University of Pisa

Computer Science Department

Notes of Advanced Software Engineering

Based on the lectures of A. Brogi, S. Forti

A.Y. 2022-2023

Disclaimer: these notes are not a substitute for lessons and exercises. They contain both grammatical and logical errors and may be misleading or unclear.
Use them with caution.

Contents

Chapter 1

Software Products

Software products are generic software systems that provide functionality that is useful to a range of customers. Many different types of products are available from large-scale business systems (e.g. MS Excel) through personal products (e.g. Evernote) to simple mobile phone apps and games (e.g. Suduko). Software product engineering methods and techniques have evolved from software engineering techniques that support the development of one-off, custom software systems. Custom software systems are still important for large businesses, government and public bodies. They are developed in dedicated software projects. The starting point for the software development is a set of *'software requirements'* that are

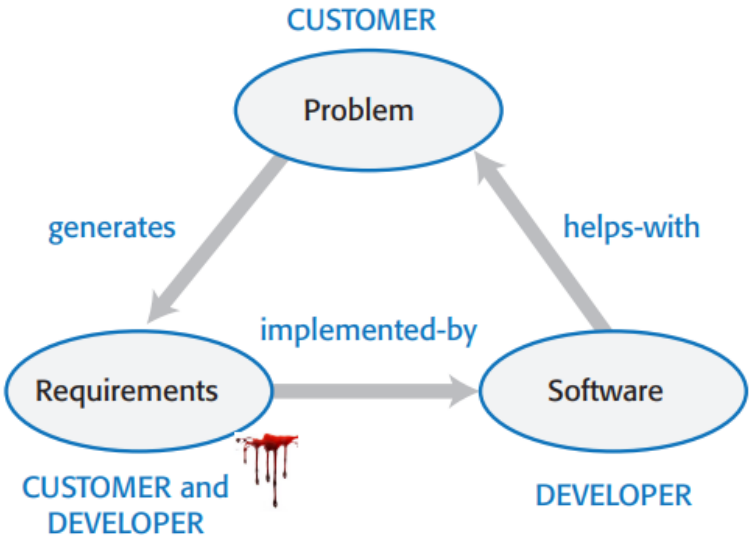


Figure 1.1: The 23 Design Patterns of the Gang of Four; Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software (1995).

owned by an external client and which set out what they want a software system to do to support their business processes. The software is developed by a software company (the contractor) who design and implement a system that delivers functionality to meet the requirements. The customer may change the requirements at any time in response to business changes (they usually do). The contractor must change the software to reflect these requirements changes. Custom software usually has a long-lifetime (10 years or more) and it must be supported over that lifetime. Another paradigms is represented by **product-based Software Engineering** as pictured in 1.2

The starting point for product development is a business opportunity thatis identified by individuals or a company. They develop a software product to take advantage of this opportunity and sell this to customers. The company who identified the opportunity design and implement a setof software features that realize the opportunity and that will be useful to customers.The software development company are responsible for deciding on the development timescale, what features to include and when the product should change. Rapid delivery of software products is essential to capture the market for that type of product.

This two type of SE can define also the **Software Execution models**:

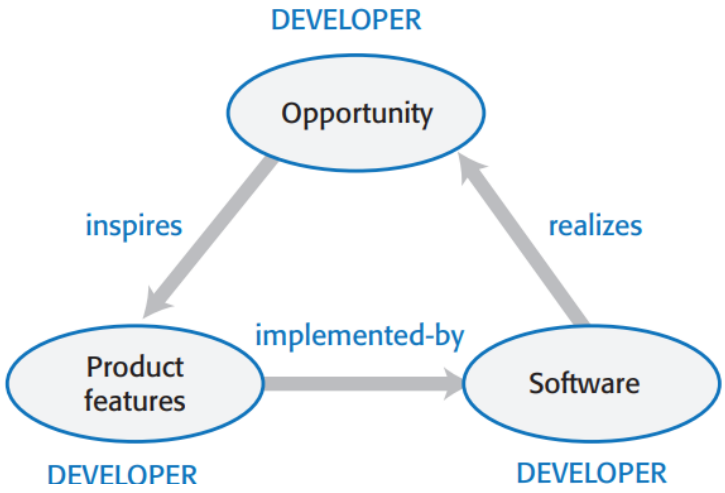


Figure 1.2: Here developer decides product features and following evolutions

- *Stand-alone*: The software executes entirely on the customer's computers.
- *Hybrid*: Part of the software's functionality is implemented on the customer's computer but some features are implemented on the product developer's servers.
- *Software service*:All of the product's features are implemented on the developer's servers and the customer accesses these through a browser or a mobile app.

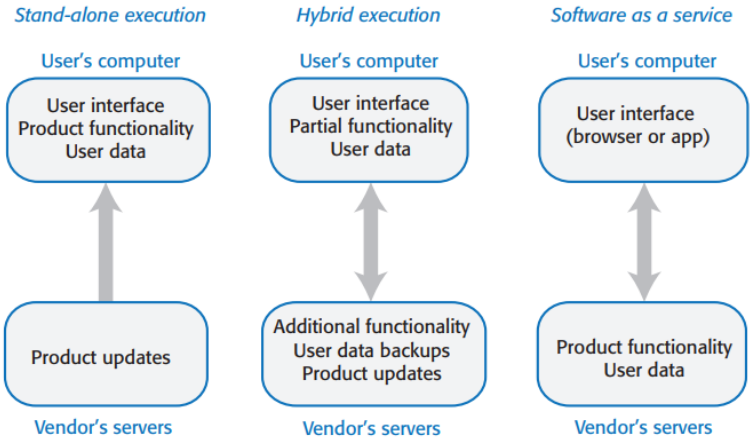


Figure 1.3: Here developer decides product features and following evolutions

1.1 Product vision

The starting point for software product development is a 'product vision'. Product visions are simple statements that define the essence of the product to be developed. The product vision should answer three fundamental questions:

- What is the product to be developed?
- Who are the target customers and users?
- Why should customers buy this product?

This allow to present a general template, pictured in 1.4 and some examples in 1.5. The product vision is determined by:

- *Domain experience*: The product developers may work in a particular area (say marketing and sales) and understand the software support that 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 see simpler and better ways of providing comparable functionality and propose a new system that implements this. New products can take advantage of recent technological developments such as speech interfaces.
- *Customer experience*: The software developers may have extensive discussions with prospective customers of the product to understand the problems that they face, constraints, such as interoperability, that limit their flexibility to buy new software, and the critical attributes of the software that they need.
- *Prototyping and playing around*:Developers may have an idea for software but need to develop a better understanding of that idea and what might be involved in developing it into a product. They may develop a prototype system as an experiment and 'play around' with ideas and variations using that prototype system as a platform.

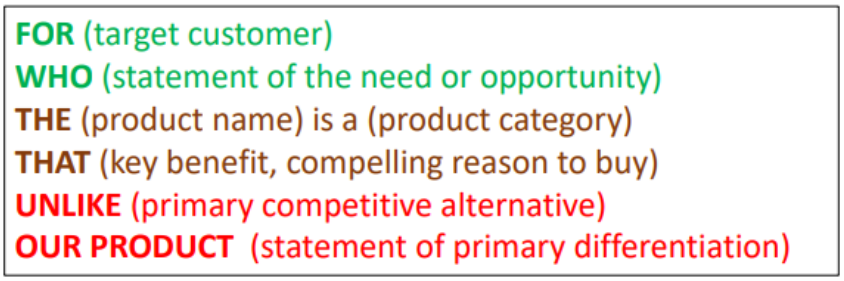


Figure 1.4: Here developer decides product features and following evolutions

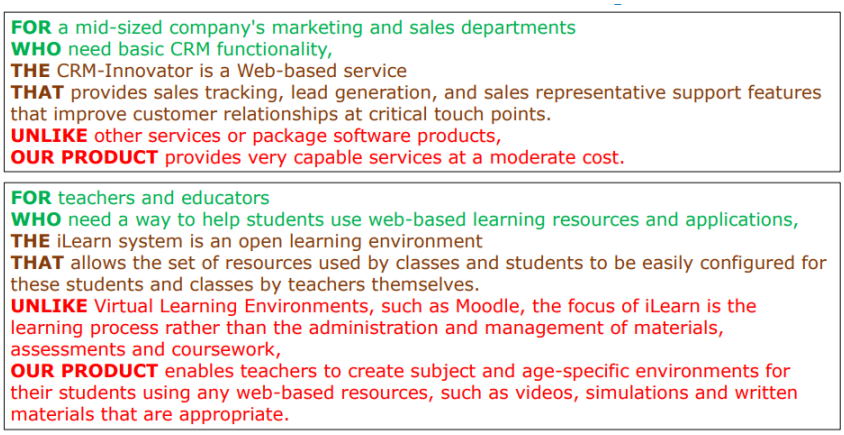


Figure 1.5: Here developer decides product features and following evolutions

1.2 Software Product Management

Software product management is a business activity that focuses on the software products developed and sold by the business. Product managers (PMs) take overall responsibility for the product and are involved in planning, development and product marketing. Product managers are the interface between the organization, its customers and the software development team. They are involved at all stages of a product's lifetime from initial conception through to withdrawal of the product from the market. Product managers must look outward to customers and potential customers rather than focus on the software being developed. Product management concerns

- Business needs: PMs have to ensure that the software being developed meets the business goals of the software development company.
- Technology constraints: PMs must make developers aware of technology issues that are important to customers.

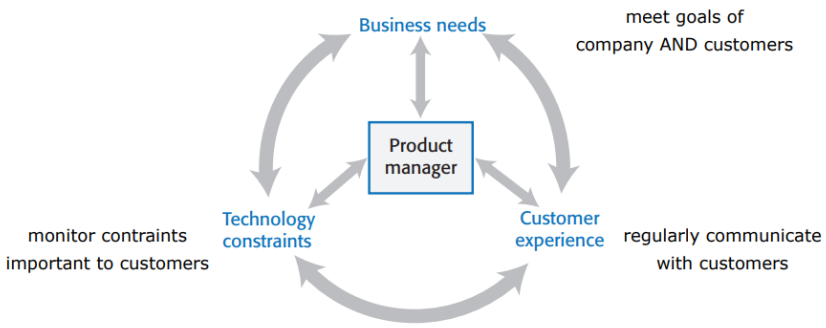


Figure 1.6: Here developer decides product features and following evolutions

- Customer experience: PMs should be in regular contact with customers and potential customers to understand what they are looking for in a product, the types of users and their backgrounds and the ways that the product may be used.

Technical interactions of PMs are:

- Product roadmap (set goals, milestones, success criteria)
- User story and scenario (to identify product features)
- Product backlog (to-do list to complete project development)
- Acceptance testing (to verify that release meets set goals)
- Customer testing (to get feedback on usability & fit of features)
- UI design (monitor simplicity/naturality)

1.3 Product prototyping

Product prototyping is the process of developing an early version of a product to test your ideas and to convince yourself and company funders that your product has real market potential. You may be able to write an inspiring product vision, but your potential users can only really relate to your product when they see a working version of your software. They can point out what they like and don't like about it and make suggestions for new features.

A prototype may be also used to help identify fundamental software components or services and to test technology. Building a prototype should be the first thing that you do when developing a software product. Your aim should be to have a working version of your software that can be used to demonstrate its key features. You should always plan to throw-away the prototype after development and to re-implement the software, taking account of issues such as security and reliability.

There are **two stages** of prototyping, allowing to have a first-reduce prototype in 6 weeks:

- *Feasability demonstration*: You create an executable system that demonstrates the new ideas in your product. The aims at this stage are to see if your ideas actually work and to show funders and/or company management the original product features that are better than those in competing products.
- *Customer demonstration*: You take an existing prototype created to demonstrate feasibility and extend this with your ideas for specific customer features and how these can be realized. Before you develop this type of prototype, you need to do some user studies and have a clearer idea of your potential users and scenarios of use.

Chapter 2

Agile Software Engineering

Software products must be brought to market quickly so rapid software development and delivery is essential. Virtually all software products are now developed using an agile approach.Agile software engineering focuses on delivering functionality quickly, responding to changing product specifications and minimizing development overheads. A large number of 'agile methods' have been developed: there is no 'best' agile method or technique. It depends on who is using the technique, the development team and the type of product being developed.

Plan-driven development evolved to support the engineering of large, long-lifetime systems (such as aircraft control systems) where teams may be geographically dispersed and work on the software for several years. This approach is based on controlled and rigorous software development processes that include detailed project planning, requirements specification and analysis and system modelling. However, plan-driven development involves significant overheads anddocumentation and it does not support the rapid development and delivery of software. Agile methods were developed in the 1990s to address this problem: these methods focus on the software rather than its documentation, develop software in a series of increments and aim to reduce process bureaucracy as much as possible. The **Agile manifesto** state that: "We are uncovering better ways of developing software by doing it and helping others to do it. Through this work, we have come to value:

- individuals and interactions over processes and tools;
- working software over comprehensive documentation;
- customer collaboration over contract negotiation;
- responding to change over following a plan.

While there is value on the items on the right, we value the items on the left more."

All agile methods are based around incremental development and delivery. Product development focuses on the software featues, when a featue does something for the software user. With incremental development, you start by prioritizing the features so that the most important features are implemented first: you only define the details of the feature being implemented in an increment. That feature is then implemented and delivered. Users or surrogate users can try it out and provide feedback to the development team. You then go on to define and implement the next feature of the system as pictured in 2.1- That process can be syntethized by following these twelve principles:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

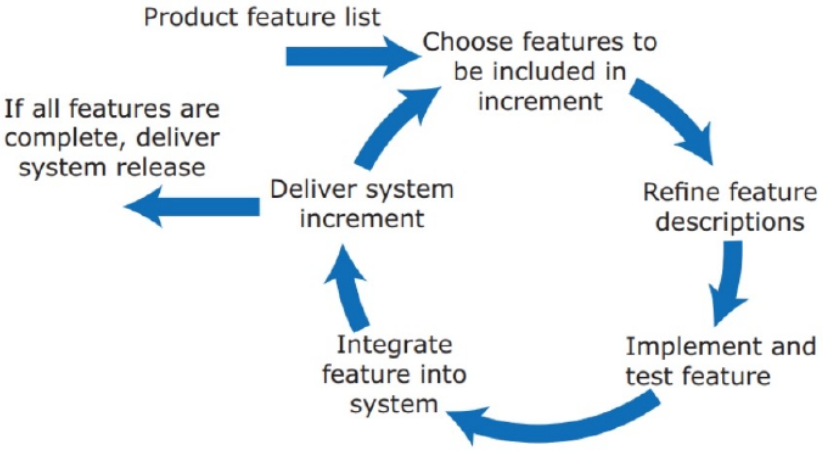


Figure 2.1: Here developer decides product features and following evolutions

6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity–the art of maximizing the amount of work not done–is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

2.1 Extreme programming

The most influential work that has changed software development culture was the development of Extreme Programming (XP). The name was coined by Kent Beck in 1998 because the approach was developed by pushing recognized good practice, such as iterative development, to 'extreme' levels. Extreme programming focused on 12 new development techniques, pictured in 2.2, that were geared to rapid, incremental software development, change and delivery. Some of these techniques are now widely used; others have been less popular.

- Incremental planning/user stories: There is no 'grand plan' for the system. Instead, what needs to be implemented (the requirements) in each increment are established in discussions with a customer representative. The requirements are written as user stories. The stories to be included in a release are determined by the time available and their relative priority.
- Small releases: The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the previous release.
- Test-driven development: Instead of writing code then tests for that code, developers write the tests first. This helps clarify what the code should actually do and that there is always a 'tested' version of the code available. An automated unit test framework is used to run the tests after every change. New code should not 'break' code that has already been implemented.
- Continuous integration: As soon as the work on a task is complete, it is integrated into the whole system and a new version of the system is created. All unit tests from all developers are run automatically and must be successful before the new version of the system is accepted.

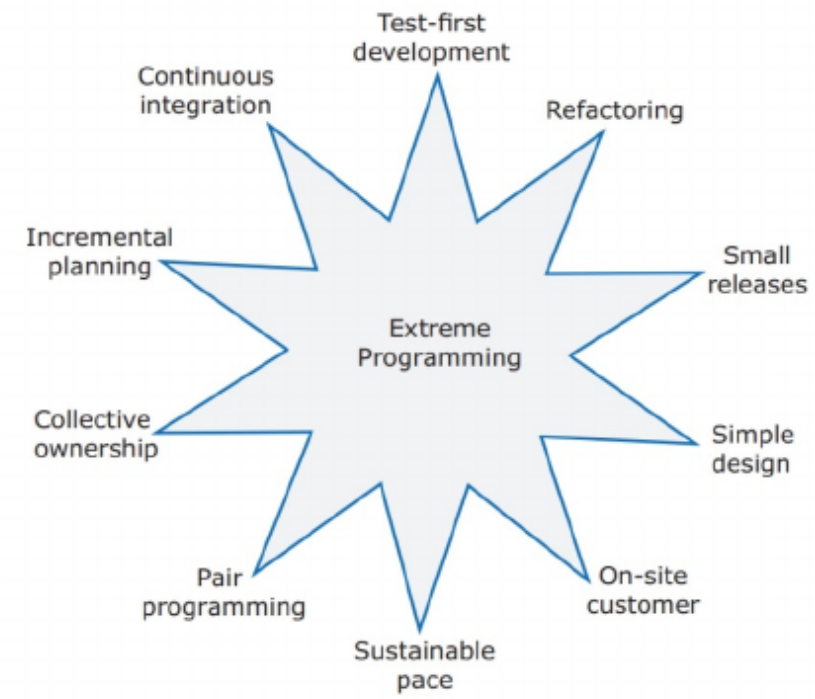


Figure 2.2: Here developer decides product features and following evolutions

- Refactoring: Refactoring means improving the structure, readability, efficiency and security of a program. All developers are expected to refactor the code as soon as potential code improvements are found. This keeps the code simple and maintainable.

2.2 Scrum

Software company managers need information that will help them understand how much it costs to develop a software product, how long it will take 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.Plans always change so anything apart from short-term plans are unreliable.

Scrum is a lightweight framework that helps people, teams and organizations generate value through adaptive solutions for complex problems. The Scrum Framework contains a set of principles and rules to following order to achieve a common goal.

Scrum is founded on **empiricism**, it asserts that knowledge comes from experience and making decisions based on what is observed and **lean thinking**: it reduces waste and focuses on the essentials. Scrum employs an iterative, incremental approach to optimize predictability and to control risk.

The Scrum theory **pillars** are:

- Transparency: The emergent process and work must be visible to those performing the work as well as those receiving the work.
- Inspection: Scrum artifacts and the progress toward agreed goals must be inspected frequently and diligently to detect potentially undesirable variances or problems.
- Adaptation:If any aspects of a process deviate outside acceptable limits or if the resulting product is unacceptable, the process being applied or the materials being produced must be adjusted. The adjustment must be made as soon as possible to minimize further deviation.
-

Successful use of Scrum depends on people becoming more proficient in living five values: *Commitment, Focus, Openness, Respect, Courage.*

2.3 Scrum in practice

The scrum's main actors are:

- Scrum team: composed by Product owner, Scrum Master, Developers
- Artifacts: composed by Product backlog, Sprint backlog
- Scrum events: Sprint planning, execution and review

There are two main key roles in Scrum. The firstone is the **Product owner** that is responsible for ensuring that the development team are always focused on the product they are building rather than diverted into technically interesting but less relevant work. In product development, the product manager should normally take on the Product Owner role. The second one is the **Scrum master** that is a Scrum expert whose job is to guide the team in the effective use of the Scrum method. The developers of Scrum emphasize that the ScrumMaster is not a conventional project manager but is a coach for the team. They have authority within the team on how Scrum is used. In many companies that use Scrum, the ScrumMaster also has some project management responsibilities.

2.3.1 Scrum in Sprints

In Scrum, software is developed in sprints, which are fixed-length periods (2 - 4 weeks) in which software features are developed and delivered. During a sprint, the team has daily meetings (Scrums) to review progress and to update the list of work items that are incomplete. Sprints should produce a 'shippable product increment'. This means that the developed software should be complete and ready to deploy.

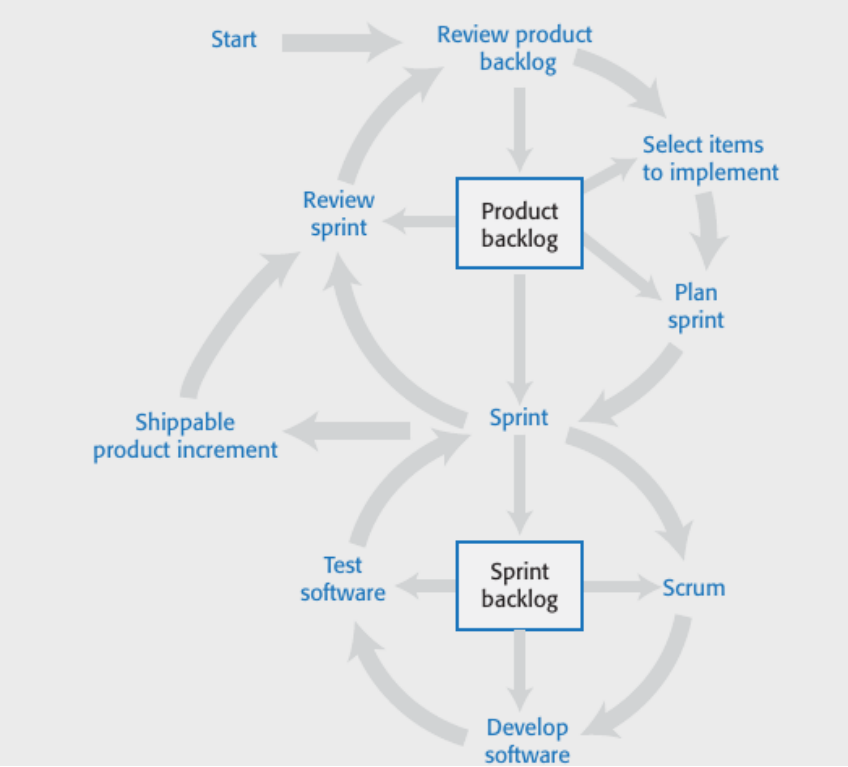


Figure 2.3: Scrum cycles

The **Product backlog** is a to-do list of items to be implemented that is reviewed and updated before each sprint. The **sprints are timeboxed** in a fixed-time (2-4 week) periods in which items from the product backlog are implemented. Also teams are **self-organized**, making their own decisions and work by discussing issues and making decisions by consensus.

The **product backlog** is a list of what needs to be done to complete the development of the product. The items on this list are called product backlog items (PBIs). The product backlog may include a variety of different items such as product features to be implemented, user requests, essential development activities and desirable engineering improvements. The product backlog should always be prioritized so

that the items that be implemented first are at the top of the list. This is an example of **product backlog items**:

1. As a teacher, I want to be able to configure the group of tools that are available to individual classes. (feature)
2. As a parent, I want to be able to view my childrens' work and the assessments made by their teachers. (feature)
3. As a teacher of young children, I want a pictorial interface for children with limited reading ability. (user request)
4. Establish criteria for the assessment of open source software that might be used as a basis for parts of this system. (development activity)
5. Refactor user interface code to improve understandability and performance. (engineering improvement)
6. Implement encryption for all personal user data. (engineering improvement)

The product backlog item states:

- Ready for consideration: these are high-level ideas and feature descriptions that will be considered for inclusion in the product. They are tentative so may radically change or may not be included in the final product.
- Ready for refinement: the team has agreed that this is an important item that should be implemented as part of the current development. There is a reasonably clear definition of what is required. However, work is needed to understand and refine the item.
- Ready for implementation: the PBI has enough detail for the team to estimate the effort involved and to implement the item. Dependencies on other items have been identified.

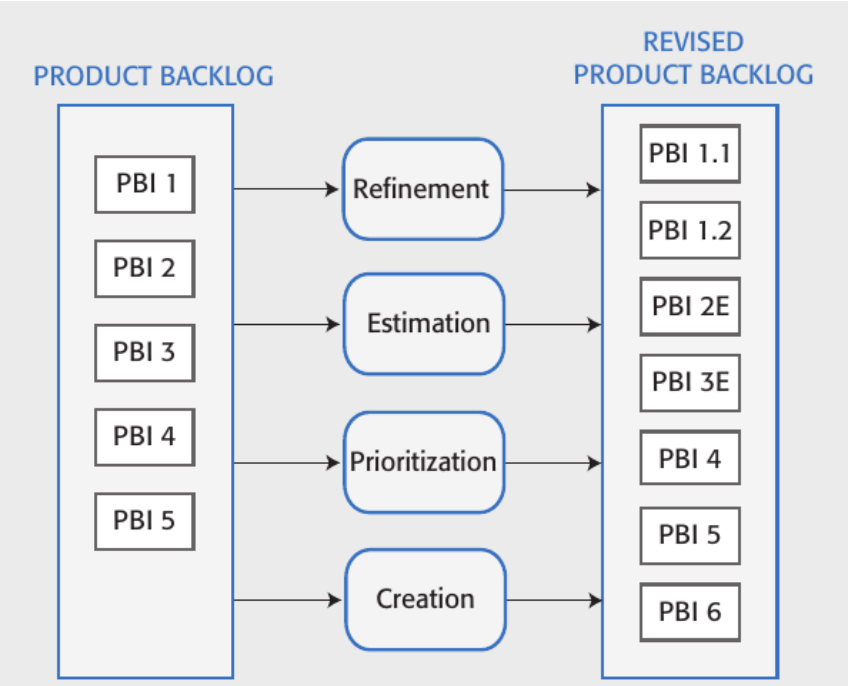


Figure 2.4: Scrum cycles

The main **product backlog activities** are concentrated on:

- Refinement: Existing PBIs are analysed and refined to create more detailed PBIs. This may lead to the creation of new product backlog items.
- Estimation: The team estimate the amount of work required to implement a PBI and add this assessment to each analysed PBI.

- Creation: New items are added to the backlog. These may be new features suggested by the product manager, required feature changes, engineering improvements, or process activities such as the assessment of development tools that might be used.
- Priorization: The product backlog items are reordered to take new information and changed circumstances into account.

2.3.2 PBI estimation metrics

The two main metrics are:

- **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. Several people may work on an item, which may 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 derived originally by comparing user stories, but they can be used for estimating any kind of PBI. Story points are estimated relatively. The team agree 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.

2.4 Sprint activities and planning

Products are developed in a series of sprints, each of which delivers an increment of the product or supporting software.Sprints are short duration activities (2-4 weeks) and take place between a defined start and end date. Sprints are timeboxed, which means that development stops at the end of a sprint whether or not the work has been completed. During a sprint, the team work on the items from the product backlog.

We can distinguish three main activities (in picture 2.5):

- Sprint planning: Work items to be completed in that sprint are selected and, if necessary, refined to create a sprint backlog. This should not last more than a day at the beginning of the sprint.
- Sprint execution: The team work to implement the sprint backlog items that have been chosen for that sprint. If it is impossible to complete all of the sprint backlog items, the sprint is not extended. The unfinished items are returned to the product backlog and queued for a future sprint.
- Sprint reviewing: The work done in the sprint is reviewed by the team and (possibly) external stakeholders. The team reflect on what went well and what went wrong during the sprint with a view to improving their work process.

2.4.1 Sprint planning

Allows to establish an agreed sprint goal: Sprint goals may be focused on software functionality, support or performance and reliability. Decide on the list of items from the product backlog that should be implemented, then create a sprint backlog: this is a more detailed version of the product backlog that records the work to be done during the sprint.

