Advanced Software Engineering - Appunti

Francesco Lorenzoni

September 2023

Contents

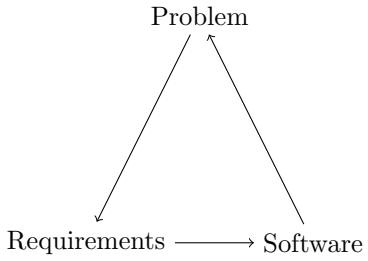
Chapter 1

Introduction

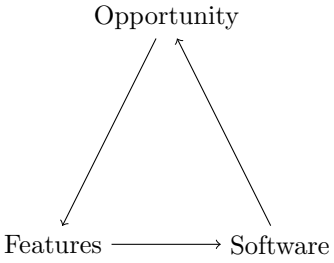
27 - Settembre

1.1 Product based

In *Project-based SE* there is loop which nowdays cripples software since its early stages of development. This is due to mutable nature of **requirements**, which often change throughout time along the features implemented by the software.



Product-based SE is opposed to *Project-based SE* and the above pictures changes as follows.



1.2 Agile

Agile is a collection of principles and methods applied in the software development field.

Opposed to project-based SE, in Agile the client is requested to express the requirements not in technical terms but in features.

Agile suggests an incremental development model

Principles

1. **Satisfy** customer through early and **continuous delivery** of valuable software
2. Welcome **changing requirement**, even late in development. Agile processes harness change for the customer's comptetitive advantage
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the **shorter timescale**

4. Business people and devs must **work together** daily throughout the prokect
5. Build projects around motivated **individuals** and give them the environment and support they need
6. The most efficient and effective method of conveying information to and within a dev team is **face-to-face conversation**
7. **Working software** is the primary measure of progress
8. Agile processes promote sustainable development
9. Continuous attention to technical excellence and good **design** enhances agility
10. **Simplicity** i.e. art of maximizing the amount of work not done is essential
11. The best architectures, requirements, and designs emerge from **self-organizing** teams.

Extreme Programming was proposed as part of the agile methodology.

1.3 Scrum

Since requirements changes are rather frequent, long-term plans are unreliable, hence SE aims to formulate short-term plans.

Scrum is found on **empiricism** and **lean thinking**; it asserts that knowledge comes from experience, and that decisions should be made on observations.

Other key terms are code **Transparency** among the team and with the customer, **Inspection** of produced code and software (artifacts), **Adaptation** to changes in features and requirements.

The **Scrum Team** is composed by:

1. **Product Owner**: must ensure that the dev team is always focused on the goal
2. **Scrum Master**: Scrum expert which drives the team to apply properly the Scrum framework.
3. **Developers**: actual ~~*monkeys*~~ people which write code

In scrum SW is developed in **sprints**, i.e. fixed-length periods with a specific goal to be achieved.

- ◇ Product backlog: to-do list of items to be implemented
- ◇ Timeboxed sprints
- ◇ Self-organizing teams

1.3.1 Product Backlog

Key point of the scrum methodology, it is a to-do list and its items are called **PBIs**[1]. It is **prioritized**, so that the items that be implemented first are at the top of the list

Product Backlog Activities

- ◇ **Refinement** Existing PBIs are analysed and refined to create more detailed PBIs. This may lead to the creation of new product backlog items.
- ◇ **Estimation** The team estimate the amount of work required to implement a PBI and add this assessment to each analysed PBI.
- ◇ **Creation** New items are added to the backlog. These may be new features suggested by the product manager, required feature changes, engineering improvements, or process activities such as the assessment of development tools that might be used.
- ◇ **Prioritization** The product backlog items are reordered to take new information and changed circumstances into account.

[1] *Product Backlog Item*

1.3.2 Timeboxed Sprints

Even if at the end of a srpint the goal hasn't been reached, "no worries", the work stops anyway; there will be a new sprint which will include the work which has not been implemented in the previous one.

1.3.3 Scrum Meetings

During a sprint, the team has daily meetings (*Scrums*) to **review** progress and to update the list of work items that are incomplete.

1.3.4 Agile suggested techniques

- **Test automation** As far as possible, product testing should be automated. You should develop a suite of executable tests that can be run at any time.
- **Continuous integration** Whenever anyone makes changes to the software components they are developing, these components should be immediately integrated with other components to create a system. This system should then be tested to check for unanticipated component interaction problems.

1.3.5 Sprint reviews

At the end of each sprint there is a review meeting which involves the *whole* team. The *product owner* has the ultimate authority to decide wether the sprint goal has been reached or not. The sprint review should include a process review, in which the whole team shares ideas on how to improve their way of working.

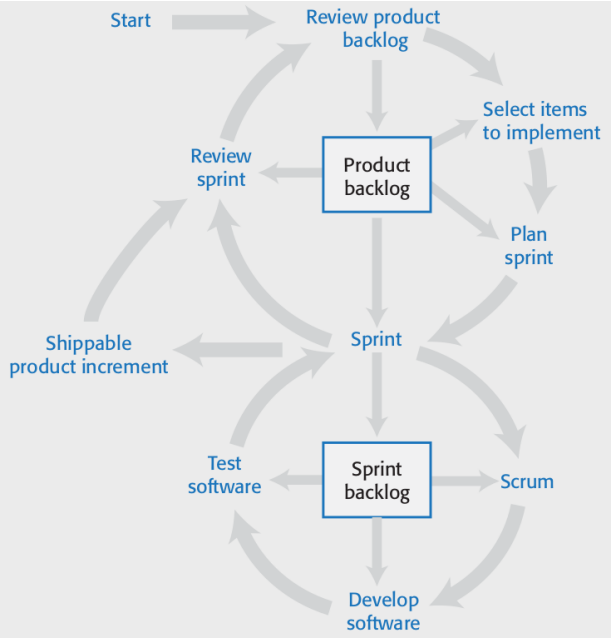


Figure 1.1: Scrum cycles

Chapter 2

Features, Scenarios, Stories

3 - Ottobre

Which factors drive the design of SW products?

- ◇ Inspiration
- ◇ Business/consumer needs not met by existing products
- ◇ Dissatisfaction with existing products
- ◇ Technical changes making new product types possible

Product-based software engineering needs less *requirements documentation* than project-based SE, since the requirements are not set by customers and it is allowed for them to change. The focus is instead on **features** (fragments of functionality); to understand which features are needed, we must first understand which may be **potential users**, through interviews, surveys, informal user analysis and consultation.

Flow-chart

User representations — **personas** — and natural language descriptions — **scenarios** and **stories** — help driving the identification of product **features**:

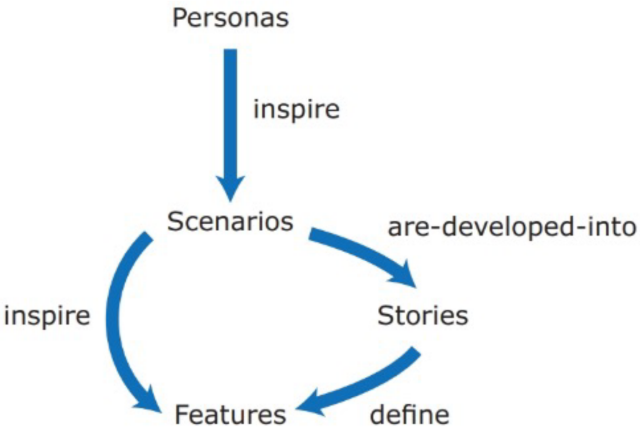
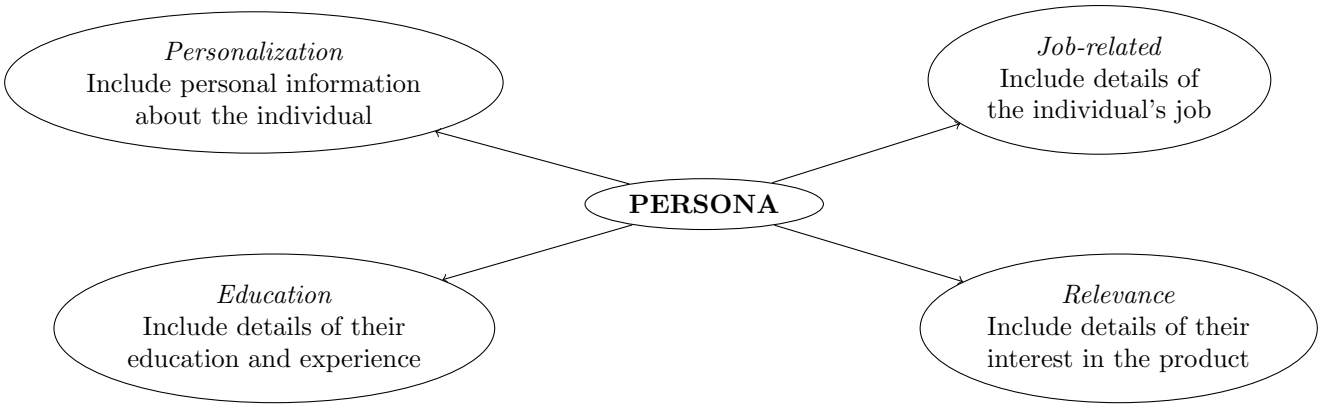


Figure 2.1: Features flowchart

2.1 Personas

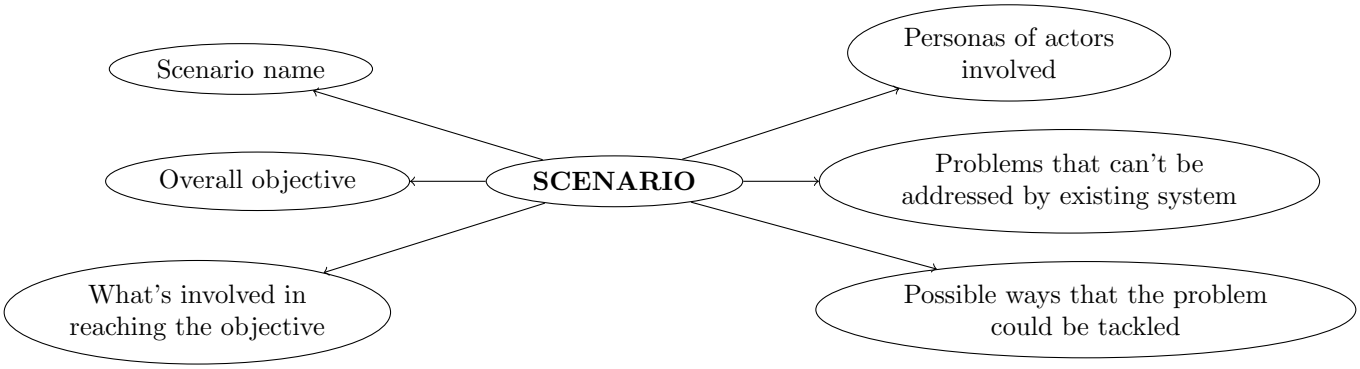
Personas represent the types of target users for our product. Each personas should highlight which are *background*, *skills* and *experience* of potential users. Usually only a couple of personas (max 5) are needed to identify ***key*** *product features.*



There are conflicting opinions about whether personas should include *photos* or not.
Photos may be misleading, since *"personas are not about how users look, but what they do"* (Steve Cable). *"Detailed personas encouraged the team to assume that demographic information drove motivations"* (Sara Wachter-Boettcher).

2.2 Scenarios

Having defined personas, to discover product features, it would aid to define *user interactions* with the product: a **scenario** is a narrative written from *user's perspective* describing a situation in which a user is using our product's features to do something she wants to do.
Scenarios are **not** *specifications*! They lack details and may be incomplete.



A proper amount of scenarios usually is 3-4 for each persona, aiming to cover the persona's main responsabilities. Each team member should create scenarios and discuss them with the rest of team and (possibly) users.

2.3 User Stories

As a	<role>
I want to	<do something>
So that	<reason/values>

Table 2.1: User Stories

Knowledge sources for feature design

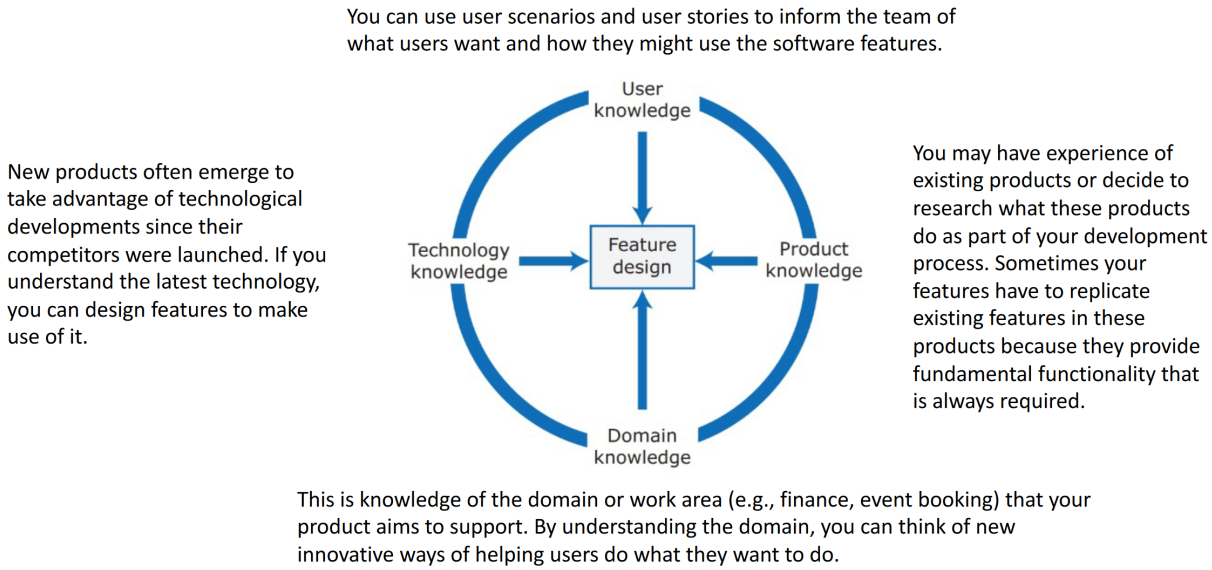


Figure 2.2: feature knowledge

While **scenarios** are high-level stories of product use, **User stories** are more fine-grained narratives. They allow to organize and chunk work into units which represent actual value to the customer, ultimately building software incrementally from the users perspective. Longer stories can be split into shorter stories, and to eventually prioritize them.
These words should recall *1. 3. 7. 10.* agile principles described in Section 1.2.
In fact, usually a *Scrum product backlog* is a set of *user stories* sorted according to priority.

Even though it is possible to express all functionalities described in a *scenario* using *user-stories*, scenarios can read more naturally, make stories understanding easier, and provide more context.

2.4 Feature identification

Our goal is to get a list of features that define our product, keeping in mind some **properties**:

- **Independence** → a feature should not depend on how other system features are implemented and should not be affected by the order of activation of other features
- **Coherence** → features should be linked to a single item of functionality. They should not do more than one thing and should never have side effects
- **Relevance** → system features should reflect the way users normally carry out some task. They should not offer obscure functionality that is rarely required.

To derive features from scenarios and user stories, the dev team should discuss these and start prototyping to demonstrate first novel and critical features.

2.4.1 Feature Creep

Number of product features grows as new potential users are envisaged. To avoid this, 4 questions should be considered:

1. Does this feature add something new or is it simply an alternative way of doing something already supported?
2. Can this feature be implemented by extending an existing feat rather than adding a new one?
3. Is this feat likely to be important and used by most software users?

4. Does this feat provide general functionality or is it a very specific feat?

Chapter 3

User Stories

Consider **TicTacToe** — aka *Tris* or *Tiro Filetto* — and its basic rules. Suppose you have to develop a software that allows playing TicTacToe, i.e. our *Product*.

Proceeding in steps, let's define the **Personas** who would use our product.

Chapter 4

Seminario - Imola informatica

4.1 Takeaway Messages

Performance per se is not an accurate measure, there are many factors when developing SW systems which affect perfomance, like usability and efficiency.

4.2 Project Path

$$\textbf{Demand} \longrightarrow \textbf{Plan} \longrightarrow \textbf{Design} \longrightarrow \textbf{Develop} \longrightarrow \textbf{Release} \tag{4.1}$$

This (sadly not) deprecated path 4.1 leads to *situation rooms* and subsequent performance degradation, unsatisfaction and possible skyrocketing costs.

Performance should drive the whole production process, it shouldn't be treated as a post-go live concern, otherwise it may lead to the so called *situation rooms*[1].

4.3 Fitness function

A **fitness function** provides a summarised measure of how close a given design solution is to achieving the set aims.

4.4 Performance Best practices

"Starbucks does not use two-phase commit": they aim to maximize throughput, by using an employee chain to serve customers, from ordering to delivering coffee.

Enforce business process performance with adequate fitness functions:

- ◇ involve key stakeholders
- ◇ automatically assess and evaluate
- ◇ continuously review and tune

It is important to design IT architectures and solutions with real-world requirements in mind. For example "a customer shouldn't have to wait for more than 2s to *order* a coffee".
In distributed architectures, network's technical aspects and metrics must be taken into account: latency, available

[1] Often named also *war rooms*

bandwidth, dedicated or shared, network billing models...
Aside from requirements, also costs, performance and observability should be kept in mind.
To measure progress fitness function must be fed periodically with real-time data. Most of the times testing only in production is the only way to go, since mirroring the production environment and using/managing it during development would be hugely costful. However, precisely for this reason, production testing shouldn't be the only testing method.

4.5 Bad habits

- ◇ Worrying about performance only late in development
- ◇ Last minute testing
- ◇ Focusing only on performance as a technical POV, not user/business pov

Enable a culture for performance across your entire value stream and embed it in business processes as well as IT systems.

Takeaway message

Evolutionary *architectures* need **fitness functions** and it is mandatory to continuosuly refine **fitness functions**.

Chapter 5

Software Architecture

12 - Ottobre

Architecture is the fundamental organization of a software system embodied in its *components* their relationships to each other and to the environment, and the principles guiding its design and evolution.

5.1 Component

A **component** is the element of implementing a coherent set of features; it can be seen as a collection of services, possibly used by other components, either directly or through an API.

Architectural design issues

- ◇ Non-functional product characteristics: security, reliability, availability... These aspects are as important as functional properties, to develop a successful product.
- ◇ Product lifetime: in case of developing a hopefully long-term product, its architecture must be able to evolve and adapt: *microservices*, for instance, easily allow scalability increasing the lifetime of our product.
- ◇ Software compatibility: Usually there may be legacy modules in the system, thus compatibility may be crucial, and it may lead to limiting architectural choices.
- ◇ Number of users: Releasing software on the internet truly complicates the prediction of the number of users for a product, it may vary a lot, thus the architecture must allow scaling up and down according to it.
- ◇ Software reuse: reusing components from other products or open-source software might save a lot of *time* and *effort*, however it may force some architectural design choices.

5.2 Non-functional quality attributes

1. Responsiveness *Does the system return results in reasonable time?*
2. Reliability *Do features behave as expected?*
3. Availability *Can the system deliver services when requested by the users?*
4. Security *Does the system protect itself and user data from attacks and intrusions?*
5. Usability *Are the users able to access (quickly) the features they need?*
6. Maintainability *Can the system be easily updated with undue costs?*
7. Resilience *Can the system recover in case of failure or intrusion?*

This typically aren't ~~*features*~~ *attributes* (?) implemented in the mid-development **prototypes**, they usually refer to the **final product**. Implementing these in the prototypes would increase too much the time taken to develop such

prototypes. Besides, note that optimizing one non-functional attribute might affect others, so, depending on our product and our resources, it must be considered whether to focus on one attribute instead of another one.

Security ⟹ *Performance*, *Usability*

Availability ⟹ *Time-to-Market*, *Simplicity*

5.2.1 Maintainability

For example let's consider which design choices would improve *maintainability*. First, recall that indicates how difficult and expensive is to make changes after the product release.
Two good practies are **decompose** the system into small self-containing parts and to avoid **shared data-structures**. Speaking of *shared data-structures*, a shared and *centralized* DB, might act as a bottleneck or, even-worse, as a single point of failure.
Using smaller local DBs for each **component** which later synchronize with the main one would avoid these two issues, however it introduces the need for **consistency** techniques and rules.

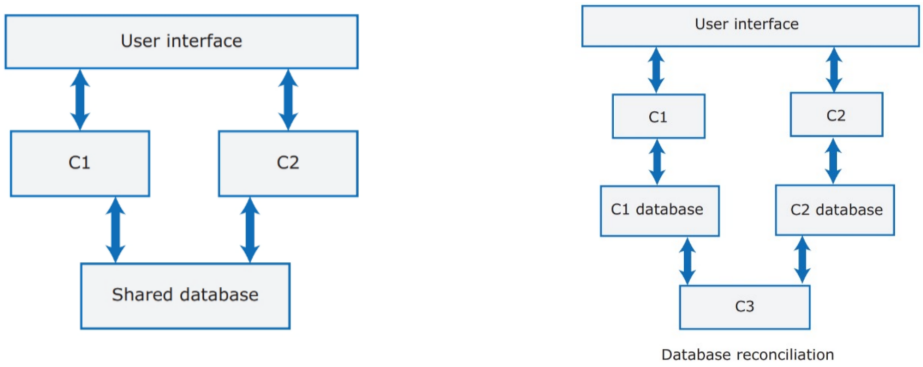


Figure 5.1: Centralized vs Component DBs

5.2.2 System Decomposition

Let's dig in deeper into system decomposition, by first introducing some definitions:

- ◇ **Service**: coherent unit of functionality
- ◇ **Component**: software unit offering one or more services
- ◇ **Module**: set of components

The agile manifesto suggests that: *Simplicity is essential*; regarding decomposition this is particularly true since as the number of components increases, the number of relationships betweeen them increases at a faster rate, thus it necessary to **manage complexity**; there are a few techniques to do so:

- ◇ **Separation of concerns**
- ◇ **Stable interfaces**
- ◇ **Implement once**

One way to implement this is to have a **layered architecture** where to each *layer* corresponds a *concern*, and the components within the same layer are independent and do not overlap in functionality. There are also some "concerns", which are in fact non-functional attributes like *security, performance* and *reliability*, which are *"cross-cutting"* i.e. they affect the whole system in a "vertical way" (see Fig 5.2.2) and they define the interaction between layers.

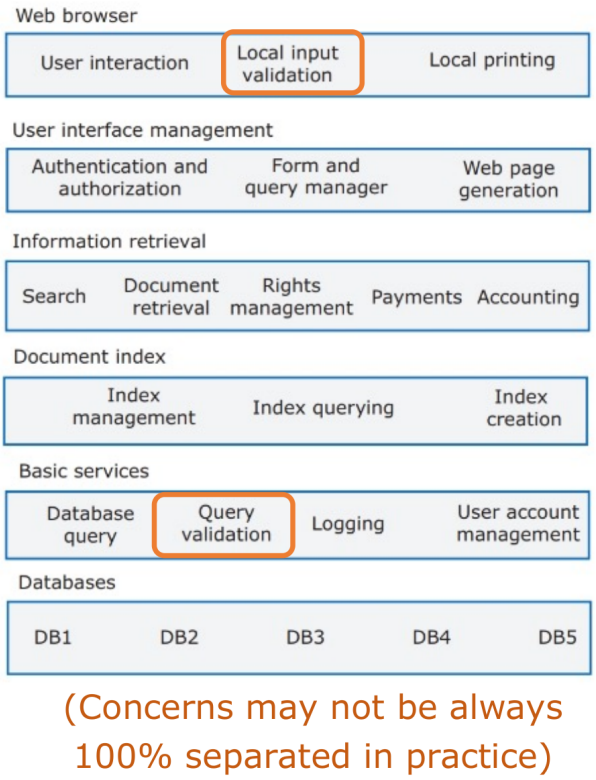


Figure 5.2: Layered architecture

System decomposition must be done in conjunction with choosing technologies for the system. Despite implying a mixup between implementation and design,it is necessary. For example, the choice of a particular type of DB affects components at higher levels, or choosing to support interfaces on mobile devices implies the need for mobile UI toolkits.

