectrum, there are services that allow for deploying right from a source code repository and so-called "lambda functions" that are deployed just for single requests.
storage	Cloud storage options and their trade-offs have been discussed in Section 15.13.
database	Database services are preinstalled multi-tenant databases or automatically installed single-tenant databases with different data models, i.e. relational databases, NoSQL databases etc.
networking	Networking services include the provision of content delivery networks (cf. Section 15.12), of permanent public IP addresses, firewalls, load balancing, virtual private clouds and virtual private networks.
developing tools	Developing tools include repositories for versioning source code, build artifacts and containers, build automation (continuous delivery pipelines), software development kits, and plugins for popular IDEs
administration & monitoring	Cloud services can usually be configured via command line tools, via a web UI or programmatically via a web API. Monitoring services include tools for metrics, logging, and tracing (cf. Section 9.1).
security	Security services include the automated provisioning of X.509 certificates (for secure communication via https), key management, centralized identity management (IAM)
analytics	Analytics services include services for storing, transferring, converting and analysing huge amounts of (event) data. Data warehouses (cf. Section 15.9) are also included in this category.
artificial intelligence	Cloud computing provides potentially vast computing power, which lead to another spring in the field of artificial intelligence [51]. Cloud AI services typically support special AI use cases like language processing and translation, speech and image recognition, or machine learning.
internet of things	Although "things" in the internet of things are rather on the edge of a system and not in the cloud, certain services like message brokers in the cloud can also support IoT use cases.
misc. services	Some services are versatile, support very specific use cases or contribute to diverse categories from above. So, they are not included in one of the above categories. Message brokers (cf. Section 14.4**Fehler! Verweisquelle konnte nicht gefunden werden.**) are among these as well as, e.g. email services, notification services, chatbots, blockchains, media transcoders or API management services.

Table 14: Cloud service categories

17 Google Cloud

This chapter provides an overview about the most important concepts and cloud services of the **Google Cloud**. We are particularly interested in the IaaS/PaaS offering **Google Cloud Platform (GCP)**. The SaaS offering **G Suite** is not in the scope of this course nor this lecture notes. Moreover, we just describe basic concepts and services of the Google Cloud. Please refer to the official documentation [52] for further information or to the GCP tutorials [53] for further training.

17.1 Resources

Resources of a cloud-based system are not directly accessible by customers. Instead, each resource is configured through a data object. In the following, we use the term **resource** homonymously for an actual resource and its backing data object. A resource is typically an instance of some cloud service.

In GCP, one can create a virtual machine using the cloud service "Compute Engine".

17.2 Projects

In GCP, each resource is assigned to exactly one **project**, i.e. a project is a logical aggregation of resources. Normally, a project corresponds to a single system cluster.

Projects are identified by

- a **project name** that need not be unique and thus can be arbitrarily chosen,
- a **project id**, whose suffix is generated in order to make the project id globally unique,
- a **project number** which is completely generated and globally unique.

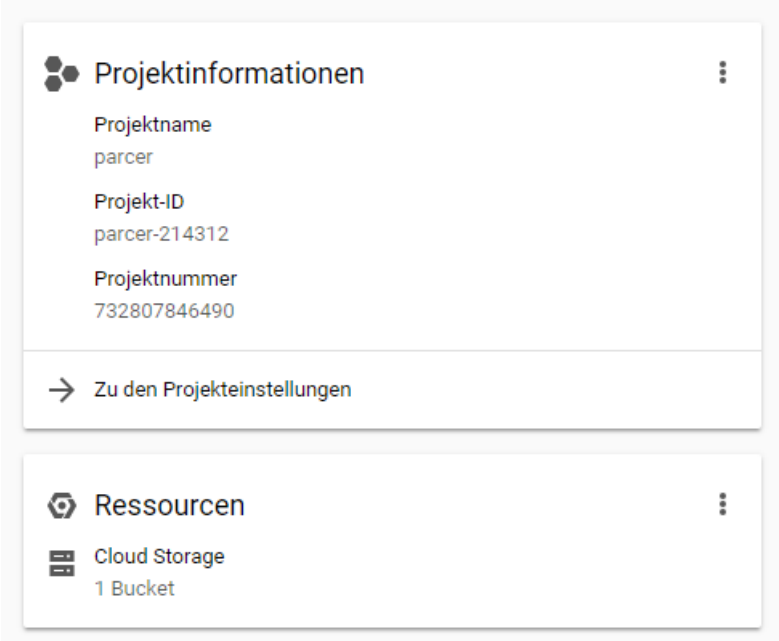


Figure 53: Project with its identifiers and a single resource

17.3 Administration

Cloud customers can access their projects and resources via RESTful web APIs (cf. Section 21.6), which are exposed by the Google Cloud Platform. On the client side, there are three ways to access these APIs:

- via the command line tool **gcloud**
- via **client libraries** for diverse programming languages
- via a web UI called **Cloud Console**, accessible via `https://console.cloud.google.com` which particularly provides a browser-based terminal called **Cloud Shell** with a preinstalled `gcloud`

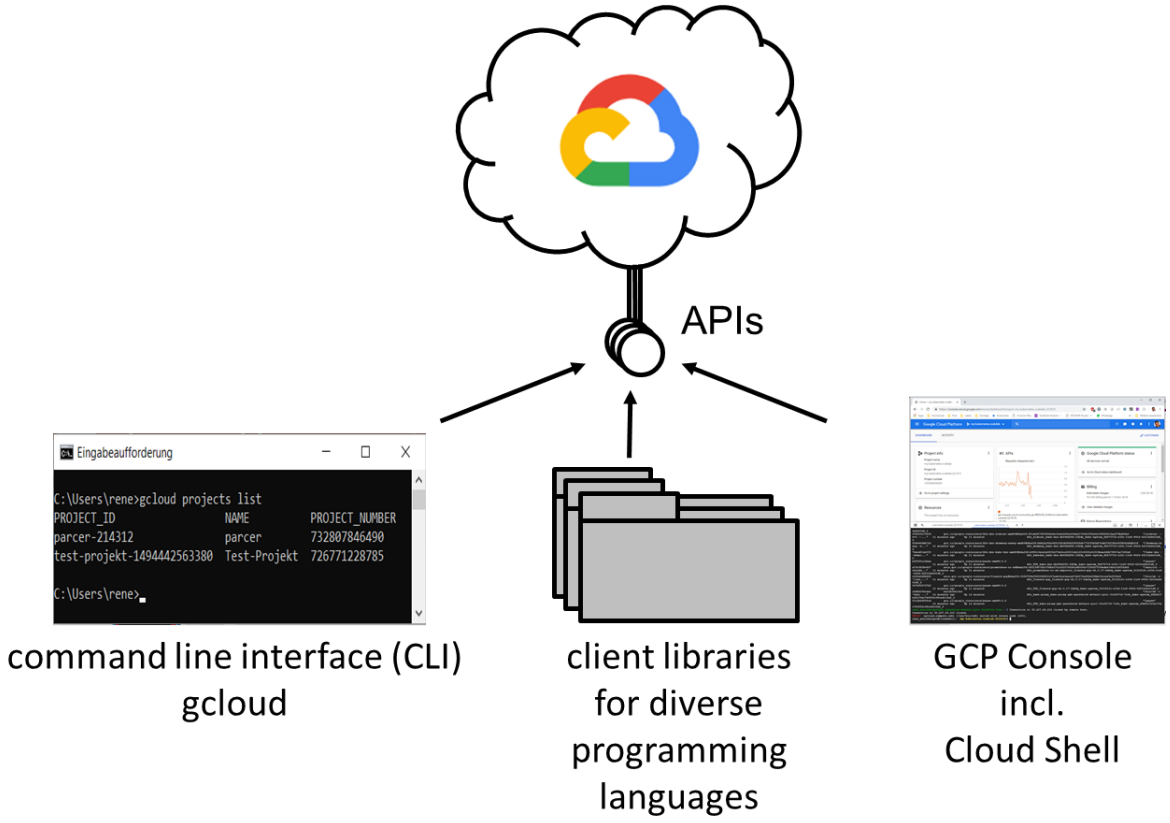


Figure 54: Ways to access Google Cloud Platform's web APIs

In the following, we mostly make use of gcloud, which is documented in [54]. Since gcloud exposes all services and options of the Google Cloud Platform to the command line, it has many commands. Therefore, commands are organized in a hierarchy. A command line typically looks like this

```
glcoud <command_group> <command> <subcommand>
```

Figure 55 depicts a small fraction of the command hierarchy.

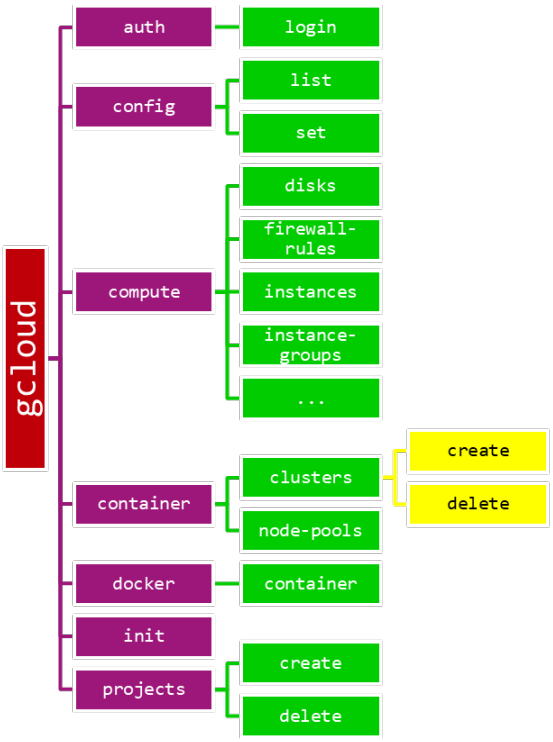


Figure 55: gcloud command hierarchy

17.4 Regions and Zones

After all, each actual resource is located in some data center. In GCP data centers are called **zones**. Different zones might have slightly different equipment, i.e. just some zones provide single-tenant hosts, GPUs, local SSDs etc. `https://goo.gl/maps/Ew2YADyoKoq` shows a street view from the inside of a GCP data center.

Zones are geographically grouped into **regions**. Roughly speaking, a region corresponds to a city area like Frankfurt am Main and contains about three zones. At the time of writing, GCP has 18 regions worldwide.

Regions have a unique name like `europe-west3` for Frankfurt. A zone has the name of its region suffixed by a letter, like `europe-west3-a`.

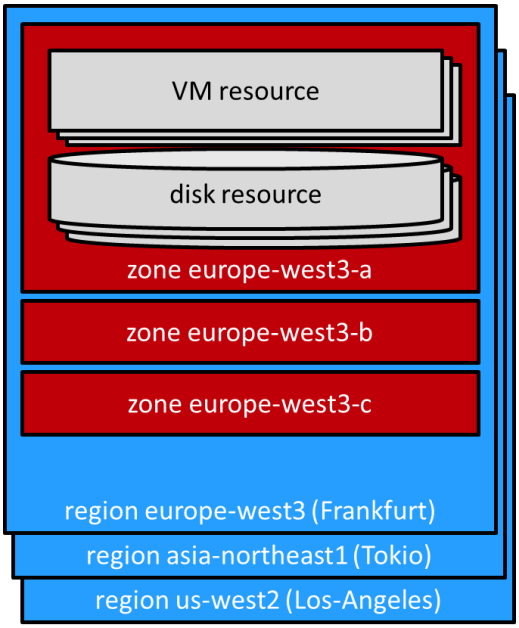


Figure 56: Regions and zones

17.5 Pricing

Two factors underly pricing of resources: (1) The quality of a resource, e.g. number of CPU cores and RAM for a virtual machine and (2) the actual use of a resource, which is measured by cloud providers (cf. Section 16.1). The "use" is measured in usage duration and/or used capacity or throughput depending on the used resource.

Price calculations are very complicated. In order to give a rough impression: As by October 2018, a virtual machine with 4 vCPUs, 15 GB RAM, and a 375 GB SSD costs about 100 $ per month. Other pricing benchmarks are as follows:

- network
  - 0,10 $ / GB for region ex-/inbound traffic
  - 0,01 $ / GB for interzonal traffic within region
  - 0,06 $ / h / VPN tunnel
- 1 vCPU
  - 20 $ / month
  - 13 $ / month (fixed for 3 years)
  - 6 $ / month for pre-emptive VMs
- RAM
  - 3 $ / GB / month
- disks
  - 0,05 $ / GB / month (default HDD)
  - 0,30 $ / GB / month (SSD in particular region)

17.6 Billing

A project must be linked with a **billing account** before resources can be created within it. Resource costs are accumulated in a project and transferred from the linked billing account in certain intervals. Billing accounts can be saved by unlinking projects. However, this eventually leads to the deletion of resources within the respective project after some time and is recommended just for test projects.

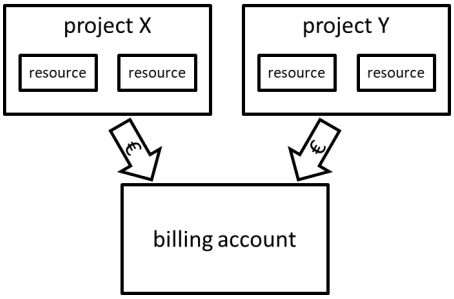


Figure 57: Resources, projects and billing accounts

17.7 Storage

As described in Section 15.13, storage is a hypernym for many kinds of storages. Figure 58 depicts a kind of decision tree for GCP storage options with common use cases in the leaves.