In a sprint plan, the team decides which items in the product backlogshould be implemented during that sprint. Key inputs are the effort estimates associated with PBIs and the team's velocity. The output of the sprint planning process is a sprint backlog.: is a breakdown of PBIs to show the what is involved in implementing the PBIs chosen for that sprint. During a sprint, the team have daily meetings (scrums) to coordinate their work.

A scrum is a short, daily meeting that is usually held at the beginning of the day. During a scrum, all team members share information, describe their progress since the previous day's scrum, problems that have arisen and plans for the coming day. This means that everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them.

Scrum meetings should be short and focused. To dissuade team members from getting involved in long discussions, they are sometimes organized as 'stand-up' meetings where there are no chairs in the meeting room.

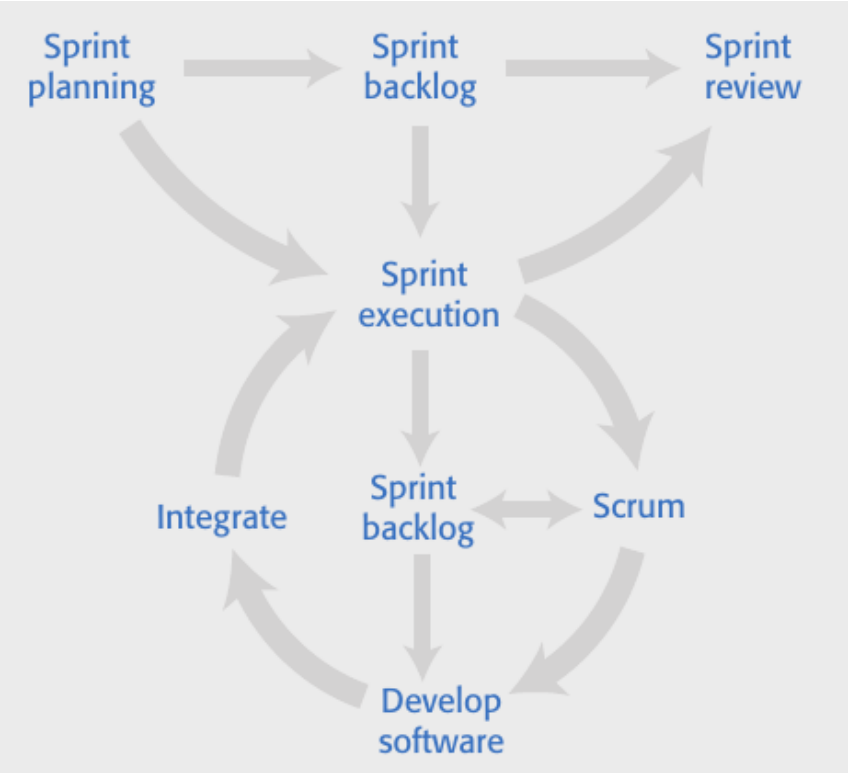


Figure 2.5: Scrum cycles

During a scrum, the sprint backlog is reviewed. Completed items are removed from it. New items may be added to the backlog as new information emerges. The team then decide who should work on sprint backlog items that day.

Scrum does not suggest the technical agile activities that should be used. However, there are two practices that are strongly suggested by agile software engineers to be used in a sprint:

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

At the end of each sprint, there is a review meeting, which involves the whole team. This meeting allows to reviews whether or not the sprint has met its goal,sets out any new problems and issues that have emerged during the sprint. This is a way for a team to reflect on how they can improve the way they work. The product owner has the ultimate authority to decide whether or not the goal of the print has been achieved. They should confirm that the implementation of the selected product backlog items is complete.

The sprint review should include a process review, in which the team reflects on its own way of working and how Scrum has been used. The aim is to identify ways to improve and to discuss how to use Scrum more productively, following the self-organizing team philosphy 2.7.

2.4.2 Team composition and coordination

The ideal Scrum team size is between 5 and 8 people: teams have to tackle diverse tasks and so usually require people with different skills, such as networking, user experience, database design and so on.

They usually involve people with different levels of experience.A team of 5-8 people is large enough to be diverse yet small enough to communicate informally and effectively and to agree on the priorities of the team. The advantage of a self-organizing team is that it can be a cohesive team that can adapt to change. Because the team rather than individuals take responsibility for the work, they can cope with people leaving and joining the team. Good team communication means that team members inevitably learn something about each other's areas.

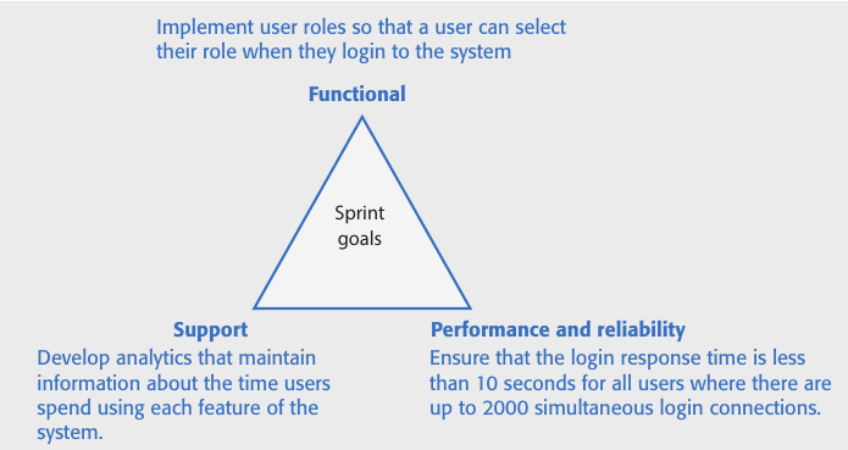


Figure 2.6: Scrum cycles

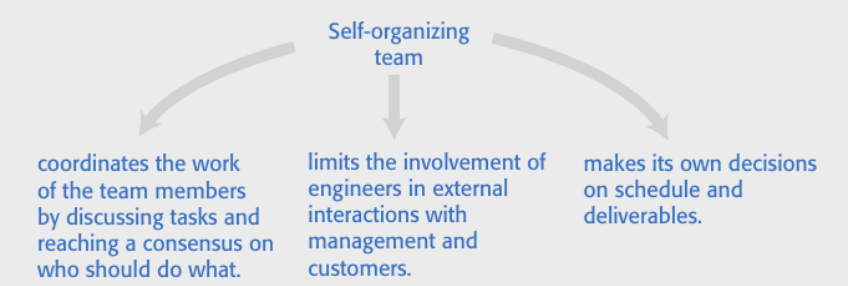


Figure 2.7: Scrum cycles

The developers of Scrum assumed that teams would be co-located. They would work in the same room and could communicate informally. Daily scrums mean that the team members know what's been done and what others are doing. However, the use of daily scrums as a coordination mechanism is based on two assumptions that are not always correct:

- Scrum assumes that the team will be made up of full-time workers who share a workspace. In reality, team members may be part-time and may work in different places. For a student project team, the team members may take different classes at different times.
- Scrum assumes that all team members can attend a morning meeting to coordinate the work for the day. However, some team members may work flexible hours (e.g. because of childcare responsibilities) or may work on several projects at the same time.

The **External interactions** are interactions that team members have with people outside of the team. In Scrum, the idea is that developers should focus on development and only the ScrumMaster and Product Owner should be involved in external interactions. The intention is that the team should be able to work on software development without external interference or distractions.

2.5 Project management

In all but the smallest product development companies, there is a need for development teams to report on progress to company management. A self-organizing team has to appoint someone to take on these responsibilities: because of the need to maintain continuity of communication with people outside of the group, rotating these activities around team members is not aviable approach. The developers of Scrum did not envisage that the ScrumMaster should also have project management responsibilities: in many companies, however, the ScrumMaster has to take on project management responsibilities.They know the work going on and are in the best position to provide accurate information and project plans and progress.

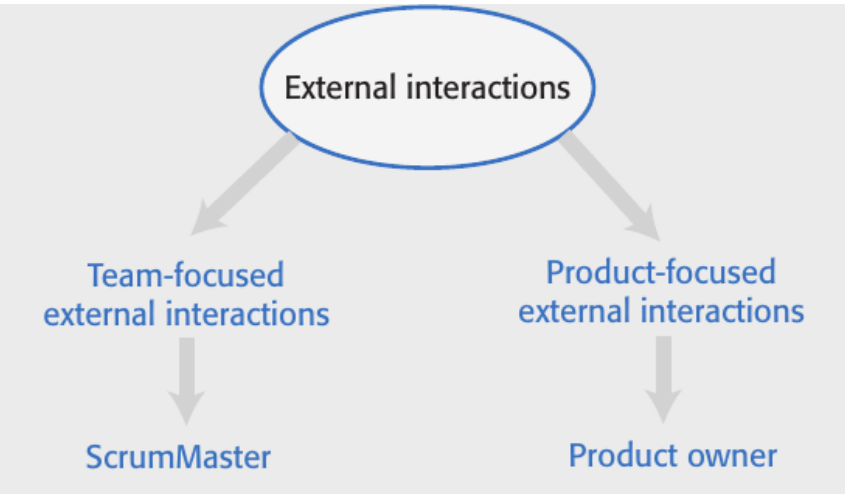


Figure 2.8: Scrum cycles

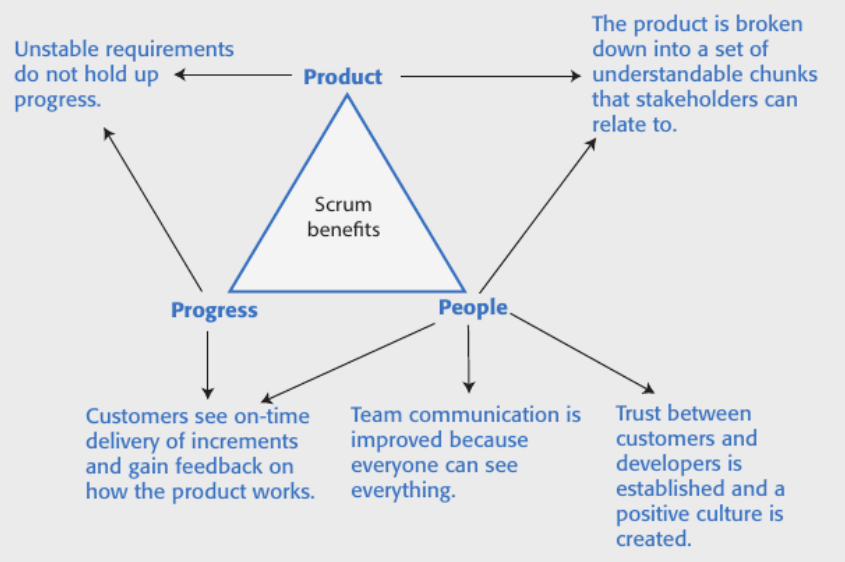


Figure 2.9: Scrum cycles

Chapter 3

Scenarios, personas & stories

The factors driving the design of software products are *inspiration, business/consumer needs to met by existing products, dissatisfaction with existing products and technical changes making new product types possible.*

Product-based software engineering needs less requirements documentation that project-based software engineering because requirements not set by customers and requirements can change.

Identify product features (= fragments of functionality) instead need to understand potential users by interviews/surveys, informal user analysis/consultations. Remember that business managers buy product, employee (may not want to) use it.

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

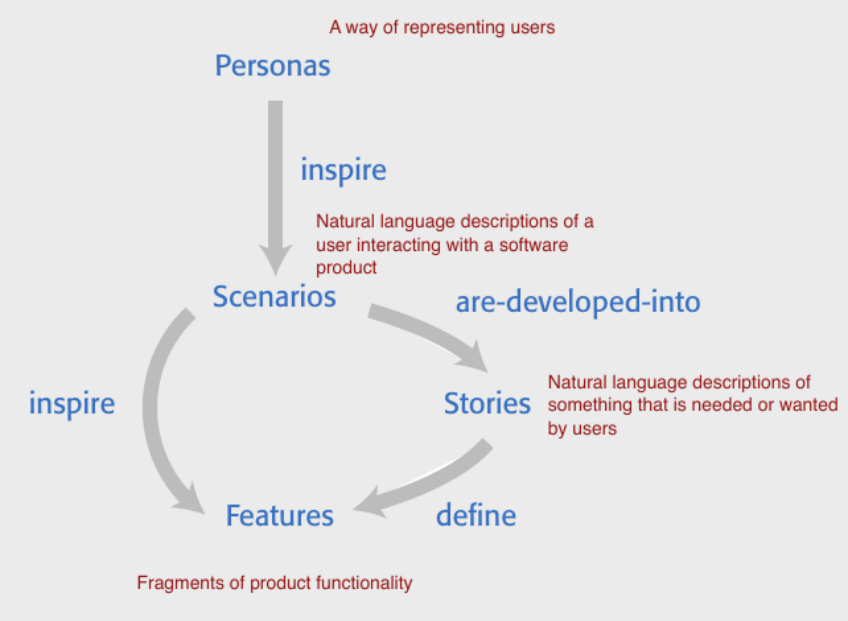


Figure 3.1: Scrum cycles

3.1 Personas

Who are the target users for our product? Need to understand potential users to design features useful for them by identifying the background, skills and experience of potential users. Personas are (imagined) types of product users: for a dentist agenda: dentist, receptionist, patient. Generally we need a few (1-2, max 5) personas to identify key product features.

- Personalization: You should give them a name and say something about their personal circumstances. This is important because you shouldn't think of a persona as a role but as an individual. 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 business, you should say something about their job and (if necessary) what that job involves. For some jobs, such as a teacher where readers are likely to be familiar with the job, this may not be necessary.
- Education: You should describe their educational background and their level of technical skills and experience. This is important, especially for interface design.
- Relevance: If you can, you should say why they might be interested in using the product and what they might want to do with it.

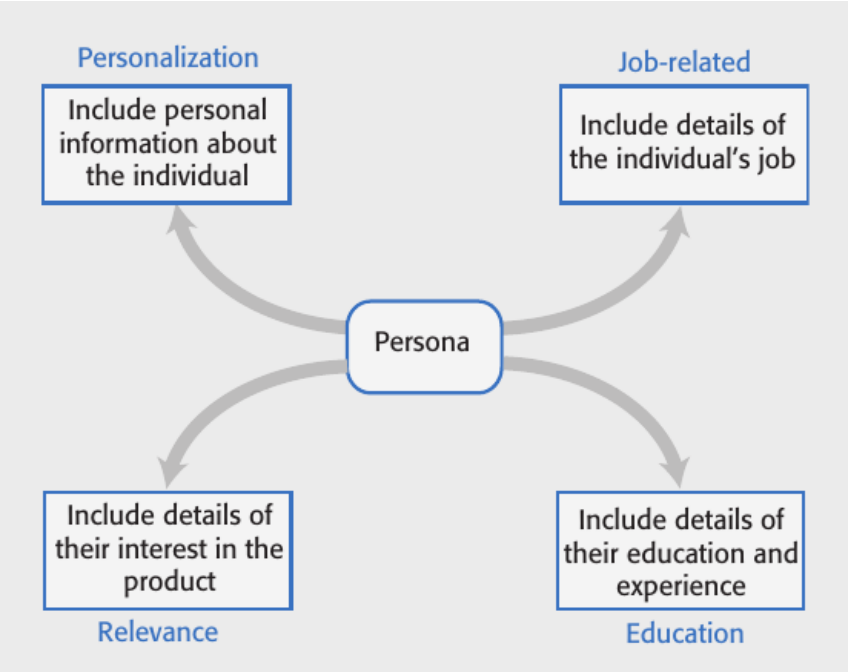


Figure 3.2: Scrum cycles

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 moreeffectively. Using photos is misleading:ersonas shouldn't be about how people look, but what they do. (Steve Cable) Detailed personas encouraged the team to assume that demographic information drove motivations. (Sara Wachter-Boettcher). Persona profiles with a smiling photo result in an increase in willingness to use the personas. (Joni Salminen et al.). Gender has a mediating influence on perceived attributes of males and females. (Francine M. Deutsch)

An example of persona that doesn't have a technical background and wants only a product that support administration is in 3.3.

Personas allow developers to "step into the users'shoes": when studying the users is not possible (e.g. for some new products) we can develop **proto-personas**.

3.2 Scenarios

A scenario allow to discover product features, we can define scenarios of user interactions with the product Scenario = narrative describing a situation in which a user is using our product's features to do something she wants to do. In figure 3.4 an example. So, a **scenario** is a narrative, high-level scenarios facilitate communication and stimulate design creativity. Scenarios are not specifications, though (they lack detail and may be incomplete). Generally 3-4 scenarios for each persona are suggested, allow to cover main responsabilities of persona. They're written from user's perspective: each team member should individyally create some scenarios, and then discuss them with rest of team and with user (if possible).

Emma, a history teacher
Emma, age 41, is a history teacher in a secondary school (high school) in Edinburgh. She teaches students from ages 12 to 18. She was born in Cardiff in Wales where both her father and her mother were teachers. After completing a degree in history from Newcastle University, she moved to Edinburgh to be with her partner and trained as a teacher. She has two children, aged 6 and 8, who both attend the local primary school. She likes to get home as early as she can to see her children, so often does lesson preparation, administration and marking from home.

Emma uses social media and the usual productivity applications to prepare her lessons, but is not particularly interested in digital technologies. She hates the virtual learning environment that is currently used in her school and avoids using it if she can. She believes that face-to-face teaching is most effective. She might use the iLearn system for administration and access to historic films and documents. However, she is not interested in a blended digital/face-to-face approach to teaching.

Figure 3.3: Scrum cycles

Fishing in Ullapool
Jack is a primary school teacher in Ullapool, teaching P6 pupils. He has decided that a class project should be focused around the fishing industry in the area, looking at the history, development and economic impact of fishing.

As part of this, students are asked to gather and share reminiscences from relatives, use newspaper archives and collect old photographs related to fishing and fishing communities in the area. Pupils use an iLearn wiki to gather together fishing stories and SCRAN (a history archive site) to access newspaper archives and photographs. However, Jack also needs a photo-sharing site as he wants students to take and comment on each others' photos and to upload scans of old photographs that they may have in their families. He needs to be able to moderate posts with photos before they are shared, because pre-teen children can't understand copyright and privacy issues.

Jack sends an email to a primary school teachers' group to see if anyone can recommend an appropriate system. Two teachers reply and both suggest that he uses KidsTakePics, a photo-sharing site that allows teachers to check and moderate content. As KidsTakePics is not integrated with the iLearn authentication service, he sets up a teacher and a class account with KidsTakePics.

He uses the the iLearn setup service to add KidsTakePics to the services seen by the students in his class so that, when they log in, they can immediately use the system to upload photos from their phones and class computers.

Figure 3.4: Scrum cycles

3.3 User stories

Differently from scenarios, an **user stories** are finer-grain narratives that set out in a more detailed and structured way a single thing that a user wants from a software system, as stated in 3.5. User stories should focus on a clearly defined system feature or aspect of a feature that can be implemented within a single sprint: if the story is about a more complex feature that might take several sprints to implement, then it is called an **epic**.

"As a system manager, I need a way to backup the system and restore either individual applications, files, directories or the whole system": there is a lot of functionality associated with this user story. For implementation, it should be broken down into simpler stories with each story focusing on a single aspect of the backup system.

You can develop user stories by refining scenarios: three user stories can be identified from the following fragment of Emma's scenario, as pictured in 3.5.

" From home, she logs onto the iLearn system using her Google account credentials. Emma has two iLearn accounts – her teacher account and a parent account associated with the local primary school. The system recognises that she is a multiple account owner and asks her to select the account to be used. She chooses the teacher account and the system generates her personal welcome screen. As well as her selected applications, this also shows management apps that help teachers create and manage student groups."

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 can

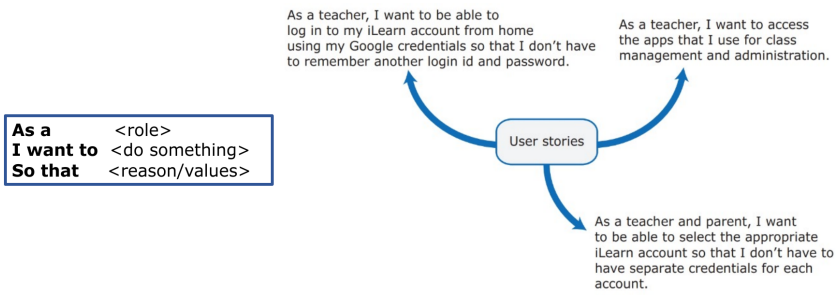


Figure 3.5: Scrum cycles

be read more naturally, make it easir to understand stories and provide more context.

3.3.1 Feature identification

Your aim in the initial stage of product design should be to create a list of features that define your product. A feature is a way of allowing users to access and use your product's functionality so the feature list defines the overall functionality of the system. Features should be:

- Independence: features 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 they should never have side-effects.
- Relevance: features should reflect the way that users normally carry out some task. They should not provide obscure functionality that is hardly ever required.

The **knowledge sources for features design** are multiple like:

- User knowledge: You can use user scenarios and user stories to inform the team of what users want and how they might use it the software features.
- Product knowledge: You may have experience of existing products or decide to research what these products do as part of your development process. Sometimes, your features have to replicate existing features in these products because they provide fundamental functionality that is always required.
- 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 do what they want to do.
- 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.

At the same time, the **factors** in feature set design are undirectly proportional each other as in 3.6.

3.3.2 Feature creep

Feature creep occurs when new features are added in response to user requests without considering whether or not these features are generally useful or whether they can be implemented in some other way. Too many features make products hard to use and understand and there are three reasons why feature creep occurs:

1. Product managers are reluctant to say 'no' when users ask for specific features.
2. Developers try to match features in competing products.
3. The product includes features to support both inexperienced and experienced users.

To avoid feature creep we need to answer the questions pictured in 3.7.

Features can be identified directly from the product vision or from scenarios. You can highlight phrases in narrative description to identify features to be included in the software.

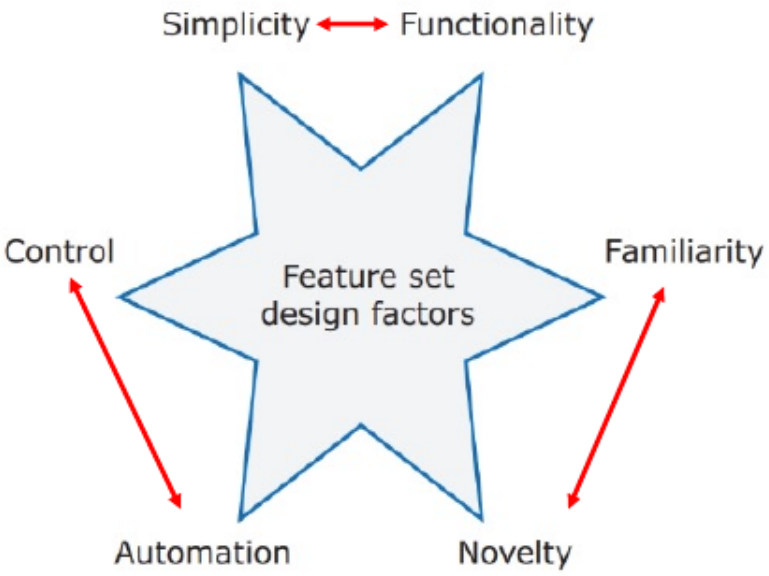


Figure 3.6: Scrum cycles

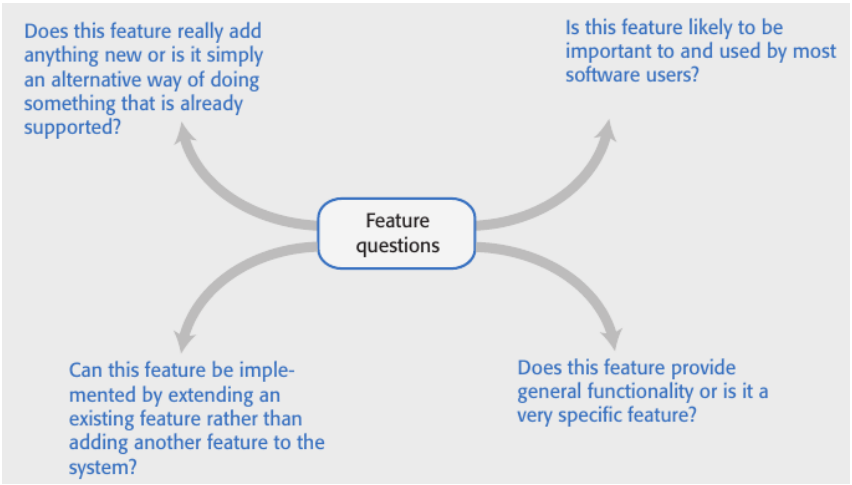


Figure 3.7: Scrum cycles

Chapter 4

Software Architecture

To create a reliable, secure and efficient product, you need to pay attention to architectural design which includes:

- its overall organization,
- how the software is decomposed into components,
- the server organization
- the technologies that you use to build the software.The architecture of a software product affects its performance, usability, security, reliability and maintainability.

There are many different interpretations of the term 'software architecture'. We present the **IEEE definition of 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 and evolution."*

4.0.1 Components

A component is an element that implements a coherent set of functionality or features. Software component can be considered as a collection of one or more services that may be used by other components: when designing software architecture, you don't have to decide how anarchitectural element or component is to be implemented.Rather, you design the component interface and leave theimplementation of that interface to a later stage of the development. An abstraction in 4.1.

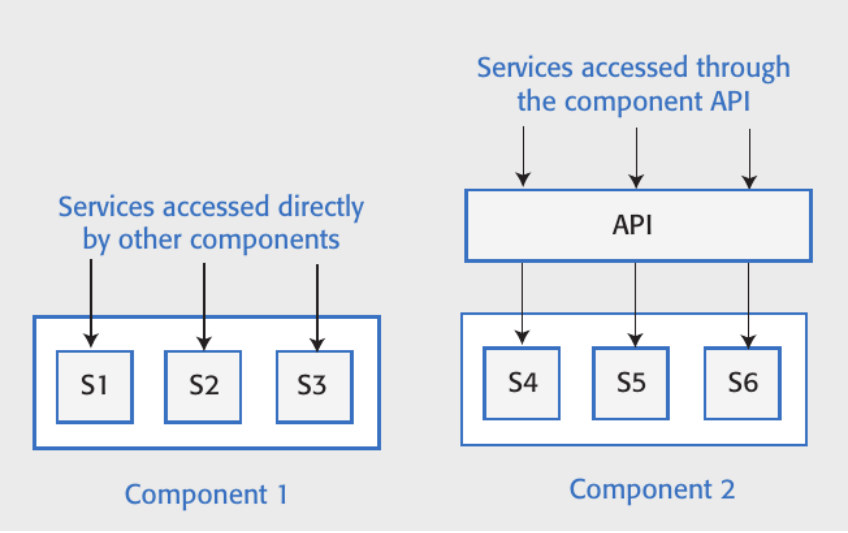


Figure 4.1: Scrum cycles

There are plenty of issues that influence the architectural decisions, pictured in 4.2:

- **Nonfunctional product characteristics**: Nonfunctional product characteristics such as security and performance affect all users. If you get these wrong, your product will is unlikely to be a commercial success. Unfortunately, some characteristics are opposing, so you can only optimize the most important.
- **Product lifetime**:If you anticipate a long product lifetime, you will need to create regular product revisions.You therefore need an architecture that is evolvable, so that it can be adapted to accommodate new features and technology.
- **Software reuse**: You can save a lot of time and effort, if you can 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.

