Advanced Software Engineering - Appunti

Francesco Lorenzoni

September 2023

Contents

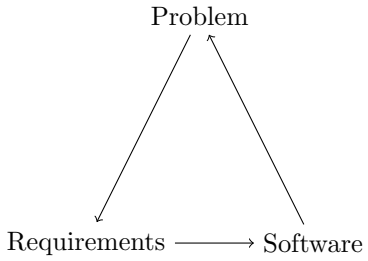
Chapter 1

Introduction

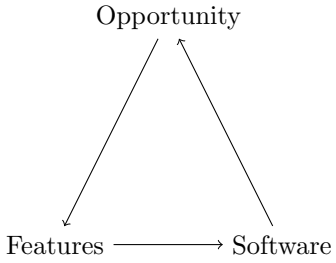
27 - Settembre

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 porjects 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 dev
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

... **Prod Backlog Revised**
PBI Estimation Metrics

1.3.1 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.

1.3.2 Scrum Meetings

1.3.3 Agile activities

Test automation Continuous integration

1.3.4 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.

Team size

1.4 5 - Ottobre

Chapter 2

Features, Scenarios, Stories

2.1 3 - Ottobre

Which factor 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**:

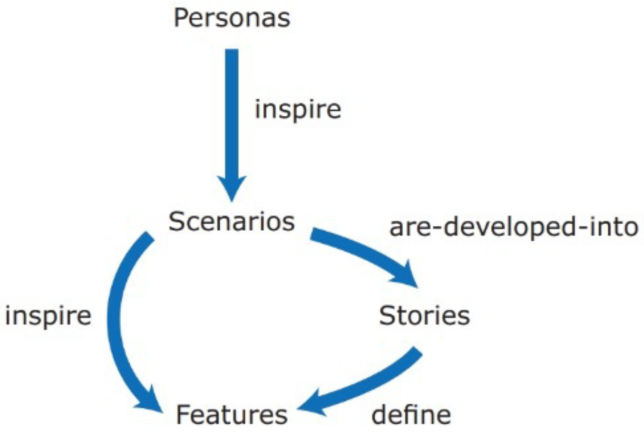
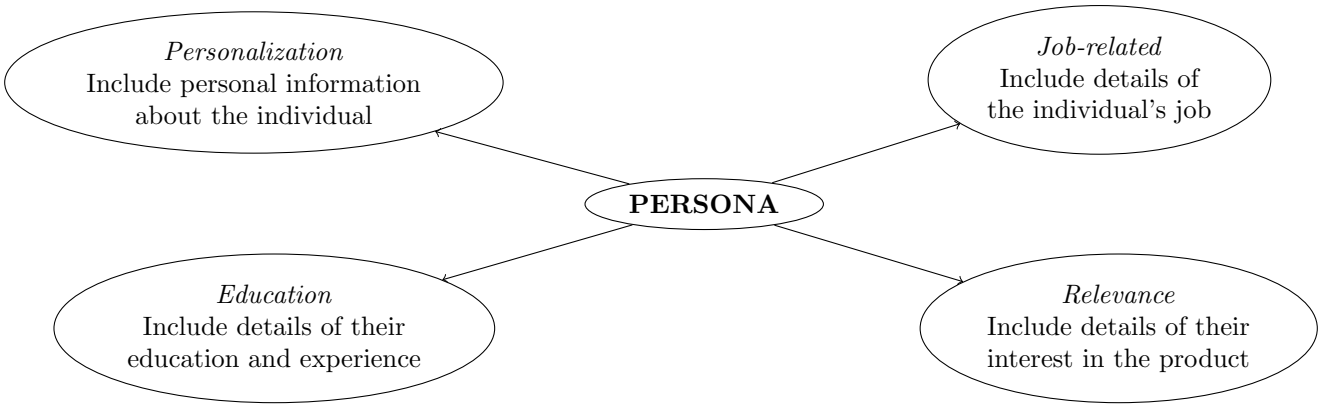


