Advanced Software Engineering - Appunti

Francesco Lorenzoni

September 2023

Contents

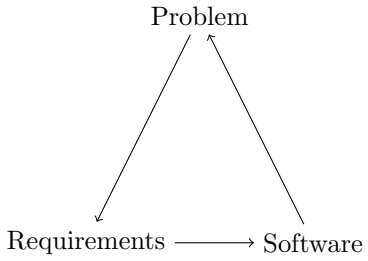
Chapter 1

Introduction

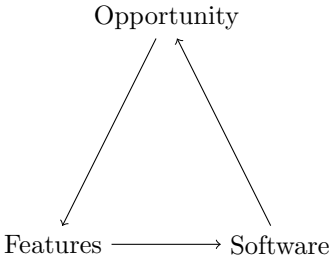
27 - Settembre

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.

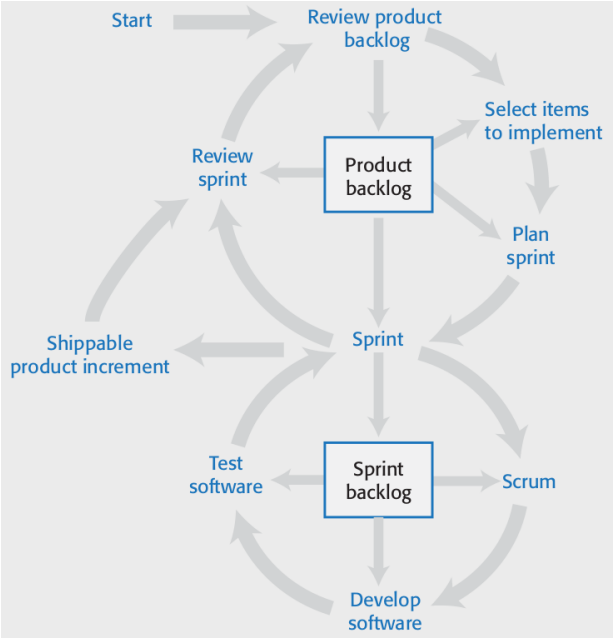


Figure 1.1: Scrum cycles

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**:

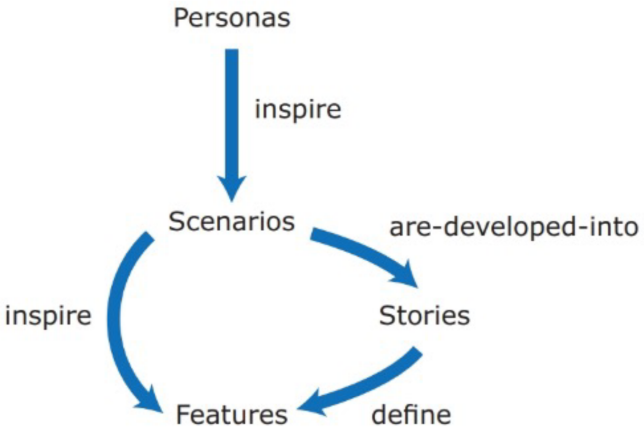
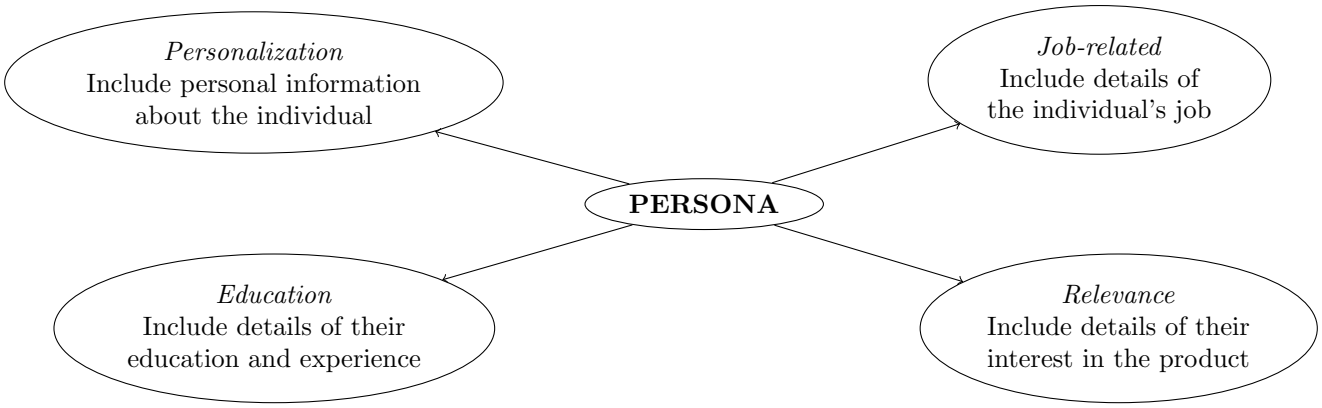


Figure 2.1: Features flowchart

2.1 Personas

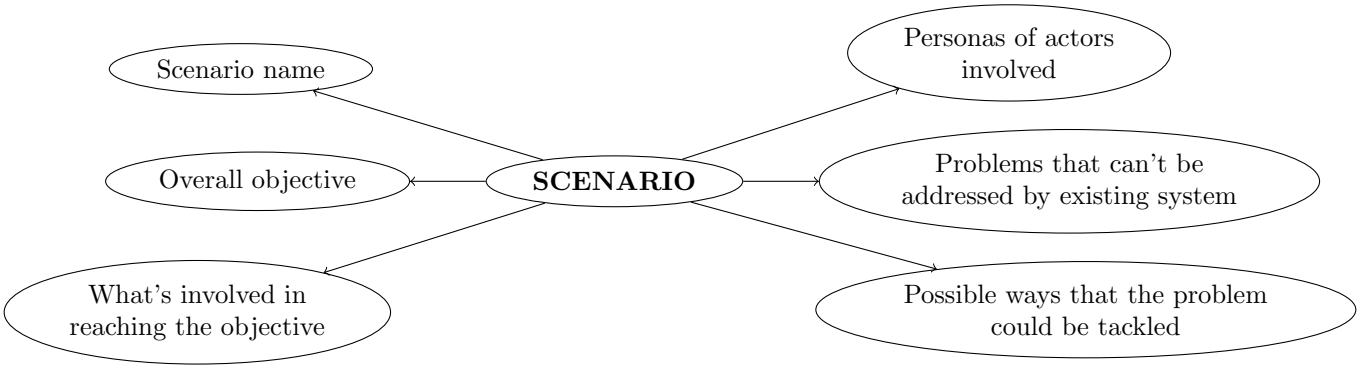
Personas represent the types of target users for our product. Each personas should highlight which are *background*, *skills* and *experience* of potential users. Usually only a couple of personas (max 5) are needed to identify ***key*** *product features.*