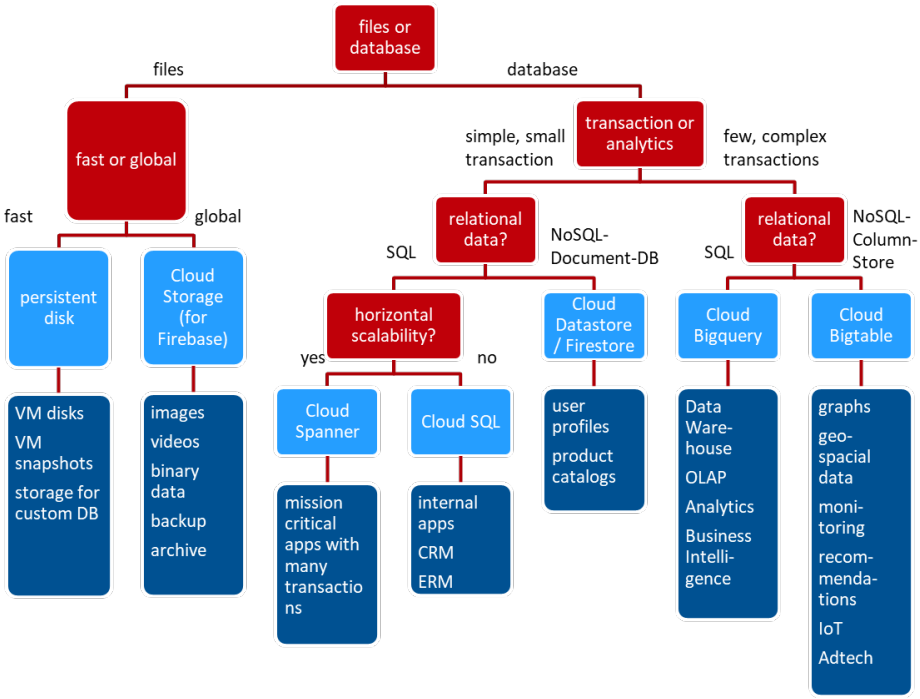


Figure 58: GCP storage options

The left-hand side of Figure 58 depicts file-based storages, where, for the sake of simplicity, files and objects in buckets are treated to be the same. Table 15 characterizes both file-base storage options, namely persistent disks and cloud storage.

	Persistent disk	**Cloud storage**
Access	- VMs must be in same zone/region - read-write just by single VM - via OS calls	- global access - read-write just by multiple VMs - via REST-API
Performance	comparably fast	comparably slow
Maximum size	10 terabytes	multiple petabytes
Data model	- block-oriented (unformatted) - file-oriented (if formatted) - no versioning	- objects in "buckets" - versioning is possible
Use cases	- file server - storage for (No)SQL data-bases	- content distribution - backup, archive - data analytics (Map-Reduce)
Classes	- HDD / SSD / RAM - regional / zonal / local	- multi-regional (e.g. web content) - regional (e.g. data analytics) - nearline (e.g. backup) - coldline (e.g. archive)

Table 15: Persistent disks vs. Cloud Storage

17.8 A GCP-based System Cluster

Figure 59 depicts a UML deployment diagram for a typical system cluster in GCP.

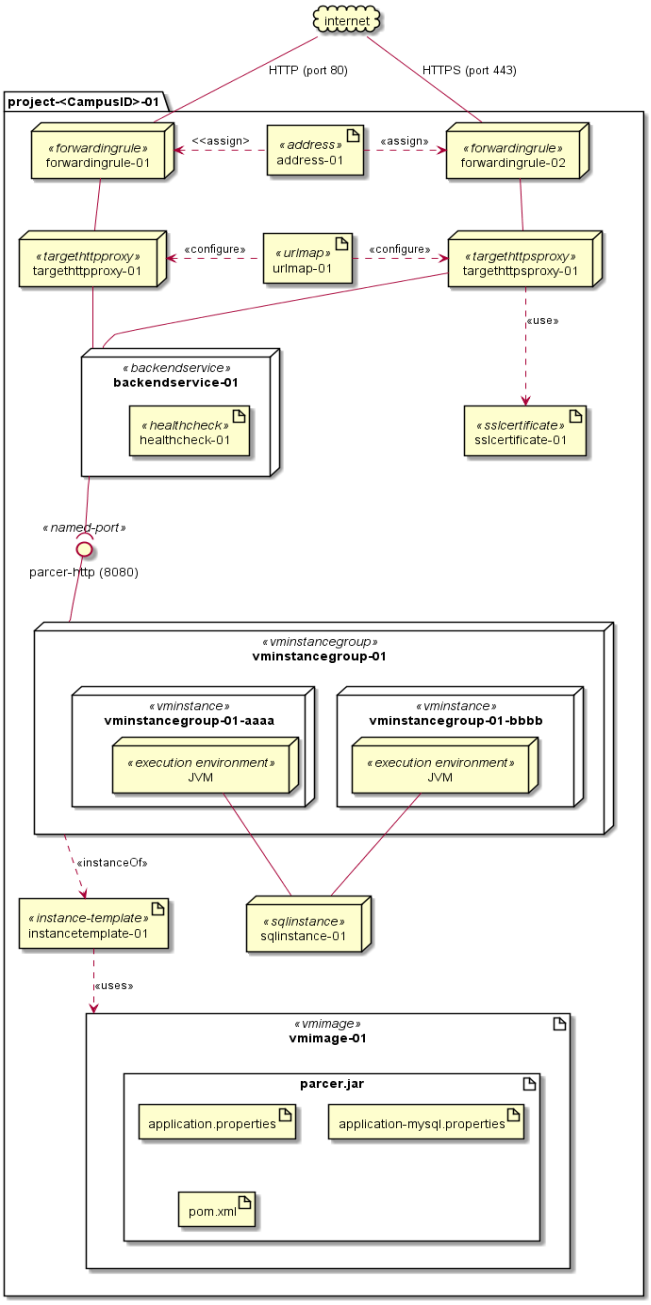


Figure 59: System cluster in GCP

17.8.1 Virtual Machine Instances

The cluster comprises the following resources (from bottom to top): A **VM image** is a disk image, i.e. a zipped boot disk for a VM instance. It contains necessary operating system files plus middleware or even a deployed application, parcer.jar in our case. **VM templates** reference such VM images but also contain specifications for VM instance resources (number of vCPUs, amount of RAM). **VM instances** can be instantiated from a VM template, which are identical up to their IP addresses and hostnames. Such VM instances can process the same workload and are therefore grouped into **VM instance groups**. Figure 60 shows these relations.

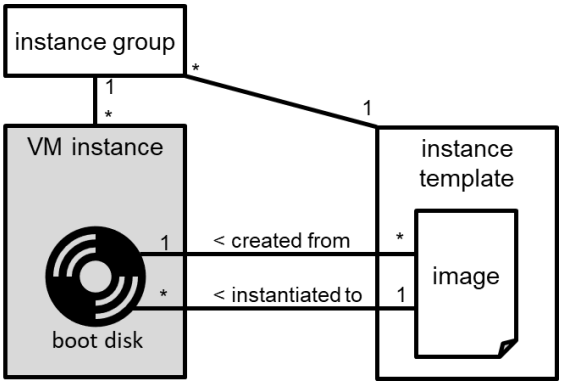


Figure 60: VM instances, groups, templates, and images

All VM instances expose the same TCP or UDP ports, which can be named at the VM instance group level as **named ports**.

17.8.2 Load Balancing

Incoming packets from the internet first hit a **forwarding rule** since it is this forwarding rule that is associated with a public IP **address**. If the packets constitute a http(s) request, the request is forwarded to a **http(s) target proxy**. If it is an https target proxy, it has an associated **SSL certificate** and terminates SSL (TLS) encryption. An associated URL map configures a mapping from requested URLs to **backend services**, which is capable of forwarding the request to healthy VM instances in VM instance groups. **Health checks**, which can be assigned to backend services, periodically check if VM instances respond reliably.

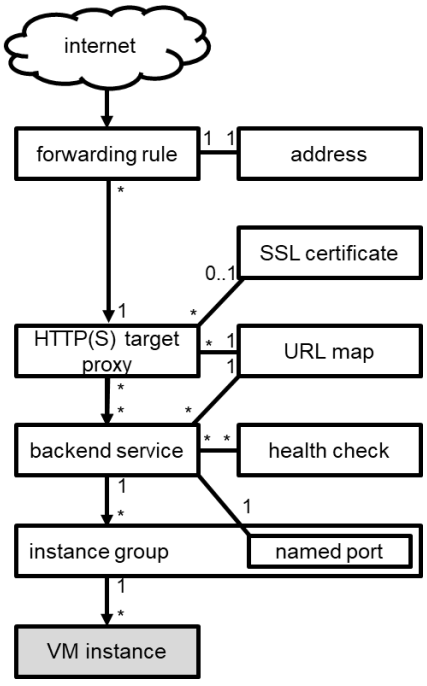


Figure 61: Load balancing

17.9 Scaling SQL Instances

Scaling out databases is a little more complicated than scaling out VM instances. In GCP a database management system is a set of so-called **SQL instances**. Besides the database itself, these SQL instances manage binary logs. A **binary log** buffers data modification requests (INSERT, DELETE,

CREATE etc.) before they are executed in the database. In order to create a **read replica**, which is a replica that receives only read requests (SELECT), the database gets a **backup**, which is restored to another SQL instance. Each subsequent modifying request is then propagated to both databases from there on.

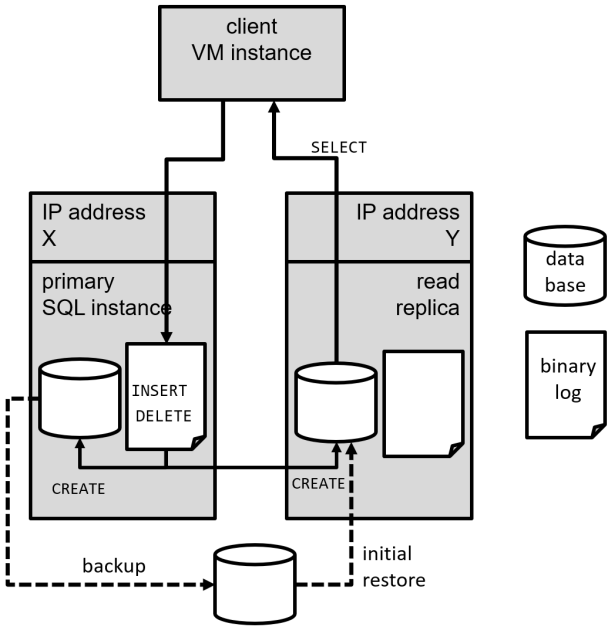


Figure 62: Read replicas

17.10 Database Failover

Failover in GCP databases can cross different zones. This keeps up the availability of such databases even in case of a disaster, which affects the whole data center (zone). In Figure 63, we have an active-passive-database (cf. Section 8.4.2), with an active **primary instance** in zone `europe-west-3-a` and the passive one in `europe-west3-b` with a **failover replica**.

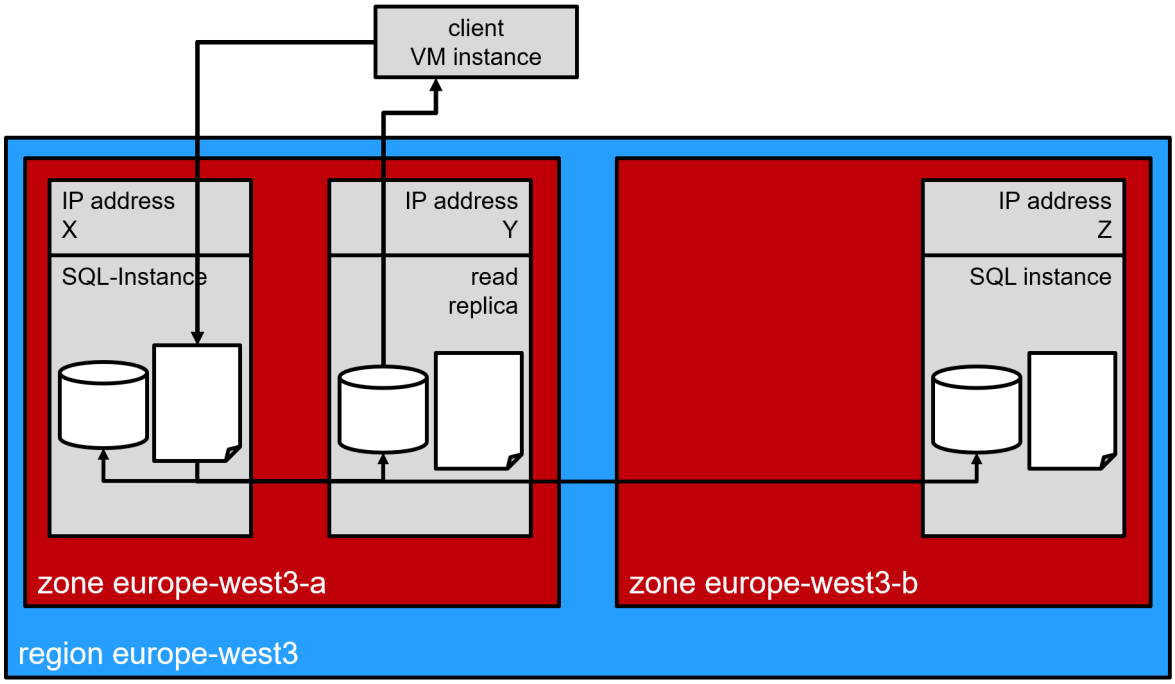


Figure 63: Database before failover

If the read-write-instances of `europe-west3-a` goes down, the SQL instance of `europe-west3-b` takes over. In order to keep the failure opaque to the database clients (e.g., VM instances), another read replica is created in `europe-west3-b` and the IP addresses are reassigned such that database connections in VM instances need not be reconfigured at all. Because a later failure could strike `europe-west3-b`, the former primary instances in `europe-west3-a` serves as the new failover replica once it is up again.

