Advanced Software Engineering

Emiliano Sescu
https://github.com/Faxatos

September - December 2023

Abstract

These notes follow the structure of the course and the lessons presented by Professor Brogi. Along with the slides used in the lectures, these notes will help you prepare for the theoretical part of the exam. The topics included are:

- Software Products
- Agile Software Engineering
- Features, Scenarios, and Stories
- Software Architecture
- Cloud-based Software
- Microservices Architecture
- Security and Privacy
- Business Process Modeling
- Testing
- DevOps and Code Management
- Edge-Cloud Continuum
- Quantum Software Engineering

These notes are available for free! If you find them helpful, please consider leaving a follow on my GitHub account: https://github.com/Faxatos ☺

Contents

Chapter 1

Software products

1.1 Project-based Software Engineering

Project-based software engineering was the first form of software engineering. It starts from the desire of the customers, and at the end of the project you obtain software to be given to the user.

Developers usually try to transition from products originated from different customer to a single product that satisfy all types of customers. The problem with project-based software engineering was that the customer have to generate the requirements in a certain format about things that they generally don't understand. The customer doesn't think in terms of software requirements (requirements specify what is the software we need to implement), so it forces the customer to generate them. The requirements, also, are not static: they change each time we meet customers, because they understand what the software can do for them.

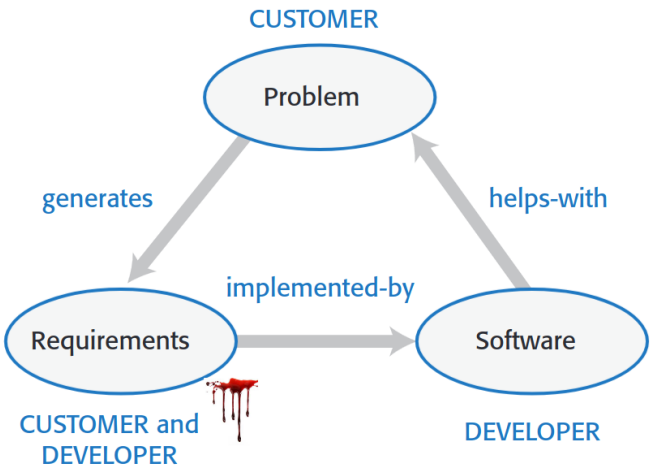


Figure 1.1: Life cycle of project-based software engineering

For most business, users don't need customized software, but they need generalized software.

1.2 Product-based Software Engineering

In **product-based software engineering** the group of developers decides which are the software features, and how the software will change. The customer disappears from the cicle, so the developer choose which project they will build. The developers must identify features that meet the needs of the customers.

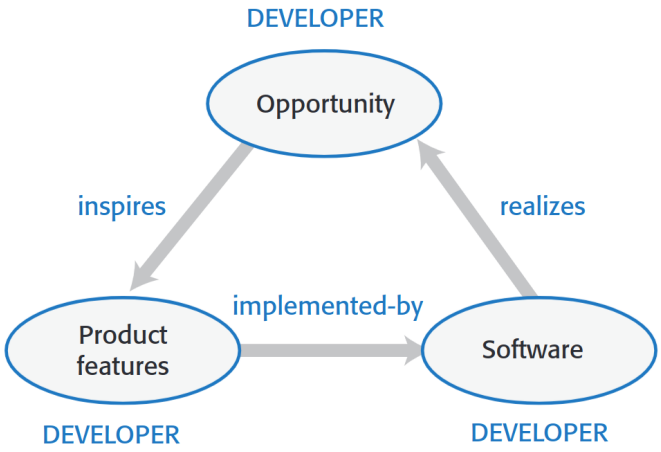


Figure 1.2: Life cycle of product-based software engineering

One of the products based software engineering requirements is that we need to develop software quickly in order to not let someone else steal that market share. That's why time is critical, and that's why agile methods are used.

There are 3 types of **software execution models**:

- **Stand-alone execution**: The user interface, functionalities, and data are all stored on the user's computer. The vendor is responsible for creating and sending updates.
- **Hybrid execution**: Part of the application is on the end-user device (user interface, user data, and all updates from the vendor), and part is on the vendor's server (business logic and user data backups).
- **Software as a service** (SaaS): The user doesn't need to install anything (they just have an interface), meaning the program is hosted on the vendor's server. This makes it easier to manage and update the program without needing to distribute it.

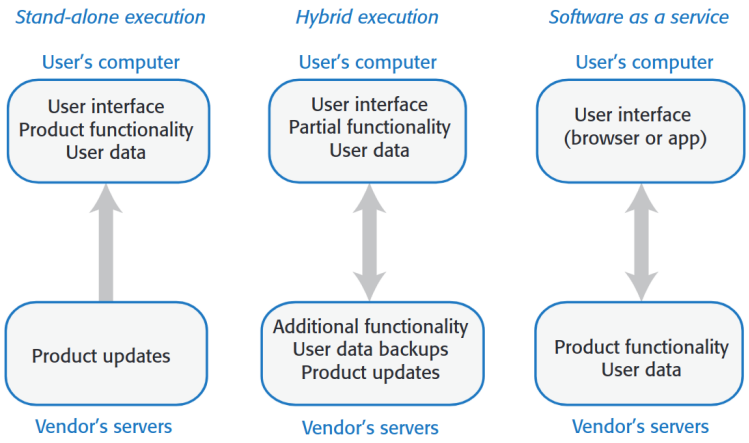


Figure 1.3: Execution models

1.3 Product vision

For any new promising software program, there should be a **vision for the program**: Who are the targeted customers? What is the product to be developed? Why should customers buy the product?
For programmers, it's easier to build cards with the following structure:
FOR (target customer) WHO (statement of the need or opportunity)
THE (product name) is a (product category) THAT (key benefit, compelling reason to buy)
UNLIKE (primary competitive alternative) OUR PRODUCT (statement of primary differentiation)

These are the relevant aspects to building a product vision:

- **Domain experience**: The product developers may work in a particular area (such as marketing and sales) and understand the software support they need. They may be frustrated by the deficiencies in the software they use and see opportunities for an improved system.
- **Product experience**: Users of existing software (such as word processing software) may identify simpler and better ways to provide comparable functionality and propose a new system to implement this. New products can also take advantage of recent technological developments (such as speech interfaces).
- **Customer experience**: Software developers may have extensive discussions with prospective customers to understand the problems they face, as well as constraints, such as interoperability, that limit their flexibility to buy new software. Developers also consider the critical attributes customers need in the software.

- **Prototyping and experimentation**: Developers may have an idea for software but need to develop a better understanding of the concept and what is involved in turning it into a product. Prototyping allows for the exploration of new ideas and feedback from customers (since customers think in terms of functionalities), guiding the path for the next prototype.

1.4 Software product management

The **Product Manager** must ensure that the development team implements features that deliver real value to customers. They must also find a balance between these three forces (we'll see how some Agile techniques help the Product Manager achieve these results):

- **Business needs**: The entire product development process must be managed while considering business needs. The PM needs to ensure that the product satisfies the goals of both the customer and the company.
- **Customer experience**: Regularly gather feedback from customers to understand their experience with the product.
- **Technology constraints**: Consider the various technology constraints the company may have. The Product Manager must take into account the type of hardware and technology the customer already uses.

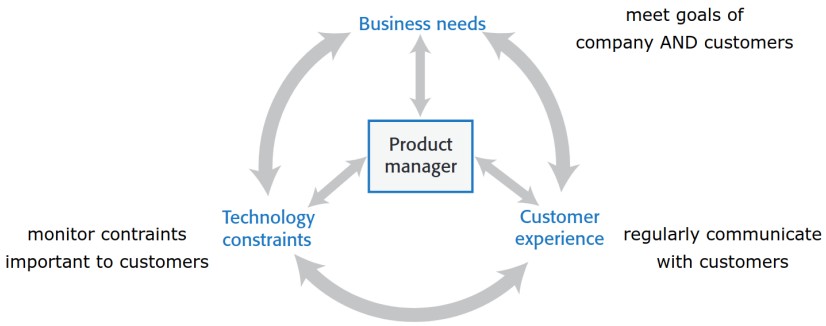


Figure 1.4: Product Manager duties

The Product Manager will have some **technical interactions** such as:

- The **Product roadmap**, which includes setting goals, milestones, success criteria, and alternative paths in case the goals are not reached.
- **User stories and scenarios** to identify product features.
- The **Product backlog**, a to-do list for completing project development.
- **Acceptance testing**, which is used throughout the product lifecycle to verify that releases meet the set goals.

- **Customer testing**, which involves gathering feedback on usability and features.
- **UI design** to ensure simplicity and a natural user interface.

Chapter 2

Agile Software Engineering

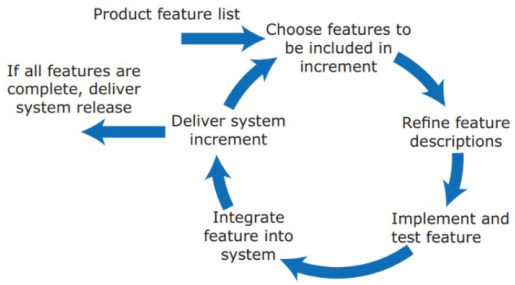
For the past 50 years, software engineering was primarily focused on plan-driven development, which included:

- Detailed project planning (a particularly heavy part)
- Requirement specification (also a very intensive part)
- Analysis and design methods
- Comprehensive system documentation
- Formal quality assurance

At the beginning of the new millennium, a manifesto for agile software development was published, emphasizing:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation (even though documentation is useful for those who will modify the code later, making it detailed is very costly)
- **Customer collaboration** over contract negotiation (to include the customer in the software development process)
- **Responding to change** over following a plan (since no one can predict the future)

The radically new perspective of Agile is to move away from thinking about applications the way they were previously thought of and instead try to align with how users think about the system (as a set of functionalities). One of the Agile principles is **Incremental Development**: it involves selecting a few functionalities and then implementing them.



Activity	Description
Choose features to be included in an increment	Using the list of features in the planned product, select those features that can be implemented in the next product increment.
Refine feature descriptions	Add detail to the feature descriptions so that the team members have a common understanding of each feature and there is sufficient detail to begin implementation.
Implement and test	Implement the feature and develop automated tests for that feature that show that its behavior is consistent with its description. I explain automated testing in Chapter 9.
Integrate feature and test	Integrate the developed feature with the existing system and test it to check that it works in conjunction with other features.
Deliver system increment	Deliver the system increment to the customer or product manager for checking and comments. If enough features have been implemented, release a version of the system for customer use.

Figure 2.1: Continuous Integration and Continuous Deployment cycle

This cycle is then repeated until the final system is completed. These rounds, where we implement features, can sometimes fail, either due to a misinterpretation of required functionalities or because the customer changes their mind. This process, which can be visualized as a pipeline, is called **Continuous Integration and Continuous Delivery/Continuous Deployment** (CICD).

Here are twelve more **Agile principles**:

- Our highest priority is to satisfy the customer through *early and continuous delivery of valuable software*. (as soon as there's a usable piece of software, it should be deployed)
- **Welcome changing requirements**, even late in development. Agile processes harness change for the customer's competitive advantage. (it is unrealistic to have a full and complete set of functionalities from users at the beginning of development. Programmers should adopt a mindset that embraces changing requirements)
- **Deliver working software frequently**, from a couple of weeks to a couple of months, with a preference for the shorter timescale. (CICD pipeline)
- Business people and developers must **work together** daily throughout the project.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- **Working software is the primary measure of progress**. (not the delivery of new documentation or specifications)
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.

- **Simplicity**—the art of maximizing the amount of work not done—is essential.
- The best architectures, requirements, and designs emerge from **self-organizing teams**. (rather than having separate teams in different rooms)
- At regular intervals, the **team reflects** on how to become more effective, then tunes and adjusts its behavior accordingly. (periodically evaluate and adjust practices)

2.1 Extreme Programming

Extreme Programming (XP) is one of the techniques proposed as part of Agile methodology. The key points of XP are shown in the following image:

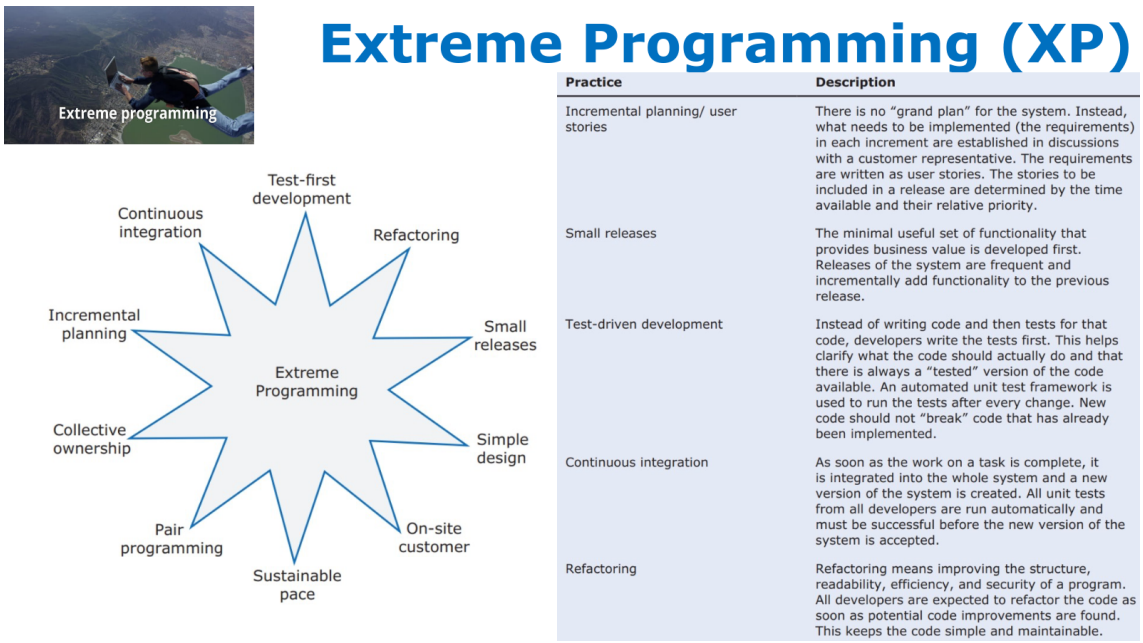


Figure 2.2: Extreme Programming principles

2.2 Scrum

Scrum is a **lightweight framework** that helps individuals, teams, and organizations generate value through adaptive solutions to complex problems. The Scrum framework contains a set of principles and rules to follow in order to achieve a common goal.

The motivation behind Scrum is that software company managers need information that helps them understand the cost of developing a software product, the development timeline, and when the product can be brought to market. Plan-driven development provides this information through long-term development plans that identify deliverables—items the team will deliver and when these will be delivered.

However, plans always change, making long-term plans unreliable, except for short-term planning.

Scrum is based on **Empiricism**, which asserts that knowledge comes from experience and that decisions should be made based on observation. It is also rooted in **Lean thinking**, which focuses on reducing waste and emphasizing essentials. Scrum employs an iterative, incremental approach to optimize predictability and control risk. The pillars of Scrum are:

- **Transparency**: The progress and work should be visible to those performing the work and those receiving the work.
- **Inspection**: Scrum artifacts and progress toward agreed goals must be frequently and diligently inspected to detect potentially undesirable variances or problems.
- **Adaptation**: If any aspects of a process deviate (e.g., new requirements) beyond acceptable limits, or if the resulting product is unacceptable, the process being applied or the materials being produced must be adjusted. These adjustments should be made as soon as possible to minimize further deviation.

Successful use of Scrum depends on people becoming more proficient in living its five core values: **Commitment** (respect deadlines), **Focus**, **Openness** (being open to new solutions), **Respect** (for teammates and others), and **Courage** (to propose new solutions that better align the product with its vision).

2.2.1 Scrum Team

A **self-organizing team** coordinates its work by discussing tasks and reaching a consensus on who should do what. This minimizes the involvement of engineers in external interactions with management and customers. The team makes its own decisions on schedules and deliverables.

External interactions refer to interactions team members have with individuals outside the team. In Scrum, the idea is that developers should focus on development, while only the Scrum Master and Product Owner should handle external interactions. The goal is for the team to focus on software development without external interference or distractions.

Product Owner

The Product Owner is responsible for ensuring that the development team stays focused on building the product rather than getting sidetracked by technically interesting but less relevant work. In product development, the Product Manager typically assumes the Product Owner role.

Product-focused external interactions are managed by the Product Owner.

Scrum Master

The Scrum Master is an expert in Scrum whose job is to guide the team in effectively using the Scrum method. Scrum developers emphasize that the Scrum Master is not a traditional project manager but rather a coach for the team. They have authority within the team regarding how Scrum is used. In many companies, the Scrum Master also takes on some project management responsibilities.

In all but the smallest product development companies, development teams must report progress to company management. A self-organizing team needs to appoint someone to handle these responsibilities. Because maintaining continuity in communication with people outside the group is crucial, rotating these activities among team members is not a viable approach. Although the Scrum developers did not envision the Scrum Master also assuming project management responsibilities, in many companies, *the Scrum Master takes on project management tasks* because they have the best understanding of the work in progress and are in the best position to provide accurate information and project plans.

Team-focused external interactions are managed by the Scrum Master.

Developers

Self-organizing teams make their own decisions, working by discussing issues and reaching a consensus. The ideal **Scrum team size** is between 5 and 8 people. Teams need to tackle diverse tasks, often requiring members with different skills, such as networking, user experience, database design, and more. They also typically include individuals with varying levels of experience. A team of 5 to 8 people is large enough to be diverse yet small enough to communicate informally and effectively, allowing for agreement on team priorities.

The advantage of a self-organizing team is that it can become cohesive and adapt to change. Because the team, rather than individuals, takes responsibility for the work, it can manage transitions when team members leave or join. Effective communication within the team means that members inevitably learn about each other's areas of expertise.

The developers of Scrum assumed that teams would be co-located, working in the same space to communicate informally. Daily scrums help team members stay updated on what's been done and what others are working on. However, two assumptions behind the daily scrum are not always correct:

- Scrum assumes that the team is made up of full-time workers sharing a workspace. In reality, team members may work part-time or in different locations. For student project teams, members may take different classes at different times.

- Scrum assumes that all team members can attend a morning meeting to coordi-

nate the day's work. However, some members may work flexible hours (e.g., due to childcare responsibilities) or may work on several projects simultaneously.

2.2.2 Artifacts

Product Backlog

The product backlog is a list of tasks that need to be completed to develop the product. This to-do list of items is reviewed and updated before each sprint.

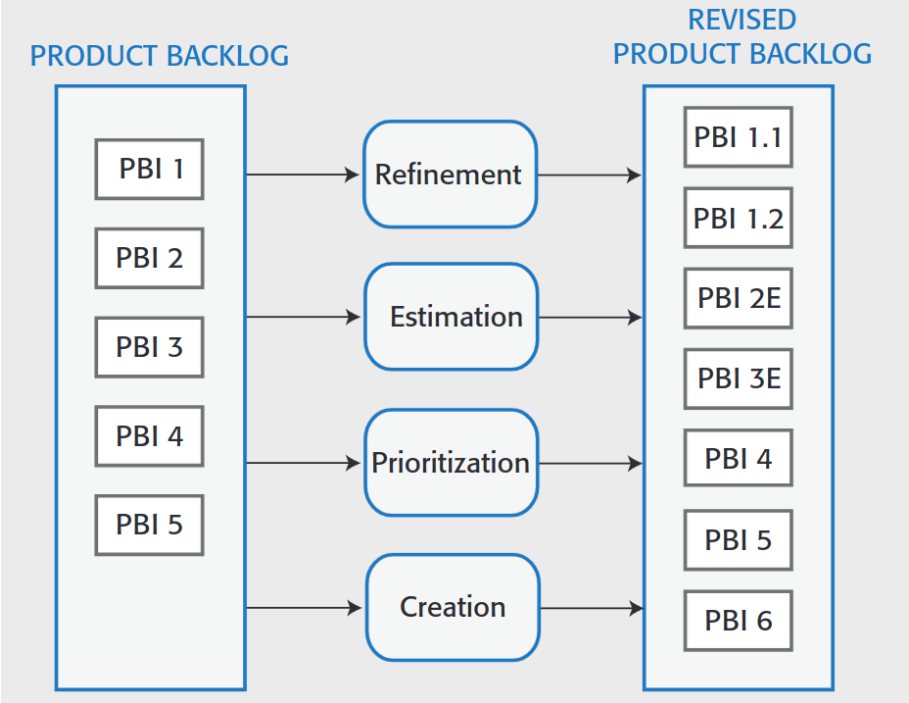


Figure 2.3: Product backlog during the action process

As illustrated in the image above, items that are to be implemented are first selected and prioritized. According to their priorities, they are sorted and will be implemented in the next sprint. Below are some examples of product backlog items:

- *As a teacher, I want to be able to configure the group of tools that are available to individual classes.* (feature)
- *As a parent, I want to be able to view my children's work and the assessments made by their teachers.* (feature)
- *As a teacher of young children, I want a pictorial interface for children with limited reading ability.* (user request)
- *Implement encryption for all personal user data.* (engineering improvement)

Product backlog items can exist in different states:

- **Ready for consideration**: High-level ideas and feature descriptions are under consideration for inclusion in the product. These are tentative and may change or not be included in the final product.
- **Ready for refinement**: The team agrees that this is an important item to be implemented in the current development cycle. There is a reasonably clear definition of what is required, though further refinement is needed.
- **Ready for implementation**: The product backlog item has enough detail for the team to estimate the effort required and begin implementation. Dependencies on other items are also identified.

Sprint Backlog

The sprint backlog is a focused list of tasks selected from the product backlog that the team commits to completing during the sprint. It is created during sprint planning, where items are chosen based on the sprint goal and broken down into smaller, actionable tasks.

Unlike the product backlog, which contains all potential features and improvements, the sprint backlog is **limited to what can be achieved within the sprint's time frame**. As the sprint progresses, the team updates the sprint backlog to track completed tasks and any changes that arise.

The sprint backlog helps the team stay aligned with their sprint goal and manage progress efficiently. Tools like burn-down charts are often used to visualize the remaining work and time in the sprint.

2.2.3 Scrum Events

In Scrum, software is developed in fixed-length periods called **sprints**, typically lasting 2-4 weeks. During a sprint, the team holds daily meetings (Scrums) to review progress and update the list of incomplete work items. A sprint aims to produce a '*shippable product increment*,' meaning that the developed software should be complete and ready to deploy. Sprints are **timeboxed**, meaning development stops at the end of the sprint, regardless of whether all work has been completed. The team works on items from the product backlog during a sprint.

There are three main types of **sprint activities**:

- **Sprint Planning**: The team selects the work items to be completed during the sprint, and if necessary, refines them to create a sprint backlog. This planning session should not last longer than a day at the start of the sprint.
- **Sprint Execution**: The team works to implement the sprint backlog items. If it becomes impossible to complete all items, the sprint is not extended; instead, unfinished items are returned to the product backlog for future sprints.

- **Sprint Review**: The team and possibly external stakeholders review the completed work. They reflect on what went well, what went wrong, and how to improve their work processes.

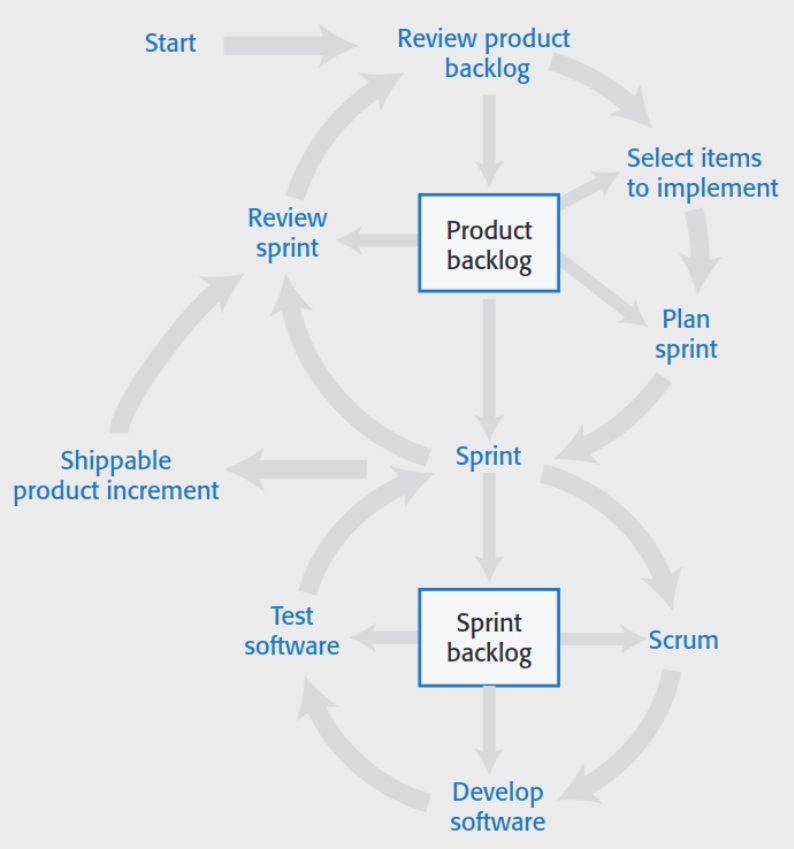


Figure 2.4: Sprint activities within the Scrum cycle

Sprint Planning

Sprint planning involves establishing an agreed sprint goal, which may focus on software functionality, support, performance, or reliability. To define the goal, the team selects which items from the product backlog should be implemented. The outcome of this process is a **sprint backlog**, a more detailed version of the product backlog, listing the tasks to be completed during the sprint.

Throughout the sprint, the team holds **daily Scrum meetings** to coordinate their work. These meetings, often called *scrums*, are short and typically held at the beginning of the day. During the meeting, each team member shares their progress, any problems encountered, and their plans for the day. This ensures that everyone on the team knows what is happening and allows for quick re-planning if needed. Scrum meetings should remain short and focused. To avoid prolonged discussions, they are sometimes held as 'stand-up' meetings, where no chairs are provided.

During a Scrum, the sprint backlog is reviewed, completed items are removed, and new items may be added as new information arises. The team then decides which team member will work on specific sprint backlog items for the day.

Sprint Execution

Scrum does not prescribe specific technical agile activities to be used during a sprint, but two practices are commonly recommended by agile software engineers:

- **Test Automation**: As much testing as possible should be automated. A suite of executable tests should be developed, which can be run at any time to ensure software quality.
- **Continuous Integration**: Whenever changes are made to software components, they should be immediately integrated with the other components to form a complete system. This system should then be tested to check for any unanticipated issues from component interactions.

Sprint Review

At the end of each sprint, the team conducts a sprint review meeting, involving all team members. The meeting reviews whether the sprint goal was met, addresses any new issues that arose during the sprint, and provides a chance for the team to reflect on how to improve their work process.

The product owner has the authority to decide whether the sprint goal has been achieved. They confirm whether the implementation of the selected product backlog items is complete. Additionally, the sprint review should include a process review where the team reflects on how they've used Scrum and discusses how to improve productivity in the next sprint.

2.2.4 Execution

Depending on the state of the item, there are different possible actions:

- **Refinement**: Existing PBIs are analyzed and refined to create more detailed PBIs. This may lead to the creation of new product backlog items.
- **Estimation**: The team estimates the amount of work required to implement a PBI and adds this assessment to each analyzed PBI.
- **Creation**: New items are added to the backlog. These may include new features suggested by the product manager, required feature changes, engineering improvements, or process activities such as assessing development tools that might be used.

- **Prioritization**: New items are prioritized within the backlog. These may also include new features suggested by the product manager, required feature changes, engineering improvements, or process activities such as assessing development tools that might be used.

We can see the effects of these actions in Figure 2.3. There are metrics that we can apply to PBIs in order to choose the next state or action. The following factors are usually considered:

- **Effort required**: This may be expressed in person-hours or person-days, i.e., the number of hours or days it would take one person to implement that PBI. This is not the same as calendar time, as several people may work on an item, which can shorten the calendar time required.
- **Story points**: Story points are an arbitrary estimate of the effort involved in implementing a PBI, taking into account the size of the task, its complexity, the technology that may be required, and the 'unknown' characteristics of the work. They were originally derived by comparing user stories but can be used for estimating any kind of PBI. Story points are estimated relatively; the team agrees on the story points for a baseline task, and other tasks are estimated by comparison with this, e.g., more/less complex, larger/smaller, etc.

Chapter 3

Features, Scenarios, and Stories

These are the factors that drive the design of software products:

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

We'll understand how to reach the specification of the software product we want to create. We saw that discussing requirements with the client is usually hard, mostly because of the language barrier and the fact that most clients don't even know which functionalities they want.

We observed that product-based software engineering requires less documentation than project-based software engineering. In Agile, requirements are not set by customers and change often. That's why developers need to identify product features: they must understand potential users and how to attract them (through interviews, surveys, informal consultations, and so on).

User representations (**personas**) and natural language descriptions (**scenarios and stories**) help identify product **features**.

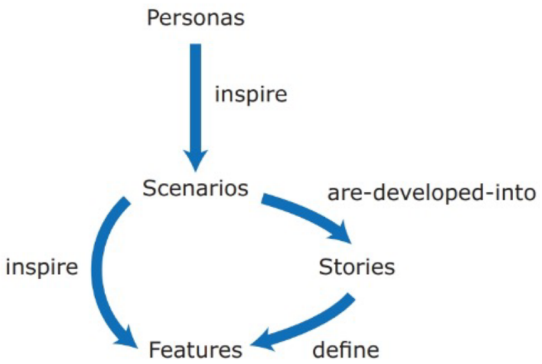


Figure 3.1: Features generation process

3.1 Personas

Personas are the target users for our product. Programmers need to understand potential users to design features that are useful for them. The background, skills, and experience of potential users are important because these factors will influence the user experience and the user interface. Generally, only a few (1-2, max 5) personas are required to identify key product features. Personas allow developers to "step into the users' shoes."