Figure 2.1: Features flowchart

2.2 Personas

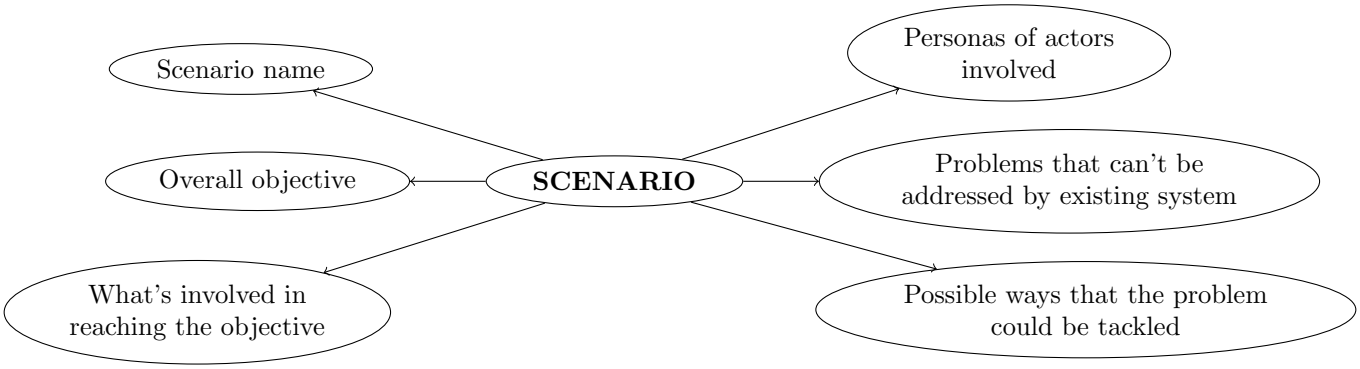
Personas represent the types of target users for our product. Each personas should highlight which are *background*, *skills* and *experience* of potential users. Usually only a couple of personas (max 5) are needed to identify ***key*** *product features.*



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).

2.3 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.