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

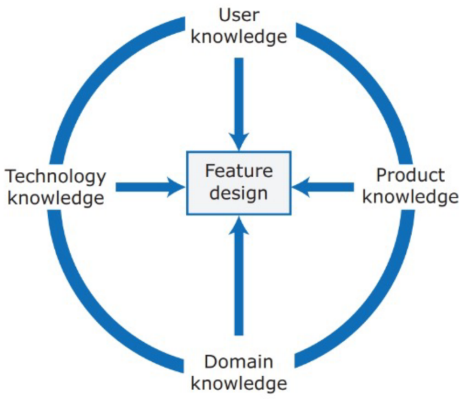
As a	<role>
I want to	<do something>
So that	<reason/values>

Table 2.1: User Stories

Knowledge sources for feature design

You can use user scenarios and user stories to inform the team of what users want and how they might use the software features.

New products often emerge to take advantage of technological developments since their competitors were launched. If you understand the latest technology, you can design features to make use of it.



You may have experience of existing products or decide to research what these products do as part of your development process. Sometimes your features have to replicate existing features in these products because they provide fundamental functionality that is always required.

This is knowledge of the domain or work area (e.g., finance, event booking) that your product aims to support. By understanding the domain, you can think of new innovative ways of helping users do what they want to do.

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.

2.4 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** → systesm 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

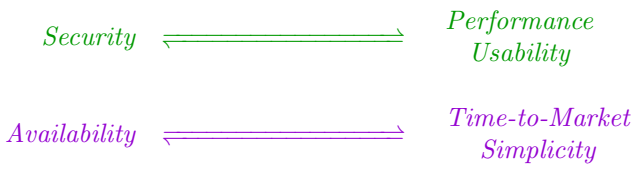
- ◇ Non-functional product characteristics: security, reliability, availability... These aspects are as important as functional properties, to develop a successful product.
- ◇ Product lifetime: in case of developing a hopefully long-term product, its architecture must be able to evolve and adapt: *microservices*, 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.



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.

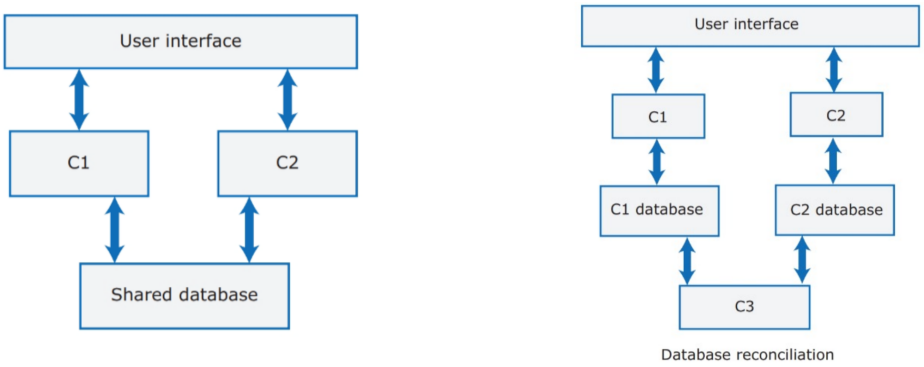


Figure 5.1: Centralized vs Component DBs

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.

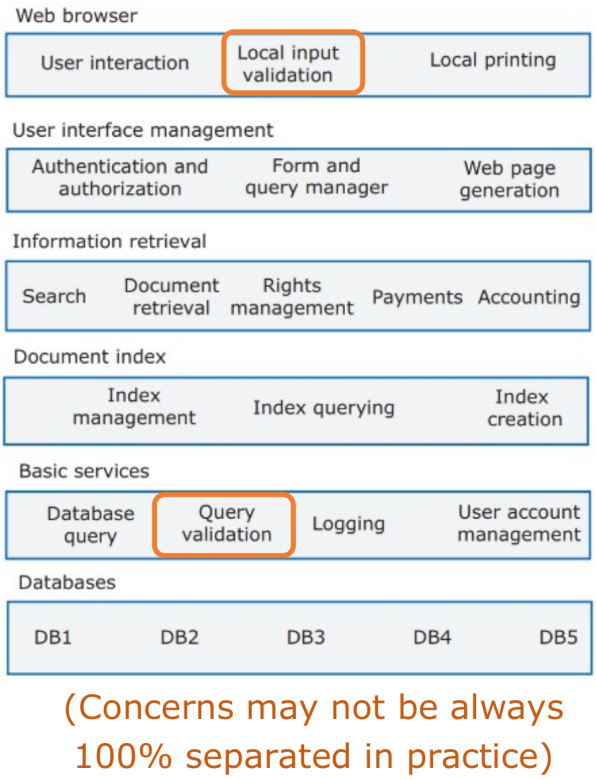


Figure 5.2: Layered architecture

System decomposition must be done in conjunction with choosing technologies for the system. 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 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

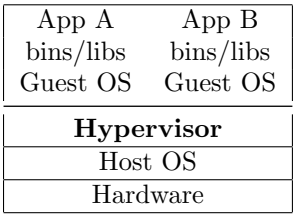
19 - Ottobre

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.