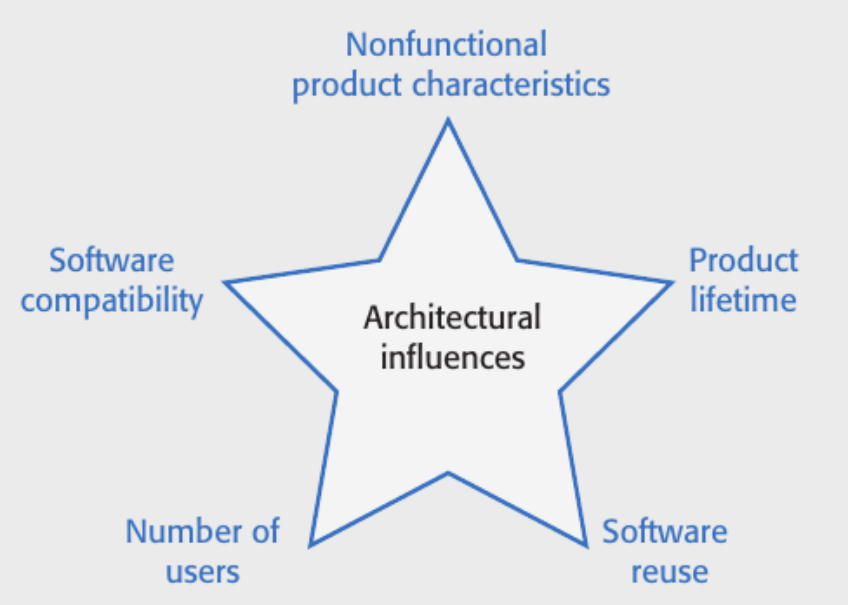


Figure 4.2: Scrum cycles

The main **non-functional quality attributes** needs to follow these questions to be effective:

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

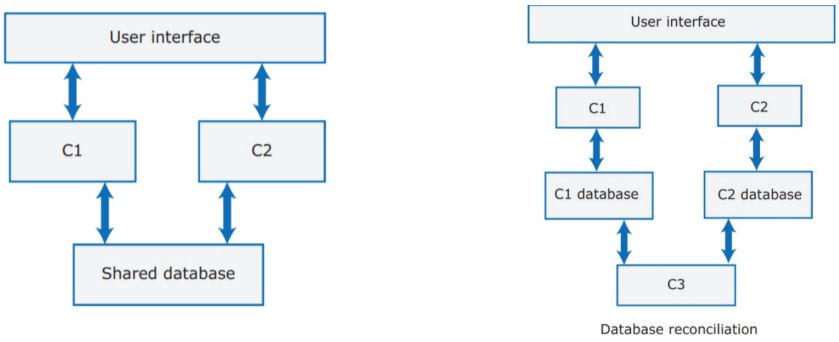


Figure 4.3: Scrum cycles

Overall, it's noticeavle to remember that optimizing one non-functional attribute affects others (*like security vs performance usability or availability vs time-to-market semplicity*). Let's examine **mantainability**: it's the difficult/expensiveness to make changes after release. It's a good practice to decompose system into small self-contained parts and avoid shared data structure. The figure 4.3 describe a scenario in which we have initially a **shared database** but if it fail then the two instances cannot work properly. We can split the database in two (*by some criteria that allows instances to operate almost independently*) and use a thid instance as a reconciliation component. So If one component needs to change the database organization, this does not affect the other component and the system can continue to provide partial service in the event of a database failure, surely at cost of implementing a mechanism for eventual data consistency.

4.0.2 System decomposition

Complexity in a system architecture arises because of the number and the nature of the relationships between components in that system. When decomposing a system into components, you should try to avoid unnecessary software complexity by **localize relationships**, if there are relationships between components A and B, these are easier to understand if A and B are defined in the same module and by **reducing shared dependencies** where components A and B depend on some other component or data, complexity increases because changes to the shared component mean youhave to understand how these changes affect both A and B. It is always preferable to use local data wherever possible and to avoid sharing data if you can, as pictured in 4.5. In figure 4.4 an example of component relationships.

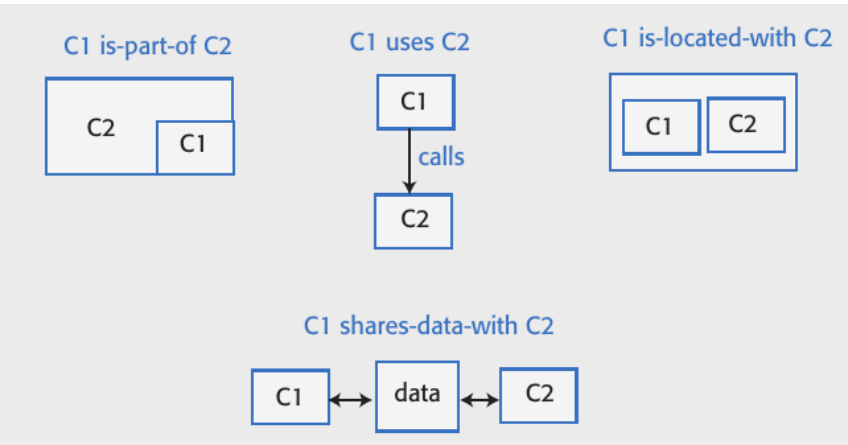


Figure 4.4: Scrum cycles

In a **layered architecture** each layer is an area of concern and is condiered 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 does not include implementation information, as stated in 4.6.

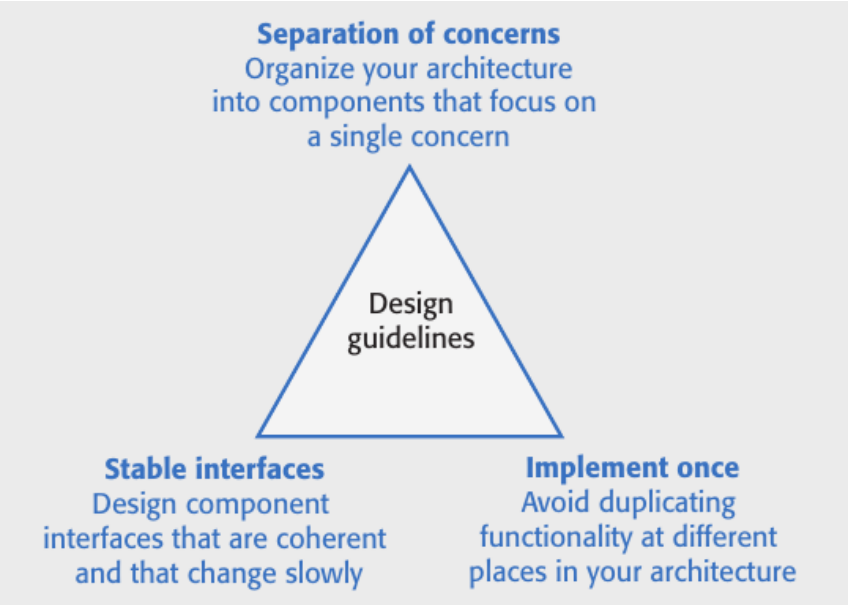


Figure 4.5: Scrum cycles

Cross-cutting concerns

Cross-cutting concerns are concerns that are systemic, that is, they affect the whole system: in a layered architecture, cross-cutting concerns affect all layers in the system as well as the way in which people use the system. Cross-cutting concerns are completely different from the functional concerns represented by layers in a software architecture.Every layer has to take them into account and there are inevitably interactions between the layers because of these concerns.The existence of cross-cutting concerns is the reason why modifying a system after it has been designed to improve its security is often difficult.

In a **Security architecture** different technologies are used in different layers, such as an SQL database or a Firefox browser. Attackers can try to use of vulnerabilities in these technologies to gain access. Consequently, you need protection from attacks at each layer as well as protection, at lower layers in the system, from successful attacks that have occurred at higher-level layers.If there is only a single security component in a system, this represents a critical system vulnerability. If all security checking goes through that component and it stops working properly or is compromised in an attack, then you have no reliable security in your system. By distributing security across the layers, your system is more resilient to attacks and software failure (remember the Rogue One example earlier in the chapter).

In a generic **web-based application** the layer functionality can be identified into:

1. Browser-based or mobile user interface: A web browser system interface in which HTML forms are often used to collect user input. Javascript components for local actions, such as input validation, should also be included at this level. Alternatively, a mobile interface may be implemented as an app.

2. Authentication and UI management: A user interface management layer that may include components for user authentication and web page generation.

3. Application-specific functionality: An 'application' layer that provides functionality of the application. Sometimes, this may be expanded into more than one layer.

4. Basic shared services:A shared services layer, which includes components that provide services used by the application layer components.

5. Database and transaction management: A database layer that provides services such as transaction management and recovery. If your application does not use a database then this may not be required.

Generally speaking, *system decomposition* must be done (partly) in conjunction with choosing technologies for your system: e.g choice of using relational database affects components at higher layers or choice of supporting interfaces on mobile devices calls for using corresponding UI development toolkits.

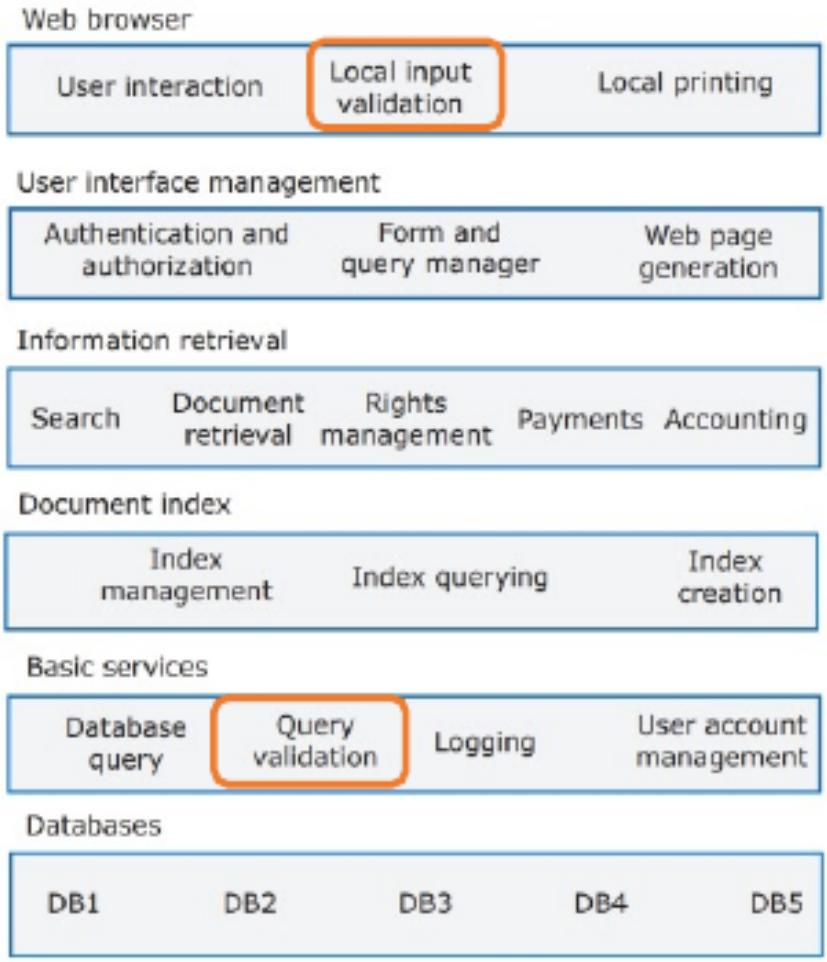


Figure 4.6: The orange highlight that concerns may not be always full separated in practice

4.1 Distribution architecture

The distribution architecture of a software system defines the servers in the system and the allocation of components to these servers. **Client-server architectures** are a type of distribution architecture that is suited to applications where clients access a shared database and business logic operations on that data.In this architecture, the user interface is implemented on the user's own computer or mobile device: functionality is distributed between the client and one or more server computers, like in 4.7.

In a client-server architecture, a widely used pattern is **MVC** or **Mode-View-Cntroller** that allows to model a decouple system from it's design phase. Each view registers with model so that if model changes, all views can be updated.

4.1.1 Client-server communication

Client-server communication normally uses the HTTP protocol. The client sends a message to the server that includes an instruction such as GET or POST along with the identifier of a resource (usually a URL) on which that instruction should operate. The message may also include additional information, such as information collected from a form. HTTP is a text-only protocol so structured data has to be represented as text. There are two ways of representing this data that are widely used, namely XML and JSON: XML is a markup language with tags used to identify each data item. Also JSON is a simpler representation based on the representation of objects in the Javascript language.

4.1.2 SOA - Service-Oriented Architecture

Services in a service-oriented architecture are stateless components, which means that they can be replicated and can migrate from one computer to another: many servers may be involved in providing services-

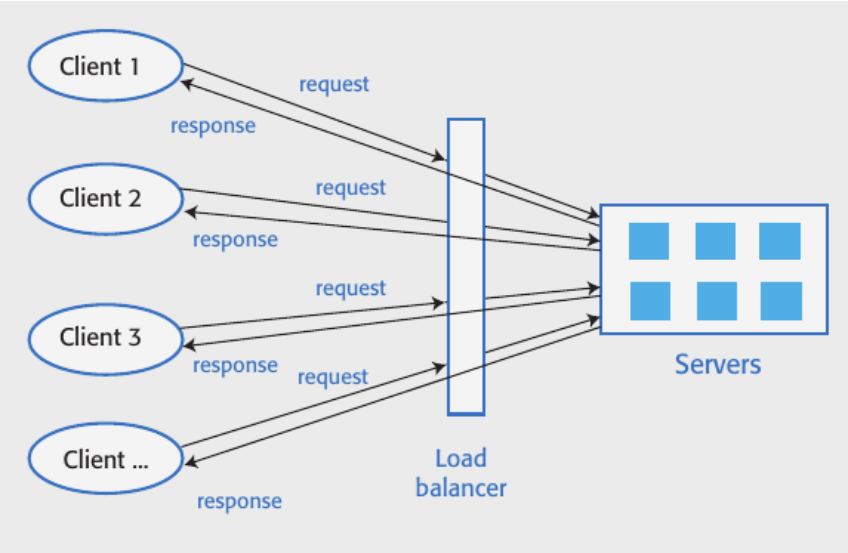


Figure 4.7: The orange highlight that concerns may not be always full separated in practice

A service-oriented architecture is usually easier to scale as demand increases and is resilient to failure.

To **choose a distribution architecture** we must considerate the following areas:

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

4.2 Technology choices

The technology choices must concentrate on component like *database, platform, server, open source software* and *development tools*. These choices affect and constrain the overall system architecture and it's difficult to change them during the development phase.

4.2.1 Database

There are two kinds of database that are now commonly used: Relational databases, where the data is organised into structured tables and NoSQL databases, in which the data has a more flexible, user-defined organization. Relational databases, such as MySQL, are particularly suitable for situations where you need transaction management and the data structures are predictable and fairly simple. NoSQL databases, such as MongoDB, are more flexible and potentially more efficient than relational databases for data analysis. NoSQL databases allow data to be organized hierarchically rather than as flat tables and this allows for more efficient concurrent processing of 'big data'.

4.2.2 Delivery platform

Delivery can be as a web-based or a mobile product or both. On Mobile platform we can encounter issues like intermittent connectivity so you must be able to provide a limited service without network connectivity. Another issue can be the lack of processing power so you need to minimize computationally-intensive operations or the power management issue in which the life's battery is limited so you should try to minimize the power used by your application.

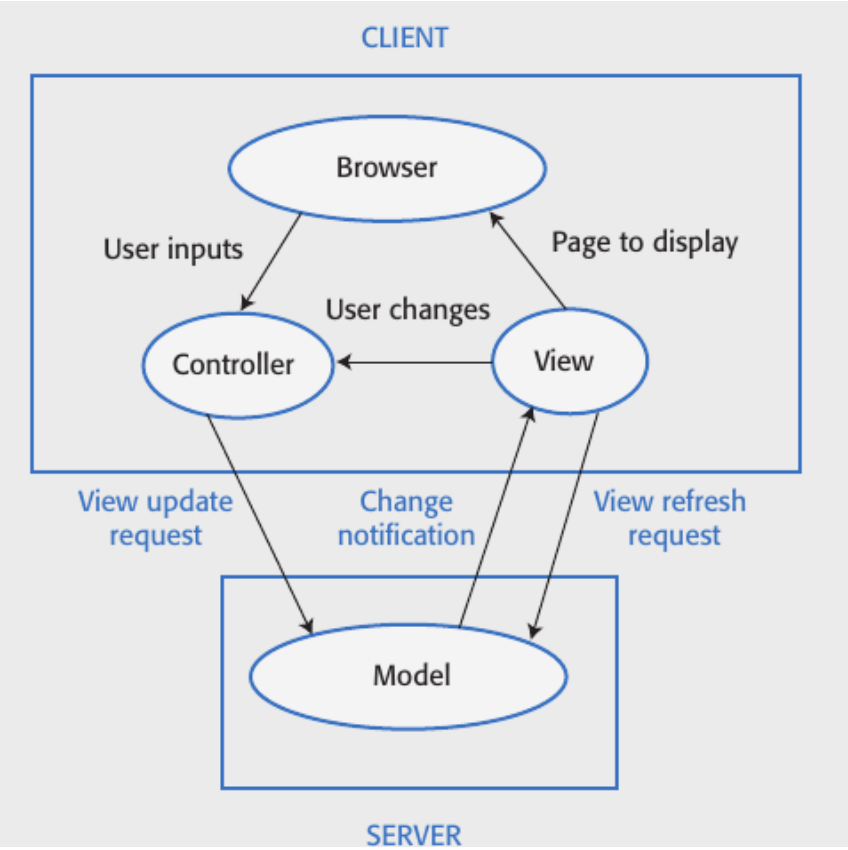


Figure 4.8: MVC Model

To deal with these differences, you usually need separate browser-based and mobile versions of your product front-end. You may need a completely different decomposition architecture in these different versions to ensure that performance and other characteristics are maintained.

4.2.3 Server

A key decision that you have to make is whether to design your system to run on customer servers or to run on the cloud. For consumer products that are not simply mobile apps I think it almost always makes sense to develop for the cloud but business products, it is a more difficult decision.

Some businesses are concerned about cloud security and prefer to run their systems on in-house servers. They may have a predictable pattern of system usage so there is less need to design your system to cope with large changes in demand. An important choice you have to make if you are running your software on the cloud is which cloud provider to use to avoid the **vendor lock-in** problem.

4.2.4 Development technology

Development technologies (e.g. mobile development toolkit, web application framework) influence the architecture of your product (*e.g. many web development frameworks assume use of model-view-controller architectural pattern*). Also, the development technology that you use may indirectly influence the architecture of your product (*e.g. if your team is used to relational databases then switching to a non-relational one can be a slow process*).

4.3 EIP - Enterprise Integration Patterns

The enterprise applications distributed around the world are composed by heterogeneous services: some of them are sources of data, some are storing services or computing services. Under a develpment view, on some services we can act by modifying the structure but, usually, we don't have much control on some services (*like a MongoDB instance*). This implies that in a software ecosystem we can have different data

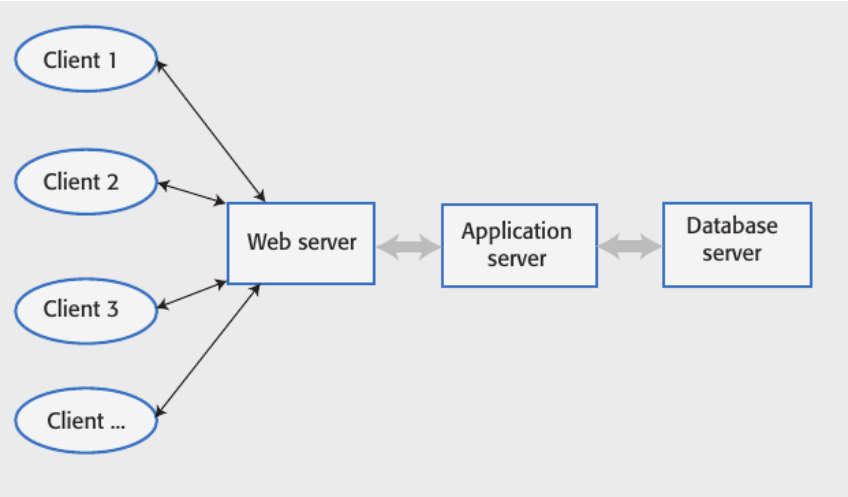


Figure 4.9: Multi-tier client-server architecture

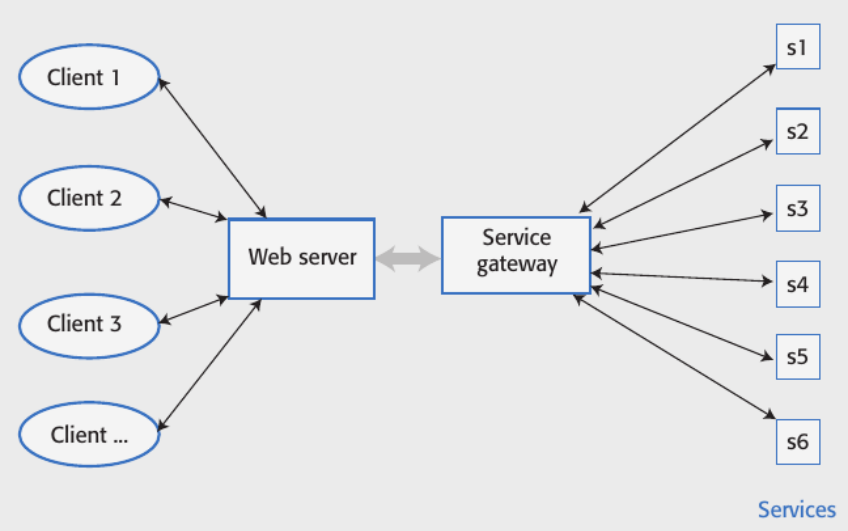


Figure 4.10: SOA model

types with different representations also for the data within the same context. So, enterprise applications are **compelx distributed multi-service applications** whose services must play together, being them suitably integrated

The **architectural** question is how to integrate multiple different services to realize enterprise applications that are *coherent (by understanding each other), extensibile, maintanable* and *reasonably simple to understand*. This is the key idea behind the **enterprise application integration patterns** that allows to manage the complexity behind every system, tolerate the changes and be a unified model by introducing **pattern based** methods.

4.4 Patterns

A **pattern** is an high-level abstraction of accepted, reusable solutions to recurring problems. Typically, patterns are given in terms of the following schema:

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

When facing a problem, considering existing patterns that are applicable to solve such problem saves us from re-inventing the wheel and making the same mistakes as others.

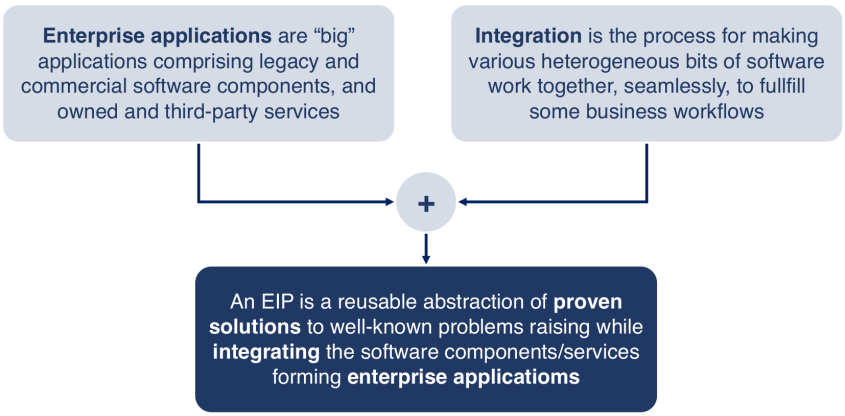


Figure 4.11: EIPs context model

EIPs allows different paradigms along the systems componenets: there are 65 integration patterns described in *Gregor Hohpe, Bobby Woolf,* ***Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions***, describing how each component can be integrated in relation with it's functional and non-functional requirements. The schema in 4.12 list a portion of them.

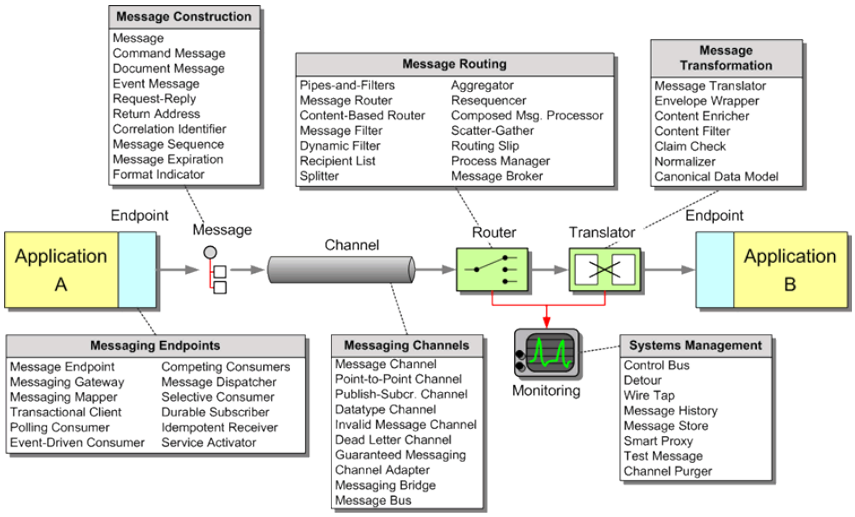


Figure 4.12: EIPs component based patterns

4.4.1 Messages

All systems need to communicate each other: internally, the same happens for componenets and subsystems. This evolves around a **message** that is a discrete piece of data sent from a service to another. It's typically structured into header (*considered as metadata*) and body (*the real data payload*). We can distinguish three types of messages according to picture 4.13.

Figure 4.13: First message content classification

The message-based communication allows to create services that are *loosely coupled* by using a simple exchange pattern (*a one-way pattern between sender and receiver*). The messages's format and metadata can also be independent of the integrated services. The communication is realized via **channels**: it's an abstraction for components sending messages from a source to destination. Under the hood, they can be implemented in many different ways (*RPC, HTTP, TCP, etc*). In 4.14 an abstracted schema. The channels are **one-way** so communications are natively **asynchronous**: to realize a synchronous

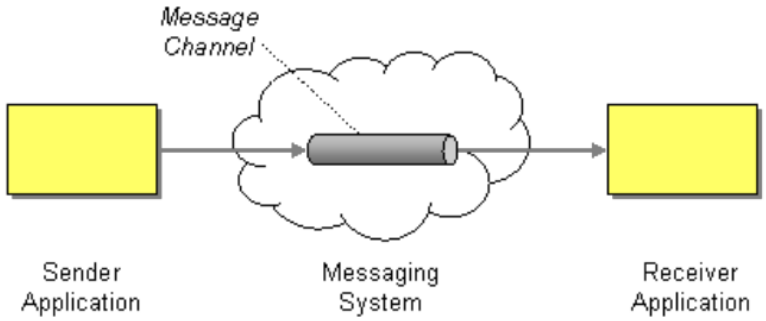


Figure 4.14: One-way communication with channel

communication we need to use two different (*"parallel"*) channels.

We can also distinguish in different types of channels: a **point-to-point** (4.15a) channel and a **publish-subscribe** (4.15b) one. The first one allow to ensure that only one receiver will receive a particular message. The messages's order is not stricly imposed and can be dependent upon the specific functional features to be implemented. The second one differently allows to deliver a copy of incoming messages to each receiver.