5.3 Distribution architecture

Now, how can we define servers and the allocation of components to servers?

A very common way is the **Client-Server architecture** aka Model-View controller where usually the communication between client and server happens with HTTP along with JSON (/XML). Client requests to a server are then muxed on many slave nodes which elaborate the requests.

There are some choices which must be made when designing the distribution architecture:

- **Data type and Data Updates**: whether the data should be centralized or spread around and later on synchronized.
- **Change frequency**: if frequent changes are foreseen it is advisable to isolate components as separate services to allow easy and uncostful changes
- **System execution platform**: the service being accessed over the internet or being a business system, leads to consistently different design architecture.

5.4 Technologies choices

It is difficult and costful to change technologies mid-development, thus it is important to the adequate considerations in advance and choose them properly. Let's consider first which aspects of the architecture are strongly technology-related:

- **Database** SQL $\overset{?}{\longleftrightarrow}$ noSQL

- ◇ **Platform** Mobile app $\overset{?}{\longleftrightarrow}$ web platform
- ◇ **Server** Dedicated in-house servers $\overset{?}{\longleftrightarrow}$ cloud
- ◇ **Open-source** Any suitable *open-source* solutions to be incorporated?
- ◇ **Development tools** Any limitations on the architecture imposed by the chosen development tools?

Chapter 6

Enterprise Applications

In Enterprise applications there heterogeneous services, data sources and participants, all connected via network.
In short, **Enterprise Applications** are *complex distributed multi-service applications* whose services must work together, them being suitably **integrated**.

6.1 Integration

The architectural question is how to integrate multiple different services to realize enterprise applications that are

- ◇ *coherent*
- ◇ *extensible*
- ◇ *maintainable*
- ◇ (reasonably) simple to *understand*

This is really what enterprise application integration was conceived for

- ◇ complexity management
- ◇ change management
- ◇ **pattern-based**

Pattern refers to high-level abstraction of accepted, reusable solutions to recurring problems.
When facing a problem, considering existing patterns that are applicable to solve such problem saves us from re-inventing the wheel and making the same mistakes as others

Typically, patterns are given in terms of

- ◇ **problem statement** including involved software components
- ◇ **context** including involved actors
- ◇ **forces** clarifying the problem rationale and importance
- ◇ **solution** given abstractly, and independent of its actual implementations

An **EIP** (*Enterprise Integration Pattern*) is a reusable abstraction of proven solutions to well-known problems raising while integrating the software components/services forming enterprise applicatioms.

6.2 Messages and Communication Means

A **message** is a discrete piece of data sent from a service to another, typically structured into header and body, and generally sent through **one-way channels**. Thus by itself the communication is *asynchronous*, but it can be made *synchronous* by duplicating a channel and thus creating a bidirectional communication.

Channels supports extends to two main categories:

- ◇ **Point-to-Point** $(1:1)$ channels ensure that only one receiver will receive a given message
- ◇ **Publish-Subscribe** $(1:N)$ channels deliver a copy of the message to each receiver

Channels generally require the data to serialized in some way: to this extent **adapters** "translate" (*adapt*) application-specific data into a format suitable to be sent; note that adapters are place between the *application* and the *channel*, not the *receiver*.
Besides, a **message endpoint** for each node/application is required, to handle channel connections and queue/handling the messages received before letting the application process them.

Message endpoints and **channels** provide the most trivial integration possible, however they do not solve the problem when the *receiver application* requires the data to be in a specific format e.g. `JSON`. **Message translators** are intermediary entities which can translate and eventually filter data.

6.3 Deeper message integration

Some problems are still not solved e.g. message *routing*, *splitting*, *aggregating*, etc.

That's where **pipes** and **filters** architecture style comes in.
Messages travel through many *filters* (and components) processing them. Components flush messages into **pipes** they are connected to.
Clearly this is (again) a flexible abstraction which can adapt to circumstances and environment.

This architecture, along with endpoints and channels, includes **Content Enrichers**, which add contextual information which the source may not have, **Routers** which may be *content-based* (header/body) or *context-based* (testing/production env).
Speaking of **routers**, such components are connected to multiple channels and contain the logic to discern which is the correct channel onto which a message should be sent.

6.3.1 Composite Patterns

Patterns may be composed to build other patterns. **Normalizers**, for instance, enable data received from different sources to be normalized and then sent in a proper format to receivers; behind the curtain, a *normalizer* is nothing more than a router and a set of translators.

6.3.2 Parallelism

Sometimes some integration steps may be computed in **parallel** and then results from such processes may be **aggregated** to decide which actions to perform.

Splitters break out composite messages into a series of individual messages, which can be *processed independently*.
Aggregators collect and store individual messages until a complete set of related messages has been received, ultimately publishing a single message which be processed as a whole.

6.4 Handling Problems

- ◇ **Validate** messages
- ◇ **Architectural smells** and **refactoring**
- ◇ **Security** issues
- ◇ **Isolate** features in integrated applications
- ◇ **Deploy serverlessly** integrated applications

Chapter 7

Cloud computing

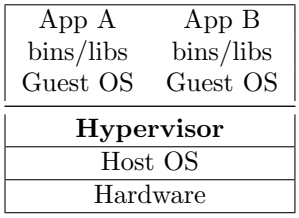
19 - Ottobre

Powerful hardware and high-speed networking have made room for the development of cloud computing. **Cloud computing** allows virtual resources accessible **on demand**, and provides many advantages against the standard old-style scaling methods i.e. buying resources.

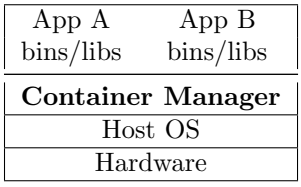
- ◇ *Scalability*
- ◇ *Elasticity*
- ◇ *Resilience*
- ◇ *Cost*

7.1 Virtualization and Containers

Virtual machines on a single physical machine are managed by hypervisor



Containers instead exclude one layer of abstraction, saving up a lot of resources



7.2 Docker

Docker exploits container-based virtualization to run multiple isolated guest instances on the same OS. Software is packaged into **images** which are read-only templates to instantiate and run containers. External **volumes** can be mounted to ensure data persistance when used by multiple containers or by the host machine.

It is possible to **stack** multiple docker *images*, and if desired create a new image as a result of stacking other ones.

Chapter 8

* as a Service

In traditionally distributed software, customers had to configure, manage updates, while producers had to mantain different product versions; in **Saas** instead, a product is delivered as a **service**, thus there's no need to install anything, and there is a form of monthly (and/or pay-per-usage) subscription, to use the service.

8.1 Benefits and cons

***Producer** point of view*

Pros

- ◇ Regular cash flow
- ◇ Easier and less costful update management
- ◇ Continuous deployment: a new software version can be deployed as soon as it is tested
- ◇ Payment flexibility, making room for different subscription plans possibly attracting a wider range of users
- ◇ Try-before-you-buy options are easily available without fearing piracy, besides they'd make the product look more appealing to new customers
- ◇ Telemetry and data collection are way easier and hardly avoidable by customers

***Consumer** point of view*

Pros

- ◇ Mobile, laptop and desktop access
- ◇ No **upfront costs** for software or servers
- ◇ Immediate and transparent software updates
- ◇ Reduced software **management costs**

Cons

- ◇ **Privacy** regulation conformance
- ◇ **Security** concerns
- ◇ Network constraints may limit the usability of the software
- ◇ Data exchange from/to services might be difficult if the service doesn't provide a suitable API
- ◇ No control over updates
- ◇ Service **lock-in**

8.2 Design issues

When designing Saas there are some critical points the developers has to make decisions on:

- ◇ **Authentication** method: federated auth, personal auth, *Google/Linkedin/...* credentials...

- ◇ Whether some features should be available **locally**
- ◇ **Info leakage**
- ◇ *Multi-tenant* vs *multi-instance* **DB management**: i.e. single repository vs separate copies of system and database

8.2.1 DB management

Multi-tenant

Multi-tenant systems forsee a single DB schema shared by all system's users, where DB items are tagged with a *tenant identifier* to provide some form of "logical isolation".

Mid-size and large businesses rarely want to use a generic multi-tenant software, they often prefer a customized version adapted to their own requirements. Generally it is of interest to have a custom UI mutating according to the user, along with custom optional fields accessible only by some classes of users. In case there's the need to expand the DB schema to achieve such results, **storage waste** becomes a major concern.

Security is the major concern of corporate customers with multi-tenant systems, since a centralized DB may represent a single point-of-failure for data leak or damage.
A common solution is to implement **multilevel access control**, checking data access both at the organizational level and at individual level. A technlogy which can clearly help in this matter is **encryption**, however it might cripple performance.

Multi-instance

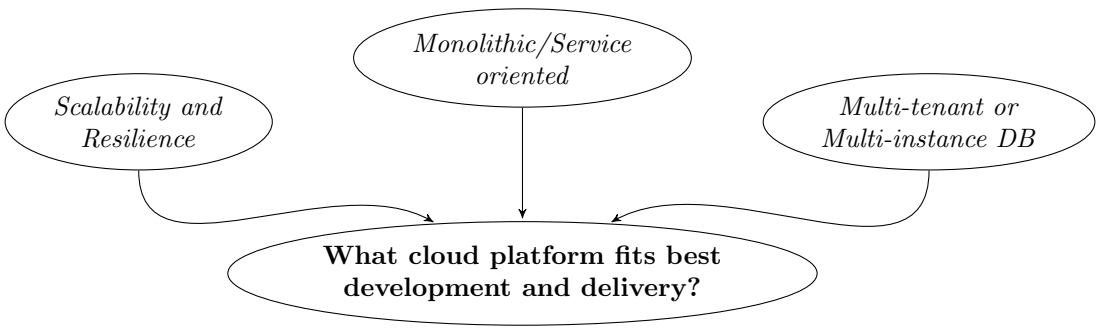
Multi-instance may mainly be **VM**-based or **Container**-based.
With *containers*, each user has an **isolated** version of software and database running in a set of container, defining a solution perfect for products whose users work onto independently from others. Besides, since there is no direct sharing of a single data structure, many security aspects are easy to manage.
With the other solution instead, for each **customer** there is a VM running an instance of the DB **shared** and accessible to all customer's users.

Let's consider *pros and cons* of such a solution. Flexibility, security, scalability and resilience are clearly some key points of *multi-instance DBs*. However update management difficulty and cloud VMs renting costs are not negligible.

Wrapping up, there are three possible ways of providing a customer **database** in a *cloud-based* system:

1. As a **multi-tenant** system, *shared by all customers* for your product. This may be hosted in the cloud using large, powerful servers.
2. As a **multi-instance** system, with *each customer database* running on its own *virtual machine*.
3. As a **multi-instance** system, with *each database* running in its own *container*. The customer database may be distributed over several containers.

8.3 Architectural decisions



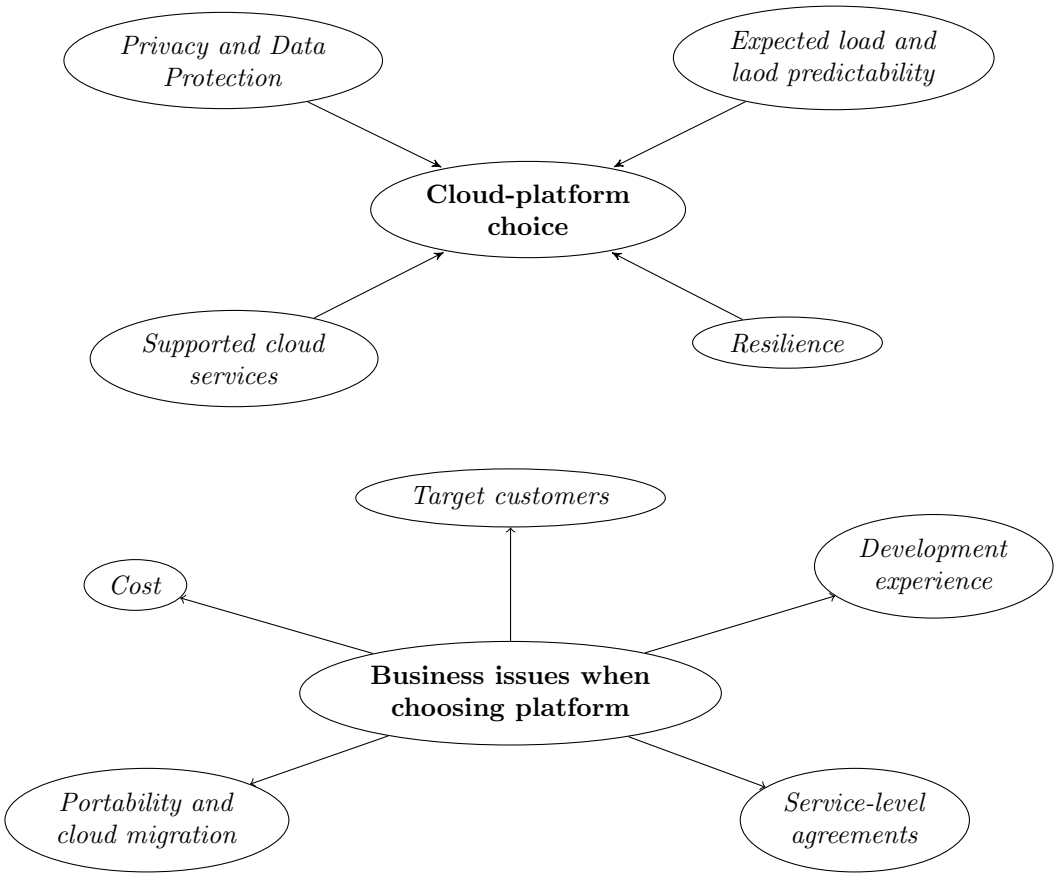
8.3.1 Scalability

To allow scalability on cloud-based systems, the product implementation must be organized so that the individual software components can be replicated and run in parallel, besides, also a load-balancing mechanism must be implemented.

8.3.2 Resilience

Achieving resilience can be done through *hot/cool standby*. The difference lies in the fact that while cool standby relies restarting the system using backup copies of the data, hot standby forsees a clone-instance running at the same time of the main one with a mirrored DB: in case of system failure, there is and entire backup system which can take its place while the main recovers. This is clearly more costful, but more effective.

8.4 Choosing cloud platform



Chapter 9

Kubernetes

Kubernetes takes into consideration the following questions:

- ◇ What happens if a container crashes?
- ◇ What happens if the machine running a container fails?
- ◇ How to handle communication between multiple containers how to enable a network between them?
- ◇ In a multi-machine system, which one should run a container?

The answer is **container orchestration**, implemented by the de facto standard *Kubernetes*:
K8s manages the entire **lifecycle** of individual containers, *spinning up* and *shutting down* resources as needed, e.g. if a container shuts down unexpectedly, K8s reacts by launching another container in its place.
K8s provides a mechanism for applications to **communicate** with each other even as underlying individual containers are created and destroyed.
Given a set of container ***workloads***[1] to run and a set of machines on a cluster, the container orchestrator examines each container and **determines** the **optimal machine** to schedule that workload.

9.1 Design Principles

One of the key points of K8s is the ***declarativeness***: it allows us to simply define the **desidered state** of our system, and it will automatically detect and intervene in case the state doesn't meet the specified needs.

Such desired state is defined as a collection of **objects**:

- ◇ each object has a specification in which you provide the desired state and a status which reflects the current state of the object
- ◇ K8s constantly polls each object to ensure that its status is equal to the specification
- ◇ if an object is unresponsive, K8s will spin up a new version to replace it
- ◇ if a object's status has drifted from the specification, K8s will issue the necessary commands to drive that object back to its desired state

K8s integrates perfectly with microservice-based architecture and with the principle of ***decoupling***, i.e. decomposing a system into smaller *decoupled* services which can be scaled and updated independently.

K8s is designed in a way which implies that to get the most from containers and container orchestration, deploying ***immutable infrastructure*** should be preferred. *Immutable*, even if it seems odd, indicates that containers shouldn't be too complicate ar alter their state, and that the user shouldn't, for example, log into a container and change libraries, code etc. Instead, since containers are by nature **ephimeral**, once there's an update, a new container image should be built, instantiated and replace the old one. This also allows easy roll-back by simply going back to a previous container image.

[1] Rings a bell? Discussed in IRA's course when talking about **network microsegmentation**

9.2 K8s Objects

9.2.1 Pod

Pods consists of one or more (tightly related!) *containers*, a shared *networking layer* and shared *filesystem volumes*.

9.2.2 Deployment

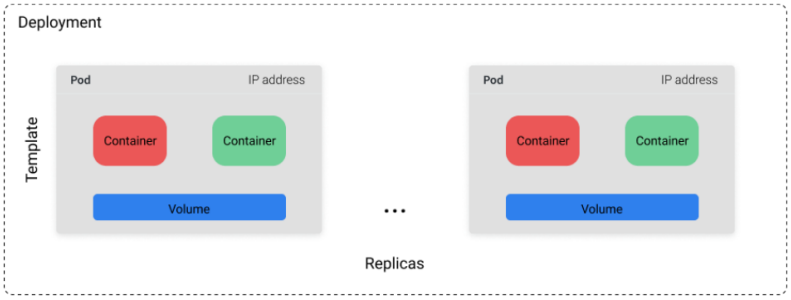


Figure 9.1: Deployment Object

A **Deployment** object includes a collection of Pods defined by a template and a *replica count*, indicating how many copies of the template should be running.

The cluster will always try to have n Pods available e.g. if we define a deployment with a replica count of 10 and 3 of those Pods crash, 3 more Pods will be scheduled to run on a different machine in the cluster.

9.2.3 Service

Each Pod is assigned a **unique IP** address that we can use to communicate with it; a legit question may arise: How to communicate with Pods if the set of Pods running as part of the Deployment can change at any time?

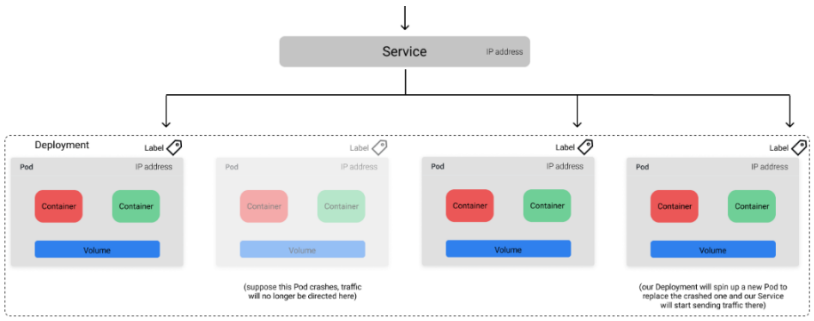


Figure 9.2: Service Object

K8s **Service** object provides a stable endpoint to direct traffic to the desired Pods even as the exact underlying Pods change due to updates/scaling/failures.
Services know which Pods they should send traffic to based on labels ($\langle key\text{-}value, pairs \rangle$) which we define in the Pod *metadata*.

9.2.4 Ingress

Service objects allows us to **expose** applications behind a stable endpoint only available to internal cluster traffic;
To expose our application to traffic *external* to our cluster, we need to define an **Ingress** object, which allows to select which Services to make publicly available.

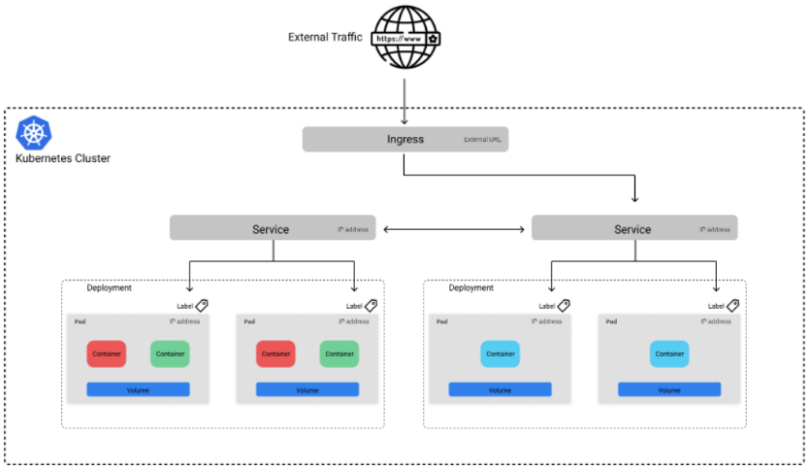


Figure 9.3: Ingress Object

Other objects - Not discussed

9.3 Control Plane

Two types of machines in a cluster:

1. **master node** - (often single) machine that contains most of the control plane components
2. **worker node** - machine that runs the application workloads

Let's dig into the components of both node types.

9.3.1 Master node

User provides new/updated object specification to **API server** of master node, which validates update requests and acts as unified interface for questions about cluster's *current state*, stored in a distributed key-value store `etcd`

The **scheduler** determines where objects should be run

- ◇ asks the API server which objects haven't been assigned to a machine
- ◇ determines which machines those objects should be assigned to
- ◇ replies back to the API server to reflect this assignment

The **controller-manager** monitors cluster state through the API server, and in case the actual state differs from desired state, the controller-manager will make changes via the API server to drive the cluster towards the desired state.

9.3.2 Worker node

The **kubelet** act a node's "agent" which communicates with the API server to see which container workloads have been assigned to the node. It responsible for spinning up pods to run these assigned workloads, and for announcing — when a node first joins the cluster — a node's existence to the API server, so that the scheduler can assign pods to it.

kube-proxy enables containers to communicate with each other across the various nodes on the cluster.

Besides these two components there are only **Pods** left, discussed earlies.

9.4 Concluding remarks

When should you ***not*** use *K8s*?

- ◇ If you can run your workload on a single machine
- ◇ If your compute needs are light
- ◇ If you don't need high availability and can tolerate downtime
- ◇ If you don't envision making a lot of changes to your deployed services
- ◇ If you have a monolith and don't plan to break it into microservices

Besides this, a simpler yet less powerful alternative is *Docker-swarm*, usually preferred in environments where simplicity and fast development are prioritized.

Chapter 10

Microservices

We will discuss how to decompose non-trivial systems into **components**, but first let's consider the advantages of components.

Components can be developed in **parallel** by different teams, and can be **reused**, **replaced**, **distributed** across multiple computers.
However, note also that to be effective components must be easily replicated, run in parallel, and migrated, thus allowing to correctly exploit **cloud-based** scalability, reliability, and elasticity. A simple yet powerful to design components which respect such requirements, is to use **stateless** services that maintain persistent information in a local db.

10.1 Software service

Software service

- ◇ *Definition* $\longrightarrow$ Software component that can be accessed over the Internet
- ◇ Given an *input*, a service produces a corresponding *output*, without **side effects**.
- ◇ Service is accessed through its published interface indipendently from its implementation details, which are hidden.
- ◇ Services do **not** maintain any **internal state**.
- ◇ **State information** is either stored in a **database** or maintained by the service requestor.
- ◇ State information can be included in service request, updated state info in service result.
- ◇ Services can be *dynamically reallocated* from one virtual server to another, enancing scalability

Amazon rethinked what a service should be

- ◇ a single service should be related to a *single business function*.
- ◇ services should be completely **independent**, with their **own database**.
  This is completely opposed to the previous way of implementing DBs, since a **unique DB** was usually exploited as an integration solution between heterogeneuous components/services.
- ◇ Each service should manage its own user interface
- ◇ It must be possible to **replace** or **replicate** a service without changing other services

From here we get to **microservices**, i.e. small-scale stateless services that have a single responsibility.