Let's examine the key features of personas:

- **Personalization**: You should give them a name and describe their personal circumstances. It is sometimes helpful to use an appropriate stock photograph to represent the person in the persona. Some studies suggest that this helps project teams use personas more effectively.
- **Job-related**: If your product is targeted at businesses, you should describe their job and (if necessary) what the job involves. For some jobs, such as teaching, where readers are likely familiar with the role, this may not be necessary.
- **Relevance**: If possible, you should explain why they might be interested in using the product and what they might want to do with it.
- **Education**: You should describe their educational background and their level of technical skills and experience. This is important, especially for interface design.

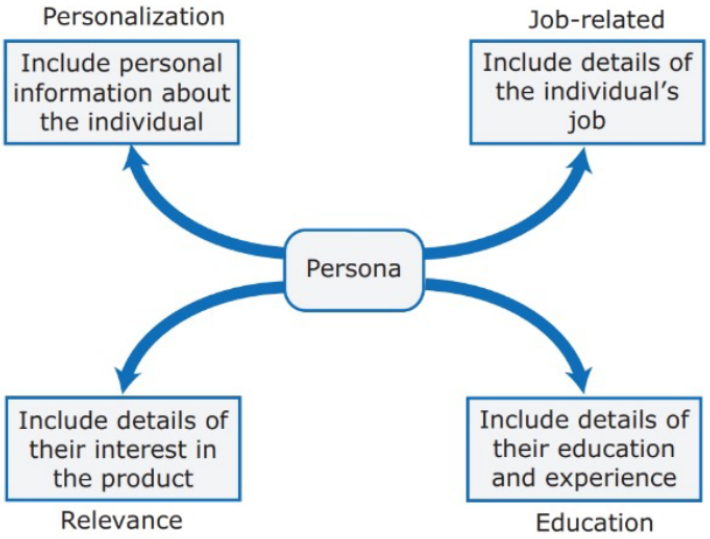


Figure 3.2: Personas' key features

Some developers also use an appropriate stock photograph to represent the person in the persona. Studies have shown that we have biases depending on the photo, so not everyone accepts the use of pictures. When studying users is not possible (e.g., for some new products), we can develop proto-personas: these are imagined personas.

3.2 Scenarios

Once the personas are created, the next step is to work with scenarios. To discover product features, we can define **scenarios** of user interactions with the product. A scenario is a narrative describing a situation in which a user is using our product's features to accomplish a specific goal.

Let's examine the key features of scenarios:

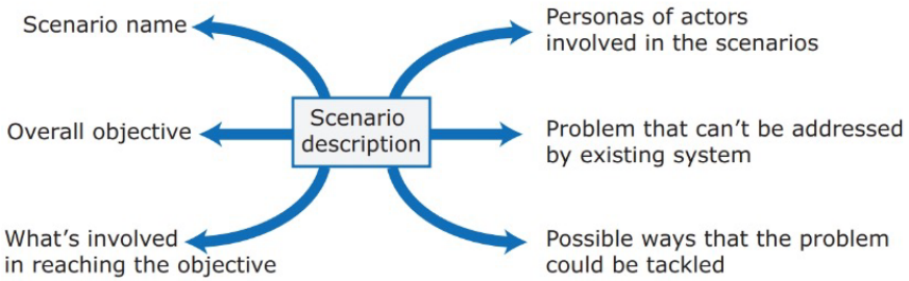


Figure 3.3: Scenario's key features

Narrative, high-level scenarios facilitate communication and stimulate design creativity. Scenarios are not specifications, though; they lack details and may be incomplete. Usually, several scenarios (e.g., 3-4) are required for each persona, covering the main responsibilities of the persona. The scenarios are written from the *user's perspective*. Each team member should individually create some scenarios, then discuss them with the rest of the team and, if possible, with users.

3.3 User Stories

Scenarios are high-level stories of product use, while user stories are finer-grained narratives. They follow a structure, which is as follows:

As a <*role*> I want to <*do something*> so that <*reason/values*>.

User stories are not part of the Agile manifesto, but they help to adhere to Agile principles:
- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Working software is the primary measure of progress.
- Simplicity, the art of maximizing the amount of work not done, is essential.

User stories also help visualize the progress made in our program. We can write the short stories on sticky notes to stick them on a to-do board. We need to remember that *stories that don't create business value for the customer are work that isn't going to count as progress. Additionally, creating stories that depend on one another might create deadlocks.*

The Scrum product backlog is often a set of user stories. Long stories (*epics*) must be broken into simpler stories. Stories are associated with **priorities** (and possibly also with an estimate of effort needed to implement the story) and sorted according to priority (*requirements triage*). It is possible to express all the functionalities described in a scenario as user stories, but scenarios read more naturally, making it easier to understand stories and providing more context (we can sell stories to stakeholders).

The final goal of the user stories is to identify the features that define our product. These features should, in principle, have the following properties:

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

The knowledge required for feature design is:

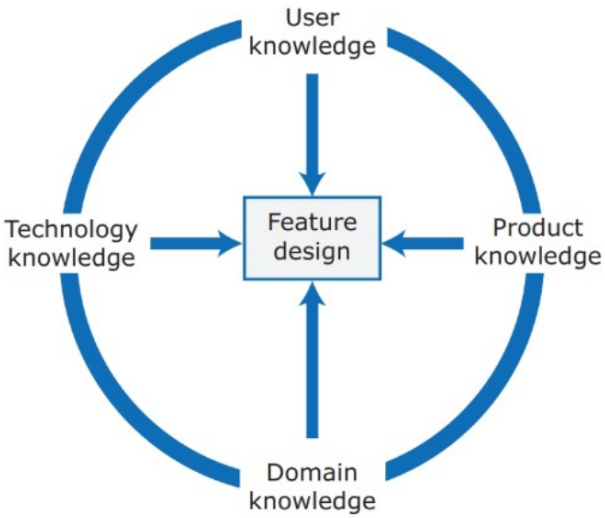


Figure 3.4: Feature design knowledge required

- **User Knowledge**: You can use user scenarios and user stories to inform the team of what users want and how they might use the software features.
- **Product Knowledge**: You may have experience with 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.
- **Domain Knowledge**: 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 accomplish their goals.
- **Technology Knowledge**: 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.

It's important to find a balance between opposing factors in feature design.

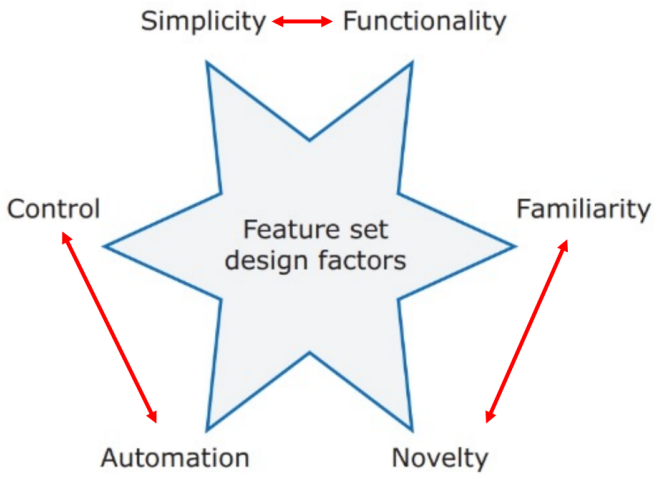


Figure 3.5: Factors in feature design

The number of product features grows as new potential users are envisaged (this is also known as **Feature Creep**). This growth is dictated by marketing executives who want to meet all users' demands. Marketing also pressures teams to include competitors' features and has the desire to support both experienced and inexperienced users. To avoid feature creep, making feature questions usually allows understanding which features are pointless.

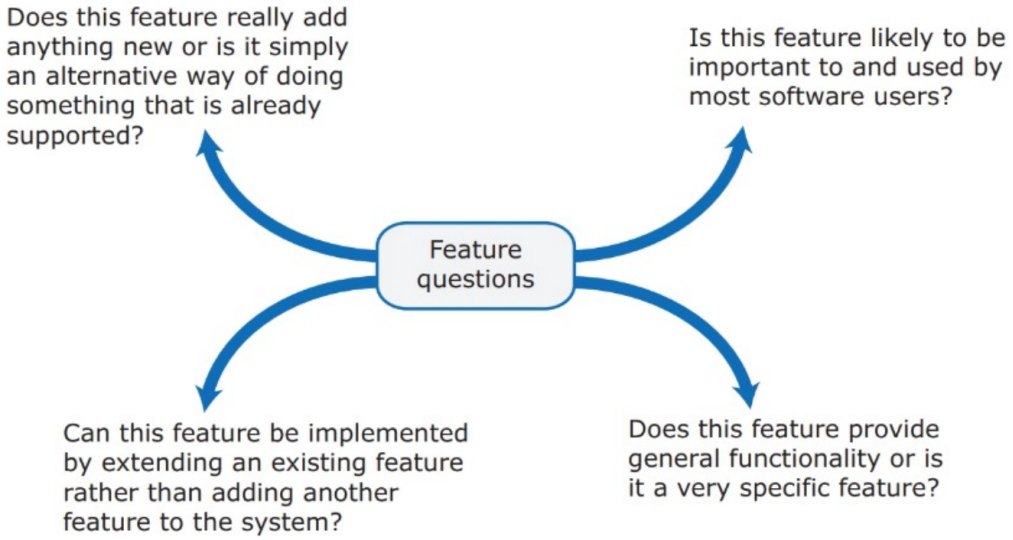


Figure 3.6: Feature questions to avoid feature creep

The product development team must meet to discuss scenarios and stories to extract the list of product feature descriptions. This consists of detecting the input and activation for the action required in the feature and then writing the expected output for the action.

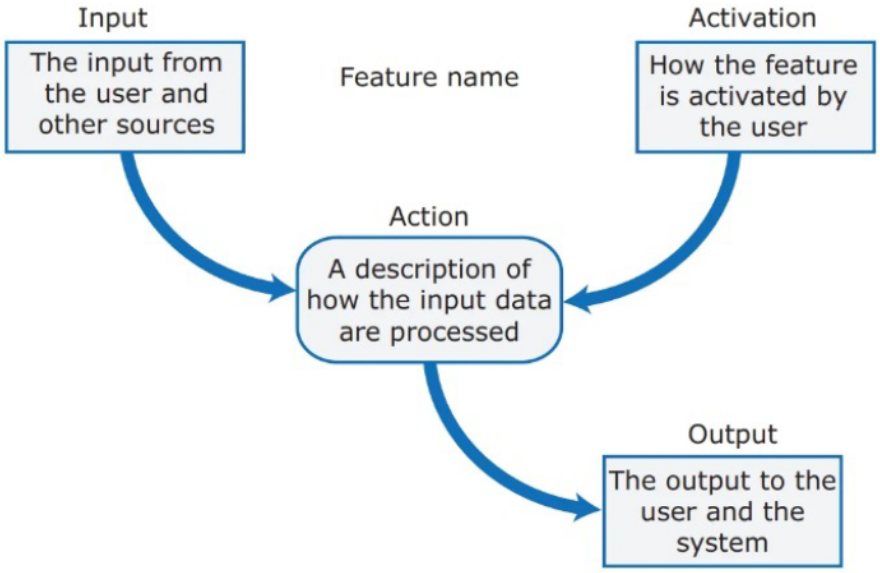


Figure 3.7: Feature derivation

Chapter 4

Software Architecture

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. The software architecture affects *performance, usability, security, reliability, maintainability*, and so on.

Architecture is made up of **components**, where components implement a coherent set of **features** (services that may be used by other components). Here's an example of component usage:

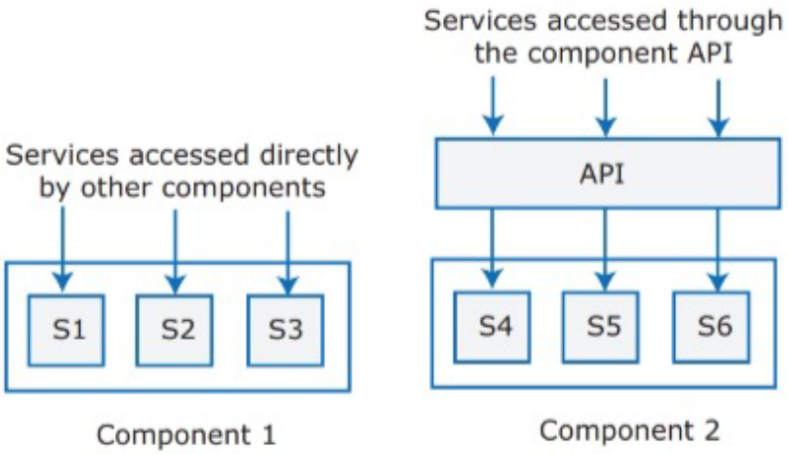


Figure 4.1: API to call component services

In all cases where the software is large, developers will need to develop **APIs** (Application Programming Interfaces) to allow communication between components that might even be implemented with different technologies.

When we design our software, we need to take into account different aspects. These are some issues that are usually addressed:

- **Non-functional product characteristics**: Non-functional product characteristics such as security and performance affect all users. If you get these

wrong, your product is unlikely to be a commercial success. All these characteristics have a cost (e.g., availability, security, and so on).

- **Product lifetime**: If you anticipate a long product lifetime, you need to create regular product revisions. You therefore need an architecture that can evolve, so that it can be adapted to accommodate new features and technologies.
- **Software reuse**: You can save a lot of time and effort if you reuse large components from other products or open-source software. However, this constrains your architectural choices because you must fit your design around the software that is being reused.
- **Number of users**: If you are developing consumer software delivered over the internet, the number of users can change very quickly. This can lead to serious performance degradation unless you design your architecture so that your system can be quickly scaled up and down.
- **Software compatibility**: For some products, it is important to maintain compatibility with other software so that users can adopt your product and use data prepared using a different system. This may limit architectural choices, such as the database software that you can use.

4.1 Non-functional Quality Attributes

When we talk about **non-functional product characteristics**, we are defining a set of specifications that describe the system's operational capabilities and constraints and attempt to improve its functionality. This is a list of the most popular ones:

- **Responsiveness**: Does the system return results to users in a reasonable time?
- **Reliability**: Do the system features behave as expected by both developers and users?
- **Availability**: Can the system deliver its services when requested by users?
- **Security**: Does the system protect itself and users' data from unauthorized attacks and intrusions?
- **Usability**: Can system users access the features they need and use them quickly and without errors?
- **Maintainability**: Can the system be readily updated and have new features added without undue costs?
- **Resilience**: Can the system continue to deliver user services in the event of partial feature failure or an external attack?

For each non-functional quality attribute we implement, we also need to test them. This increases the product cost. Usually, optimizing one non-functional attribute affects others (e.g., to achieve more security with longer keys, we may need to sacrifice some performance or usability).

The prototypes don't need to respect these characteristics because we want to quickly build a prototype.

Good practice for maintainability is to decompose the system into small, self-contained parts and avoid shared data structures. Not having centralized data structures is a good practice because if one component needs to change the database organization, this does not affect the other components. The system can also continue to provide partial service in the event of a database failure.

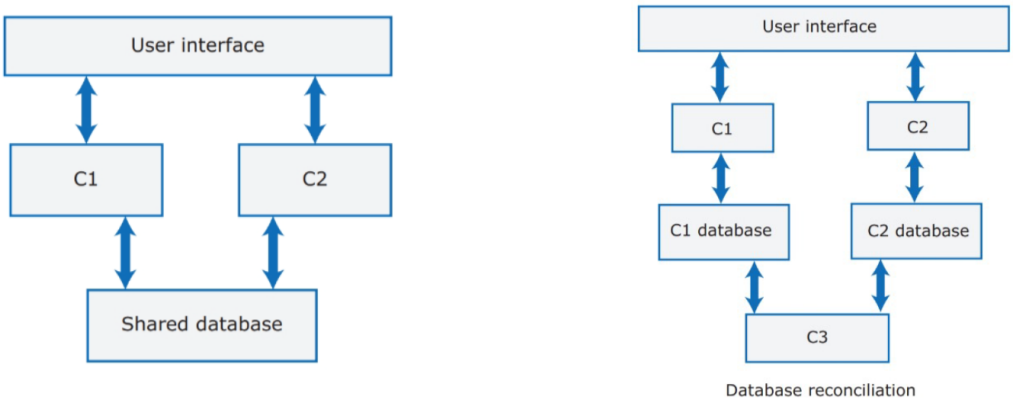


Figure 4.2: Database partitioning

The cost of this practice is *eventual consistency*: if component C1 makes a change (see Figure 4.2), sooner or later that change must be reported to the C3 database. This is an additional cost. This also applies in the case of availability: if you want more availability, you can deploy multiple databases, but this requires synchronizing multiple databases. When something goes wrong in a distributed system, it's hard to roll back transactions. That's why we usually give up on consistency in order to achieve availability.

4.2 System Decomposition

As we already said, a *service* is a coherent unit of functionality, a *component* is a software unit offering one or more services, and a *module* is a set of components.

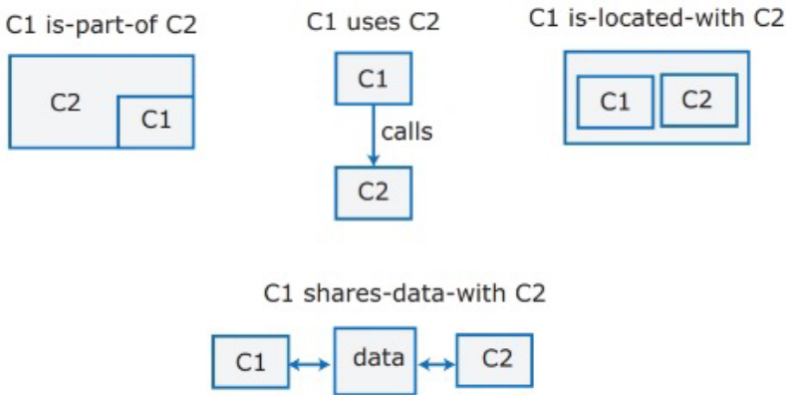


Figure 4.3: Example of component relationships

As the number of components increases, the number of relationships tends to increase at a faster rate. Therefore, we should **decompose** as much as needed. The reason we want to decompose is to have simpler parts that can be maintained separately. In order to decompose a system, one good idea is *separation of concerns*, as we will see with microservices that implement one business feature. We also want to have *stable interfaces*, unless this is really necessary. The dogma of *implement once* contrasts with the idea of independent teams that work with independent systems, but there aren't fixed rules; we always need to find trade-offs.

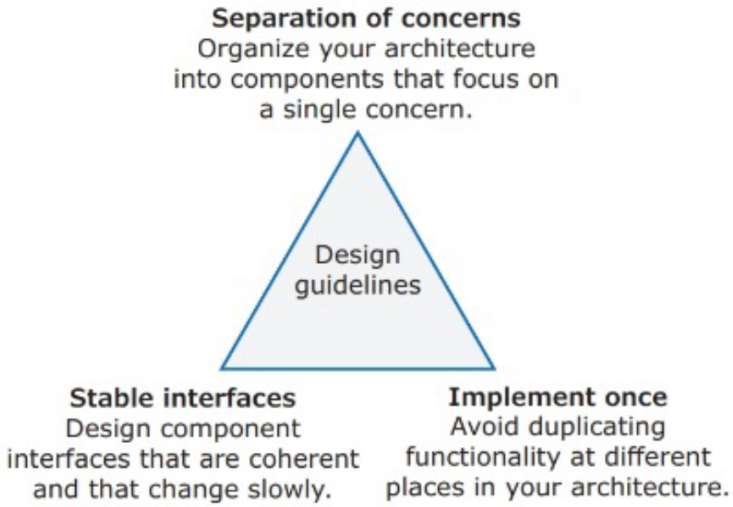


Figure 4.4: Design guidelines to control complexity

Another aspect of system decomposition is having a **layered architecture**. Each layer is an area of concern and is considered separately from other layers. Within each layer, the components are independent and do not overlap in functionality. The architectural model is a high-level model that doesn't include implementation information. Software architects usually customize *generic layers* by adding or removing layers until the architecture achieves a good form for its goals.

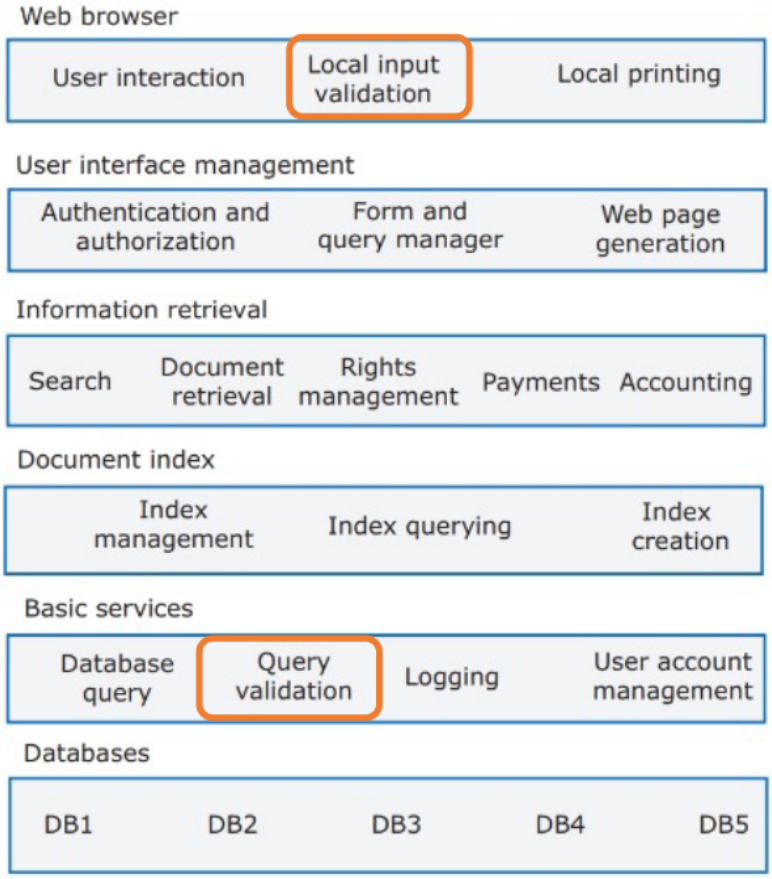


Figure 4.5: Example of a layered architecture

There are some systemic concerns that affect the whole system, so every layer must take them into account. **Cross-cutting concerns** (such as *security, performance, reliability, validation*, and so on) add interactions between the layers. In the example shown in Figure 4.5, we can see that validation is considered in multiple layers.

On one hand, when we design architecture, we don't want to discuss implementation (because we don't want to limit the design with implementation details), but at the same time we need to consider possible implementations to understand how to design the architecture (e.g., the choice of using a relational database affects components at higher layers).

4.3 Distribution Architecture

The **distribution architecture** defines servers and the allocation of components to servers, specifying where the software will be deployed and run during production.

The most common type is the **client-server architecture**, which is suited for applications in which clients access a shared database and perform business logic operations on that data. Clients interact with *load balancers* in the system, which are deployed on a cluster of servers.

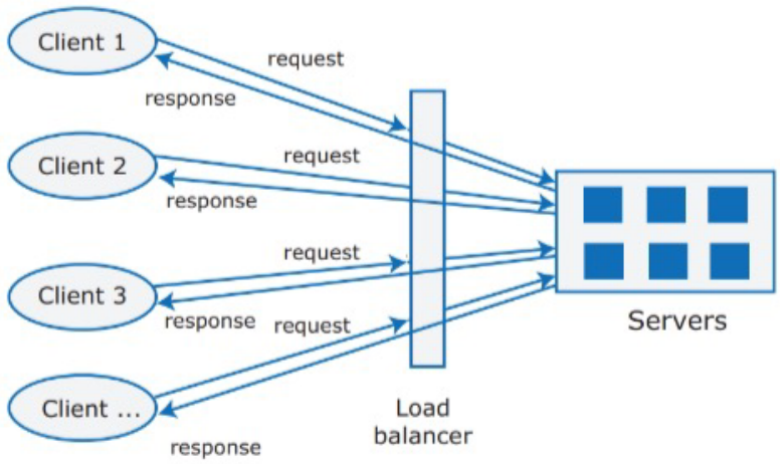


Figure 4.6: Example of a client-server architecture

Model-View-Controller is a widely used *pattern*. This pattern helps to structure the code by dividing it into the *Model part*, the *View part*, and the *Controller part*. In many cases, this also helps to understand how to implement the communications between the parts, which affects the simplicity and efficiency of the architecture (in many cases, the communication is implemented with HTTP + JSON).

The pattern shown in Figure 4.7 is able to separate the logic of data presentation from business logic. This pattern is positioned at the logical or business level and is presented in a multi-tier architecture.

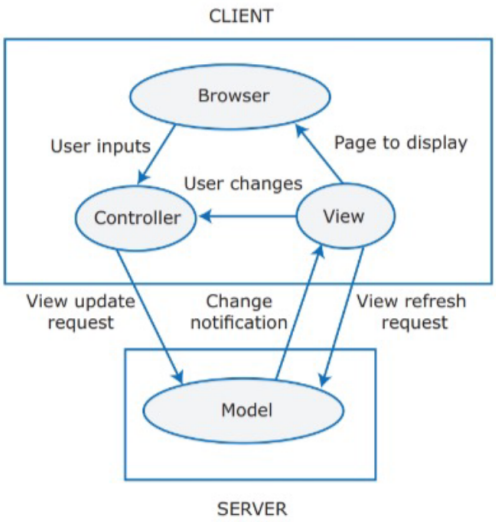


Figure 4.7: Model-View-Controller

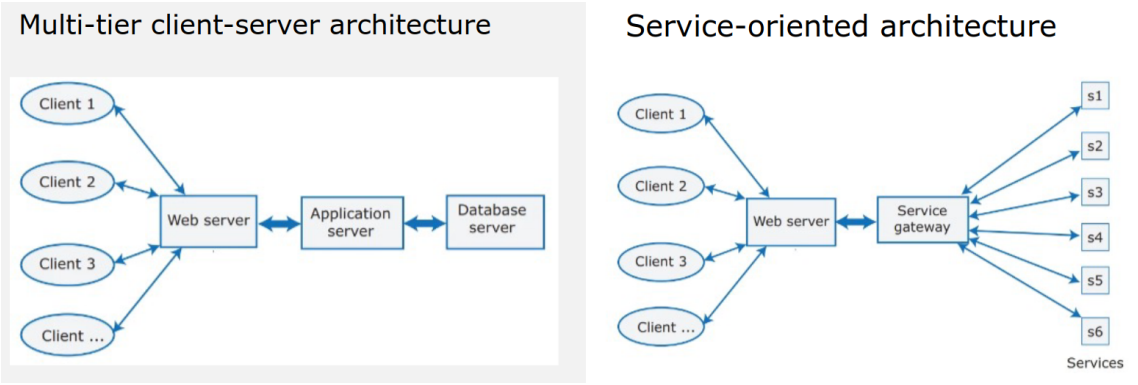


Figure 4.8: Common types of client-server architecture

Client-server architectures can be implemented in different ways (as shown in Figure 4.8). The *Service-oriented architecture* is slightly different: it has a Service gateway which splits the service requests to the appropriate service.
The choice of distribution architecture depends on several factors:

- **Data type and data updates**: If you are mostly using structured data that may be updated by different system features, it is usually best to have a single shared database that provides locking and transaction management. If data is distributed across services, you need a way to keep it consistent, and this adds overhead to your system.
- **The system execution platform**: If you plan to run your system on the cloud with users accessing it over the Internet, it is usually best to implement it as a service-oriented architecture because scaling the system is simpler. If your product is a business system that runs on local servers, a multi-tier architecture may be more appropriate.
- **Change frequency**: If you anticipate that system components will be regularly changed or replaced, then isolating these components (inside containers) as separate services simplifies those changes.

4.4 Technology Choices

There are several "relevant" technology choices. These are considered relevant because changes in product design will reflect on these technologies, resulting in increased complexity and costs required to modify them during development.

This is a list of some relevant technologies:

- **Database**: You should use either a *relational SQL database* (where data is organized into structured tables, particularly suitable when transaction management is needed and data structures are predictable and simple) or an *unstructured NoSQL database* (where data has a more flexible, user-defined organization. This option is more efficient for data analysis, and data can be organized hierarchically. It also supports efficient concurrent processing of 'big data').

- **Delivery Platform**: This refers to the platform where the product will ultimately run. Delivery can be web-based or mobile, but each platform presents its own challenges (e.g., mobile device issues include intermittent connectivity, processor power, power management, reduced screen size, and on-screen keyboard). It's advisable to decide on the platform early in order to begin the development phase as soon as possible.

- **Server**: Will the product run in a public cloud, or will it use in-house servers? Most consumer products utilize a set of microservices, which are containerized and deployed in the cloud (with their Service-Oriented Architecture, SOA). In contrast, business products are more concerned with cloud security issues (e.g., where and how client data will be stored). Once it's decided that the product will be launched in the cloud, the next decision is to choose a provider (note that it can be challenging to migrate a product across cloud providers due to vendor lock-in).

- **Open Source**: Are there suitable open-source components that could be incorporated into the product? Are these open-source components secure enough?

- **Development Tools**: Development technologies (e.g., mobile development toolkits, web application frameworks) influence the architecture of your product. The development technology that developers are familiar with may indirectly affect the architecture of the product.