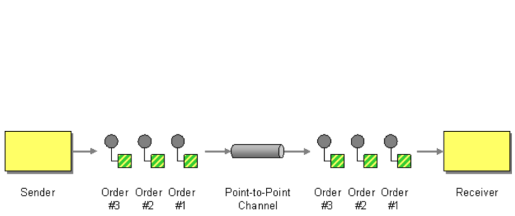
4.4.2 Binding messages

Application services are typically independent of the messaging systems: they needs some *binding* components to abstract the specific details of the messaging system between the two services. In 4.16a we present an **adapter** that enable application-specific data to be sent to channels. Typically it's used for the business logic layer/data manipulation. In 4.16b we present a **message endpoint** that enable application services to send/receive messages to/from channels.

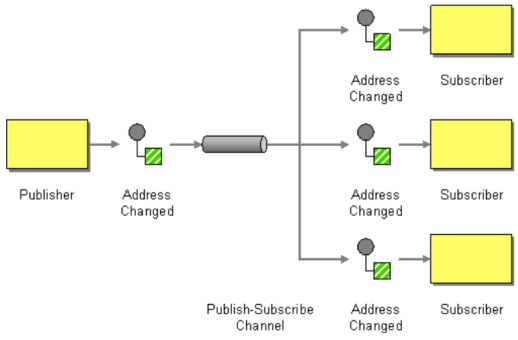
A simple integration is represented by using message endpoints and channels: the message endpoints is used by services to send/receive messages, channels allows to transport messages from a service to another.

4.4.3 Message transformation

Sending and receiving services may expect different data types or formats both for the header and body. So we introduce a **message translator** to be able to decode the correct/expected message format. The

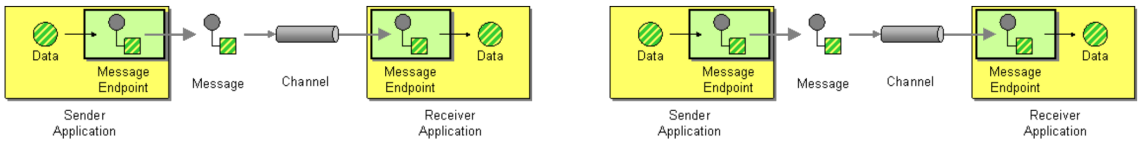


(a) Point-topoint



(b) Publish-subscribe

Figure 4.15: Channel types



(a) Point-topoint (b) Publish-subscribe

Figure 4.16: Binding components

schema in 4.17 describe the message transformation by convert a message from a *red-one* type to a *green* one.

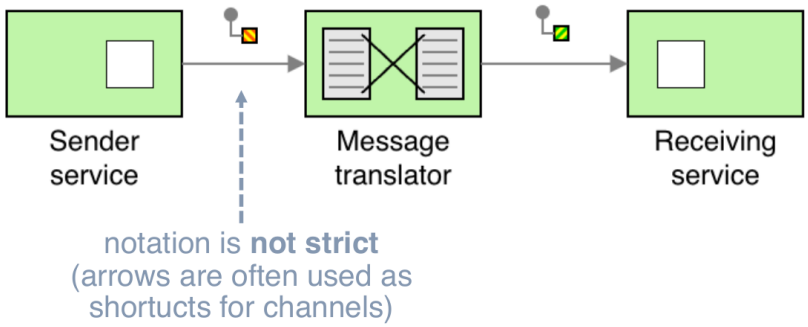


Figure 4.17: Message translator pattern

Message translators enable transforming messages, but other other steps may need to take place For instance, we need to reasonate about:

- How to route messages to different/multiple endpoints? (*e.g. to allow data differentiate computation*)
- How to split messages? (*e.g in case of a large JSON body*)
- How to aggregate messages?

4.4.4 Pipes and filters

The pipes and filters architectural style (plus other EIPs) enable structuring the more complex integration needed by enterprise applications: messages pass through multiple steps/components processing it (the filters). These components send message down the channels (pipes) they are connected to. Note that pipes and filters all deal with the same message/channel abstraction, and can be composed flexibly depending on the circumstances.

Let's see a navigated example called **Loan Broker** example in 4.18. The incoming requests are indicated by the green messages, the responses are the red messages: we'll introduce the unknown components marked with *?* following.

The first component along the path is the **component enricher** (4.19): when sending messages from a service to another, the target service may require more information than the source service can provide. The content enricher uses information inside the incoming message (*e.g. key fields*) to retrieve data from an external source. The retrieved data is appended to messages, typically by extending the header/trailer of the message or by encapsulating it inside another message representation. Original information may be carried over or no longer needed, depending on the specific needs of the receiving service. Another component is the **message routing** (*in 4.18 the Recipient List*) that allows a form of *dynamic routing* of messages based on some criteria. This component is needed in a context where endpoint can change or flexibility among the semantics of the messages is required. Some criteria on which we can base the routing are **content-based** in which we can route messages basaed on message type (*in headers*) or message content (*in body*); or **context-based** that can route messages based on contextual information (*e.g. a level 7 Load Balancer*). Some routing pattern are listed in 4.20. In general, a message router is connected to multiple channels and contains the logic to decide which channel it should send to.

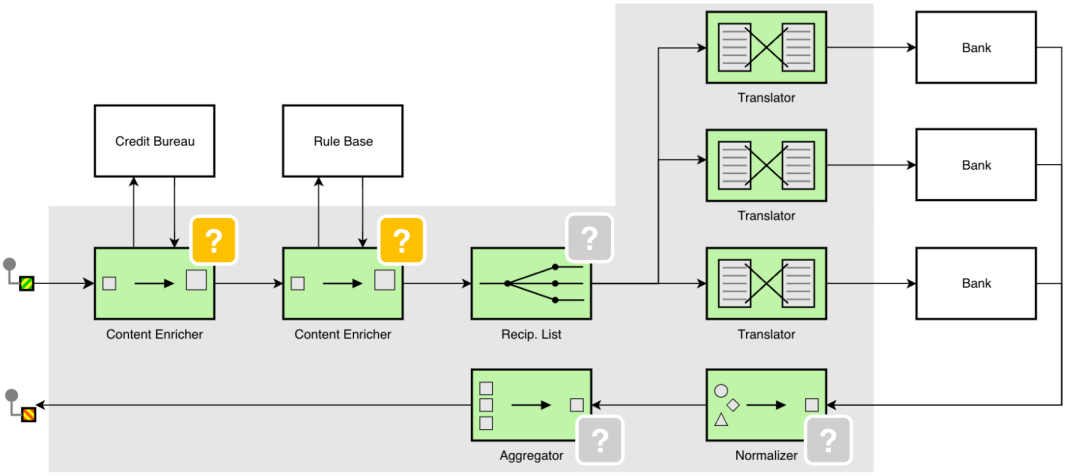


Figure 4.18: Loan Broker example schema

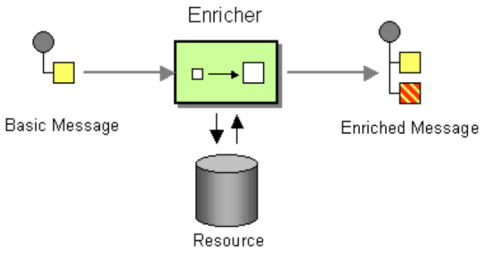


Figure 4.19: Loan Broker example schema

Let's examine the **message filtering** listed in 4.20: suppose that an enterprise applications sells gadget and widgets, also sending price changes/promtions to large customers. A message filter can eliminate undersired messages froma channel besd ona given criteria, for example if a customer is interested only in the widgets items. Given a message passed throught the message filter, we have only one output channel: if a message matches the given criteria, it's routed to the output channel; otherwise is discarded. A **recipient list** inspects an incoming message, determines the list of desired recipients and forwards the message to all channels addociated with the recipients in the list, as pictured in 4.21a.

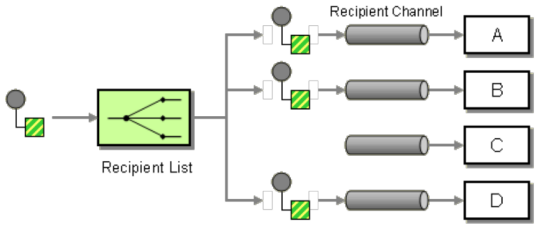
Following the Load Broker example 4.18 we introduce the **normalizer**: enable translating messages to match a common data format. It can be realized as a composition of other EIPs like a message router routing incoming messages to the most suited translator and a set of **translator** that transform incoming messages to a common format, as in 4.22.

The two last component we introduce works in opposite directions. Rather than defining long sequential integrations, some integration steps may go in **parallel**, for example when sending the loan request to multiple different banks. These are independent processes which can be **splitted** and executed in parallel: the result of such processes can be **aggregated** and decisions are made on how to proceed. A **splitter** enables breaking out hte composite message into a series of individual messages: each output message contains a different portion of the original message. Also, each output message can be processed differently from others by exploiting content-nased routers to ship them to different processing chains. A stateful **aggregator** enables collecting and storing individual messages until a complete set of related messages has been received: then publishes a single message distilled from the individual messages. The published message can be processed as a whole.

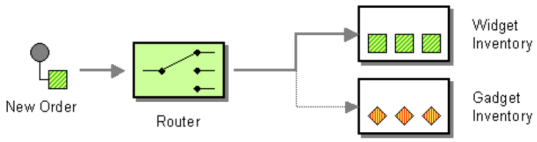
Message routing pattern	Consumed msgs	Published msgs	Stateful?
Message Filter	1	0 or 1	No*
Content-based Router	1	1	No*
Recipient List	1	multiple (incl. 0)	No
Splitter	1	multiple	No
Aggregator	multiple	1	Yes

* typically stateless, but sometimes could be stateful
(we also discuss a concrete example later)

Figure 4.20: Routing pattern



(a) Recipient list pattern



(b) Content-based router

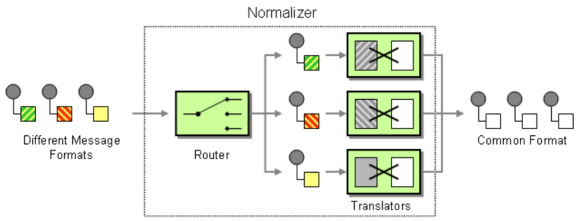
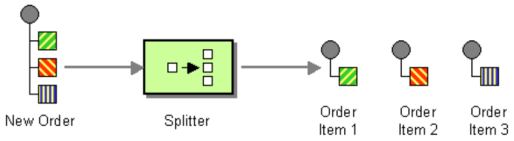
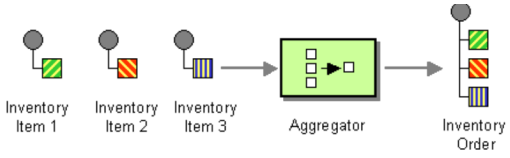


Figure 4.22: Normalizer pattern



(a) Splitter



(b) Aggregator

Chapter 5

Cloud-based Software

The cloud is made up of very large number of remote servers that are offered for rent by companies that own these servers: **Cloud-based servers** are *'virtual servers'*, which means that they are implementedin software rather than hardware. You can rent as many servers as you need, run your software on these servers and make them available to your customers: cloud servers can be started up and shut down as demand changes. The main advantages, as pictued in **??** are:

- **Scalability**: reflects the ability of your software to cope with increasingnumbers of users.As the load on your software increases, your software automatically adapts so that the system performance and response time is maintained.
- **Elasticity**: is related to scaleability but also allows for scaling-down as well as scaling-up. That is, you can monitor the demand on your application and add or remove servers dynamically as the number of users change.
- **Resilience**: means that you can design your software architecture to tolerate server failures. You can make several copies of your software concurrently available. If one of these fails, the others continue to provide a service.
- **Cost**:You avoid the initial capital costs of hardware procurement, shifting to a *pay-per-use* paradigm.

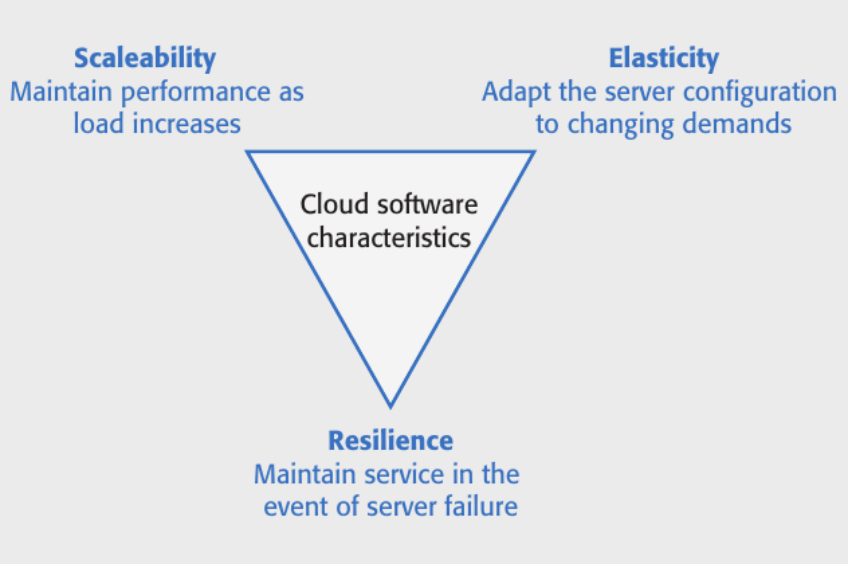


Figure 5.1: Scalability, resilience and elasticity triangle

5.1 Virtualization: from VMs to containers

If you are running a cloud-based system with many instances of applications or services, these all use the same operating system, you can use a simpler virtualization technology called 'containers'. Using containers accelerates the process of deploying virtual servers on the cloud: containers are usually megabytes in

size whereas VMs are gigabytes and also containers can be started and shut down in a few seconds rather than the few minutes required for a VM. Containers are an operating system virtualization technology that allows independent servers to **share a single operating system**. They are particularly useful for providing isolated application services where each user sees their own version of an application. An abstraction in 5.2.

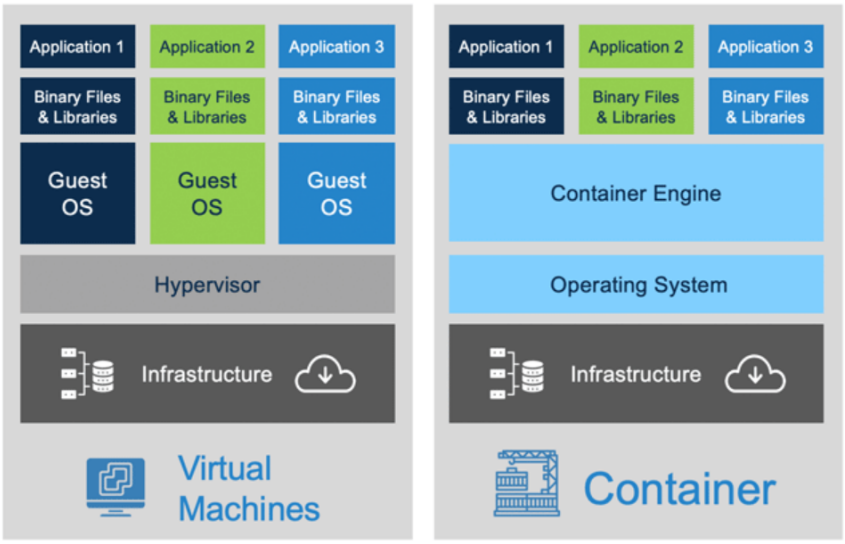


Figure 5.2: VMs VS Containers

Containers were developed by Google around 2007 but containers became a mainstream technology around 2015. **Docker** is one of the most known containerization technology in the world:

Docker is a platform that allows us to run applications in an isolated environment
Docker allows us to develop and run portable applications by exploiting containers

Containerization technologies came from the past: UNIX *chroot* command provided a simple form of filesystem isolation as the FreeBSD's *jail* utility that extended chroot in 1998 to sandbox processes. In 2005 Google started developing **cgroups** for Linux kernel and began moving its infrastructure to containers. Only in 2008 the *Linux Containers* (LXC) provided a compelte containerization solution. Subsequently evolution happens in 2013 when Docker addded the **portable images** and *firendly UI*.

So, to recap, Docker is a container management system that allows users to define the software to be included in a container as a Docker image. It also includes a run-time system that can create and manage containers using these Docker images.

Docker container system is composed by (5.3):

- *Docker daemon*: This is a process that runs on a host server and is used to setup, start, stop, and monitor containers, as well as building and managing local images.
- *Docker client*: This software is used by developers and system managers to define and control containers
- *Dockerfiles*: Dockerfiles define runnable applications (images) as a series of setup commands that specify the software to be included in a container. Each container must be defined by an associated Dockerfile.
- *Image*: A Dockerfile is interpreted to create a Docker image, which is a set of directories with the specified software and data installed in the right places. Images are set up to be runnable Docker applications.
- *Docker Hub*: This is a registry of images that has been created. These may be reused to setup containers or as a starting point for defining new images.
- *Containers*: Containers are executing images. An image is loaded into a container and the application defined bby the image starts execution. Containers may be moved from server to server without modification and replicated across many servers. You can make changes to a Docker container (e.g. by modifying files) but you then must commit these changes to create a new image and restart the container.

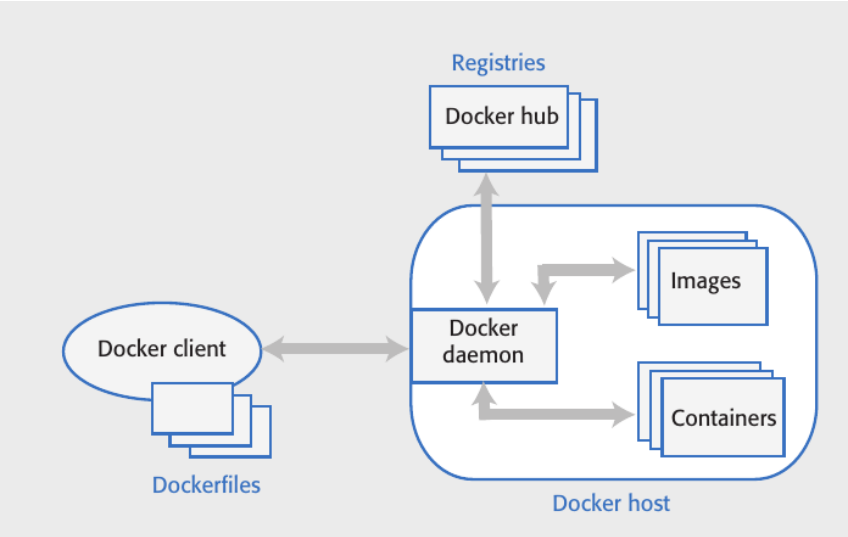


Figure 5.3: Docker container system

It's noticeable to remember that containers are **ephimeral** so, to provide persistecy, we need to use **volumes** that can be mounted to running containers.

5.1.1 Docker images

Docker images are directories that can be archived, shared and run on different Docker hosts. Everything that's needed to run a software system - binaries, libraries, system tools, etc. is included in the directory. **Images** are read-only template used to create containers. A Docker image is a base layer, usually taken from the Docker registry (*which can be a* ***private*** *or* ***public*** *one*), with your own software and data added as a layer on top of this: the layered model means that updating Docker applications is fast and efficient. Each update to the filesystem is a layer on top of the existing system. To change an application, all you have to do is to ship the changes that you have made to its image, often just a small number of files. Image can also be identified by `repository:tag` pairs. In figure 5.4 the commands flow, starting from a `Dockerfile` to a running container.

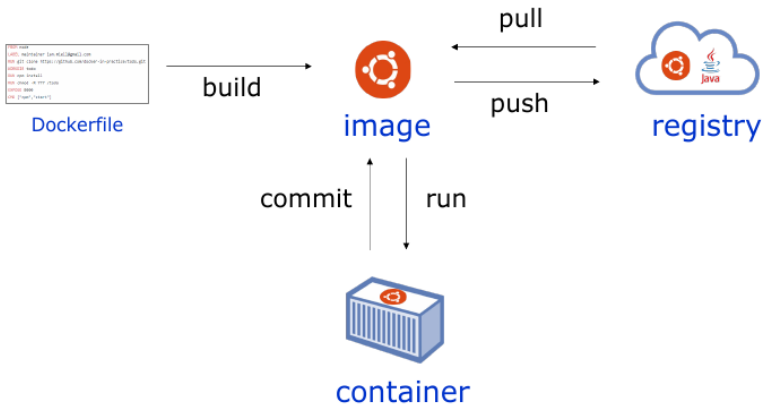


Figure 5.4: Docker commands flow

5.1.2 Advantages of Docker

They solve the problem of software dependencies. You don't have to worry about the libraries and other software on the application server being different from those on your development server.

Instead of shipping your product as stand-alone software, you can ship a container that includes all of the support software that your product needs. They provide a mechanism for software portability across

different clouds. Docker containers can run on any system or cloud provider where the Docker daemon is available. They provide an efficient mechanism for implementing software services and so support the development of service-oriented architectures. They simplify the adoption of DevOps: this is an approach to software support where the same team are responsible for both developing and supporting operational software.

5.1.3 Docker Compose

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

Compose works in all environments: production, staging, development, testing, as well as CI workflows. It also has commands for managing the whole lifecycle of your application:

- Start, stop, and rebuild services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service

We briefly present some useful commands to remember:

- `‘docker-compose up‘` is the command to deploy a Compose app. It expects the Compose file to be called docker-compose.yml or docker-compose.yaml, but you can specify a custom filename with the -f flag.It's common to start the app in the background with the -d flag.
- `‘docker-compose stop‘` will stop all of the containers in a Compose app without deleting them from thesystem. THe app can be easily restarted with docker-compose restart.
- `‘docker-compose rm‘` will delete a stopped Compose app. It will delete containers and networks, but it will not delete volumes and images.
- `‘docker-compose restart‘` will restart a Compose app that has been stopped with docker-compose stop.If you have made changes to your Compose app since stopping it, these changes will not appear in the restarted app. You will need to re-deploy the app to get the changes.
- `‘docker-compose ps‘` will list the container in the Compose app. It shows current state, the command that one is running, and network ports.
- `‘docker-compose down‘` will stop and delete a running Compose app. It deletes containers and networks, but not volumes and images

Extensive example and documentation can be found here:

- Microsoft - Defining your multi-container application with docker-compose.yml
- Docker Compose Documentation - Key features and use cases

5.2 Everything as a service

The idea of a service that is rented rather than owned is fundamental to cloud computing:

- Infrastructure as a service (IaaS): Cloud providers offer different kinds of infrastructure service such as a compute service, a network service and a storage service that you can use to implement virtual servers. (*e.g. salesforce.com*)
- Platform as a service (PaaS): This is an intermediate level where you use libraries and frameworks provided by the cloud provider to implement your software. These provide access to a range of functions, including SQL and NoSQL databases. (*e.g. Heroku, Azure, GAE*)
- Software as a service (SaaS): Your software product runs on the cloud and is accessed by users through a web browser or mobile app. (*EC2, S3*)

5.2.1 SaaS - Software as a Service

Software products were initially installed on customers' computers: customers had to configure software, deal with software updates and product company had to maintain different product versions. SaaS differently allows to deliver product as a service in which customers do not have to install software, they pay a subscription and access product from remote.

Benefits of SaaS for product providers are:

- *Cash flow*: with a regular cash flow customers pay periodic subscription or pay-per-usage
- *Update management*: You control product updates, customers receive update at same time so no need of simulateneously maintaining several versions. This allows to reduce costs
- *Continous deployment*:You can deploy new software versions as soon as changes have been made and tested
- *Payment flexibility*:Different payment options can attract wider range of customers e.g. small companies or individuals can avoid paying large upfront software costs
- *Try before you buy*:You can make early free/low-cost product available and get customer feedback
- *Data collection*:You can easily collect data on product usage and customers

Altought there are some cons for customers like:

- Privacy regulation conformance: e.g. EU countries have strict laws on storage of personal info
- Security concerns: Customers may not want to pass data control to external provider
- Network constraints: Can limit response time when much data transfer
- Data exchange: Can be difficult if cloud does not provide suitable API
- Loss of control over updates
- Service updates

Some issues whine designing for a SaaS product are:

- Local vs remote processing: some featues can be executed only locally: this can reduce network traffic, increasing response speed but can also increase consumption for battery-powered devices
- Authentication: product own authentication system vs. federated authentication vs. Third-part authentication (*Google, Linkedin, etc*).
- Information leakage: security risks with multiple users from multiple organizations
- Multi-tenant vs multi-instance database management: single repository vs separate copies of system and database

5.2.2 Multi-tenant systems

A multi-tenant database is partitioned so that customer companies have their own space and can store and access their own data. There is a single database schema, defined by the SaaS provider, that is shared by all of the system's users: items in the database are tagged with a tenant identifier, representing a company that has stored data in the system. The database access software uses this tenant identifier to provide 'logical isolation', which means that users seem to be working with their own database.

Mid-size and large businesses rarely want to use generic multi-tenant software, often prefer a customized version adapted to their own requirements by customizing:

- *Authentication*: Businesses may want users to authenticate using their business credentials rather than the account credentials set up by the software provider. I explain, in Chapter 7, how federated authentication makes this possible.
- *Branding*: Businesses may want a user interface that is branded to reflect their own organisation.
- *Business rules*: Businesses may want to be able to define their own business rules and workflows that apply to their own data.

- *Data schemas*: Businesses may want to be able to extend the standard data model used in the system database to meet their own business needs.
- *Access control*: Businesses may want to be able to define their own access control model that sets out the data that specific users or user groups can access and the allowed operations on that data.

UI configurability can be implemented by employing user profiles: users asked to select their organization or provide their business email address. Product uses profile information to create personalized version of interface like generic company name and logo replace with company's ones or some features of menus are disabled (as in 5.5).

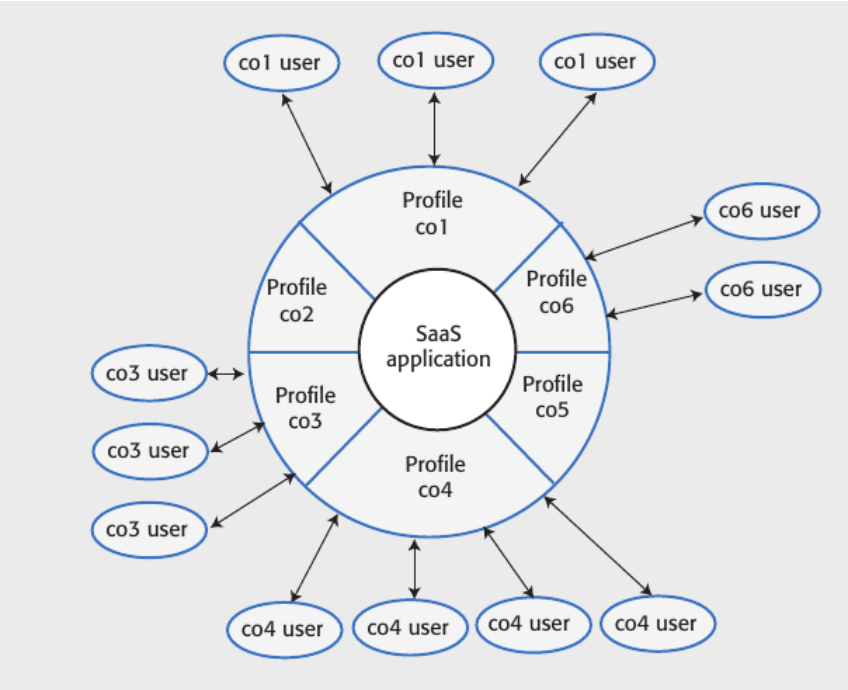


Figure 5.5: User profiles for SaaS access

To **adapt the DB schema** in a multi-tenant system we can evaluate two different solution:

1. *Solution 1*: You add some extra columns to each database table and define a customer profile that maps the column names that the customer wants to these extra columns. However it is difficult to know how many extra columns you should include. If you have too few, customers will find that there aren't enough for what they need to do.