2.4 User Stories

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

Table 2.1: User Stories

Knowledge sources for feature design

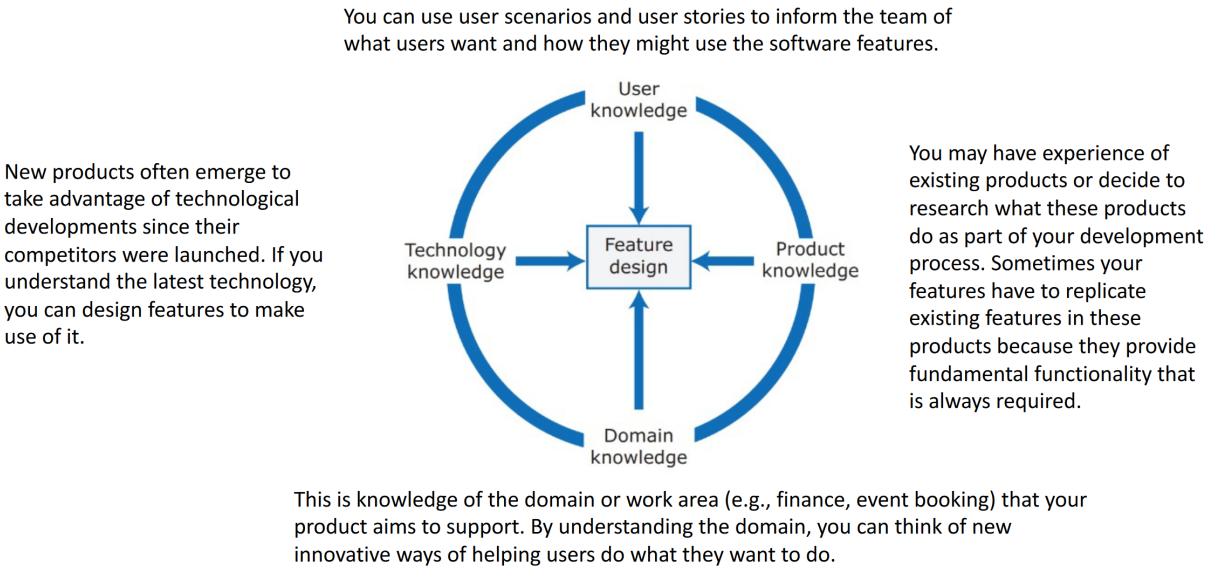


Figure 2.2: feature knowledge

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

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

2.5 Feature identification

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

- **Independence** → a feat should not dfepend on how other system feats are implemented and should not be affected by the order of activation of other feats
- **Coherence** → feats should be linked to a single item of functionality. They should not do more than one thing and should never have side effects
- **Relevance** → systesm feats 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.

2.5.1 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.

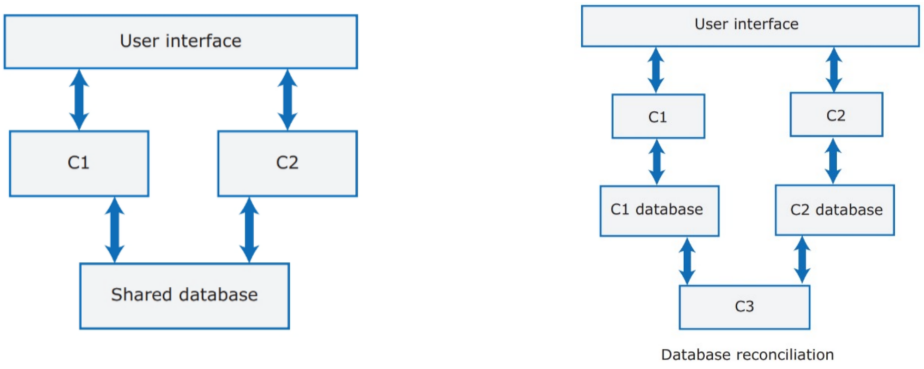


Figure 5.1: Centralized vs Component DBs

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

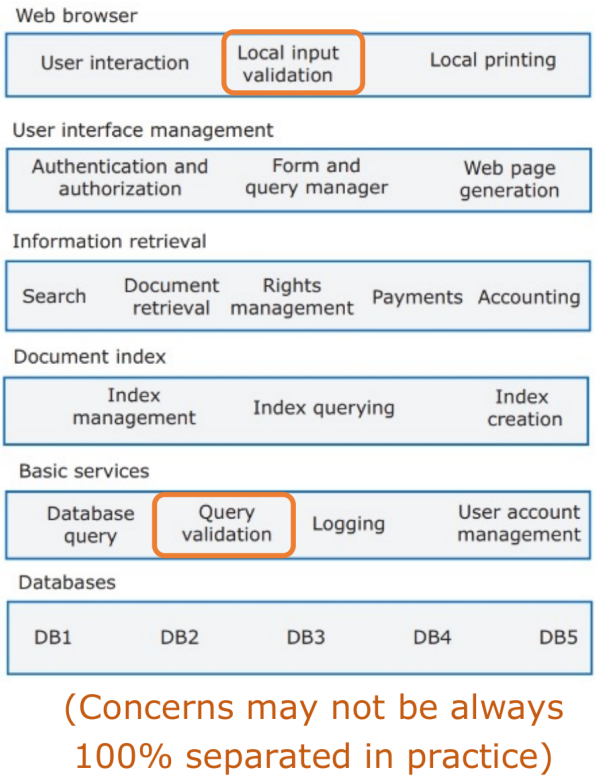


Figure 5.2: Layered architecture

"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.

System decomposition must be done in conjunction with choosing technologies for the system. In this there is a mixup happening between implementation and design, however 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 trasnlate 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 syle 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, *normalizer* are 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

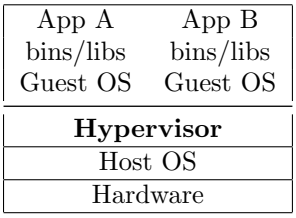
19 - Ottobre

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.