4.5 Enterprise Application Integration

Enterprise applications are large software systems typically designed to operate in a corporate environment, such as business or government. These applications are extensive systems where users can typically only see the front-end, while a lot of backend functionality (made up of multiple services) supports it. The various heterogeneous services can be classified as *sources* (services that only make requests), *sinks* (services that only respond to requests), or a mix of both categories. These services communicate using various *heterogeneous data types* (with different representations, even for analogous data), and this information can be shared with

different organizations over the *network*. This complexity makes enterprise applications distributed multi-service applications whose services must work together and be suitably integrated.

The architectural question is how to integrate all of this in a coherent, extensible, and maintainable way. There is a need to manage these complexities, and one approach is to use **patterns**. A pattern is a high-level abstraction of accepted, reusable solutions for recurring problems. Patterns are defined in terms of problem statements (what problem the pattern solves), context (there may be different patterns for the same problem), forces (why you should consider using a pattern for the problem), and solutions (how to apply the pattern to solve the problem). The idea of patterns is to avoid reinventing the wheel and instead use established patterns. **Enterprise Application Integration** (EAI) is a reusable abstraction of proven solutions to well-known problems that arise while integrating the software components and services forming enterprise applications.

4.5.1 Patterns

These patterns were studied more than 20 years ago. Among all the patterns, the most commonly used are **messaging patterns**, as shown in Figure 4.9.

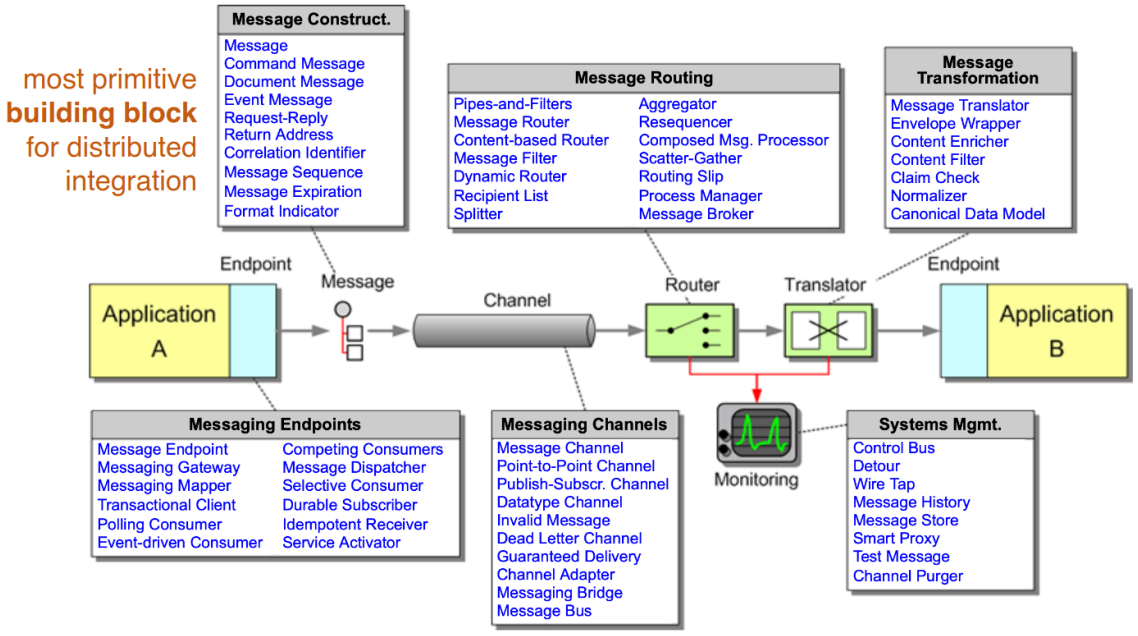


Figure 4.9: Messaging Patterns[1]

A message is a discrete piece of data sent from one service to another. Typically, messages are structured into a *header* (metadata) and a *body* (payload). Concrete examples of messages include *document messages* (pure data), *event messages* (notifications of events), and *command messages* (commands sent to perform actions).

[1] https://www.enterpriseintegrationpatterns.com/patterns/messaging/

Figure 4.10: Example of concrete messages

The exchange of messages means that the services are not tightly connected but can communicate through messages. Message-based communication enables *loose coupling*, and it is realized through **channels**: abstractions for components sending messages from a source to a destination (implemented in various ways, such as RPC, HTTP, TCP, and so on). Channels are *one-way*, allowing communications to be natively asynchronous. Synchronous (request/response) communications use two channels. Application services are typically independent of the messaging systems; thus, **adapters** are used to enable application-specific data to be sent to channels. **Message endpoints** allow application services to send and receive messages to and from channels. There are different types of channels: the one shown in Figure 4.11 is a *point-to-point* channel (which ensures that only one receiver will receive a particular message), whereas another type is the *publish-subscribe* channel (where the publisher delivers a copy of incoming messages to each subscriber).

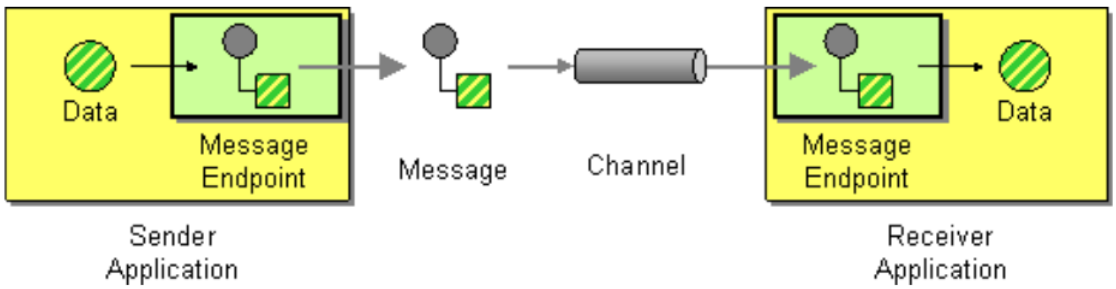


Figure 4.11: Simple integration of message-based communication

This implementation alone is not sufficient: what happens if the receiving service expects a different data format? A **message translator** allows the message to be adapted to suit the receiving services, but additional steps may still be necessary (e.g., how to route messages to different or multiple endpoints, how to split messages, and how to aggregate messages).

4.5.2 Pipes and Filters

The **pipes and filters architectural style** (along with other EIPs) enables structuring the more complex integrations required by enterprise applications. Messages pass through multiple processing steps/components (the filters), while components send messages down the channels (pipes) to which they are connected. The idea is that messages flow through the pipes, and as they reach a filter, they may be transformed before reaching the sink (the target of the message).

One of the most common examples in the field of EAI is the *Loan and Broker design*: the *loan company* sends a request, the *Credit Bureau* validates the request, the *Rule Base* decides which office should process the request, and then the *Banks* will be notified to accept or reject the loan. All the responses will be aggregated and sent back to the loan company. The shaded area in the Loan Broker example represents the Loan service.

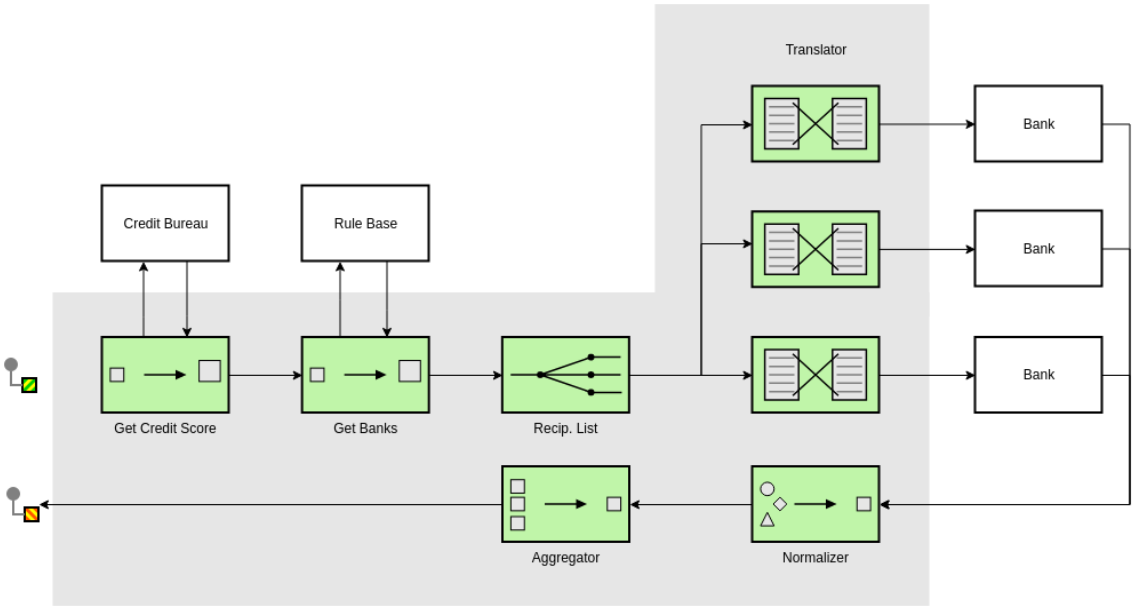


Figure 4.12: Loan Broker Design

This is a list of additional patterns used that haven't been described until now:

- **Content Enricher**: This pattern uses information within the incoming message (e.g., key fields) to retrieve data from an external source. In the example shown in Figure 4.12, the calls *Get Credit Score* and *Get Banks* are examples of content enrichment. Retrieved data is appended to the message. Original information may either be retained or discarded.
- **Routers**: There are mainly two categories for message routing: *content-based routers* route messages based on message type (in headers) or message content (in the body), while *context-based routers* route based on contextual information retrieved from a central configuration location.

In general, a message router connects to multiple channels and contains the logic to decide which channel it should send to.

Message routing pattern	Consumed msgs	Published msgs	Stateful?
Message Filter	1	0 or 1	No*
Content-based Router	1	1	No*
Recipient List	1	multiple (incl. 0)	No
Splitter	1	multiple	No
Aggregator	multiple	1	Yes

Figure 4.13: Message routing patterns

A **recipient list** inspects an incoming message, determines the list of desired recipients, and forwards the message to all channels associated with the recipients in the list. A **content-based router** enables routing each message to the correct recipient based on the message content.

- **Normalizer**: This pattern enables translating messages to match a common data format. This is usually realized as a composition of multiple patterns: it is possible to combine several patterns to create *composite patterns* (as is the case with the normalizer).

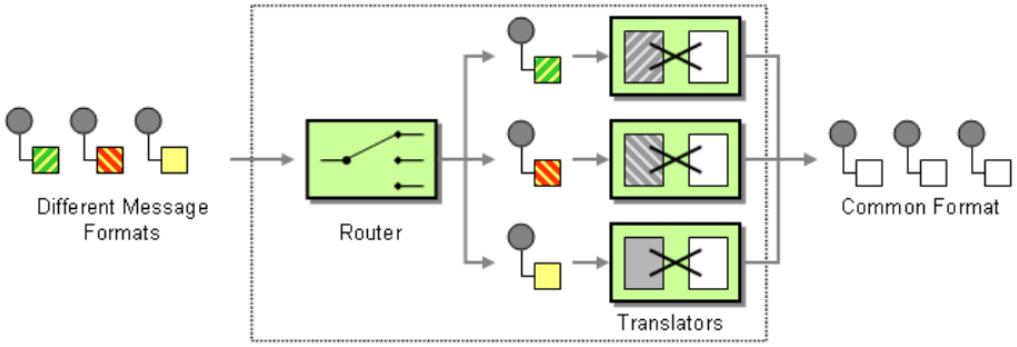


Figure 4.14: Normalizer schema

- **Aggregator**: A (*stateful*) aggregator collects and stores individual messages until a complete set of related messages has been received (see Figure 4.13 for reference). This is usually utilized when some integration steps may occur in parallel: independent processes can be executed in parallel, and the results of these processes can be aggregated to inform subsequent decisions on how to proceed.

Chapter 5

Cloud-based Software

The revolution of cloud-based software began with the combination of two aspects: *powerful computer hardware* and *high-speed networking*, which have led to the development of cloud computing, and *virtualized resources* (e.g., compute, storage, platform) accessible on demand.

The main advantages of cloud-based software are:

- **Scalability**: Maintains performance as load increases (useful when developers do not know the number of the product's customers).
- **Elasticity**: Adapts server configuration to changing demands (we want to avoid *underprovisioning* and *overprovisioning*).
- **Resilience**: Maintains service in the event of server failures.
- **Cost**: Most cloud services are *pay-per-use* (no need to buy hardware).

5.1 Virtualization and Containers

As mentioned before, one of the main aspects that led to cloud-based software is *virtualized resources*. When we refer to *compute power*, we mean servers and many other devices. It is possible to host many virtual servers inside the same physical server, which utilize virtual machines.

The left stack of Figure 5.1 refers to a machine with an *OS* and a *Hypervisor* (which enables virtualization; the one shown in the figure is a Level 2 Hypervisor, which depends on the OS. The Level 1 Hypervisor is bound to the hardware). The orange boxes are **virtual machines**, which are partitions of the physical machine.

The right stack consists of **containers**: rather than providing isolation (as it is with virtual machines), containers share the same OS to exploit the OS kernel's capability of allowing multiple isolated user-space instances. Containers run on a *container manager*, which operates on the OS. The advantage of OS sharing (which leads to much lighter and faster container startup) is counterbalanced by security issues (such as memory contamination from other containers).

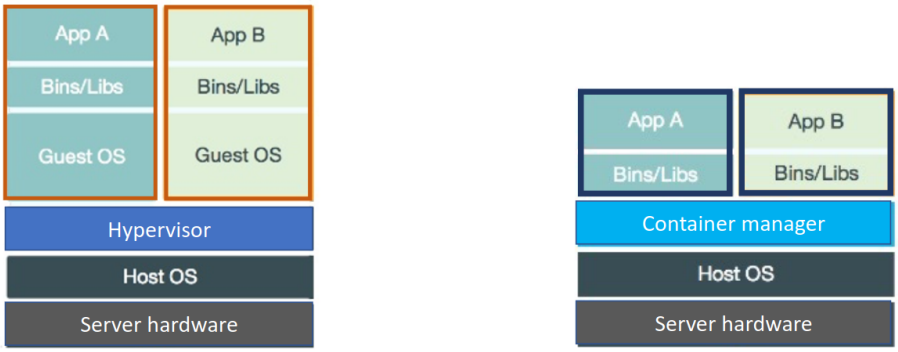


Figure 5.1: Virtual Machines and Containers

5.1.1 Docker

Portability of applications has been one of the biggest problems in software engineering: traditionally, there was one environment for developing and testing and another for production and distribution (the environment where the application was deployed and distributed to the user). These two environments were not the same, which meant that more time and money were required each time an application was moved between different environments.

Docker is a platform that allows us to run applications in an isolated environment. Docker enables the development and running of portable applications by utilizing containers: once an application is developed, it is encapsulated inside a container. When the container is opened in a new environment, it will not have portability issues, thanks to **Docker Engine** (used for creating and running containers). The *Docker platform* also contains **Docker Hub**, which is used for distributing containers.

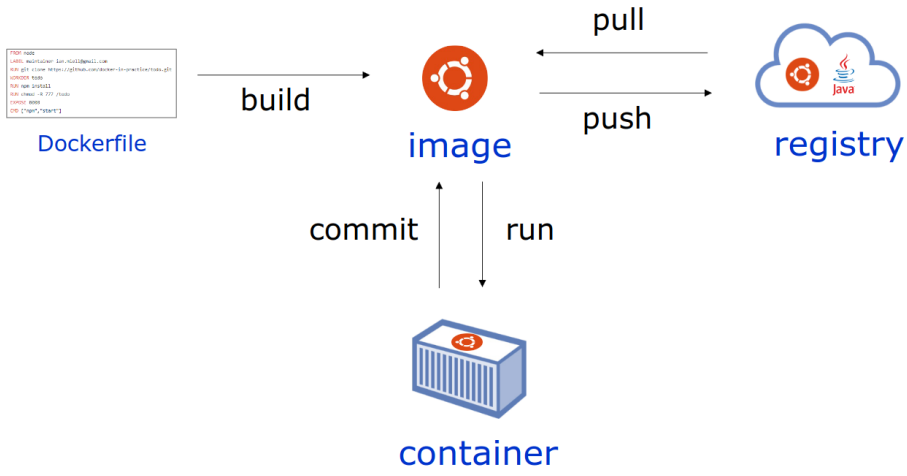


Figure 5.2: Docker Life Cycle

Software components are packaged into **images**, which are utilized as read-only templates to create and run **containers**. Since the templates are read-only, *exter-*

nal volumes are required to ensure data persistence. The images are stored in a (private or public) **Docker registry**: a registry structured in repositories, where each repository contains a set of images for different versions of software (images are identified by pairs repository:tag). Images are structured into layers, where each layer is in turn an image (the lowest layer is called the base image).

As we can see from Figure 5.2, the command **pull** retrieves an image from the *registry*, while **push** uploads an image to the *registry*. Images can be built from a *Dockerfile* (see the documentation[1]) using the command **build**. Images can also be run in a *container* using the command **run**. To modify the *application* (and save a new version), the command **commit** must be used.

When an image generates multiple processes inside a container, the life of the container will be associated with the main process. It is good practice to create one container per process, so when the life of a process ends, it will not affect the other processes. In cases where the application consists of multiple services, Docker provides a tool called **Docker Compose** (see the documentation[2]), which not only allows running one container per service but also manages networking (it is possible to call the locally hosted services using their names) and communication between containers.

5.2 Everything as a Service

There are different deployment models of cloud computing:

- **IaaS**: IaaS provides (virtualized) servers, storage, and networking. The IaaS provider manages all infrastructure. The client is responsible for all other aspects of the deployment (e.g., OS, application). Some examples include EC2 and S3.
- **PaaS** (Platform as a Service): PaaS provides a complete platform as a service (VMs, OS, services, SDKs, etc.). The PaaS provider manages infrastructure, OS, and enabling software. The client is responsible for installing and managing the application. Some examples include Heroku, Azure, and GAE.
- **SaaS** (Software as a Service): SaaS provides software on demand for use, accessible via thin clients or APIs. The SaaS provider manages infrastructure, OS, and application. The client is responsible for nothing, which is why this is the service with the largest market share. An example is Salesforce.com.

The following Figure 5.3 represents an analogy of what each deployment model provides, and what clients need to manage on their own.

[1] https://docs.docker.com/engine/reference/builder/
[2] https://docs.docker.com/compose/

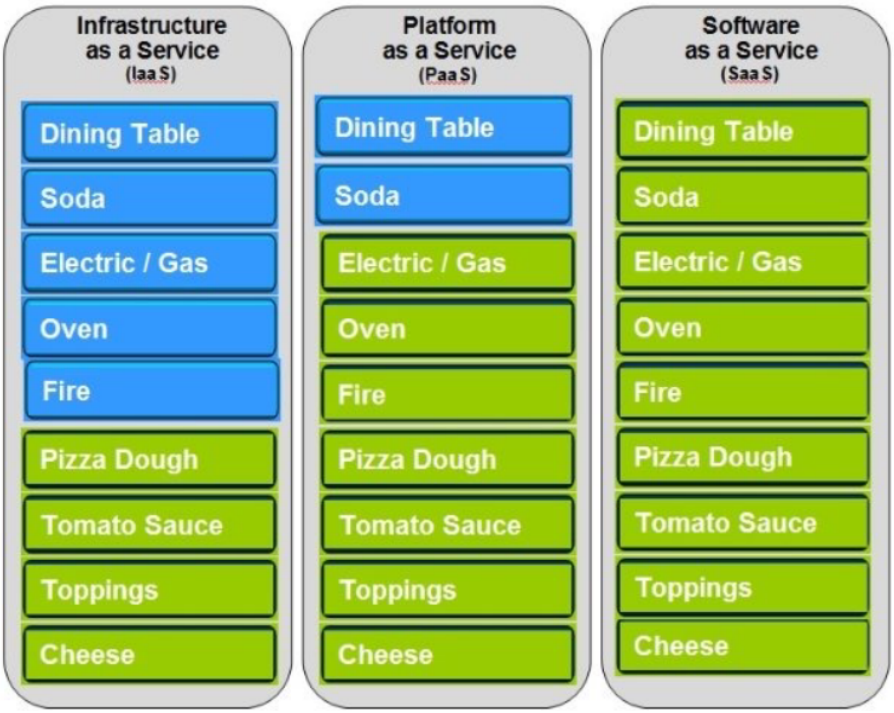


Figure 5.3: Pizza as a Service

5.2.1 SaaS

Before SaaS (*Software as a Service*), software products were initially installed on customers' computers. Customers had to configure the software and manage software updates, while software product companies had to maintain different product versions. With SaaS, the product is delivered as a service, allowing customers to avoid installation; instead, they pay a subscription and access the product remotely.

Some benefits of SaaS are:

- **Cash flow**: There is a regular cash flow, which means that customers pay periodic subscriptions or pay-per-use.
- **Update management**: The client controls product updates, so customers receive updates simultaneously. This eliminates the need to maintain multiple versions, resulting in reduced costs.
- **Continuous deployment**: The client can deploy new software versions as soon as changes have been made and tested.
- **Payment flexibility**: Different payment options can attract a wider range of customers (e.g., small companies or individuals can avoid paying large upfront software costs).

- **Try before you buy**: The client can make an early free or low-cost product available to gather customer feedback (e.g., a product free for a few weeks to collect feedback).
- **Data collection**: The client can easily collect data on product usage and customer interactions.

Some cons of SaaS are:

- **Privacy regulation compliance**: Different countries have varying strict laws regarding the storage of personal information.
- **Security concerns**: Customers may be hesitant to relinquish data control to an external provider, but this data needs to be managed by the SaaS provider. What happens when the SaaS provider experiences a data breach? Will they notify the clients?
- **Network constraints**: These can limit response times during heavy data transfers.
- **Data exchange**: If the cloud does not provide suitable APIs, exchanging data can be difficult. This is contract-based: depending on the offering, you might have access to different sets of data.
- **Loss of control over updates**: This freedom of updates can lead to excessive changes (e.g., new functionalities added daily, impacting the user interface).
- **Service lock-in**: It can be challenging to move code from one SaaS provider to another.

After deciding that SaaS will be the deployment model, several design issues must be addressed:

- **Local or remote processing**: Some features can be executed locally, which reduces network traffic and increases response speed but may also increase power consumption for battery-powered devices.
- **Authentication**: There are usually three kinds of authentication: the product's own authentication system, federated authentication (based on mutual trust relationships between a Service Provider (SP) such as an application vendor and an external Identity Provider (IdP)), and individual credentials (e.g., Google, LinkedIn).
- **Information leakage**: If the service is offered to multiple organizations, there's a security risk when a member of one organization handles data belonging to another.

- **Multi-tenant or multi-instance database management**: Multi-tenant systems use a single repository (with advanced techniques to partition data and control those partitions), while multi-instance systems have separate copies of the system and database.

5.2.2 Multi-tenant and Multi-instance Systems

Multi-tenant Systems

Multi-tenant systems use a single database schema shared by all system users. The items in the database are tagged with a tenant identifier to provide "logical isolation," meaning that users will only be able to access data linked to their tenant, as shown in Figure 5.4.

Stock management					
Tenant	Key	Item	Stock	Supplier	Ordered
T516	100	Widg 1	27	S13	2017/2/12
T632	100	Obj 1	5	S13	2017/1/11
T973	100	Thing 1	241	S13	2017/2/7
T516	110	Widg 2	14	S13	2017/2/2
T516	120	Widg 3	17	S13	2017/1/24
T973	100	Thing 2	132	S26	2017/2/12

Figure 5.4: Multi-tenant database schema example

Some advantages of a *multi-tenant system* are:

- **Resource utilization**: The SaaS provider has control over all resources used by the software and can optimize it to make effective use of these resources.
- **Security**: Multi-tenant databases must be designed for security since data for all customers are stored in the same database. They are, therefore, likely to have fewer security vulnerabilities than standard database products. Security management is also simplified as there is only a single copy of the database software to patch if a security vulnerability is discovered.
- **Update management**: It is easier to update a single instance of software rather than multiple instances. Updates are delivered to all customers simultaneously, ensuring that everyone uses the latest version.

However, there are also challenges associated with a *multi-tenant system*:

- **Inflexibility**: Customers must all use the same database schema, with limited scope for adapting this schema to individual needs.

- **Security**: Since data for all customers is maintained in the same database, there is a theoretical risk of data leaking from one customer to another. More seriously, if there is a database security breach, it can affect all customers.

- **Complexity**: Multi-tenant systems are usually more complex than multi-instance systems due to the need to manage many users, which increases the likelihood of bugs in the database software.

Mid-size and large businesses rarely want to use generic multi-tenant software; they often prefer a **customized version** adapted to their own requirements. One solution is to add extra fields to each table and allow customers to use these fields as they wish. However, predicting how many extra columns to include is challenging (too few will be insufficient, and too many will lead to wasted space), and different customers are likely to need different types of columns. A second solution is to add a field to each table that identifies a separate "extension table," allowing customers to create these extension tables to reflect their needs. This approach adds a new layer of complexity to a system that is already complex.

Security is the major concern for corporate customers using multi-tenant databases. Since information from all customers is stored in the same database, a software bug or attack could expose the data of some or all customers to others. To mitigate these risks, *multilevel access control* (where access to data is controlled at both the organizational and individual levels) and *encryption of data* (to ensure that corporate users' data cannot be viewed by people from other companies in the event of a system failure) are typically applied. Usually, only sensitive data is encrypted.

Multi-instance Systems

A *multi-instance system* allows each customer to have its own system tailored to its needs, including its own database and security controls. This type of system is conceptually simpler than multi-tenant systems and avoids security concerns such as inter-organization data leakage.

There are two ways to implement multi-instance systems:

- **Virtual Machine-based**: Each software instance and database for a customer runs in its own Virtual Machine. This allows all users from the same customer to access the shared system database.

- **Container-based**: Each user has an isolated version of the software and database running in a set of containers. This solution is best suited for products where users mostly work independently with little data sharing (e.g., for individual users).

It is also possible to run containers on a virtual machine: a business could have its own VM-based system and run containers on top of this for individual users.

Some advantages of a *multi-instance system* are:

- **Flexibility**: Each instance of the system can be tailored and adapted to the customer's needs.
- **Security**: Each customer has its own database, so there's no possibility of data leakage from one customer to another.
- **Scalability**: Instances of the system can be scaled according to the needs of individual customers. For example, some customers may require more powerful servers than others.
- **Resilience**: If a software failure occurs, it will probably affect only a single customer, allowing other customers to continue working as normal.

However, there are also challenges associated with a *multi-instance system*:

- **Cost**: It is more expensive to use multi-instance systems due to the cost of renting multiple VMs in the cloud and managing multiple systems. Since VMs have a slow startup time, they may need to be rented and kept running continuously, even if there is very little demand for the service.
- **Update management**: Managing updates for many instances can be complex, especially if the instances have been tailored to specific customer needs.

5.3 Cloud Software Architecture

During the initial stages of designing a cloud software architecture, several **architectural decisions** must be made: Should the software use a multi-tenant or multi-instance database? (*Database organization*) What are the software scalability and resilience requirements? (*Scalability and resilience*) Should the software structure be monolithic or service-oriented? (*Software structure*) The answers to these questions will guide the decision regarding which cloud platform should be used for deployment and delivery.

5.3.1 Database Organization

There are three possible ways to provide a customer database in a cloud-based system:

1. As a *multi-tenant system*, shared by all customers of 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 across several containers.

Some factors that can help decide the appropriate method for providing a customer database in a cloud-based system include:

- *Target customers*: Do customers require different database schemas and personalization? If so, use a multi-instance database.
- *Transaction requirements*: Is it critical that the product supports ACID transactions, ensuring that data is consistently accurate at all times? If so, consider a multi-tenant database or a VM-based multi-instance database.
- *Database size and connectivity*: How large is the typical database used by customers? How many relationships exist between database items? A multi-tenant model is usually best for very large databases, as it allows you to focus efforts on optimizing performance.
- *Database interoperability*: Will customers wish to transfer information from existing databases? What are the differences in schemas between these and a possible multi-tenant database? What software support will they expect for the data transfer? If customers have many different schemas, a multi-instance database should be used.
- *System structure*: Are the developers using a service-oriented architecture for the system? Can the customer database be split into a set of individual service databases? If so, use a containerized multi-instance database.

5.3.2 Scalability and Resilience

Scalability

Scalability (the ability to adapt automatically to load changes) can be achieved by adding new virtual servers (**scaling out**) or by increasing the power of an existing server (**scaling up**) in response to increasing load. Scaling out is typical of cloud-based systems: the product must be organized so that individual software components can be replicated and run in parallel, while load-balancing mechanisms direct requests to different instances of the components (e.g., using PaaS).

Resilience

Resilience (the ability to continue delivering critical services in the event of system failure or malicious activity) is realized using a technique known as **hot standby**, which involves maintaining replicas of software and data in different locations (see Figure 5.5). Database updates are mirrored, and a system monitor continuously checks system status. A cheaper alternative is called **cool standby**, where, in the event of system failure, the data is restored from a backup (meaning the system will be unavailable until the data restoration is complete).