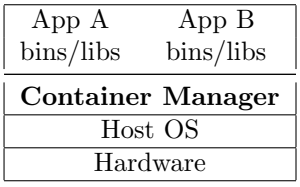
- ◇ *Scalability*
- ◇ *Elasticity*
- ◇ *Resilience*
- ◇ *Cost*

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 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 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

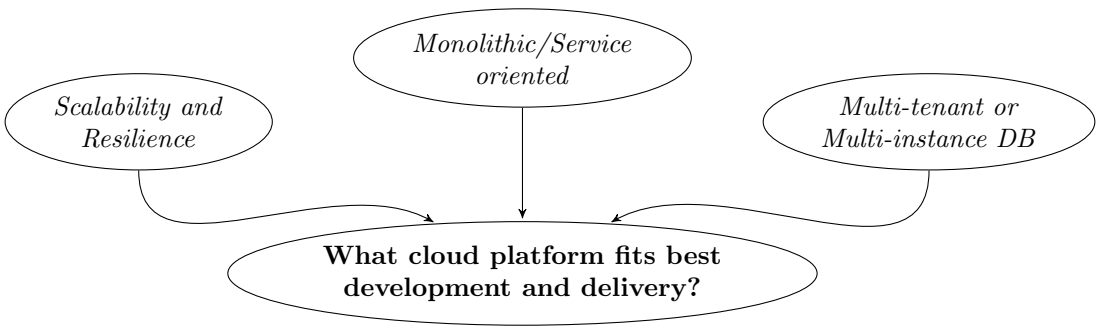
Multi-instance may mainly be **VM**-based or **Container**-based.
With *containers*, each user has an **isolated** version of software and database running in a set of container, defining a solution perfect for products whose users work onto independently from others. Besides, since there is no direct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



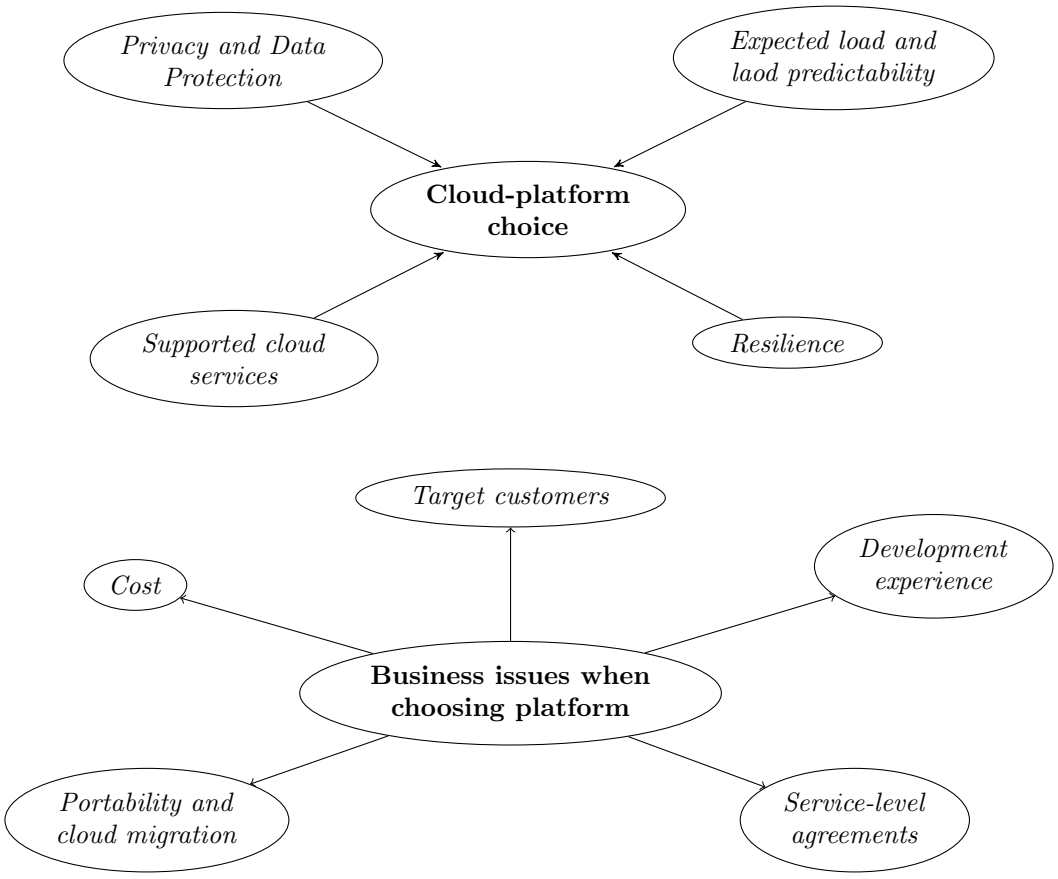
8.3.1 Scalability

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

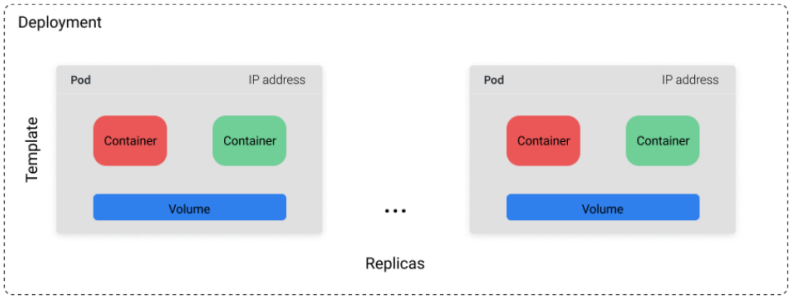


Figure 9.1: Deployment Object

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?

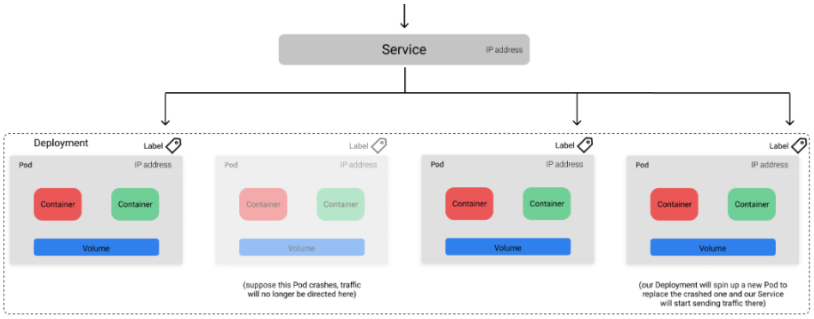


Figure 9.2: Service Object

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.