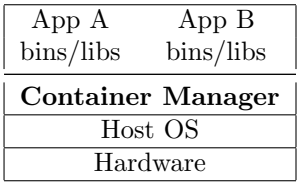
- ◇ *Scalability*
- ◇ *Elasticity*
- ◇ *Resilience*
- ◇ *Cost*

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 SO. 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 makor 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

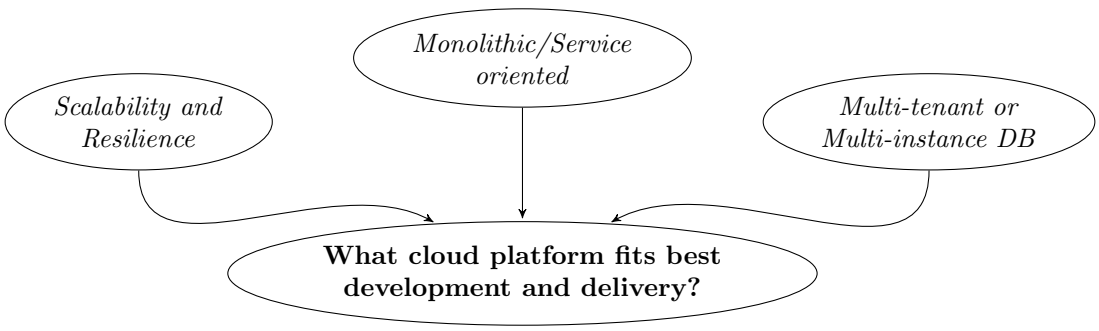
Multi-instance may mainly be **VM**-based or **Container**-based.
With *containers*, each user has an **isolated** version of software and database running in a set of container, defining a solution perfect for products whose users work onto independently from others. Besides, since there is no direct 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