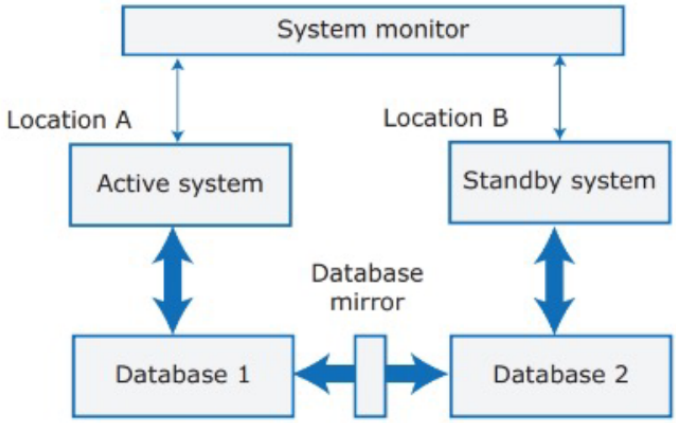


Figure 5.5: Hot standby technique schema

5.3.3 Software Structure

The main choice here is between **monolithic systems** or **fine-grained, stateless services**. The latter uses independent services that can be replicated, distributed, and migrated. This approach is particularly suitable for cloud-based software with services deployed in containers. The monolithic approach is often employed to build prototypes, especially if a fine-grained system better fits the specific situation.

When choosing a cloud platform, there are two main points to consider:

1. **Technical issues**: *Expected load and load predictability*, *resilience*, *supported cloud services* (which cloud services a provider can offer), and *privacy and data protection* (some EU countries have strict requirements regarding data protection and storage locations → necessitating cloud providers to offer guarantees on storage locations).

2. **Business issues**: *Developer experience* (the languages and technologies depend on the team's experience), *Service-level agreements* (guarantees provided by the cloud provider), *Portability and cloud migration* (to create an "exit plan" to mitigate vendor lock-in), *target customers* (customers may have older systems that need to interoperate with the new ones), and *cost* (including all developer and technology-related expenses).

5.4 Container Orchestration

As mentioned in the previous section, containers provide a lightweight mechanism for isolating an application's environment. Container images can be executed reliably on any machine, offering portability (we used Docker) from development to deployment.

Automated deployment, such as that provided by Docker, often has limited capability for optimizing memory usage and may not utilize resources effectively. Additional concerns arise: What happens if your container dies? What happens if the machine running your container fails? What if you have multiple containers that need to communicate with one another (*inter-node communication*)? How do you enable networking between containers? If your production environment consists of multiple machines, how do you decide which machine to use to run your container?

The solution comes with **container orchestration**: the architect of a system defines the containers to be run and then hands over control to the *container orchestration platform* to realize that vision.

5.4.1 Kubernetes

One of the most used container orchestration platforms is **Kubernetes**, which manages the entire lifecycle of individual containers, spinning up and shutting down resources as needed (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 the underlying individual containers are created and destroyed. Given a set of container workloads to run and a set of machines in a cluster, the container orchestrator examines each container and determines the optimal machine to schedule that workload.

Kubernetes will ingest through its API a "*Desired State Management*," which is a YAML manifest. The manifest specifies, given a set of images, which are the **Pods** (groups of images). Kubernetes will insert inside the same node all the images that are defined in the same Pod, generating as many replicas as defined. A node can handle multiple Pods. If one of the nodes (worker = container host) fails, Kubernetes detects the broken node and starts a new replica, as shown in Figure 5.6. The *Kubernetes cluster services* not only handle that but also handle communication between the nodes (the existing ones and the ones that will be created).

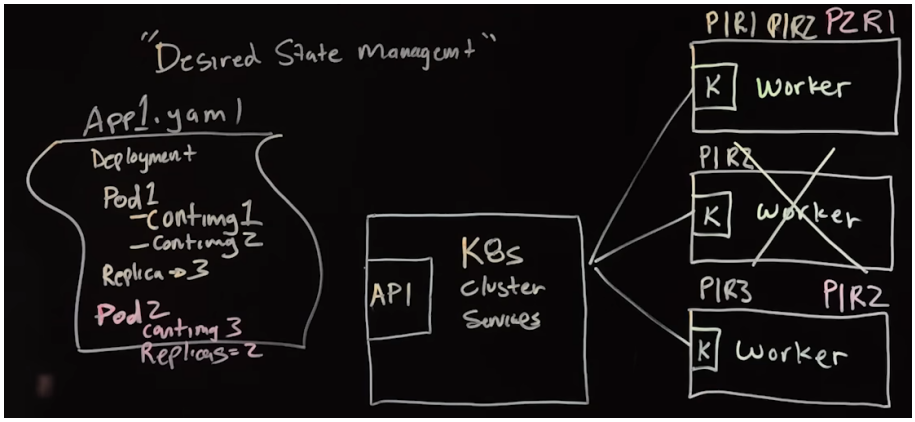


Figure 5.6: Kubernetes' management idea

5.4.2 Kubernetes Design Principles

Here are the design principles of Kubernetes:

- **Declarativeness**: A declarative approach is to simply define the desired state of a system (with the manifest). K8s will detect when the actual state of the system doesn't meet our expectations and will intervene to fix the problem, making our system self-healing. The desired state is defined by a collection of objects: each object has a specification in which you provide the desired state and a status that reflects the current state of the object. K8s constantly polls each object to ensure that its status is equal to the specification: if an object is unresponsive, K8s will spin up a new version to replace it; while if an object's status has drifted from the specification, K8s will issue the necessary commands to drive that object back to its desired state.

- **Distribution**: Kubernetes provides a single unified interface for interacting with a cluster of machines. In this way, developers don't have to worry about communicating with each machine individually.

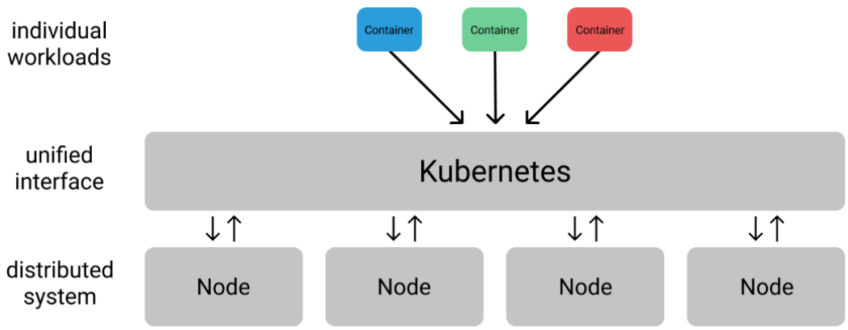


Figure 5.7: Kubernetes distribution schema

- **Decoupling**: Containers should be developed with a single concern in mind: microservice-based architecture. K8s naturally supports the idea of decoupled services that can be scaled and updated independently.

- **Immutable infrastructure**: To get the most from containers and container orchestration, an immutable infrastructure should be deployed. This means that instead of logging into a container on a machine to update a library, it is required to build a new container image, deploy the new version, and terminate the older version. The reason is that containers are designed to be ephemeral, ready to be replaced by another container instance at any time. Maintaining immutable infrastructure makes it easier to roll back applications to a previous state (e.g., if an error occurs) by simply updating the configuration to use an older container image.

5.4.3 Kubernetes objects

There are many objects defined by Kubernetes that can be specified in the **manifests** (either in JSON or YAML). Here's a list of the main ones:

- **Pod**: The minimum Kubernetes deployment unit, which is composed of one or more (tightly related) containers, a shared networking layer, and shared filesystem volumes. Pods can't be distributed by themselves. Each Pod is assigned a unique IP address that we can use to communicate with it.

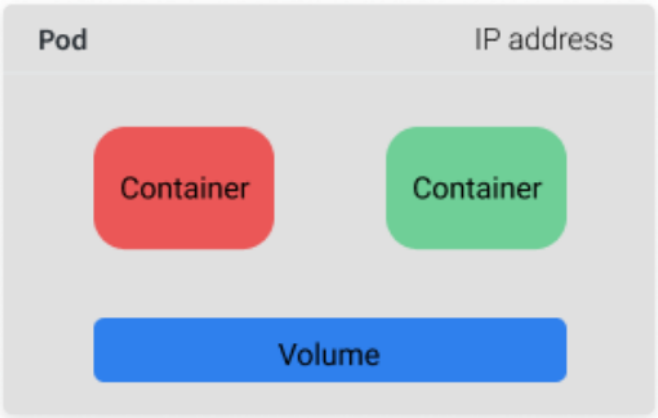


Figure 5.8: Kubernetes Pod schema

- **Deployment**: It includes a collection of Pods defined by a template and a replica count (number n of how many copies of the template we want to run). The cluster will always try to have n Pods available.

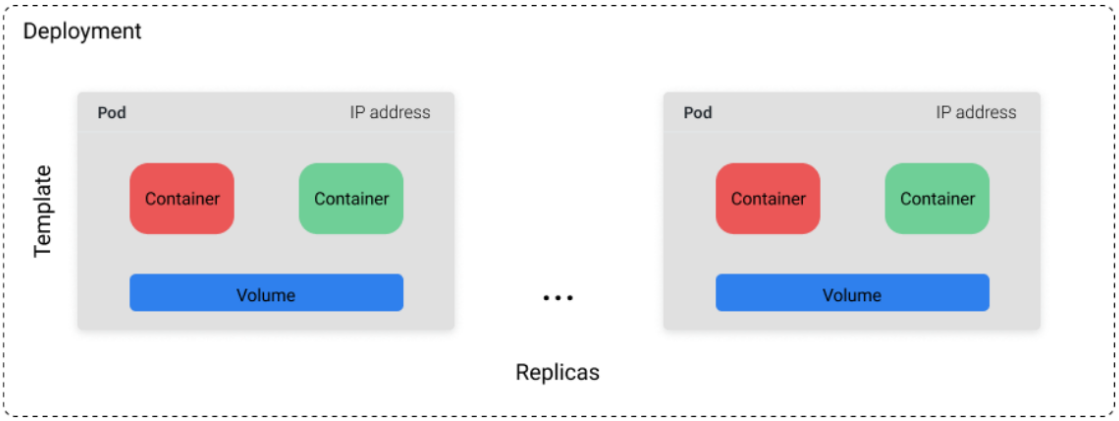


Figure 5.9: Kubernetes Deployment schema

- **Service**: The 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 (even if the set of Pods running as part of the Deployment can change at any time) based on labels (key-value pairs) that we define in the Pod metadata.

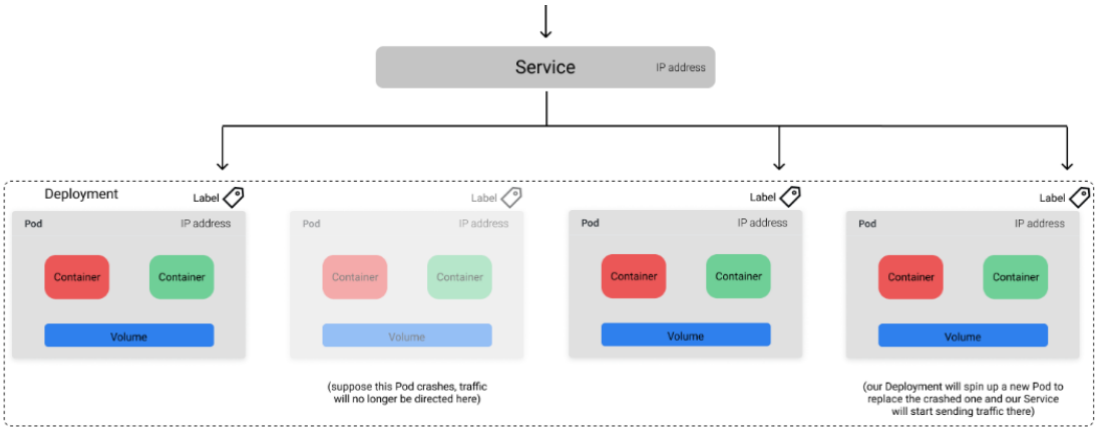


Figure 5.10: Kubernetes Service schema

- **Ingress**: To expose the application to traffic external to the cluster, it is necessary to define an Ingress object. It is possible to select which Services to make publicly available and expose applications behind a stable endpoint (only available to internal cluster traffic).

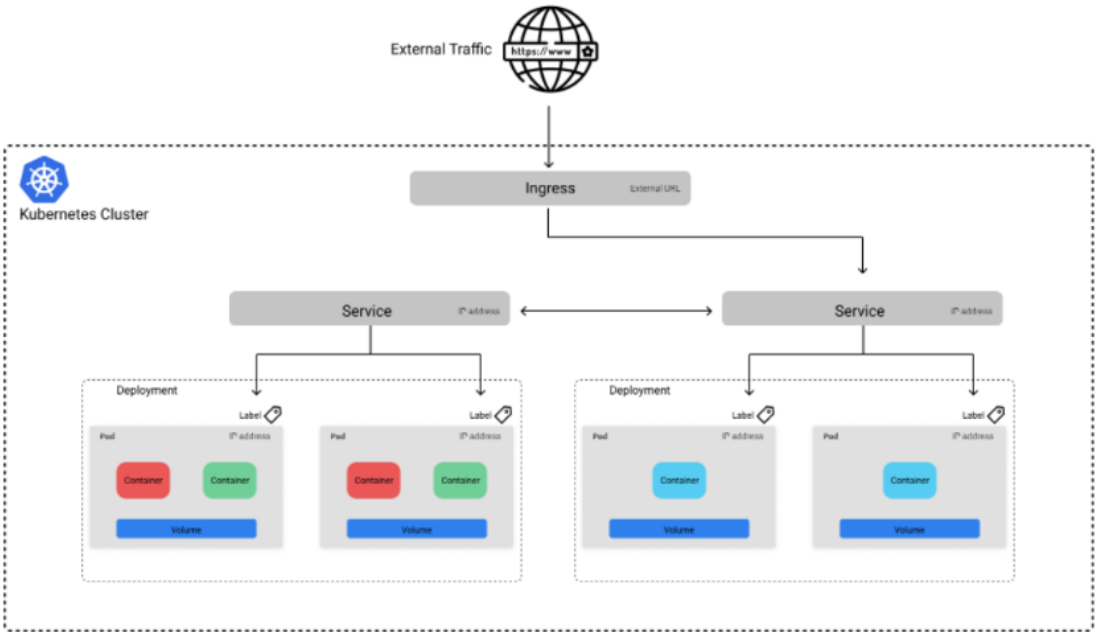


Figure 5.11: Kubernetes Ingress schema

5.4.4 Kubernetes control plane

In this subsection, a high-level description of how Kubernetes works underneath will be provided. There are two types of machines in a cluster (set of machines): the **master node** (often single), which is a machine that contains most of the control plane components, and the **worker nodes**, which are machines that run the application workloads.

Master node

Users provide new/updated object specifications (manifests) to the API server of the master node. The **API server** validates update requests (with some input verification) and acts as a unified interface for questions about the cluster's current state. The *state of the cluster* (cluster configuration, object specifications, object statuses, nodes in the cluster, object-node assignments, etc.) is stored in a distributed key-value store named *etcd*.

The Master node (as shown in Figure 5.12) is composed of the *API Server*, the distributed key-value store **etcd** (which contains the entire state of the cluster), a *controller-manager*, and a *scheduler*. The reason why the API server works as an intermediary is to reinforce the decoupling principle: each of these components is responsible for only one task (which also makes failure recovery easier).

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

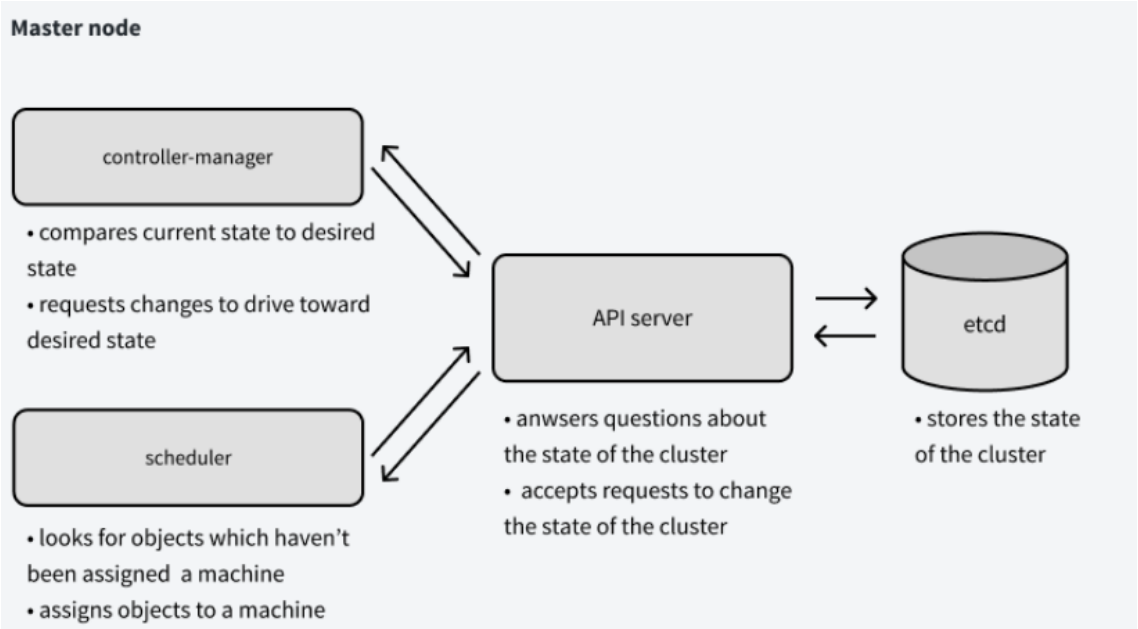


Figure 5.12: Kubernetes Master node structure

The **scheduler** decides where objects should be run, as shown in the below Figure 5.13. The scheduler asks the API server which objects haven't been assigned to a machine (1), determines which machines those objects should be assigned to (2), and then replies back to the API server (3) to reflect this assignment (4).

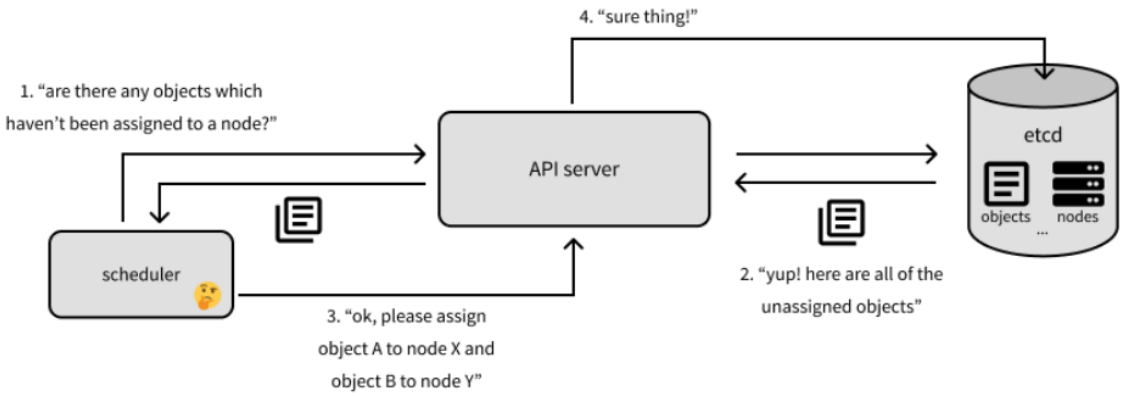


Figure 5.13: Scheduler life-cycle

Worker node

Every worker node contains a **kubelet**, which acts as the node's "agent" that communicates with the API server to see which container workloads have been assigned to the node. It is responsible for spinning up pods to run these assigned workloads. When a node first joins the cluster, the kubelet announces the node's existence to the API server so the scheduler can assign pods to it. Workers also contain **kube-proxy**, which enables containers to communicate with each other across the various nodes in the cluster. With Figure 5.14, it is possible to observe the entire Kubernetes structure.

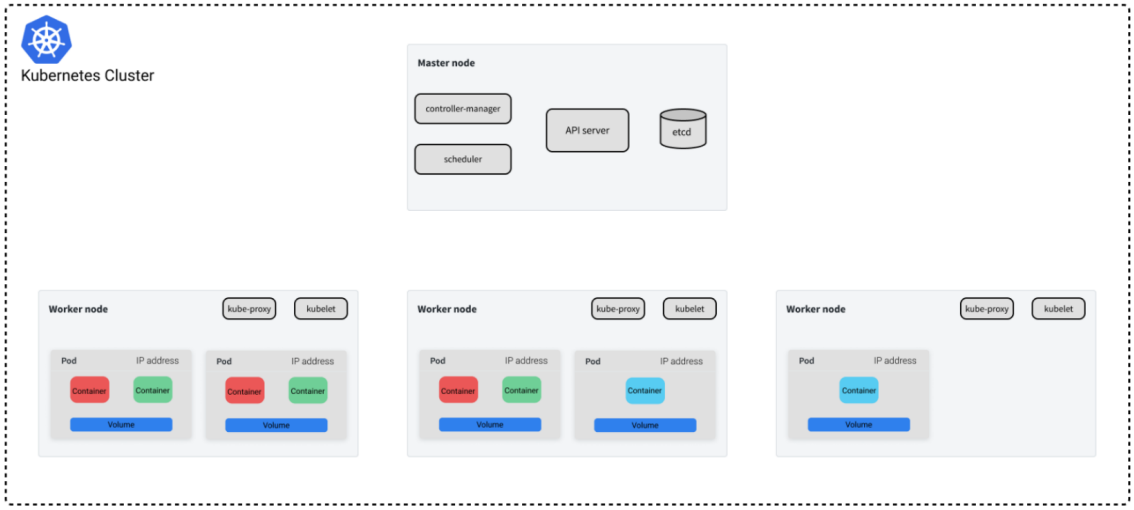


Figure 5.14: Kubernetes entire structure

To conclude this chapter, in which cases is the use of Kubernetes not the best choice?

- 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.

There's also a way to do container orchestration with Docker (it's called *Docker Swarm*), but Kubernetes is a much stronger orchestrator. Still, they have a partnership, so there's no lock-in between Docker and Kubernetes.

Chapter 6

Microservices Architecture

This chapter looks at **Microservices Architectures**, covering the basic ideas, principles, and benefits that make it popular in modern software development. First, let's answer an important question: How do we break a system into *components*?

Breaking a system into components is a key step in adopting Microservices Architectures. Here are some reasons for doing this: it allows different teams to work in parallel, promotes reusability, makes it easier to distribute systems across multiple computers, and supports smooth replication, parallel execution, and migration to take full advantage of cloud capabilities like scalability, reliability, and flexibility.

As we explore the main principles of Microservices Architectures, we highlight the importance of stateless services that store *persistent information in local databases*. This approach helps ensure agility and scalability, allowing services to move between virtual servers as needed, while also helping to create resilient and fault-tolerant systems.

Software components that can be accessed over the Internet are called **software services**. A service is accessed through its *published interface*, where all implementation details are hidden. Given an input, a service produces a corresponding output without side effects. Services *do not maintain any internal state* because state information is either stored in an explicit database or maintained by the service requestor. Interactions where the state is maintained involve service requests that have the state and the service result integrated with the answer to the request.

Some history that led to microservices architectures:

- **Service-oriented architecture** (**SOA**, late 1990s): applications made up of many blocks, where each block is a class or an object. Those (*independent*) blocks could have been written in different technologies since they communicated with each other (or the Internet) through public interfaces.

- **Web Services** (early 2000s): services were standard-based (XML, SOAP, WSDL, and many others), where the standards were used to describe the services (input, output, type of parameters, and so on). Standards were good, but having many standards made them hard to handle.
- **Modern service-oriented systems** (nowadays): much simpler interaction protocols, much simpler interfaces (with more efficient formats for encoding message data), which led to lower overhead and faster execution.

Amazon Web Services[1] had a significant impact on this service world, providing a manifesto with guidelines on **how services should be implemented**: a service should be related to a *single business function*; services should be completely *independent*, with their *own databases*; a service should manage its own user interface; it should be possible to replace/replicate a service without changing other services. This leads to **microservices**: generally small-scale stateless services that have a single responsibility.

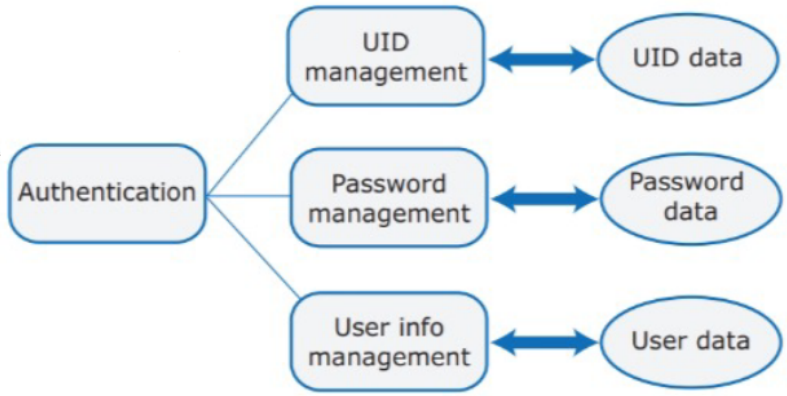


Figure 6.1: Example of a system using an authentication module

Taking a system that uses an authentication module (providing user registration, authentication using UID/password, two-factor authentication, user information management, password reset), we can see from Figure 6.1 an example of a segmentation into microservices. In order to *identify the microservices* that might be used for the authentication system, developers need to break coarse-grained features into more detailed functions, look at the data used, and identify a microservice for each logical data item to be managed (each service will have its own data). In the case that one service needs the data of another service, it can request it through interfaces or just replicate some critical data (facing consistency), minimizing the amount of replicated data management.

[1] https://aws.amazon.com/

6.0.1 Microservices

We have already defined microservices as *independent* (service interfaces not affected by changes to other services) and (generally) small-scale services that can be combined to create applications. It must be possible to modify and redeploy a service without changing/stopping other services.
Some of the characteristics that microservices must have are:

- **Self-contained**: Microservices do not have too many external dependencies. They manage their own data and implement their own user interfaces. This allows them to deploy and change the applications immediately.
- **Lightweight**: Microservices communicate using lightweight protocols so that the service communication overhead is low.
- **Implementation independent**: Microservices may be implemented using different programming languages and may use different technologies in their implementation.
- **Implementation deployable**: Each microservice runs in its own process and is independently deployable using automated systems.
- **Business-oriented**: Microservices should implement business capabilities and needs rather than simply provide a technical service. This characteristic coincides with Agile frameworks.

Two of the measures used for microservices are **coupling** (which measures the number of inter-component relationships) and **cohesion** (which measures the number of intra-component relationships). Developers should achieve *low coupling* and *high cohesion*: *low coupling* means independent services and independent updates, while *high cohesion* means less inter-service communication overhead.

Each service should do one thing only, and it should do it well (**Single Responsibility Principle**). In order to understand *how big a microservice should be*, we can use *the rule of twos*: a service can be developed, tested, and deployed by a team in two weeks. The team can be fed with two large pizzas (8-10 people).

This amount of people is required because they have to implement service functionality, develop code that makes the service completely independent, process incoming and outgoing messages, manage failures (there will most likely be service and interaction failures), manage data consistency (which becomes severe in the case of data replication) when data are used by other services, maintain the service's own interface, test the service and service interactions, and support the service after deployment (you build it, you run it). Years ago, there were different teams for development and production stages, but now DevOps (the same people who build and maintain the service) is widely used.

6.1 Architecture

The main motivations behind the need for microservices architectures are *shortening lead time for new features and updates* and *scaling*. Deploying microservices in separate containers allows for quickly stopping and restarting the microservice without affecting the other services, and replicas can be quickly deployed. If these two reasons are important for the application, then microservices are the way to go; otherwise (in the case of a simple application), a monolithic application will work just fine.

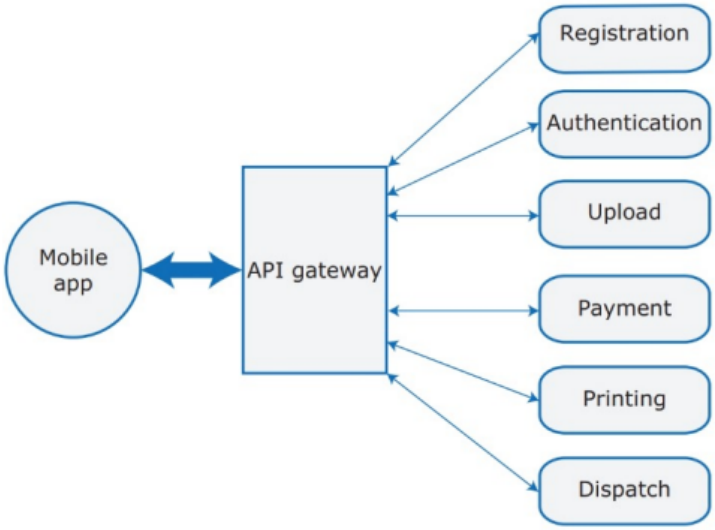


Figure 6.2: Photo-printing system for mobile devices

From Figure 6.2, it's possible to understand that microservices allow us to identify the basic services of an application, as we have separated services for each area of functionality. It also shows that it's good practice to use an **API gateway** to provide a single point of contact from the mobile application to the microservices. It will be the gateway's job to redirect requests to the appropriate microservice.

There are five different **architectural design decisions** that need to be addressed when it comes to microservices architectures:

1. What are the microservices that make up the system?
2. How should microservices communicate with each other?
3. How should data be distributed and shared?
4. How should the microservices in the system be coordinated?
5. How should service failure be detected, reported, and managed?

These points will be developed throughout the rest of the section.

6.1.1 System Decomposition

The **system decomposition** is one of the most important design choices, but it is also not an easy task. Having too many microservices will lead to *low cohesion* (which means a lot of communication overhead), while having too few will lead to *high coupling* (which means less independence for updates, deployment, and so on).

Some tips are:

- Find the right balance between fine-grained functionality and system performance.
- Follow the *common closure principle* (elements likely to be changed at the same time should stay in the same service).
- Associate services with business capabilities.
- Services should have access only to the data they need (with some data propagation mechanisms).