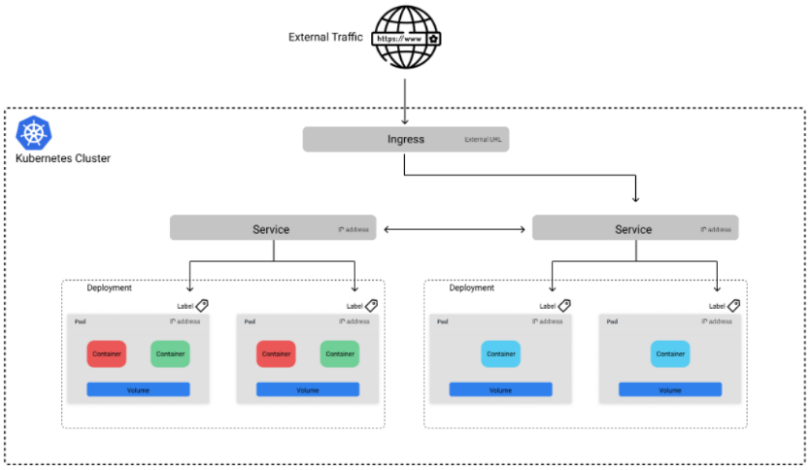


Figure 9.3: Ingress Object

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

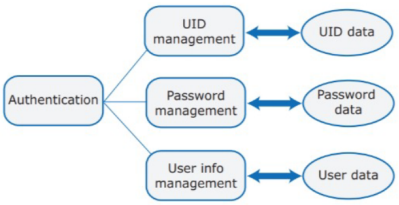


Figure 10.1: Result of microservices identification in our authentication example

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). 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

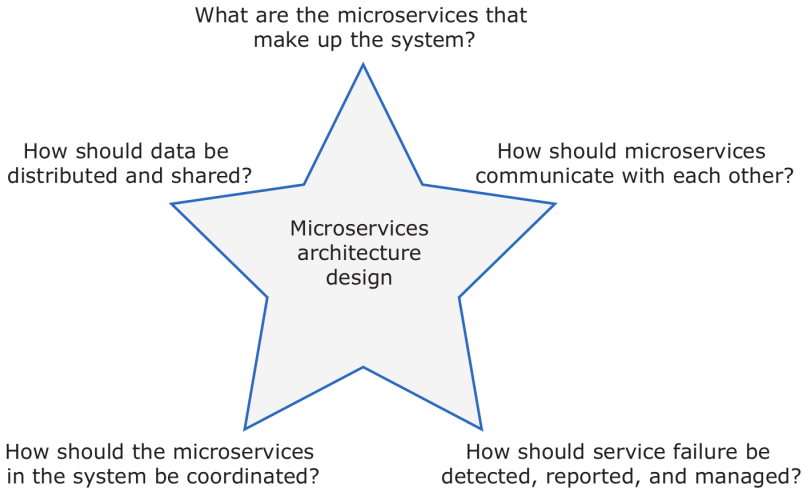


Figure 10.1: Microservices Decisions

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*

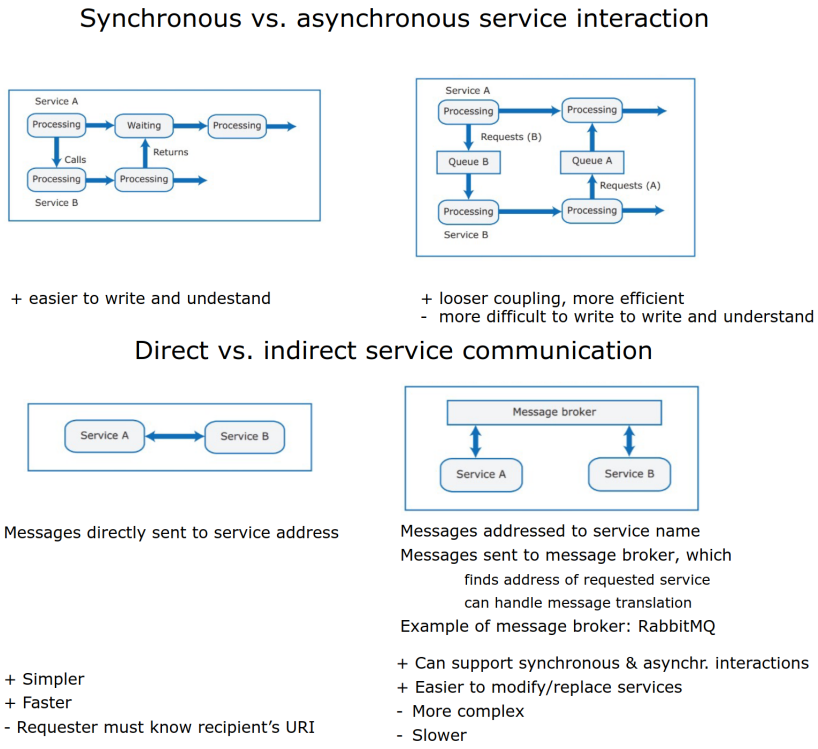


Figure 10.2: Microservices communication

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.

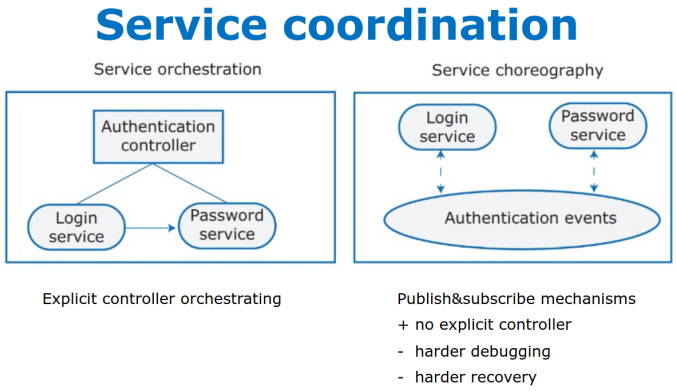


Figure 10.3: Netflix approach on Coordination

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.

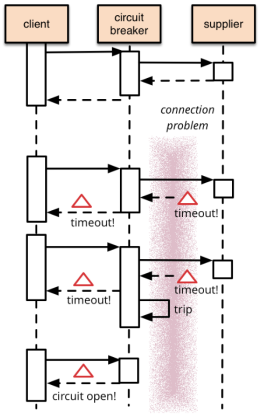


Figure 10.4: Circuit breaker

10.7 RESTful services

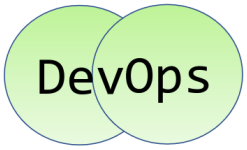
*RE*presentational *S*tate *T*ransfer (**REST**) was originally introduced as an architectural style, and then it has developed as an abstract model of the Web architecture to guide the redesign and definition of HTTP and URIs.