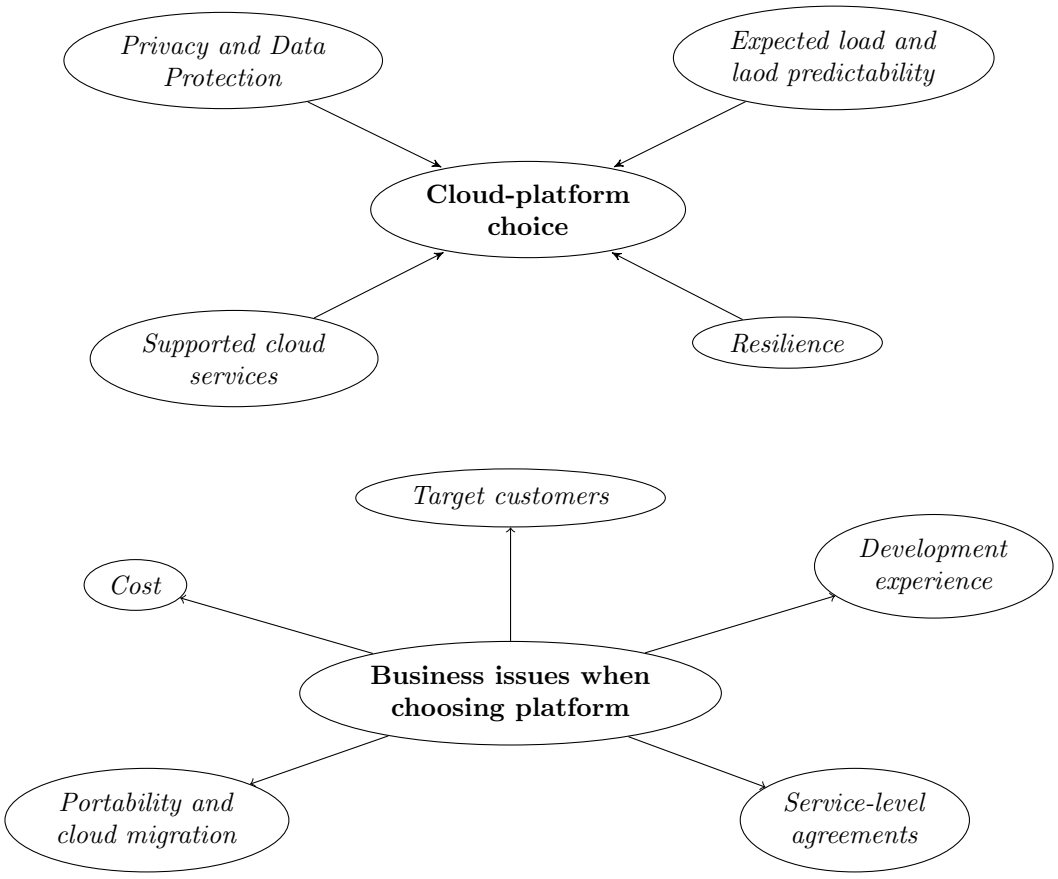
8.3.1 Scalability

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

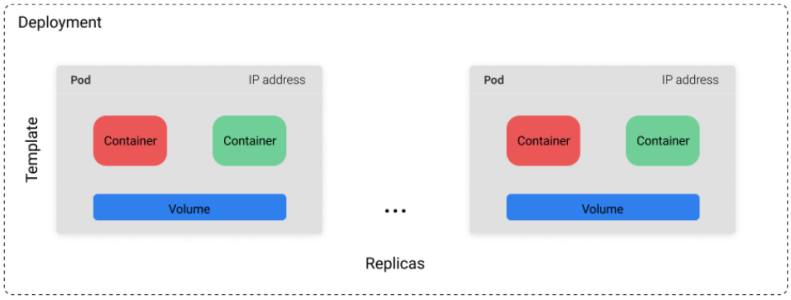


Figure 9.1: Deployment Object

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?

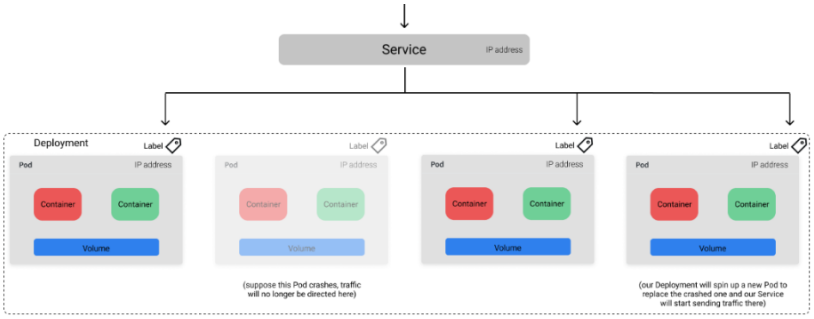


Figure 9.2: Service Object

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.

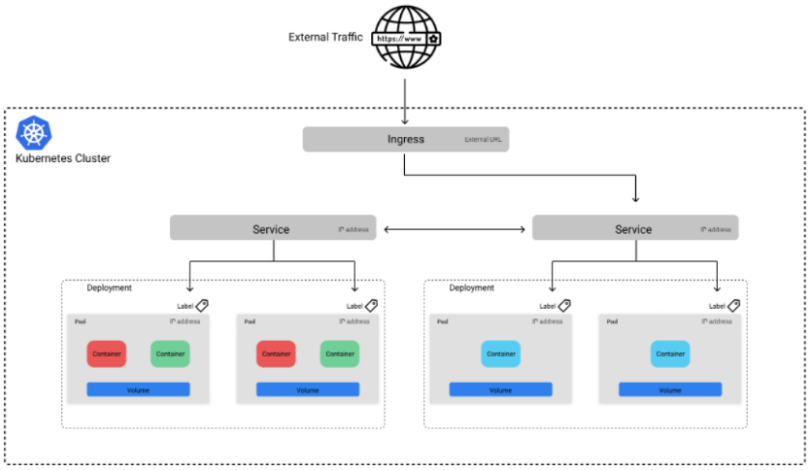


Figure 9.3: Ingress Object

Others - 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 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* ⟶ 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