It's usually hard to think in terms of microservices; that's why it's common to start with a monolith and decompose it later. One way to understand how microservices should be split is to look at the data services have to manage.

6.1.2 Service Communications

Another design choice is how the services communicate.

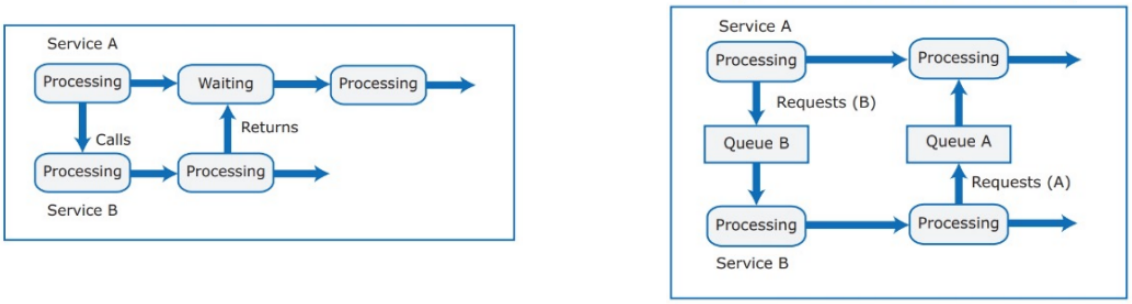


Figure 6.3: Synchronous vs. asynchronous service interaction

As shown in Figure 6.3, one choice is to go for **synchronous** or for **asynchronous** communication: the first one (which in the Figure is the one on the left) is easier to write and understand, while the second one (which in the Figure is the one on the right) has low coupling, is more efficient, but is more difficult to write (since a queue for the requests is required) and to understand.

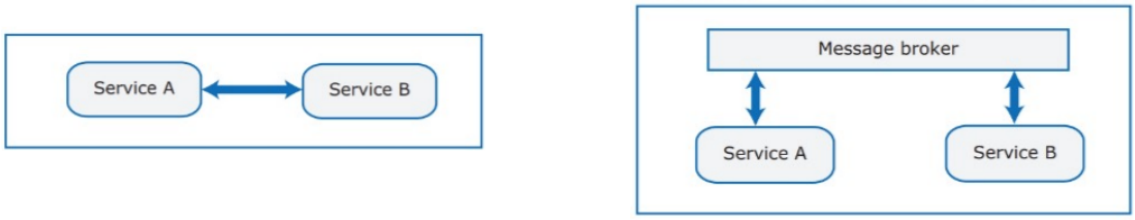


Figure 6.4: Direct vs. indirect service communication

The second choice is to either choose **direct** or **indirect** service communication: the first one consists of allowing one service to send messages directly to another service (it's simpler and faster, but the requester must know the other services' URIs), while the second one utilizes a message broker or a queue that finds the address of the requested service and handles the message translation. The indirect communication supports both synchronous and asynchronous interactions; it is easier to modify and replace services, but it is more complex and slower.

6.1.3 Data Distribution and Sharing

Each microservice should **manage its own data**, as we want to achieve independence between microservices (so other microservices shouldn't know how data is organized). There will be data dependencies between microservices, and some of the ways to deal with them are:

- Data sharing should be as minimal as possible.
- Sharing should be read-only, with a few services responsible for data updates.
- Include a mechanism to keep database copies used by replicated services consistent.

One mechanism to keep database copies consistent is **ACID**[2] **transactions**, where updates are serialized (which means that the database moves from one consistent state to another) to avoid inconsistency. While in distributed systems we must trade off data consistency and performance, with microservices systems, developers must design the system to *tolerate some degree of data inconsistency*. Two types of data inconsistency that must be managed are:

- **Dependent data inconsistency**: Actions or failures of one service can cause data managed by another service to become inconsistent.
- **Replica inconsistency**: Several replicas of the same service may be executing concurrently, each with its own database copy, and each updating its own database copy (so those databases should be *eventually*[3] *consistent*).

[2]ACID: Atomicity, Consistency, Isolation, Durability
[3]eventually: sooner or later

CAP Theorem

Brewer's conjecture (PDOC 2000) states that: "*it is impossible for a web service to provide Consistency, Availability, and Partition tolerance at the same time*", where *Consistency* means that each service returns the correct response to each request, *Availability* means that each request eventually receives a response, and *Partition tolerance* means that services can be partitioned into multiple groups, and the network can delay or lose arbitrarily many messages among services.

A later reformulation (made in 2012 by Gilbert & Lynch) states that: *In a network subject to communication failures, it is impossible for any web service to implement an atomic read/write shared memory that guarantees a response to every request* (which implies that you must choose between *consistency* and *availability*). A simple proof sketch is: Let S1 and S2 be two services belonging to two different network partitions, where every message from S1 to S2 may be delayed or lost. Since S2 can't always be sure whether S1 received the write request or not (because the confirmation from S1 to S2 may be delayed or lost), in the case that S2 sends two different write requests with values V1 and V2, S2 cannot distinguish between the following two cases: S1 received a write request of value V1 and sent an OK response; or S1 received a write request of value V2 and sent an OK response. This leads to a state in S2 that cannot determine whether to return V1 or V2 in response to a read request (because S2 should reply with the same value that S1 does).
Some practical solutions are:

- Guarantee availability and provide best-effort consistency (general solution).
- Guarantee strong consistency and provide best-effort availability (this solution is implemented when strong consistency is required (e.g financial applications)).
- Trade off consistency and availability (e.g., tolerate being one hour out of date but not one day out of date).

The Saga Pattern

How can we implement distributed transactions with a pattern? **The Saga pattern** consists of implementing each business transaction that spans multiple services as a *saga*, where a *saga* is a sequence of local transactions (see Figure 6.5). Each local transaction updates a database and triggers the next local transaction(s) in the saga. If a local transaction fails, then the saga executes a series of compensating transactions (instead of rolling back, since in microservices sometimes rolling back isn't possible).

Figure 6.5: Saga example

There are two ways of coordinating sagas:
Choreography, where each local transaction publishes events that triggers the next local transaction(s); or **Orchestration**, where an orchestrator tells participants which local transactions to execute.

There are again two ways of compensating transactions:
Backward model, where changes made by previously executed local transactions are undone; and **Forward model**, where the "retry later" principle is applied.

The *Netflix approach* consists of the replication of data in n nodes using *Apache Cassandra*[4] to achieve eventual consistency: the system tries to update as much as possible, ensuring that at least (n/2 + 1) of the replicas must respond. This is one example of a compromise that can be achieved with eventual consistency.

6.1.4 Service Coordination

There are two possible ways to implement **service coordination**:

- **Orchestration**, where an orchestrator tells the microservices how they should "play".
- **Choreography**, where microservices manage themselves.

We can think of orchestration as a semaphore and choreography as a roundabout.

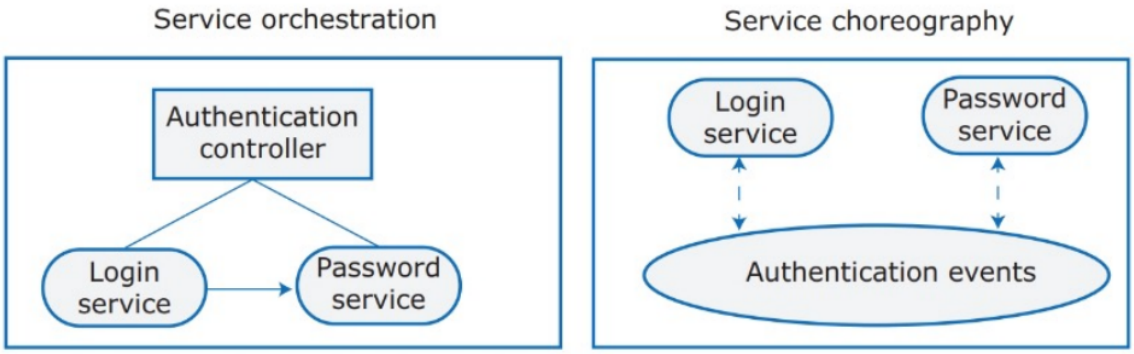


Figure 6.6: Example of orchestration and choreography

From Figure 6.6, we can see that in the case of service orchestration, there's an explicit controller orchestrating the system (in the example, the controller is named Authentication controller), while in the case of service choreography, an event-based synchronization is used (in the example, a Publish & Subscribe mechanism named Authentication events). The latter allows for the absence of an additional explicit controller (to avoid single points of failure), but it is harder to debug and recover in case of failure.

A good practice is to always start with orchestration (easier to design) and switch to choreography only if the product is inflexible or hard to update.

[4] https://cassandra.apache.org/_/index.html

6.1.5 Failure management

Something will go wrong, unavoidably (e.g., a service offers a functionality F by invoking two other services, each available 99% of the time and independent of one another. F will probably be NOT available for around 30 minutes each day). It's important, especially in distributed environments, to have some kind of **failure management** in case of failure: services must be designed to cope with failures.

There are mainly three types of failures:

- **Internal service failure**: These are conditions that are detected by the service and can be reported to the service requester in an error message. An example of this type of failure is a service that takes a URL as input and discovers that it is an invalid link.
- **External service failure**: These failures have an external cause that affects the availability of a service. A failure may cause the service to become unresponsive, and actions have to be taken to restart the service.
- **Service performance failure**: The performance of the service degrades to an unacceptable level. This may be due to a heavy load or an internal problem with the service. External service monitoring can be used to detect performance failures and unresponsive services.

One of the most typical solutions used for microservices is **circuit breakers**: they are design patterns that create resilient microservices by limiting the impact of service failures and latencies. The idea is that (take Figure 6.7 as a reference) an additional component (the circuit breaker) is added between the service client and the remote service (supplier). This additional component takes a request from the client, forwards it to the remote service, and then starts an internal timer. In the case that the remote service works fine, then the additional component receives the answer from the remote service and forwards it to the service client. If the timer triggers a timeout (either because the remote service failed or was too slow), then the circuit breaker tells the client that the remote service failed (avoiding the clients getting stuck waiting for the remote service response). In case of multiple failures, the circuit breaker *trips*, and for the duration of a timeout period, all attempts to invoke the remote service will fail immediately. After the timeout expires, the circuit breaker allows a limited number of test requests to pass through. If those requests succeed, the circuit breaker resumes normal operation. Otherwise, the timeout period begins again.

An interesting example of a circuit breaker is implemented by *Spotify*: the search service sometimes fails (since the service has to search through a huge index of songs), so Spotify quickly refreshes the page and uses the time users take to write the name of the song as a cooldown period. Error messages do not pop up.

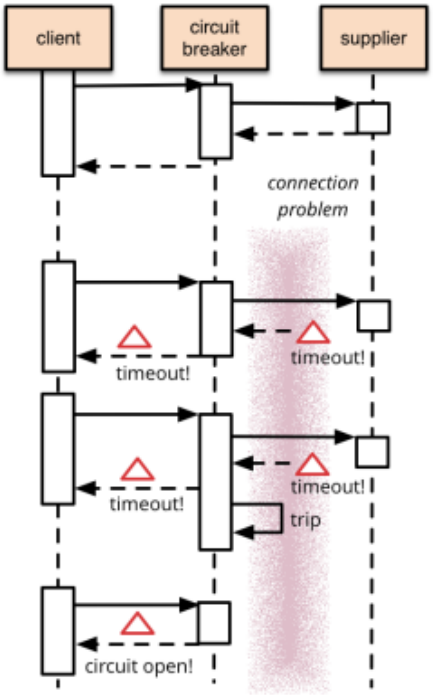


Figure 6.7: Example of a circuit breaker sequence diagram

There are many ways to test failure management. One interesting way implemented by Netflix is *Chaos Monkey*[5] (which comes from chaos engineering), a tool that randomly terminates VM instances and containers running inside a production environment. This type of testing is done during the production phase; that's why this is called a *bravely* conducted test.

6.2 RESTful services

Microservices are **RESTful services**. REpresentational State Transfer (REST) has four main principles:

- **Resource identification through URIs**: the service exposes a set of resources, where each resource is identified by a URI[6].
- **Uniform interface**: clients invoke HTTP methods to create/read/update/delete resources (with POST and PUT to create and update the state of a resource, DELETE to delete a resource, and GET to retrieve the current state of a resource).
- **Self-descriptive messages**: Requests contain enough contextual information to process messages, since information can be accessed in various formats (e.g.,

[5] https://netflix.github.io/chaosmonkey/

[6] Uniform Resource Identifier (URI) is a character sequence that identifies a logical (abstract) or physical resource

HTML, XML, JSON, plain text, PDF, JPEG, etc.). This is why resources are decoupled from their representation.

- **Stateless interactions through hyperlinks**: every interaction with a resource is stateless, which means that servers contain no client state, and any session state is held on the client. Stateless interactions rely on the concept of explicit state transfer.

There are many other things to say about REST, but they have already been covered during the bachelor's degree.

6.3 Service deployment

For the past 70 years, for each software project, a group of developers managed the entire life cycle based on project-based software engineering (requirement definition, specifications, prototype, testing, integration, quality of experience, and so on). Once the software is completed, the operations team (which did not participate in the development) puts the software into production (possibly in a different environment from where the software was developed) and then responds to customer problems.

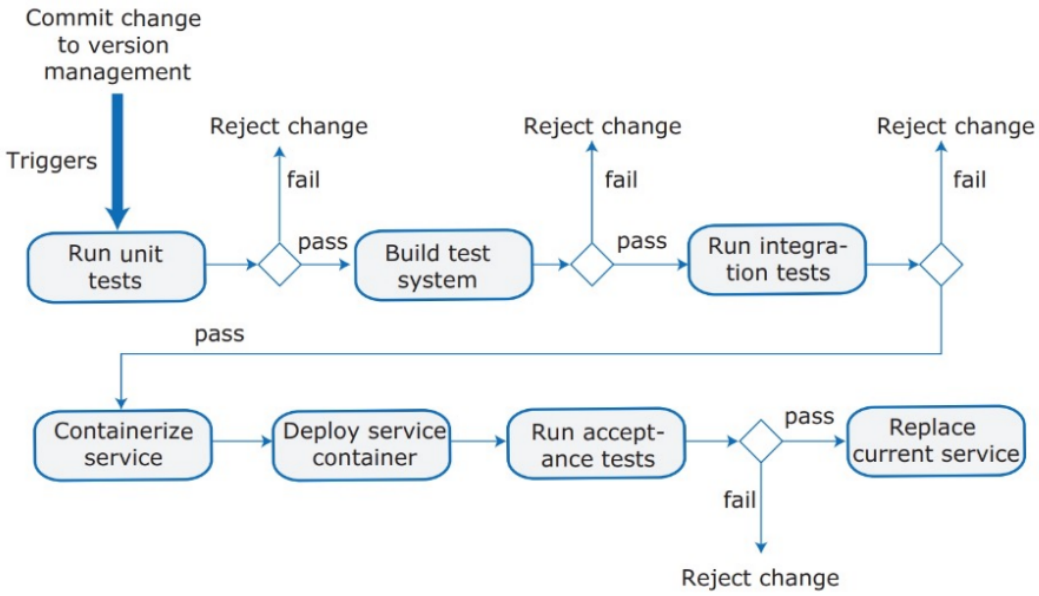


Figure 6.8: Continuous Deployment pipeline

The new era of software engineering is DevOps (a portmanteau of "development" and "operations"): the same team is responsible for service development, deployment, and management. From Figure 6.8, we can see the pipeline that software developers use when there's a commit request: the pipeline starts unit tests, makes the build, runs the integration, containerizes, deploys, runs the tests, and if everything goes right, replaces the current service with the new commit.

6.3.1 Monitoring

Testing cannot prevent 100% of unanticipated problems, which means that there is a need to monitor the deployed services. If you do not monitor microservices, you cannot know the state of the services, but excessive monitoring can lead to performance degradation. As with many things in microservices, a trade-off must be found.

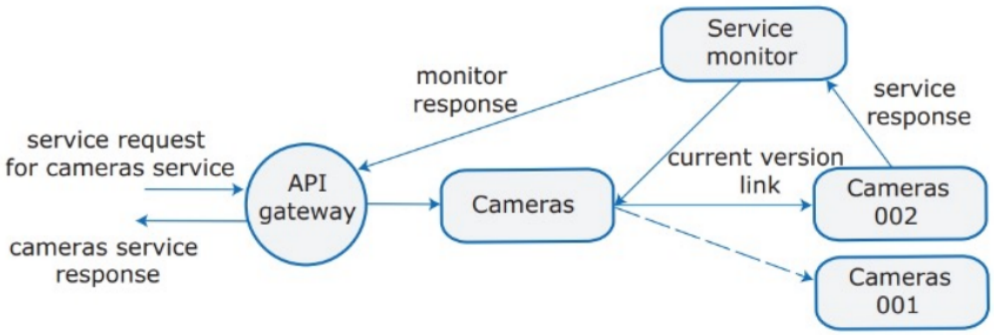


Figure 6.9: Monitoring example

The example shown in Figure 6.9 illustrates that it's also possible to increase reliability with monitoring: when a new version of a service is introduced, it's possible to maintain the old version while changing the "current version link" to point to the new service. If monitoring detects a problem with the new version of the "Cameras 002" service, it switches the "current version link" back to version 001 of the Cameras service.

6.4 Architectural smells

An **architectural smell** is a commonly used architectural decision that negatively impacts system lifecycle qualities. Once the microservice principles are defined, how can architectural smells affecting the design principles of microservices be detected and then resolved via refactoring?

This section will show, based on a *multivocal review* (which uses both scientific articles and technical websites), the most recognized *architectural smells* for microservices and the architectural *refactorings* to resolve them.
The **design principles** that will be taken into consideration are:

- Independent deployability: the microservices forming an application should be independently deployable.
- Horizontal scalability: the microservices forming an application should be horizontally scalable (which means the possibility of adding/removing replicas of individual microservices).
- Isolation of failures: failures should be isolated (avoiding the cascade effect).

- Decentralization: decentralization should occur in all aspects of microservice-based applications, from data management to governance.

6.4.1 Smells and refactoring

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

Independent deployability

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

Horizontal scalability

One common *architectural smell* for horizontal scalability is endpoint-based service interaction. This is a smell because, if there is an interaction with a specific service instance (as shown in Figure 6.10), adding more replicas of the service won't scale the application since the interaction is made with only one of the replicas. Some *refactorings* for the described *smell* are to add a service discovery that will indicate to the requester which replica they must use (applied in 55% of cases), add a message router that will redirect the request to one of the replicas (e.g., load balancer, applied in 31% of cases), or add a message broker that will collect the requests, which are then processed by the replicas (e.g., message queue, applied in 14% of cases). The reason that the message broker is the least used is that the code of the microservice has to be modified.

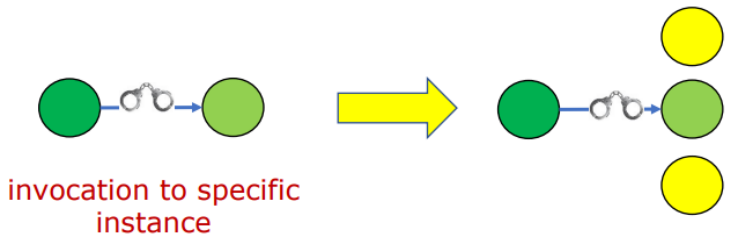


Figure 6.10: Endpoint-based service interaction

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

Isolation of failures

One *architectural smell* for isolation of failures is wobbly service interaction: the interaction of m1 with m2 is *wobbly* when a failure of m2 can trigger a failure of m1. Some *refactorings* for the described *smell* are to add a circuit breaker (applied in 42% of cases), use timeouts (applied in 22% of cases), add a bulkhead (applied in 20% of cases), or add a message broker (applied in 16% of cases).

Decentralization

One common *architectural smell* for decentralization is shared persistence. Three proposed solutions (which are better described in Figure 6.11) are split database (applied in 50% of cases), add a data manager that acts as a gateway between the services and the database (applied in 22% of cases), or merge services (applied in 9% of cases).

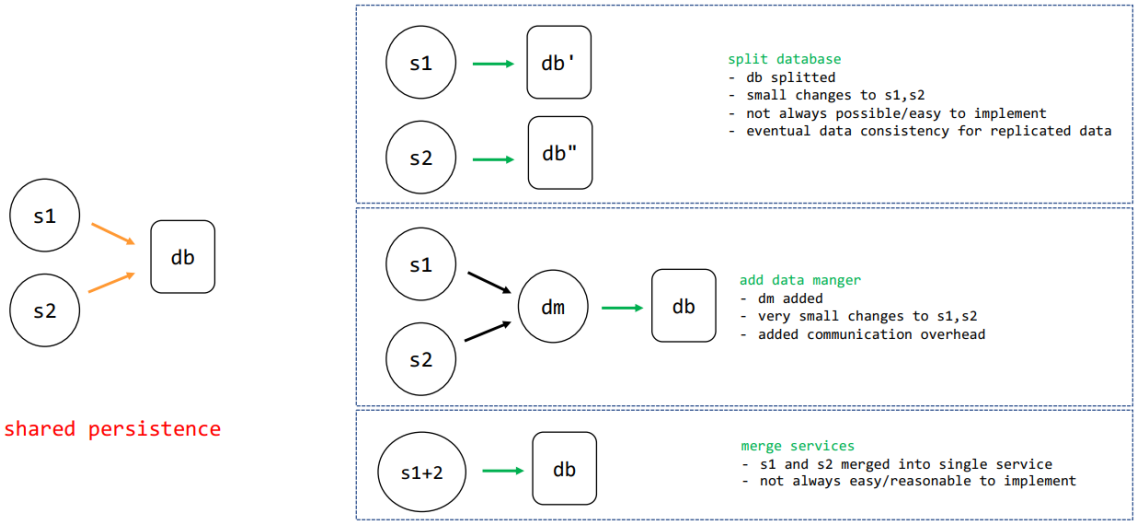


Figure 6.11: Solutions for shared persistence

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

6.4.2 A toolchain for microservices

Now that we know about these *architectural smells*, we want to identify and solve them. The first step is to create a graphical representation of our architecture, as shown in Figure 6.12. This is done to better understand the structure of the application.

These models also allow for *team-based views*, as each service is developed by a different team. This is particularly useful in cases where there may be smells between two or more microservices managed by different teams. It enables teams to detect these smells and make necessary arrangements to address and fix them.

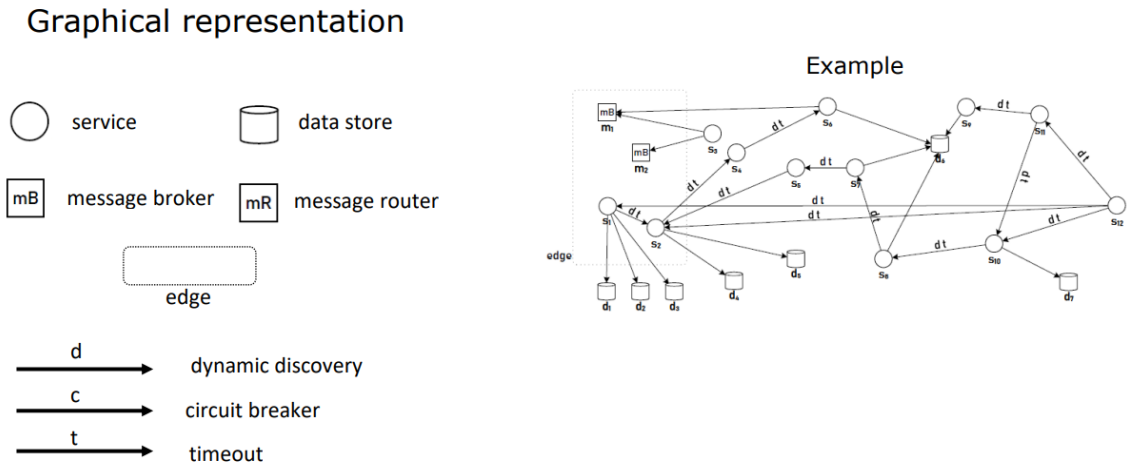


Figure 6.12: Modelling application architecture

There are tools that take these models as input and provide, as output, the smells present in the application. One developed by the University of Pisa is MicroFreshner[7]. It may be hard to obtain the modeled application architecture in the case of a large system, so tools have been created (such as MicroMiner and MicroTOM) to obtain those models by providing the K8s manifest and performing some dynamic analysis of the running application (noting that container orchestration does change application behavior). From Figure 6.13, we can see the toolchain used to fix microservices smells.

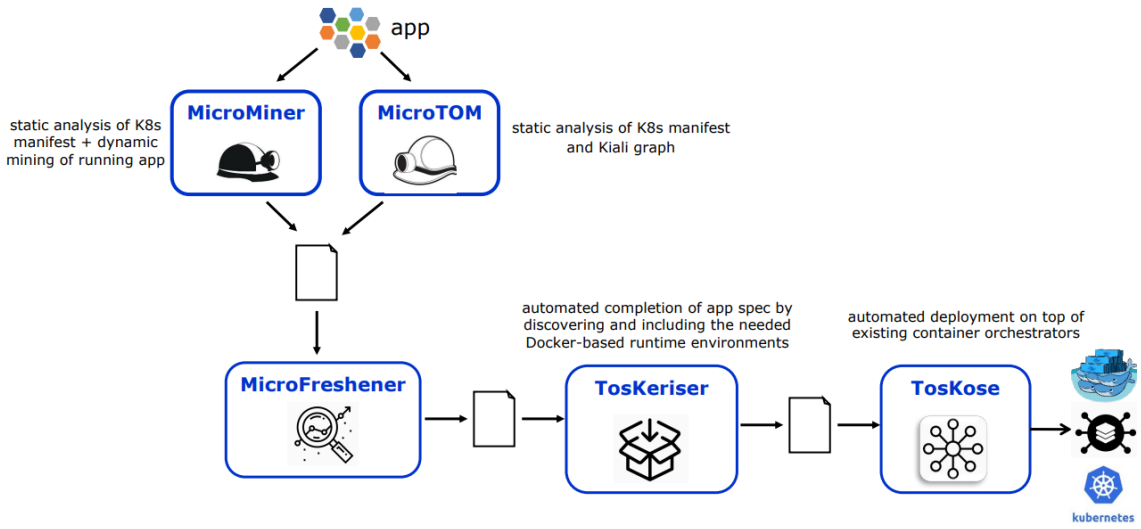


Figure 6.13: Toolchain for microservices

Once the smells have been analyzed and some fixes have been made inside MicroFreshner, it is possible to take the result of MicroFreshner and automatically deploy the fixed architecture on top of the existing container orchestrator.

[7] https://github.com/di-unipi-socc/microFreshener

Chapter 7

Security and Privacy

Software security is a high-priority concern for both product developers and users (since malicious attacks can cause losses to both). Figure 7.1 shows the main types of security threats, even though there are some attacks that combine those threats (e.g., a ransomware attack threatening the integrity of system data also threatens system availability).

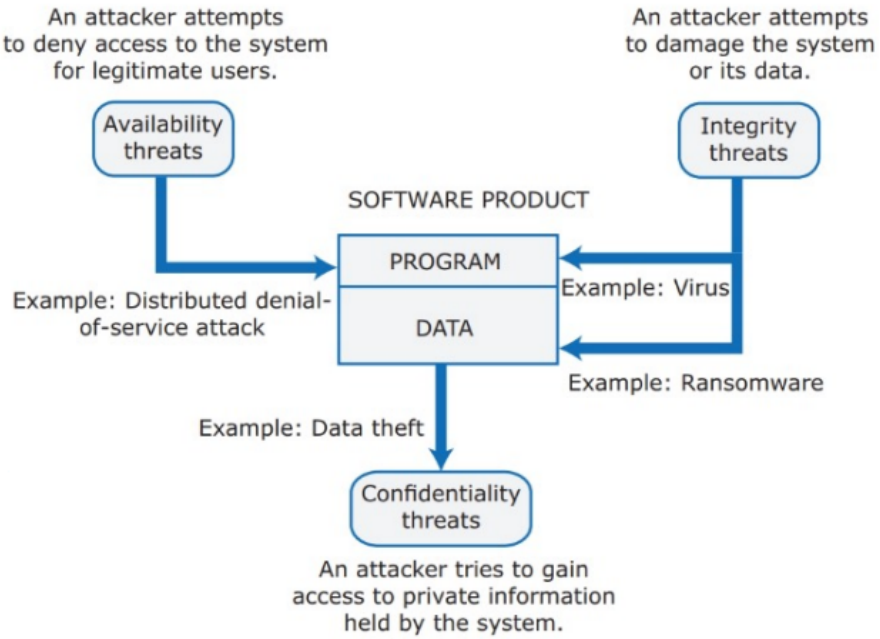


Figure 7.1: Main types of security threats

Security is a **system-wide issue**: application software depends on the operating system, web server, language run-time system, database, frameworks, tools, and so on. Attacks may target any level of the system infrastructure stack, starting from the network.

Some system management activities to maintain security are:

- **Authentication and authorization standards and procedures** to ensure that all users have strong authentication and properly set up access permissions.
- **System infrastructure management** to keep infrastructure software properly configured and to promptly apply security updates patching vulnerabilities.
- Regularly **monitoring attacks** to promptly detect them and trigger resistance strategies to minimize the effects of an attack.
- **Backup policies** to keep undamaged copies of program and data files that can be restored after an attack.

7.1 Attacks and defenses

7.1.1 Injection attacks

An injection attack happens when a malicious user uses a valid input field to input malicious code or database commands to damage the system. One of the most common classes of attacks is **buffer overflow attacks**, which occur when an attacker can carefully craft an input string that includes executable instructions and overwrites memory. If a function return address is overwritten, control can be transferred to malicious code (e.g., on operating systems/libraries written in C/C++, which do not check whether array assignments are within array bounds).