   If you cater for customers who need a lot of extra columns, however, you will find that most customers don't use them, so you will have a lot of wasted space in your database. Different customers are likely to need different types of columns. Picture 5.6a.

2. *Solution 2*: allow customers to add any number of additional fields and to define the names, types and values of these fields. The names and types of these values are held in a separate table, accessed using the tenant identifier.Unfortunately, using tables in this way adds complexity to the database management software. Extra tables must be managed and information from them integrated into the database. Picture 5.6b

Information from all customers is stored in the same database in a multi-tenant system so a software bug or an attack could lead to the data of some or all customers being exposed to others. **Key security issues** are multilevel access control and encryption.

Multilevel access control means that access to data must be controlled at both the organizational level and the individual level. You need to have organizational level access control to ensure that any database operations only act on that organization's data. The individual user accessing the data should also have their own access permissions.

Encryption of data in a multitenant database reassures corporate users that their data cannot be viewed by people from other companies if some kind of system failure occurs.

Stock management								
Tenant	Key	Item	Stock	Supplier	Ordered	Ext 1	Ext 2	Ext 3
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			

(a) Solution 1

Tab1

Stock management						
Tenant	ID	Item	Stock	Supplier	Ordered	Ext 1
T516	100	Widg 1	27	S13	2017/2/12	E123
T632	100	Obj 1	5	S13	2017/1/11	E200
T973	100	Thing 1	241	S13	2017/2/7	E346
T516	110	Widg 2	14	S13	2017/2/2	E124
T516	120	Widg 3	17	S13	2017/1/24	E125
T973	100	Thing 2	132	S26	2017/2/12	E347

Tab2

Field names		
Tenant	Name	Type
T516	'Location'	String
T516	'Weight'	Integer
T516	'Fragile'	Bool
T632	'Delivered'	Date
T632	'Place'	String
T973	'Delivered'	Date

Tab3

Field values		
Record	Tenant	Value
E123	T516	'A17/S6'
E123	T516	'4'
E123	T516	'False'
E200	T632	'2017/1/15'
E200	T632	'Dublin'
E346	T973	'2017/2/10'

(b) Solution 2

Multi-instance systems are SaaS systems where each customer has its own system that is adapted to its needs, including its own database and security controls. Multi-instance, cloud-based systems are conceptually simpler than multi-tenant systems and avoid security concerns such as data leakage from one organization to another.There are two types of multi-instance system:

- **VM-based** multi-instance systems are multi-instance systems where the software instance and database for each customer runs in its own virtual machine. All users from the same customer may access the shared system database.
- *Container-based* multi-instance systems* These are multi-instance systems where each user has an isolated version of the software and database running in a set of containers.

Some pros of multi-instance databases are:

- Flexibility: Each instance of the software can be tailored and adapted to a customer's needs. Customers may use completely different database schemas and it is straightforward to transfer data from a customer database to the product database.
- Security: Each customer has their own database so there is no possibility of data leakage from one customer to another.
- Scaleability: Instances of the system can be scaled according to the needs of individual customers. For example, some customers may require more powerful servers than others.
- Resilience: If a software failure occurs, this will probably only affect a single customers. Other customers can continue working as normal.

Also, the disadvantages are:

- Cost: It is more expensive to use multi-instance systems because of the costs of renting many VMs in the cloud and the costs of managing multiple systems. Because of the slow startup time, VMs may have to be rented and kept running continuously, even if there is very little demand for the service.
- Update management: It is more complex to manage updates to the software because many instances have to be updated. This is particularly problematic where individual instances have been tailored to the needs of specific customers.

5.2.3 Architectural decisions

First, we analyze **DB organization**. there are three possible ways of providing a customer database in a cloud-based system:

1. As a multi-tenant system, shared by all customers for your product. This may be hosted in the cloud using large, powerful servers.

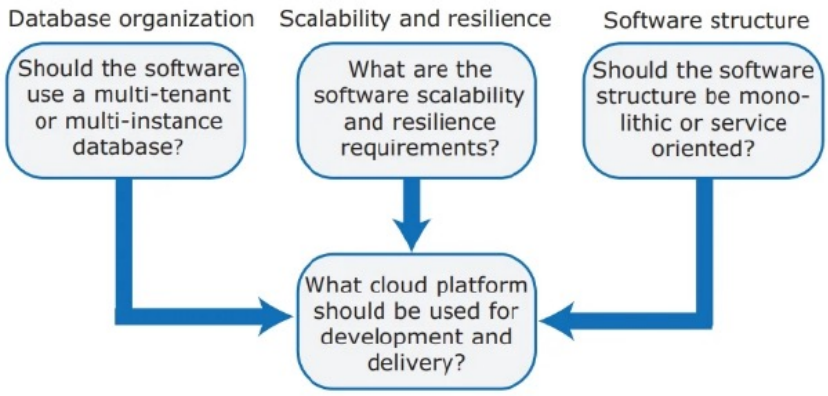


Figure 5.7: Architectural decision diagram

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

The factors to observe regarding the choice are:

- Target customers: Do customers require different database schemas and database personalization? Do customers have security concerns about database sharing? If so, use a multi-instance database.
- Transaction requirements: Is it critical that your products support ACID transactions where the data is guaranteed to be consistent at all times? If so, use 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 are there between database items? A multi-tenant model is usually best for very large databases as you can focus effort 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 multitenant database? What software support will they expect to do the data transfer? If customers have many different schemas, a multi-instance database should be used.
- System structure: Are you using a service-oriented architecture for your system? Can customer databases be split into a set of individual service databases? If so, use containerized, multi-instance databases.

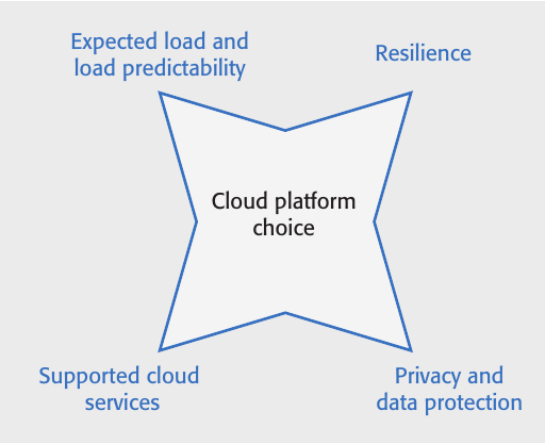
Different types of customers have different expectations about software products. Some examples are:

- Consumers or small businesses do not expect branding and personalization, local authentication system, or varying individual permissions → You can use a multi-tenant database with a single schema
- Large companies are more likely to want a database adapted to their needs → Possible to some extent with a multi-tenant system, easier with a multi-instance database
- For products in which database has to consistent at all times (e.g. finance) → You need a transaction-based system: either a multi-tenant database or a database per customer running on a virtual machine (with all users from each customer sharing the VM-based database)

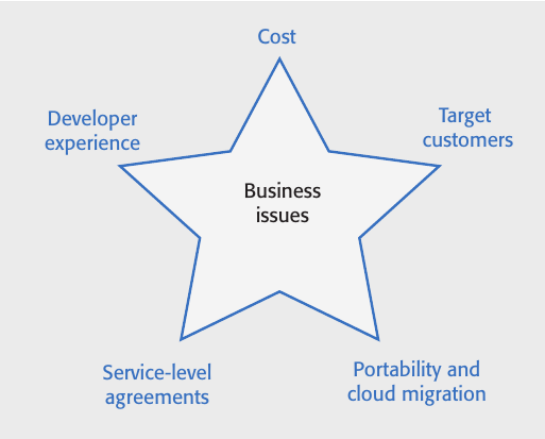
5.2.4 Scalability

The scaleability of a system reflects its ability to adapt automatically to changes in the load on that system. You achieve scaleability in a system by making it possible to add new virtual servers (scaling-out) or increase the power of a system server (scaling-up) in response to increasing load.

In cloud-based systems, scaling-out rather than scaling-up is the normal approach used. Your software has to be organized so that individual software components can be replicated and run in parallel. Also



(a) Technical issues



(b) Business issues

you need load-balancing mechanisms to direct requests to different instances of the components (e.g. with PaaS).

5.2.5 Resilience

The resilience of a system reflects its ability to continue to deliver critical services in the event of system failure or malicious system use. To achieve resilience, you need to be able to restart your software quickly after a hardware or software failure.

Resilience relies on redundancy: the first behaviour is called **"hot standby"** in which the replicas of the software and data are maintained in different locations. Database updates are mirrored so that the standby database is a working copy of the operational database. A system monitor continually checks the system status. It can switch to the standby system automatically if the operational system fails.

A second behaviour is knowed as **"cool standby"** in which you should use redundant virtual servers that are not hosted on the same physical computer and locate servers in different locations. Ideally, these servers should be located in different data centers: if a physical server fails or if there is a wider data center failure, then operation can be switched automatically to the software copies elsewhere.

5.2.6 Software structure

An object-oriented approach to software engineering has been that been extensively used for the development of client-server systems built around a shared database. The system itself is, logically, a **monolithic system** with distribution across multiple servers running large software components. The traditional multi-tier client server architecture is based on this distributed system model.

The alternative to a monolithic approach to software organization is a service-oriented approach where the system is decomposed into **fine-grain, stateless services**: because it is stateless, each service is independent and can be replicated,distributed and migrated from one server to another. The service-oriented approach is particularly suitable for cloud-based software with services deployed in containers.

Cloud platforms include general-purpose clouds such as Amazon Web Services or lesser known platforms oriented around a specific application, such as the SAP Cloud Platform. There are also smaller national providers that provide more limited services but who may be more willing to adapt their services to the needs of different customers.

There is no 'best' platform and you should choose a cloud provider based on your background and experience, the type of product that you are developing and the expectations of your customers. You need to consider both **technical issues** (5.8a) and **business issues** (5.8b) when choosing a cloud platform for your product.

5.3 Kubernetes

K8s 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 underlying individual containers are created and destroyed. Given a set of container workloads to run and a set of machines

on a cluster, the container orchestrator examines each container and determines the optimal machine to schedule that workload .

Kubernetes is like any other cluster – a bunch of nodes and a control plane. The control plane exposes an API, has a scheduler for assigning work to nodes, and state is recorded in a persistent store. Nodes are where application services run. It can be useful to think of the control plane as the brains of the cluster, and the nodes as the muscle. In this analogy, the control plane is the brains because it implements all of the important features such as auto-scaling and zero-downtime rolling updates. The nodes are the muscle because they do the every-day hard work of executing application code.

Orchestrator is just a fancy word for a system that takes care of deploying and managing applications.

To make this happen, you start out with an app, package it up and give it to the cluster (Kubernetes). The cluster is made up of one or more **masters** and a **bunch of nodes**. The masters, sometimes called heads or head nodes, are in-charge of the cluster. This means they make the scheduling decisions, perform monitoring, implement changes, respond to events, and more. For these reasons, we often refer to the masters as the **control plane**.

The nodes are where application services run, and we sometimes call them the data plane. Each node has a reporting line back to the masters, and constantly watches for new work assignments. To run applications on a Kubernetes cluster we follow this simple pattern:

1. Write the application as small independent microservices in our favourite languages.
2. Package each microservice in its own container.
3. Wrap each container in its own Pod.
4. Deploy Pods to the cluster via higher-level controllers such as; Deployments, DaemonSets, StatefulSets, CronJobs etc.

5.3.1 Master and nodes

A **Kubernetes master** is a collection of system services that make up the control plane of the cluster. The simplest setups run all the master services on a single host. However, this is only suitable for labs and test environments. For production environments, multi-master high availability (HA) is a must have. This is why the major cloud providers implement HA masters as part of their hosted Kubernetes platforms such as Azure Kubernetes Service (AKS), AWS Elastic Kubernetes Service (EKS), and Google Kubernetes Engine (GKE). Generally speaking, running 3 or 5 replicated masters in an HA configuration is recommended.

Let's take a quick look at the different master services that make up the control plane.

- **API Server**:All communication, between all components, must go through the API server. We'll get into the detail later in the book, but it's important to understand that internal system components, as well as external user components, all communicate via the same API. It exposes a RESTful API that you POST YAML configuration files to over HTTPS. These YAML files, which we sometimes call manifests, contain the desired state of your application. This desired state includes things like; which container image to use, which ports to expose, and how many Pod replicas to run. All requests to the API Server are subject to authentication and authorization checks, but once these are done, the config in the YAML file is validated, persisted to the cluster store, and deployed to the cluster.

- **Cluster store**:The cluster store is the only stateful part of the control plane, and it persistently stores the entire configuration and state of the cluster. The cluster store is currently based on etcd, a popular distributed database. As it's the single source of truth for the cluster, you should run between 3-5 etcd replicas for high-availability, and you should provide adequate ways to recover when things go wrong. On the topic of availability, etcd prefers consistency over availability. This means that it will not tolerate a split-brain situation and will halt updates to the cluster in order to maintain consistency. However, if etcd becomes unavailable, applications running on the cluster should continue to work, you just won't be able to update anything.

- **Control manager**: The controller manager implements all of the background control loops that monitor the cluster and respond to events. It's a controller of controllers, meaning it spawns all of the independent control loops and monitors them. Some of the control loops include; t*he node controller, the endpoints controller, and the replicaset controller.* Each one runs as a background watch-loop that is constantly watching the API Server for changes -– t*he aim of the game is to

ensure the current state of the cluster matches the desired state (more on this shortly*). The logic implemented by each control loop is effectively this:

1. Obtain desidered state
2. Observe current state
3. Determine differences
4. Reconcile differences

Each control loop is also extremely specialized and only interested in its own little corner of the Kubernetes cluster. No attempt is made to over-complicate things by implementing awareness of other parts of the system – each control loop takes care of its own business and leaves everything else alone. This is key to the distributed design of Kubernetes and adheres to the Unix philosophy of building complex systems from small specialized parts.

- **Scheduler**: At a high level, the scheduler watches the API server for new work tasks and assigns them to appropriate healthy nodes. Behind the scenes, it implements complex logic that filters out nodes incapable of running the task, and then ranks the nodes that are capable. The ranking system is complex, but the node with the highest-ranking score is selected to run the task.

  When identifying nodes that are capable of running a task, the scheduler performs various predicate checks. These include; is the *node tainted*, are there any affinity or anti-affinity rules, is the required network port available on the node, does the node have sufficient free resources etc. Any node incapable of running the task is ignored, and the remaining nodes are ranked according to things such as; does the node already have the required image, how much free resource does the node have, how many tasks is the node already running. Each criterion is worth points, and the node with the most points is selected to run the task.

  If the scheduler cannot find a suitable node, the task cannot be scheduled and is marked as pending. The scheduler isn't responsible for running tasks, just picking the nodes a task will run on.

Nodes

Nodes are the workers of a Kubernetes cluster. At a high-level they do three things:

1. Watch the API Server for new work assignments
2. Execute new work assignments
3. Report back to the control plane (via the API server)

Let's look at the three major components of a node:

- **Kubelet**:It's the main Kubernetes agent, and it runs on every node in the cluster. In fact, it's common to use the terms node and kubelet interchangeably.

  When you join a new node to a cluster, the process installs kubelet onto the node. The kubelet is then responsible for registering the node with the cluster. Registration effectively pools the node's CPU, memory, and storage into the wider cluster pool. One of the main jobs of the kubelet is to watch the API server for new work assignments. Any time it sees one, it executes the task and maintains a reporting channel back to the control plane. If a kubelet can't run a particular task, it reports back to the master and lets the control plane decide what actions to take.

  For example, if a Kubelet cannot execute a task, it is not responsible for finding another node to run it on. It simply reports back to the control plane and the control plane decides what to do.

- **Container Runtime Interface (CRI)**:The Kubelet needs a container runtime to perform container-related tasks -– things like pulling images and starting and stopping containers. In the early days, Kubernetes had native support for a few container runtimes such as Docker. More recently, it has moved to a plugin model called the Container Runtime Interface (CRI). At a high-level, the CRI masks the internal machinery of Kubernetes and exposes a clean documented interface for 3rd-party container runtimes to plug into.

- **Kube-proxy**: The last piece of the node puzzle is the kube-proxy. This runs on every node in the cluster and is responsible for local cluster networking. For example, it makes sure each node gets its own unique IP address, and implements local IPTABLES or IPVS rules to handle routing and load-balancing of traffic on the Pod network.

There is surely a **Kubernetes DNS**: the cluster's DNS service has a static IP address that is hard-coded into every Pod on the cluster, meaning all containers and Pods know how to find it. Every new service is automatically registered with the cluster's DNS so that all components in the cluster can find every Service by name. Some other components that are registered with the cluster DNS are StatefulSets and the individual Pods that a StatefulSet manages.

5.3.2 Declarative model and desired state

The declarative model and the concept of desired state are at the very heart of Kubernetes. In Kubernetes, the declarative model works like this:

1. Declare the desired state of an application (microservice) in a manifest file 2. POST it to the API server 3. Kubernetes stores it in the cluster store as the application's desired state 4. Kubernetes implements the desired state on the cluster 5. Kubernetes implements watch loops to make sure the current state of the application doesn't vary from the desired state

Let's analyze these steps deeper. Manifest files are written in simple YAML, and they tell Kubernetes how you want an application to look. This is called the desired state. It includes things such as; which image to use, how many replicas to run, which network ports to listen on, and how to perform updates.

Once you've created the manifest, you POST it to the API server. The most common way of doing this is with the `kubectl` command-line utility. This sends the manifest to the control plane as an HTTP POST, usually on port 443.

Once the request is authenticated and authorized, Kubernetes inspects the manifest, identifies which controller to send it to (e.g. the Deployments controller), and records the config in the cluster store as part of the cluster's overall desired state. Once this is done, the work gets scheduled on the cluster. This includes the hard work of pulling images, starting containers, building networks, and starting the application's processes.

Finally, Kubernetes utilizes background reconciliation loops that constantly monitor the state of the cluster. If the current state of the cluster varies from the desired state, Kubernetes will perform whatever tasks are necessary to reconcile the issue.

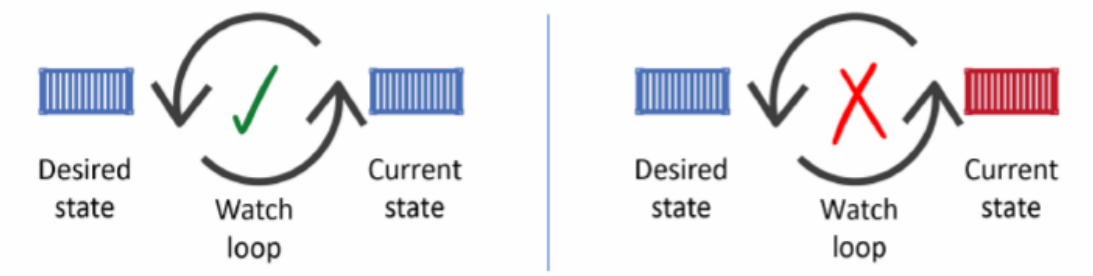


Figure 5.9: Watch loop concept

Not only is the declarative model a lot simpler than long scripts with lots of imperative commands, it also enables self-healing, scaling, and lends itself to version control and self-documentation. It does this by telling the cluster how things should look. If they stop looking like this, the cluster notices the discrepancy and does all of the hard work to reconcile the situation. But the declarative story doesn't end there – things go wrong, and things change. When they do, the current state of the cluster no longer matches the desired state. As soon as this happens, Kubernetes kicks into action and attempts to bring the two back into harmony.

5.3.3 Pods

In the VMware world, the atomic unit of scheduling is the virtual machine (VM). In the Docker world, it's the container. Well... in the Kubernetes world, it's the Pod. It's true that Kubernetes runs containerized apps. However, you cannot run a container directly on a Kubernetes cluster – containers must always run inside of Pods. The very first thing to understand is that the term Pod comes from a pod of whales – in the English language we call a group of whales a pod of whales. As the Docker logo is a whale, it makes sense that we call a group of containers a Pod. The simplest model is to run a single container per Pod. However, there are advanced use-cases that run multiple containers inside a single Pod. These multi-container Pods are beyond the scope of what we're discussing here.

At the highest-level, a Pod is a ring-fenced environment to run containers. The Pod itself doesn't actually run anything, it's just a sandbox for hosting containers. Keeping it high level, you ring-fence an

area of the host OS, build a network stack, create a bunch of kernel namespaces, and run one or more containers in it.

If you're running multiple containers in a Pod, they all share the same **Pod environment**. This includes things like the IPC namespace, shared memory, volumes, network stack and more. As an example, this means that all containers in the same Pod will share the same IP address (the Pod's IP): If two containers in the same Pod need to talk to each other (container-to-container within the Pod) they can use ports on the Pod's localhost interface.

5.3.4 Deployments

At a high level, you start with application code. That gets packaged as a container and wrapped in a Pod so it can run on Kubernetes. However, Pods don't self-heal, they don't scale, and they don't allow for easy updates or rollbacks. Deployments do all of these. As a result, you'll almost always deploy Pods via a Deployment controller.

It's important to know that a single Deployment object can only manage a single **Pod template**. For example, if you have an application with a Pod template for the web front-end and another Pod template for the catalog service, you'll need two Deployments. However, as you saw in the previous figure, a Deployment can manage multiple replicas of the same Pod. For example, the figure could be a Deployment that currently manages two replicated web server Pods.

The next thing to know is that Deployments are fully-fledged objects in the Kubernetes API. This means you define them in manifest files that you POST to the API Server. The last thing to note, is that behind-the-scenes, Deployments leverage another object called a ReplicaSet. While it's best practice that you don't interact directly with ReplicaSets, it's important to understand the role they play. Keeping it high-level, Deployments use ReplicaSets to provide self-healing and scaling. As shown in figure, think of

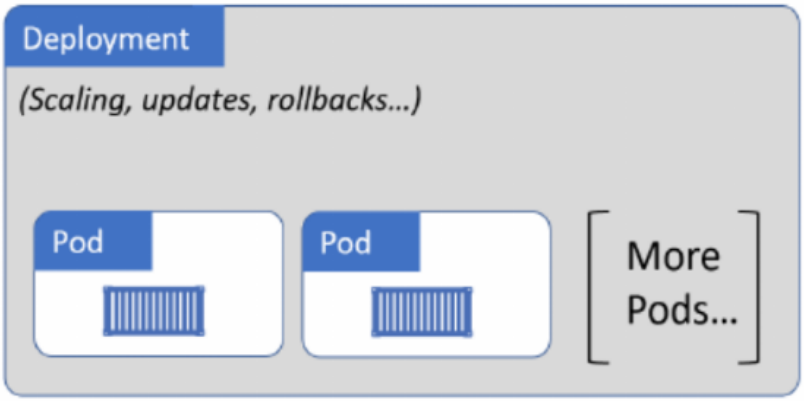


Figure 5.10: Deployment encapsulate Pods

Deployments as managing ReplicaSets, and ReplicaSets as managing Pods. Put them all together, and you've got a great way to deploy and manage applications on Kubernetes.

Pods augment containers by allowing **co-location of containers, sharing of volumes, sharing of memory, simplified networking and so on**. BUt they offer nothing in the way of self-healing and scalability: here **Deployments** enter the game.

Deployments augment Pods by adding things these things, that means:

- if a Pod managed by a Deployment fails, it will be replaced (self-healing)
- if a Pod managed by a Deployment sees increased load, you can easily add more of the same Pod to deal with the load (scaling)

Before going any further, it's critical to understand three concept that are fundamental to everything about K8s:

- **Desired state**: is what you want.
- **Current state**: is what you have; if match with desidered state, everything is fine.
- **Declarative model**: is a way of telling Kubernetes what your desired state is, without having to get into the detail of how to implement it. You leave the how up to Kubernetes.

undamental to desired state is the concept of background reconciliation loops or control loops.

For example, ReplicaSets implement a background reconciliation loop that is constantly checking whether the right number of Pod replicas are present on the cluster. If there aren't enough, it adds more. If there are too many, it terminates some. **Kubernetes is constantly making sure that current state matches desired state.**

These very-same reconciliation loops enable scaling. For example, if you POST an updated config that changes replica count from 3 to 5, the new value of 5 will be registered as the application's new desired state. The next time the ReplicaSet reconciliation loop runs, it will notice the discrepancy and follow the same process, start the read alert and spinning up two more replicas.

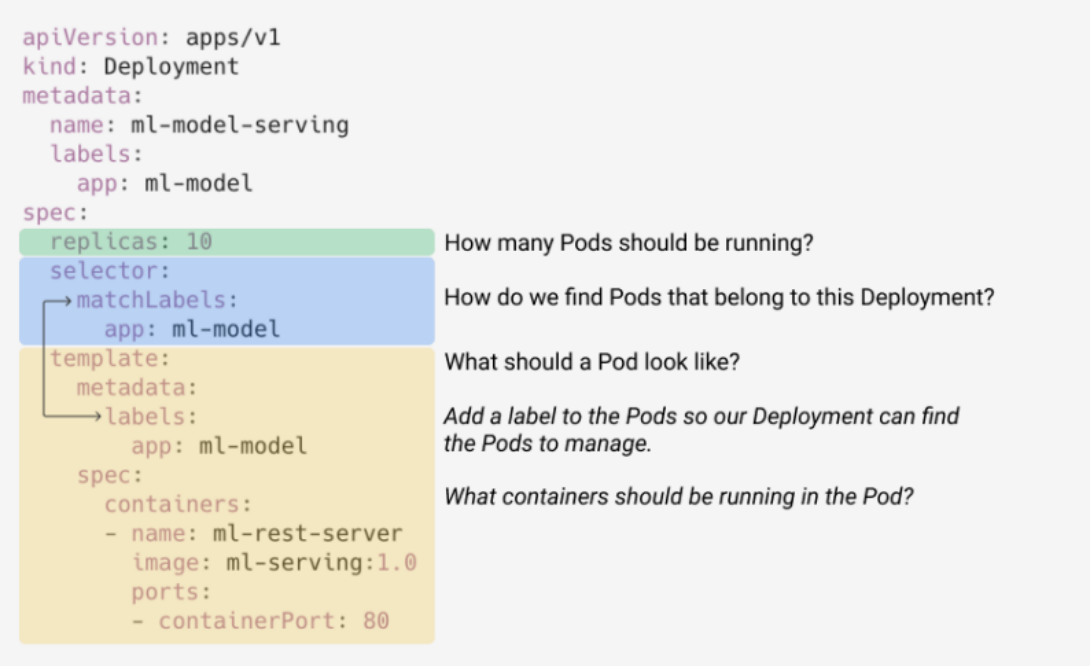


Figure 5.11: `deployment.yaml`

Right at the very top you specify the API version to use. Assuming that you're using an up-to-date version of Kubernetes, Deployment objects are in the apps/v1 API group. Next, the `.kind` field tells Kubernetes you're defining a Deployment object. The `.metadata` section is where we give the Deployment a name and labels. The `.spec` section is where most of the action happens. Anything directly below `.spec` relates to the Pod. Anything nested below `.spec.template` relates to the Pod template that the Deployment will manage. In this example, the Pod template defines a single container. `.spec.replicas` tells Kubernetes how may Pod replicas to deploy. `spec.selector` is a list of labels that Pods must have in order for the Deployment to manage them.