10.2 Example - Auth system

Let's consider a system using an authentication module providing:

1. user registration

2. authentication using UID/password
3. two-factor authentication
4. user information management
5. password reset

Below are listed some ideas to identify which microservices might be used for the authentication system:

1. Break coarse-grain features into more detailed functions
2. Look at the data used and identify a microservice for each logical data item to be managed
3. Minimize amount of replicated data management

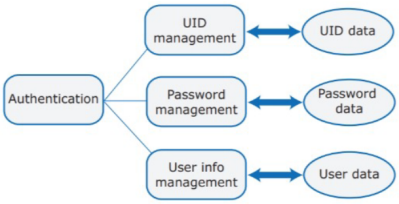


Figure 10.1: Result of microservices identification in our authentication example

10.3 Microservices - Key Points

Microservices

- ◇ **Small-scale**[1]services that can be combined to create applications
- ◇ **Independent**, i.e. service interface must not be affected by changes to other services
- ◇ Possible to modify and re-deploy service **without changing/stopping** other services

More specifically, we can list more specific charateristics on how microservices should be:

- ◇ *Self-contained*
- ◇ *Lightweight*
- ◇ *Implementation independent*
- ◇ *Independently deployable*
- ◇ *Business-oriented*

When speaking of services "size" two measures come in handy:

1. **Coupling** measures number of *inter*-component relationships.
   *Low **coupling** ⟶ independent services, independent updates*
   idea ⟶ if two services interact too much, then they should have be unified in a single service.
2. **Cohesion** measures number of *intra*-component relationships.
   *High **cohesion** ⟶ less inter-service communication overhead*
   idea ⟶ if a service intracommunicates too much with itself, then it should be split into smaller services.

The key principle which the two measures aim to address is the "Single responsibility principle", which states that each service should do one thing only and should do it well.

Responsibility ≠ single functional activity

Another way to determine how "big" should a microservice be is the *"Rule of twos"*, stating that a Service can be developed, tested, and deployed in two weeks by a team which can be fed with two large pizzas (8-10 people). Many people are required for a single service for various reasons —e.g. developing, decoupling, testing, etc.— but the most groundbreaking one is that *services should be **supported** and maintained **after** deployment*

[1]Actually sometimes they may be not so small

10.4 Motivations

1. Accelerated rebuild and redeployment for **shorten lead time** for *new features* and *updates*
2. **Scale**, effectively

Microservices architecture perfectly assesses these two points, since each microservice can be deployed in a separate container, allowing for quick **stop/restart** without affecting other services and for quick deployment of service **replicas**.

10.5 Design decisions

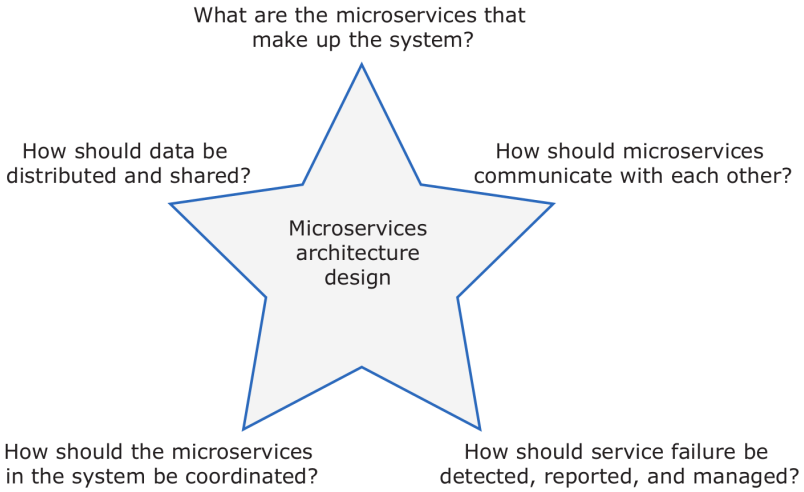


Figure 10.1: Microservices Decisions

How to decompose system into a set of microservices? They should be not too many (low cohesion $\longrightarrow$ communication overhead) and also not too few (high coupling $\longrightarrow$ less independency for updates/deployment/..)

Understanding how to decompose is not a trivial task:

- ◇ Balance fine-grain functionality and system performance
- ◇ Follow the "common closure principle" (elements likely to be changed at the same time should stay in same service)
- ◇ Associate services with business capabilities
- ◇ Services should have access only the data they need (+ data propagation mechanisms)

10.6 Service Communications

Ideally each microservice should manage its *own data*, but this arises the problem of dealing with possible **data dependencies**.
To avoid major issues, data sharing should be as little as possible and should mostly be *read-only*, with few services responsible for data updates. Besides, it is advisable to include a mechanism to keep **consistent db copies** used by replicated services.

Shared database architectures employ `ACID`[2] transactions to serialize updates and avoid inconsistency. In distributed systems we must trade-off data **consistency** and **performance**, hence microservices systems must be designed to **tolerate** some degree of data **inconsistency**.

1. *Dependent data inconsistency*
   Actions/failures of one service can cause data managed by another service to become inconsistent

[2] *Atomicity Consistency Isolation Durability*

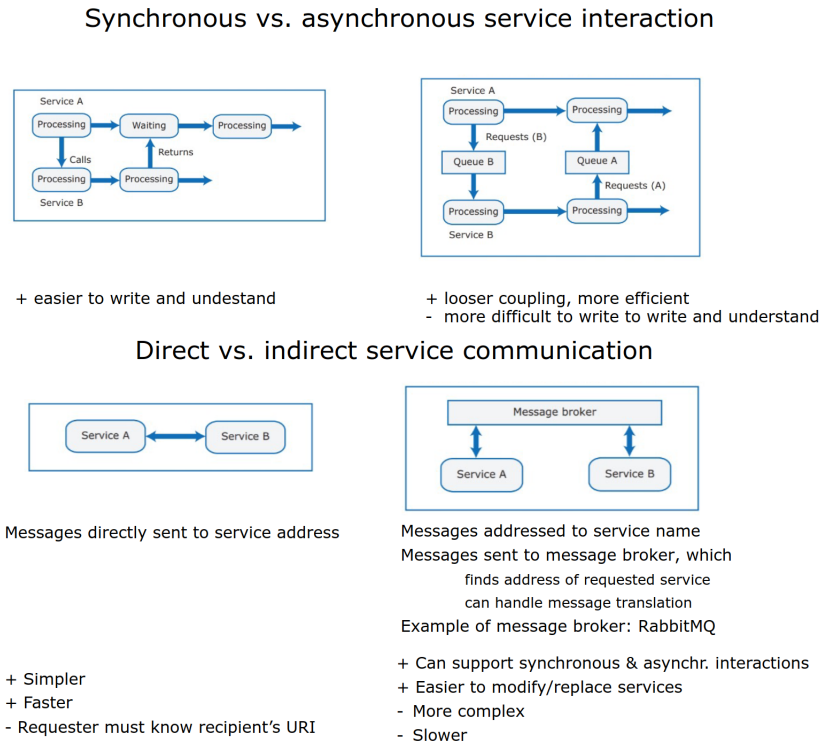


Figure 10.2: Microservices communication

2. *Replica inconsistency*
   Several replicas of the same service may be executing concurrently, each one updating its own db copy. Thus, we need to make these dbs *"eventually consistent"*[3]

10.6.1 CAP and Saga

CAP Theorem

It is **impossible** for a web service to provide *Consistency*, *Availability* and *Partition-tolerance* at the same time

In presence of a *network Partition*, you cannot have both *Availability* and *Consistency*

1. **Consistency**: any read operation that begins after a write operation must return that value, or the result of a later write operation
2. **Availability**: every request received from a non-failing node must result in a response
3. **Partition-tolerance**: services can be partitioned into multiple groups and network can delay lose arbitrarily many messages among services

The **Saga pattern** provides a possible solution to handle consistency between transactions.

Implement each business transaction that spans between multiple services as a **saga**, i.e. a sequence of local transactions. Each local transaction updates a database and triggers next local transaction(s) in the saga; in case such local transaction fails then the saga executes a series of **compensating transactions**.

***Coordinating** sagas*

1. *Choreography*: each local transaction publishes event that triggers next local transaction(s)
2. *Orchestration*: an orchestrator tells participants which local transactions to execute

[3] i.e."Sooner or later will be consistent", which is not "*maybe* consistent *maybe not*"

1. *Backward model*: **undo** changes made by previously executed local transactions
2. *Forward model*: "**retry** later"

10.6.2 Netflix Approach

Netflix eventual consistency is backed up by *Apache Cassandra*: in a nutshell, to replicate data in n nodes: "write to the ones you can get to, then fix it up afterwards"; A quorum is used as threshold, e.g. $(n/2+1)$ of the replicas must respond.

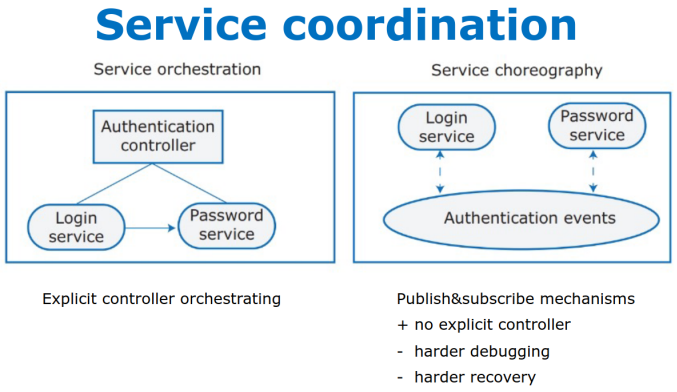


Figure 10.3: Netflix approach on Coordination

10.6.3 Failure management

Something will **unavoidably** go wrong, regardless of everything. A system must be able to cope with failures.

Consider for instance a service S invoking two other services A and B which guarantee 99% availability, then:

$$downtime(S) = 30\frac{min}{\textbf{day}}$$

30 minutes per day of **downtime** is a lot!

1. **Internal** failure: Conditions detected by a service and which can be reported to the *requestor* service through an error message for instance.
   e.g. invalid link/resource not found.
2. **External** failure: External cause which affects the availability of a service, possibly leading to its unresponsiveness.
3. **Performance** failure: Performance has degregated to an unacceptable level.

One way to cope with failures is to exploit **Circuit breakers** (Fig 10.4), which are based on timeouts to compensate unresponsiveness of a requested service.
Another way is to "bravely" test using the **Chaos Monkey** paradigm, which causes at a given rate random failures in the tested system.

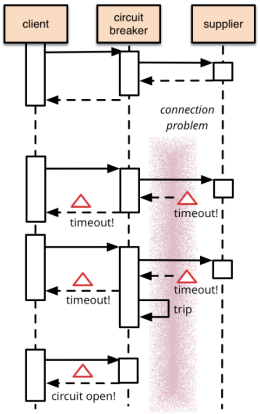


Figure 10.4: Circuit breaker

10.7 RESTful services

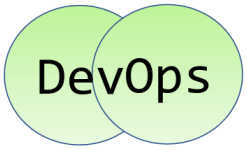
*RE*presentational *S*tate *T*ransfer (**REST**) was originally introduced as an architectural style, and then it has developed as an abstract model of the Web architecture to guide the redesign and definition of HTTP and URIs.

"a network of Web pages forms a virtual state machine, with each action resulting in a transition to the next state of the application by transferring a representation of that state to the user"

***REST** principles*

1. *Resource identification through URIs*:
   i. Service exposes set of resources identified by `URI`s
2. *Uniform interface*:
   i. Clients invoke `HTTP` methods to *create/read/update/delete* resources:
   ii. `POST` and `PUT` to create and update state of resource
   iii. `DELETE` to delete a resource
   iv. `GET` to retrieve current state of a resource
3. *Self-descriptive messages*:
   i. Requests contain enough context information to process message
   ii. Resources decoupled from their representation so that content can be accessed in a variety of formats (e.g., `HTML`, `XML`, `JSON`, `plain text`, `PDF`, `JPEG`, etc.)
4. *Stateful interactions through hyperlinks*:
   i. Every interaction with a resource is stateless
   ii. Server contains no client state, any session state is held on the client
   iii. Stateful interactions rely on the concept of explicit state transfer

10.8 DevOps



Same team responsible for service
development, **deployment** and **managements**.

However, regardless of how intense it is, testing *cannot* prevent 100% of unanticipated problems, thus it is mandatory to **monitor** the deployed services. Heavy monitoring may affect the performance of services and the overall

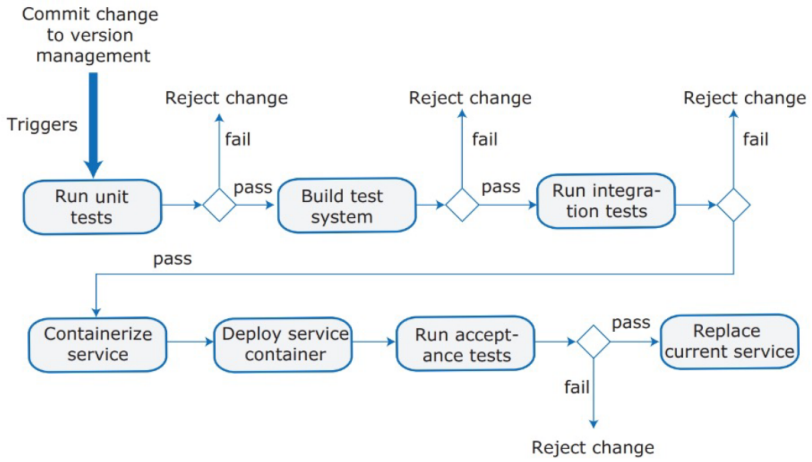


Figure 10.5: Continuous deployment pipeline

usability of a product, thus it must be sufficient to meet our needs and not more.
Monitoring allows to detect the failure of a service and **rollback** if needed. When introducing a *new* version of a service, you maintain the *old* version, changing only the "current version link" to point at the new service, remaining able to revert it if needed.

10.9 Concluding Remarks

Microservices - Pros

- Shorter lead time
- Effective scaling

Microservices - Cons

- Communication overhead
- Complexity
- "Wrong cuts"
- "Avoiding data duplication as much as possible while keeping microservices in isolation is one of the biggest challenges"

"A poor team will always create a poor system"

Don't even consider microservices unless you have a system that's too complex to manage as a monolith

[M. Fowler]

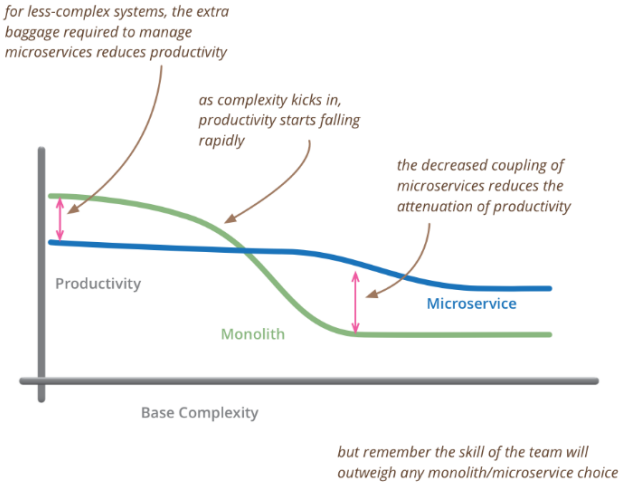


Figure 10.6: Takeaway message

Chapter 11

Architectural Smells and Refactorings

11.1 Definition

A review of white and grey literature aimed at a determining the most recognised **architectural smells**[1] for *microservices* and the architectural **refactorings** to resolve them. In other terms, the problem is for a system designer to understand whether the designed infrastucture respects the microservices principles, and, in case it doesn't, how to refactor it according to such principles.

11.2 Design Principles

1. **Independent deployability**: Microservices forming an application should be indipendently deployable
2. **Horizontal scalability**: They should also be horizontally scalable
3. **Isolation of failures**: Failures should be isolated to avoid cascaded side-effects
4. **Decentralization**: Decentralization should occur in all aspects of microservice-based applications, from data management to governance

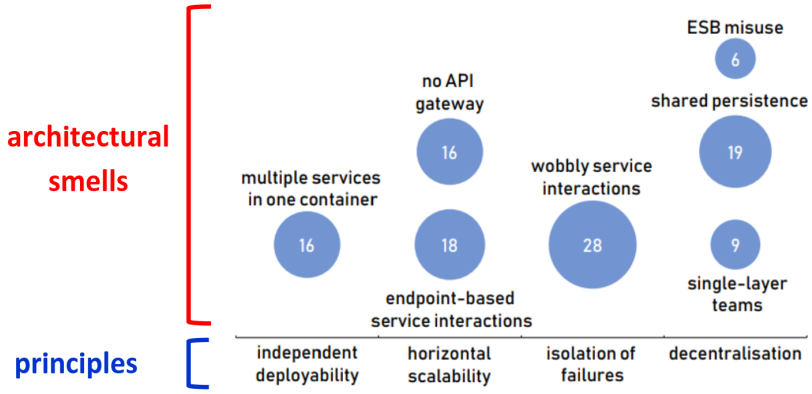


Figure 11.1: Smells Principles
Here we can see smells, i.e. wrong choices, and the principle they violate

[1] i.e. the *"smell"* is caused by an ***unproper*** architectural design choice

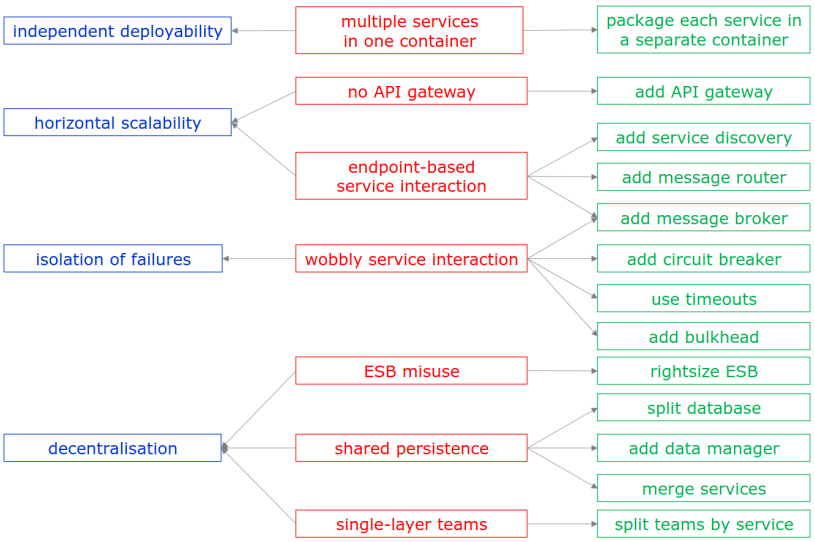


Figure 11.2: Solutions

Here instead, we see also which are the design fixes we can implement to eliminate the bad smells.

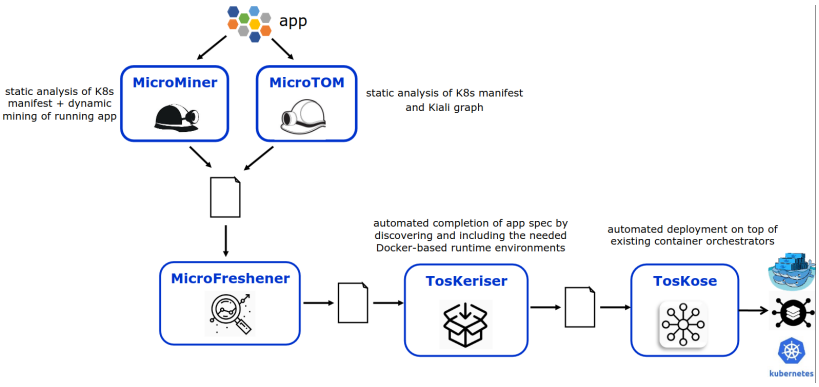


Figure 11.3: Microservices toolchain

Chapter 12

Security and Privacy

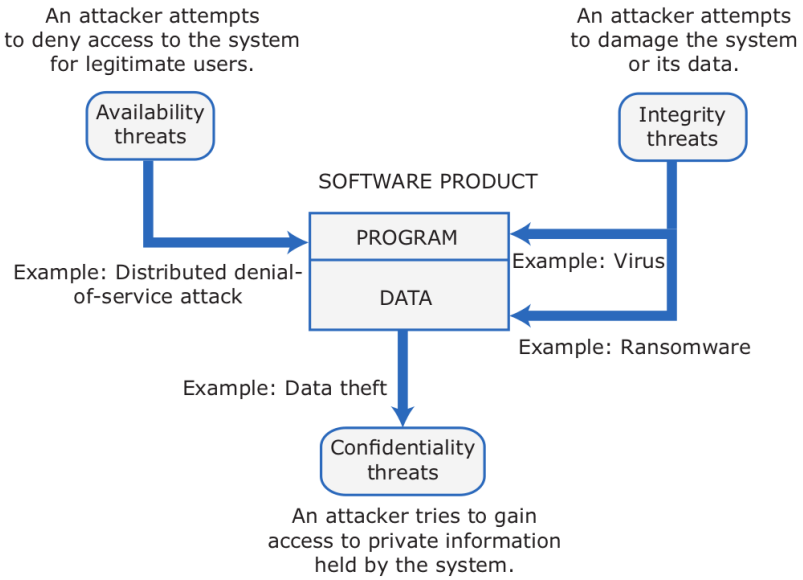


Figure 12.1: Security threats

Security is a *system-wide issue*: Application software depends on operating system, web server, language run-time system, database, frameworks and tools, which may *all* be targeted by attacks.

There are some system management procedures whose aim is to increase security, the most obvious ones are **authentication** and **authorization** (later discussed in Sections 12.2 and 12.2). *System infrastructure management* aims to keep infrastructure software configured and to promptly apply security updates patching vulnerabilities. Regularly *monitoring attacks* enhances the ability to detect them and trigger resistance strategies to minimize the impact. To achieve resilience instead, *backup policies* are defined to keep undamaged copies of program and data files to be restored after an attack.

12.1 Attacks types

12.1.1 Injection Attacks

Malicious users try to crash the system by sending invalid input values. The defense is definining a robust input validation.

Examples

1. Buffer overflow attacks
2. SQL Poisoning

12.1.2 Session hijacking

A *Session* is a time period during which user's auth with a web app is valid; it allows the user to not having to re-authenticate for subsequent system interactions. A session is closed when the user logs out or due to a "times out" caused by no user inputs for some time.

An attacker may acquires a valid session cookie through *Cross-site scripting* (*active* hijacking) or *traffic monitoring* (*passive* hijacking), and then impersonate a legitimate user.

Possible defenses include:

- ◇ Encryption (`HTTPS`)
- ◇ Multifactor authentication
- ◇ Short timeout sessions

Cross-site Scripting

`XSS` (i.e. *cross-site scripting*) falls in the category of injection attacks, since it consists in adding malicious code —to leak information— to a web page returned from a server to a user, which inputs precious data handled by the malware.

12.1.3 Denial-of-Service attacks

DoS are intended to make system unavailable for normal use, and were typically implemented by sending a high number of requests to overload servers, resulting in the unavailability of services provided by such servers.

An historical **DoS** technique exploited the TCP 3-way handshake. DoS developed in *Distributed DoS* (**DDoS**), i.e. sending requests from multiple IP addresses.

The most basic DoS techniques now have standard countermeasures to block them. Widely used basic countermeasures include:

1. IP tracking
2. Temporary users lockout after failed authentication

12.1.4 Brute Force

Attackers may try to guess missing authentication information by generating all possible combinations of characters, possibly by knowing partial information on the string to be generated.

12.2 Authentication

Approaches

1. **Knowledge**-based authentication
   Personal secret knwon by the user.
   Passwords are often insecure, forgotten, reused, and besides the user can be fooled into inserting it in fake websites (*phishing*).
2. **Possession**-based authentication
   Possessing a physical device which provides tokens.
3. **Attribute**-based authentication
   Biometric information of the user.
4. **Multi-factor**:
   Combining the above. This is becoming way more common and standardized.

Some services provide a **federated** authentication method e.g. *"Login with Google"*, typically implemented using library `OAuth`, which rely on a trusted third party authenticator; such method is widely used on mobile devices, where typing passwords is inconvenient, but in general can fit in many scenarios, with the downside of forcing the product provider to share some data with the (external) authenticator.