Another common class of attacks is **SQL injection attacks**, which occur when an attacker can do an injection attack when user input is part of an SQL command. An easy countermeasure is to check input validity.

7.1.2 Session hijacking attacks

In many applications, there's the concept of a *session*, which is a time period during which a user's authentication with a web app is valid. This is usually achieved with session cookies (tokens) sent from the server to the client in each HTTP request, so the user doesn't have to re-authenticate for subsequent system interactions (the session is closed when the user logs out or when the system "times out").

Session hijacking is performed by an attacker who wants to acquire a valid session cookie in order to **impersonate a legitimate user**. An attacker can obtain the session cookie with a cross-site scripting attack or with traffic monitoring (which is easy on unsecured Wi-Fi networks and unencrypted data). The attack can either be **active** (the attacker carries out user actions on a server) or **passive** (the attacker simply monitors client-server traffic looking for valuable information, such as passwords, credit card numbers, and so on).

Some trivial defenses for this type of attack are to *encrypt client-server network traffic* (using, for example, HTTPS), use *multifactor authentication* to require confirmation of new actions that may be damaging, and use relatively *short timeouts* on sessions.

7.1.3 Cross-site scripting attacks

Another form of injection attack is cross-site scripting attacks, where the attacker **adds malicious JavaScript code to the web page** returned from server to client. The malicious script is executed when the page is displayed in the victim's browser, with the intention of stealing customer information or directing customers to another website (cookies may be stolen, making a session hijacking attack possible).

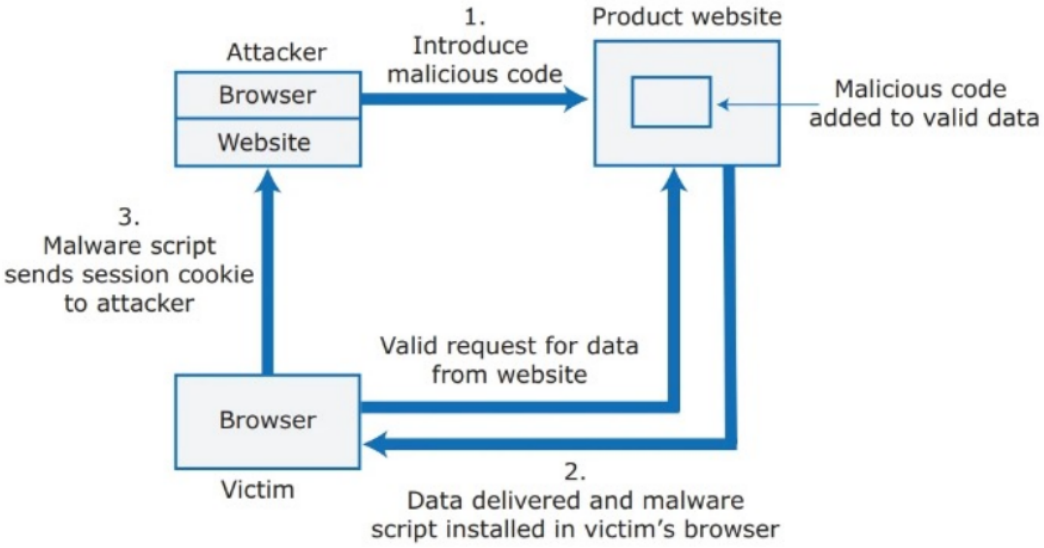


Figure 7.2: Cross-site scripting attack

Some defenses for this kind of injection are (form) input validation, checking input from the database before adding it to the generated page, and employing the HTML "encode" command (info added to the web page is not executable).

7.1.4 Denial-of-Service attacks

Denial-of-Service attacks (DoS attacks) are intended to **make system unavailable** for normal use. This is usually performed with the usage of distributed computers (that have usually been hijacked) that are sending hundreds of thousands of requests for service to a web application. Those attacks aims to boycott server provider or to demand ransom payment. For this attack there are specialised software for detecting and dropping incoming packets, which helps to prevent the flooding of requests. More contermeasures are temporary user lockouts (e.g. lockout user by repeatedly failing authentication with user email address as login name), and IP address tracking (to restrict lockout when failed logins come from unusual IP addresses).

7.1.5 Brute force attacks

An attacker has only some information (e.g., a valid login name, but not the password) and repeatedly tries to guess the missing information. An attacker can use a string generator to **generate all possible combinations** of letters and numbers. The defenses for this type of attack are to convince (which means force) users to set long passwords that are not in a dictionary and are not common words. Another security layer can be added with the use of two-factor authentication.

7.2 Authentication and Authorization

7.2.1 Authentication

The objective of authentication is ensuring that users of your system are who they claim to be. Some approaches to achieve authentication are:

- **Knowledge-based authentication** relies on users providing secret, personal information when registering. This type of authentication has weaknesses, such as *insecure passwords*, *phishing attacks* (users click on an email link pointing to a fake site that collects login and password), users using the *same password*, and users regularly *forgetting passwords* (a password recovery mechanism is needed, which means potential vulnerability if credentials have been stolen). Some ways to make this authentication method more secure are to force users to set strong passwords and to add personal questions for password recovery.

- **Possession-based authentication** relies on users having a physical device that can be linked to the authenticating system and that generates/displays information known to the authenticating system (e.g., the system sends a code to the user's phone number or a special-purpose device that generates one-time codes).

- **Attribute-based authentication** relies on a unique biometric attribute of the user (e.g., fingerprint, face).

In cases where a system wants to store confidential user information, multi-factor authentication (e.g., password, then a code received on a mobile phone) is the best practice in order to add another layer of security.

Usually, developers don't implement authentication from scratch, but they use available toolkits and libraries (e.g., OAuth[1]). Even when using already implemented libraries, there is still a lot of programming effort involved in order to implement a secure and reliable authentication system. This is why authentication systems are often outsourced with a **federated identity system**, which means using an external service for authentication (e.g., Google, Facebook, and so on), as shown in Figure 7.3. These external services for authentication are proven to be

[1] https://oauth.net/2/

secure and are likely to be much better than any independent system implemented by the product provider. On top of that, it allows the product provider to avoid maintaining their own database of passwords/secrets, as well as to gain additional user information (if the user agrees).

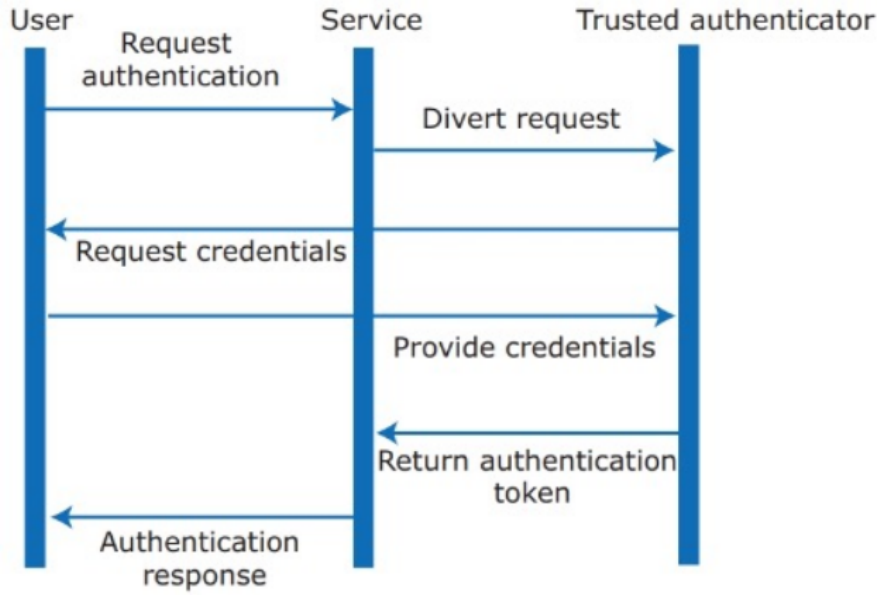


Figure 7.3: Federated identity sequence diagram

Mobile device authentication plays by different rules, since usually typing passwords on mobile keyboards is inconvenient. An alternative to the written password could be to install an *authentication token* on the mobile device, but this would lead to some weaknesses (if the device is stolen/lost, someone else can get access to the product). A safer alternative is to use *individual users' digital certificates* (issued by trusted providers).

7.2.2 Authorization

While *authentication* aims to ensure that the user is who they claim to be, *authorization* wants to control that the user can access resources. It is required to have **access control** for multiuser products, where the access control policy *must* reflect data protection rules that limit access to personal data (to prevent legal actions in case of a data breach).

One popular way to implement an access control policy is with **Access Control Lists** (ACLs): users are classified into groups (which dramatically reduces the size of ACLs), different groups can have different rights on different resources, and hierarchies of groups allow assigning rights to subgroups/individuals. We can see an example of an Access Control List in Figure 7.4.

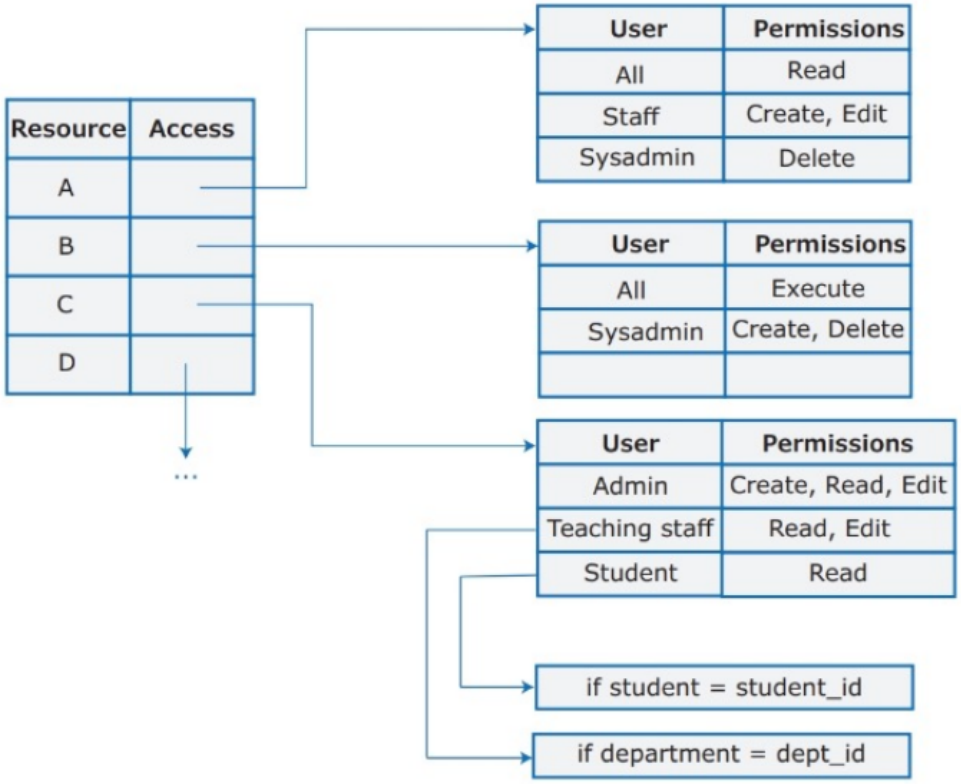


Figure 7.4: Example of Access Control Lists

7.3 Encryption

Encryption consists of making a text (typical data) unreadable by applying an algorithmic transformation to it. The text is encrypted with a secret key, travels through unsafe channels, and is then encrypted with the same or another key when it reaches its destination.

Modern encryption techniques are considered "practically uncrackable" using currently available technology, even if history tells us that apparently uncrackable encryption may become crackable when new technology becomes available (what if/when quantum computers become commercially available one day?).
Usually, encryption is divided into two categories:

- **Symmetric encryption**: older than the other method, used for centuries, but still used nowadays. Symmetric encryption uses the same key for encoding and decoding, but the sharing of the key is the weakness of this type of encryption.
- **Asymmetric encryption**: Asymmetric encryption uses different keys for encoding and decoding. This is the most secure encryption method, but also the most expensive one in terms of computation required for key generation. It's common practice to start the communication between two subjects using asymmetric encryption to share a symmetric key for the rest of the communication.

Transport Layer Security (TLS) is a widely adopted security protocol designed to facilitate privacy and data security for communications over the Internet. A primary use case of TLS is encrypting the communication between web applications and servers, such as web browsers loading a website. This protocol can be built on top of the HTTP protocol, generating **HTTPS**, which is a standard practice for websites. TLS is also used to verify the identity of web servers with the use of **digital certificates** sent from servers to clients. Digital certificates are issued by trusted identity verification services (CAs). An example of a typical client-server handshake is shown in Figure 7.5.

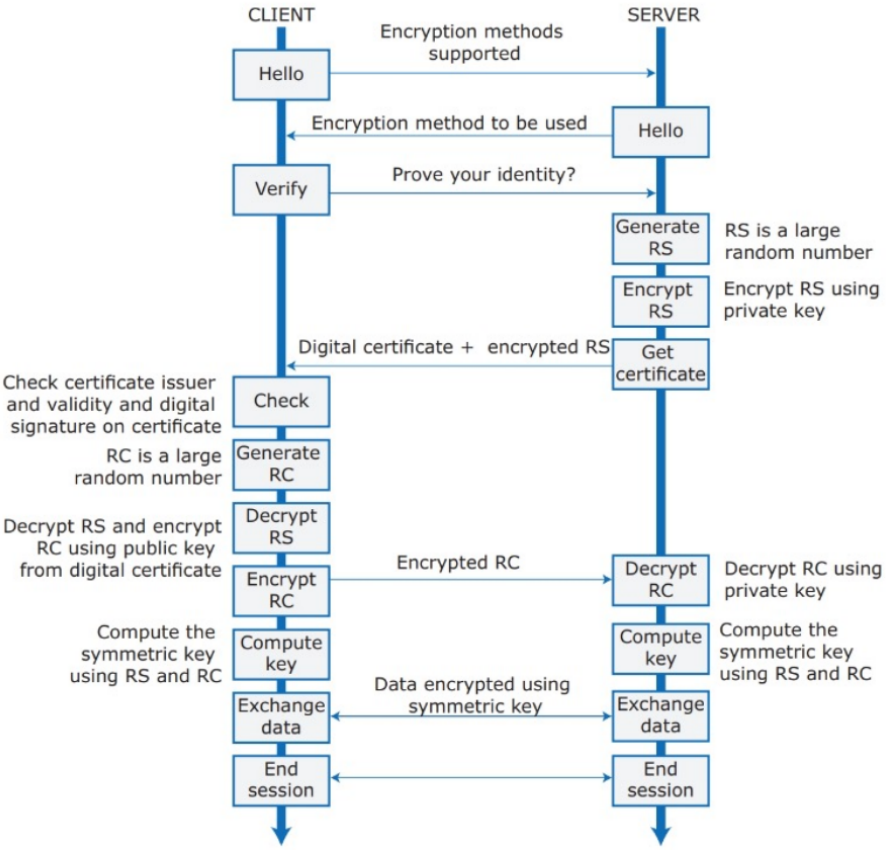


Figure 7.5: TLS client-server interaction to generate symmetric key for exchanging data

It's interesting to understand when user data should be encrypted, and to do so we'll categorize data into:

- **Data in transit** should always be encrypted since we want to achieve privacy in our communications.
- **Data at rest** (stored) should always be encrypted: in case of attacks, we don't want the attackers to obtain users' information in clear text.
- Encrypting and decrypting **data in use** (i.e., actively processed) slows down system response time.

Encryption of data is possible at four different levels in the system:

- **Application level**: The application decides what data should be encrypted and decrypts that data immediately before it is used. This not only causes performance issues (security doesn't come for free), but it also requires some sort of key management.
- **Database level**: The DBMS may encrypt the entire database when it's closed, with the database decrypted when it is reopened. Alternatively, individual tables or columns may be encrypted or decrypted.
- **Files level**: The operating system encrypts individual files when they are closed and decrypts them when they are reopened.
- **Media level**: The operating system encrypts disks when they are unmounted and decrypts these disks when they are remounted (useful for stolen/lost laptops).

All those levels require a key in order to encrypt and decrypt data. Data protection regulations may require that data copies are kept for years and stored securely, which means that if encryption keys get lost, encrypted data become permanently inaccessible! Keys should be changed periodically, and the database must maintain multiple timestamped versions of keys. To make life easier, **Key Management Systems** (KMS) are used to ensure that keys are securely generated, stored, and accessed by authorized users. Figure 7.6 shows how KMS are used.

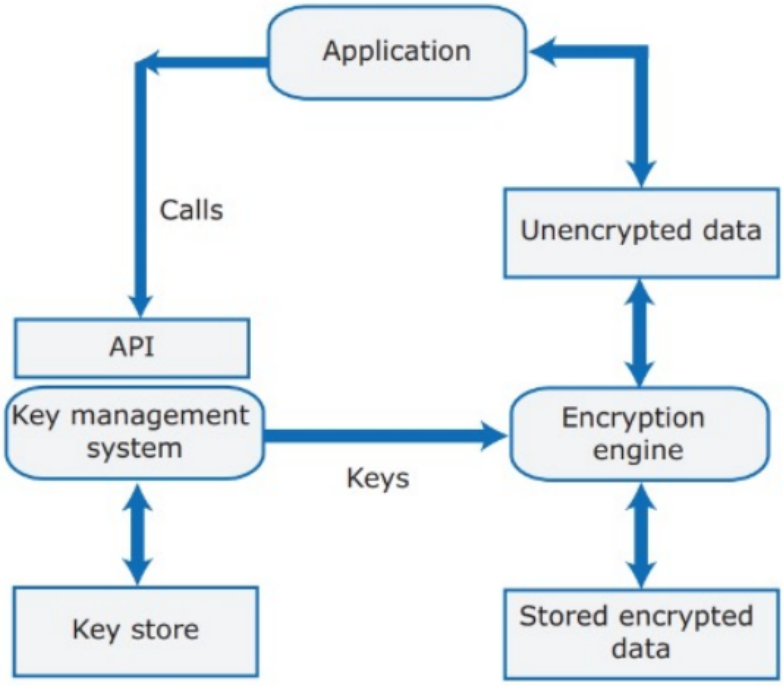


Figure 7.6: KMS usage schema

7.4 Privacy

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

Figure 7.7: Data protection laws

Some data protection principles are:

- **Awareness and Control**: Users of your product must be made aware of what data is collected when they are using your product and must have control over the personal information that you collect from them.
- **Purpose**: You must tell users why data is being collected, and you must not use that data for other purposes.
- **Consent**: You must always have the consent of a user before you disclose their data to other people.
- **Data lifetime**: You must not keep data for longer than you need to. If a user deletes an account, you must delete the personal data associated with that account.
- **Secure storage**: You must maintain data securely so that it cannot be tampered with or disclosed to unauthorized people.
- **Discovery and error correction**: You must allow users to find out what personal data you store. You must provide a way for users to correct errors in their personal data.
- **Location**: You must not store data in countries where weaker data protection laws apply unless there is an explicit agreement that some stronger data protection rules will be upheld.

There are also *business reasons* for paying attention to information privacy:

- If the system's conformance to privacy regulations does not match data protection regulations, the provider may be subject to legal actions or may not be able to sell their product.

- If a provider sells a business product, the provider's business customers may require privacy safeguards (not to be at risk with their users).
- Leakage or misuse of client information can damage the provider's reputation.

The information that software needs to collect depends on the functionality of the product and on the business model the provider uses. Here are more tips about data privacy:

- 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.

7.5 Security Smells

7.5.1 Challenges of Securing Microservices

This subsection will outline the general properties of attacks and security specific to microservices. The illustration will be made with a list of key points:

- *The broader the attack surface, the higher the risk*: Since microservices trade data between each other using inter-service communication via remote calls, this means that (potentially) there are a large number of entry points, which **broadens the attack surface**. The security of the application depends on the weakest link: only one weak entry point can compromise the whole application.
- *Distributed security screening affects performance*: Each microservice has to carry out independent security screening. This means that each service may need to connect to a remote security token service. Repeated, distributed security checks affect **performance**. A workaround for this problem is to *trust the network* and skip security checks at each microservice, but a real solution (and also an industry trend) is to use *zero-trust networking principles*, though overall performance must be considered.

- *Bootstrapping trust among microservices needs automation*: Service-to-service communication must take place on protected channels. Suppose those services are using *certificates*; each microservice must be provisioned with a certificate (and corresponding private key) to authenticate itself to another microservice during service-to-service interactions. The receiver must know how to validate the certificate of the calling microservice, so there is a need to **bootstrap trust** between microservices (including the need to revoke and rotate certificates). To manage large-scale deployments of hundreds of microservices, *automation* is needed.

- *Tracing requests spanning multiple microservices is challenging*: Logs can be aggregated to produce *metrics* that reflect system state (e.g., average invalid access requests per hour) and may trigger alerts. Traces help track a request from the point where it enters the system to the point where it leaves. Unlike monolithic applications, a request in a microservices deployment may span multiple microservices, making **correlating requests among microservices challenging**.

- *Containers complicate credentials/policies handling*: Containers are *immutable servers* (great for simplifying deployment and achieving horizontal scalability), so they don't change state after spin-up. Since each service needs to maintain a dynamic list of allowed clients and a dynamic set of access-control policies, the only way to modify these lists periodically is to keep credentials in the container filesystem and **inject them at boot time**.

- *Distribution makes sharing user context harder*: When a request starts to move inside the system, there is a lot of information we want to associate with the request (e.g., whether it's a premium or non-premium request). The challenge is to *build trust between microservices* so that the receiving microservice accepts the user context passed from the calling microservice (a security attack can occur if user information is manipulated). A popular solution is **JSON Web Token** (JWT), a security token used to share user context among microservices.

- *Security responsibilities distributed among different teams*: Different independent teams can use **different technology stacks** (such as security best practices and various tools for static and dynamic security testing). Security responsibilities are distributed across different teams. Organizations often adopt a hybrid approach with a centralized security team and security experts within the teams.

7.5.2 Smells and Refactoring

A **security smell** is a commonly used architectural decision that indicates possible security violations in microservice-based applications. Once the microservice principles are defined, how can security smells that affect security properties of microservices be detected and resolved via refactoring?

This section will show, based on a *multivocal review* (which uses both scientific articles and technical websites), which are the most recognized *security smells* for microservices and the security *refactorings* to resolve them.
The **security properties** that will be considered are:

- Confidentiality: The degree to which a product or system ensures that data is accessible only to those authorized to have access.
- Integrity: The degree to which a system, product, or component prevents unauthorized modification of computer programs or data.
- Authenticity: The degree to which the identity of a subject or resource can be proven to be the one claimed.

Since some *security smells* violate multiple *security properties*, the following list will show all the security smells, their related security properties, and how to refactor them:

- Insufficient access control: Some services of an application don't implement access control, leading to a *confused deputy problem* (an attacker obtaining data they shouldn't access), causing potential violations of the confidentiality of data (and business functions). Client permissions need to be verified at request time without introducing extra latency and contention with frequent calls to a centralized service. One way to refactor this smell is to use OAuth 2.0, a token-based security framework for delegated access control.
- Publicly accessible microservices: Some microservices are directly accessible by external clients. Each such microservice must check authentication and authorization for each request (increasing the cost of application maintenance), but this also increases the exposure of credentials, which could lead to confidentiality violations. The most cited refactoring is the implementation of an API gateway, which can (from behind a firewall) enforce authentication, authorization, throttling, and message content validation.
- Unnecessary privileges to microservices: Sometimes microservices are granted unnecessary access levels, permissions, or functionalities that are not needed to deliver their business functions. This mostly arises because programmers copy and paste similar patterns, leading to unnecessary access. When resources are

unnecessarily exposed, the attack surface against confidentiality and integrity is increased. The refactoring is to follow the Least Privilege Principle. [2]

- "Home-made" crypto code: The usage of "home-made" cryptography code can cause confidentiality, integrity, and authenticity issues, which can be worse than no encryption at all (because of the false sense of security). The solution is to use encryption libraries that are heavily tested by the community and regularly reviewed and patched (avoid experimental encryption algorithms).

- Non-encrypted data exposure: A microservice-based application accidentally exposes sensitive data (e.g., data stored without encryption or protection that has vulnerabilities), allowing intruders to access or modify data, including credentials, leading to violations of confidentiality, integrity, and authenticity. The refactor is to encrypt all sensitive data at rest: All sensitive data should always remain encrypted and only be decrypted when needed. Most DBMSs support automatic encryption, but encryption can also be applied at the application level, OS level, and cache level. It's important to note that encryption is resource-consuming, so critical data should be identified, and encryption should be applied only when needed.

- Hardcoded secrets: Hardcoded secrets (e.g., API keys, client secrets, credentials) should never be stored in environment variables because they could be accidentally exposed (e.g., exception handlers may send info to a logging platform), leading to violations of confidentiality, authenticity, and integrity. The solution is to encrypt secrets at rest, avoid storing credentials alongside applications or in source code repositories, and refrain from using environment variables to pass secrets.

- Non-secured service-to-service communications: Microservices interacting without secure communication channels could expose data to man-in-the-middle, eavesdropping, and tampering attacks, leading to potential violations of confidentiality, integrity, and authenticity. A widely accepted solution is to use mutual Transport Layer Security, which encrypts data in transit and ensures its integrity and confidentiality. Mutual TLS also allows a microservice to legitimately identify the microservice it is communicating with (*mutual authentication*).

- Unauthenticated traffic: It's crucial that microservices authenticate one another (especially when user context is passed). If traffic is not authenticated, microservices are exposed to security attacks like data tampering, denial of service, or privilege elevation, leading to authenticity issues. As mentioned before, Transport Layer Security allows mutual authentication. It's also possible to use

[2] **Least Privilege Principle**: *Allow running code only the permissions needed to complete the required tasks and no more.*

OpenID Connect, which uses JWT containing authenticated user information, where microservices verify user identity with authorization servers.

- Multiple user authentication: As software engineers, it's tempting to make users authenticate from different points. Each access point constitutes a potential attack vector for intruders to authenticate as end users, leading to authenticity issues. The most suggested approach is to use Single Sign-On (single entry point), which facilitates log storage and auditing. Single sign-on can be achieved by employing an API gateway and OpenID Connect (to share user contexts).
- Centralized authorization: Authorization can be enforced at the edge of the application (API gateway) and/or by each microservice. If authorization is only handled at the edge, the "central" authorization point becomes a bottleneck, reducing performance and efficiency. There is also the possibility of a "confused deputy problem": microservices trust the gateway based on its mere identity, leading to potential violations of authenticity. The solution is to enact a decentralized authorization approach by transmitting an access token (e.g., JWT) together with each request to a microservice and granting access to the caller only if a known token is passed.

7.5.3 Refactor or Not Refactor?

This question should be addressed on a case-by-case basis: **solving a smell may affect other properties**! For example, centralized and decentralized authorization: while switching to decentralized authorization follows certain declared *design principles* and ensures *security properties*, centralized authorization is easier to *maintain* and offers *better performance*.

Modeling soft goal interdependencies as **graphs** can help with visualization and (automated) trade-off analysis. An example is shown in Figure 7.8.

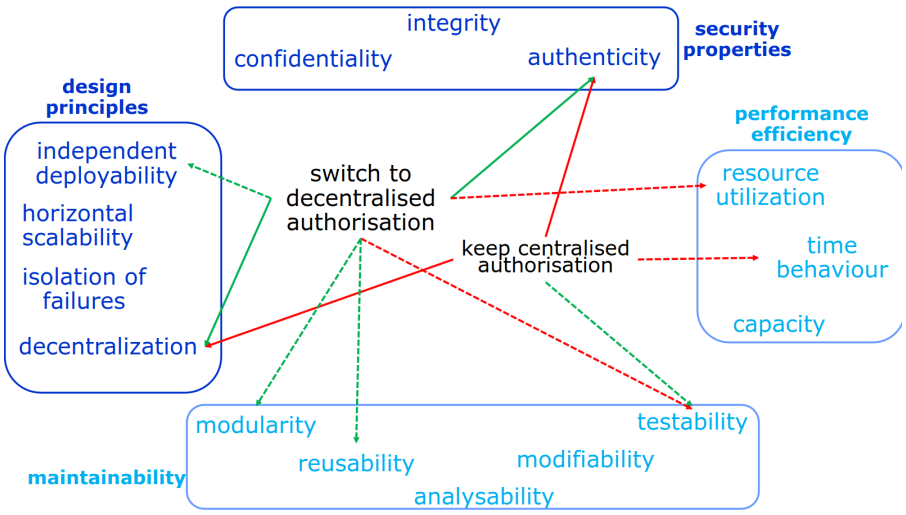


Figure 7.8: Comparison between centralized authorization and decentralized authorization

Chapter 8

Business Process Modeling

Business process management is the systematic method of examining an organization's existing business processes and implementing improvements to make its workflow more effective and efficient (IBM made a fortune with business process management). A **business process** is a set of business activities that represent the required steps to achieve a business objective. A **business process model** (Figure 8.1) consists of a set of activity models and execution constraints among them. A **business process instance** represents a concrete case in the operational business of a company, consisting of activity instances.