5.3.5 Service

We've just learned that Pods are mortal and can die. However, if they're managed via *Deployments* or *DaemonSets*, they get replaced when they fail. But replacements come with totally different IP addresses. This also happens when you perform scaling operations – scaling up adds new Pods with new IP addresses, whereas scaling down takes existing Pods away. Events like these cause a lot of *IP churn*.

The point I'm making is that **Pods are unreliable**, which poses a challenge. Assume you've got a microservices app with a bunch of Pods performing video rendering. *How will this work if other parts of the app that need to use the rendering service cannot rely on the rendering Pods being there when they need them?*

This is where Services come in to play. **Services provide reliable networking for a set of Pods.** Figure shows the uploader microservice talking to the renderer microservice via a Kubernetes Service object. The Kubernetes Service is providing a reliable name and IP, and is load-balancing requests to the two renderer Pods behind it.

Services are fully-fledged objects in the Kubernetes API – just like Pods and Deployments. They have a front-end that consists of a stable DNS name, IP address, and port. On the back-end, they load-balance across a dynamic set of Pods. As Pods come and go, the Service observes this, automatically

updates itself, and continues to provide that stable networking endpoint. The same applies if you scale the number of Pods up or down. New Pods are seamlessly added to the Service and will receive traffic. Terminated Pods are seamlessly removed from the Service and will not receive traffic.

That's the job of a Service – it's a stable network abstraction point that provides TCP and UDP load-balancing across a dynamic set of Pods. As they operate at the TCP and UDP layer, Services do not possess *application intelligence* and cannot provide application-layer host and path routing. For that, you need an **Ingress**, which understands HTTP and provides host and path-based routing.

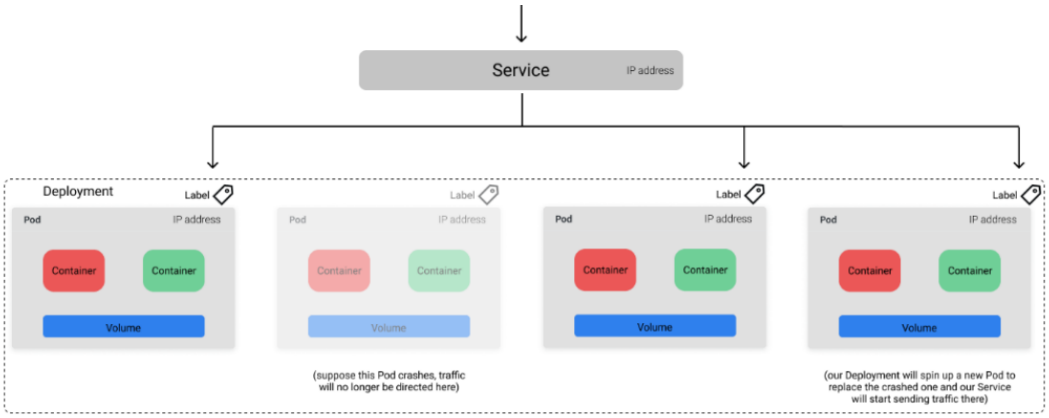


Figure 5.12: Service schema

```yaml
apiVersion: v1
kind: Service
metadata:
  name: ml-model-svc
  labels:
    app: ml-model
spec:
  type: ClusterIP
  selector:
    app: ml-model
  ports:
  - protocol: TCP
    port: 80
```

How do we want to expose our endpoint?

How do we find Pods to direct traffic to?

How will clients talk to our Service?

Figure 5.13: `service.yaml`

In the example files, the Service has a label selector (spec.selector) with a single value `app=ml-model`. This is the label that the Service is looking for hen it queries the cluster for matching Pods. The Deployment specifies a Pod template with the same `app=ml-model` label. It's these two attributes that loosely couple the Service to the Deployment's Pod.

For some parts of your application (for example, frontends) you may want to expose a Service onto an external IP address, that's outside of your cluster.

Kubernetes `ServiceTypes` allow you to specify what kind of Service you want.

Type values and their behaviors are:

- **ClusterIP**: Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default that is used if you don't explicitly specify a type for a Service.
- **NodePort**: Exposes the Service on each Node's IP at a static port (the NodePort). To make the node port available, Kubernetes sets up a cluster IP address, the same as if you had requested a Service of type: ClusterIP.
- **ExternalName**: Exposes the Service externally using a cloud provider's load balancer.
- **LoadBalancer**: Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up.

5.3.6 Ingress

Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.

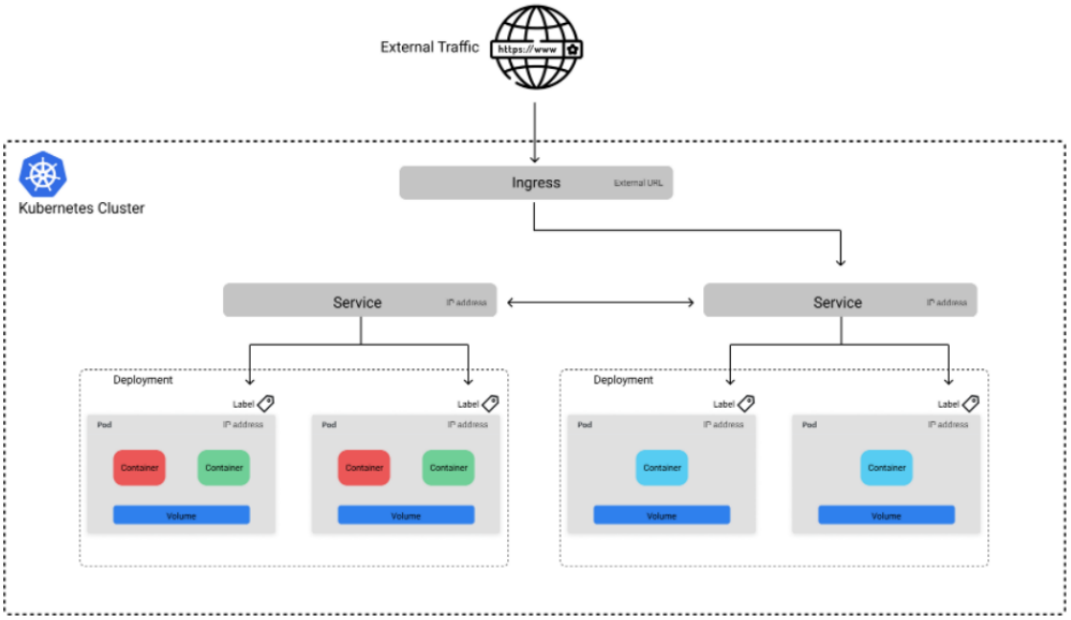


Figure 5.14: Ingress schema

You must have an Ingress controller to satisfy an Ingress. Only creating an Ingress resource has no effect. You may need to deploy an Ingress controller such as ingress-nginx. You can choose from a number of Ingress controllers.

Ideally, all Ingress controllers should fit the reference specification. In reality, the various Ingress controllers operate slightly differently. An Ingress needs `apiVersion, kind, metadata` and *spec* fields. The name of an Ingress object must be a valid DNS subdomain name. Ingress frequently uses annotations to configure some options depending on the Ingress controller, an example of which is the rewrite-target annotation. Different Ingress controllers support different annotations. Review the documentation for your choice of Ingress controller to learn which annotations are supported.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ml-product-ingress
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /app
        backend:
          serviceName: user-interface-svc
          servicePort: 80
```

Configure options for the Ingress controller.

How should external traffic access the service?

What Service should we direct traffic to?

Figure 5.15: `ingress.yaml`

The Ingress `spec` has all the information needed to configure a load balancer or proxy server. Most importantly, it contains a list of rules matched against all incoming requests. Ingress resource only supports rules for directing HTTP(S) traffic.

5.3.7 Evaluations

When should you NOT use K8s?

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

In comparison with K8s, **Docker Swarm** is simpler to install, have a softer learning curve and it's generally preferred in environments where simplicity and fast development are prioritized.

Chapter 6

Microservices

A software service is a software component that can be accessed from remote computers over the Internet. Given an input, a service produces a corresponding output, without side effects: the service is accessed through its published interface and all details of the service implementation are hidden. Usually services do not maintain any internal state. State information is either stored in a database or is maintained by the service requestor.

When a service request is made, the state information may be included as part of the request and the updated state information is returned as part of the service result. As there is no local state, services can be dynamically reallocated from one virtual server to another and replicated across several servers. Some modern web services of 2000s are:

- Service-oriented architecture (SOA - 1990s): Independent, stand-alone services, with public interfaces, implemented with different technologies
- Web services (early 2000s): use protocol as XML, SOAP, WSDL, plus dozens (!) of other standards. Web services exchanging large and complex XML data that augment message management overhead.
- Modern service-oriented systems: use simpler, 'lighter weight' service-interaction protocols that have lower overheads and, consequently, faster execution.

Amazon's rethinking of what a service should be: a service should be related to a single business function, should be completely independent, with their own database, should manage their own user interface, it should be possible to replace/replicate a service without changing other services. So, **microservices** are **small-scale, stateless, services that have a single responsibility**.

Let's see an example on authentication service that provide:

- user registration
- authentication using UID/password
- two-factor authentication
- user information management
- password reset

To identify the microservices that might be used for the authentication system it's necessary to break coarse-grain featues into small detailed functions, look at the data used and identify a microservice for each logical data item to be managed and also minimize the amount of replicate data management, as pictured in 6.1.

Let's define every term of our microservice informal definition:

- Self-contained: Microservices do not have external dependencies. They manage their own data and implement their own user interface.
- Lightweight: Microservices communicate using lightweight protocols, so that service communication overheads are low.
- Implementation-independent: Microservices may be implemented using different programming languages and may use different technologies (e.g. different types of database) in their implementation.

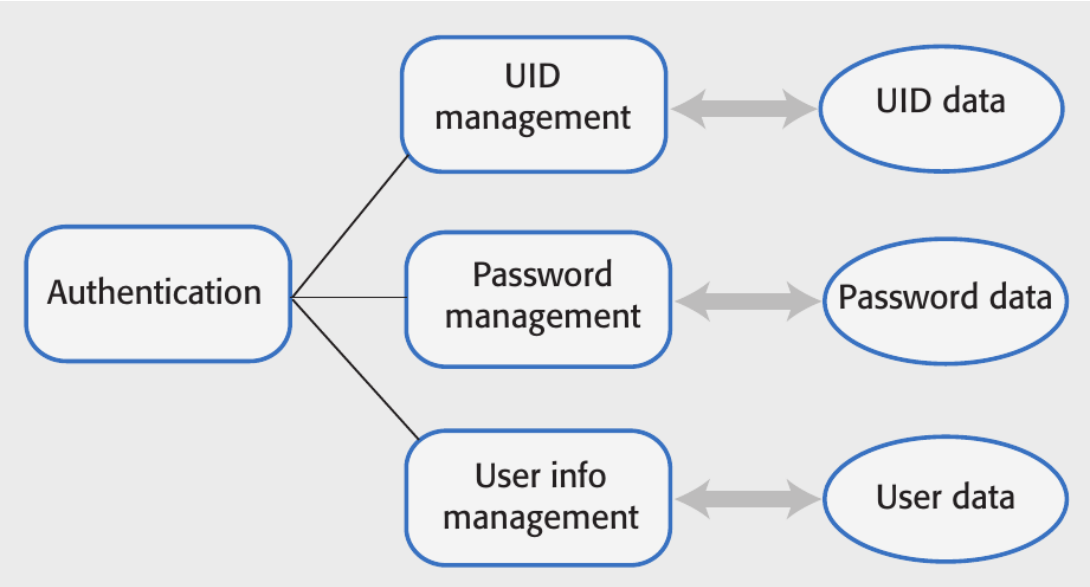


Figure 6.1: Authentication service

- Independently 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.

A well-designed microservice should have high cohesion and low coupling.

Cohesion is a measure of the number of relationships that parts of a component have with each other. High cohesion means that all of the parts that are needed to deliver the component's functionality are included in the component.

Coupling is a measure of the number of relationships that one component has with other components in the system. Low coupling means that components do not have many relationships with other components.

Each microservice should have a **single responsibility** i.e. it should do one thing only and it should do it well. However, 'one thing only' is difficult to define in a way that's applicable to all services: responsibility does not always mean a single, functional activity.

How big should a microservice be?

There is a general rule called *"Rule of twos"*: service can be developed, tested, and deployed by a team in two weeks.Team can be fed with two large pizzas (**8-10 people**).

Why so many people for a microservice?

Team of this size are necessary to implement service functionality, develop code that makes service completely independent, processing incoming and outgoing messages, manage failures (service/interactions failures), manage data consistency when data are used by other services, maintain service own interface, test service and service interactions, support service after deployment.

6.0.1 Microservices architecture

A microservices architecture is an architectural style – a tried and tested way of implementing a logical software architecture.This architectural style addresses two problems with monolithic applications:

- Problem 1: The whole system has to be rebuilt, re-tested and re-deployed when any change is made. This can be a slow process as changes to one part of the system can adversely affect other components.
- Problem 2: As the demand on the system increases, the whole system has to be scaled, even if the demand is localized to a small number of system components that implement the most popular system functions.

Microservices are self-contained and run in separate processes. in cloud-based systems, each microservice may be deployed in its own container. This means a microservice can be stopped and restarted

without affecting other parts of the system. If the demand on a service increases, service replicas can be quickly created and deployed. These do not require a more powerful server so 'scaling-out' is, typically, much cheaper than 'scaling up'.

Example: Photo-printing system for mobile devices

Imagine that you are developing a photo printing service for mobile devices. Users can upload photos to your server from their phone or specify photos from their Instagram account that they would like to be printed. Prints can be made at different sizes and on different media.
Users can chose print size and print medium. For example, they may decide to print a picture onto a mug or a T-shirt. The prints or other media are prepared and then posted to their home. They pay for prints either using a payment service such as Android or Apple Pay or by registering a credit card with the printing service provider.

The best approach is to separate services for each area of functionality. The API gateway work for insulates user app from the system's microservices, provide a single point of contact and translates app service requests into calls to microservices.

The **Decomposition guidelines** that drive the **architectural design decisions** states that development teams for each service must be autonomous.
How to decompose system into a set of microservices?
If NOT too many microservices this will have an impact on low cohesion by augment the communciation overhead, dually not too few will have impact on high coupling by introducing dependency for updates/deployments. Some general tips are:

- 1. Balance fine-grain functionality and system performance
- 2. Follow the "common closure principle": elements likely to be changed at the same time should stay in same service
- 3. Associate services with business capabilities
- 4. Services should have access only the data they need plus supporting data propagation mechanisms

6.0.2 Service communications

Services communicate by exchanging messages that include information about the originator of the message, as well as the data that is the input to or output from the request. When you are designing a microservices architecture, you have to establish a standard for communications that all microservices should follow. There are two types of interaction: **syncrhonous (direct)** and **asynchronous (indirect)**.

In a synchronous interaction, service A issues a request to service B. Service A then suspends processing while B is processing the request: it waits until service B has returned the required information before continuing execution.

In an asynchronous interaction, service A issues the request that is queued for processing by service B. A then continues processing without waiting for B to finish its computations. Sometime later, service B completes the earlier request from service A and queues the result to be retrieved by A. Service A, therefore, has to check its queue periodically to see if a result is available. The general scenario is pictued in 6.2

6.0.3 Data distribution and sharing

You should isolate data within each system service with as little data sharing as possible: if data sharing is unavoidable, you should design microservices so that most sharing is 'read-only', with a minimal number of services responsible for data updates. If services are replicated in your system, you must include a mechanism that can keep the database copies used by replica services consistent.

Shared database architectures employ ACID transactions to serialize updates and avoid inconsistency: in distributed systems we must trade-off data consistency and performance.

Microservices systems must be designed to tolerate some degree of data inconsistency. Two types of incosistency have to be managed:

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

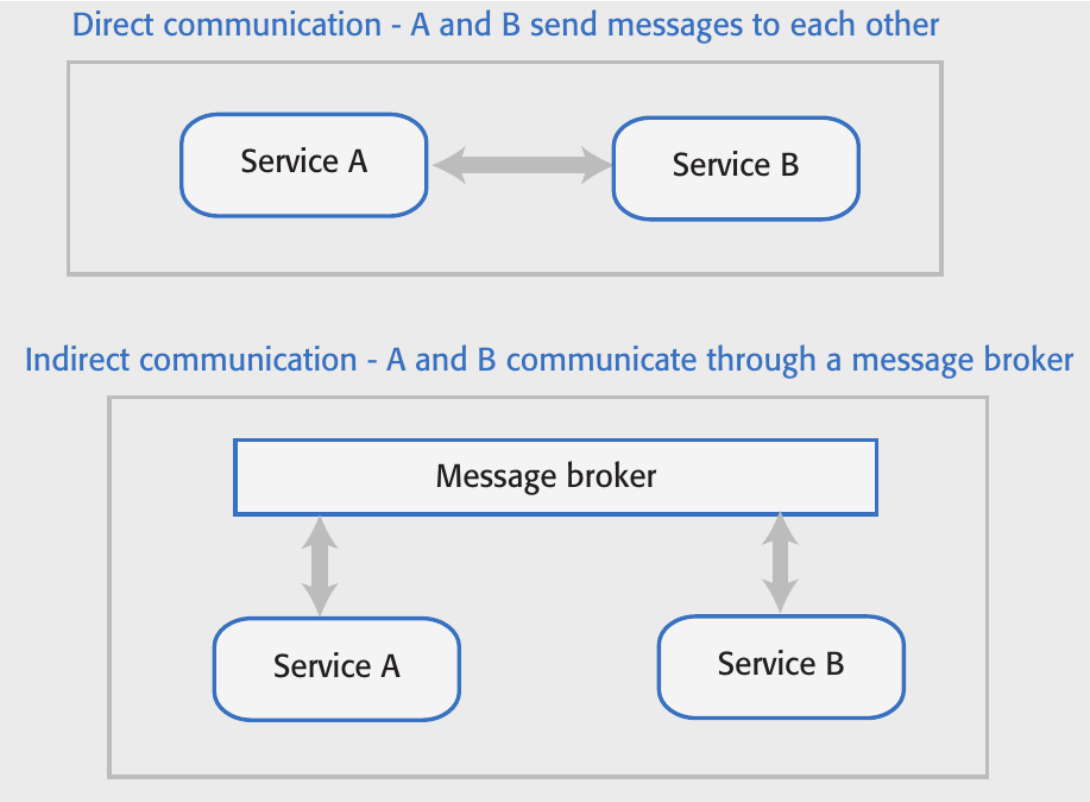


Figure 6.2: Direct vs indirect communication

2. **Replica incosistency:** Several replicas of the same service may be executing concurrently. Each with its own db copy, each updates its own db copy so there is a need to make these dbs *"eventually consistent"*.

A foundamental theorem in distributed system is the **CAP Theorem** (*Consistency, Availability, Partition tolerance*):

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

Where:

- Consistency: any read operation that begins after a write operation must return that value, or the result of a later write operation
- Availability: every request received from a non-failing node must result in a response
- Network partition: network can lose arbitrarily many messages sent from one group to another

Let's see the *proof*:
Suppose that services S1 and S2 belong to two different network partitions, and that both contain value v0. Consider the following sequences of events: `a1 = "C sends "write(v1)" to S1; E1; "S1 responds to C"` where E1 is a (possibly) empty sequence of other events which contains neither other client requests to S1 or S2, nor messages from S1 received by S2, nor messages from S2 received by S1.
Consider the following sequences of events: `a2 = "C sends "read" to S2; E2; "S2 responds to C"` where E2 is a sequence like E1. Note that `"S1 responds to C"` and `"S2 responds to C"` belong to a1 and a2, respectively, because of the Availability hypothesis. Now `a1.a2 = a2 for S2`, hence `S2 responds v0 to C.`
Because of the Consistency hypothesis, S2 should instead respond v1 to C. Contradiction.

The Saga pattern

Implement each business transaction that spans multiple services as a saga: a saga is a sequence of local transactions, each local transaction updates a database and triggers next local transaction(s) in the saga. If a local transaction fails then the saga executes a series of compensating transactions Two ways of coordinating sagas:

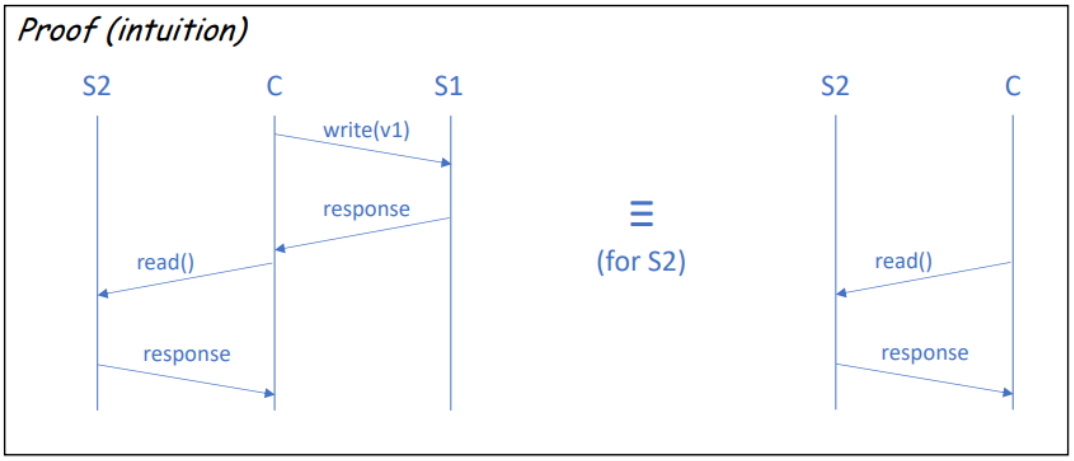


Figure 6.3: CAP Theorem proof by intuition

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

This introduce the necessity of introducing a mechanism for compensating transaction. Two model are available:

- Backward model: undo changes made by previously executed local transactions
- Forward model: "retry later" at regular interval

For example, *Netflix* approach is to replicate data in n nodes by writing to the ones you can get to, then fix it up afterwards. It use a **quorum** (*e.g. (n/2 + 1) of the replicas must respond*) by executing a consensus algorithm (the one used by *Apache Cassandra*).

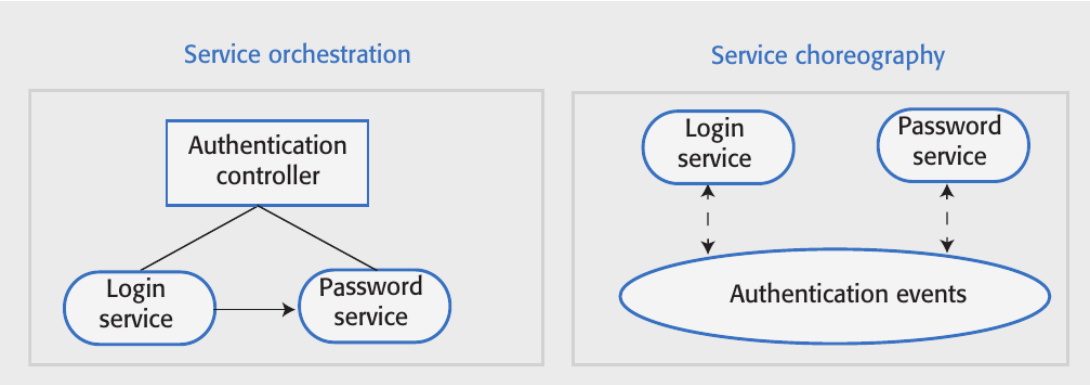


Figure 6.4: Service coordination

6.0.4 Failure management

Services must be designed to cope with failures like:

- Internal service failure: these are conditions that are detected by the service and can be reported to the service client in an error message. An example of this type of failure is a service that takes a URL as an input and discovers that this is an invalid link.
- External service failure: these failures have an external cause, which affects the availability of a service. 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.

Some mechanism used to manage failures are **timeouts** and **circuit breakers**.
A timeout is a counter that this associated with the service requests and starts running when the request is made: once the counter reaches some predefined value, such as 10 seconds, the calling service assumes that the service request has failed and acts accordingly. The problem with the timeout approach is that every service call to a 'failed service' is delayed by the timeout value so the whole system slows down. Instead of using timeouts explicitly when a service call is made, we suggests using a **circuit breaker**: like an electrical circuit breaker, this immediately denies access to a failed service without the delays associated with timeouts.

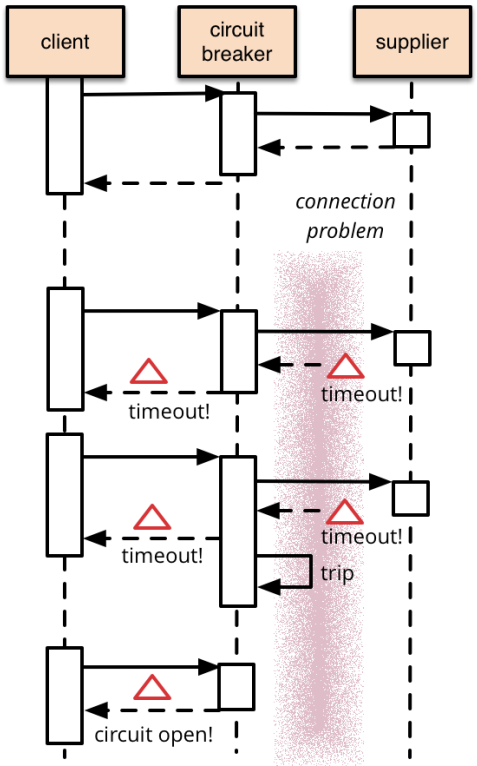


Figure 6.5: Circuit breaker pattern

6.0.5 REpresentational State Transfer

The REST (REpresentational State Transfer) architectural style is based on the idea of transferring representations of digital resources from a server to a client. Resources are accessed via their unique URI and RESTful services operate on these resources. As stated by *Roy Fielding*:

"each action resulting in a transition to the next state of the application by transferring a representation of that state to the user 6.6"

REST follow some simple REST principles:

- Resource identification through URIs: Service exposes set of resources identified by URIs
- Uniform interface
  - Clients invoke HTTP methods to create/read/update/delete resources:
  - POST and PUT to create and update state of resource
  - DELETE to delete a resource
  - GET to retrieve current state of a resource
- Self-descriptive messages
  - Requests contain enough context information to process message
  - Resources decoupled from their representation so that content can be accessed in a variety of formats (e.g., HTML, XML, JSON, plain text, PDF, JPEG, etc.)

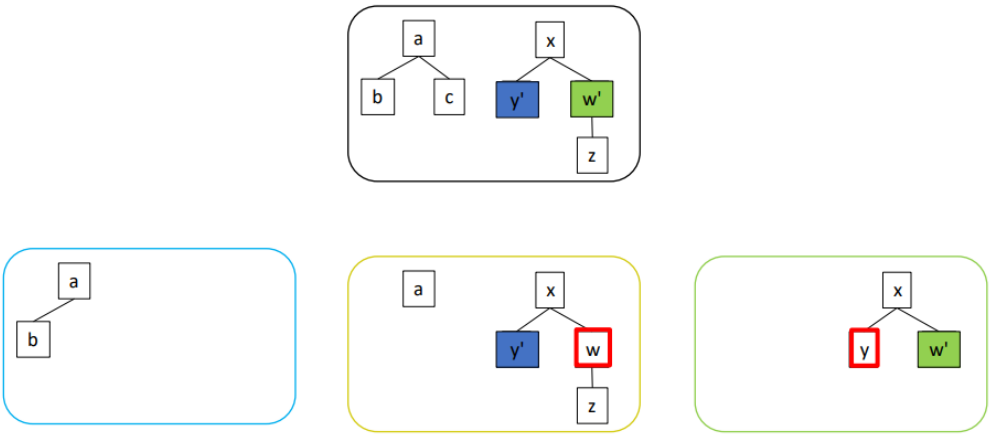


Figure 6.6: State transfer: coordinating the information around several instances

- Stateful interactions through hyperlinks
  - Every interaction with a resource is stateless
  - Server contains no client state, any session state is held on the client
  - Stateful interactions rely on the concept of explicit state transfer

barbera.com