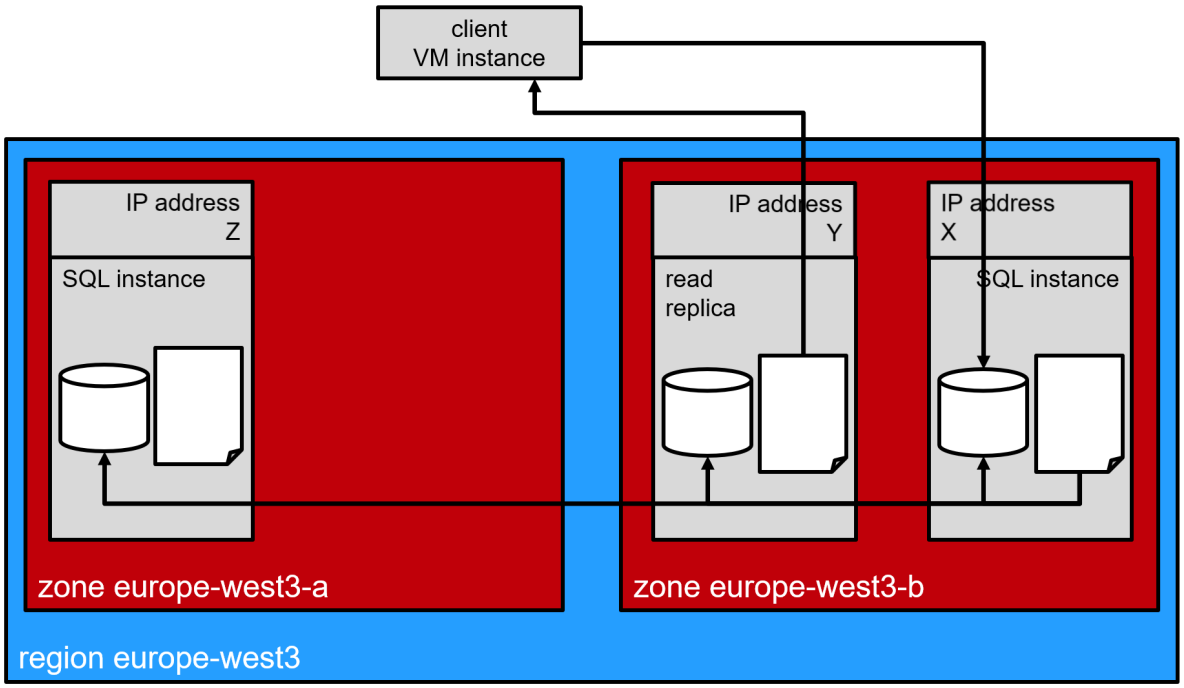


Figure 64: Database after failover

18 Machine Virtualization Techniques

In Section 4.3.3 we have learned that a program always runs a process, which manages resources for the program (memory in particular). A process runs on a machine with an operating system, which is a kind of runtime environment. In order to stress that process A runs on machine B, we say that A is **guest** of B, which is in turn is called **host**.

When **machine**[36] **virtualization** comes into play, we have even more runtime environments in-between: Processes run in "in-between runtime environments" that run on machines operating systems. So, processes are guests of these in-between runtime environments, which are in turn guests of the machine's operating system.

The purpose of such in-between runtime environments is to **isolate** their guest's process from those of other guests. Isolation means that the processes of different guests cannot interfere with each other via shared resources. Consequently, resources of bare-metal machines are **better utilized** / **less wasted** if a potentially conflicting process does not run exclusively on a bare-metal machine but along with others yet contained in a guest.

18.1 Virtualization Technologies

There are different kinds of virtualization technologies that differ in how they work and trade the qualities of Section 18.2.

18.1.1 Processes

Mundane operating system processes already provide some kind of virtualization: As explained in Section 4.3, **processes** are a small-footprint mechanism of operating systems to isolate running programs from each other by means of dedicated virtual memory address spaces (s. Figure 65). This makes it virtually impossible for a process to overwrite the memory of another process by accident.

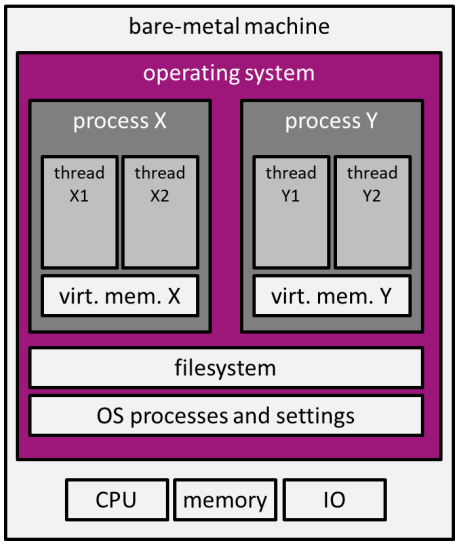


Figure 65: Virtualization by means of processes

[36] There are also other kinds of virtualization like **network virtualization** or **storage virtualization**, which are no subjects of this chapter.

18.1.2 Containers

Container virtualization makes use of Linux Kernel concepts for isolation (s. Chapter 19). Basically, a container is a Linux process hierarchy that is provided with an exclusive file system (mounts), host-name, namespace for users, groups and process ids, and network. Moreover, usage of resources like memory and CPU can be limited per container but is still by means of system calls to the kernel of the container's host operating system.

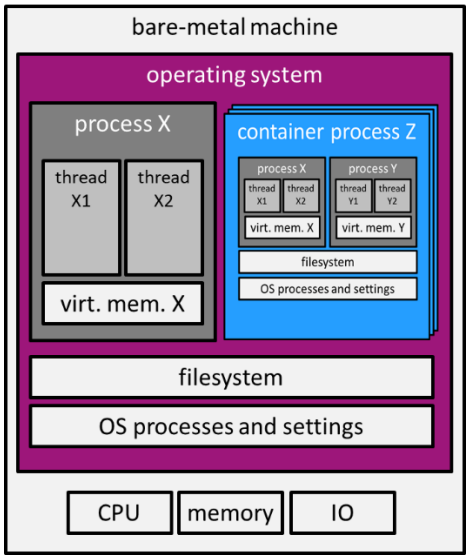


Figure 66: Container virtualization

18.1.3 Virtual Machines

Virtual machines abstract from real bare-metal machine hardware resources like CPUs, memory, and I/O devices. Operating systems can be installed on a virtual machine just like in a bare-metal machine. Ideally, an operating system cannot decide if it is installed on bare-metal hardware or on a virtual machine.

Virtual machines run on **hypervisors**, which provide virtual resources like vCPUs and map accesses to these resources to real hardware resources. There are two types of hypervisors:

A **native hypervisor**, a.k.a. **type-1 hypervisor**, directly runs on bare-metal machines. Figure 67 (left) depicts a typical stack on top of a native hypervisor. The main advantage of native hypervisors is its performance compared to hosted hypervisors. "VMware ESXi" is a widely used instance for native hypervisors.

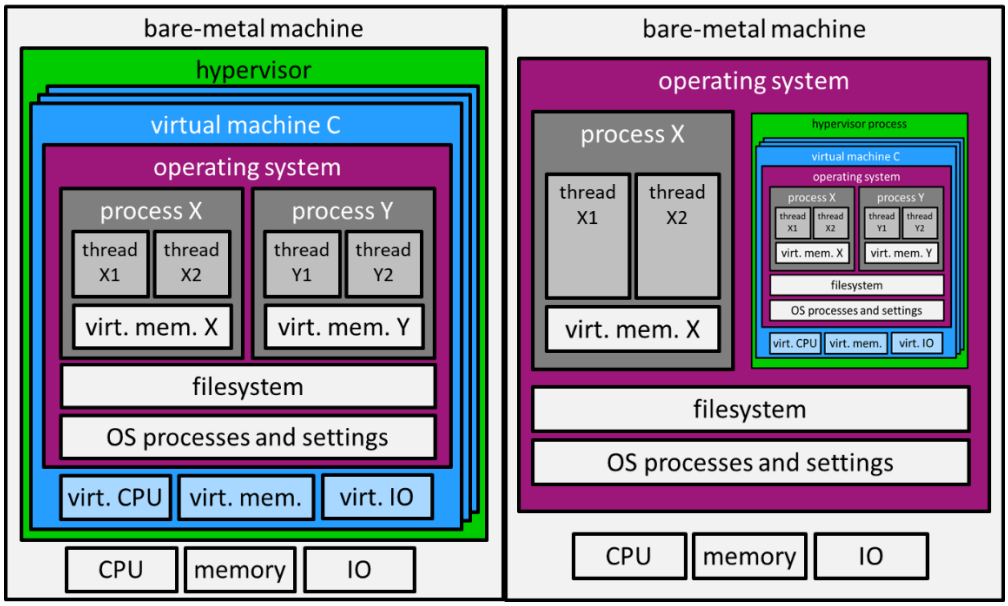


Figure 67: Native hypervisor (type 1) and hosted hypervisor (type-2)

A **hosted hypervisor**, a.k.a. **type-2 hypervisor** is being installed on top of a common host operating system as depicted in the right side of Figure 67. A hosted hypervisor creates a process in that **host** operating system for each virtual machine with its **guest** operating system. Of course, normal processes can still run in the host operating system side-by-side with the hosted hypervisor. This is the main advantage of a hosted hypervisor compared to a native hypervisor. "Oracle VirtualBox" is a popular hosted hypervisor.

18.1.4 Bare-Metal Machines

On the other end of the spectrum, on could allocate a dedicated **bare-metal machine** for each single process[37]. This would provide a maximum level of isolation between processes but of course a waste of resources.

18.2 Virtualization Technologies Trade-offs

In machine virtualization, one has to trade the following qualities.

	Isolation	**Host Dependence**	**Waste**	**Provisioning**
Processes	low	very high	small	very fast + easy
Containers	medium	high	medium	fast + easy
Virtual machines	high	low	big	easy
Bare-metal machines	very high	none (since no host)	very big	hard

Table 16: Virtualization technologies overall comparison

[37] Of course, this is a simplified view since one practically cannot prevent common operating systems form starting several processes on their own.

18.2.1 Isolation

- The evolution of programming languages has yielded concepts like, e.g., namespaces with accessibility rules that particularly allow for isolating parts of a single program from each other. This isolation, e.g., avoids that certain program parts can access certain variables and overwrite their values.
- Not just variables in a single program might be subject for conflicts. **Multitasking**, i.e., concurrently running programs, is common in all contemporary operating systems. Conflicts between different programs arise if they
- overwrite each other's data, may it be in memory or in some kind of storage, e.g. file system,
- reserve resources that are mutually exclusive, e.g., common TCP ports like 80 or 443,
- do not agree on certain system-wide settings like the hostname or the environment variable JAVA_HOME that points to a Java runtime environment, or
- unjustly compete with each other on resources that are shareable but limited like CPU, memory or I/O and thus must be fairly shared among resources.

Such conflicts cannot be avoided by means of programming languages concepts. Here, isolation must care for sharing and managing resources in a fair manner, such that processes get the "illusion" of complete access to these resources although they actually are limited and compete with others.

As Table 16 depicts, **processes** are rated comparably "low" with respect to isolation as just have their own virtual memory address space. Moreover, processes are not limited in resource consumption.

Containers additionally have their own filesystem, hostname, users, groups and process id namespaces, network interfaces (thus port number namespaces) as well as hostnames which lowers the risk of conflicts here. However, containers are less isolating than virtual machines as they do not use ring-1 hardware isolation [55]. Therefore, they are particularly susceptible to kernel vulnerabilities like "Dirty COW" [56] that break process isolation.

Virtual machines almost provide the same degree of isolation as bare-metal machines do. However, virtual resources must finally be mapped to hardware resources and thus compete with each other. For example, a process in one virtual machine could consume an unproportionable big share of the host's network interface maximum throughput. Processes in other virtual machines on the same host might suffer from this so-called **noisy neighbour**.

18.2.2 Host Dependence

- Machine virtualization requires a host machine. Such host machines may constrain the flexibility for machine virtualization, i.e. what might actually run in a virtualized machine.

In **containers**, processes have to be compatible with the Linux Kernel of the container's host. Processes in a **virtual machine** have to be compatible with the operating system running in that virtual machine but not necessarily with the operating system of the virtual machine's host.

18.2.3 Waste

- Waste of resources basically occurs due to two factors.
- **Redundancy**: Different kinds of virtualizations require different extends of data duplication and therefore waste of storage and memory.

- **Reservation**: Different kinds of virtualizations preallocate resources due to efficiency reasons. These are then mostly left unused but reserved and blocked from being used for other purposes. This is particularly true for the resource types memory and storage. Conversely, **utilization** of resources is the opposite of wasting resources due to preallocation. Cloud providers in particular are interested in avoiding such kinds of waste in their bare-metal machines.
- **Processes** might waste memory if the respective programs allocate memory (by means of, e.g., `malloc()`) without actually using it. However, this is somewhat inevitable and up to the developer to prevent.
- Since **containers** are just process hierarchies, they are comparable to mere processes with respect to waste. Just if the layering as explained in Section 19.3.2 is not used properly, containers might produce redundant copies of files in the host's file system.

During start-up, a **virtual machine** allocates a certain amount of memory, e.g., 4 gigabytes, independently of what the processes running in that virtual machine allocate in sum from that virtual machine. Modern virtual machines allow for certain dynamic adaptations of the allocated memory but these are usually bounded by certain limits that have to be guessed and configured in advance (minimum memory, startup memory, maximum memory). Moreover, such dynamic adaptations lower performance and thus give rise for a trade-off. The same is true for storage: Virtual machines have to preallocate storage in the host for their mounted filesystems independently how much space in these filesystems is actually used. Moreover, the boot disk image of each virtual machine carries a whole copy of an operating system with all its files. This results in a storage footprint in the gigabytes for each single virtual machine boot disk image.