"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

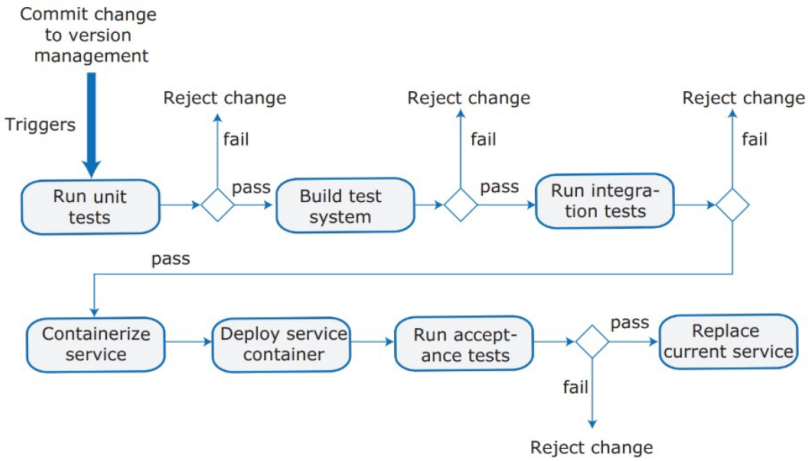


Figure 10.5: Continuous deployment pipeline

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]

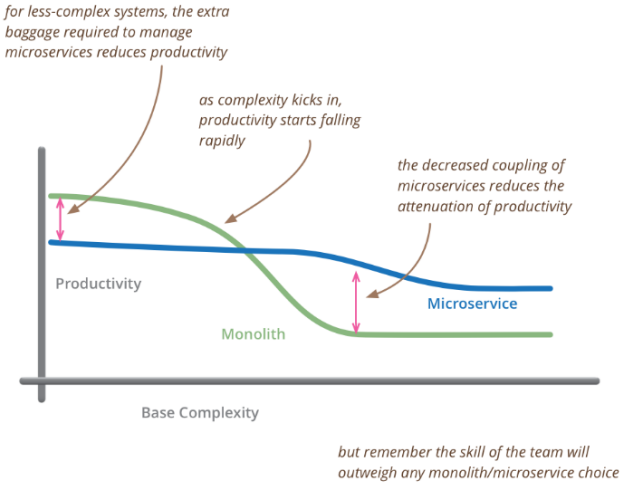


Figure 10.6: Takeaway message

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 features**: 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

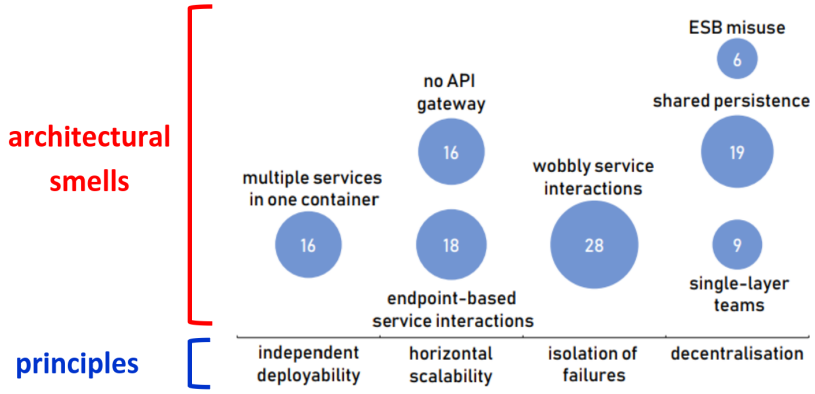


Figure 11.1: Smells Principles
Here we can see smells, i.e. wrong choices, and the principle they violate