```
GET /customers/fred
```

```
200 OK
<customer>
   <name>Fred Flinstone</name>
   <address> 45 Cave Stone Road, Bedrock</address>
   <orders>http://barbera.com/customers/fred/orders</orders>
</customer>
```

```
GET /customers/fred/orders
```

```
200 OK
<orders>
   <customer>http://barbera.com/customers/fred</customer>
   <order id="1">
      <orderURL>http://barbera.com/orders/1122</orderURL>
      <status>open</status>
   </order>
   ...
</orders>
```

```
GET /orders/1122
```

```
200 OK
<order>
   <customer>http://barbera.com/customers/fred</customer>
   <item quantity="1">brontoburger</item>
</order>
```

```
PUT /orders/1122
<order>
   <customer>http://barbera.com/customers/fred</customer>
   <item quantity="50">brontoburger</item>
</order>
```

```
200 OK
```

Figure 6.7: Example of REST exchange

6.0.6 Service deployment automation

The service development teams decide which programming language, database, libraries and other support software should be used to implement their service. Consequently, there is no 'standard' deployment

configuration for all services. It is now normal practice for microservice development teams to be responsible for deployment and service management as well as software development and to use **continuous deployment**: means that as soon as a change to a service has been made and validated, the modified service is redeployed.

Continuous deployment depends on automation so that as soon as a change is committed, a series of automated activities is triggered to test the software: if the software 'passes' these tests, it then enters another automation pipeline that packages and deploys the software. The deployment of a new service version starts with the programmer committing the code changes to a code management system such as Git.This triggers a set of automated tests that run using the modified service. If all service tests run successfully, a new version of the system that incorporates the changed service is created. Another set of automated system tests are then executed. If these run successfully, the service is ready for deployment.

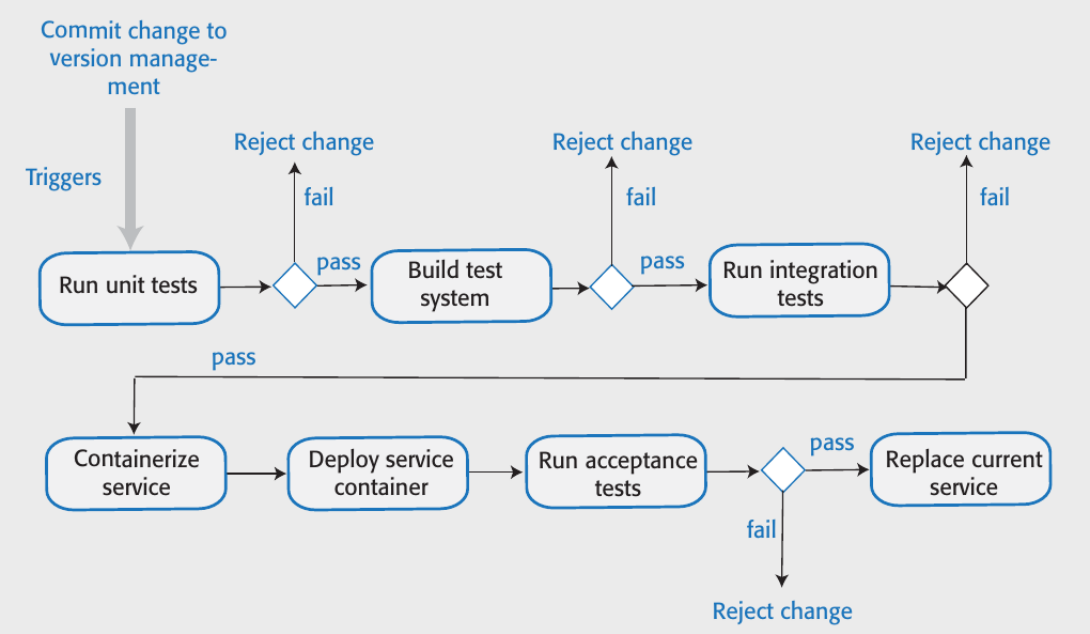


Figure 6.8: A continous deployment pipeline

Testing cannot prevent 100% of unanticipated problems so there is a need to monitor the deployed services. **Monitoring** allows to roll back to previous version in case of service failure.
Example: When you introduce a new version of a service, you maintain the old version but change the "current version link" to point at the new service. If monitor detects a problem with new version of "Cameras 002" service, it switches the "current version link" back to version 001 of the Cameras service. The picture in 6.9 is explicative.

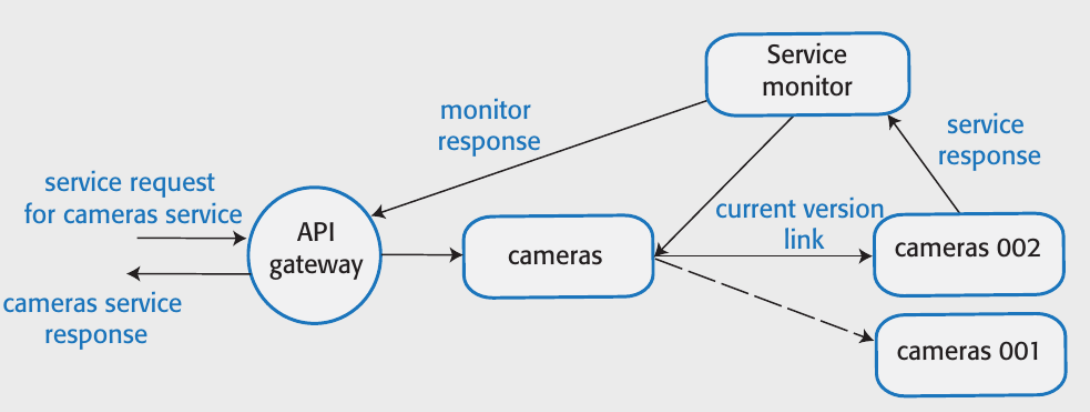


Figure 6.9: A versioned service example

6.0.7 Architectural smells and refactorings

An **architectural smell** is a commonly used architectural decision that negatively impacts system lifecycle qualities. An academic review of white and grey literature (41 references) aimed at identifying the most recognised architectural smells for microservices, and the architectural refactorings to resolve them.
The **design principles** to follow to avoid architectural smells are:

- Independent deployability: the microservices forming an application should be independently deployable
- Horizontal scalability: the microservices forming an application should be horizontally scalable providing the possibility of adding/removing replicas of single microservice
- Isolation of failures: failures should be isolated
- Decentralization: decentralisation should occur in all aspects of microservice-based applications, from data management to governance

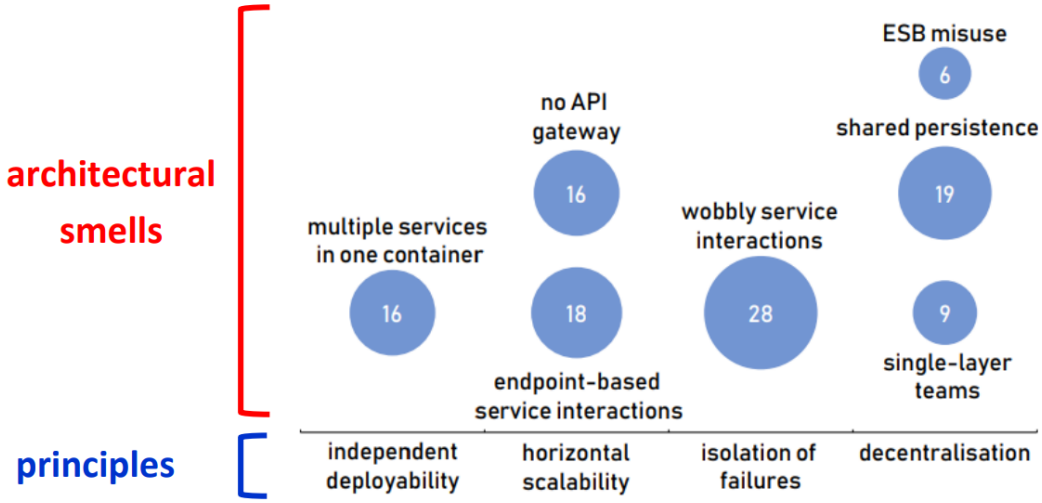


Figure 6.10: Categorization of architectural smells and principles

The schema pictured in 6.11 show the architectural smells and principles taxonomy with the proposed refactoring to solve the identified smell.

6.0.8 MicroFreshener

MicroFreshener is a tool for editing app specifications, automatically identifying architectural smells and applying architectural refactorings to resolve the identified smells. We can model the application architecure by using the semantic showed in 6.12.

MicroFreshener is a (freely) usable to analyse and refactor microservice-based apps, used in industrial case study (*e.g. starting from 4 no API gateway smells and 1 shared persistence smell we obtained a 2 API gateway and 1 data manager refactored architecture*). It's also used to conduct controlled experiment.
It's noticeable to remark that proposed refactoring at **architectural level** does not necessarely correspond to refactoring at **implementation level**: concrete implementation of refactoring are left to application manager (like in design patterns). Also, in case of different proposed refactoring, there is not the best to be applied but it's dependent on the application, as in the following figure. Referring to 6.13 upper refactoring can be ok if the `mB` can be efficiently support multiple topics. Differently, lower refactoring introduces a new `mB`.

MicroFreshener example

Some MicroFreshener refactoring proposal are showed in 6.14 and 6.15.

Another example involve a the `microase` application: the application architecture design replicated in

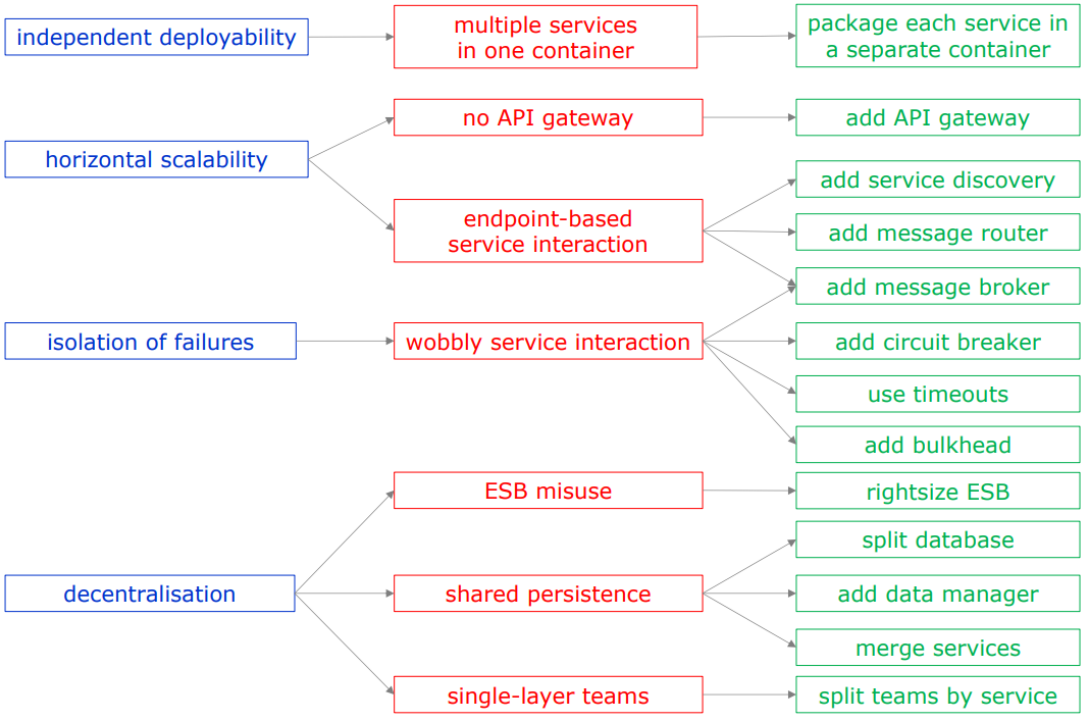


Figure 6.11: Architectural smell → Violated principles → Proposed refactoring

MicroFreshener is shown in 6.18. Here we deep dive into relevated smells and proposed refactoring by MicroFreshener:

- **No API gateway**: The no API gateway smell occurs whenever the external clients of an application directly interact with some internal services. If one of such services is scaled out, the horizontal scalability of microservices may get violated because external clients may keep invoking the same instance, without reaching any replica.
  The proposed solution is to **Add an API Gateway message router (MR)**: by using the modeler we need to delete the arrow from `Ext user` to `gw`, add communication pattern node message router and connect it with `ext user` and all services, as pictured in 6.16.

- **Shared Persistency**: The shared persistency smell occurs whenever multiple services access or manage the same DB, possibly violating the decentralisation design principle (i.e. business logic of an application should be fully decentralised and distributed among its microservices, each of which should own its own domain logic).
  The proposed solution is to **Add a adata manager** but it adds the complexity of a new microsrvices to implement and mantain. A smarter refactoring consists of using `stats` as data manager (*noticing that it represent a natural candidate for such role*) obtaining the diagram shown in 6.17.

- **Wobbly service interaction**: The interaction of a microservice m_i with another microservice m_f is wobbly when a failure in m_f can result in triggering a failure also in m_i. This typically happens when m_i is directly consuming one or more functionalities offered by m_f, and m_i is not provided with any solution for handling the possibility of m_f to fail and be unresponsive (which can lead to failure cascades). We have spotted another modelling error: Flask requests are equipped with 30 second timeouts by default: the proposed solution is to **Use timeout**.

- **Endpoint-based interaction**: The endpoint-based interaction smell occurs in an application when one or more of its microservices invoke a specific instance of another microservice (e.g., because its location is hardcoded in the source code of the microservices invoking it, or because no load balancer is used). If this is the case, when scaling out the latter microservice by adding new replicas, these cannot be reached by the invokers, hence only resulting in a waste of resources. The proposed solution is to **Add Service Discovery**.

Depending on the Deployment, the architecture might change: by using **Docker compose deployment** the overlay network DNS of Docker acts as a dynanic service discovery for running service instances so

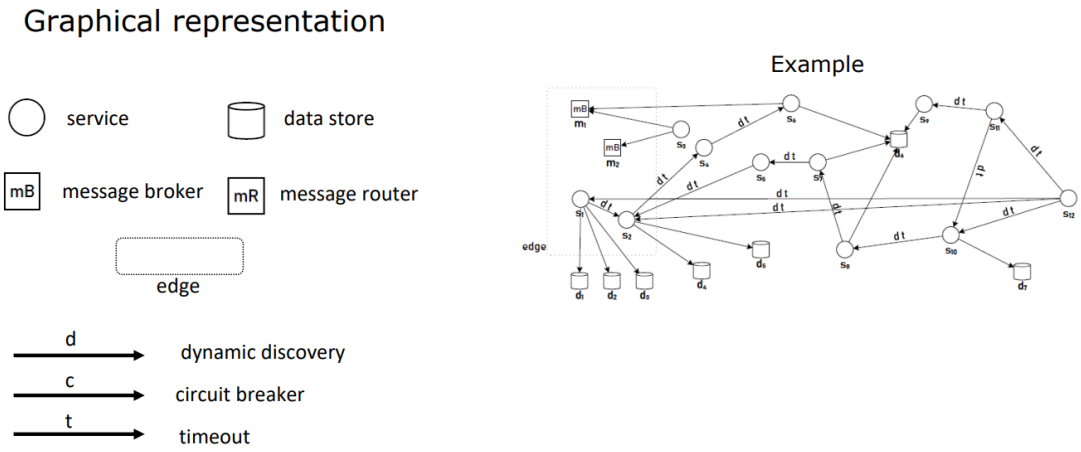


Figure 6.12: Semantic used to modelling application architecture

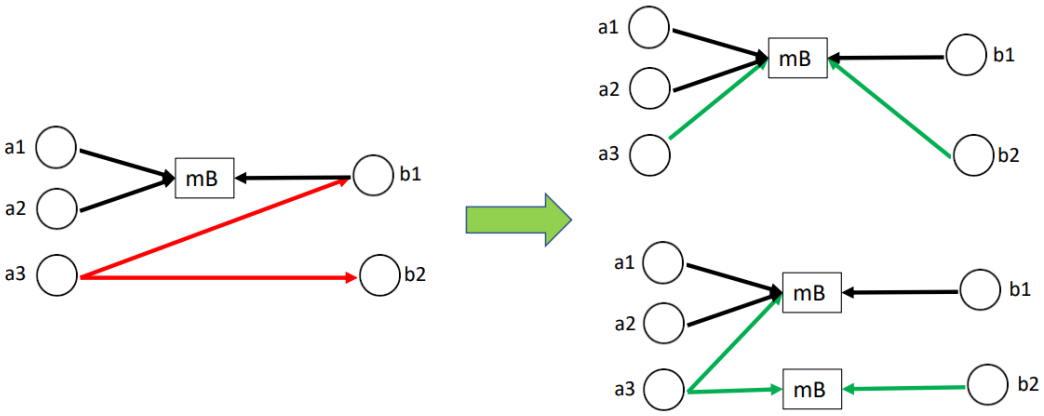


Figure 6.13: Different refactoring proposed

there aren't smells. The same happen for **Kubernetes deployment** in which the `Deployment` object handles the replica set and the `Service` object introduces a message router among instances, deleting any of the already mentioned smells.

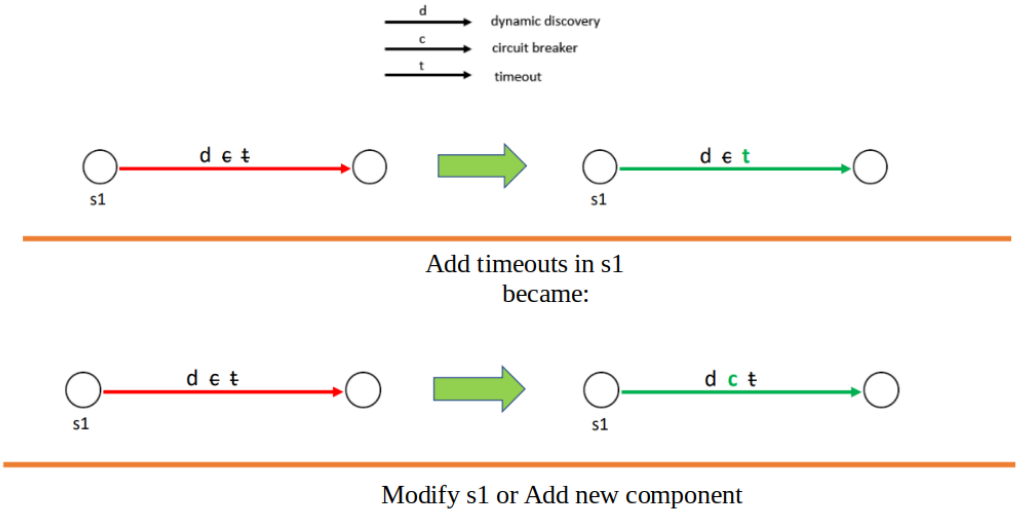


Figure 6.14

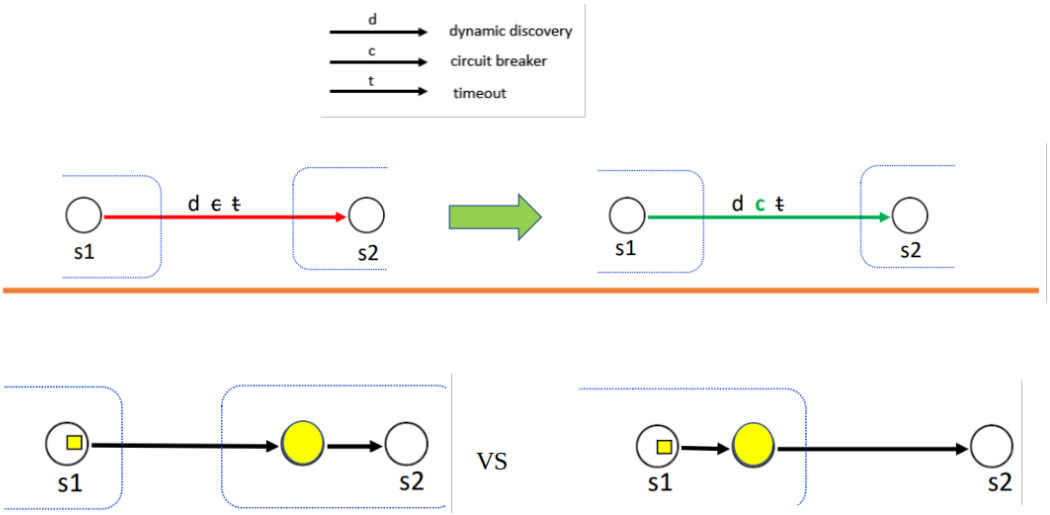


Figure 6.15

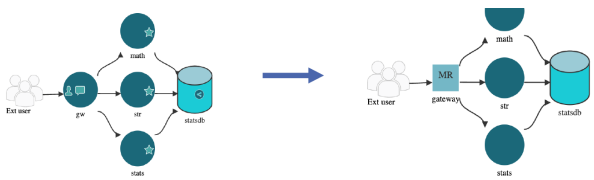


Figure 6.16

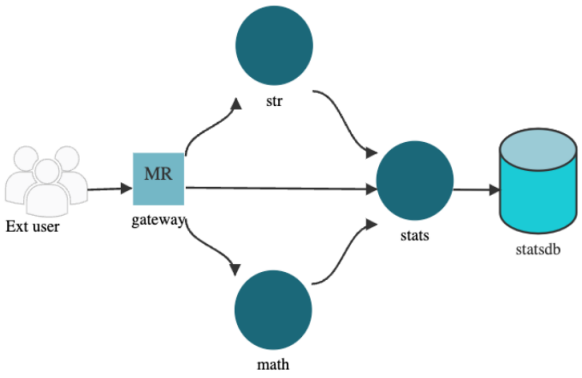


Figure 6.17

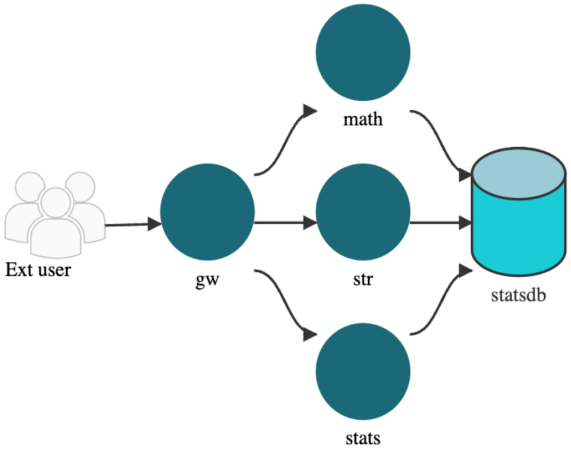


Figure 6.18

Chapter 7

Security and Privacy

Software security should always be a high priority for product developers and their users: if you don't prioritize security, you and your customers will inevitably suffer losses from malicious attacks. In the worst case, these attacks could can put product providers out of business: if their product is unavailable or if customer data is compromised, customers are liable to cancel their subscriptions. Even if they can recover from the attacks, this will take time and effort that would have been better spent working on their software. There are three types of security threat (7.1):

- Availability threats: An attacker attempts to deny access to the system for legitimate users
- Integrity threats: An attacker attempts to damage the system or its data.
- Confidentiality threats: An attacker tries to gain access to private information held by the system

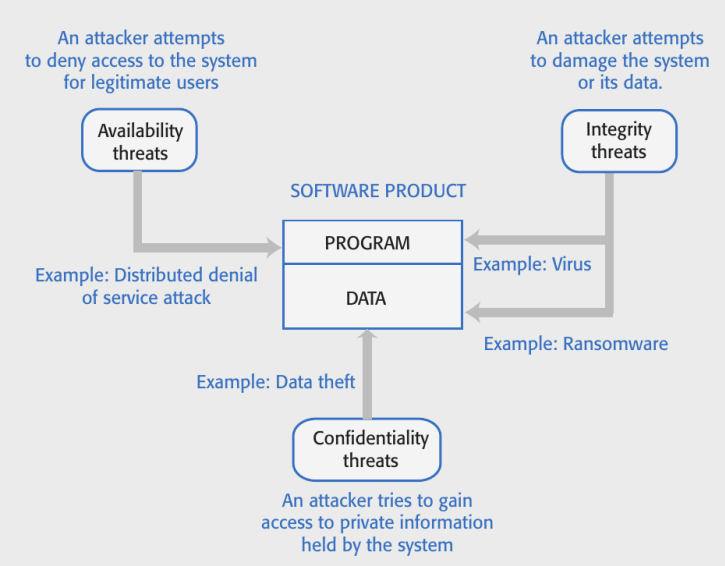


Figure 7.1: Types of security threat

Security is a sistem-wideissue: application software depends on operating system, web server, language run-time system, database, frameworks and tools. Attacks may target any level of the system infrastructure stack, starting from the network, as pictured in 7.2. We need these **System management activities** to mantain security:

- 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 effects of attack

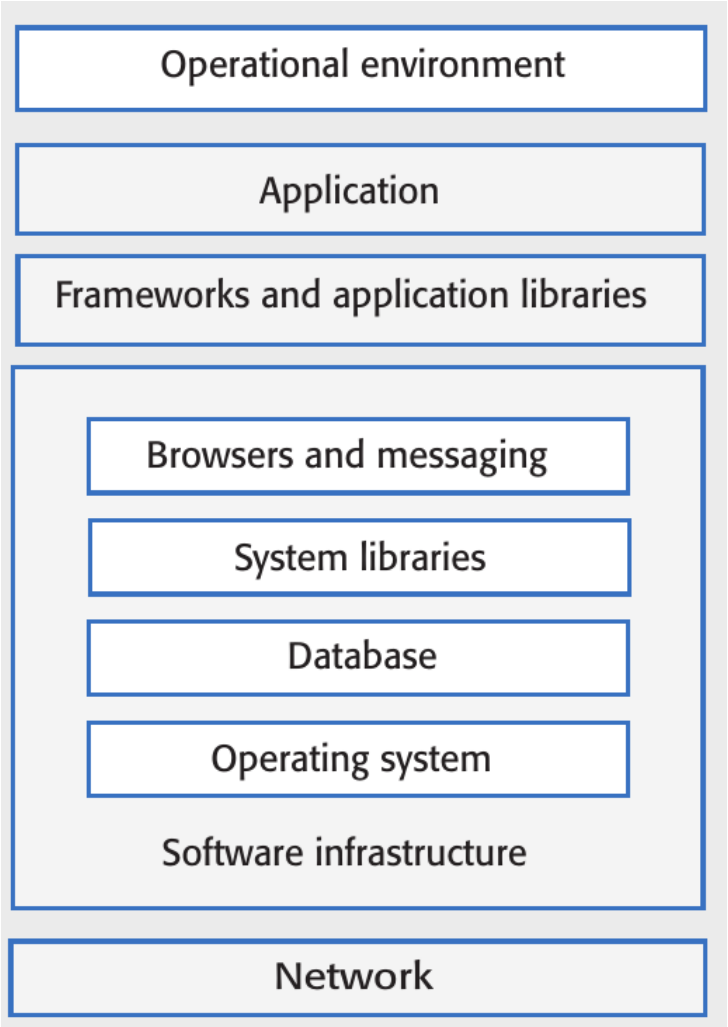


Figure 7.2: Technology stack as attack surface

- Backup policies to keep undamaged copies of program and data files that can be restored after an attack

This help users to mantain security: for example, by enabling multifactor authentication and auto-logout to reduce unauthorized access or user command logging to help diagnosis and recovery, or to deter malicious actors.

7.0.1 Attacks and defenses

Injecton attack

The first attack we describe is called **Injection attack**: malicious user uses a valid input field to input malicious code or database commands to damage the system. Common type of injectiton attack include buffer overflow attacks and SQL poisoning attacks. An example of the former can be represented by the following scenarios:

- e.g. on operating systems/libraries written in C/C++, which do not check whether array assignments are within array bounds
- attacker can carefully craft input string that includes executable instructions and overwrites memory
- if a function return address is overwritten, control can be transferred to malicious code

Here an example of **SQL poisoning attack**: attacker can do injection attack when user input is part of an SQL command.