18.2.4 Provisioning

Given that the level of isolation suffices, different virtualization techniques differ in how fast and easy a program can be provided with a runtime environment.

Of course, creating a **process** for a program in a ready-to-use operating system is much faster and easier than to order and install and configure a dedicated **bare-metal machine**. Moreover, practice shows that it can be hard to uphold immutability of bare-metal machine and, mainly due to lack of thorough documentation, reproduce a certain configuration, e.g., for horizontal scaling or for setting up an identical system cluster as an additional deployment stage in a continuous delivery pipeline. Machines that have such a unique and virtually irreproducible configuration are often **called snowflake machines**.

The small storage footprint leverages ecosystems of freely available **container** images. Moreover, container images are small enough to be versionable and transferrable deployment artifacts. It is faster and easier to provision a container than a virtual machine, let alone a bare-metal machine.

As demonstrated in Section 17.8.1, provisioning of identical **virtual machines** by means of virtual machine images is quite easy but neither fast nor slow.

19 Container-based Virtualization

Containers are another virtualization technology. With respect to the trade-offs in Table 16, containers are a compromise between OS processes and virtual machines. Containers are hierarchies of processes running in the host's operating system, which are isolated from each other and whose lifecycle is managed by a container engine (cf. Section 18.1.2).

19.1 Isolation via Namespaces

By means of Linux **namespaces** [57], processes in different containers are much more isolated from each other than those within the same container. Namespaces establish a view for processes in a container on certain resources and settings like the process list, users, mounts, hostname that is different from each other and that of the host.

Namespace	Purpose
inter-process communication (IPC)	Restricts communications of processes in separate namespaces
process identifier (PID)	Processes that are visible in a container are just a subset of processes running on the host. Moreover, the host's process identifiers (PID) of these processes are mapped to PIDs starting from 0 within the container.
user (USER)	Like for PIDs, a container might be provided with just a subset of the host's users and groups. Likewise, the respective user ids (UID) and group ids (GID) are mapped to different ids which a process in a container sees. In particular, on can map the root user of a container to a non-root user in the host.
mount (MNT)	Mount points are set namespace-wide and not system-wide. In combination with a `pivot_root` system call (change root), each container can be assigned a different root directory.
network (NET)	Network interfaces are visible just in one network namespace at some point in time but can be moved between namespaces. Virtual network interfaces pairs in two different namespaces can be used to create a bridge to physical interfaces in a certain namespace. Each network namespace has its own network stack with a private IP address, routing tables, firewall rules etc.
UTS (UTS)	A UTS namespace contains certain properties (observable via Linux command `uname -a`) which particularly includes the hostname.
control groups (CGROUPS)	Isolates subhierarchies in cgroup hierarchies from each other. Processes within a CGROUP namespace cannot collectively exceed the resources limits that have been set from outside.

Table 17: Linux namespaces

19.2 Resource Sharing with Cgroups

So, namespace mostly control *what processes see and are able to access*. Containers also make use of Linux **control groups (cgroups)** [58]. These are used to justly apportion *how much processes can consume*, i.e. the allocation of certain resources like memory, CPU or network I/O.

The connection between cgroups and the CGROUP namespaces of Table 17 is that by isolating cgroup subhierarchies into namespaces, processes in a single namespace have to share resources assigned to that namespace and cannot "steal" from other namespaces.

On its creation, one can set the memory limit for a Docker container. In the host's filesystem, this limit is stored in the file `/sys/fs/cgroup/docker/$CID/memory/memory.limits_in_bytes`. In the container (with some container id CID), the same file appears at `/sys/fs/cgroup/memory/memory.limits_in_bytes`. So, the limits for `docker` process group but also other Docker containers are out of reach for the container as these are just not visible in the containers filesystem.

19.3 Docker

19.3.1 Basic Docker Concepts

In Figure 68, we see basic concepts of Docker. The Docker engine is a set of processes which run on a Linux based host. **Containers** are created based on **images** which are local to the host. Missing images can be downloaded from remote **registries**. The download or creation of images or containers is done by means of a command line tool **Docker CLI**, which can be executed on a different machine than the host.

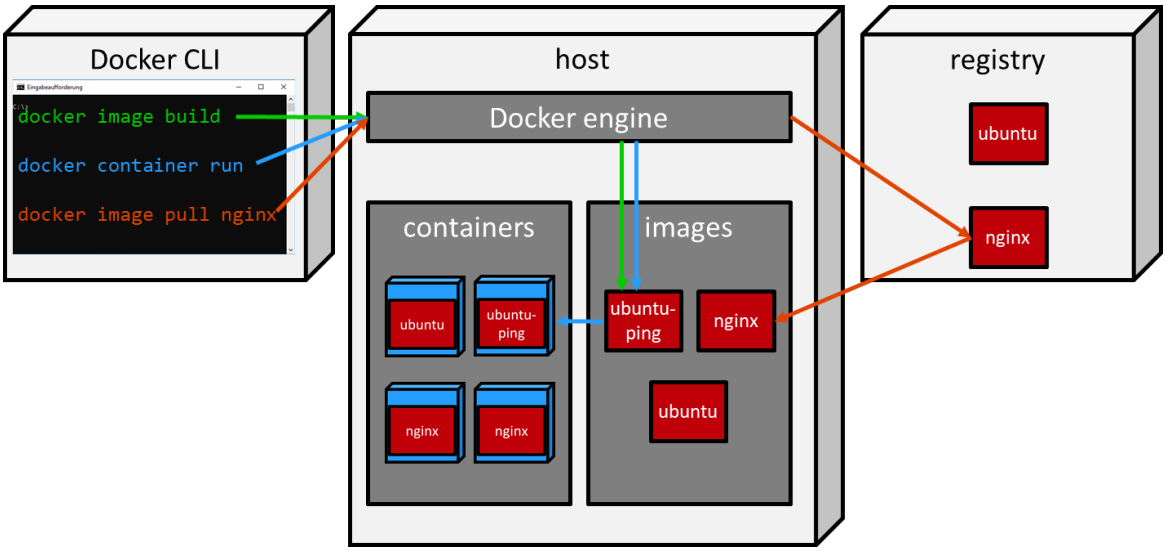


Figure 68: Basic docker concepts

19.3.2 Layers

Images are comprised of linearly prioritized **layers**. Each layer contains a partial file system, i.e. files and directories. A container cannot access partial filesystems directly but a filesystem that results from merging the layers. In case of a conflict, i.e. two layers have each a file with the same path and filename, the layer with the higher priority wins. These kinds of filesystems are also called **union file systems (UnionFS)**.

In Figure 69, both layers have a file `/dirA/fileA2`. In the resulting **effective filesystem**, the file is taken from layer 2 due to its higher priority 2.

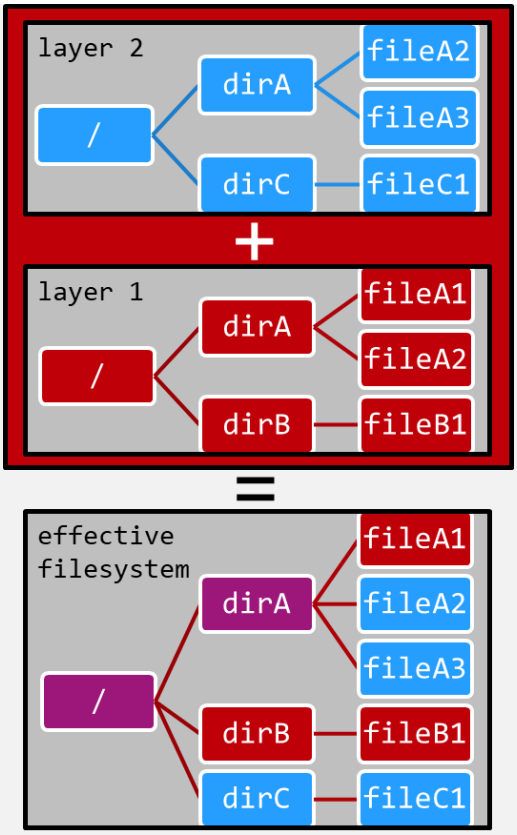


Figure 69: Layer example

Writes in the resulting filesystem are redirected to another layer with highest priority, called **thin-read-write layer**. Modifications to an existing file lead to a **copy-on-write** of that file into the thin-read-write layer.

19.3.3 Images

Images contain lists of read-only layers. In practise, images are not entirely different from each other: They often use the same layers which contain libraries and executables of operating systems or middleware. Since each layer need to be transferred to and stored once per docker host, this leads to a much smaller storage footprint compared to virtual machine images. Layers and images can be uniquely identified by hash values which are computed based on a layer's contents.

19.3.4 Dockerfiles

Dockerfiles contain instructions for the Docker CLI that specify how to build a new image. The `FROM` instruction basically includes all layers from another image. In the example of Figure 70, layers of the image `ubuntu:18.04` are included. (This figure depicts the layers and attributes of an image together with the Dockerfile instructions that yield layers.) Other instructions like `RUN` create new layers by executing Linux commands like `apt-get` or `copy`. In the example of Figure 70, the two additional layers add the `ping` command to the commands already included in the `ubuntu:18.04` image resulting in a new image `ubuntu-ping:1.0`.

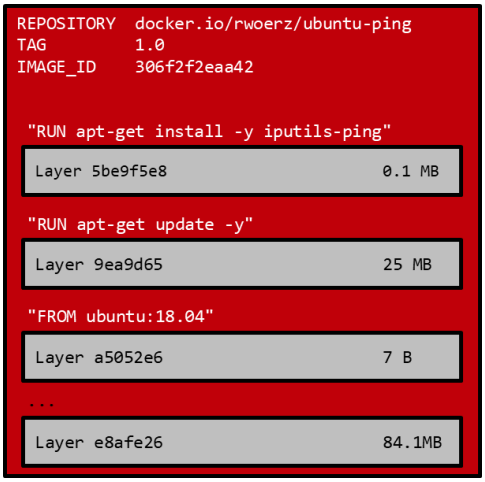


Figure 70: Example image and Dockerfile instructions

19.3.5 Tags, Repositories and Registries

Like layers, images can be uniquely identified by their SHA-256 hash value. Since these identifiers are not quite mnemonic, one can identify images by `<repository>:<tag>`. A **repository** contains one or more revisions of an image and a **tag** identifies a specific revision like `1.0`. If just `<repository>` is provided, `<tag>` defaults to the value `"latest"`. A value for `<repository>` is not just an identifier but also a locator for a repository. It consists of the following segments: `<registryHost>:<registryPort>/<namespace>/<name>`.
Here, `<registryHost>:<registryPort>` identify a server called **registry** that stores images. It is optional and defaults to `docker.io:443`. An optional **namespace**—totally different from the namespace of Section 19.1—is a means to group different repositories in a single registry and is often equal to a username in a public registry. The **name** is mandatory[38].

For example and putting it all together, a certain image might be identified and located by `localhost:5000/rwoerzbe/ubuntu-ping:1.0`
which means that `dockerd` would pull it from registry `localhost:5000` within the namespace `rwoerzbe` and name `ubuntu-ping` with tag `1.0`

19.3.6 Containers

Just like for virtual machines, a Docker **container** can be created based on a docker image. At the bottom line, a container is just a hierarchy of user space processes that run on the docker host and to system calls to the docker host's kernel[39] like any other process.

What make processes in a container different from other processes is that they are isolated from processes outside that hierarchy by means of namespaces and cgroups (cf. Section 19).

[38] Unfortunately, Docker's terminology around registries, repositories, names, namespaces, and tags is flawed even in the official Docker documentation [35]. That is why the docker command `tag` sets a `<repository>:<tag>` instead of just a `<tag>`. Moreover, `<repository>` is sometime referred to as `<image>` or `<name>`.

[39] This is an important difference from virtual machines. Since images usually are based on some Linux derivative image, one might think that each container runs its own kernel. Those Linux images, however, are rather included in order to provide Linux tools like bash and libraries.

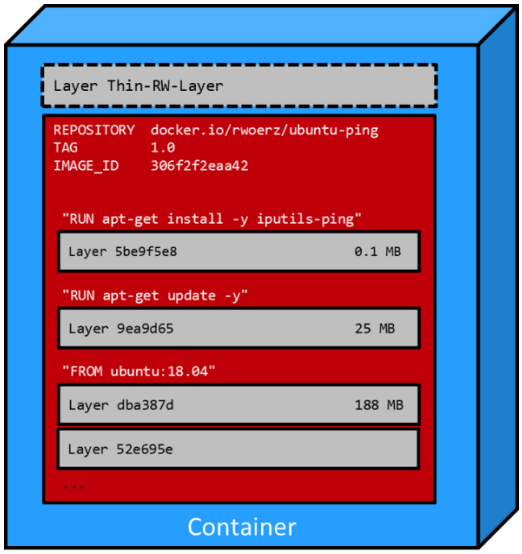


Figure 71: A Docker container

Figure 71 exemplifies the instantiation of the image from Figure 70. Since processes in a container might write to files, each container has a thin read-write layer. Writes to this thin read-write layer get lost each time the container is restarted.

19.3.7 Docker Processes

Actually, containerized processes are not created and treated like normal host processes. Docker has a bunch of processes with certain responsibilities:

The Docker Deamon **dockerd** exposes an API via a local UNIX socket (`/var/run/docker.sock`) for remote clients via TCP (default port 2375). If necessary, the **docker-proxy** maps ports of the docker host to ports which a container exposes. In a single host, all container lifecycles are managed by **containerd** [59]. This includes "the complete container lifecycle of its host system, from image transfer and storage to container execution and supervision to low-level storage to network attachments and beyond". Here, containerd delegates the actual creating, starting, and stopping of containers to the command line tool **runc**. When containers are started by runc, it creates a child process named **containerd-shim**. Child processes of containerd-ship are exactly those that are considered to be the container processes. Then, runc reparents container-shim to its own parent containerd. A containerd-shim process keeps the container processes alive after runc has exited and even if containerd or dockerd terminate. Moreover, it reports back container process' (exit) status to containerd.