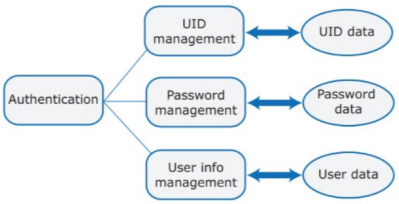


Figure 10.1: Result of microservices identification in our authentication example

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). May 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

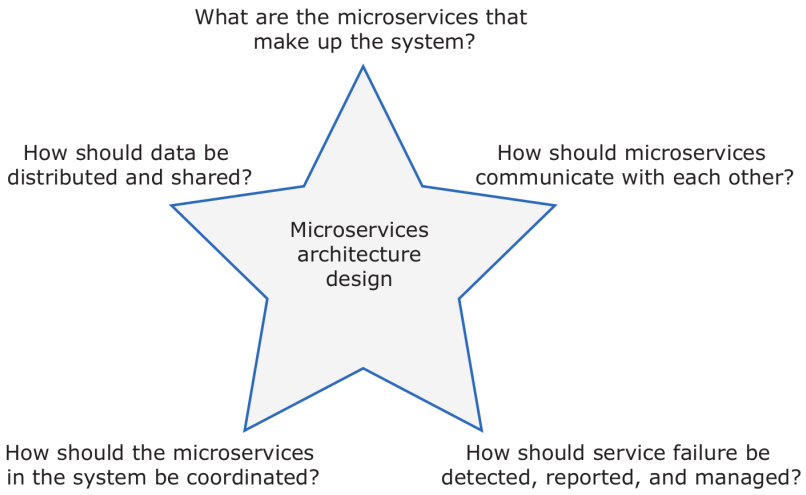


Figure 10.1: Microservices Decisions

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*

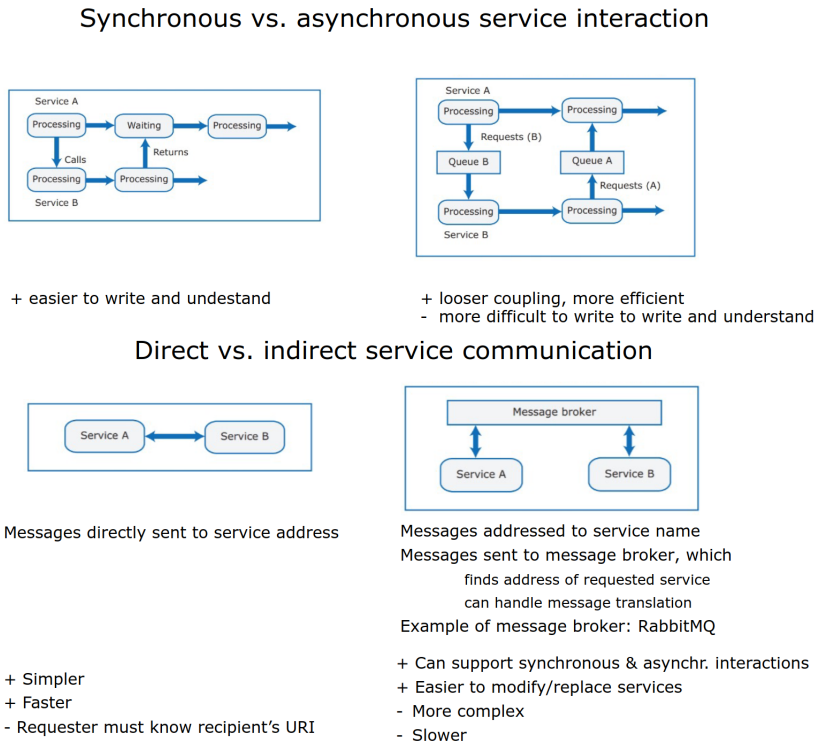


Figure 10.2: Microservices communication

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.