12.3 Authorization

Authorization involves checking that an authenticated user can *access resources*, while **authentication** is only about ensuring that the user is who they claim to be.

Access Control Lists (`ACL`s) are widely used to implement access control policies, they allow classifying individuals into groups, dramatically reducing ACLs size, and definining hierarchies of groups.
ACLs often realized by relying on ACL of underlying file or db system.

12.4 Encryption

Encryption means making a document unreadable by applying an algorithmic transformation to it. Modern encryption techniques are considered "practically uncrackable" using currently available technology.

There are well-known symmetric and asymmetric encryption techniques. `HTTPS` uses server-client interaction to generate a secret for symmetric encryption.
`HTTPS` = `HTTP` + ~~`SSL`~~ `TLS` *(Transport Layer Security)*
TLS is used to verify *identity* of web server and to encrypt communications, it does so by exploiting a digital certificate sent from server to client, issued by a trusted identity verification service (`CA`).

In general encryption should be adopted **whenever possible**, for both *in transit* and *at rest* data, while for *in use* data it is known that may cripple performance, and mechanisms of key management are used instead.

Ideally, if keys get lost, encrypted data become permanently inaccessible. Keys should be changed periodically and multiple timestamped versions of keys should be mantained, creating the need for a **Key Management System** (*KMS*) to make sure that keys are securely generated, stored, and accessed by authorized users.

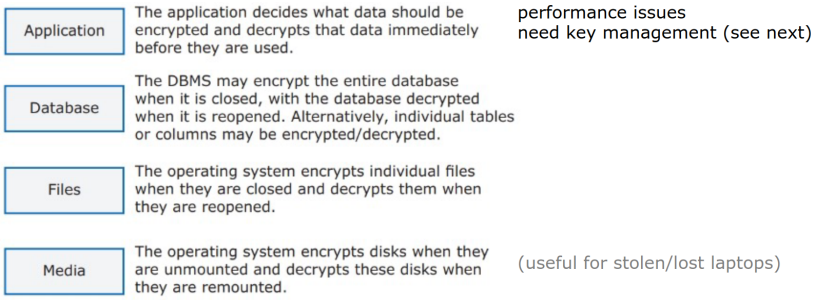


Figure 12.2: Encryption is possible in all system levels

12.5 Privacy

Privacy is a social concept that relates to the collection, dissemination, and appropriate use of personal information held by a third party.

There may business reasons for paying attention to information privacy:

- ◇ If your conformance to privacy regulations does not match data protection regulations, you may be subject to legal actions / cannot sell your product
- ◇ If your sell a business product, your business customers may require privacy safeguards (not to be at risk with their users)

◇ Leakage/misuse of client information can damage your reputation

The information that your software *needs* to collect depends on the *functionality* of your product and on the business model you use; we can provide some tips based on this assumption:

◇ Do not collect personal information that you do not need
◇ Establish a privacy policy defining how personal/sensitive information about users is collected, stored, and managed
◇ Make clear if you use users' data to target advertising or to provide services that are paid for by other companies
◇ If your product includes social network functionalities so that users can share information, you should ensure that users understand how to control the information they share

12.6 Microservices

In microservices there is much inter-service communication via remote calls and thus a large number of entry points, considerably broadening the **attack surface**. Besides, each microservice requires carrying out an individual **security screening**, to assess changes in time, given the mutable structure of this architetural style.

The most used approach is *ZeroTrust* networks, which –sadly– have some impact on performance

12.6.1 Communication

Service-to-service communication must take place on *protected channels*, and typically exploits the use of **certificates**, one for each microservice. Even though it's effective, revoking and rotating certificates on hundreds of microservices is a complex task, and thus should be automated.

12.6.2 Logging

A request to a microservices deployment may span multiple microservices, making correlating requests among microservices challenging. To this extent logs and traces help a lot:
Logs can be aggregated to produce metrics that reflect system state (e.g. average invalid access requests per hour) and that may trigger alerts on security or performance; **Traces** instead help you tracking a request from the point where it enters the system to the point where it leaves the system.

◇ **logging**: *Prometheus* and *Grafana* to monitor incoming requests
◇ **Tracing**: *Jaeger* and *Zipkin* for distributed tracing

12.6.3 Containers against statefulness

Containers are *immutable servers*, they don't change state after being spinned up; This is a great property which simplifies deployment and allows horizontal-scalability, but for each service we need to maintain a **dynamic list** of allowed clients and a dynamic set of access-control policies, which *cannot* be stored inside the container. A solution is to use a `push/pull` model to get updated policies from some policy admin endpoint.
Each service must also maintain its own **credentials**, which need to be rotated periodically, but this can be handlend by keeping credentials in container filesystem, by injecting them at *boot time*.

Since containers have no state, and requests span distributedly, **user context** has to be *passed explicitly* from one microservice to another, so there must be a way to ensure trust between microservices so that receiving microservice accepts user context passed from calling microservice.

Typically implemented with *JSON Web Tokens* (`JWT`)

Confused Deputy What is the "confused deputy problem" in microservices?

1. application does not enact access control in some microservices, allowing the attacker to get data it shouldn't be able to get
2. microservices trust the gateway based on its mere identity, leading to potential violation of authenticity

12.6.4 Architetural smells

Property	Smell	Solution
Confidentiality	Insufficient Access Control	`OAuth 2.0`
Confidentiality	Publicly accessible microservices	`API` gateway
Confidentiality Integrity	Unnecessary privileges to microservices	apply *Least Privilege*
Confidentiality Integrity Authenticity	"Home-made" crypto code	use *known* *encryption techniques*
Confidentiality Integrity Authenticity	Data Exposure	encrypt all ***at-rest data***
Confidentiality Integrity Authenticity	Hardcoded secrets	encrypt all ***at-rest secrets***
Confidentiality Integrity Authenticity	Non-secure service-to-service communications	Mutual `TLS`
Authenticity	Unathenticated traffic	Mutual `TLS` `OpenID Connect`
Authenticity	Multiple user authentication	`API` gateway `OpenID Connect`
Authenticity	Centralized authorization	Single *sign-on*[1] Decentralize authorization

Table 12.1: Microservices security smells and refactorings

Chapter 13

Business Process Modeling

Business process management is the systematic method of examining your organization's existing business processes and implementing improvements to make your workflow more effective and efficient. A business **process** is a set of business acivities that represent the required steps to achieve a business objective.
A business process **model** consists of a set of activity models and execution constraints among them. A BP **instance** represents a concrete case in the operational business of a company, consisting of activity scenarios.

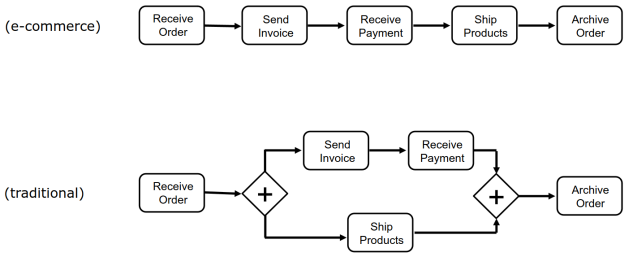
The following is an example of a BP model descripted in natural language:

"When we receive a new order, an invoice should be sent to the customer. The order should be archived only after receiving the payment. The requested products must be shipped to the customers"

Activities

1. Receive order
2. Send invoice
3. Archive order
4. Receive payment
5. Ship product

Figure 13.1: How can we draw arrows between activies?



13.1 Notation for BPM

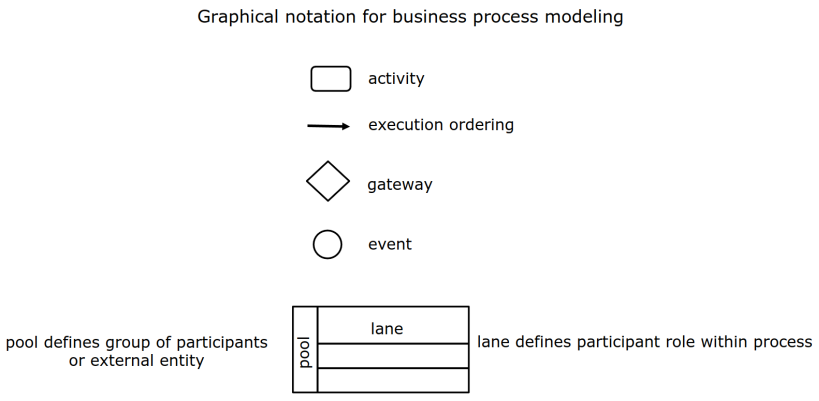


Figure 13.1: Business process model Notation

Such notation is used as syntax to define BP models; There are tools that support it and is deeply useful since **proving properties** of business process models is a crucial aspect of business process management.

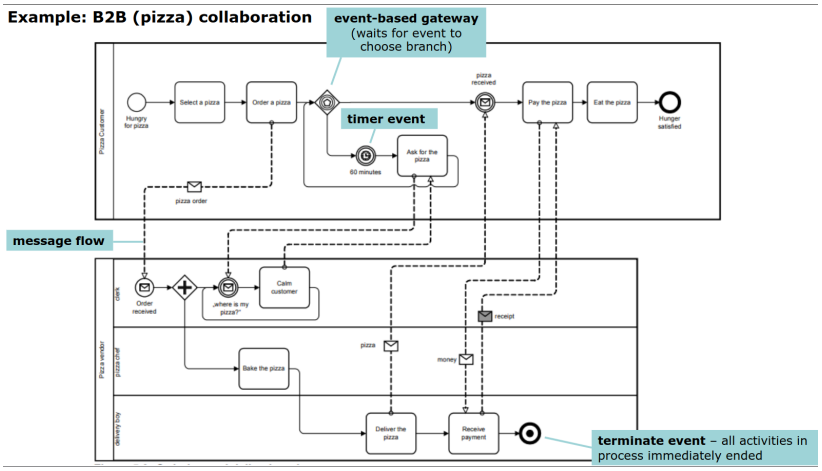


Figure 13.2: Pizza BPM example

13.2 Workflow Nets

Workflow Nets are an extension of **Petri Nets**, and are one of the best known techniques for specifying business processes in a **formal** and **abstract** way.

Petri nets consist of **places**, **transitions** and direct **arcs** connecting places and transitions. Transitions model ***activities***, places and arcs model ***execution constraints***. System dynamics are represented by **tokens**, whose distribution over the places determines the *state* of modelled system; a transition can *"fire"* if there is a *token* for each of its input *places*: when it fires, one token is removed from each input place and one token is added to each input place.

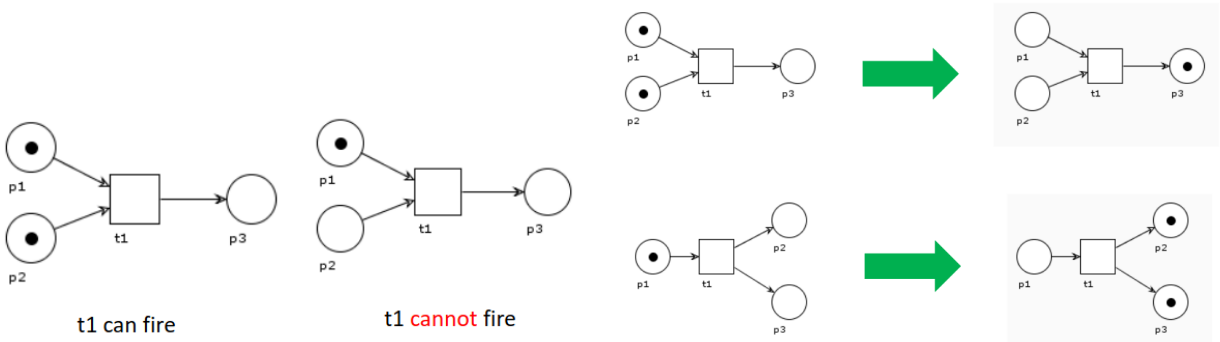


Figure 13.3: Transition firing

A *Petri net* is a ***workflow net*** `iff`

1. There is a *unique* ***source*** *place* with no incoming edge
2. There is a *unique* ***sink*** *place* with no outgoing edge
3. All places and transitions are located on some path from the initial place to the final place

In workflow nets, there may be "sugared" explicit versions of `AND` and `OR`, and transitions may be annotated with triggers to indicate which entity is responsible for the transitions to fire.

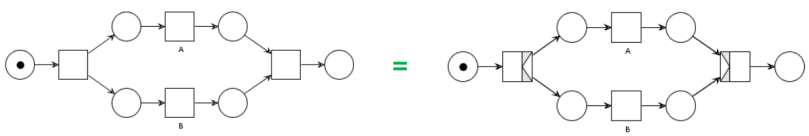


Figure 13.4: Explicit `AND-split` and `AND-join` transitions

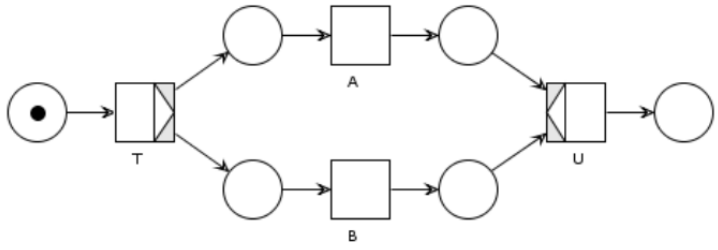


Figure 13.5: Explicit `OR`

Soundness

A workflow net is –unformally– **sound** `iff`:

1. every net execution starting from the initial state eventually leads to the final state

   Recall that:

   - **Initial state** - one token in the sink place, no tokens elsewhere
   - **Final state** - one token in the source place, no tokens elsewhere

2. every transition occurs in at least one net execution

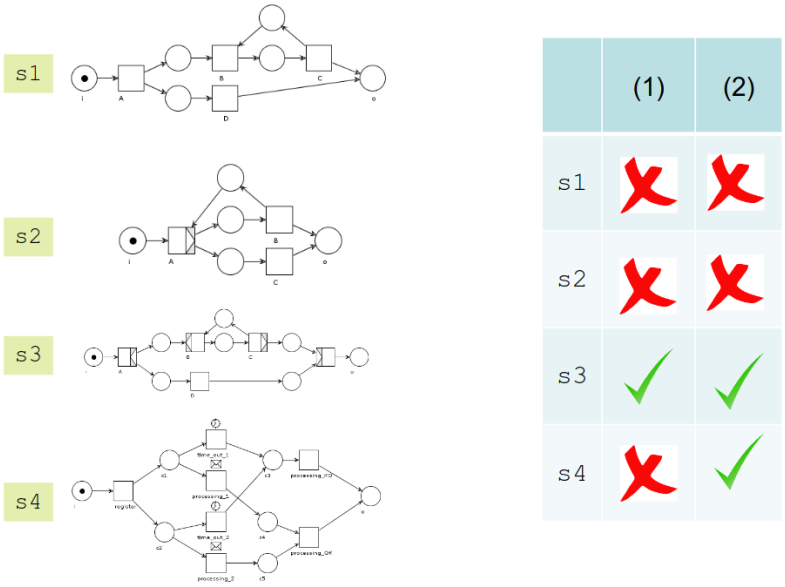


Figure 13.6: Soundness check with examples

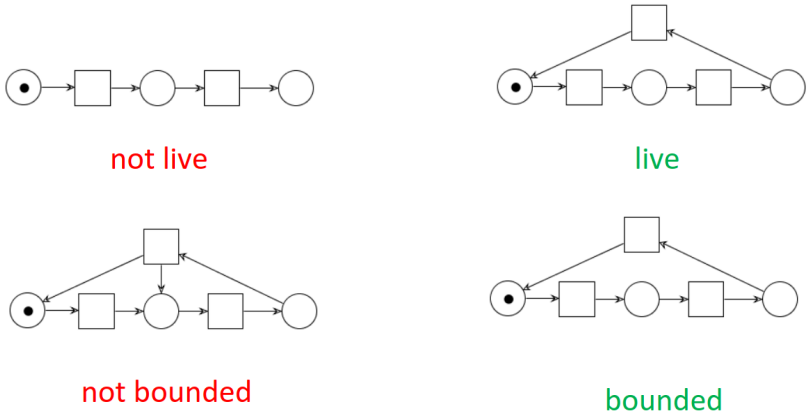
(More) Notation

- ✉ message activated
- 🕒 time activated
- 👤 user activated

13.2.1 Formally

A Petri net (PN, M) is **live** $\iff$ for any reachable state M' and every transition t there is a reachable state M'' reachable from M' where t is enabled.

A Petri net (PN, M) is **bounded** $\iff$ for each place p there is an $n \in N$ such that for each reachable state M' the number of tokens in p in M' is less than n.



Theorem 13.2.1 *A worflow net N is* ***sound*** $\iff$ *$(\check{N},i)$ is both live and bounded, where $\check{N}$ is N extended with a transition from the sink place o to the source place i.*

13.2.2 BPMN to Workflow Nets

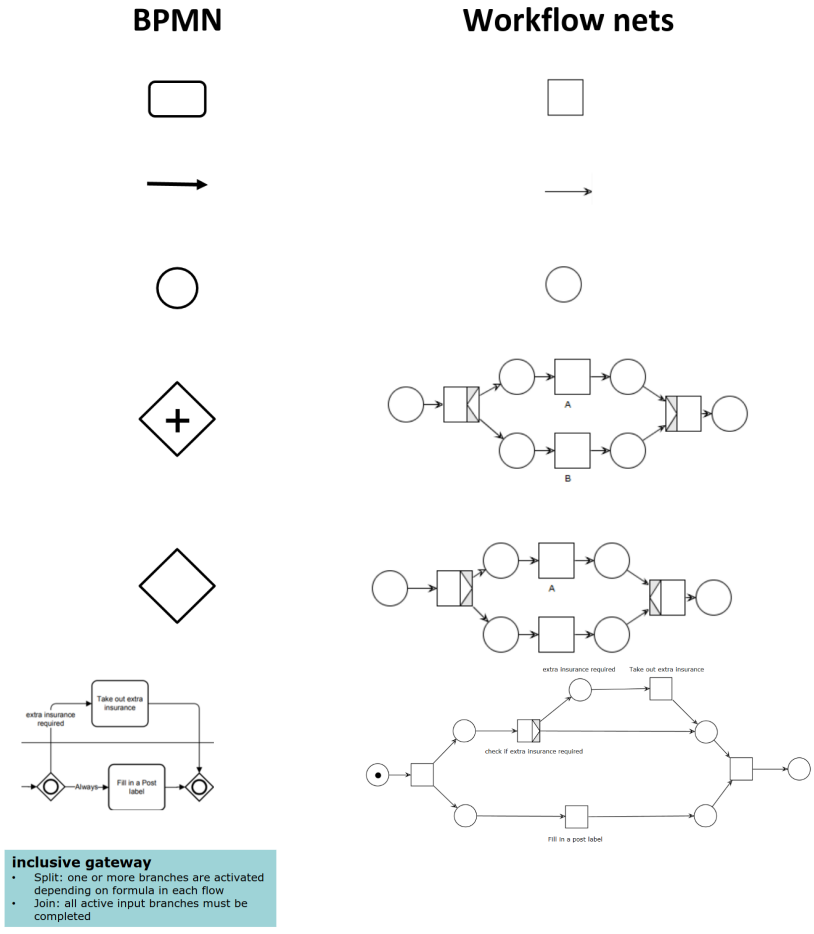


Figure 13.7: From BPMN to Workflow Nets

Chapter 14

Testing

There are various types of testing, depending on what is their purpose.

1. **Functional** Testing: test that the functionalities of the system work as expected and to discover eventual bugs
2. **User** Testing: Test that the product is useful to and usable by end users
   i. *Alpha* $\longrightarrow$ *"Do you really want these features?"*
   ii. *Beta* $\longrightarrow$ Early version of product distributed to users to check product usability and effective interoperability
3. **Performance** and **load** testing: Test that system **responds quickly** to service requests and it can handle different loads and **scales** gracefully as the load increases
4. **Security** testing: Find vulnerabilities that attackers may exploit

There are two main things to remember when talking about testing, the first is an old known quote from Dijkstra:

*"Program testing can be used to show the **presence** of bugs, but never to show their **absence**"*

Edsger W. Dijkstra

The other is that software **testing** *it **not** equal to* software **verification**, which instead involves representing the software in a model to prove some properties.

14.1 Functional testing

The first thing to address is testing **coverage**: all code should be executed at least once by the test suite.

Testing should[1] start on the day you start writing code, and should possibly be automated, to considerably simplify the **develop/test cycle** and to allow functional testing to be a staged activity.

14.1.1 Unit Testing

Unit testing consists in testing program units (e.g. function, method) in isolation.

Theorem 14.1.1 (Unit testing principle) *If a program unit behaves as expected for a set of inputs that have some shared characteristics, it will behave in the same way for a larger set whose members share these characteristics.*

[1] Actually, finding someone who starts testing on day 0 –or even day 100– is pretty hard.

For example:
If your program behaves correctly on the input set $1, 5, 17, 45, 99$, you may conclude that it will also process all other integers in the range 1 to 99 correctly

To exploit the unit testing principle 14.1.1 it is crucial to identify **equivalence partitions** i.e. sets of inputs that will be treated the same in your code; they allow to test a program using several inputs from each equivalence partition. Besides, keep in mind that programmers make mistakes at **boundaries**: identify equivalence partition and their boundaries and choose inputs at these boundaries.

14.1.2 Feature testing

Features must be tested to show that a functionality is implemented as **expected** and that it meets the real **users' needs**. There are two types of feature testing that address these two key aspects:

1. **Interaction** testing addresses the testing between multiple units, possibly developed by different teams
2. **Usefulness** testing, which the *product manager* should contribute to, aims to understand whether a feature is what users actually want.

14.2 System and Release testing

Aside from single functionalities, the system should be tested *as a whole*; to achieve so it is recommended to use a set of scenarios/user stories to identify users' end-to-end pathways, to consider the whole system.
This is necessary for several reasons:

1. To discover **unexpected/unwanted interactions** between the features
2. To discover if system features work together effectively to support what users really want to do
3. To make sure system operates as expected in the **different environments** where it will be used
4. To test *responsiveness*, *throughput*, *security*, and other **quality** attributes

Release testing instead addresses testing of a system intended to be released to customers, performing tests in the real operational environment, rather than in the test one.
The aim of is to decide if the system is *good enough* to **release**, *not* to detect bugs in the system.

Preparing a system for release involves **packaging** the system for deployment, installing used software and libraries, configure parameters; many mistakes can be made in such process, creating the need for release testing.

If you deploy on the cloud, an automated continuous release process can be used.

14.3 Test Automation

Automated testing is widely used in *product development* companies. To facilitate the task there are many testing frameworks available for all widely used programming languages.
Executable tests check that software return expected result for input data.

A good practice is to **structure** automated tests in three parts:

1. **Arrange**
   Set up the system to run the test, i.e. define test parameters, mock objects, etc.
2. **Action**
   Call the unit that is being tested with the test parameters.
3. **Assert**
   Assert what should hold if test executed successfully.

```
#Arrange - set up the test parameters
p=0
r=3
n= 31
result_should_be = 0
#Action - Call the method to be tested interest = interest_calculator (p, r, n)  #
    Assert - test what should be true self.assertEqual (result_should_be, interest)
```

Note that *test code* can include **bugs**! To avoid this tests should be made as simple as possible and they should be reviewed along with the code that they test.

Unit tests are the easiest to automate, and if properly done can reduce (not eliminate) the need for feature tests.

GUI-based testing expensive to automate, since they multiple assertions are needed to check that a feature has executed as expected; API-based testing is preferrable from this point-of-view.

System testing involves testing system as a *"surrogate user"*, performing sequences of user actions. *Manual* system testing is definitely **boring** and error prone, so in general should be avoided in favor of testing tools that record a series of actions and **automatically replay** them.