Figure 72 depicts a cut-out of a Linux process hierarchy within a docker host. We can see three processes which are supposed to be instantiated from a `nginx` image. The leftmost one has an additional `bash` process, e.g. due to an administrator which has logged into this container via `docker exec`. The container just runs `nginx`. The rightmost container is a container in the making, i.e. `runc` is still running and has not yet re-parented `container-shim` to `containerd`.

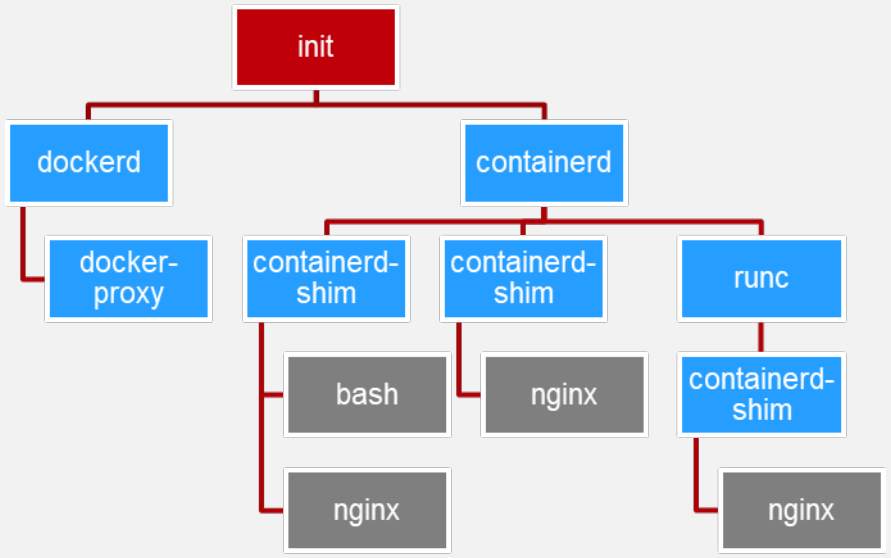


Figure 72: Docker processes

19.3.8 Mounts

Files in a Docker container can be stored in four different ways in the filesystem. Figure 73 and Table 18 summarize them.

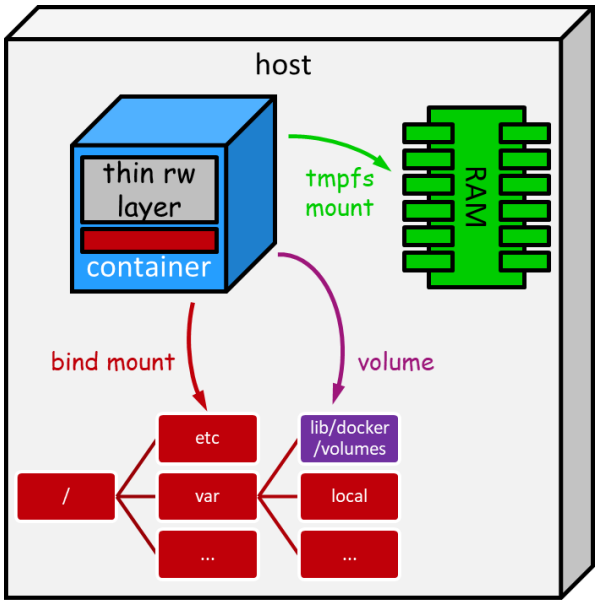


Figure 73: Docker storage options

Storage option	Durability	Sharable	Description
thin writable container layer	until container removed	no	If a process in a container writes to a file in an arbitrary directory of the container's filesystem, those writes are earthed at a thin writable layer owned by that container. Such thin writable layers have< higher priority than any other layer in a container's image. Existing files from lower layers are copied to this writable layer on write access (copy-on-write). However, a writeable layer is bound to its specific container. If the container is removed, so is the thin writable layer.

Storage option	Durability	Sharable	Description
tmpfs mount	until container stopped	no	With tmpfs mounts, one can create RAM disks in the host and mount them in a specific directory in the container's filesystem. Of course, writes in these tmpfs mounts are very fast. However, contents of these directories again do not survive a container restart. Moreover, tmpfs mounts are not shareable, i.e. no other container nor the host can access a tmpfs mount.
bind mount	until deleted on host	among containers and host	A bind mount is a directory in the host, which is mirrored as a directory in the filesystem of one or more containers.
volume	until deleted on host	among containers	Like bind mounts, volumes appear just as directories in the container's filesystem. On the host, the volume is directory at a specific location and a specific subdirectory structure, i.e. it is not meant to be accessed by processes on the host. Different volume drivers allow for storing the volume not only on a host's disk but also on other types of storages, e.g. cloud storage.

Table 18: Docker storage options

19.3.9 Docker Command Line Interface (CLI)

Like `gcloud` (cf. Section 17.3), Docker can be managed by a single executable `docker`. Docker commands are grouped into commands that revolve around images, container, and volumes just to name some of them. Figure 74 shows the most important commands. Please refer to [60] for an exhaustive documentation about the Docker CLI.

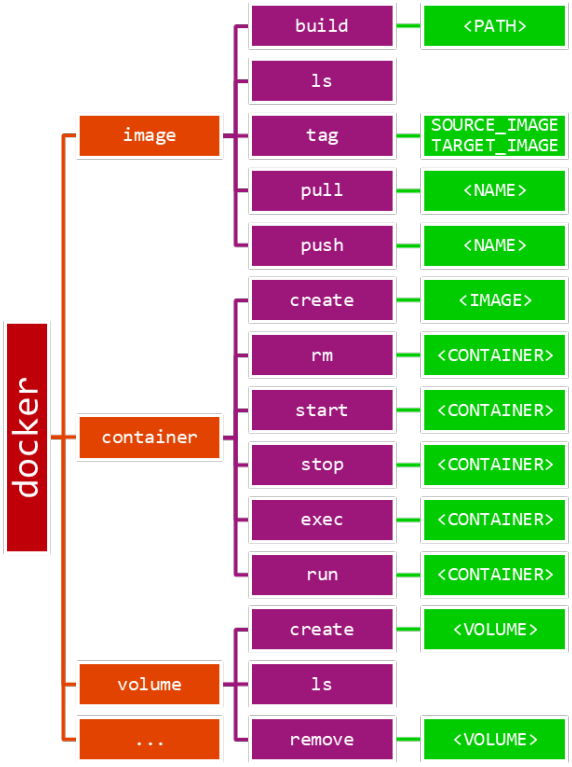


Figure 74: Docker CLI

20 Container Orchestration

20.1 Container Orchestration Features

Based on container-based virtualization, container orchestration typically provides the following additional features:

Feature	Description
Optimized resource utilization	Just with container-based virtualization, an administrator has to decide which container should run on which host. Container orchestration automates the scheduling of containers to hosts using some kind of binpacking algorithm and thereby optimizing the overall host utilization.
Self-healing	Just with container-based virtualization, if a host dies, all of its containers die with it forever. With container orchestration, containers are automatically recreated on healthy hosts.
Horizontal scaling	Containers with the same image can be created automatically due to increased workload even in an automatic fashion.
Service discovery	Clusters of containers with the same image serve as endpoints for a so called "service" which can be assigned with a constant domain name and which can be discovered by other containers using common DNS.
Load balancing	Container orchestration provides load balancing mechanisms in order to distribute workload to containers with the same image.
Rollouts/-backs without observable downtimes	Containers based on the same image can be automatically updated to a new image version one by one. Together with load balancing, the whole cluster of containers seem to be complete reliable, i.e. has no downtime that is observable from their clients.
Secret- and configuration-management	Software systems under development normally use distinct clusters for development, tests and production. Configuration data and secrets like password in particular differ between these clusters and are thus better be stored and versioned apart from other data.
Storage orchestration	Cloud providers differ in what kinds of storage options they offer (cf. Section 17.7) and how these storages can be accesses. Container orchestration provides a layer of abstraction above these cloud provider specific storage options thus making containers independent of a certain provider specific storage option.

Table 19: Container orchestration features

20.2 Basic Kubernetes Concepts

As of 2019, **Kubernetes (k8s)** [61] is the predominant container orchestration tool. Figure 75 depicts Kubernetes' main concepts: Kubernetes manages **Kubernetes clusters** of nodes that are either master nodes or worker nodes.

20.2.1 Nodes and Processes

Worker nodes[40] are (virtual) machines that run system specific processes realizing system resources of Chapter 15. These processes run in containers that are managed by the container runtime **containerd** (cf. Section 19.3.7). Each container is assigned to exactly one pod. A **pod** is the smallest unit of computation that Kubernetes manages and can contain one or more containers. The

[40] Note that we defined a node to be a process (or cohrerent set of processes) in a network (cf. 4.3.5), where a Kubernetes worker node is a (virtual) machine.

kube-proxy on a worker node cares for mapping ports that are exposed by containers to exposed ports of the node. A kubelet is a process on each worker node that realizes the control plane of Kubernetes: It communicates with the **apiserver** in order to receive pod lifecycle commands and report back status of the worker node.

Master nodes manage the state of a Kubernetes cluster. The **controller manager** is a process with different **controllers**, which are basically control loops, e.g. program loops that regulate the state of the Kubernetes cluster. It cares for noticing and reacting: If a worker node fails, it regulates the desired number of pods, creates and updates endpoints which connects Kubernetes services to pods and manages security tokens. The **kube-scheduler** assigns newly created pods to worker nodes. Administration data like the desired state is persistently stored in the key-value store **etcd**. Communication between a master node and a worker node takes place via the **apiserver**. This is also true for the command line tool **kubectl**, which sends admin commands to master nodes.

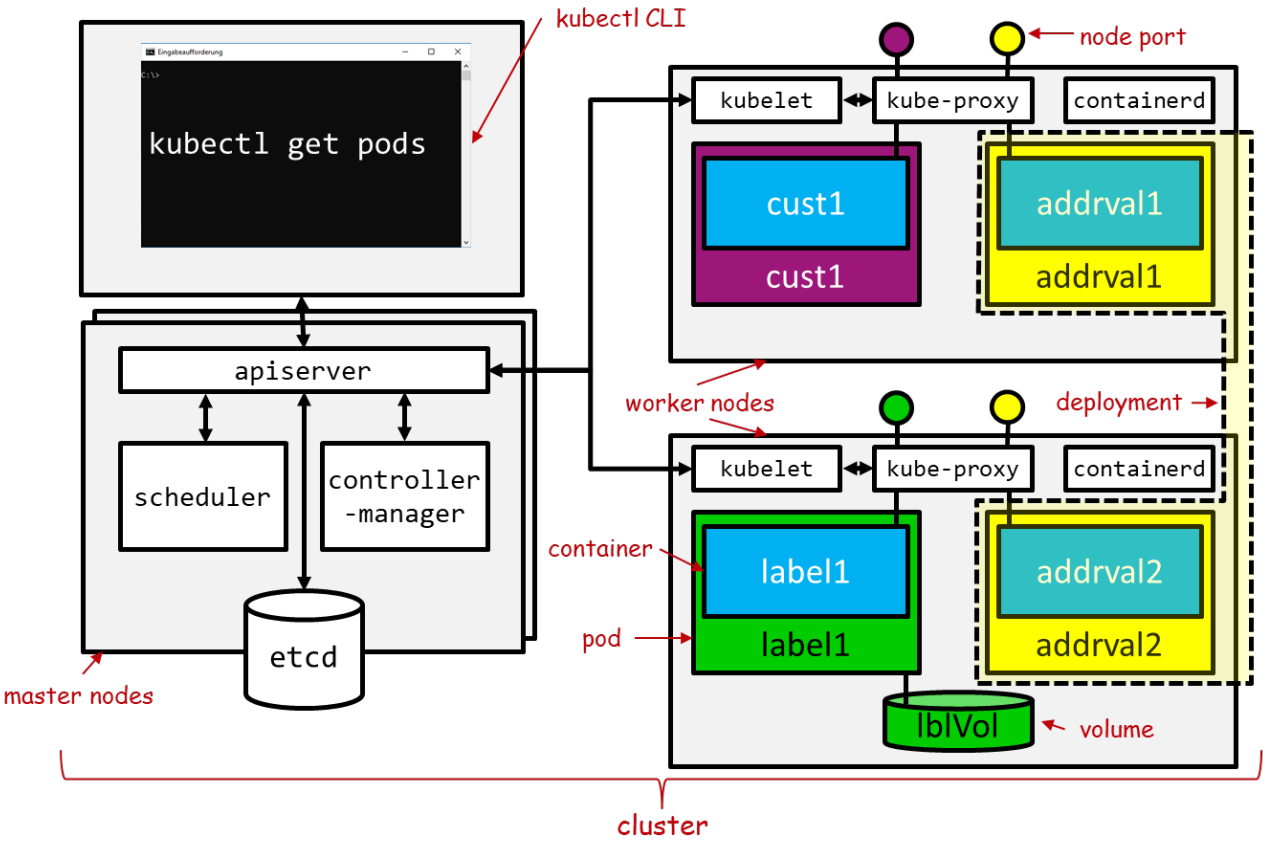


Figure 75: Basic Kubernetes concepts

20.2.2 Kubernetes Objects

In Kubernetes the constituents of cluster's state are defined by an object model. Figure 76 depicts a cut-out of the **object meta model** which specifies how objects of different types are related to each other. This is similar to GCP project which is likewise **software-defined** and comprised of interrelated resources (cf. Chapter 17).