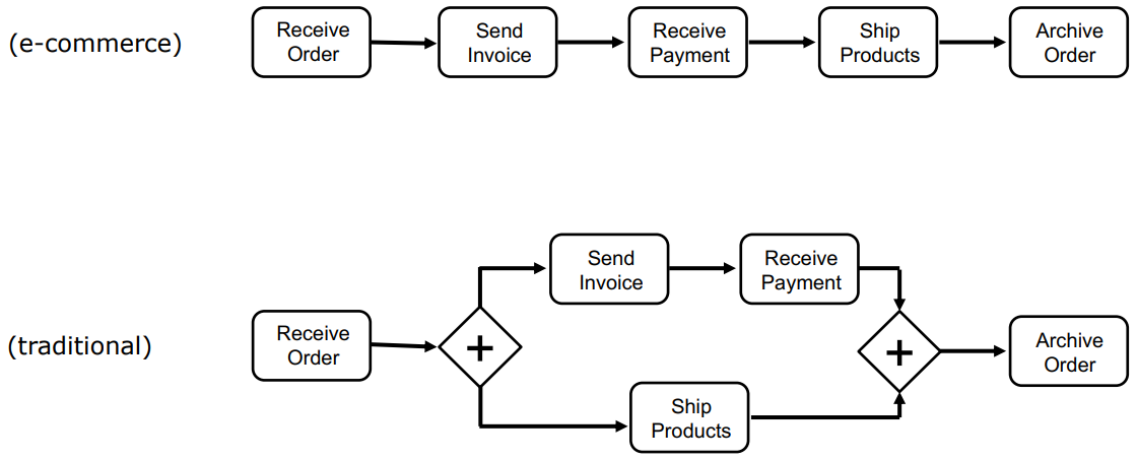


Figure 8.1: Example of a business process model

8.1 BPMN

Business Process Model and Notation (BPMN) is the graphical notation for business process modeling, as shown in Figure 8.2.

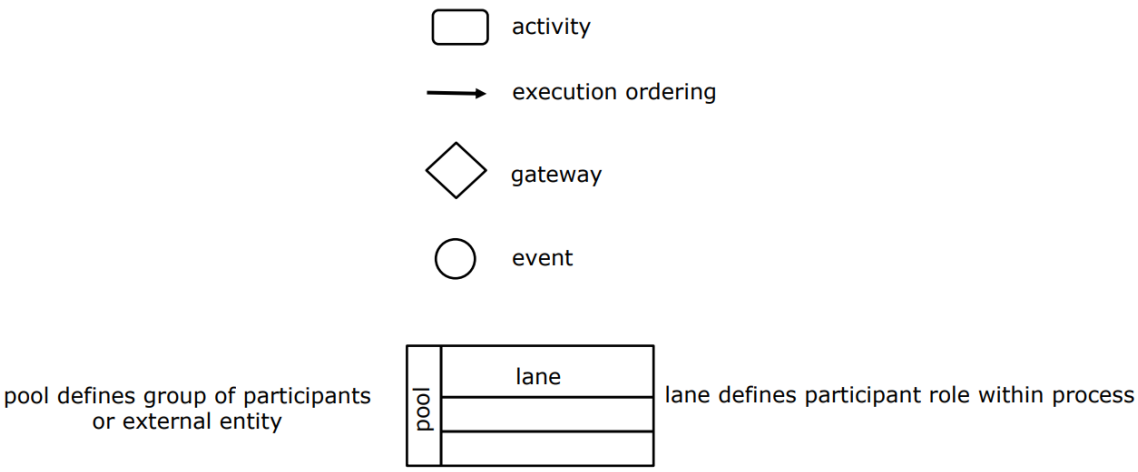


Figure 8.2: Graphical notation for business process modeling

An example of a complete BPMN is shown in Figure 8.3. This example demonstrates how **parallel gateway**, **exclusive gateway**, and **inclusive gateway** work.

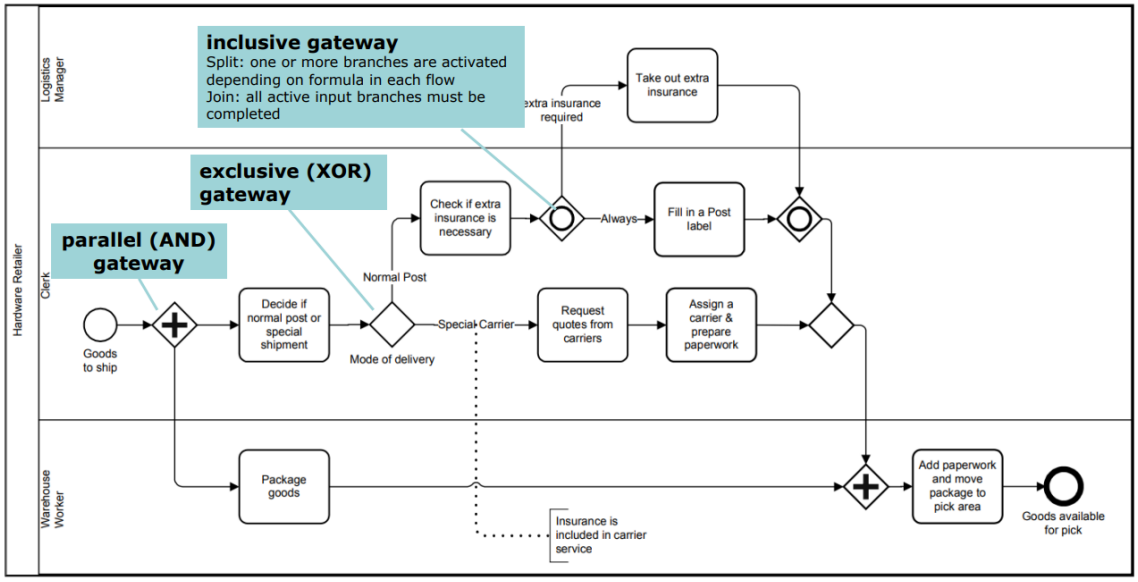


Figure 8.3: Shipment process of hardware retailer example

Another example is shown in Figure 8.4, where we can see how multiple processes can interact with each other using **message flow**. As we can see, there is a main process and a secondary one. When a **terminate event** is reached, all activities in the process immediately end. In the main process, we also see a **timer event** and an **event-based gateway**.

The last example in Figure 8.5 shows the **sub-process** notation and the **attached event** notation (both error → interrupting and escalation → non-interrupting).

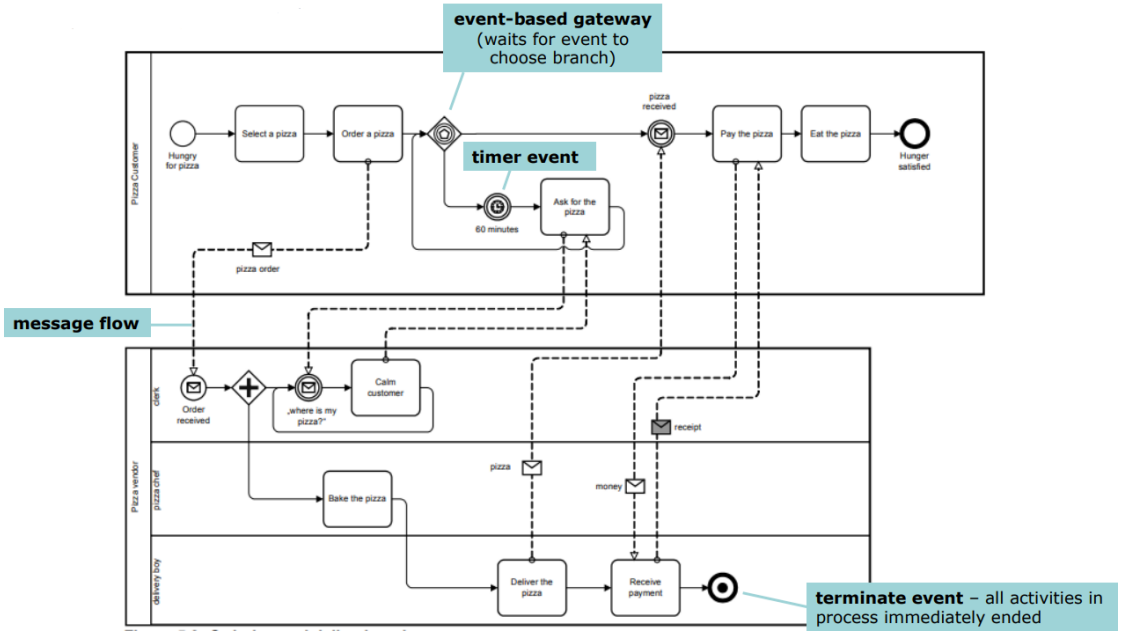


Figure 8.4: B2B (pizza) collaboration example

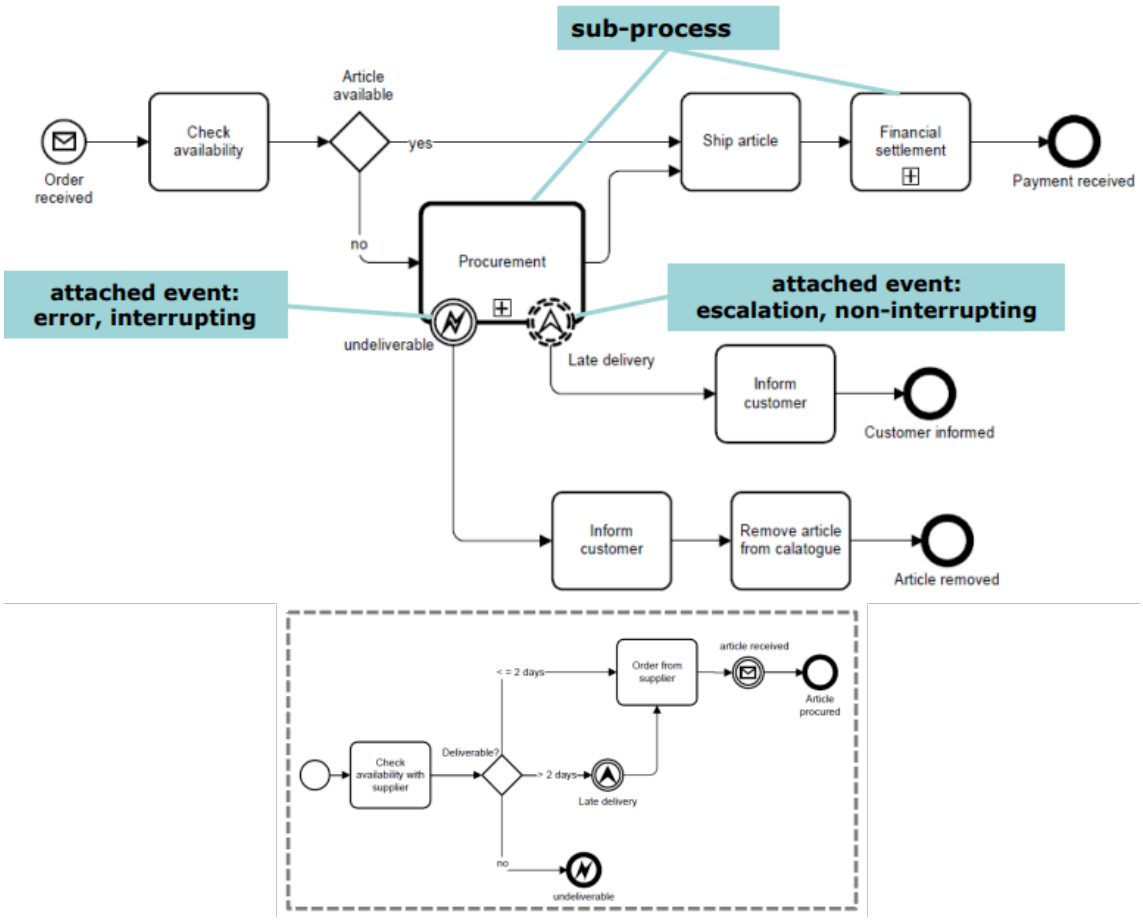


Figure 8.5: Order fulfillment and procurement example

The ability to **prove properties of business process models is crucial** in business process management.

8.2 Workflow Nets

Workflow nets are an extension of **Petri nets**, and they are one of the best-known techniques for specifying business processes in a formal and abstract way. The graphical representation of Petri nets *eases communication* between different stakeholders. **Process properties can be formally analyzed**, with the support of various tools that are available.

Figure 8.6 shows how Petri nets ease the reading of a business process model, translating the example shown in Figure 8.1.

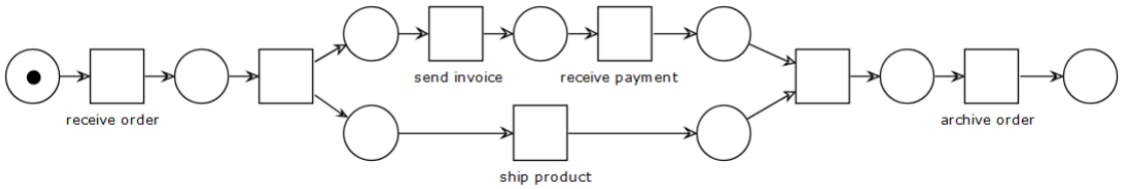
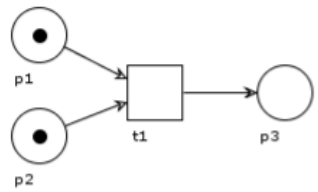
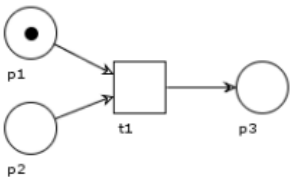


Figure 8.6: Petri net of the previous BPM example (Figure 8.1)

Petri nets consist of **transitions** (squares), **places** (circles), and direct **arcs** connecting places and transitions. Transitions model activities, and places and arcs model execution constraints. System dynamics are represented by **tokens**, whose distribution over the places determines the state of the modeled system. A transition *can fire* if there is a token in each of its input places, as shown in Figure 8.7.



t1 can fire



t1 cannot fire

Figure 8.7: Petri nets input rule

If a transition *fires*, one token is removed from each input place and one token is added to each output place, as shown in Figure 8.8.

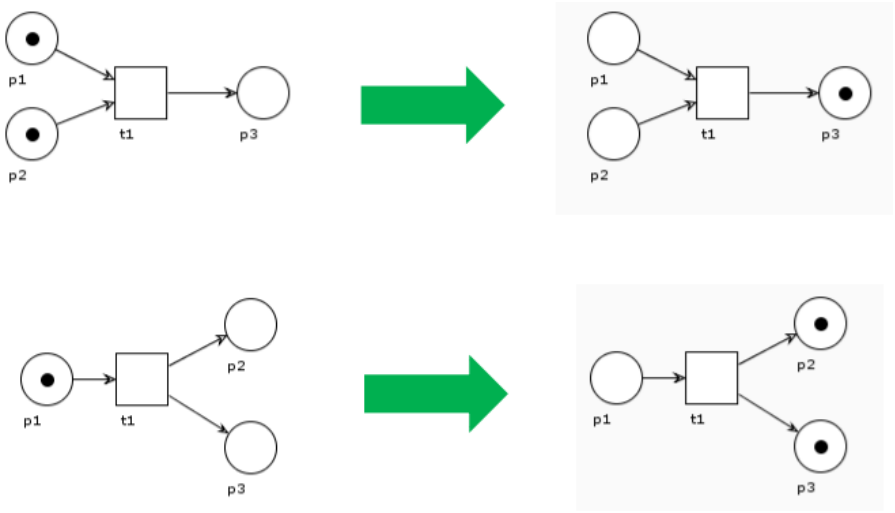


Figure 8.8: Petri nets output rule

The concept of **workflow nets** enhances Petri nets with notations that simplify the representation of business processes. Like Petri nets, workflow nets focus on the control flow behavior of a process: **transitions** represent activities, **places** represent conditions, and **tokens** represent process instances. Figure 8.9 shows some composition patterns that can be used with Petri nets.
A Petri net is a ***workflow net*** *if and only if*:

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

A workflow net is ***sound*** *if and only if*:

1. Every net execution starting from the initial state (one token in the **source place**, no tokens elsewhere) eventually leads to the final state (one token in the **sink place**, no tokens elsewhere).
2. Every transition occurs in at least one net execution.

Notice that workflow nets abstract from data, meaning that data-dependent choices are modeled as "blind" choices. As a consequence, the analysis may consider "more branches than needed." This means that a *not sound workflow net* **must be interpreted only as a "warning"** of possible problems that may arise at runtime (e.g., the analysis of a branch that will never be executed may determine that the net is not sound, even if the application will never execute such a branch).

Conversely, the analysis of an iteration may fail to determine that the application will never terminate: a *sound workflow net* **cannot be interpreted as a guarantee** that the application will always terminate its execution.

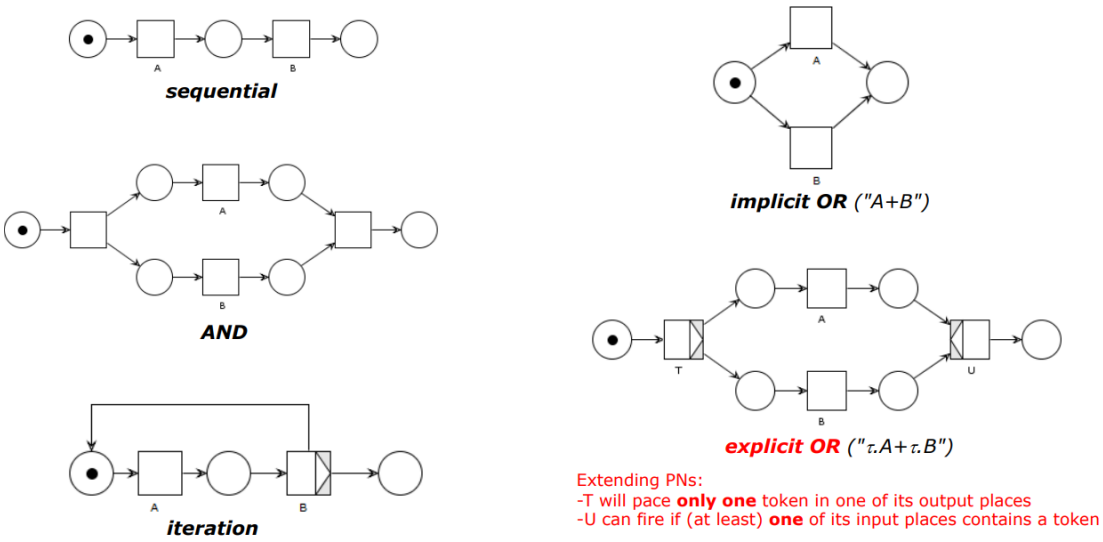


Figure 8.9: Petri nets composition patterns

How can we formally (and automatically) establish whether a net is sound? We first need to define a couple of properties:

- A Petri net (PN, M) is ***live*** if and only if for every reachable state M' and every transition t, there is a state M'' reachable from M' where t is enabled.

- A Petri net (PN, M) is ***bounded*** if and only if for every place p there is an $n \in \mathbb{N}$ such that for each reachable state M', the number of tokens in p in M' is less than n.

Theorem: A workflow net N is sound if and only if $(\check{N}, \{i\})$ is live and bounded, where $\check{N}$ is N extended with a transition from the sink place o to the source place i

Chapter 9

Testing

Testing is the systematic process of executing a program using data that simulates user input, aiming to ensure that the software functions as intended and meets the specified requirements.

At its core, testing serves as a methodical exploration to observe the program's behavior, providing valuable insights into its functionality. The fundamental principle guiding testing is to compare the expected behavior of the software with its actual behavior during execution. To put it simply:

- If the observed behavior aligns with what is expected, the test is deemed successful.
- If the observed behavior deviates from expectations, the test is considered unsuccessful or failed.

Some types of testing are:

- **Functional testing**: Functional testing assesses whether a software application performs its functions as expected. It involves evaluating the individual features and functionalities of the software to ensure that they conform to the specified requirements. This type of testing focuses on inputs, outputs, and the *overall behavior of the system* under various conditions.
- **User testing**: User testing, also known as usability testing, focuses on evaluating the software from an end-user perspective. It aims to validate that the application is user-friendly, intuitive, and *meets the needs of its intended users*. This type of testing often involves real users interacting with the system to identify any usability issues, design flaws, or areas for improvement in the user experience.
- **Performance and load testing**: Performance testing assesses the responsiveness, speed, and *overall efficiency* of a software application under varying conditions. Load testing, a subset of performance testing, evaluates how the

system behaves when subjected to simulated or expected levels of concurrent user activity. These tests help identify bottlenecks, measure system scalability, and ensure the software performs optimally under *different workloads*.

- **Security testing**: Security testing focuses on identifying vulnerabilities and weaknesses within a software system to *safeguard it from potential threats and unauthorized access*. This type of testing encompasses various techniques to assess the application's resistance to attacks, data breaches, and unauthorized manipulation. Security testing helps ensure that sensitive information is protected and that the software adheres to established security standards and practices.

Note that program testing can be used to show the presence of bugs, but never to show their absence: software testing will not substitute *software verification*. The rest of the chapter will discuss **Functional testing** and **Security testing** in more detail.

9.1 Functional testing

Functional testing is a comprehensive set of program tests that ensure all the code is executed at least once to test the overall behavior of the system. Testing should start on the day developers begin writing code. The *develop and test* cycle is simplified by automated tests. When we say that functional testing is a **staged activity**, it means that the testing process is organized and conducted in distinct stages or phases (see Figure 9.1). It's important to note that these stages are not strictly linear and often overlap. Additionally, **regression testing** is an ongoing process that may be performed at each stage to ensure that new changes or features do not introduce defects into previously tested functionality. The division into these stages helps systematically identify and address issues at different levels of the software development cycle.

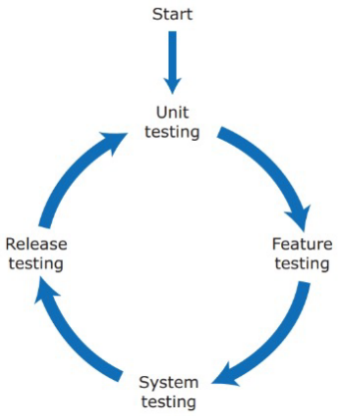


Figure 9.1: Functional testing life cycle

9.1.1 Unit testing

The goal of unit testing is to test program units (e.g., function, method) in **isolation**. The following is the unit test principle: *If a program unit behaves as expected for a set of inputs that have some shared characteristics, it will behave the same way for a larger set whose members share these characteristics* [1].

[1]Example: if a program behaves correctly on the input set 1, 5, 17, 45, 99, it is possible to conclude that it will also process all other integers in the range 1 to 99 correctly.

It's important to identify **equivalence partitions**: sets of inputs that will be treated the same in a code fragment, and then test the program using several inputs from each equivalence partition. Programmers usually make mistakes when defining boundaries, which is why it's recommended to use those boundaries as input for that partition. Some unit test guidelines are:

- *Test edge cases*: If your partition has upper and lower bounds (e.g., length of strings, numbers, etc.), choose inputs at the edges of the range.
- *Force errors*: Choose test inputs that force the system to generate all error messages. Choose test inputs that should generate invalid outputs.
- *Fill buffers*: Choose test inputs that cause all input buffers to overflow.
- *Repeat yourself*: Repeat the same test input or series of inputs several times.
- *Overflow and underflow*: If your program performs numeric calculations, choose test inputs that can cause it to calculate very large or very small numbers.
- *Don't forget null and zero*: If your program uses pointers or strings, always test with null pointers and strings. If you use sequences, test with an empty sequence. For numeric inputs, always test with zero.
- *Keep count*: When dealing with lists and list transformations, keep count of the number of elements in each list and check that these are consistent after each transformation.
- *One is different*: If your program deals with sequences, always test with sequences that have a single value.

9.1.2 Feature testing

Knowing that a *product feature* implements some useful user functionality, the goal of feature testing is to test that a functionality *is implemented as expected* and *meets the real needs of users*. Since features are normally implemented by multiple interacting program units, there are two types of tests:

- **Interaction tests**: Testing interactions between units (developed usually by different developers). This type of test can also reveal bugs in program units that were not exposed by unit testing.
- **Usefulness tests**: Testing that a feature implements what users are likely to want. The *Product manager* should be involved in designing usefulness tests.

To generate feature tests, developers must **create them from scenarios or user stories** (e.g., from user story to test: User registration: As a user, I want to be able to log in without creating a new account so that I don't have to remember another

login ID and password. → Initial login screen: Test that the screen displaying a request for Google account credentials is correctly displayed when a user clicks on the "Sign in with Google" link. Test that the login is completed if the user is already logged in to Google.).

9.1.3 System Testing

The goal of system testing is to test the system as a whole, in order to:

- Discover unexpected/unwanted interactions between the features.
- Discover if system features work together effectively to support what users really want to do.
- Ensure the system operates as expected in the different environments where it will be used.
- Test responsiveness, throughput, security, and other quality attributes.

One piece of advice for generating system testing is to use a set of scenarios and user stories to identify users' end-to-end pathways since we want to test data flow across different modules.

9.1.4 Release Testing

The goal of release testing is to test a system that is intended for release to customers. Release testing tests the system in its real operational environment (rather than in a test environment). The aim is to decide if **the system is good enough to release**, not to detect bugs in the system. This type of testing is necessary because preparing a system for release involves packaging the system for deployment, installing required software and libraries, configuring parameters (and mistakes can be made in that process). If the software is deployed on the cloud, an automated continuous release process can be used.

9.2 Security Testing

Objectives of security testing are to **find vulnerabilities** that an attacker may exploit, and to **provide convincing evidence** that a system is **sufficiently secure**. Finding vulnerabilities is harder than finding bugs, as developers must test for something the software should NOT do (potentially an infinite number of tests). Normal functional tests may not reveal vulnerabilities, and the software stack (OS, libraries, databases, and so on) on which a product depends may contain vulnerabilities.

Comprehensive security testing requires specialist knowledge (e.g., involve external specialists for penetration testing, which is expensive). Usually, companies adopt a **risk-based approach** by identifying the main security risks to a product and developing tests to demonstrate that the product protects itself from these risks.

It is possible to automate some of these tests, but others inevitably require manual checking of behavior and files.
Here are some examples of security risks:

- An unauthorized attacker gains access to a system using authorized credentials.
- An authorized individual accesses resources that are forbidden to that person.
- The authentication system fails to detect an unauthorized attacker.
- An attacker gains access to the database using an SQL injection attack.
- Improper management of HTTP sessions.
- HTTP session cookies are revealed to an attacker.
- Confidential data are unencrypted.
- Encryption keys are leaked to a potential attacker.

When testing security, developers need to think like an attacker rather than a normal end-user, deliberately trying to do the wrong thing and repeating actions multiple times.

9.3 Test Automation

Automated testing is widely used in product development companies, as manual system testing is tedious and error-prone. Executable tests check that software returns the expected result for input data, as shown in Figure 9.2.
A good practice is to structure automated tests into three parts:

1. *Arrange*: Set up the system to run the test (define test parameters).
2. *Action*: Call the unit that is being tested with the test parameters.
3. *Assert*: Assert what should hold if the test is executed successfully.

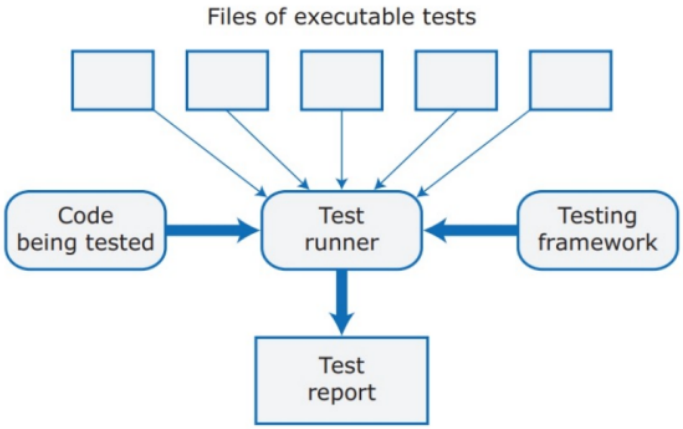


Figure 9.2: Test runner scheme

Testing frameworks are available for all widely used programming languages, so developers are free to use their code. Test code used for testing *can include bugs*. Good practices to reduce the chances of test errors are to make tests as simple as possible and review all tests along with the code they test.

Unit tests are the easiest to automate. Good unit tests reduce (but do not eliminate) the need for feature tests, which is beneficial because GUI-based testing is expensive to automate (see an example in Figure 9.3), while API-based testing is preferable. It is necessary to perform multiple assertions to check that the feature executed as expected.

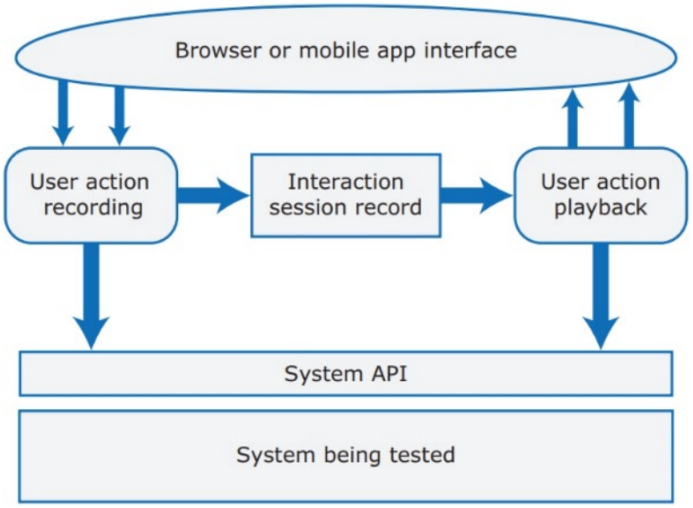


Figure 9.3: GUI-based testing cycle

9.4 Test-Driven Development

There are some Agile frameworks, such as *Extreme Programming*, that are test-driven, meaning that developers must first write executable tests and then write the code to pass the tests. Figure 9.4 shows how test-driven development can be incorporated within the development cycle.

Let's analyze the pros of this technique:

- Systematic approach: Tests are clearly linked to code sections, so there are no untested sections.
- Tests help in understanding program code.
- Simplified, incremental debugging.
- Simpler code (this last point is arguable).