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

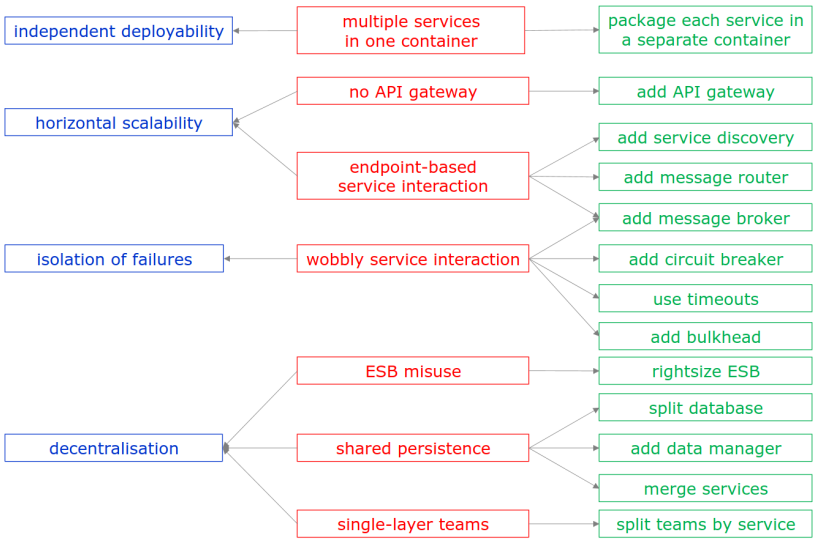


Figure 11.2: Solutions

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

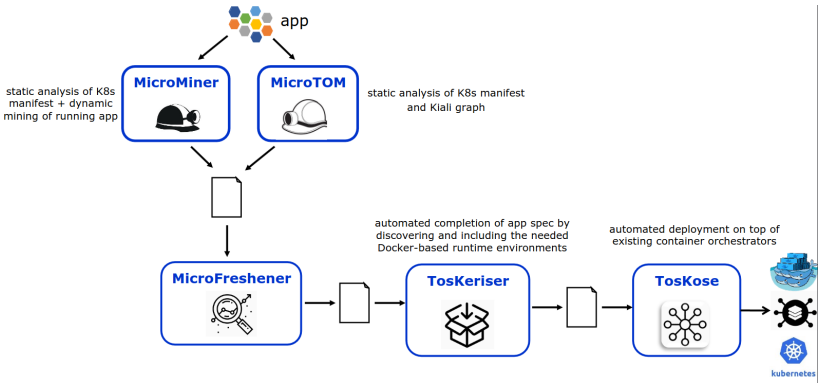


Figure 11.3: Microservices toolchain

Chapter 12

Security and Privacy

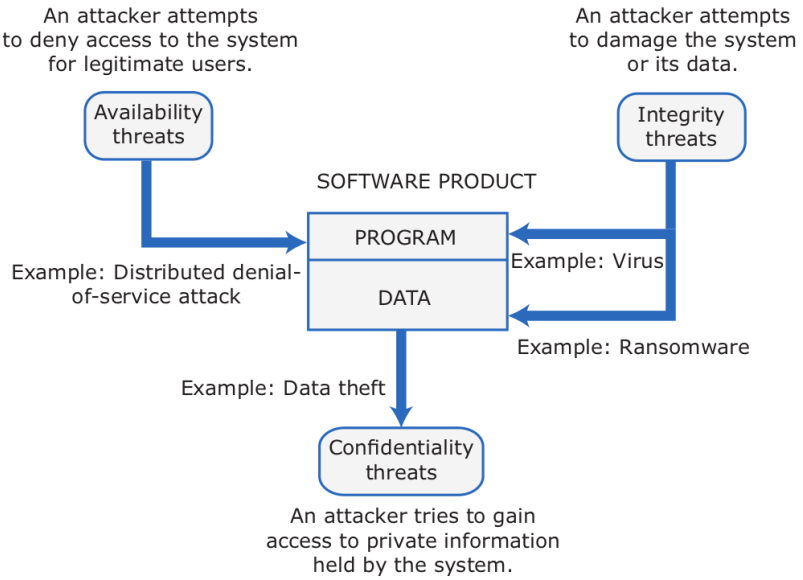


Figure 12.1: Security threats

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.

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.

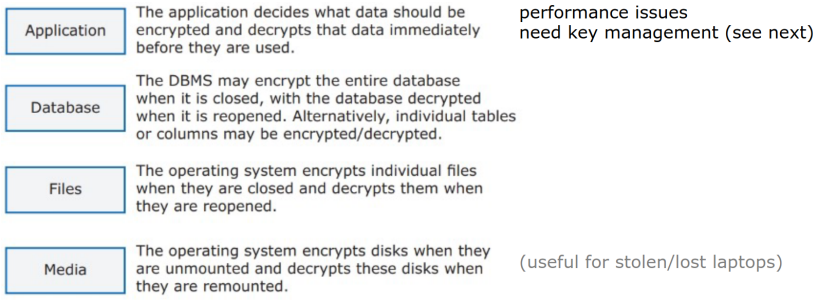


Figure 12.2: Encryption is possible in all system levels

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

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.

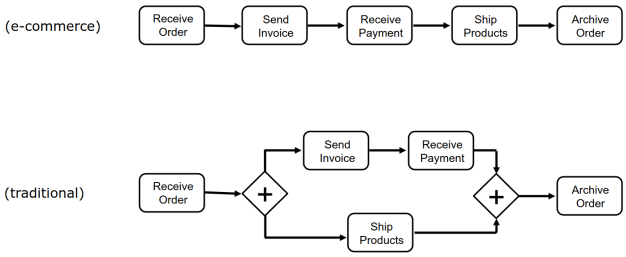
The following is an example of a BP model descripted in natural language:

"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

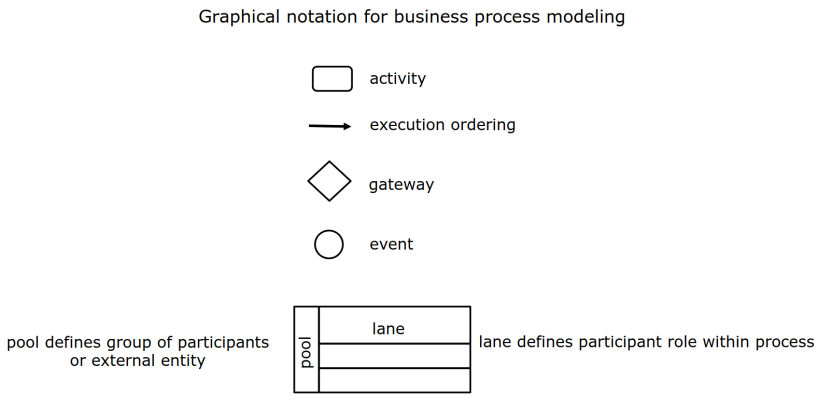


Figure 13.1: Business process model Notation

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.

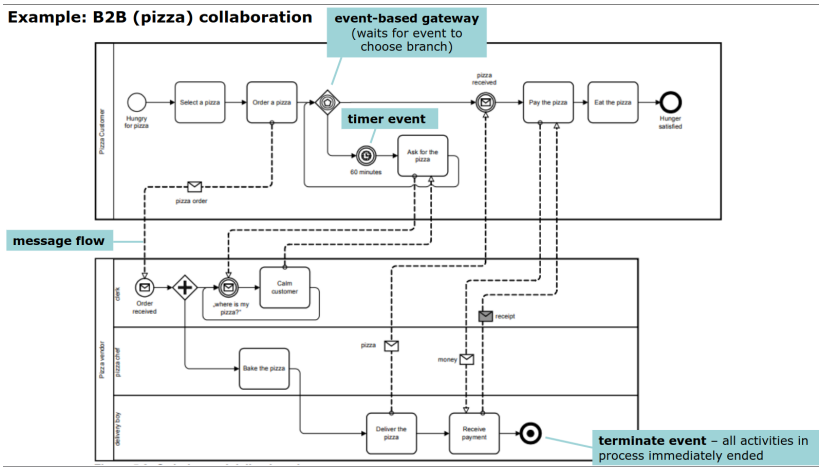


Figure 13.2: Pizza BPM example

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.