14.4 Test-driven development

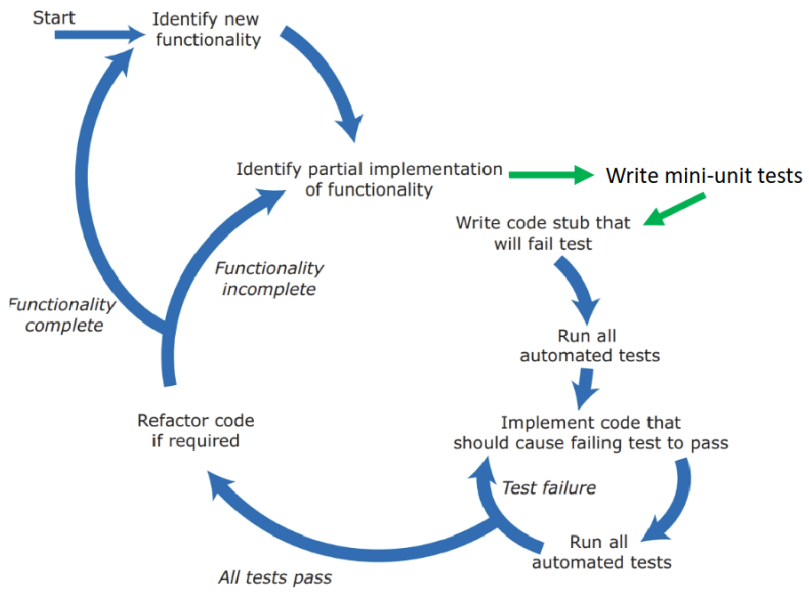


Figure 14.1: Test-driven development schema

Theorem 14.4.1 (Extreme Programming) *"First write executable test, then write the code"*

Pros

1. By exploiting such systematic approach, tests are clearly **linked** to code sections, almost eliminating the chance to have untested snippets.
2. Tests help understanding program code
3. Simplified, incremental debugging
4. (Arguably) simpler code

Cons

1. Difficult to apply TDD to system testing
2. TDD *discourages* radical program changes
3. TDD leads you to focus on the tests rather than on the problem you are trying to solve
4. TDD leads you to think too much about implementation details rather than on overall program structure
5. Hard to write "bad data" tests

14.5 Security Testing

The two main objectives of **security testing** are finding *vulnerabilities* exploitable by attackers and providing convincing evidence that the system is sufficiently *secure*.

Finding vulnerabilities is harder than finding bugs, because it implies testing for something that software should *not* do, thus having a potentially infinite number of tests.
Normal functional tests may not reveal vulnerabilities, besides the **software stack** (OS, libraries, dbs, ...) on which your product depends may contain vulnerabilities, and some of these may arise in the composition of these components.

Comprehensive security testing requires **specialist knowledge** and typically implies a risk-based approach:

1. identify main security **risks** to your product
2. develop tests to demonstrate that the product **protects** itself from these risks

When testing security, you need to think like an *attacker* rather than a normal end-user.
While it is generally possible to *automate* some of these tests, others inevitably require manual checking of behaviour/-files.

14.6 Limitations of testing

1. You test code against your understanding of what that code should do; so if you have misunderstood the **purpose** of the code, then this will affect both the code and the tests.
2. Testing may not provide **coverage** of all the code you have written.
   Test-Driven development shifts the problem to code **incompleteness**.
3. Testing does *not* really tell you anything about some attributes of a program e.g. *readability*, *structure*, *evolvability*...

Often a single **code reviewer** is used, they may be part of same *DevOps* team, but not necessarily. The reviewer can also comment on *readability* and *understandability* of code, which are aspects that testing cannot address.

A single review session should focus on $200 - 400$ lines of code, and typically implies using checklist, e.g.

Checklist e.g.

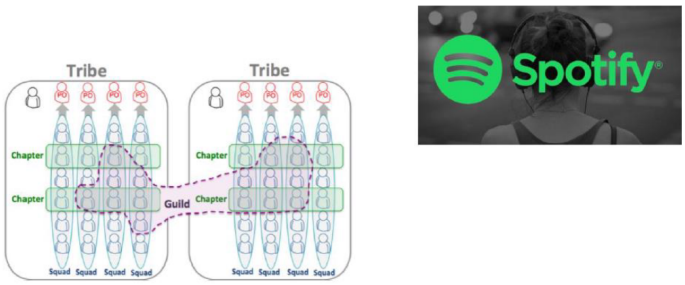
1. Are meaningful variables and function names used? (*General*)
2. Have all data errors been considered and tests developed for these? (*General*)
3. Are all exceptions explicitly handled? (*General*)
4. Are default function parameters used? (`Python`)
5. Are types used consistently? (`Python`)
6. Is the indentation level correct? (`Python`)

Reviews can be "triggered" by commits on shared repositories, and can be eased up by **code review tools**, eventually paired with messaging systems like *Slack*.

14.7 Takeaway quotes

"Pay attention to zeros. If there is a zero, someone will divide by it."

"Testers don't break software, software is already broken."



Chapters are team members working within a special area (e.g., front office developers, back office developers, database admins, testers).

A **guild** is a community of members across the organization with shared interests (e.g., web technology, test automation), who want to share knowledge, tools code and practices.

Chapters (sometimes guilds) do **code reviews** for squads. Two «+1» required to merge.

Figure 14.2: Testing in Spotify

Chapter 15

DevOps

In the old-style software engineering there are **separate** teams addressing software development, deployment, release and support. The development team, or a separate maintenance team, is responsible for software updates and maintenance.
This approach introduces many issues, which can be summarized as **difficult communication** amongst teams, due to different tools, skills, etc. leading to requiring multiple to days to release an "urgent" patch.

Amazon re-engineered its software into (micro)services assigning both service development and service support to the same team.

15.1 Principles

Principles

D1 - *Everyone is responsible for everything.*

i. All team members have joint responsibility for developing, delivering, and supporting the software.

D2 - *Everything that can be automated should be automated*

i. All activities involved in testing, deployment, and support should be automated if it is possible to do so. There should be mimimal manual involvement in deploying software.

D3 - *Measure first, change later.*

i. DevOps should be driven by a measurement program where you collect data about the system and its operation. You then use the collected data to inform decisions about changing DevOps processes and tools.

Benefits

1. *Faster deployment*
   Software can be deployed to production more quickly because communication delays between the people involved in the process are dramatically reduced.
2. *Reduced risk*
   The increment of functionality in each release is small so there is less chance of feature interactions and other changes that cause system failures and outages.
3. *Faster repair*
   DevOps teams work together to get the software up and running again as soon as possible. There is no need to discover which team was responsible for the problem and to wait for them to fix it.
4. *More productive teams*
   DevOps teams are happier and more productive than the teams involved in the separate activities. Because team members are happier, they are less likely to leave to find jobs elsewhere.

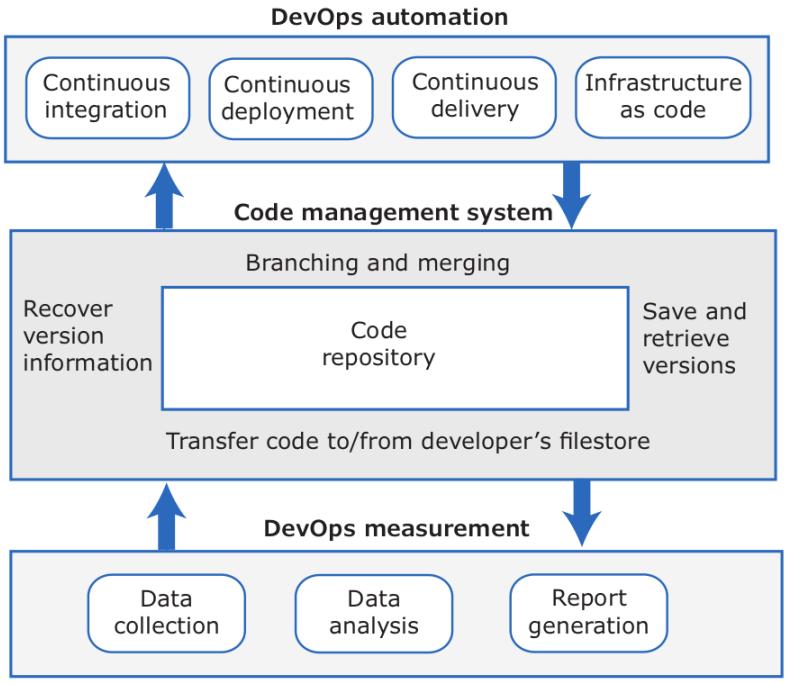


Figure 15.1: Code management according to DevOps

15.2 Code Management

Code must be **managed** using `git`, which is a *decentralized system* which provides some crucial benefits:

1. **Resilience**
   i. Every dev has its *own copy* of the project
   ii. If the shared repository is damaged or subjected to a cyber-attack, work can continue and a previous state may be *restored*
   iii. Devs can *work offline*
2. **Speed**
   i. Committing changes to the repository is a fast, *local* operation
3. **Flexibility**
   i. Local experimentation is much simpler i.e. Developers can safely try different approaches without exposing their experiments to other project members

15.3 Automation

`GitHub` provides a mechanism called **WebHooks** which trigger DevOps **automation tools** in respose to updates to the project repository.

15.3.1 Continuous integration

Continuous integration means that every time that a dev commits a change to project's main branch, an *updated executable* version of the system is **built** and **tested**.

System **integration** ($\rightarrow$ *building*) is more than simply "compiling source code", but includes many steps like installing `db` software, loading test data, preparing config files, running system tests, and so on.

If a system is *infrequently integrated*, problems can be difficult to isolate, and fixing them would considerably slow down system development.

With continuous integration, an integrated version of the system is created and tested every time a change is pushed to the system's shared code repository. On completion of the push operation, repository sends message to integration server to build a new version of the product.

Breaking the Build

To avoid breaking a working build, usually an **"integrate twice"** approach is preferred: the integrated system version is created and tested **locally**, and then —if tests succeed— the changes get pushed to the project repository and the main integration server gets triggered, resulting in a new official build.

Besides, no dev wants the stigma of "breaking the build", so everyone is very careful and check its code before pushing to project repo.

15.3.2 Continuous delivery and Deployment

Continuous delivery –and *deployment*– refers to the testing environment, which must be the product's operating environment.

Continuous deployment means that every new release of the system should be seamlessy made available to users as soon as a change gets committed[1] to the main branch.

With continuous *integration*, new system builds get tested –locally by devs and– by the integration server, but such development differs from the **real production environment**. The production server may have different filesystem organization, access permissions, installed applications; in other words, new bugs may show up.

Continuous delivery ensures that changed system is ready for delivery to customers, by performing **feature tests** in *production* environment to make sure that environment does not cause system failures, and **load tests** to check how software behaves as number of users increases.

Containers are the simplest way to create a replica of a production environment.

So, in order to **deliver**, after initial integration testing a staged test environment must be configured to crete a **replica** of production environment, where **acceptance tests** on *functionality*, *load* and *performance* can be ran into.

Continuous **deployment** is practical only for cloud-based systems and can be instead can be summarized in three steps:

1. software and data are transferred to production servers
2. new service requests stopped, older version processes outstanding transactions
3. switch to new system version and restart processing

However note that, there may be **business reasons** to *avoid deploying* every system change to customers, like the presence of incomplete features, too frequent UI changes, or desire/need to synchronize releases with known business cycles[2]

15.3.3 Infrastructure as Code

Manually maintaining a computing infrastructure with tens or hundreds of servers is expensive and error-prone: different servers may require different dependencies, drivers, packages... some may be virtual, others physical...

[1] Shouldn't it be tested first? ☺

[2] solar/academic year, seasons, tax payments...

The process of updating software on servers should be automated by using a **machine-readable model** of the infrastructure $\longrightarrow$ *"Infrastructure as Code"*.

Configuration Management (`CM`) tools, like *Puppet* and *Chef* and *Ansible*, can automatically install software and services on servers according to the infrastructure definition, so that when changes have to be made, the infrastructure model is updated and CM tool makes the changes to all servers.

This provides tracking and prevents all the human-based errors related to configuring, installing and managing software on servers, ultimately resulting in many advantages:

1. **Visibility**
   Infrastructure defined as stand-alone model that can be read/understood/reviewed by whole DevOps team.
2. **Reproducibility**
   Installation tasks will always be performed in the same sequence, same environment will be always created. You do not have to rely on people remembering the order that they need to do things.
3. **Reliability**
   Automation avoids simple mistakes made by system administrators when making same changes to several servers.
4. **Recovery**
   Infrastructure model can be versioned and stored in a code management system. If infrastructure changes cause problems, you can easily revert to older version and reinstall the environment that you know works.

Summing up, **Infrastructure as Code** indicates that instead of delegating the system infrastructure managing to human administrators or developers, *machine-readable* models of the infrastructure [3] should be provided, on which the product executes are used by configuration management tools to automatically build the software's execution platform; the software to be installed, such as compilers, libraries, DBMS, packages and so on, is included in the infastructure model.

15.4 Measuring DevOps

Your own DevOps process should be continuously measured in order to improve it and thus to enhance software quality and reduce deployment times.

1. **Process** measurements
   collect and analyze data on development/testing/deployment processes
2. **Service** measurements
   collect and analyze data on software's performance/reliability/acceptability to customers
3. **Usage** measurements
   collect and analyze data on how customers use your product (help you identify issues and problems with the software itself)
4. **Business success** measurements
   collect and analyze data on how your product contributes to overall business success

   hard to isolate contribution of DevOps to business success - that may be due e.g. to DevOps introduction or to better managers

Measuring software and its development, should be automated but is a complex process: defining suitable metrics and ways to collect them is away from trivial.

Consider the basic example of record *lead time* for implementing a change: is the *"lead time"* the overall elapsed time or the time spent by developers? How to take into account higher priority changes whose implementation slowed down other changes?

[3] network, servers, routers, etc.

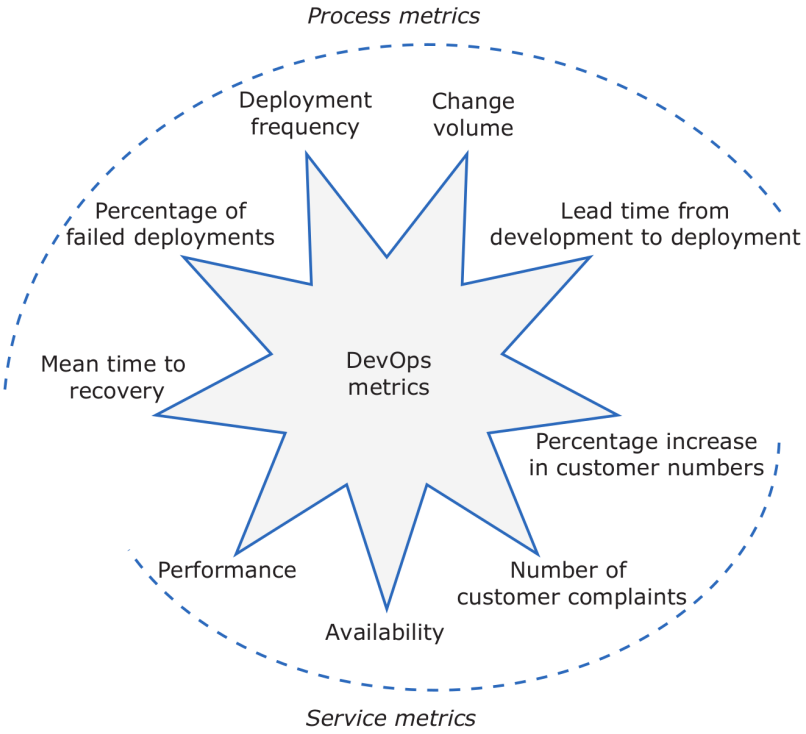


Figure 15.2: DevOps Metrics

1. Use **continuous integration tools** like *Jenkins* to collect data on *deployments*, successful tests, etc.
2. Use **monitoring services** featured by cloud providers like *Amazon's Cloudwatch* to collect data on *availability* and *performance*
3. Collect *customer-supplied data* from **issue management systems**
4. Add **instrumentation** to your product to gather data on its performance and how it is used by customers
    i. use `log` files, where `log` entries are timestamped events reflecting customer actions and/or software responses
    ii. `log` as many events as possible
    iii. use available `log` analysis tools to extract useful information about on how your software is used

Chapter 16

Seminario Domotz

16.1 First Speaker

16.2 Second Speaker

1. Unit tests
2. Integration tests
3. End to End tests

is

Chapter 17

Cloud-Edge Continuum

Cloud computing, even if it provides potentially unlimited storage and processing capabilities, has three key issues which may be critical in some situations:

1. Latency
2. Internet connection Required
3. Lots of Data over the network

Conversely, relying on an infrastucture on the "*Edge*", i.e. close to the IoT device, offers low latency but limited storage and processing capabilities.

So the key idea of the **Cloud-Edge Continuum** is to extend the cloud towards the IoT world into a *distributed* and *heterogenous* infrastructure to get the best of both Cloud and Edge approaches.
Hence, applications should be **microservice**-based and **containerised**, allowing them to be deployed on a *continuous Cloud-Edge infrastructure*.

17.1 Placing applications

There are many issues the infrastructure has to manage;

1. **Hardware** and **Software** requirements
2. **Data Gravity**
3. **Security**
   Need to match application's security requirements with infrastructure's security capabilities Need to model (non-monotonic, conditionally transitive) trust relations among different stakeholders
4. **Sustainability**
5. **Infrastructure**

Placing applications in the Cloud-Edge Continuum is challenging and ***NP**-Hard*; approaches to solve the problem include `ML`, `MILP`[1], *Declarative Reasoning* supported by an *Inference Engine*.

17.2 Managing applications

Infrastructure and applications will most-likely mutate throughout time, so the eventual changes must be managed, and in order to do so, they must *both* be monitored.

[1] Mixed-Integer Linear Programming

There must a be a process of *continuous reasoning*, to exploit compositionality to differentially analyse a large-scale system:

- ◇ mainly focusing on the latest changes introduced in the system, and
- ◇ re-using previously computed results as much as possible

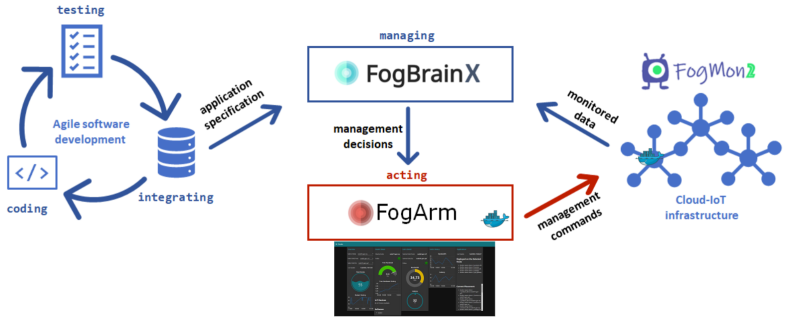


Figure 17.1: Monitoring ⟶ reasoning ⟶ enacting

- ◇ Osmotic Management
- ◇ Decentralized management (bacteria-inspired)
- ◇

Chapter 18

Exam Questions

18.1 What is product-based software engineering?

Product-based software engineering refers to the approach and methodology used in the development of software products. Unlike project-based development, where the primary focus is on delivering a specific solution for a client, product-based development centers around creating software that will be marketed, sold, and used by a broader audience. In product-based software engineering, the same company is responsible for deciding on both the features that should be part of the product and the implementation of these features; customers, even if considered, do **not** decide on software features, releases or attributes, which is entirely up to the developer. As showed above:

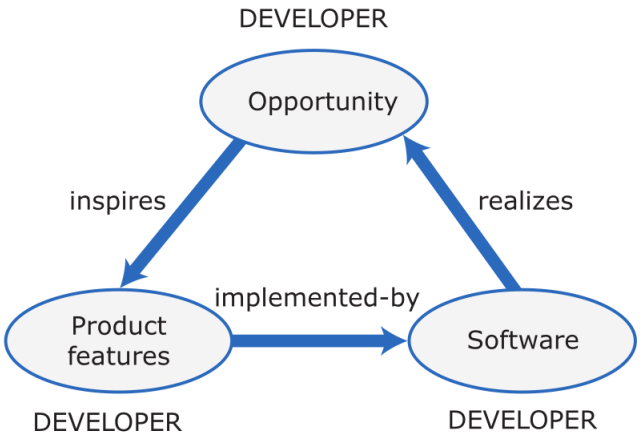


Figure 18.1: product based software engineering cycle

- ◇ Because the product developer is responsible for identifying the **opportunity**, they can decide on the **features** that will be included in the software product. These features are designed to appeal to potential customers so that there is a viable market for the software. The product manager must ensure that development team implements features that deliver real value to customers.

  In order to do so, he should regularly *communicate with customers*, *monitor technology constraints* relevant for customers, and keep the *focus on goals* of both company and customers; in practice, this translates to

  - Product roadmap - goals, milestones, success criteria
  - User stories and scenarios
  - Product backlog (is product $manager \neq owner$?)
  - Acceptance testing - verify that release meets set goals
  - Customer testing
  - Monitor usability in UI design

- ◇ Product companies can decide when to change their product or take their product off the market. The software developer decides also the platforms on which the software will be implemented, and so on.

Product-based SE addresses the problem of mutable requirements which is becoming more and more common, crippling most Project-based software from early development stages.
Two key differences from Project-based SE are:

1. Quick software delivery is critical
2. Software development starting point is the *product vision*, i.e.
   - FOR target customer
   - WHO need or opportunity
   - THE product name is a product category
   - THAT key benefit, compelling reason to buy
   - UNLIKE primary competitive alternative
   - OUR PRODUCT primary differentiation
3. **Prototyping** is the foundation of whole product development process, opposed to a project defined in advance.
4. Customers, even if considered, do **not** decide on software features, releases or attributes, which is entirely up to the developer.

18.2 What is the incremental development and delivery advocated by Agile? What are the key Scrum practices?

18.2.1 Incremental development and delivery

Incremental development and delivery consists in breaking down the software development process into a set of features, with each doing something for the user, allowing developers not to worry about the details of all the features, you define only the details of the features that you plan to include in an increment.

The features are prioritized in a way so that the most important features are implemented first

Activities of incremental development typically are:

- **Choose features to be included in an increment**: Using the list of features in the planned product, select those features that can be implemented in the next product increment.

  Features with high priority should be preferred over others.
- **Refine feature descriptions**: Add detail to the feature descriptions so that the team members have a common understanding of each feature and there is sufficient detail to begin implementation.
- **Implement and test**: Implement the feature and develop automated tests for that feature that show that its behavior is consistent with its description.
- **Integrate feature and test**: Integrate the developed feature with the existing system and test it to check that it works in conjunction with other features.
- **Deliver system increment**: Deliver the system increment to the customer or product manager for checking and comments. If enough features have been implemented, release a version of the system for customer use.

18.2.2 Key Scrum practices

Scrum is a lightweight framework that helps people, teams and organizations to create value through adaptive solutions to complex problems. In Scrum, the software product is developed in a series of sprints, each of which delivers an **increment** of the product or supporting software. The Key Scrum practices are the following:

These can be used in any agile development process even if other Scrum aspects are not applied.

Product backlog : This is a to-do list of items —*features(/user stories), engineering improvements, tools assessment* and other *process activities*— to be implemented that is reviewed and updated before each sprint. Items in the product backlog are:

- ready for consideration
- ready for refinement
- ready for implementation

product backlog activities are:

- **refinement**: Existing PBIs are analysed and refined to create more detailed PBIs. This may lead to the creation of new product backlog items.
- **estimation**: The team estimate the amount of work required to implement a PBI and add this assessment to each analysed PBI. It is expressed in person-hours or person days or arbitrary "story points".

- ◇ **creation**: New items are added to the backlog, possibly suggested by the product manager.
  Even if called "features" in the Scrum lecture slides, they are typically expressed as *user stories*.

  As a teacher, I want to be able to configure the group of tools that are available to individual classes. (feature / user story) (Ref. Slide 13)

- ◇ **prioritization**: The product backlog items are reordered to take new information and changed circumstances into account.