A Kubernetes **state** is modelled by an instance of this meta model. It conforms to the object meta model and so do Kubernetes **specifications** which define partial object models in the format of "Yet Another Markup Language (YAML)" or "JavaScript Object Notation (JSON)". Moreover, the object meta model is reflected in the commands of the kubectl command line interface, which is just a client of the RESTful API that also speaks in terms of the object meta model.

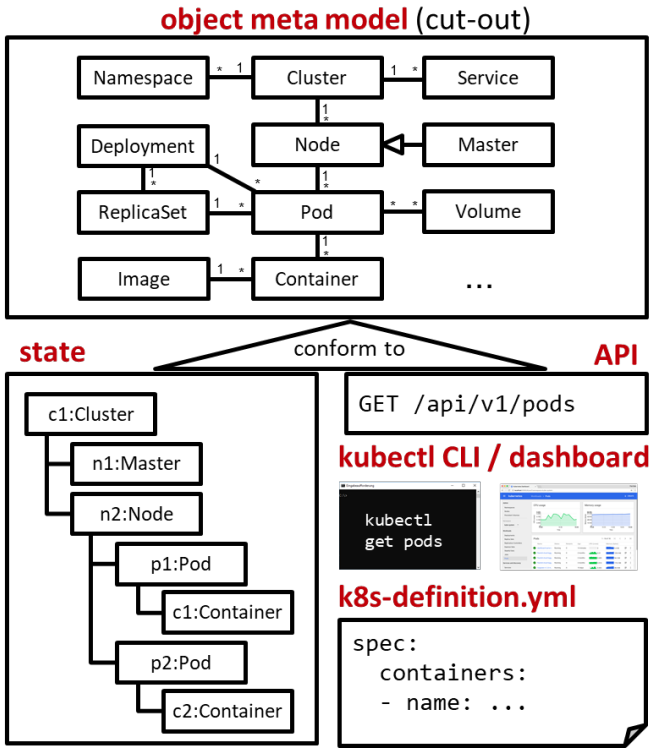


Figure 76: Kubernetes' object model

20.2.3 States

In Kubernetes, we talk about three states of a Kubernetes cluster as depicted in Figure 77. A **desired state** is a set of properties, an **actual state** is supposed to adhere to. Such properties typically are the number of pods with containers of a certain image.

The desired state is modified intentionally via calls to the API, e.g. by means of the kubectl CLI or via autoscaling. If it changes, the scheduler and controller manager cares for making the actual state consistent again with the desired state. If the actual state deviates from the desired state due to a worker node failure, the controller manager realignes the actual state with the desired state by creating the concerned pods on other worker nodes.

Sometimes the **observed state**, e.g. what kubectl or the dashboard outputs, lags a little behind the actual state.

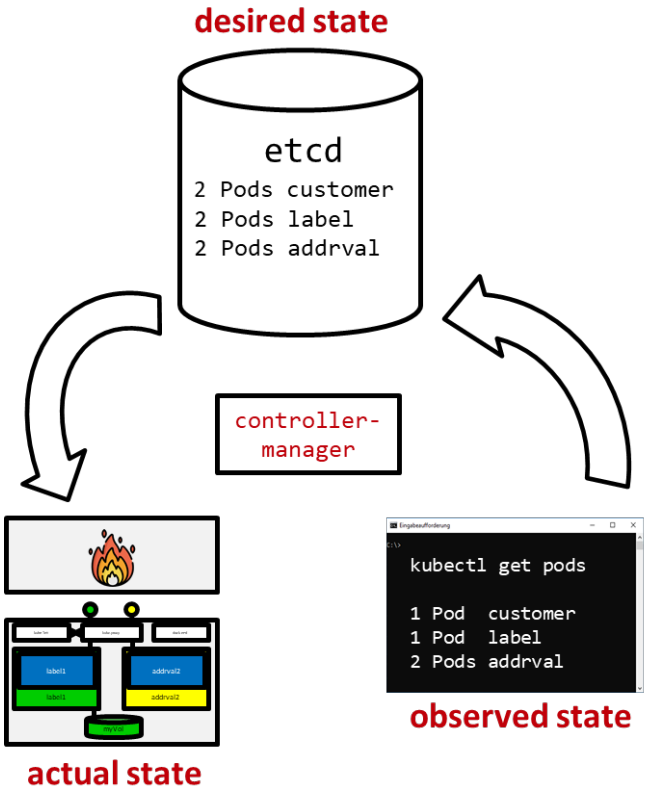


Figure 77: States in Kubernetes

20.2.4 YAML Definition Files

One can successively build a desired state by repeatedly calling the API via lengthy kubectl commands. A better practice is to define the desired state in definition files which are normally formatted in the Yet Another Markup Language (YAML). Such files have sections, each of which define the properties of an object of a certain **kind**, i.e. meta object in the meta object model. **Names** and **labels** can be defined beneath metadata and are relevant for selecting the respective object later on. The **spec** is specific to the kind. Normally, there is just a definition for a single object per file. If there are more than one, these are separate by a triple dash ---.

```
apiVersion: v1
kind: Pod
metadata:
  name: label
  labels:
    app: parcer-label
spec:
  # definition specific to the "kind"
---
apiVersion: v1
kind: Service
metadata:
  ...
spec:
  ...
```

For example, a definition for a deployment could look like the following:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: addrval-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: addrval
  template:
    metadata:
      labels:
        app: addrval
    spec:
      containers:
      - name: addrval
        image: addrval:1.1
        ports:
        - containerPort: 8083
```

Here, we define a Deployment which has two pods each of which containing a container based on the image `addrval:1.1` and that expose a port `8083`.

If this specification is applied by means of

```
kubectl apply -f filename.yml
```

then two pods are created on the worker nodes.

21 Hypertext Transfer Protocol

In a distributed system, quality attributes like maintainability, dependability and performance do not only depend on what programs run the nodes but also how nodes communicate. This is particularly true for client-server systems, where clients (a) might be developed by independent parties, e.g. customers of the company that implements the server-side, and (b) are connected to the server-side via the internet, which is notoriously unreliable, insecure, and susceptible to high latencies depending on the proximity of the client and the server.

Using the right protocol right and designing application programming interfaces (API) in a proper way, is crucial especially for large systems. With protocol, we mean protocols on the ISO/OSI application layer and their influence on architectural styles.

As we have seen in Section 7.2, there are different types of workloads. Request-oriented workloads are usually conveyed by means of the Hypertext Transfer Protocol in version 1.1 (**HTTP/1.1**) [62] or version 2 (**HTTP/2**) [30].

In HTTP is intrinsically request-response oriented. Request and response message are finite despite HTTP/2 particularly adds a concept also called streams to the standard. However, these streams are just a way to emulate TCP on the application layer, i.e. bidirectionally transfer frames between client and server, where each frame is part of a certain request or response message of finite and predetermined length.

21.1 Resources

HTTP is about transferring data records in request and response messages

- which are called **web resources** (or just resources),
- which are **distributed** over the so-called World Wide Web,
- which are identified, addressable, and can refer to each other by **uniform resource locators**, and
- which have different and negotiable **representations** also called **media types** (formerly: MIME type) like `text/html`, `text/xml`, `application/pdf`, `text/javascript`, `text/css`, `application/octet-stream`.

So, a resource is conceptually different from its representation. A resource might be anything, e.g. be a geographic location or some record in a database. Both of them cannot be embedded a such a message in its original form but just a representation of them.

21.2 Request and Response Messages

Request and response messages have to parts: a mandatory **header** and an optional **body**.

The header contains numerous lines which are mostly key-value pairs which constitute meta data. Particularly, a server can infer an **effective request URI** from the header.

The body contains the actual representation of a resource.

21.3 Uniform Resource Identifiers

A **uniform resource identifier (URI)** [63] is a character string that uniquely identifies a web resource like the examples below.

```
ftp://ftp.is.co.za/rfc/rfc1808.txt
http://www.ietf.org/rfc/rfc2396.txt
ldap://[2001:db8::7]/c=GB?objectClass?one
mailto:John.Doe@example.com
news:comp.infosystems.www.servers.unix
tel:+1-816-555-1212
telnet://192.0.2.16:80/
urn:oasis:names:specification:docbook:dtd:xml:4.1.2
```

A **uniform resource locator (URL)** is a URI that is also an address, i.e. contains information about how to look up a certain resource. It contains multiple segments as shown in Figure 78

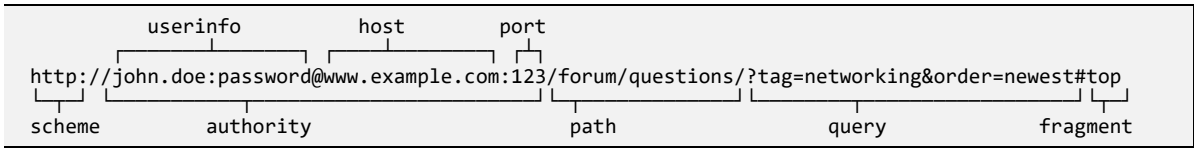


Figure 78: Uniform Resource Locator

21.4 Request Methods

Each request has a mandatory **request line** in its header. In this request line, the **request method** (sometimes called **verb**) is a mandatory segment.

Some request methods are **idempotent**, that is in a sequence of identical requests with such methods just the first request modifies a resource but not the subsequent requests -- given that there are no concurrent requests to the same resource.

The PUT method creates or updates resource identified by a URI. Subsequent identical updates via PUT do not alter the resource any more.

Some idempotent methods are said to be even **safe**, i.e. they do not modify resources at all thus do not modify the server-side state of a web application. It is particularly safe to do a second identical request with such methods if the first had no response (timeout). Idempotency implies safety but not the other way around.

The GET methods merely reads a resource but does not modify it.

A method is said to be **cacheable** if the respective response might be served from a cache in a subsequent (GET) request.

Method	Properties	Description
GET	safe + cacheable	Used to transfer a resource representation to the client. The following prints out the response for the given resource. `curl -X GET http://httpd.apache.org/docs/2.4/en/license.html -is`
POST	(cacheable)	Requests the server to process the resource representation enclosed in the POST request body. The kind of processing is identified by the request URI.

Method	Properties	Description
		A common "kind of processing" is just to add a resource to an existing collection of resources. The request body carries the resource representation to add, the request URI identifies the collection and the response body might return the resource representation with additional attributes like an internal ID. The following adds a resource to a collection `curl -X POST -d "key=v" https://ptsv2.com/t/1zvfr-1561032323/post -is`
PUT	idempotent	Requests to create or, if already existent, replace a resource targeted by the request URI by the resource represented in the request's body.
DELETE	idempotent	Requests the origin server to unmap the request URI the respective resource. From a client's point of view, the resource is then inaccessible by means of that URI. However, it is up to the origin server how to implement this unmapping.
HEAD	safe + cacheable	Identical to GET except the body in the response is omitted. `curl -X HEAD http://httpd.apache.org/docs/2.4/en/license.html -is`
TRACE	safe	Just echoes a resource representation in a request in the respective response. This method is for mere debugging purposes since the response also contains potential (header) modifications made by intermediate proxy servers. This method is often disabled due to security reasons.
OPTIONS	safe	Response options and requirements (as key-value pairs) that are available and have to be considered by a client when dealing with a resource but not the response (in some representation) itself. In particular, valid request methods are included in the response. The following command line yields a response for the given resource `curl -X OPTIONS http://httpd.apache.org/docs/2.4/en/license.html -i`
CONNECT	-	Used to tell a proxy to establish a TCP/IP connection with the given server and forward any subsequent TCP packets with the server IP to that server. This is particularly useful if a client is aware that it communicates with a server via an outbound proxy and the communication is supposed to be TLS encrypted (https).

Table 20: HTTP request methods

21.5 HTTP Caching

In the World Wide Web, client and server may be up to 20,000 km apart and have network routes of limited capacity. Therefore, response time (cf. Section 7.3) is an intrinsic problem which can be mitigated using caches (cf. Section 15.12). However, caching itself has an intrinsic problem, too: One has to trade the benefits, i.e. reduced response times, with the downsides, i.e. a higher risk for inconsistencies due to stale caches and security issues due to unauthorized use of cache contents.

HTTP has built-in support for web caching [38]. Conceptually, a **HTTP cache** (a.k.a. web cache) is a local store for previous response messages that came from another cache or the **origin server**. This local store is attached to a server that controls the storage, retrieval and deletion of such response messages. A HTTP cache is either a **shared cache** that serves responses for multiple clients or a **private cache** that serves responses just for a single user of a single client. Browsers[41] are the predominant clients for HTTP. They have a private cache for each of their (operating system) users.

[41] HTTP specifications rather speak the more generic term "user agent", which is a client program that is user facing, i.e., has a user interface.

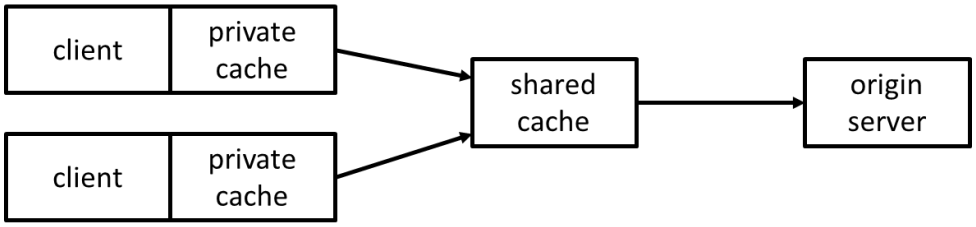


Figure 79: HTTP caches

The caching behaviour of a response and thus the trade-off between network latency and transmission time on the one hand and consistency and security on the other hand can be controlled by numerous **cache directives**. These are key-value pairs in the HTTP header field `Cache-Control` (like `max-age`, `s-maxage`, `no-store`, and `private`) and a date-time for `Expires`. These directives determine (a) if a response is stored in the first place by a HTTP cache and (b) whether it can be used, i.e. is fresh enough to be a **cache hit** for a request, i.e. a **cache miss** due to the response staleness.

The directive `no-cache` causes a **validation requests** that includes an `ETag` value of the cached response in the `If-None-Match` field. `ETag` values are checksums (often hash values) for the resource's contents. Only if the validation response from the origin server is positive with a status code `304 Not Modified`, the cached resource is a cache hit. The response time for validation requests mostly consists of propagation delays (cf. Section 7.3). Transmission delays are short due to the small messages exchanged in the validation. Therefore, validation requests are particularly helpful for big resources where transmission delay clearly dominates over propagation delay.

Figure 80 shows the initial request and respective response for the resource `/file`. The response headers particularly include a `max-age` of two minutes after which the response needs to be renewed. Until then, validation requests suffice.

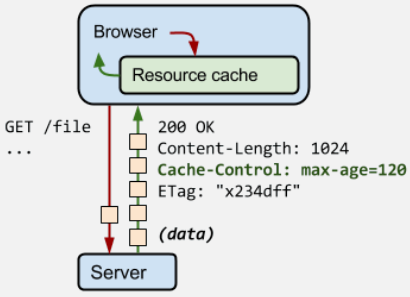


Figure 80: Initial request for resource "/file" [64]

Figure 81 exemplifies a validation request. Here, the private cache (resource cache) already contains a response with `ETag x234dff`.

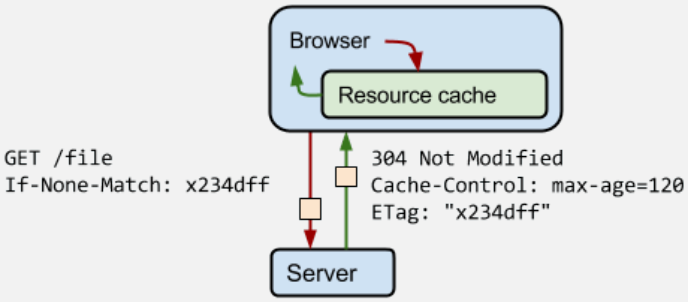


Figure 81: Validation request and response [64]

For a full explanation of all directives please refer to [65].

A typical setting for `Cache Control` looks like the one in Figure 82. The first response often contains a server-side generated HTML resource. Since this often is highly dynamic, it should better not be reused from a cache without validation. However, the HTML contains URL references to other resources, which have to be fetched as well. The `style.3da37df.css` is a more stable resource and bears no secrets. The filename and thus the URL to the resource contains **fingerprints**, `3da37df` in this case. This is a hash value that depends on the contents of that file. Therefore, if the CSS changes to a new revision, this hash value changes and therefore the URL for this resource. Since a resource revision (with a constant URL) never changes by design, the `max-age` could be set to a very high value, e.g., one year. The same is true for the `script.8sd34ff.js`. However, this script might contain confidential data and should not be stored in shared caches, hence the `private`. The `photo.jpg` does not use fingerprinting. However, one can live with a stale `photo.jpg` for one day at most, so the `max-age` is one day here.

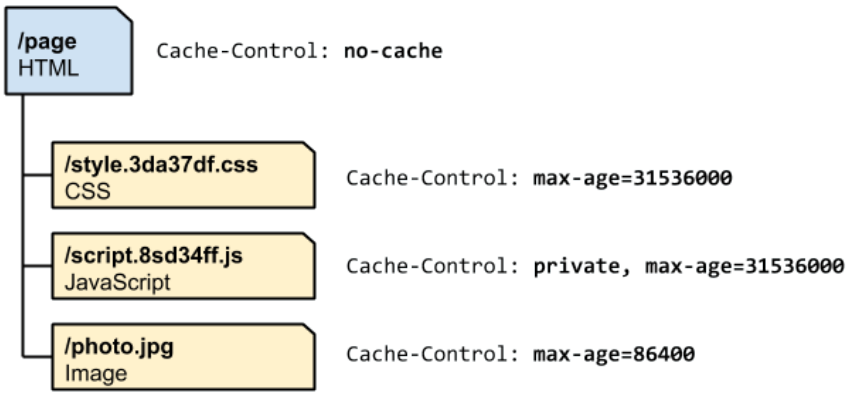


Figure 82: Cache Control example [64]

21.6 Representational State Transfer (REST)

Representational State Transfer (REST) as defined by [66, p. 76 ff.] is an architectural style for web applications and their web APIs that builds upon HTTP. This architectural style is defined by the following principles.

21.6.1 Client-Server Architecture

REST is just applicable for client-server architectures with a request-response communication style. Peer-to-peer architectures or unidirectional messaging cannot be subject of REST.

21.6.2 Statelessness

The term "statelessness" is ambiguous: Of course, a web application is in a sense stateful since all of the everchanging data it stores (in a database) can be considered to be a (very complex) state.

> In ReWo-Rate, the set and interrelations of users, products, transaction and ratings can be considered the application's state.

In HTTP and REST in particular, the term statelessness is used in a narrower sense: For web applications, a **session** between a client and a server, e.g. a coherent and subsequent sequence of request and responses, is associated with a session state[42]. A session state contains state information

[42] Session state and session are often used synonymously.

that is relevant for just the duration of that session such as navigational state and authentication state.

HTTP itself does not maintain session states (as opposed to, e.g., the File Transfer Protocol (FTP)). A request must be understood by a server independent of other (preceding) requests from the same client. Consequently, a server must understand a request even if it has recently been restarted (and thus all process memory data has been gone) or preceding requests have been processed by other servers in the same cluster.

Although, neither HTTP nor REST define how the path segment of a URL has to look like (and how mnemonic it has to be), a URL like