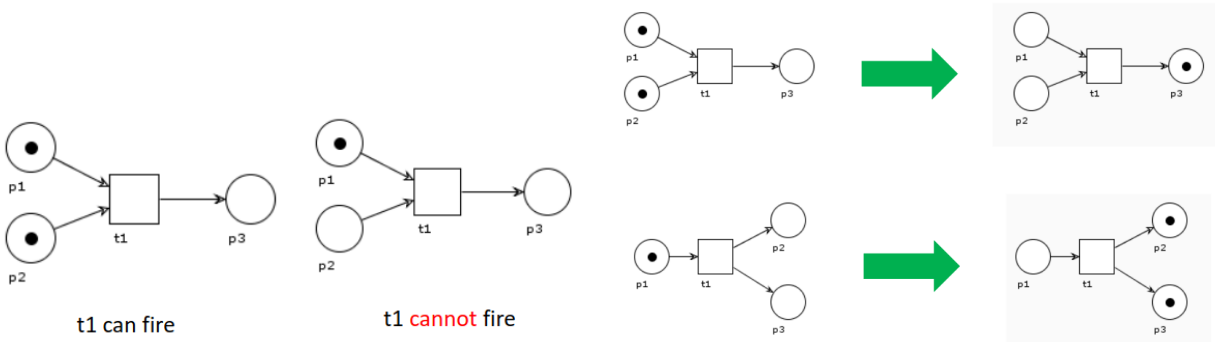


Figure 13.3: Transition firing

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.

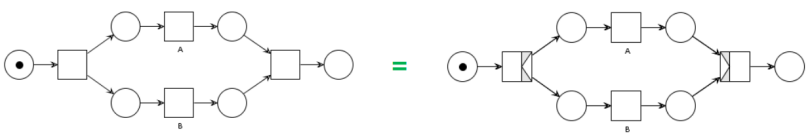
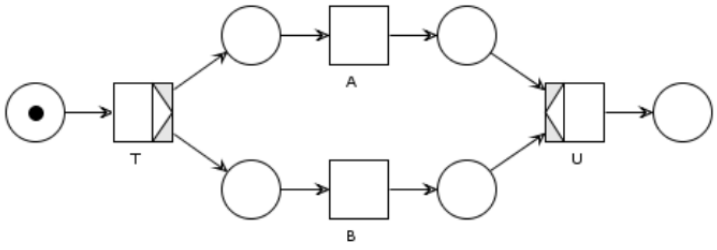


Figure 13.4: Explicit `AND-split` and `AND-join` transitions



explicit OR *("τ.A+τ.B")*

Extending PNs:
-T will pace **only one** token in one of its output places
-U can fire if (at least) **one** of its input places contains a token

Figure 13.5: Explicit `OR`

Soundness

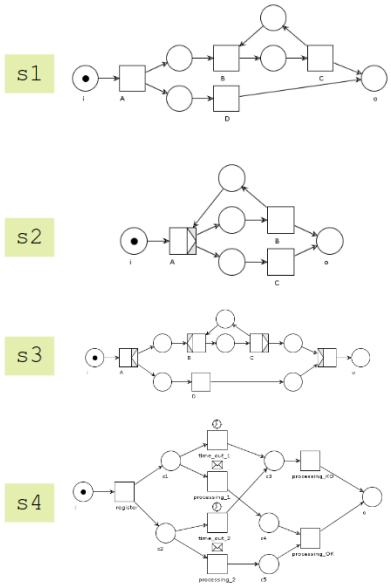
A workflow net is –unformally– **sound** `iff`:

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



	(1)	(2)
s1	✗	✗
s2	✗	✗
s3	✓	✓
s4	✗	✓

Figure 13.6: Soundness check with examples

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.

not live live

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

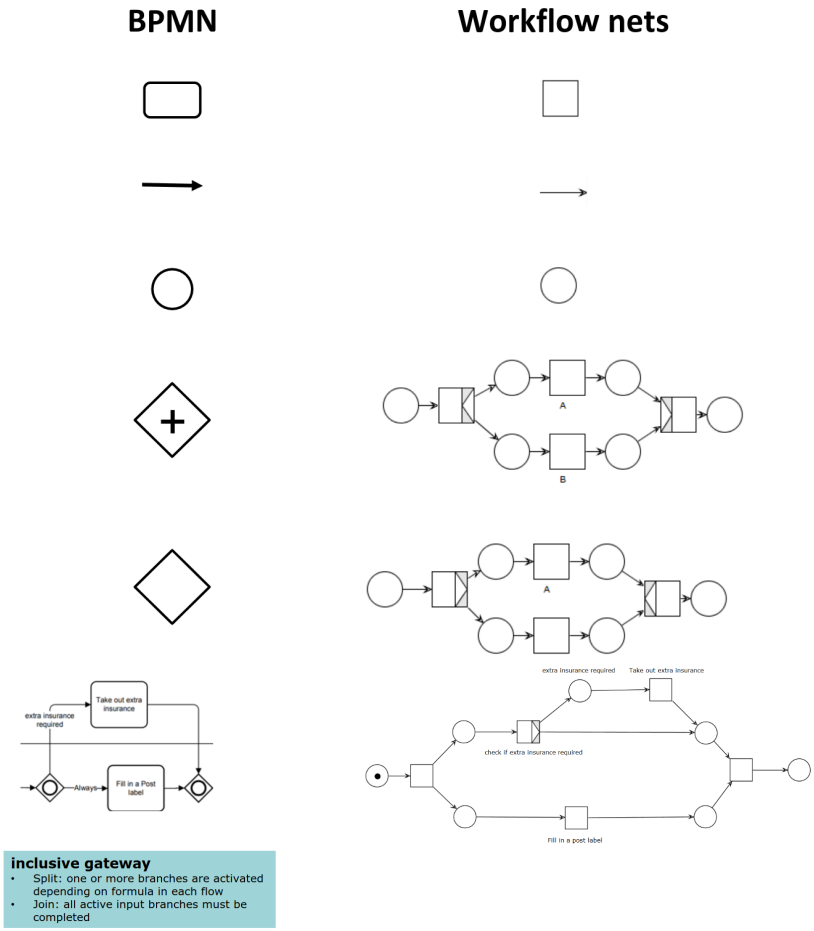


Figure 13.7: From BPMN to Workflow Nets

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?" 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