Timeboxed sprints : Fixed-time (2-4 week) periods in which items from the product backlog are implemented; The main sprint activities are:

- ◇ **Sprint planning**: Work items to be completed in that sprint are selected and, if necessary, refined to create a sprint backlog. This should not last more than a day at the beginning of the sprint.
- ◇ **Sprint execution**: The team work to implement the sprint backlog items that have been chosen for that sprint. If it is impossible to complete all of the sprint backlog items, the sprint is not extended. The unfinished items are returned to the product backlog and queued for a future sprint. Usually during sprints two agile activities are used:

  test automation : As far as possible, product testing should be automated. You should develop a suite of executable tests that can be run at any time.
  continuous integration : Whenever anyone makes changes to the software components they are developing, these components should be immediately integrated with other components to create a system. This system should then be tested to check for unanticipated component interaction problems.

- ◇ **Sprint reviewing**: The work done in the sprint is reviewed by the whole team and (possibly) external stakeholders. The team reflect on what went well and what went wrong during the sprint with a view to improving their work process. The product owner has the ultimate authority to decide whether or not the goal of the sprint has been achieved. They should confirm that the implementation of the selected product backlog items is complete.

Self-organizing teams : Self-organizing teams make their own decisions and work by discussing issues and making decisions by consensus (ideal size between 5 and 8 people), without a project manager. The advantage of a heterogeneous self-organizing team, where speech communication is preferred over work documentation amongst team members, is that it can be a cohesive team that can adapt to change.
In a Scrum project, the Scrum Master and the Product Owner should be jointly responsible for managing interactions with people outside the team.
Scrum assumes that teams share a workspace and that all team members can attend a morning meeting to coordinate the work for the day.

18.3 What are personas, scenarios, user stories, and features?

- ◇ **Personas** are (imaginary) types of product users. They are used to understand potential users in order to design features that will be useful to them. They give background, skills and experience of potential users. Personas allow developers to "step into the users' shoes".

  Usually only a couple of personas (max 5) are needed to identify key product features.

  They should include personal, education and job-related information, along with other details which may indicate their interest in the product

- ◇ **Scenarios** are narratives describing a situation in which a user is using our product's features to do something he/she wants to do. High-level scenarios facilitate communication and stimulate design creativity. Scenarios are not specifications, so they lack detail and may be incomplete. They are written from the user's perspective (3-4 for each persona).

  They should be created by every team member and discussed together, possibly also with customers.

  Even though it is possible to express all functionalities described in a *scenario* using *user-stories*, scenarios can read more naturally, make stories understanding easier, and provide more context.

- ◇ **User stories** are finer-grained narratives that describe, in a more detailed and structured way, "a single what" a user wants from a software system. The standard format of a user story is:

  As a $< role >$, I $< want/need >$ to $< do\ something >$

  They allow to organize and chunk work into single units which represent actual value to the customer, ultimately building software incrementally from the users perspective. A story which involves multiple sprints to be implemented (hence is in some sense "longer") is called an "**epic**", and can be split into shorter stories. An example of an epic may be:

  *"**As a** system manager, **I need** a utility tool **to** back up and restore individual applications, files, or the whole system"*

  Usually a *Scrum product backlog* is a set of *user stories* sorted according to priority.

◇ A **feature** is a way for users to access and use the functionality of your product, so the feature list defines the overall functionality of the system. The properties of the features are shown below:

- *independence*: a feature should not depend on how other system features are implemented and should not be affected by the order of activation of other features
- *coherence*: features should be linked to a single item of functionality and not have side effects
- *relevance*: system features should reflect the way users normally carry out some task, rather than introducing obscure or rarely needed functionality

They should result from combining User Knowledge (expressed as User Stories/Personas/Scenarios), Domain, Product and Technology knowledge

18.4 What is the role of non-functional quality attributes and decomposition in a software architecture? What is a distribution architecture? What are the technology choices that affect a software architecture? What are the main features of Enterprise Integration Patterns?

18.4.1 Non-functional quality attributes and decomposition

Non-functional quality attributes have the role to describe aspects of a software system that are not related to its specific functionalities but are crucial for evaluating its overall performance. These attributes are critical to the final product, but not to the prototype.

Note that optimizing one non-functional attribute might affect others

Non-functional quality attributes are:

◇ **responsiveness**: Does the system return results to users in a reasonable time?
◇ **reliability**: Do the system features behave as expected by both developers and users?
◇ **availability**: Can the system deliver its services when requested by users?
◇ **security**: Does the system protect itself and users' data from unauthorized attacks and intrusions?
◇ **usability**: Can system users access the features that they need and use them quickly and without errors?
◇ **maintainability**: Can the system be readily updated and new features added without undue costs?
◇ **resilience**: Can the system continue to deliver user services in the event of partial failure or external attack?

Decomposition increases the maintainability of the system by decomposing the system into small **self-contained** parts: $Service < Component < Module$ where *Service* is a coherent unit of functionality, and the *Component* a unit providing a set of related services. Too much decomposition increases the complexity of the architecture, and consequently the integration between components. Good practices to control the complexity are:

◇ **separation of concerns**: organize your architecture into components that focus on a single concern;
◇ **stable interfaces**: design interfaces that are coherent and that change slowly;
◇ **implement once**: avoid duplicating functionality at different places in your architecture.

◇ **Localize relationships** between components by putting them in the same module
◇ **Avoid data sharing** among components

Despite implying a mixup between implementation and design, system decomposition must be done in conjunction with choosing technologies for the system.

18.4.2 Distribution architecture

A **distribution architecture** (figure 18.2) defines servers and allocation of components to servers. For example, client–server architectures are a type of distribution architecture that is suited to applications in which clients access a shared database and business logic operations on those data.

These applications include several servers, such as web servers and database servers. Access to the server set is usually mediated by a load balancer, which distributes requests to servers. It is designed to ensure that the computing load is evenly shared by the set of servers.

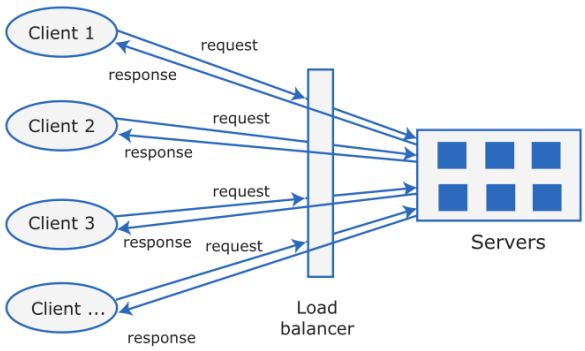


Figure 18.2: client-server architecture

The majority of software products are now web-based products, so they have a client-server architecture, usually based on the **Model-View-Controller (MVC) pattern**. The term *model* is used to mean the system data and the associated business logic. and is shared and maintained on the server. Each client has its own updated view of the data, with the controller (residing on the client-side) acting as an intermediary.

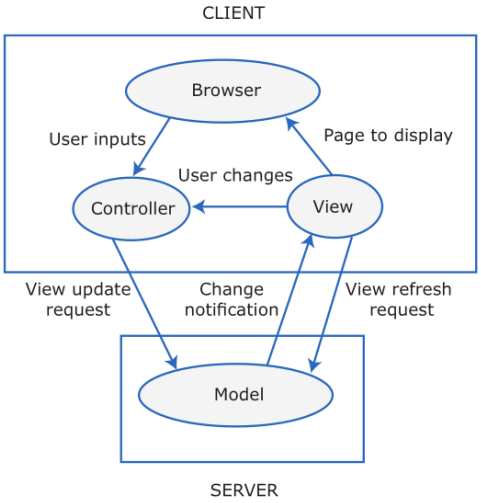


Figure 18.3: mvc pattern

Many web-based applications use **a multi-tier architecture** with several communicating servers, each with its own responsibilities.

An alternative to a multi-tier client–server architecture is a **service-oriented architecture** where many servers may be involved in providing services. Services in a service-oriented architecture are stateless components, which means that they can be replicated and can migrate from one computer to another. A service-oriented architecture is usually easier to scale as demand increases and is resilient to failure. Multi-tier and service-oriented architectures are the main types of distribution architecture for web-based and mobile systems. You have to decide which of these to choose for your software product. Factors that influence what architecture to choose are:

- **Data type and data updates**: If you are mostly using structured data that may be updated by different system features, it is usually best to have a single shared database that provides locking and transaction management.

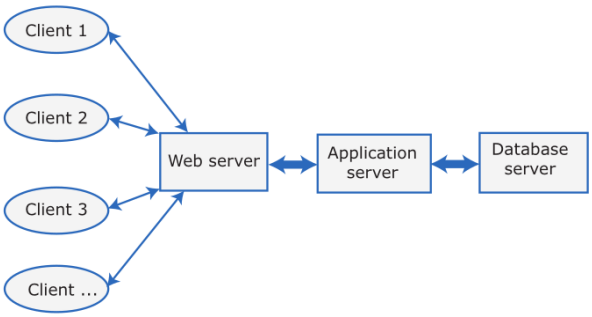


Figure 18.4: multi-tier client-server architecture

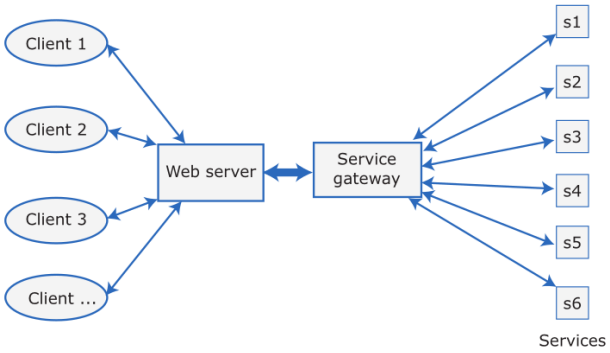


Figure 18.5: service oriented architecture

If data are distributed across services, you need a way to keep them consistent, and this adds overhead to your system.

- **Change frequency**: If you anticipate that system components will regularly be changed or replaced, then isolating these components as separate services simplifies those changes.
- **The system execution platform**: If you plan to run your system on the cloud with users accessing it over the Internet, it is usually best to implement it as a service-oriented architecture because scaling the system is simpler. However, if your product is a business system that runs on local servers, a multi-tier architecture may be more appropriate.

18.4.3 Technologies choices

The **technologies choices** that affect and constrain the overall system architecture and that are difficult/expensive to change during the development are:

- **database**: should you use a relational SQL database or an unstructured NoSQL database?
- **platform**: should you deliver your product on a mobile app and/or a web platform?
- **server**: should you use dedicated in-house servers or design your system to run on a public cloud? If a public cloud, should you use Amazon, Google, Microsoft, or some other option?
- **open source**: are there suitable open-source components that you could incorporate into your products?
- **development tools**: do your development tools embed architectural assumptions about the software being developed that limit your architectural choices?

18.4.4 Enterprise Integration Patterns

An EIP is a reusable abstraction of proven solutions to well-known problems that arise when integrating the software components/services that make up enterprise applications.

They mainly address the problem of handling the communication between services, data sources and participants.

The foundation of the integration achieved by EIPs are one-way channels (point-to-point or publish-subscribe) and message endpoints, but may be enriched by various other components, such as

1. i. *Adapters* - Application to Channel
   ii. *Translators* - Format suitable for receivers
   iii. *Normalizers* (Router+Translators) - Normalizing data coming from multiple sources in various formats in a single output
   iv. *Content Enrichers* - Add context info to messages
2. i. *Routers* - Route messages to desired receivers
   ii. *Splitters* - Duplicate/split message to multiple receivers
   iii. *Aggregators* - Aggregate messages coming from multiple sources, blocking until all messages are received

18.5 What is a Docker image/container? What is Docker Compose? What are the differences between multi-tenant and multi-instance SaaS systems? What are the design principles of K8s? How does K8s control plane work?

18.5.1 Docker

- A **docker container** is a standalone and executable software package that includes everything needed to run a piece of software, including the code, runtime, libraries, and system tools.

  Unlike VMs container-based virtualization does not require emulating an entire OS

  Docker containers are based on images and can be started, stopped, moved, and deleted independently of the underlying infrastructure.
- An **docker image** is a software component which is exploited as read-only templates to create and run containers. images are built from a set of instructions (using a Dockerfile) and include a snapshot of the application and its dependencies. images are stored in a (private or public) Docker registry (e.g. Docker Hub).
- **Docker Compose** is a tool to define and run multi-container Docker applications. It allows you to define an entire application stack, including services, networks, and volumes, in a single file, named docker-compose.yml. All services defined in the file can be started with the command *docker-compose up*. It simplifies the process of managing multiple containers that need to work together as part of a larger application.

18.5.2 Multi-tenant vs multi-instance

The difference between multi-tenant and multi-instance systems is that:

multi-tenant systems use a single database schema shared by all system's users. The items in database are tagged with tenant identifier to provide "logical isolation", so that customer companies have their own space and can store and access their own data.

Mid-size and large businesses usually want a version of the multi tenant software that is adapted to their own requirements

multi-instance systems means that each customer has its own system tailored to its needs, including its own database and security controls. It can be VM-based or Container-based (also you can run multiple containers for individual users on top of a VM-based system). Conceptually a multi-instance system in simpler than a multi-tenant system and avoids concerns such as inter-organization data leakage.

Let's consider *pros and cons* of such a solution. Flexibility, security, scalability and resilience are clearly some key points of *multi-instance DBs*. However update management difficulty and cloud VMs renting costs are not negligible.

Wrapping up, there are three possible ways of providing a customer **database** in a *cloud-based* system:

1. As a **multi-tenant** system, *shared by all customers* for your product. This may be hosted in the cloud using large, powerful servers.
2. As a **multi-instance** system, with *each customer database* running on its own *virtual machine*.

Advantages	Disadvantages
Resource utilization The SaaS provider has control of all the resources used by the software and can optimize the software to make effective use of these resources.	**Inflexibility** Customers must all use the same database schema with limited scope for adapting this schema to individual needs. I explain possible database adaptations later in this section.
Security Multi-tenant databases have to be designed for security because the data for all customers are held in the same database. They are, therefore, likely to have fewer security vulnerabilities than standard database products. Security management is also simplified as there is only a single copy of the database software to be patched if a security vulnerability is discovered.	**Security** As data for all customers are maintained in the same database, there is a theoretical possibility that data will leak from one customer to another. In fact, there are very few instances of this happening. More seriously, perhaps, if there is a database security breach, then it affects all customers.
Update management It is easier to update a single instance of software rather than multiple instances. Updates are delivered to all customers at the same time so all use the latest version of the software.	**Complexity** Multi-tenant systems are usually more complex than multi-instance systems because of the need to manage many users. There is, therefore, an increased likelihood of bugs in the database software.

Figure 18.6: pros and cons of multi-tenant

Advantages	Disadvantages
Flexibility Each instance of the software can be tailored and adapted to a customer's needs. Customers may use completely different database schemas and it is straightforward to transfer data from a customer database to the product database.	**Cost** It is more expensive to use multi-instance systems because of the costs of renting many VMs in the cloud and the costs of managing multiple systems. Because of the slow startup time, VMs may have to be rented and kept running continuously, even if there is very little demand for the service.
Security Each customer has its own database so there is no possibility of data leakage from one customer to another.	**Update management** Many instances have to be updated so updates are more complex, especially if instances have been tailored to specific customer needs.
Scalability Instances of the system can be scaled according to the needs of individual customers. For example, some customers may require more powerful servers than others.	
Resilience If a software failure occurs, this will probably affect only a single customer. Other customers can continue working as normal.	

Figure 18.7: pros and cons of multi-instance

3. As a **multi-instance** system, with *each database* running in its own *container*. The customer database may be distributed over several containers.

18.5.3 K8s

The design principles of K8s are:

- ◇ **declarativeness**: we simply define the desired state of our system and K8s will detect when the actual state of the system doesn't meet our expectations and it will intervene to fix the problem, making our system self-healing. The desired state is defined by a collection of objects, each of which has a specification in which you specify the desired state and a status that reflects the current state of the object. K8s constantly polls each object to ensure that its status is equal to the specification (e.g. if an object is unresponsive, K8s will spin up a new version to replace it).
- ◇ **distribution**: K8s provides a unified interface for interacting with a cluster of machines. We don't have to worry about communicating with each machine individually.
- ◇ **decoupling**: Containers should be developed with a single concern in mind (implements a microservice-based architeture). K8s naturally supports the idea of decoupled services which can be scaled and updated independently.
- ◇ **immutable infrastructure**: to get the most out of containers and container orchestration, we should use an immutable infrastructure. During the life-cycle of a project (development - testing - production) we should use the same container image. Containers are designed to be ephemeral, ready to be replaced by another container instance at any time. Maintaining immutable infrastructure makes it easier roll back applications to previous state (eg. if an error occurs) - we can simply update our configuration to use an older container image.

The K8s control plane relies on the **master node**: (often unique) machine that contains most of the control plane components:

- The user provides new/updated object specification to **API server**(figure 18.8) of master node, then API server validates update requests and acts as unified interface for questions about cluster's current state. The state of cluster is stored in a distributed key-value store etcd, a distributed key-value store.

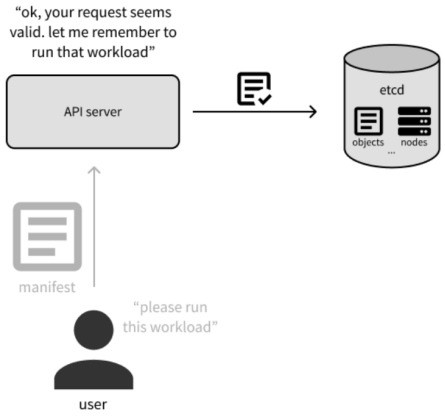


Figure 18.8: api server of the master node.

- The **scheduler**(figure 18.9) determines where to run objects by asking the API server which objects have not been assigned to a machine, determining which machines those objects should be assigned to, and responding back to the API server to reflect this assignment.

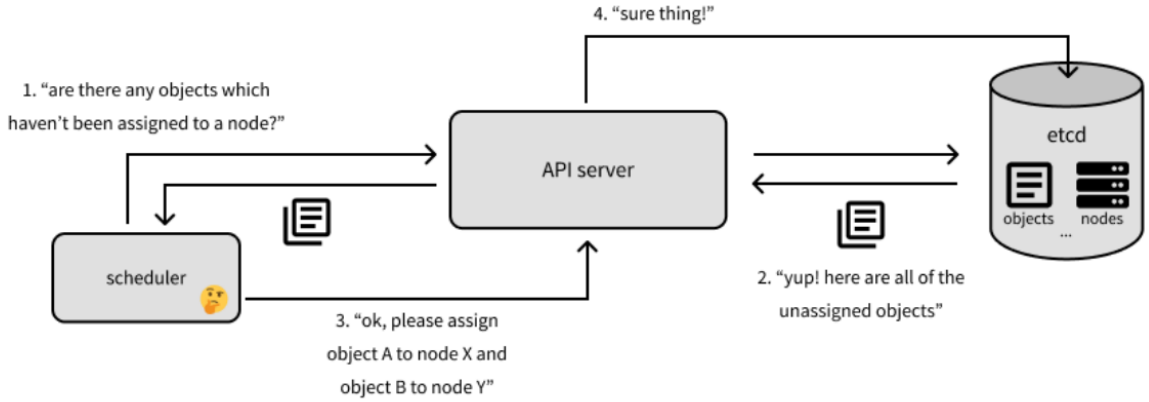


Figure 18.9: scheduler of the master node.

- The **controller-manager**(figure 18.10) monitors cluster state through the API server. If actual state differs from desired state, the controller-manager will make changes via the API server to drive the cluster towards the desired state.

18.6 What are the main (dis)advantages and characteristics of microservices? What does the CAP theorem tell us? Which refactoring can be applied to resolve architectural smell X?

18.6.1 Microservices: characteristics, pro and cons

Microservices are small-scale services that can be combined to create applications. They are independent (the service interface is not affected by changes to other services), allowing a service to be modified and redeployed without modified/stopping other services.

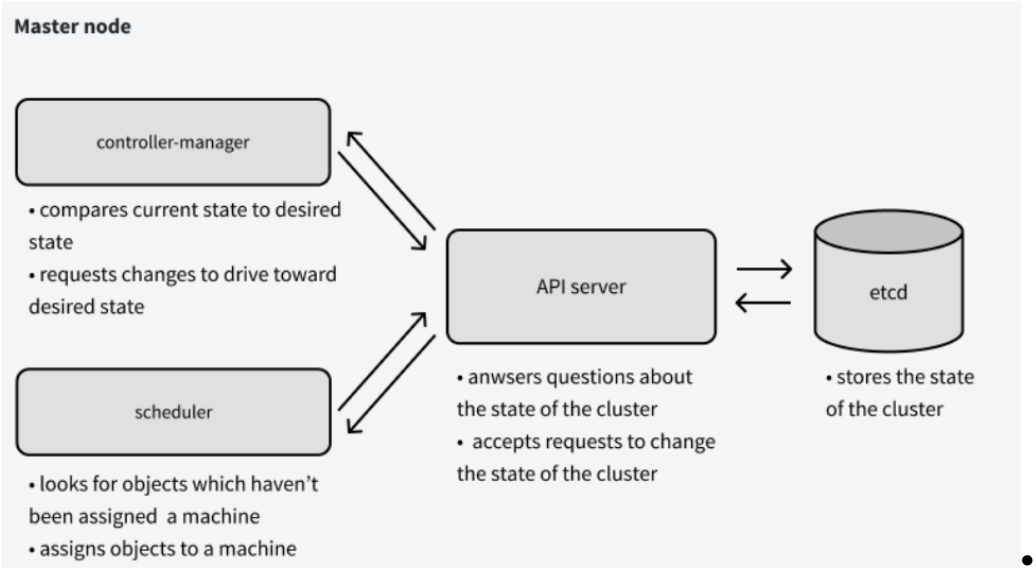


Figure 18.10: controller manager of the master node.

The main characteristics of microservices are the following:

- ◇ **self-contained**: microservices do not have external dependencies. They manage their own data and implement their own user interface.

  User interface?
- ◇ **lightweight**: microservices communicate using lightweight protocols so that service communication overhead is low.

  Sacrifing the generality that's inherent in the design of web service protocols
- ◇ **independent implementation**: microservices may be implemented using different programming languages and may use different technologies (e.g. different types of database) in their implementation.
- ◇ **independently deployable**: each microservice runs in its own process and is independently deployable using automated systems.
- ◇ **business-oriented**: microservices implement business capabilities and needs, rather than simply provide a technical service.

The main two reasons (pro) to adopt microservices are:

1. **Reduced lead time for new features and updates**: this is made possible by *autonomous teams* working on different, independent and decoupled services, and *continuous deployment*.
2. **Scale**: it is effectively possible to scale up or down a service, e.g. in case of large amount of load, it is possible to deploy new replica to balance the load and avoid congestion.

On the other hand there are also some cons, that are:

- ◇ **communication overhead**: Microservices communicate with each other over a network. This can introduce latency and additional complexity, especially in cases where frequent communication is required between microservices.
- ◇ **complexity**: Coordinating and managing a large number of microservices can lead to challenges in understanding, deploying, and maintaining the entire system.
- ◇ **"wrong cuts"**: If the splitting of the monolothic application is done incorrectly, it may lead to suboptimal microservices that are either too coarse-grained or too fine-grained.

  See also *cohesion* and *coupling*, regarding the number of intra/intercommunications.
- ◇ **"avoiding data duplication as much as possible while keeping microservices in isolation is one of the biggest challenges"**

◇ **"A poor team will always create a poor system"**: Success with microservices relies heavily on collaboration, communication, and skilled teams. Inadequate team collaboration or lack of expertise can lead to poorly designed and implemented microservices.