```
http://example.com/someResourceCollection/next
```

clearly hints to a violation of this principle because a "next" is just meaningful if the server stores an iterator state as part of its session state.

Before the advent of REST, it was common practise to store state in a server process memory as some kind of data record, which is then called **server-side session**. Cookies [67] were invented particularly for being used as session cookies, i.e. for referencing a particular server-side session in every request. These server-side sessions require that requests from the same client-side session are always routed to the server process holding the server-side session by means of sticky sessions (cf. Section 15.3).

In Java, one typically uses the Java Servlet API [68] in order to access the session during the processing of a request like this

```
HttpSession httpSession = httpServletRequest.getSession();
int requestCounter = (Integer)httpSession.getAttribute("counter");
httpSession.setAttribute("counter", requestCounter + 1);
```

This very simple example just stores the number of requests so far in the session state.

Of course, this has a negative impact on reliability: Server-side sessions typically hold **navigational data**, **authentication data** as well as sometimes **preliminary transactional data**, e.g., content of a shopping cart. In case of a server failure these states are lost for the concerned clients which have to re-login into another server. One might mitigate this by **session-replication** which synchronizes server-side sessions among the servers in a cluster. However, session-replication opposes scalability since the synchronization effort grows quadratically with each new server in the cluster.

If the data in the request suffice to process a request, then there is no need to store state in a server-process' memory. However, session state is not completely inevitable in web applications. For example, no user wants to provide his or her authentication credentials in every request or wants to lose the navigational state, e.g. the current page in a paginated list. Therefore, REST advocates to keep the session state on the client side.

Navigational states can easily be expressed in resource representations by means of **hypermedia**: A resource representation may contain **links** that trigger actions which are valid in the current navigational state.

For example, a page in a paginated list is such a resource which typically contains links for forward and backward navigation.

Authentication states can be kept in **cookies** which are key-values that are put in every request header that targets a certain server. **JSON Web Token (JWT)** [40] is a commonly used standard format particularly for authentication information that can be transferred via a cookie. Please note that a JWT cookie is not a mere reference to some server-side data record but a self-contained data record (i.e. a value, not a reference) that is supposed to be understood by every application server of the same kind behind a load balancer.

In theory, even **preliminary transaction data** could be transmitted back and forth in each request and response. In order to keep these messages small, it is often better to keep these data on the server-side however not as a server-side session, i.e. a data structure in the memory of a single (application) server process. Instead, such data should either be persisted in a database like normal transactional data or at least in a persistent cache service (cf. Section 15.8). Casually said, the server has to click the save button after every request.

> Contents of a web shop's shopping cart or user's input like receiver addresses or payment information during checkout are an example for preliminary[43] transactional data. These data become final transactional data after a legally binding purchase. Although it would not be catastrophic yet annoying if these data get lost along with a session.

21.6.3 Uniform Interface

"Uniform interface" means that a RESTful API, i.e. the API of a system that adheres to the REST principles, follows the constraints below:

a) Resource identification in requests

Request messages refer to resources by means of URIs.

b) Resource manipulation

Representations of resources are transferred between client and server back and forth. A client has enough information about a resource to manipulate it properly and replaced the updated resource via request method PUT or delete it.

c) Self-descriptive messages

Messages, files, or data serializations in general are the more **self-descriptive** the less additional schema information is need to in order to process them. Such schemas include encoding information, delimiters for single values, mnemonic key names and even an internal (hierarchical) structure between single values.

> A data record with two values "René" and "1978" can be serialized in Java like this
>
> ```
> dataOutputStream.writeUTF("René");
> dataOutputStream.writeInt(1978);
> ```
>
> This produces a byte sequence which look like this in hex code:
> `00:05:52:65:6E:C3:A9:00:00:07:BA`.
>
> A (developer of a) receiver of this message has no chance to deserialize such a byte sequence without additional schema information. Such a byte sequence is the opposite of being self-descriptive.

[43] The term "preliminary" is from a business perspective, e.g., before a legal transaction took place. From a technical database point of view, it does not matter if a transaction contains preliminary or final data.

Another way to serialize such data is to put it into a XML or JSON document like this:

`{"name" : "René", "yearOfBirth" : 1978}`.

Here, encodings, delimiters and even meanings of the values are much clearer. A developer would know how to de such a message and how to process the values in a reasonable way.

Self-descriptive messages come with the price of a higher data volume because the meta data, i.e. keys, delimiters and structures expressed by brackets and colons consume space on their own.

d) Hypermedia as the Engine of Application State (HATEOAS)

Use cases of typical web applications are often implemented in processes that cause a sequence of requests and responses between clients and servers. Valid follow-up requests depend of the state the process currently is in.

In a web shop, at least one item has to be put in a shopping cart before the shopping cart can be checked out.

Hypermedia as the Engine of Application State (HATOAS) goes one step further than just self-descriptiveness. It includes links to other resources which are relevant in the current process state. Using the OPTIONS request method, a client can explore what can be done with these resources. In a way, HATOAS promotes the same properties for web APIs then what the World Wide Web promotes for web UIs, i.e., documents that are linked with each other. Clients, e.g. client developers, are supposed to discover a web API starting from an initial resource.

A POST request to a URI `/shoppingCarts/123/items` adds a new item to the shopping cart `123`. The response is supposed to include a link to the newly created item in that shopping cart like `/shoppingCarts/123/items/1` and to the shopping cart itself `/shoppingCarts/123`. A client could, e.g., remove that item from the shopping cart again by using the `DELETE` request method on `/shoppingCarts/123/items/1` or `POST`ing a `{ checkout: true }` to the `shoppingCarts/123` in order to start the checkout.

Note that we make use of the versatility of the request method `POST` which can be used for much more than just adding new elements to collections according to its official semantics. Also note that HATEOAS cannot completely relive clients from knowing about details what to do with resources, e.g., about valid messages in POST requests.

21.6.4 Cacheability

Cache directives are optional HTTP headers. REST promotes extensive use of them.

21.6.5 Layered System

In a layered system, a client should not care if it communicates with an origin server or some intermediate like a share cache or load balancer.

21.6.6 Richardson Maturity Model

As REST is just a set of principles which can either be adhered to or not, there is not binary choice of either having a RESTful API or not. The so-called Richardson Maturity Model [69] presents four levels of maturity w.r.t. REST principles. Martin Fowler provides a good discussion of this model in [70] which is just summarized in the following.

Level	Description
0 - Plain old XML	Devoid of REST principles. HTTP is used as a mere tunnel. There is no notion of resources, just a single endpoint URL that accepts only POST requests. The Simple Object Access Protocol (SOAP) or XML Remote Procedure Calls (XML-RPC) work that way.
1 - Resources	Adds the notion of resources with different URLs
2 - HTTP request methods	Uses appropriate requests methods for each request. In particular, it uses GET requests if applicable in order to profit from HTTP caching.
3 - HATEOAS	The topmost level requires that APIs are self-descriptive to the extent of HATEOAS.

Table 21: Levels of the Richardson Maturity Model

21.6.7 OpenAPI

The **OpenAPI Specification** [71] specifies a language for machine-readable interface files for RESTful APIs. These interface files are written in either YAML or JSON. They define the resource URL, request and response messages along with their respective headers and bodies.

As these interface files are machine-readable there is tooling available for creation of API mocks like Swagger UI as well as stubs and proxies for numerous languages.

21.7 GraphQL

In REST, the inner contents of resources and therefore responses are defined by the server side. However, API designers often do not exactly know about the needs of the API's clients, which is the normal case in the World Wide Web. So, clients have an **overfetching** issue, i.e., just use some data from the response and thus suffer from unnecessary additional server-side processing times and transmission times for the unused data. Conversely, client also have to deal with **underfetching** data, i.e., have to do multiple requests and wait multiple response times in order to gather the required data.

GraphQL [72] is a data query and manipulation language for HTTP-based APIs. It particularly tackles the problem from above. Bodies of request messages contain the exact needs of a client. They contain keys of those values a client is interested in.

This example is taken from the GraphQL playground at `https://graphqlbin.com/v2/6RQ6TM` . It demonstrates an API that serves facts from the Star Wars universe.

A request message body could look like this

```
query {
  Person(name: "Darth Vader") {
    id
    name
    birthyear
    films {
        title
    }
  }
}
```

which yields the response