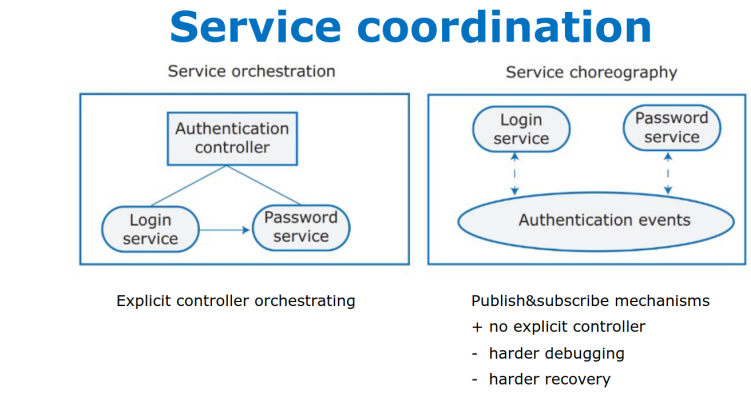


Figure 10.3: Netflix approach on Coordination

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.

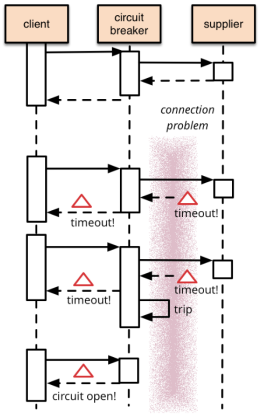


Figure 10.4: Circuit breaker

10.7 RESTful services

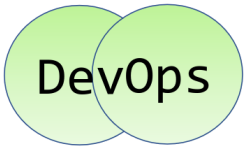
*RE*presentational *S*tate *T*ransfer (**REST**) was originally introduced as an architectural style, and then it has developed as an abstract model of the Web architecture to guide the redesign and definition of HTTP and URIs.

"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

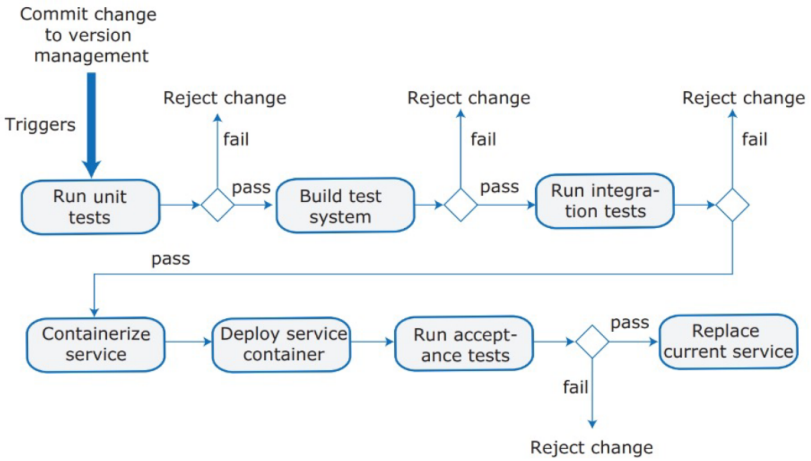


Figure 10.5: Continuous deployment pipeline

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]

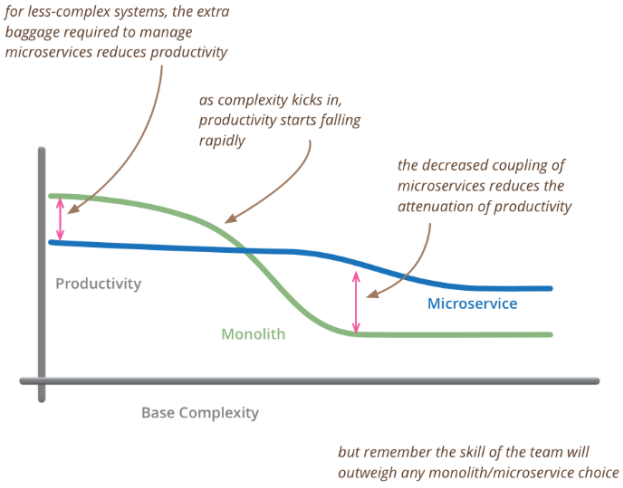


Figure 10.6: Takeaway message