The observation on teams impacts more than any microservices-over-monolith choice dictated by system complexity

18.6.2 CAP Theorem

The CAP theorem says that "It is impossible for a web service to provide Consistency, Availability and Partition-tolerance at the same time", where:

◇ **Consistency**: each service returns the "right" response to each request,
◇ **Availability**: each request eventually receives a response,
◇ **Partition-tolerance**: services can be partitioned into multiple groups, and network can delay/lose arbitrarily many messages among services.

18.6.3 Architectural smell

An architectural smell is a commonly used architectural decision that negatively impacts system lifecycle qualities. An architectural smell is a threat to one or more of these —microservices— design principles:

◇ **Independent deployability**: The microservices forming an application should be independently deployable;
◇ **Horizontal scalability**: The microservices forming an application should be horizontally scalable;
◇ **Isolation of failures**: Failures should be isolated;
◇ **Decentralization**: Decentralization should occur in all aspects of microservice-based applications, from data management to governance.

There are several ways to resolve a smell by refactoring a microservice. During the course we saw MicroFreshener which is a tool which edits app specifications, automatically identifies architectural smells and allow the user to apply one of the suggested refactors to the architecture. Then, the application manager can implement concretely the refactoring modifying the code of the application.

During the course we discussed which are the possible solutions for each architectural smell, which are the same that MicroFreshener proposes. They are summarized in the table below:

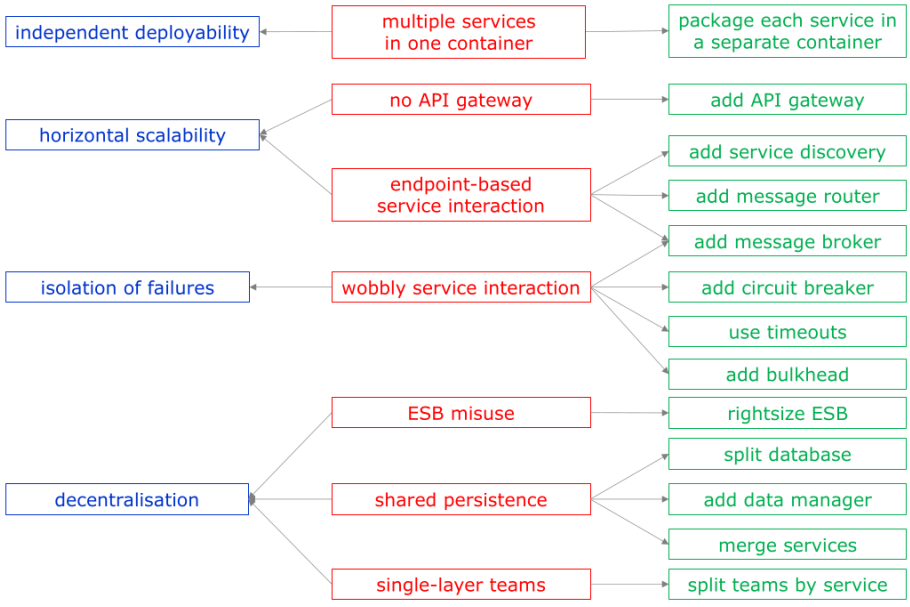


Figure 18.11: Smells and corresponding solutions

- ◇ **Message broker**: between services. This helps decouple services, improve fault tolerance, and enable event-driven architecture
- ◇ **Message router**: route messages based on content, headers, or other criteria.
- ◇ **Bulkhead**: pattern indicating t use separate resources (e.g., databases, thread pools) for different microservices, preventing failures in one area from affecting others.
- ◇ **rightsize ESB**:
  textit"ESB" stands for Enterprise Service Bus, i.e. an architectural pattern whereby a centralized software component performs integrations between applications.
  An ESB designed for a monolithic architecture may become a bottleneck, ⟶ break down its components into more lightweight and focused communication mechanisms. Consider replacing the monolithic ESB with more decentralized and targeted communication patterns, such as direct HTTP/REST calls or message brokering.

Emiliano - Smells and refactoring

All the architectural smells will be categorized based on the design principle they violate. For each smell will be presented some refactorings to solve them.

Independent deployability

One *architectural smell* for independent deployability is multiple services in one container (because ideally we want to have one container per microservices). It is possible to insert some microservices that interact a lots inside the same container, but this can lead to various problems. The solution for this smell is to package each service in a separate container.

Horizontal scalability

One first *architectural smell* for horizontal scalability is endpoint-based service interaction. This is a smell because, in the case it's present an interaction to a specific service istance (as shown in Figure 18.12), adding more replicas of the service won't scale the application, since the interaction is made with only one of the replicas. Some *refactoring* for the described *smell* are to add a service discovery that will indicate to the requester which replica he must use (applied in the 55% of cases), add a message router that will redirect the request to one of the replicas (e.g. load balancer, applied in the 31% of cases) or add a message broker that will collect the requests, which are then collected from the replicas (e.g. message queue, applied in the 14% of cases). The reason that the message broker is the least used is because the code of the microservice has to be modified.

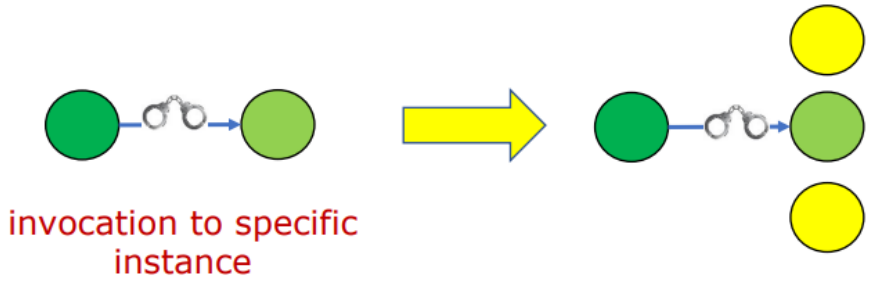


Figure 18.12: Endpoint-based service interaction

Another *architectural smell* is the absence of an API gateway, because clients can invoke directly the services (it's similar to endpoint-based service interaction smell). The solution is simply to add an API gateway, since not only resolve the smell, but can also be useful also for authentication, throttling, and so on.

Isolation of failures

One *architectural smell* for isolation of failures is wobbly service interaction: the interaction of m1 with m2 is *wobbly* when a failure of m2 can trigger a failure of m1. Some *refactoring* for the described *smell* are to add a circuit breaker

(applied in the 42% of cases), use timeouts (applied in the 22% of cases), add a bulkhead (applied in the 20% of cases), or add a message broker (applied in the 16% of cases).

Decentralization

One first *architectural smell* for decentralization is shared persistence. Three proposed solutions (that are better described with the Figure 18.13) are split database (applied in the 50% of cases), add data manager that acts as a gateway between the services and the database (applied in the 22% of cases) or merge services (applied in the 9% of cases).

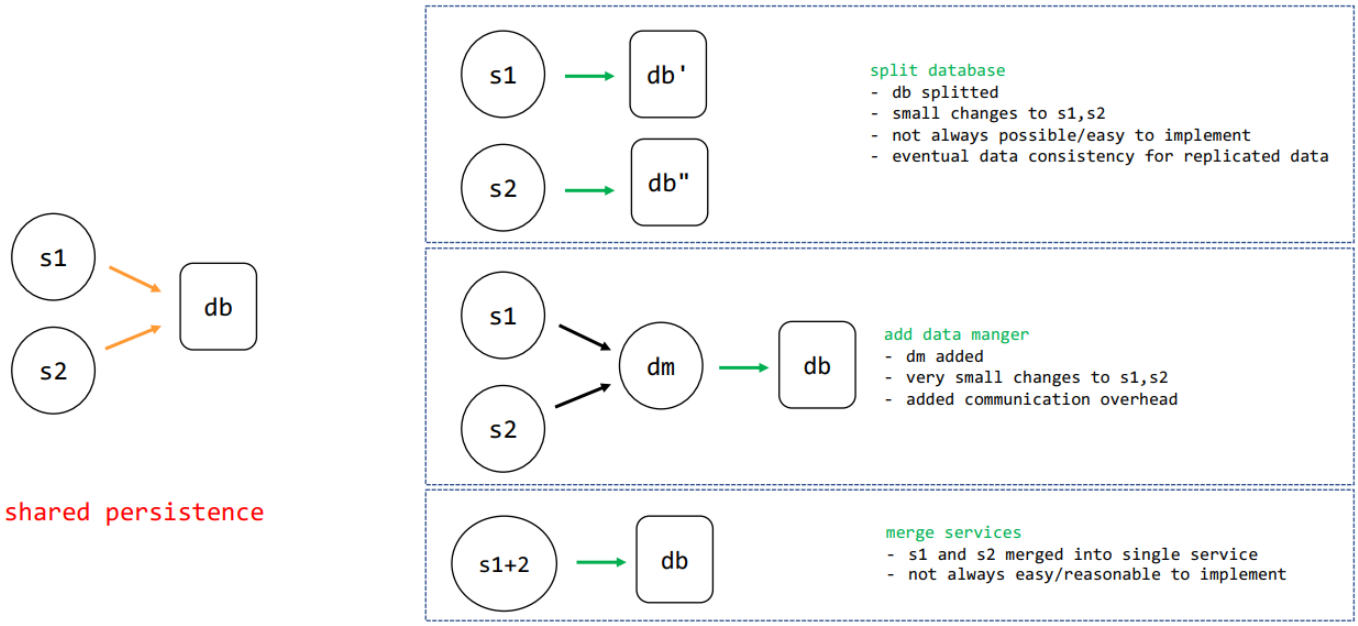


Figure 18.13: Solutions for shared persistence

The last *architectural smell* is the single-layer teams, which goes in contrast to the idea of Agile. The solution is simply to split teams by service.

18.7 How can we feature authentication and authorization in a software product? What are the main challenges in securing microservices?

18.7.1 Authentication in software product

We can feature authentication in a software product with one of this approaches:

- ◇ **Knowledge-based authentication**: it relies on users providing secret, personal information at registering time;
- ◇ **Possession-based authentication**: it relies on users having a physical device that can be connected to the authentication system and that generates/displays information known to the authentication system (e.g. system sends code to user's phone number, or special-purpose device that generates one-time codes);
- ◇ **Attribute-based authentication**: it relies on a unique biometric attribute of the user (e.g. fingerprint, face).

Nowadays many systems now use multi-factor authentication, which is the combination of two or more of the types shown above (e.g. password, then code received on mobile phone).

Knowledge-based authentication often employed for products delivered as cloud services.

Some services provide a **federated** authentication method e.g. *"Login with Google"*, typically implemented using library `OAuth`, which rely on a trusted third party authenticator;

Weaknesses of password-based authentication

- ◇ Users choose insecure passwords that are easy to remember
- ◇ Users click on email link pointing to fake site that collects login and password → phishing attack
- ◇ Users use the same password for several sites (if there is a security breach at one site . . .)
- ◇ Users regularly forget passwords → password recovery mechanism needed → potential vulnerability if credentials have been stolen.

defenses

- ◇ Force users to set strong passwords
- ◇ Add personal questions

The level of authentication that you need depends on your product

- ◇ No need to store confidential user information → knowledge-based authentication enough
- ◇ Need to store confidential user information → use two-stage authentication

Implementing a secure and reliable authentication system is expensive and time-consuming

- ◇ Even if using available toolkits and libraries (e.g. OAuth), there is still a lot of programming effort involved
- ◇ Authentication often outsourced with a **federated identity system**

Sometimes websites offer the opportunity to "Login with Google" or "Login with Facebook." These sites rely on what is called a "federated identity" approach, where an external service is used for authentication.

the advantages of this approach are:

- ◇ user has single set of credentials, stored by a trusted identity service
- ◇ product provider doesn't have to maintain own database of passwords/secrets
- ◇ product provider can get additional user information (if user agrees)

disadvantages: product provider must share user information with external services.

Federated identity verification is ok for products aimed at individual customers and for business products, connecting to business' own identity management system.

There are various ways to implement federated authentication, but most of the major companies that offer federated authentication services use the OAuth protocol. This standard authentication protocol has been designed to support distributed authentication and the return of authentication tokens to the calling system. (bonus part from the book)

However, OAuth tokens do not include information about the authenticated user. They only indicate that access should be granted. This means it is not possible to use OAuth authentication tokens to make decisions on user privileges (for example, what resources of the system they should have access to). To get around this problem, an authentication protocol called OpenID Connect has been developed that provides user information from the authenticating system. Most of the major authentication services now use this, except Facebook, which has developed its own protocol on top of OAuth.

18.7.2 Authorization in software product

We can feature authorization by using an **access control policy** that reflects data protection rules that limit access to personal data. **Access Control Lists (ACLs)** are widely used to implement acces control policies. ACLs classify individuals into groups dramatically reducing ACLs size. Different groups can have different rights on different resources. Hierarchies of groups allow to assign rights to subgroups/individuals. ACLs are often realized by relying on ACL of underlying file or database system.

Since containers are stateless, and they can't simply load ACLs at boot time —like they do with service credentials— due to their mutable nature, usually they exploit a push/pull model to get updated policies from an admin endpoint.

18.7.3 Main challenges securing microservices

The main challenges of securing the microservices are the following:

- ◇ Reduce risk by reducing the attack surface, because the broader the attack surface, the higher the risk;
- ◇ Find the right balance between security and performance, because distributed security screening affects performance;
- ◇ Bootstrapping trust among microservices needs automation. It involves setting up mechanisms and procedures to ensure that the different components (microservices) can securely communicate and interact with each other.
- ◇ Tracing requests spanning multiple microservices;
- ◇ Containers complicate credentials/policies handling;
- ◇ Distribution makes sharing user context harder;
- ◇ Security responsibilities distributed among different teams;

18.8 What is a parallel/exclusive/inclusive gateway in BPMN? What is a workflow net? What is a sound/live/bounded net?

18.8.1 Gateways in BPMN

- ◇ **Parallel Gateway** (AND Gateway):

  meaning : All incoming paths must be completed before the process continues.
  symbol : The symbol for a parallel gateway is a diamond shape with a "+" sign in the center.
  functionality : A parallel gateway represents a point in the process where multiple parallel paths can be taken concurrently. It indicates that all incoming paths must be completed before the process can move forward.

- ◇ **Exclusive Gateway** (XOR Gateway):

  meaning : Only one of the outgoing paths is taken based on conditions.
  symbol : The symbol for an exclusive gateway is an empty diamond shape.
  functionality : An exclusive gateway is used to model a decision point where only one of the outgoing paths can be taken based on certain conditions. It represents a XOR (exclusive or) decision.

- ◇ **Inclusive Gateway**:

  meaning : Multiple outgoing paths can be taken based on conditions.
  symbol : The symbol for an inclusive gateway is a diamond shape with an "O" sign in the center.
  functionality : An inclusive gateway is used to model a decision point where multiple outgoing paths can be taken based on conditions. It represents an inclusive decision, allowing for multiple paths to be followed simultaneously if their conditions are met.

18.8.2 What is a workflow net?

A **workflow net** is an enhanced Petri net with concepts and notations that ease the representation of business processes. Like Petri nets, workflow nets focus on the control flow behaviour of a process:

- ◇ **transitions** represent activities;
- ◇ **places** represent conditions;
- ◇ **tokens** represent process instances;

A Petri net is a workflow net if and only if:

1. There is a unique source place with no incoming edge;
2. There is a unique sink place with no outgoing edge;
3. All places and transitions are located on some path from the initial place to the final place.

18.8.3 What is a sound/live/bounded net

- ◇ A workflow net is **sound** if and only if:
  1. every net execution starting from the initial state (one token in the source place, no tokens elsewhere) eventually leads to the final state (one token in the sink place, no tokens elsewhere), and
  2. every transition occurs in at least one net execution
- ◇ A Petri net (PN, M) is **live** if and only if for every reachable state M' and every transition t, there is a state M" reachable from M' where t is enabled.
- ◇ A Petri net (PN,M) is **bounded** if and only if for each place p there is a $n \in N$ such that for each reachable state M' the number of tokens in M' is less than n.

A theorem says that a workflow net N is sound if and only if $(\tilde{N}, \{i\})$ is live and bounded, where $\tilde{N}$ is N extended with a transition from the sink place o to the source place i.

18.9 What is functional testing? What is test-driven development? What are the limitations of testing?

18.9.1 Functional testing

Functional testing is used to test the overall system's functionality to discover bugs. To achieve this, a large set of program tests is written to verify that all code is executed at least once. Therefore, testing should start on the same day that code writing starts. The develop/test cycle is simplified by automated tests. Functional testing is a staged activity composed of:

1. **unit testing**: the aim of unit testing is to test program units in isolation. Tests should be designed to execute all of the code in a unit at least once. Individual code units are tested by the programmer as they are developed. A code unit is typically a function or class method but can also be a module that includes a small number of other functions.
2. **feature testing**: Code units are integrated to create features. Feature tests should test all aspects of a feature. All of the programmers who contribute code units to a feature should be involved in its testing. Feature testing aims at showing that functionalities are implemented as expected and meet the real needs of users. It uses interaction —between units— tests and usefulness tests. There must be a translation from scenarios/user stories to description of feature tests.
3. **system testing**: Code units are integrated to create a working (perhaps incomplete) version of a system.
   - ◇ To discover unexpected/unwanted interactions between features

- ◇ To determine if system features effectively work together to support users' goal
- ◇ To make sure system operates as expected in the different environments
- ◇ To test responsiveness, throughput, security and other quality attributes

4. **release testing**: test the system in its actual operational environment where it will be used by the users. The aim of this type of testing is to establish if the system is good enough to be released. It's needed since preparing a system for release involves packaging the system for deployment, installing required software and libraries, configure parameters – and mistakes can be made in that process. The system is packaged for release to customers and the release is tested to check that it operates as expected. The software may be released as a cloud service or as a download to be installed on a customer's computer or mobile device. If DevOps is used, then the development team is responsible for release testing; otherwise, a separate team has that responsibility.

18.9.2 What is test-driven development?

TDD (Test-Driven Development) is a software development approach where tests are written before the code implementation. In this approach (used by XP methodology) tests are written before you write code. The pros of this approach are the following:

- ◇ Systematic approach: tests clearly linked to code sections, no untested sections
- ◇ Tests help understanding program code
- ◇ Simplified, incremental debugging
- ◇ (Arguably) simpler code

The cons are the following:

- ◇ Difficult to apply TDD to system testing
- ◇ TDD discourages radical program changes
- ◇ TDD leads you to focus on the tests rather than on the problem you are trying to solve
- ◇ TDD leads you to think too much about implementation details rather than on overall program structure
- ◇ Hard to write "bad data" tests (this should cause the failure of the test)

18.9.3 What are the limitations of testing?

1. *Testing can only show the **presence** of errors, not their **absence***

-cite Dijkstra

2. You test code against your **understanding** of what that code should do. If you have misunderstood the purpose of the code, then this will be reflected in both the code and the tests.
3. Testing may not provide **coverage** of all the code you have written. (TDD shifts the problem to code incompleteness).
4. Testing does not really tell you anything about other **attributes** of a program (e.g. readability, structure, possible evolution).

18.10 What is DevOps automation?

In the DevOps philosophy "Everything that can be automated, should be automated". This is done using:

Continuous integration : each time a developer commits a change to the project's master branch, an executable version of the system is built and tested.
System integration (system building) is more than compiling:

- ◇ install db software and set up db with appropriate schema
- ◇ load test data into db

- ◇ config files
- ◇ check that external services used are operational

To not break the build, you should adopt an "integrate twice" approach to system integration:

1. integrate and test on your own computer
2. push code to project repository to trigger integration server

Advantages:

- ◇ **It is faster to find and fix bugs in the system**: if you make a small change and some system test fails, the problem almost certainly lies in the new code that you pushed,
- ◇ **A working system is always available to the whole team**: It can be used to test ideas and to demo system to management and customers,
- ◇ **"Quality culture" in development team**: No team members wants the stigma of breaking the build and verybody checks her work carefully before pushing it to project repo.

System building tools: Unix make(oldest), Ant, Maven(java), Rake(ruby)... Continuous integration tools: Jenkins (open-source), Travis, Bamboo...

Continuous delivery : executable software is tested in a simulated product's operating environment.Continuous Integration creates an executable version of a software system by building system and running tests on your computer or project integration server, but the *real* production environment will differ from development environment: production server may have different filesystem organization, access permissions, installed applications and new bugs may show up. Continuous delivery ensures that changed system is ready for delivery to customers

- ◇ **feature tests** in production environment to make sure that environment does not cause system failures
- ◇ **load tests** to check how software behaves as number of users increases

Containers are the simplest way to create a replica of a production environment

To deliver:

- ◇ after initial integration testing, configure staged test environment (replica of production environment)
- ◇ run system acceptance tests (functionality, load, performance)

Continuous deployment : new release of system made available to users every time a change is made to the master branch of the software. To deploy:

- ◇ software and data are transferred to production servers
- ◇ new service requests stopped, older version processes outstanding transactions
- ◇ switch to new system version and restart processing

Overall Benefits:

- ◇ **Reduced costs**: Fully automated deployment pipeline
- ◇ **Faster problem solving**: If a problem occurs, it will probably affect only a small part of the system and the source of that problem will be obvious
- ◇ **Faster customer feedback**: You can deploy new features when they are ready for customer use Users' feedback can be used to to identify improvements
- ◇ **A/B testing**: If you have many customers and several servers deploy new software version only on some servers, use load balancer to divert some customers to the new version and measure and assess how new features are used.

Infrastructure as code : machine-readable models of infrastructure (network, servers, routers, etc.) on which the product executes are used by configuration management tools to build the software's execution platform. The software to be installed, such as compilers, libraries and a DBMS, are included in the infrastructure model. When changes have to be made, infrastructure model is updated and CM tool makes the changes to all servers. Manually maintaining a computing infrastructure with tens or hundreds of servers is expensive and error-prone. **Advantages:**

- ◇ **Visibility**: Infrastructure defined as stand-alone model that can be read/understood/reviewed by whole DevOps team.

- ◇ **Reproducibility**: Installation tasks will always be performed in the same sequence, same environment will be always created. You do not have to rely on people remembering the order that they need to do things.
- ◇ **Reliability**: Automation avoids simple mistakes made by system administrators when making same changes to several servers.
- ◇ **Recovery**: Infrastructure model can be versioned and stored in a code management system. If infrastructure changes cause problems, you can easily revert to older version and reinstall the environment that you know works.

Tools: Puppet is a software configuration management tool which includes its own declarative language to describe system configuration.

18.11 What is Domotz approach to software deployment, IaC and monitoring?

Domotz, to **deploy software**, uses containerized applications through Docker containers that ensure portability, performance, isolation, deployment speed and micro-service friendliness, and at the end they automate deployment. **Infrastructure as Code (IaC)** is the process of managing and provisioning computational resources through definition files expressed in a formal language. In this case, Domotz uses a declarative approach, a concept of system state and enforcement of consistency. They use in particular a tool called **SaltStack** based on:

- ◇ States (Declarative configuration)
- ◇ Pillars (Variables)
- ◇ Grains (Host Properties)

If a new version of the container fails and the production service becomes stuck, you can roll back to the previous version of the infrastructure.

For what concerns **monitoring**, they measure everything possible according to the philosophy "if you can't measure it, you can't manage it". So they measure:

- ◇ for the operating system:
  - Hosts (cpu, memory, network, disk)
  - Containers (cpu. memory, network)
  - Processes (cpu, memory, network)
- ◇ for applications:
  - Web Servers (request latency)
  - Databases (query latency, per user stats)

For **cloud monitoring** they use a tool called **ZABBIX** that uses a Server/Agent Architecture.

Other important aspect is that they make extensive use of log files because:

- ◇ **Troubleshooting**: Identify and fix issues through detailed event recording.
- ◇ **Performance**: Monitor metrics for optimal user experience and code efficiency.
- ◇ **Security**: Detect, respond and analyse security incidents with detailed logs.
- ◇ **Compliance**: Essential for auditing, providing adherence to industry standards and regulations.
- ◇ **Scalability**: Analyse logs for capacity planning and efficient application scaling.