```
{
  "data": {
    "Person": {
      "id": "cj0nv9pa9ewco0130v4ocnhed",
      "name": "Darth Vader",
      "birthYear": "41.9BBY",
      "films": [
        {
          "title": "A New Hope"
        },
        {
          "title": "Revenge of the Sith"
        },
        {
          "title": "Return of the Jedi"
        },
        {
          "title": "The Empire Strikes Back"
        }
      ]
    }
  }
}
```

Although REST is an architectural style and GraphQL a query language, both can be seen as competitors. GraphQL APIs are level-zero-APIs w.r.t. the Richardson Maturity Model (cf. Section 21.6.6) as there is typically just one URL endpoint and just POST requests. Because of that and because response messages also depend on the request message body and not just the request URL, HTTP caching is ineffective for GraphQL.

21.8 gRPC

gRPC Remote Procedure Calls (gRPC[44]) are even a step further away from REST principles. This standard uses a binary format for messages and an overall approach which resembles the Common **Object Request Broker Architecture (CORBA)** [73], a protocol for distributed systems that was widespread in the 1990s. In contrast to CORBA, gRPC is built on top of HTTP/2 instead of the Internet Inter-ORB Protocol (IIOP).

While REST promotes ease of use through self-descriptiveness of messages, gRPC favours small footprint and thus performance.

[44] This is a recursive acronym.

22 References

[1] B. Boehm, “Architecting: How Much and When?,” in *Making Software*, Sebastopol, CA 95472, O'Reilly, 2010, pp. 161-186.

[2] L. Bossavit, The Leprechauns of Software Engineering, leanPub, 2015.

[3] R. C. Martin, “Professionalism and Test-Driven Development,” *IEEE Software,* pp. 32-36, June 2007.

[4] “Hype Cycle,” 28 October 2019. [Online]. Available: https://en.wikipedia.org/wiki/Hype_cycle. [Accessed 28 October 2019].

[5] “Hype Cycle for Application Architecture and Development, 2019,” 28 October 2019. [Online]. Available: https://www.gartner.com/en/documents/3955980/hype-cycle-for-application-architecture-and-development-. [Accessed 28 October 2019].

[6] A. Silberschatz, P. B. Galvin and G. Gagne, Operating System Concepts, 9th edition, Hoboken; NJ 07030: Wiley, 2013.

[7] *Information technology – Open Systems Interconnection – Basic Reference Model: The basic model,* 1994.

[8] E. Evans, Domain-Driven Design. Tackling Complexity in the Heart of Software, Addison-Wesley, 2003.

[9] “IOPS,” 1 April 2019. [Online]. Available: https://en.wikipedia.org/wiki/IOPS.

[10] M. Kleppmann, Designing Data-intensive Applications, O'Reilly, 2017.

[11] I. Grigorik, High Performance Browser Networking (1 ed.), O'Reilly and Associates, 2013.

[12] M. Harchol-Balter, Performance Modeling and Design of Computer Systems: Queueing Theory in Action, Cambridge University Press, 2013.

[13] D. Oppenheimer, A. Ganapathi and D. A. Patterson, “Why Do Internet Services Fail, and What Can Be Done About It?,” in *4th USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[14] P. Bourgon, “Metrics, tracing, and logging,” 21 2 2017. [Online]. Available: https://peter.bourgon.org/blog/2017/02/21/metrics-tracing-and-logging.html.

[15] C. Lilienthal, Langlebige Software-Architekturen, dpunkt.verlag GmbH, 2017.

[16] “44 U.S. Code § 3542 - Definitions,” [Online]. Available: https://www.govinfo.gov/content/pkg/USCODE-2011-title44/pdf/USCODE-2011-title44-chap35-subchapIII-sec3542.pdf. [Accessed 25 June 2019].

[17] A. van der Stock, B. Glas, N. Smithline and T. Gigler, “The Open Web Application Security Project (OWASP),” 2017. [Online]. Available: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf. [Accessed 4 August 2019].

[18] B. Len, P. Clements and R. Kazman, Software Architecture in Practice, 3rd edition, Addison-Wesley Professional, 2012.

[19] “User Stories,” [Online]. Available: https://en.wikipedia.org/wiki/User_story.

[20] “Language Server Protocol,” [Online]. Available: https://langserver.org/. [Accessed 13 February 2020].

[21] M. Page-Jones, “Comparing Techniques by Means of Encapsulation and Connascence,” *Communications of the ACM - Special issue on analysis and modeling in software development,* pp. 147-151, 2 September 1992.

[22] “connascence.io,” 2 November 2019. [Online]. Available: https://connascence.io/. [Accessed 2 November 2019].

[23] E. Gamma, R. Helm, R. E. Johnson and J. Vlissides, Design Patterns. Elements of Reusable Object-Oriented Software., Prentice Hall, 1994.

[24] R. Potvin and J. Levenberg, “Why Google Stores Billions of Lines of Code in a Single Repository,” *Communications of the ACM,,* pp. 78-87, July 2016.

[25] C. Bird, “Conway's Corollary,” in *Making Software*, O'Reilly, 2010, pp. 187-205.

[26] M. Gorman, “Understanding The Linux Virtual Memory Manager,” 7 July 2007. [Online]. Available: https://www.kernel.org/doc/gorman/pdf/understand.pdf. [Accessed 8 March 2020].

[27] “Regular expression Denial of Service - ReDoS,” [Online]. Available: https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS. [Accessed 8 March 2020].

[28] M. Fowler, “Microservices and the First Law of Distributed Objects,” 13 August 2014. [Online]. Available: https://www.martinfowler.com/articles/distributed-objects-microservices.html. [Accessed 15 Feburary 2020].

[29] I. Fette and A. Melnikov, “Internet Engineering Task Force (IETF),” December 2011. [Online]. Available: https://tools.ietf.org/html/rfc6455. [Accessed 14 February 2020].

[30] M. Belshe, R. Peon and M. Thomson, “Hypertext Transfer Protocol Version 2 (HTTP/2),” May 2015. [Online]. Available: https://tools.ietf.org/html/rfc7540.

[31] “Apache Kafka,” 10 April 2019. [Online]. Available: https://kafka.apache.org/.

[32] J. Fulton, “Web Architecture 101,” November 2017. [Online]. Available: https://engineering.videoblocks.com/web-architecture-101-a3224e126947.

[33] “Elasticstack,” 10 April 2019. [Online]. Available: https://www.elastic.co/en/products/elasticsearch.

[34] “Memcached,” 10 April 2019. [Online]. Available: https://memcached.org/.

[35] “Redis,” 10 April 2019. [Online]. Available: https://redis.io/.

[36] “Apache ActiveMQ,” [Online]. Available: http://activemq.apache.org/. [Accessed 10 April 2019].

[37] “List of DNS record types,” 10 April 2019. [Online]. Available: https://en.wikipedia.org/wiki/List_of_DNS_record_types.

[38] R. Fielding, N. Nottingham and J. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Caching,” June 2014. [Online]. Available: https://tools.ietf.org/html/rfc7234.

[39] D. (. Hardt, “The OAuth 2.0 Authorization Framework,” October 2012. [Online]. Available: https://tools.ietf.org/html/rfc6749. [Accessed 31 December 2019].

[40] M. Jones, J. Bradley and N. Sakimura, “JSON Web Token (JWT),” May 2015. [Online]. Available: https://tools.ietf.org/html/rfc7519.

[41] “OpenID Connect Core 1.0 specification,” 8 11 2014. [Online]. Available: https://openid.net/specs/openid-connect-core-1_0.html. [Accessed 31 December 2019].

[42] “React,” Facebook, [Online]. Available: https://reactjs.org/. [Accessed 8 March 2020].

[43] “TypeScript,” Microsoft, [Online]. Available: https://www.typescriptlang.org/. [Accessed 8 March 2020].

[44] “NodeJS,” [Online]. Available: https://nodejs.org/en/. [Accessed 8 March 2020].

[45] P. Mell, Grance and Tim, “The NIST Definition of Cloud Computing,” September 2011. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-145.

[46] “Heroku.com,” [Online]. Available: https://heroku.com. [Accessed 12 December 2019].

[47] “Google App Engine,” 02 December 2019. [Online]. Available: https://cloud.google.com/appengine/. [Accessed 02 December 2019].

[48] “Google Compute Engine,” 02 December 2019. [Online]. Available: https://cloud.google.com/compute/. [Accessed 2019 December 2019].

[49] “Bin packing problem,” 10 April 2019. [Online]. Available: https://en.wikipedia.org/wiki/Bin_packing_problem.

[50] “Public Cloud Services Comparison,” 19 March 2019. [Online]. Available: http://comparecloud.in/.

[51] “AI winter,” [Online]. Available: https://en.wikipedia.org/wiki/AI_winter. [Accessed 10 April 2019].

[52] “Google Cloud Platform Documentation,” 10 April 2019. [Online]. Available: https://cloud.google.com/docs/.

[53] “Google Cloud Platform Tutorials,” 10 April 2019. [Online]. Available: https://cloud.google.com/docs/tutorials.

[54] “Cloud SDK - gcloud reference,” 10 April 2019. [Online]. Available: https://cloud.google.com/sdk/gcloud/reference/.

[55] A. M. Joy, “Performance comparison between Linux containers and virtual machines,” in *2015 International Conference on Advances in Computer Engineering and Applications*, Ghaziabad, India, 2015.

[56] “CVE-2016-5195,” 19 October 2016. [Online]. Available: https://access.redhat.com/security/cve/cve-2016-5195. [Accessed 10 June 2019].

[57] “Namespace manpage,” 6 3 2019. [Online]. Available: http://man7.org/linux/man-pages/man7/namespaces.7.html.

[58] “cgroups manpage,” 03 06 2019. [Online]. Available: http://man7.org/linux/man-pages/man7/cgroups.7.html.

[59] “containerd,” 3 June 2019. [Online]. Available: https://containerd.io.

[60] 8 5 2019. [Online]. Available: https://docs.docker.com/.

[61] 14 June 2019. [Online]. Available: https://kubernetes.io.

[62] R. Fielding and J. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing,” June 2014. [Online]. Available: https://tools.ietf.org/html/rfc7230.

[63] T. Berners-Lee, R. Fielding and L. Masinter, “Uniform Resource Identifier (URI): Generic Syntax,” January 2005. [Online]. Available: https://tools.ietf.org/html/rfc3986.

[64] I. Grigorik, “HTTP Caching,” [Online]. Available: https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching. [Accessed 26 June 2019].

[65] R. Fielding, M. Nottingham and J. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Caching,” June 2014. [Online]. Available: https://tools.ietf.org/html/rfc7234.

[66] R. Fielding, REST: Architectural Styles and the Design of Network-based Software, University of California, Irvine, 2000.

[67] D. Kristol and L. Montulli, “HTTP State Management Mechanism,” October 2000. [Online]. Available: https://tools.ietf.org/html/rfc2965. [Accessed 3 November 2019].

[68] W. C. Chan and E. Burns, “Java Servlet Specification,” 2017.

[69] L. Richardson, “Richardson Maturity Model,” 2009. [Online]. Available: https://www.crummy.com/writing/speaking/2008-QCon/act3.html.

[70] M. Fowler, “Richardson Maturity Model - steps toward the glory of REST,” 18 March 2010. [Online]. Available: https://martinfowler.com/articles/richardsonMaturityModel.html. [Accessed 15 February 2020].

[71] “OpenAPI Specification,” [Online]. Available: https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.2.md. [Accessed 26 June 2019].

[72] June 2018. [Online]. Available: https://graphql.github.io/graphql-spec/June2018/. [Accessed 27 June 2019].

[73] “Common Object Request Broker: Core Specification,” March 2004. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/04-03-12.pdf. [Accessed 25 June 2019].

[74] “ISO/IEC 9126,” March 2019. [Online]. Available: https://en.wikipedia.org/wiki/ISO/IEC_9126#Developments.

[75] M. Glinz, “A Risk-Based, Value-Oriented Approach to Quality Requirements,” *IEEE Software, Volume: 25 , Issue: 2,* March 2008.

[76] ISO, “ISO/IEC/IEEE 29148:2018 - Systems and software engineering -- Life cycle processes -- Requirements engineering,” International Organization for Standardization.

[77] T. Berners-Lee, R. Fielding and L. Masinter, “Uniform Resource Identifiers (URI): Generic Syntax,” August 1998. [Online]. Available: https://tools.ietf.org/html/rfc2396.

[78] R. Fielding and J. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content,” June 2014. [Online]. Available: https://tools.ietf.org/html/rfc7231.

[79] R. Fielding, M. Nottingham and J. Reschke, “Internet Engineering Task Force (IETF),” June 2014. [Online]. Available: https://tools.ietf.org/html/rfc7234.

[80] A. Rotem-Gal-Oz, “Fallacies of Distributed Computing Explained,” [Online]. Available: http://www.rgoarchitects.com/Files/fallacies.pdf. [Accessed 25 June 2019].

[81] A. Jacobs, “The Pathologies of Big Data,” *Commun. ACM 52,* pp. 36-44, 2009.

[82] C. Jackson, “Micro Frontends,” 19 June 2019. [Online]. Available: https://martinfowler.com/articles/micro-frontends.html. [Accessed 8 Feburary 2020].

[83] “Self-Contained Systems,” InnoQ, [Online]. Available: http://scs-architecture.org/index.html. [Accessed 8 Feburary 2020].

[84] “Tool Interface Standard (TIS) Portable Formats Specification,” May 1995. [Online]. Available: https://refspecs.linuxbase.org/elf/elf.pdf. [Accessed February 2020].

[85] “PE Format,” 26 August 2019. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format. [Accessed 13 February 2020].

[86] P. Webb and D. L. J. Syer, “Spring Boot Reference Guide,” 03 February 2020. [Online]. Available: https://docs.spring.io/spring-boot/docs/2.3.x/reference/htmlsingle/#server-properties. [Accessed 8 March 2020].