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

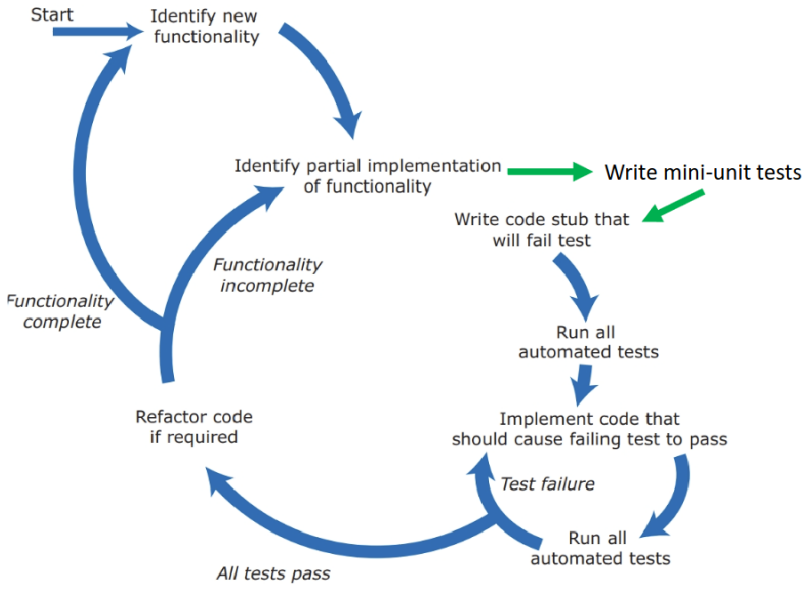


Figure 14.1: Test-driven development schema

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*.

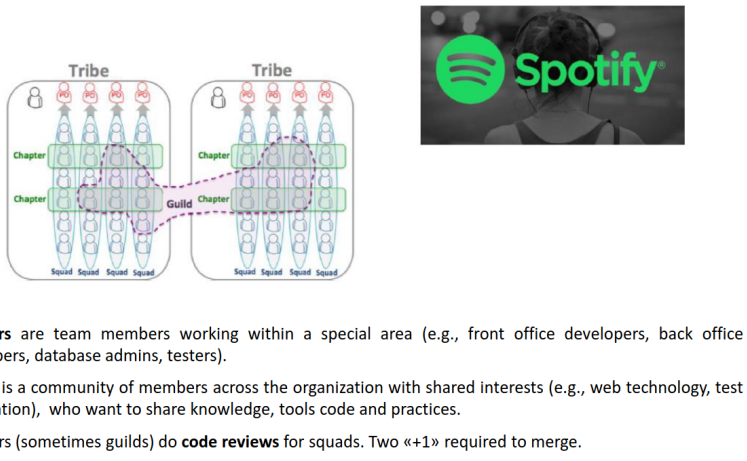


Figure 14.2: Testing in Spotify

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."

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.

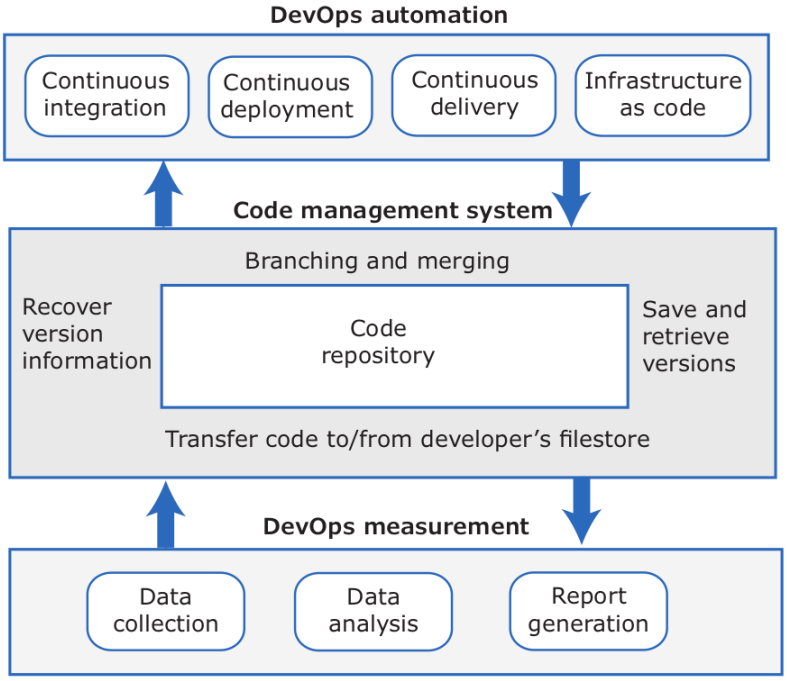


Figure 15.1: Code management according to DevOps

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.

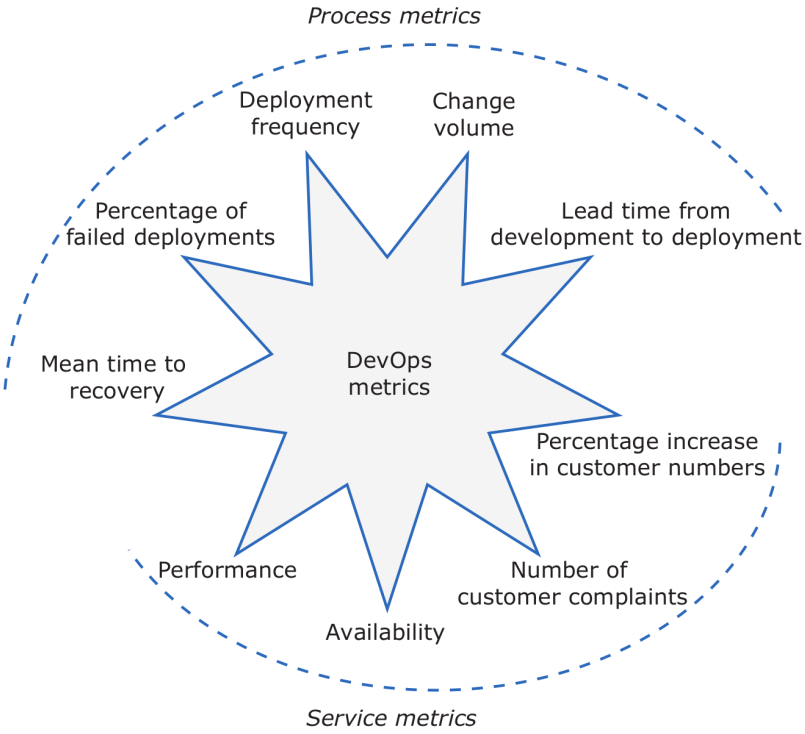


Figure 15.2: DevOps Metrics

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