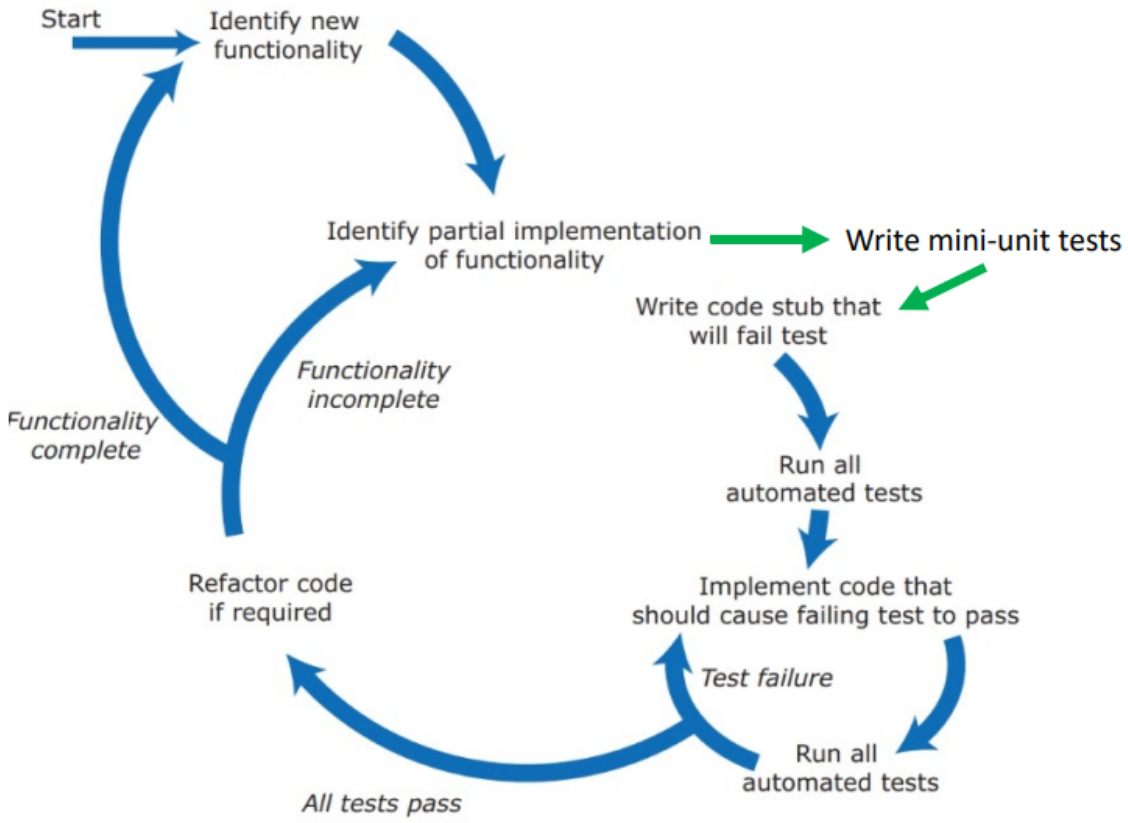


Figure 9.4: Test-driven development cycle

Let's also analyze the cons of this technique:

- Difficult to apply TDD[2] to system testing.
- TDD discourages radical program changes.
- TDD leads developers to focus on the tests rather than on the problem they are trying to solve.
- TDD leads developers to think too much about implementation details rather than overall program structure.
- Hard to write "bad data" tests.

[2]TDD: Test-driven development

9.5 Code Reviews

Testing has some limitations:

- Developers test code against their understanding of what that code should do. If you have misunderstood the purpose of the code, then this will be reflected in both the code and the tests.
- Testing may not provide coverage of all the code you have written (TDD shifts the problem to code incompleteness).
- Testing does not really provide insights into other attributes of a program (e.g., readability, structure, evolvability).

That's why **code reviews** complement testing. Figure 9.5 shows how a **Reviewer** works.

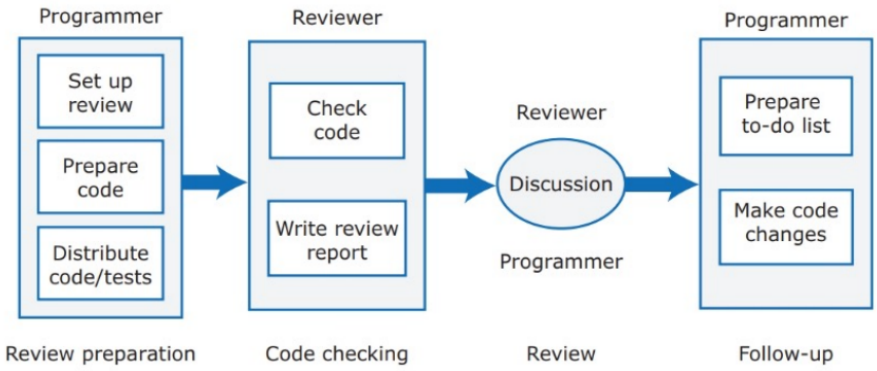


Figure 9.5: Code reviews cycle

Often, a single code reviewer is used (from the same DevOps team or otherwise). The reviewer can also comment on the readability and understandability of the code. Review sessions should focus on 200-400 lines of code and can be triggered by commits to shared repositories.

A checklist is often used by the reviewer to check common (or language-specific) points about the new section. Some examples are:

- (General) Are meaningful variable and function names used?
- (General) Have all data errors been considered and tests developed for these?
- (General) Are all exceptions explicitly handled?
- (Python) Are default function parameters used?
- (Python) Are types used consistently?
- (Python) Is the indentation level correct?

Chapter 10

DevOps

In Project-based Software Engineering, there was a *development team* responsible for developing (design, requirements, specification, prototypes, testing, integrating everything, and finally delivering tested software ready for release), and an *operations team* that was responsible for deploying and maintaining the system (by providing user support and possibly making software changes).

The issues with this model were communication delays between teams, separate teams using different tools, having different skills, or often not understanding each other's problems. Additionally, days were required to fix urgent bugs or security vulnerabilities (because the operations team didn't develop the software in the first place).
Three factors enabled a change:

- Agile software engineering **reduced software development time**, so the traditional release process became a bottleneck.
- Amazon re-engineered its software into (micro)services, **assigning both service development and service support to the same team**.
- **SaaS release of software** became possible on public/private clouds.

DevOps (**Dev**elopment + **Op**eration**s**) integrates development, deployment, and support into a single team. The DevOps **principles** are:

- **Everyone is responsible for everything**: all team members share responsibility for developing, delivering, and supporting the software.
- **Everything that can be automated should be automated**: All/most activities involved in testing, deployment, and support should be automated.
- **Measure first, change later**: DevOps should be driven by measuring collected data about the system and its operation.

Some of DevOps' **benefits** are:

- **Faster deployment** (the main benefit): a dramatic reduction in human communication delays leads to faster deployment to production (from days/weeks to hours).
- **Reduced risk**: small functionality increments in each release reduce the chance of feature interactions and system failures/outages.
- **Faster repair**: no need to discover which team is responsible for fixing a problem and wait for them to resolve it.
- **More productive teams**: DevOps teams are more productive than teams involved in separate activities.

Creating a **DevOps team** means bringing together a variety of skill sets, including software engineering, UX design, security engineering, infrastructure engineering, customer interaction, and more. The success of a DevOps team is based on a **culture of mutual respect and sharing** (everyone on the team should be involved in Scrums and other team meetings, and team members are encouraged to share their expertise with others and learn new skills) and the **support from developers for the software they have developed** (if a service fails on the weekend, the developer is responsible for getting it up and running again. If a developer is unavailable, other team members take over the task. DevOps teams focus on fixing failures as quickly as possible rather than blaming team member(s)).

10.1 Code management

During the development of a software product, tens of thousands of lines of code are written, and automated tests are created, organized into hundreds of files. Dozens of libraries are used, and different programs are needed to create/run the code, making it impossible to keep track of changes made to the software without automated support.

A **code management system** is software that supports practices to manage an evolving codebase. It is important that the *code management system* ensures that changes made by different developers do not interfere with each other and that it helps create different product versions. A *code management system* should also simplify the creation of an executable product from source code files and the running of automated tests. Figure 10.1 shows the components of a code management system: at the top, there's the CI/CD[1] pipeline, at the bottom, the DevOps measurement, and in the middle, the code management system. Inside the middle box, the main functionalities of a code management system are listed.

[1] CI/CD: Continuous Integration/Continuous Deployment-Delivery

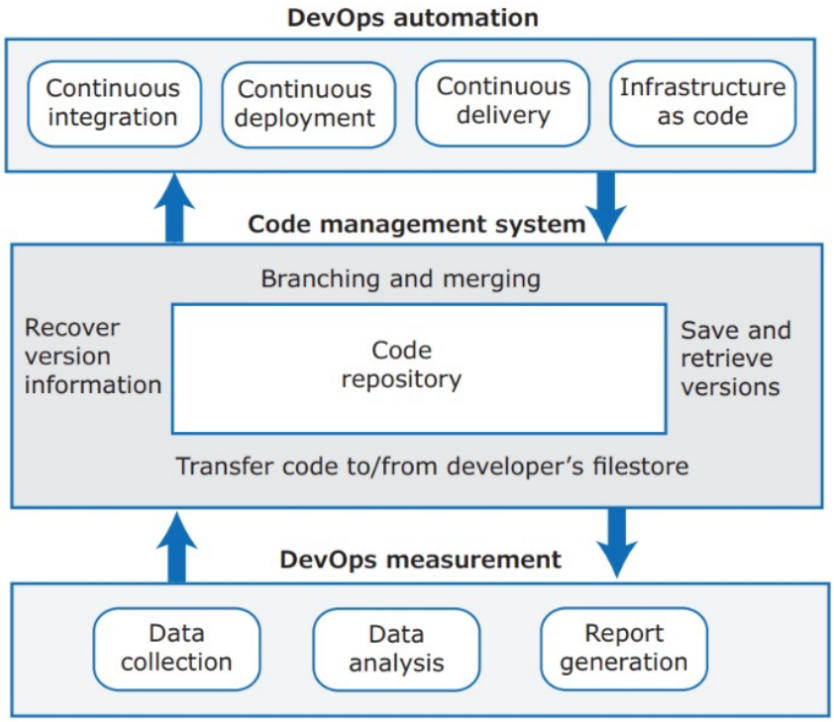


Figure 10.1: DevOps system structure

Features of code management systems are:

- **Version and release identification**: managed versions of a code file are uniquely identified when submitted to the system (managed files can never be overwritten). They can be retrieved using their identifier and other file attributes.
- **Change history recording**: when a change to a code file is submitted, the submitter must add a note explaining the reasons for the change. This helps developers understand why a new version was created.
- **Independent development**: several developers can work on the same code file simultaneously. When submitted to the code management system, a new version is created, ensuring files are never overwritten by later changes, avoiding the file overwriting problem.
- **Project support**: developers download code into a personal file store, work on it, and return it to the shared code management system. Parallel development branches can be created for concurrent work, and changes made in different branches may be merged. All files associated with a project may be checked out simultaneously.
- **Storage management**: the code management system includes efficient storage mechanisms to avoid keeping multiple copies of files with only small differences (less critical today with cheaper storage).

Initially, centralized systems were used for project support, but the **advantages of decentralized systems** (e.g., Git (*Linux's Distributed Version Control System*)) have made decentralized systems the best solution. These advantages include:

- **Resilience**: Everyone working on a project has their own copy of the repository (and people can work offline as well). If the shared repository is damaged or attacked, work can continue, and clones can be used to restore the shared repository.
- **Speed**: committing changes to the repository is a fast operation since commits can be made both locally (without the need for data transfer over the network) and in the general branch.
- **Flexibility**: since everyone has their own copy, local experimentation is much simpler and safer.

A famous decentralized system is **Git**, where there are both *private clones* of the shared repository on each developer's computer and a *shared project* repository (running on the company's server or in the cloud hosted by services like GitHub or GitLab). Figure 10.2 shows some *Git commands* and what the *Git branch* looks like.

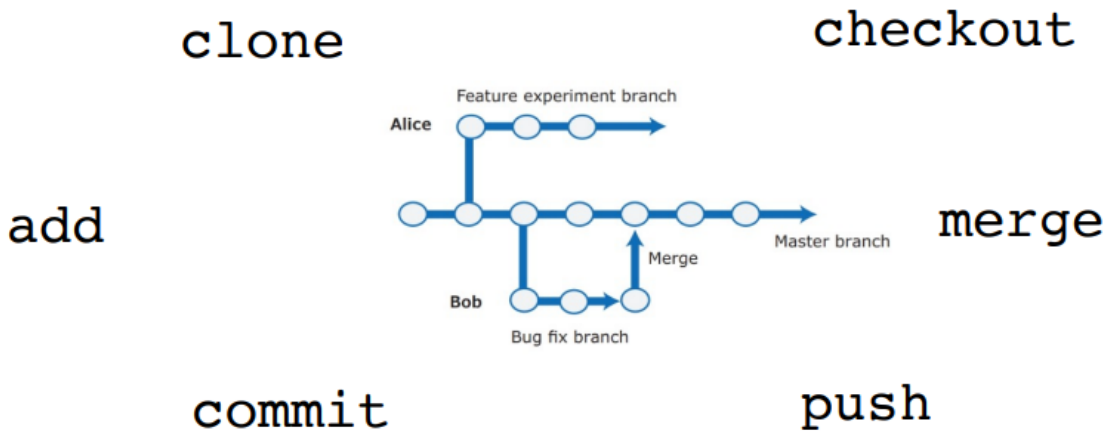


Figure 10.2: Git branch example

When discussing *Open-source software development* with Git, there's usually a group of people who decide what changes should be incorporated. GitHub uses a mechanism called *Webhooks* to trigger DevOps automation tools in response to updates to the project repository.

10.2 DevOps automation

Everything that can be should be automated. This was one of the DevOps principles that we'll discuss in detail in this section of the chapter. The following section will be divided into:

- **Continuous integration**: Each time a developer commits a change to the project's master branch, an executable version of the system is built and tested.
- **Continuous delivery**: Once the new version is built, executable software is tested in a simulated product's operating environment.
- **Continuous deployment**: A new release of the system is made available to users every time a change is made to the master branch of the software.
- **Infrastructure as code**: Machine-readable models of infrastructure (network, servers, routers, etc.) on which the product executes are used by configuration management tools to build the software's execution platform. The software to be installed, such as compilers, libraries, and a DBMS, are included in the infrastructure model.

10.2.1 Continuous integration

The reason why continuous integration should be used is that if a system is infrequently integrated, **problems can be difficult to isolate**, and fixing them slows down system development. System integration (*system building*) is more than compiling:

1. Install database software and set up the database with the appropriate schema.
2. Load test data into the database.
3. Link compiled code with libraries and other components used.
4. Check that external services used are operational.
5. Move configuration files to the correct locations (and delete old ones).
6. Run system tests to check that integration has been successful.

As we can see from Figure 10.4, 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, the repository sends a message to the integration server to build a new version of the product.

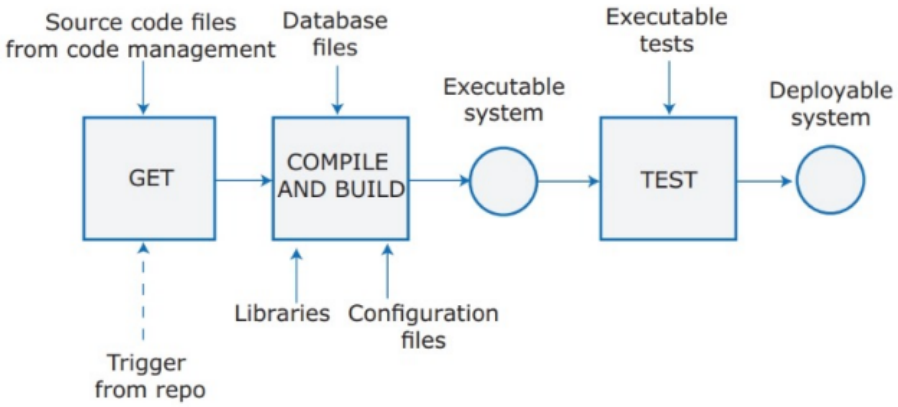


Figure 10.3: Continuous integration pipeline

It's good practice to adopt an **integrate twice** approach to system integration, which consists of a first integration on the local developer machine, and then a second one where the code is pushed to the project repository to trigger the integration server (as we can see from Figure 10.4). This is done because it is really bad to integrate a broken build into the project repository.

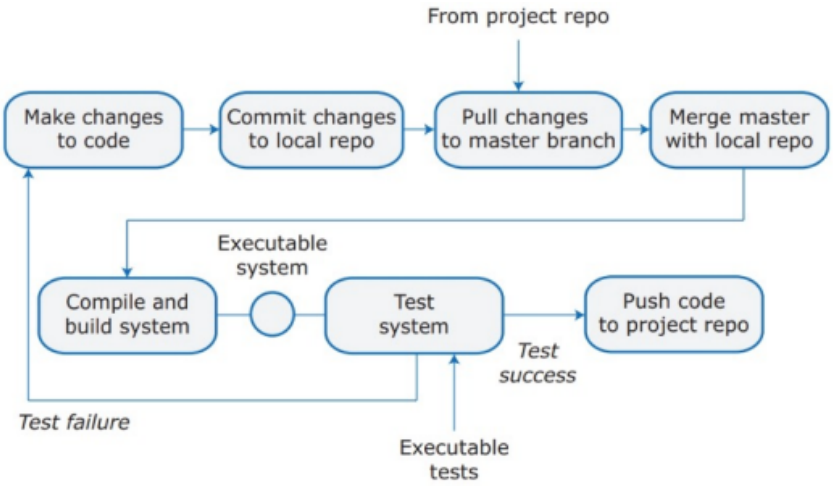


Figure 10.4: Integrate twice pipeline

The **advantages** of continuous integration are:

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

Analyzing the whole codebase every time a commit is made is really heavy, which is why **code integration tools** only repeat actions if dependent files have been changed (e.g., create new object codes only for those whose source code has changed). Modification timestamps can be used to check if some dependencies have changed.

10.2.2 Continuous delivery and deployment

With *Continuous Integration* and *source code management*, it is possible to create an executable version of a software system by building the system and running tests first on the developers' computer and then on the project integration server.

When deploying the executable version of the software into a real production environment, it's possible that something goes wrong since the **real production environment will differ from the development environment** (because the production server may have a different file system organization, access permissions, installed applications, and so on).

Continuous delivery ensures that the changed system is ready for delivery to customers by performing **feature tests** in the production environment (to make sure that the environment does not cause system failures), **system tests**, and **load tests** (to check how the software behaves as the number of users increases). *Containers* are the simplest way to create a replica of a production environment.

As we can see from Figure 10.5, to **deliver** the new system, a staged test environment is configured (after some initial *integration testing*), and then system acceptance tests (*functionality*, *load*, *performance*) are launched. To **deploy** it, software and data are transferred to production servers, then a switch to the new system version is made, and finalized with a restart of the process (it's critical to also manage situations where clients are still connected to the old version of the system).

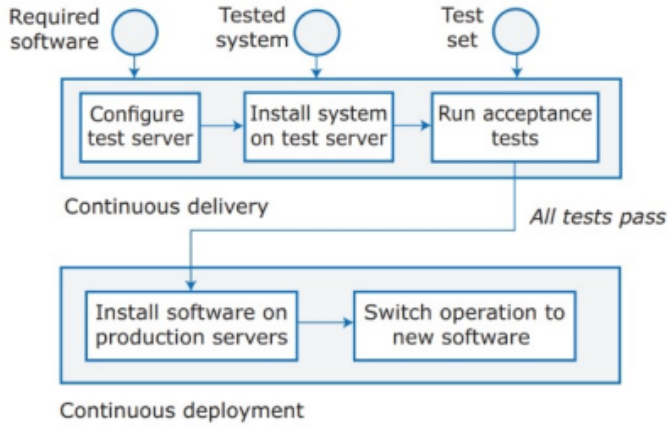


Figure 10.5: Continuous delivery and continuous deployment schema

The **benefits** of continuous deployment are:

- **Reduced costs**: fully automated deployment pipeline.
- **Faster problem solving**: if a problem occurs, it will probably affect only a small part of the system, and the source of that problem will be obvious.
- **Faster customer feedback**: it is possible to deploy new features when they are ready for customer use, and then users' feedback can be used to identify improvements.
- **A/B testing**: if there are many customers and several servers, it's possible to deploy a new software version only on some servers and use a load balancer to divert some customers to the new version, in order to measure and assess how new features are used.

Notice that it's probably best to **not deploy every single change**, since small changes may have incomplete features that could be deployed, and it's important to not disclose this to competitors until implementation is complete. Another reason is that customers may be irritated by continually changing software, especially if this affects the UI. Developers may also want to synchronize releases with known business cycles (e.g., the start of the academic year for the education market).

There are many Continuous Integration **tools** (such as *Jenkins* and *Travis*) that may also be used to support Continuous Delivery and Deployment. These tools can integrate with infrastructure configuration management tools to implement software deployment, even though for *cloud-based software*, it is often simpler to use containers in conjunction with CI tools rather than infrastructure configuration management software.

10.2.3 Infrastructure as code

Manually maintaining a computing infrastructure with tens or hundreds of servers is **expensive** and **error-prone**. This is because many different physical/virtual servers have different configurations and run different software packages, so when new versions of software become available, some servers may have to be updated, while others may not, due to dependencies with older versions of software. Tracking manually the software installed on each server is really hard, and not done every time (e.g., emergency changes are not always documented).

It would be nice to automate the process of updating software on servers by using a machine-readable model of the infrastructure: **Configuration Management** (CM) **tools** (like *Puppet*, *Chef*, and *Ansible*) can automatically install software and services on servers according to the infrastructure definition (infrastructure is represented as data, that's why we call it IaC (Figure 10.6)). When changes have to be made, the infrastructure definition model is updated, and the CM tool makes the changes to all servers.

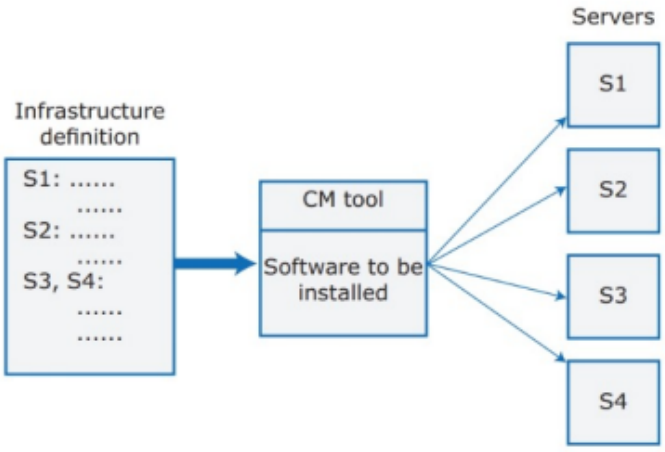


Figure 10.6: Infrastructure as code schema

The **advantages** of infrastructure as code are:

- **Visibility**: Infrastructure is defined as a stand-alone model that can be read, understood, and reviewed by the whole DevOps team.
- **Reproducibility**: Installation tasks will always be performed in the same sequence, and the same environment will always be created. You do not have to rely on people remembering the order in which they need to do things (computers are superior to humans in terms of reproducibility).
- **Reliability**: Automation avoids simple mistakes made by system administrators when making the same changes to several servers.
- **Recovery**: The infrastructure model can be versioned and stored in a code management system. If infrastructure changes cause problems, you can easily revert to an older version and reinstall the environment that developers know works.

Containers are a very effective way to deploy cloud-based products. With containers, it's simple to provide **identical execution environments** (for each type of server developers use, they define the environment they need and build an image for execution. When developers update the software, they create a new image that includes the modified software). Developers can employ a *container management system* (such as Kubernetes) for scaling, resiliency, and other orchestration properties.

10.3 DevOps measurement

In order to continuously **improve your DevOps process** to achieve faster deployment of better-quality software, measuring and analyzing product and process data is needed.
There are various types of data measurements:

- **Process measurements**: Collect and analyze data on development, testing, and deployment processes.
- **Service measurements**: Collect and analyze data on software's performance, reliability, and acceptability to customers.
- **Usage measurements**: Collect and analyze data on how customers use your product instead of using pop-ups that ask if they liked a certain product (this helps to identify issues and problems with the software itself).
- **Business success measurements**: Collect and analyze data on how your product contributes to overall business success (it's hard to isolate the contribution of DevOps to business success since that may be due to DevOps introduction or to better management).

Measuring software and its development is a **complex process**, since developers need to identify suitable metrics that give useful insights and find reliable ways of collecting and analyzing metric data. Some of the measures (e.g., customer satisfaction) must be inferred from other metrics (e.g., the number of returning customers).

Software measurement should be automated as far as possible, so developers must instrument their software to collect data about itself and use a monitoring system to collect data about software's performance and availability.
In order to collect data automatically, it's possible to:

- Use **continuous integration tools** like Jenkins to collect data on deployments, successful tests, and so on.
- Use **monitoring services** provided by cloud providers like Amazon's Cloudwatch to collect data on availability and performance.
- Collect customer-supplied data from **issue management systems**.
- **Add instrumentation to a product** to gather data on its performance and how it is used by customers (usually log files are the best option: log as many events as possible, and use log analysis tools to extract useful information on how your software is used).

Chapter 11

Emerging Paradigms in Computing

11.1 Cloud-Edge Continuum

The **Cloud-Edge Continuum** aims to combine the strengths of Edge Computing and Cloud Computing by extending cloud services to the Internet of Things (IoT). This results in a **distributed, heterogeneous infrastructure**. Key benefits of this approach include:

- **Computing Power**: Access to both cloud resources and edge devices for efficient processing.
- **Connectivity**: Improved communication between devices, facilitating seamless data transfer.
- **Low Latency**: Processing data closer to its source reduces delays and enhances responsiveness.

Future applications will primarily be **containerized, microservice-based** solutions operating on a continuous Cloud-Edge infrastructure. However, managing these applications poses challenges. Both the infrastructure and the applications are constantly changing, leading to potential issues:

Infrastructure Changes : Node workloads may shift, latency and bandwidth can vary, and nodes might join or leave the network unexpectedly. Temporary connection failures can also occur.

Application Changes : Codebases and requirements are subject to change, necessitating quick adaptations.

This highlights the need for ongoing **management** of application deployments even after the initial launch.

11.1.1 Monitoring

Effective **monitoring** is essential for tracking both applications and infrastructure. We require a lightweight, fault-tolerant system that can adapt to changes. One effective method is **continuous reasoning**, which analyzes large systems by focusing on recent changes and reusing previous results whenever possible. Considering both application and infrastructure changes is crucial in the continuum for making decisions about replacing, migrating, restarting, or scaling application services.

A recent example discussed is **FogBrainX**[1], which evaluates differences in application specifications and monitored infrastructure data. It assists in making decisions about service placement by:

- Adapting to changes in infrastructure, such as node resources or network quality, which may require migrating services,
- Adjusting to shifts in service requirements (like software, hardware, and IoT) or communication needs that might trigger updates,
- Handling additions or removals of services or communication requirements outlined in application specifications.

After FogBrainX makes its decisions, it forwards them to **FogArm**, which executes the management commands within the Cloud-IoT infrastructure.

11.1.2 Decentralized Management

When it comes to decentralized management, we have two notable approaches:

Osmotic Management : This method allows application services to adapt based on available resources and application needs. Management policies can undeploy, migrate, or scale applications up or down in real time.

Decentralized Management : Inspired by bacterial behavior, this approach involves:

- Assigning each application instance a management agent (similar to a mini-management unit),
- Using simple rules to trigger actions (like undeploying or replicating) based on monitored data,
- Allowing emerging behaviors of applications to facilitate flexibility, which is beneficial for larger infrastructures, although it can be more complex to manage.

—

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

11.2 Quantum Software Engineering

Quantum Software Engineering (QSE) focuses on applying sound engineering principles to develop, operate, and maintain quantum software and its documentation. The primary goal is to create reliable quantum software that performs efficiently on quantum computers while being cost-effective.

The Talavera Manifesto for QSE outlines several important principles, including:

- QSE should be compatible with various quantum programming languages and technologies.
- We should embrace the integration of classical and quantum computing, using quantum computers primarily for specific tasks where they excel, such as solving factorization problems.

11.2.1 Quantum Broker

One practical example of QSE is the **Quantum Broker**, which addresses the question: "Which Quantum Computer should I use to run my algorithm?" This is particularly useful for clients who may not have extensive knowledge of quantum providers.

Even after selecting a provider and running algorithms, several challenges may arise:

- **Availability**: What if the Quantum Computer becomes unavailable during execution?
- **Requirements Accuracy**: How can I balance my needs for cost, time, and accuracy?
- **Customization**: How can I modify the Quantum Computer's decision-making process to suit my requirements?

The concept of **shot distribution** is vital for optimizing quantum computations. By distributing the shots of a quantum circuit across multiple quantum computers, we can enhance performance and reliability. Running the same quantum circuit multiple times helps gather statistical information about outcomes, which is important due to the probabilistic nature of quantum mechanics. We can then combine all this data to form a complete view of the circuit's performance.

Given a quantum circuit and specific requirements, the quantum broker **selects the best set of quantum computers** for distributing the shots. For each chosen computer, it identifies the most suitable compilers and the required number of shots. This approach offers several advantages:

- **Improved Resilience**: Distributing tasks among different quantum systems increases tolerance to failures.

- **High Customization**: This method allows for significant customization in managing computations.
- **Partial Distributions**: It enables the creation of partial distributions that can be combined to generate a comprehensive view of the client's circuit execution, with the possibility of more sophisticated merging.