Also, US Authorities asked Domotz for logs so having efficient internal tools for logs retrieval is important. They use a tool called **SENTRY** with the following advantages:

1. **is Real Time**: Sentry provides real-time error tracking and monitoring. It captures errors as they occur in applications, allowing developers to respond promptly to issues.
2. **collects error**: Sentry is designed to collect and report errors and exceptions that occur in software applications. It captures detailed information about the errors, including stack traces, environment details, and user context.
3. **gives monitoring insights**: Sentry offers insights into the health and performance of applications by providing detailed error reports. Developers can gain visibility into issues affecting their applications and use this information to diagnose and resolve problems.
4. **runs more or less everywhere** (Android, Apple, Browser Javascript...): Sentry supports a wide range of platforms and technologies. It can be used for error tracking in various environments, including web applications (JavaScript), mobile applications (Android and iOS), and other platforms.
5. **it's the simplest way to have a distributed monitoring tool** Sentry simplifies the process of monitoring and tracking errors in distributed systems. It aggregates error data from different components of an application, providing a centralized view for developers.
6. **advanced monitoring options**: like register the screen to see what's going on, track user UI usage to understand where to improve, run A/B testing to improve UX (and revenues).

18.12 What are the main issues in managing applications in the Cloud-Edge Continuum? What is (an example of) Quantum Software Engineering?

18.12.1 Issues in managing applications in the Cloud-Edge Continuum

Cloud-Edge continuum tries to get the best from both Edge Computing and Cloud computing by extending the cloud towards the IoT into a **distributed, heterogeneus infrastructure**. With this approach you get:

- ◇ computing power
- ◇ connectivity
- ◇ low latency

Next-gen applications will be **Containerised, microservice-based applications deployed on a continuous Cloud-Edge infrastructure.** The main problem that arises from this kind of applications is that things constantly change:

infrastructure constantly changes : Node's workload changes, latency and bandwidth vary, nodes can join and (suddenly) leave, nodes and connections can (temporarily) fail, and so on.
applications change too : codebase's changes, reqs can change.

So there is a need to constantly and suitably **manage** application deployments after first deployment. **Monitoring** is needed because there is a need of effective(lightweight, fault-tolerant) monitoring of both applications and the infrastructure. Monitoring can be done using **continuous reasoning** which exploit compositionality to differentially analyse a large-scale system:

- ◇ by mainly **focusing on the latest changes introduced in the system**, and
- ◇ by **re-using previously computed results** as much as possible

This approach is successfully exploited by Meta's Infer (separation logic) to automate analysis after codebases' changes

. There is the need to consider both application and infastructure changes in the continuum to re-place/migrate, restart, scale application services. During the lesson we've seen **FogBrainX** [1] which is an open-source continuous reasoner written in Prolog which uses a declarative methodology. By analysing differences in the application specification and in the monitored infrastructure data, FogBrainX outputs management decisions on where to place application services by incrementally handling:

[1] https://github.com/di-unipi-socc/fogbrainx

- ◇ changes in the infrastructure (i.e. node resources, network QoS) that trigger the need for migrating one or more application services,
- ◇ changes in the services' (software, hardware and IoT) requirements or in the service-service communication (latency and bandwidth) requirements, set in the application specification, that might trigger either the need for migrating one or more application services or for simply updating the current deployment information, and
- ◇ addition or removal of services or of service-service communication requirements in the application specification.

Then, after the management decisions are passed to **FogArm** which does the managment commands to the Cloud-IoT infrastructure. There are two kind of managements in Cloud-Edge continuum:

Osmotic management Application services can adapt into different functionally equivalent flavours, depending on contextually available resources and on application requirements; the approach is a Bacteria-inspired fully decentralised management:

- ◇ each application instance is associated with an application management agent (bacterion),
- ◇ declarative management policies trigger simple operations (e.g. undeploy, migrate, replicate) based on available monitored data,
- ◇ emerging behaviour of application

makes sense for big infrastructures, more challenging.

18.12.2 Quantum Software Engineering

Quantum software engineering is the use of sound engineering principles for the development, operation, and maintenance of quantum software and the associated document, with the goal of economically obtaining quantum software that is reliable and works efficiently on quantum computers.

- ◇ QSE should be agnostic regarding quantum programming languages and technologies
- ◇ embrace the coexistence of classical and quantum computing: quantum computing should be used only for tasks which are not efficiently solvable on classical computers
- ◇ support the management of quantum software development projects
- ◇ consider the evolution of quantum software
- ◇ aim at delivering quantum programs with desirable zero defects
- ◇ promote quantum software reuse
- ◇ address security and privacy by design
- ◇ cover the governance and management of software

One example of QSE is the **quantum broker** that answers the following questions:

- ◇ *On which Quantum Computer should I run my algorithm?*

This is needed because clients may not be very knowleadgeable about quantum providers, and it provides a list of providers to be chosen. Besides there are other issues which arise when a quantum provider is chosen:

1. What happens if the Quantum Computer becomes unavailable while executing my shots?
   Shots are algorithm instances of the quantum algorithm.
2. How can I mediate between my cost, time and accuracy requirements?
3. How can I customise the Quantum Computer's decision process?

We can consider the **shot dimension** while distributing quantum computations and we can distribute the shots of a circuit among various quantum computers.

When you run a quantum circuit multiple times (multiple shots), you obtain statistical information about the measurement outcomes. Quantum computers introduce a probabilistic element due to the principles of quantum mechanics, and running the same quantum circuit multiple times helps gather statistics on the probabilities of different measurement results.

In practical terms, after executing a quantum circuit with a certain number of shots, you receive a set of measurement outcomes, and the results are represented as counts or probabilities for each possible state.

The partial distributions can then be simply added together—or merged in a more sophisticated way— to generate the whole distribution of the client circuit execution.

Given an input quantum circuit and a set of requirements, the quantum broker **selects the best set quantum computers on which to distribute the shots**. For each selected quantum computer identifies the best compiler(s) and the amount of shots. It allows to spread the shots among multiple quantum computers.

- ◇ improves the **resilience** to quantum computer failures.
- ◇ decouples the distribution policies enabling **high customisation**
- ◇ offers **partial distributions**

Chapter 19

Questions on the labs

19.1 What is the effect of a git add, branch, clone, checkout, push, commit, pull command? How does GitHub flow work?

19.1.1 Git commands

git add :

Effect : Stages changes for the next commit.
Usage :

```
git add <file>              //to stage a file
git add .                   //to stage all changes
```

Explanation : This command adds the changes in the working directory to the staging area, preparing them for the next commit.

git branch :

Effect : Creates a new branch or lists existing branches.
Usage :

```
git branch <branch_name>  //to create a new branch
git branch                //to list local branches
```

Explanation : When used with a branch name, it creates a new local branch. Without a branch name, it lists existing local branches.

git clone :

Effect : Copies a repository from a remote source to the local machine.
Usage :

```
git clone <repository_url>
git clone <repository_url> <local_ directory>
```

Explanation : Creates a local copy of a remote repository, allowing you to work on the code locally (it is possible to specify a branch using the option -b that clone the repository from that specific branch).

git checkout :

Effect : Switches branches, restores files from a specific commit or execute a partial merge.
Usage :

```
git checkout <branch_name>                //to switch branch
git checkout <commit_hash> -- <file>      //to restore a file from prev commit
git checkout <branch_name> -- <file>      //to exec a partial merge
```

Explanation : When used with a branch name, it switches to that branch. When used with a commit hash and a file, it restores the file to the state at that commit. When used with branch name and a file name, it merge the file from the specified branch to the actual branch.

git push :

Effect : Pushes local commits to a remote repository.
Usage :

```
git push <remote> <branch>
```

Explanation : Uploads local changes to the remote repository, making them accessible to others.

git commit :

Effect : Records changes in the repository.
Usage :

```
git commit -m "Commit message"
```

Explanation : Commits staged changes to the local repository, creating a new commit with a specified message.

git pull :

Effect : Fetches changes from a remote repository and merges them into the current branch.
Usage :

```
git pull <remote> <branch>
```

Explanation : Updates the local repository with changes from the remote repository, incorporating them into the current branch.

19.1.2 GitHub Flow

The GitHub flow for software lightweight, branch-based workflow made by the following steps:

1. **Create a "feature" branch from "master":** Start by creating a new branch for the task you're working on. This branch is usually based on the main or master branch. Naming conventions often include a reference to the feature or issue being addressed.
2. **Commit changes:** Make changes and commit them to your branch. Each commit should represent a logical unit of change and should have a clear and concise commit message.
3. **Submit Pull Requests (using the web interface of GitHub):** Once your changes are ready, open a pull request. This is a request to merge your branch into the main or master branch. It allows team members to review your code, provide feedback, and discuss any necessary changes.
4. **Discuss proposed changes and make more commits:** Team members review the code, discuss potential improvements, and address any concerns. This collaboration can happen directly within the pull request through comments and discussions. Based on the feedback received during the review, make any necessary changes to your code by committing them to your branch. The pull request will be automatically updated with the new commits.
5. **Merge "feature" branch into "master":** Once the changes have been reviewed and approved, the pull request can be merged into the main or master branch. This can be done either through the GitHub interface or using the command line.

19.2 What is the effect of a FROM, COPY, ADD and EXPOSE command in a Dockerfile? What are the Docker commands to build an image?

19.2.1 Base commands for Dockerfile

Below are some of the basic instructions used to construct a Dockerfile:

- **FROM**: every valid Dockerfile must start with this command. The command requires at least one required argument, the base image from which to start the build, like shown in the snippet of code below

```
FROM [--platform=<platform>] <image> [AS <name>]
```

The other arguments are all optional: with platform it is possible to specify the target platform where will be deployed the container (e.g., arm64 or amd64), while with the argument AS $< name >$ it is possible to specify the name that can be used to refer to the base image.

- ◇ **COPY**: the command copies file or directory from the $< src >$ path of the local filesystem and adds them to filesystem of the container in the $< dst >$ path

```
COPY [--chown=<user>:<group>] [--chmod=<perms>] <src>... <dest>
```

With the optional argument *chown* and *chmod* is possible to specify the owner of the file/directory and the mode to be applied to the file/directory. These two arguments are available only for Linux containers.

- ◇ **ADD**: the command functions similarly to COPY, but with the added capability of copying remote file URLs.

```
ADD [--chown=<user>:<group>] [--chmod=<perms>] [--checksum=<checksum>] <src>... <dest>
```

The *checksum* option allows for the verification of the checksum of a remote file URL, but it only supports HTTP sources.

- ◇ **EXPOSE**: the command informs Docker that the container listens on the specified network port at runtime.

```
EXPOSE <port> [<port>/<protocol>...]
```

The instruction doesn't actually publish the port, but it informs people who wants to run the container on which are the port intended to be published. To publish the port when running the container, use the -p flag on docker run to publish and map one or more ports.

19.2.2 Build an image

The docker command to build an image is **docker build**. It is possible to apply different flags and options to this command, three common examples are displayed below

```
docker build -t <image_name> .
docker build -t <image_name> . --no-cache
docker build -t <image_name> -f <file_name> .
```

The flag $-t$ allows to specify the tag of the image (e.g., a simple name or a name to publish the container image in a container registry like DockerHub); the use of the flag '–no-cache' specifies that building the image is not allowed to use cache; the flag *-f* allows to use a dockerfile that differs from one with the common name Dockerfile.

19.3 What is minikube? What is a K8s pod/deployment/service? What is the command to create/modify/check a K8s resource?

19.3.1 Minikube

Minikube is an open-source tool that facilitates running Kubernetes clusters locally for development and testing purposes. It provides a simple and lightweight way to set up a single-node or a multi-node Kubernetes cluster on your local machine. Minikube has an internal docker environment, images/questions and containers of your system are different from the ones within it.

19.3.2 Pod, deployment and service

A **Pod** is the smallest and simplest unit in Kubernetes and more pods are the basic building blocks of containerized applications of K8s. A pod has the following characteristics:

- ◇ is composed of one or more (tightly related) containers, each of which is deployed in the same namespace
- ◇ has a shared networking layer, that allows the container to communicate through *localhost*
- ◇ has shared filesystem volumes
- ◇ is assigned a unique IP address within the cluster. Containers within the Pod share this IP address and port space.

A **deployment** is a higher-level abstraction that enables declarative updates to applications. It represents a desired state for a set of Pods and their replicas. Deployments allow you to describe the desired state of your application, and Kubernetes takes care of making the necessary changes to achieve and maintain that state. Deployments are a critical resource in Kubernetes for managing application deployments, updates, and rollbacks.
A deployment is described by template file, in YAML format, that contains the specifications that define the desired state of the Deployment, between these we can find:

- ◇ container images: used to deploy the Pods that compose the deployment;
- ◇ replica count: the number of desired Pods for the application;
- ◇ other configuration settings.

The cluster will always try to have the number of Pods defined by the replica count available (e.g., if we define a deployment with a replica count of 10 and 3 of those Pods crash, 3 more Pods will be scheduled to run in the cluster).

In Kubernetes, a **Service** is an abstraction that defines a set of Pods and a policy by which to access them. Services enable the exposure and discovery of applications running within Pods to other parts of the cluster or external networks. A Service provides a stable endpoint (IP address and port) to direct traffic to the desired Pods even as the exact underlying Pods change due to updates/scaling/failures, this endpoint can be used by other services, both within and outside the cluster. An important aspect of K8s services is the type of services that we want to deploy, there are four different options:

- ◇ **ClusterIP**: exposes the Service on a cluster-internal IP address. This is the default type;
- ◇ **NodePort**: exposes the Service on each node's IP address at a static port. This makes the Service accessible externally;
- ◇ **LoadBalancer**: Exposes the Service externally using a cloud provider's load balancer;
- ◇ **ExternalName**: Maps the Service to the contents of the externalName field, which is a CNAME record pointing to an external DNS name.

19.3.3 Handle resources in K8s

The command to use to create or modify K8s resource is *kubectl apply* and the required argument to be passed is a YAML file that describe the deployment or the service.

```
kubectl apply -f <resource.yaml>
```

The initial execution creates the resource described in the file. If the template is modified by changing certain values and the command is executed again, the resource will be modified accordingly.

To obtain the status of a resource, there are two different commands available: *kubectl get* or *kubectl describe*.

```
kubectl get <resource-type> -n <namespace>
kubectl get <resource-type> -A
```

This command is used to retrieve the status of the resources of a specific type, such as nodes, deployments, services, or pods. With the flag *-n* for a specific namespace, while with the flag *-A* for all the namespaces.

```
kubectl describe <resource-type> <resource-name> -n <namespace>
```

This command, on the contrary of kubectl get, show the detail description and the status of a specific resource in a specific namespace.

19.4 What can you use MicroFreshener for?

Microfreshner is a tool for:

- ◇ editing app specifications
- ◇ automatically identifying architectural smells
- ◇ applying architectural refactorings to resolve the identified smells

Using MicroFreshner you can:

- ◇ draw microservice architectures
- ◇ analyse the architecture graph to identify smells
- ◇ suggest refactoring to resolve smells
- ◇ apply the selected refactoring to the architecture

The use of MicroFreshner doesn't allow you to change effectively the deployment of the microservice, so in order to do that, after you've done the refactoring of the architecture, you have to fix the smells in the micro-service application by writing code.

Microfreshner allows you to import a manifest.yaml that represents a k8s cluster, then you can analyse the architecture using microfreshner to find smells and (eventually) apply a refactoring. After that, you can save the updated architecture in an updated manifest file.

19.5 What is vulnerability avoidance with static security analysis? What are false positives/negatives? What are Bandit's severity and confidence?

The **vulnerability avoidance with static analysis** is based on the analysis of the system (source code or its representation) to check some property without running it. This is often done through static code analysis tools like bandit, which can automatically analyze the codebase for potential security issues, coding errors, or adherence to security best practices.

False Positives: These occur when a static analysis tool incorrectly identifies a piece of code as vulnerable or problematic, when in fact it is not. False positives can lead to unnecessary effort and time spent on investigating and addressing non-issues.

False Negatives: These occur when a static analysis tool fails to identify a genuine security vulnerability or coding issue. False negatives are concerning because they may result in real security risks being overlooked, leaving the application susceptible to attacks.

Bandit is a static analysis tool designed to find common security issues in Python code, by exploiting known patterns (plugins). Bandit was originally developed within the OpenStack Security Project and later re-homed to PyCQA. It recognizes 70 vulnerabilities out-of-the-box. To install and use bandit run the following code:

```
pip install bandit
bandit -r <path_to_code>
```

Bandit categorizes its findings into:

severity levels (Low, Medium and High) to prioritize and address issues based on their potential impact. For example, a high severity vulnerability might indicate a potentially serious threat to security, while a low severity one might pose a less critical risk; and

confidence levels (Low, Medium and High) for each result, indicating the tool's level of certainty about the identified issue. A higher confidence score means there is greater certainty that the report is accurate.

19.6 What is dynamic security analysis? What is OWASP ZAP? What is WebGoat?

Dynamic security analysis refers to the process of evaluating the security of a running application by actively testing it. Dynamic analysis interacts with the running application to identify potential security vulnerabilities. In order to do this, it generates various types of input parameters to trigger as many execution flows as possible.

OWASP ZAP is a tool for penetration testing that can function as both a vulnerability scanner and a proxy. Its ability to scan for vulnerabilities makes it a valuable tool for dynamic analysis of web applications and each scan can be composed of many phases, like spidering/crawling, active and passive scan and also fuzzing. ZAP acts as a proxy between the client app and the server. **WebGoat** is a deliberately insecure web application maintained by OWASP designed to teach web application security lessons. While running this program the machine is extremely vulnerable, so you should disconnect from the internet while using it. The program is for educational purposes only.

19.7 What is Kube-hound? What is the UPM/IAC smell? What is OpenAPI?

19.7.1 Kube-hound

Kube-hound is a tool to detect security smells in Kubernetes-based microservice applications exploiting both static analysis and dynamic anaylysis.

19.7.2 UPM/IAC smell

The **UPM smell** (Unnecessary Privileges to Microservices) refers to situations where a microservice is granted more permissions or access rights than it actually needs to fulfill its designated responsibilities. This can pose security risks and increase the potential impact of security breaches (e.g. service can read/write data in db or messages in queue even if db or queue not needed by the service to deliver its business function).**Consequences**: Resources are unnecessarily exposed, resulting in increased attack surface against confidentiality and integrity properties. **suggested refactoring**: Follow **least privilege principle** which says "allow running code only the permissions needed to complete the required tasks and no more".

IAC smell (Insufficient access control) is a security vulnerability that occurs when a system or application fails to properly enforce restrictions on user permissions, allowing unauthorized users or processes to access sensitive resources or perform actions they should not be allowed to execute. **Consequences**:

- ◇ Potential "confused deputy problem" with attacker getting data it shouldn't be able to get
- ◇ Potential violation of **confidentiality** of data (and business functions)

Suggested refactoring:

- ◇ Client permissions need to be verified at request time and client identity should be verified without introducing extra latency and contention with frequent calls to a centralized service.
- ◇ The most suggested refactoring is employ **OAuth 2.0**, a token-based security framework for delegated access control. Resource owner can grant client access to a resource on its behalf and the access is granted for limited time and with limited scope.

19.7.3 OpenAPI

OpenAPI, formerly known as Swagger, is a specification for building APIs (Application Programming Interfaces). It provides a standard way to describe RESTful APIs, making it easier for both humans and computers to understand the capabilities of a service without accessing its source code or documentation. It allows to define:

- ◇ API Specification: OpenAPI allows developers to define their APIs in a machine-readable format using JSON or YAML.
- ◇ Documentation: OpenAPI specifications can be used to generate interactive and human-readable documentation for APIs.
- ◇ Code Generation: OpenAPI specifications can be used to generate client libraries, server stubs, and other code artifacts.

19.8 What is Camunda? What are the two "usage patterns" of Camunda?

Camunda is a framework supporting BPMN for workflow and process automation. It provides a RESTful API which allows you to use your language of choice. Workflows are defined in BPMN which can be graphically modeled using the Camunda Modeler. **The 2 usage patterns of camunda** are:

- ◇ endpoint-based integration
- ◇ queue-based integration

In **endpoint-based integration** after defining a BPMN process, Camunda can directly call services via built-in connectors. It supports REST. However, it only allows scaling on process instances, NOT on microservices.

Recall that —in the lab exercise—:

1. The RESTful API calls were hidden and managed by `pycamunda` package
2. workers did not pass explicitly parameters to each other, they "subscribed" to *topics* instead, accessing dictionary keys.

```
worker.subscribe(
    topic='TroopFight',
    func=fight,
    variables=['legion',"defendants","legionS","defendantsS",'capital']
)
```

The **endpoint-based** interaction *smell* occurs in an application when one or more of its microservices invoke a specific instance of another microservice, e.g. because it's using its hardcoded location, or because no load balancer is used; scaling out the latter microservice by adding new replicas would be pointless since they would not be reachable by the invokers.

In **queue-based integration** Units of work (Tasks) are provided in a Topic Queue that can be polled by RESTful workers, possibly interacting with microservices. It allows scaling of process instances, workers and microservices.

19.9 How can you do unit/load tests with microservices? How does Locust work?

Unit tests are designed to verify that individual units or components of a software application perform as intended.To perform unit testing of Python-based microservices, one can use the *Pytest* library. In Python-based microservices architecture, functions and classes are usually defined in order to implement the expected microservice workflow. These functions and classes can be unit-tested in isolation and to achieve this goal calls to a class are mocked. To execute the unit-test is necessary to define a test file within the project's test directory. This file has to contain a number of functions equals to the number of unit tests that is necessary to execute. The single unit test uses the 'assert' method to verify that the expected results are returned.

In contrast to unit tests, **load tests** are designed to identify bottlenecks in a service under stress. Even in this case, to test Python-based microservices is possible to use **Locust**, an open-source load testing tool used by Big Companies. A locustfile.py is required in the root project folder to define user behaviours. You can run Locust using the Locust command in the root folder of the project, browse to http://localhost:8089, set up and run your tests. Then, you can analyse locust's stats and graphs, spot the bottlenecks endpoint, if any, and check the code of the endpoints in the gateway to resolve the possible problem.

19.10 What is Jenkins? What is a Jenkins pipeline? How does Jenkins exploit Git?

Jenkins is an open-source automation software, that aims to facilitate CI/CD. It assists in automating the aspects of software development that pertain to building, testing, and deploying.

BlueOcean is Jenkin's UI. It indicates where attention is needed in the pipeline, making exception handling easier and increasing productivity.

A Jenkins **pipeline** is a set of automation processes and configurations expressed in code that defines the steps, stages, and actions for continuous integration and continuous delivery (CI/CD) in Jenkins. This pipeline are depicted by a text file, the **Jenkinsfile**.
A *stage* is composed by ***steps***, plus other stuff e.g. *agent*, *environment*, *post*, ...
In the lab the three stages were:

1. Build
2. Test
3. Deliver
    i. Was the artifact generation included in this stage or is independent?

```
stage('Deliver') {...
post {
    success {
        archiveArtifacts "${env.BUILD_ID}/sources/dist/add2vals"
        sh "docker run --rm -v ${VOLUME} ${IMAGE} 'rm -rf build dist'"
    }
}}
```

Jenkins supports Git integration to automate the building, testing, and deployment of code changes. Jenkins utilizes Git as its **SCM** (source code management) system. Jenkins can connect to a Git repository through the SCM to retrieve the source code of an application. This is often the first step in a Jenkins pipeline. Jenkins supports both Git and GitHub repositories, and you can configure the repository URL and credentials in the Jenkins job or pipeline configuration.

Jenkins can be configured to detect changes in a Git repository using either webhooks or polling, with the first one more efficient and provide real-time updates. When a change is detected, Jenkins automatically triggers a build job to start the CI/CD pipeline.

Jenkins enables the configuration of build triggers based on Git events. For instance, a build job can be triggered when changes are pushed to a specific branch or when a pull request is opened or merged. Additionally, test and delivery jobs can be added to the pipeline, as seen during the lab lesson.