```
accNum = getAccountNumber ()
SQLstat = "SELECT * FROM AccountHolders WHERE accountnumber = '"
+ accNum + "';" database.execute (SQLstat)
```

Without **check input validity**, this query can be exploited by inserting `'110010010' OR '1'='1'`.

Session Hijacking

Another type of attacks is represented by **session hijacking**: a **session** is a period of time during which user's authentication with a web app is valid. Generally the session cookie (*the token*) is sent from server to client which sends the token in each HTTP request so user doesn't have to re-authenticate for subsequent system interactions. The session is closed when user logs out or when system times out. In this type of attack the attacker acquires valid session cookie and impersonates a legitimate user: he can get the token via another type of attack called **cross-site scripting attack** or by traffic monitoring (*easy on un-secured WiFi networks and unencrypted HTTP request*). We can distinguish two types of hijacking:

- **Passive**: attacker monitors client-server traffic looking for valuable information (*like password or credit card numbers*)
- **Attive**: attacker carries out user actions on server

The **defenses** adopted can be to encrypt client-server network traffic via HTTPS or use multi-factor authentication to require confirmation of new actions that may be damaging.
As mentioned, we describe the **Cross-site scripting attack** (7.3) via an example: An attacker adds malicious Javascript code to the web page that is returned from a server to a client and this script is executed when the page is displayed in the user's browser. The malicious script may steal customer information or direct them to another website. This may try to capture personal data or display advertisements. Cookies may be stolen, which makes a session hijacking attack possible.
Possible **defenses** are represented by input validation, check input from database before adding it to generated page and employ HTML `"encode"` command.

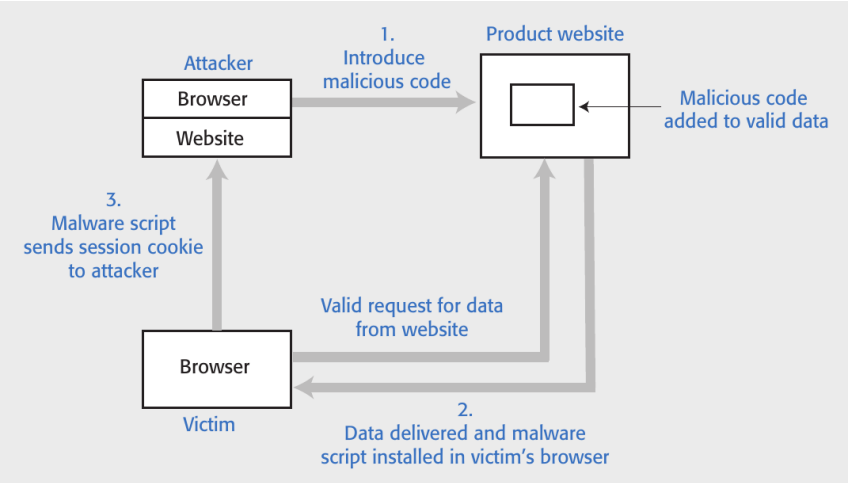


Figure 7.3: Cross-site scripting attack

Denial of Service

Another interesting type of attack is represented by **Denial of Service (DoS)** attack that are intended to make system unavailable for normal use: it's used to boycott server provider or to demand ransom payment. The historical way to carry out a DoS attack was by exploiting the TCP 3-way handshake by exhausting server resources. Nowadays, it's generally called **Distributed DoS** which involve distributed computers that have usually been hijacked to form a **botnet** sending hundreds of thousands of requests for service to a web application: for this type of attack a possible defense can be to detect and drop incoming malicious pakcets. Surely DoS attacks can also target app users by lockout users by repeatedly failing authentication with user email address as login name: some defeses can be represented by temporary user lockouts and IP address tracking and blacklist them.

Brute force attack

Brute force attacks are attacks on a web application where the attacker has some information, such as a valid login name, but does not have the password for the site.The attacker creates different passwords and tries to login with each of these. If the login fails, they then try again with a different password: attackers may use a string generator that generates every possible combination of letters and numbers and use these as passwords.To speed up the process of password discovery, attackers take advantage of the fact that many users choose easy-to-remember passwords. They start by trying passwords from the published lists of the most common passwords. Possible **defenses** are to convince or force users to set long strong password that are not in a dictionary and are not common words or use two-factor authentication.

7.0.2 Authentication

Authentication is the process of ensuring that a user of your system is who they claim to be.You need authentication in all software products that maintain user information, so that only the providers of that information can access and change it. You also use authentication to learn about your users so that you can personalize their experience of using your product. There are many approach to authenticate an user:

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

Knowledge-based authentication often employed for products delivered as cloud services but there are some **weakness** of password based authentication, like:

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

Some defences are to force users set strong password and add knowledge-based authentication (*e.g. user must answer questions*). Surely the level of authentication depends on your product: if there is no need to store confidential information so knowledge-based authentication is enough, differently if there is the need to store such information it's better to enable two-factor authentication.
Implementing a secure and reliable authentication system is expensive and time consuming: even if using available toolkits and libraries (e.g. OAuth), there is still a lot of programming effort involved so usually authentication is outsourced with a federated identity system.

Federated identity

Federated identity is an approach to authentication where you use an external authentication service: 'Login with Google' and 'Login with Facebook' are widely used examples of authentication using federated identity. The **advantage** of federated identity for a user is that they have a single set of credentials that are stored by a trusted identity service. Instead of logging into a service directly, a user provides their credentials to a known service who confirms their identity to the authenticating service. They don't have to keep track of different user ids and passwords because their credentials are stored in fewer places, the chances of a security breach where these are revealed is reduced. The main **drawback** is that the product provider must share user information with external services.
The federated identity verification (7.4) is good for product aimed at individual customers or business product, connecting to business's own identity management system.

Another useful method is the **mobile device authentication** in which we install an authentication token generator on mobile device. There are potential weaknesses if the device is stolen or lost but as an alternative we can use individual user digital certificates issued by trusted providers.

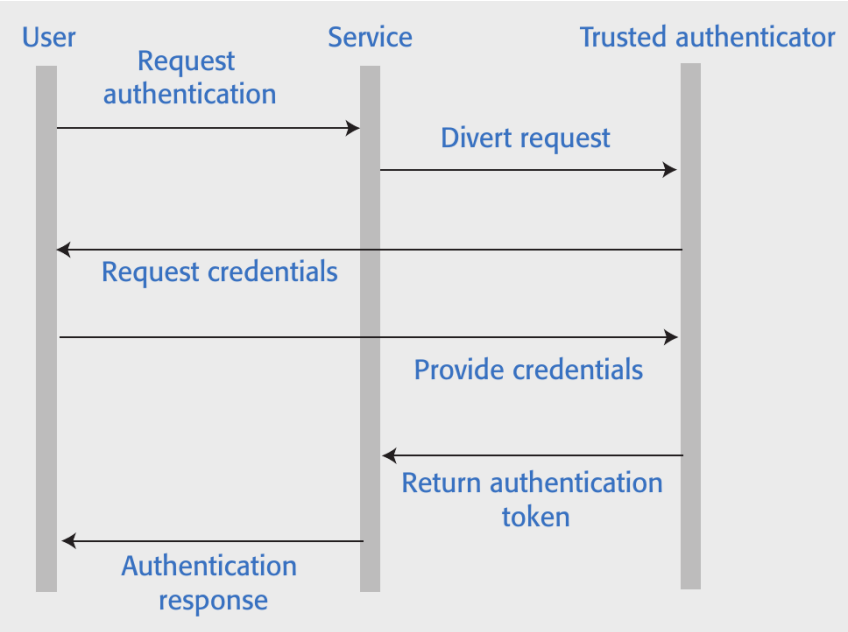


Figure 7.4: Federated identity schema

7.0.3 Authorization

Authentication involves a user proving their identity to a software system: authorization is a complementary process in which that identity is used to control access to software system resources. This policy is a set of rules that define what information (data and programs) is controlled, who has access to that information and the type of access that is allowed: access control policy must reflect data protection rules that limit access to personal data to prevent legal actions in case of data breach. ACLs are widely used to implement access control policies:

- classifying individuals into groups dramatically reduce ACLs size
- different groups can have different rights on different resources
- hierarchies of groups allow to assign rights to subgroups/individuals

ACLs often realized by relying on ACL of underlying file or db system. An example is given in 7.5.

7.0.4 Encryption

Encryption is the process of making a document unreadable by applying an algorithmic transformation to it.A secret key is used by the encryption algorithm as the basis of this transformation. You can decode the encrypted text by applying the reverse transformation.
Modern encryption techniques are such that you can encrypt data so that it is practically uncrackable using currently available technology. However, history has demonstrated that apparently strong encryption may be crackable when new technology becomes available: if commercial quantum systems become available, we will have to use a completely different approach to encryption on the Internet. The different phases of encrypt-decrypt are pictured in 7.6.

In a **symmetric encryption** scheme, the same encryption key is used for encoding and decoding the information that is to be kept secret. If Alice and Bob wish to exchange a secret message, both must have a copy of the encryption key. Alice encrypts the message with this key. When Bob receives the message, he decodes it using the same key to read its contents (see 7.7). The fundamental problem with a symmetric encryption scheme is securely sharing the encryption key. If Alice simply sends the key to Bob, an attacker may intercept the message and gain access to the key. The attacker can then decode all future secret communications.

7.0.5 Asymmetric encryption

Asymmetric encryption, does not require secret keys to be shared: an asymmetric encryption scheme uses different keys for encrypting and decrypting messages. Each user has a public and a private key:

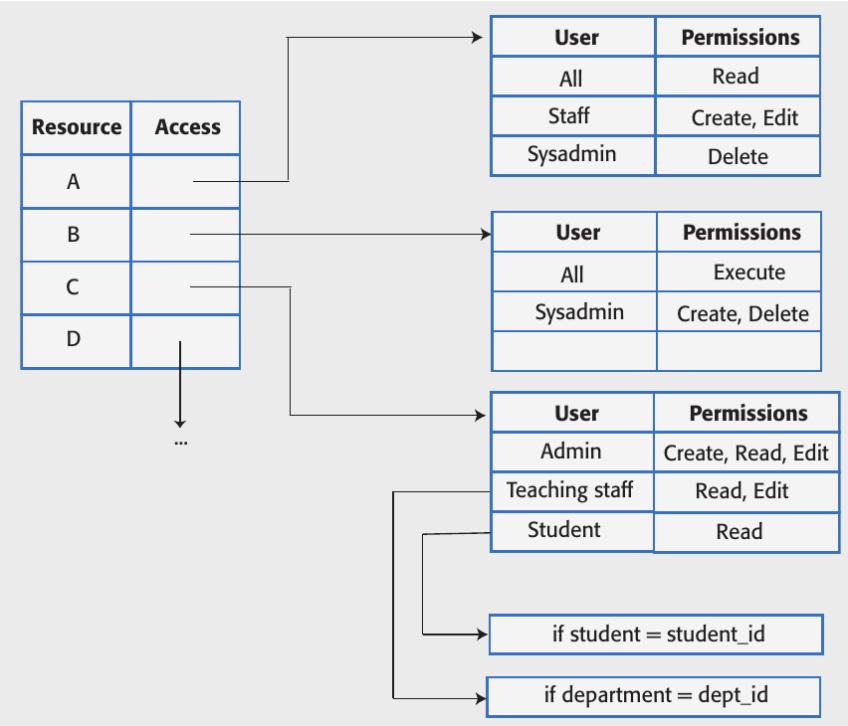


Figure 7.5: Access control lists example

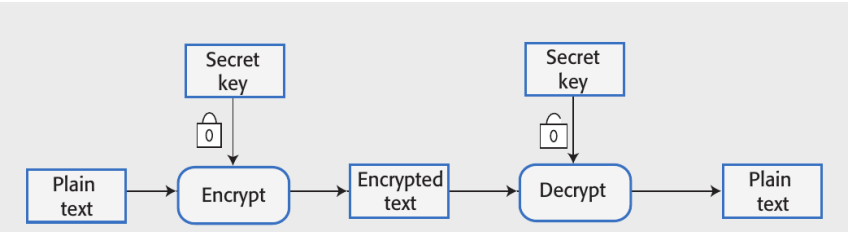


Figure 7.6: Encryption and decryption phases

messages may be encrypted using either key but can only be decrypted using the other key. Public keys may be published and shared by the key owner: anyone can access and use a published public key. However, messages can only be decrypted by the user's private key so is only readable by the intended recipient, as shown in 7.8.

The https protocol is a standard protocol for securely exchanging texts on the web. It is the standard http protocol plus an encryption layer called **TLS (Transport Layer Security)**. This encryption layer is used for 2 things:

1. to verify the identity of the web server;
2. to encrypt communications so that they cannot be read by an attacker who intercepts the messages between the client and the server

TLS encryption depends on a digital certificate that is sent from the web server to the client: Digital certificates are issued by a certificate authority (CA), which is a trusted identity verification service. The CA encrypts the information in the certificate using their private key to create a unique signature. This signature is included in the certificate along with the public key of the CA. To check that the certificate is valid, you can decrypt the signature using the CA's public key. An example is pictured in 7.9.

As a product provider you inevitably store information about your users and, for cloud-based products, user data. **Data encryption** can be used to reduce the damage that may occur from data theft: if information is encrypted, it is impossible, or very expensive, for thieves to access and use the unencrypted data. You should encrypt user data whenever it's practicable to do so:

- Data in transit: When transferring the data over the Internet, you should always use the https rather than the http protocol to ensure encryption.

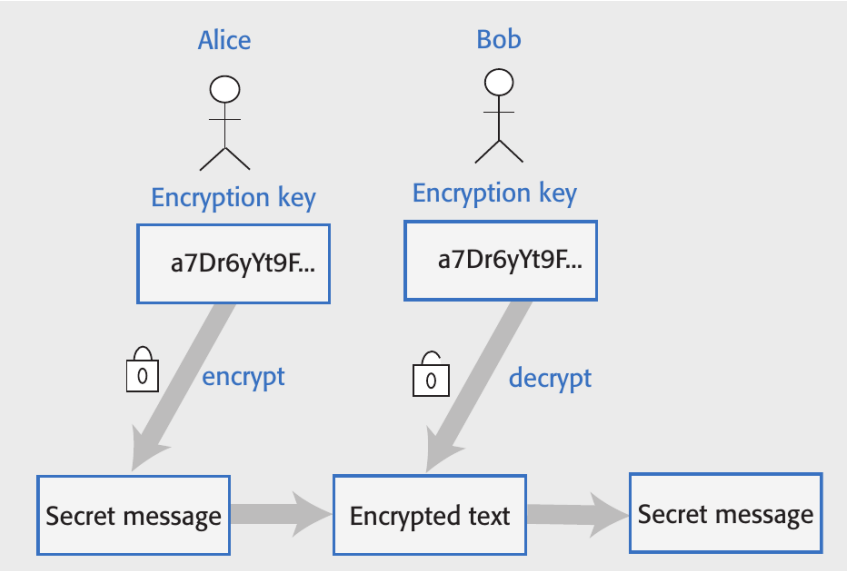


Figure 7.7: Symmetric encryption example

- Data at rest: If data is not being used, then the files where the data is stored should be encrypted so that theft of these files will not lead to disclosure of confidential information.
- Data in use: The data is being actively processed. Encrypting and decrypting the data slows down the system response time. Implementing a general search mechanism with encrypted data is impossible,

Data encryption is possible at four different levels (as shown in 7.10) in the system, despite performance issues that can be partially solved by key management systems.

7.0.6 Key Management

Key management is the process of ensuring that encryption keys are securely generated, stored and accessed by authorized users. Businesses may have to manage tens of thousands of encryption keys so it is impractical to do key management manually and you need to use some kind of automated key management system (KMS).(7.11).

7.0.7 Privacy

Privacy is a social concept that relates to the collection, dissemination and appropriate use of personal information held by a third-party such as a company or a hospital.
There are many business reasons for paying attention to information privacy: If you are offering a product directly to consumers and you fail to conform to privacy regulations, then you may be subject to legal action by product buyers or by a data regulator. If your conformance is weaker than the protection offered by data protection regulations in some countries, you won't be able to sell your product in these countries. If your product is a business product, business customers require privacy safeguards so that they are not put at risk of privacy violations and legal action by users. If personal information is leaked or misused, even if this is not seen as a violation of privacy regulations, this can lead to serious reputational damage.
Customers may stop using your product because of this so general principles to follow when manage privacy 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.

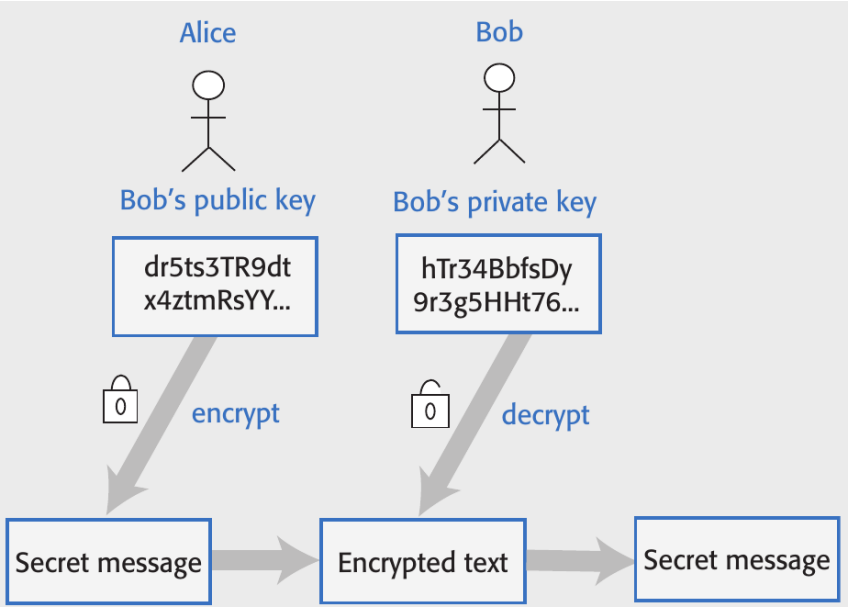


Figure 7.8: Asymmetric encryption example

- **Data lifetime**: You must not keep data for longer than you need to. If a user deletes their 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 that 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 the stronger data protection rules will be upheld.

The information that your software needs to collect depends on the functionality of your product and on the business model you use: it's better to avoid personal information that you do not need by establish a privacy policy defining how personal or sensitive information about users is collected, stoed 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.

7.1 Security and microservices

Discussing security in microservices, we need to make a comparison respect to monolith approach to security. In a monolith architecture the inter-component communication happen inside a single process, despite in microservices the broader attack surface represent an higher risk because the inter-service communication happen via remote calls so there is a large number of entry points. An example is shown in 7.12.

In a monolith architecture a security screening it's execute once and request are dispatched to corresponding componenet. Differently, in microservices each microservice has to carry out independent security screening that may need to connect to a remote security token service. The repeated distributed security checks surely also have an impact on services performance. As work-around we can use the notion of **trust-the-network** for which we skip security checks at each (*internal*) microservices: this is a different philosophy compared to the industry trends towards a **zero-trust** networking principles.
As mentioned, service-to-service communication must take place on protected channels: for example, if we use **certificates**, each microservice must be provisioned with a certificate and a corresponding private key to authenticate itself to another microservice during interactions. The recipient microservice must know how to validate the certificate associated with calling microservice: there is a way to boostrap trust between microservices by introducing **automation** for passing secrets to a large-scale deployments of hundreds of microservices.

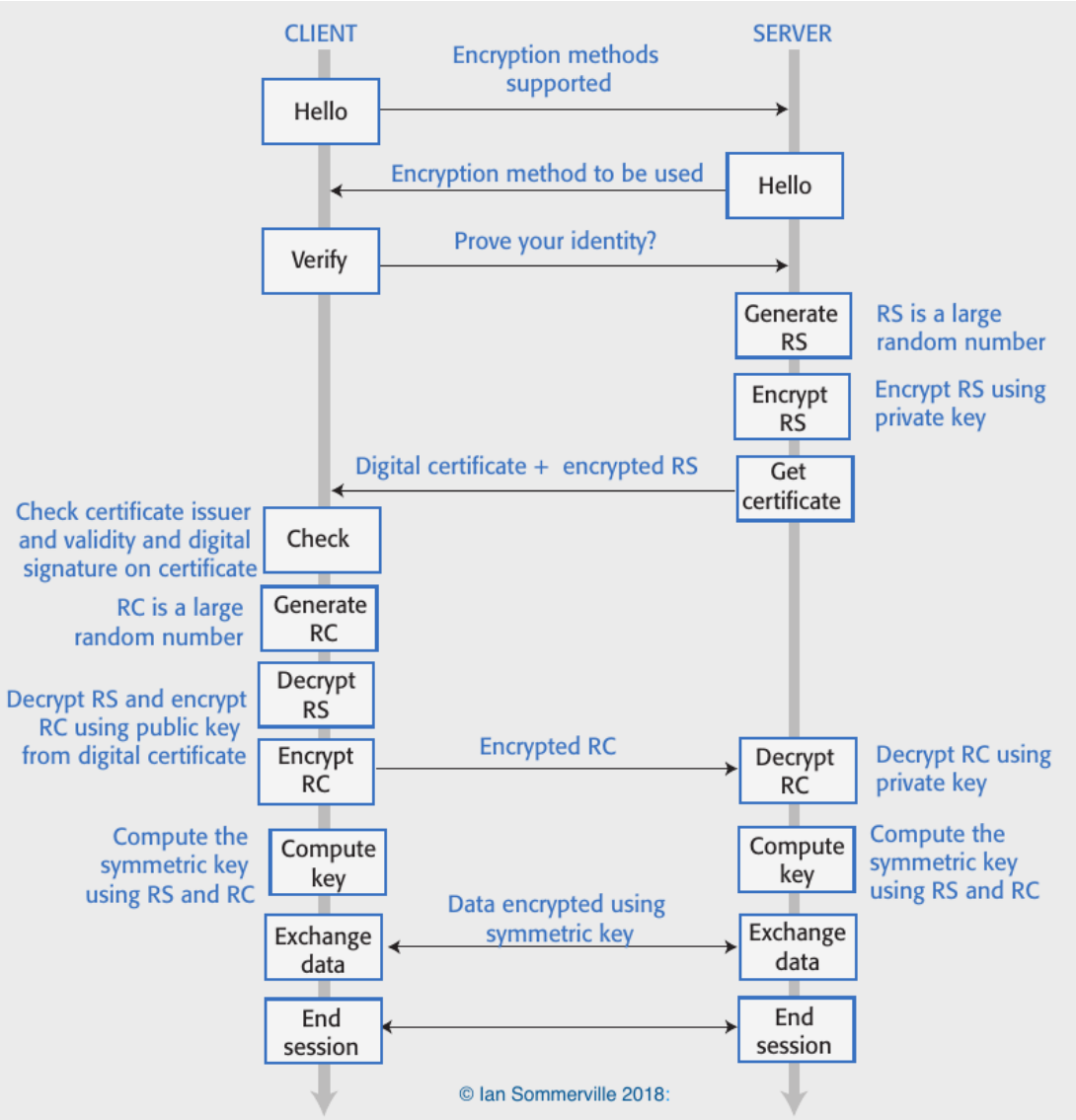


Figure 7.9: Using symmetric and asymmetric encryption in TLS

Another point is represented by **tracing** requests: we define a *log* as the recording of an event of a service. Logs can be aggregated to produce metrics that reflect the system state and that may trigger alert. Traces help track a request from point where it eneters the system to the point where it leaves the system. Differently from monolithic applications, a request to a microservices deployment may span multiple microservices so **correlating request among microservices is challenging**: some usefull tools are represented by **Grafana, Prometheus** to monitor incoming request or by **Jaeger and Zipkin** for distributed racing.
As mentioned, **containers are immutable server** so they don't change state after spin up but for each service we need to mantain a dynamic list of allowed clients and dynamic set of Access-Control policies so each service must also mantain its own credentials which need to be rotated periodically (*by keeping credentials in container filesystem and inject them at boot time*).
In a distributed environment the sharing of user context is harder than in a monolithic architecture: we need to build trust between microservices to that receiving microservice accepts user context passed from the calling microservice. A popular solution is to use **JSON Web Token (JWT)** to share user context among microservices: the key idea is that the message carrying attributes ina cryptographically safe way (*by encryption, encoding or both the data to transfer*).

To summarize what we said:

- The broader the attack surface, the higher the risk
- Distributed security screening affects performance

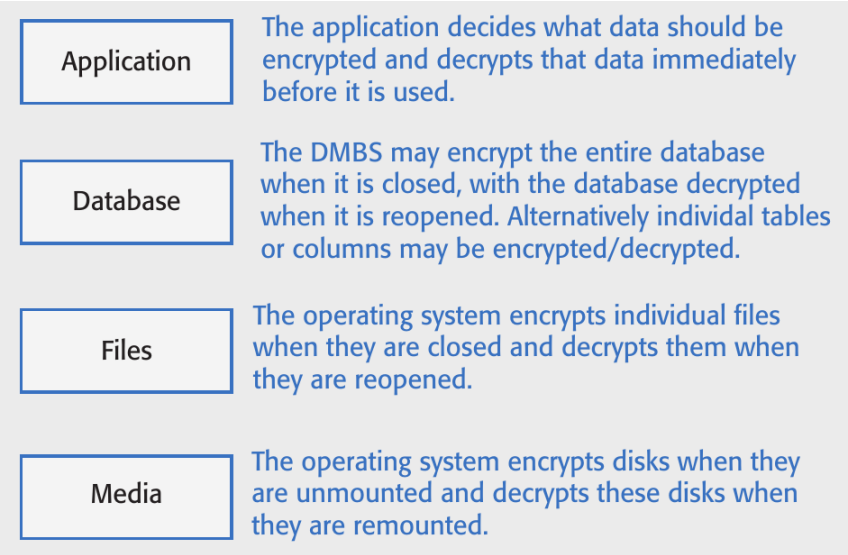


Figure 7.10: Different encryption levels

- Bootstrapping trust among microservices needs automation
- Tracing requests spanning multiple microservices is challenging
- Containers complicate credentials/policies handling
- Distribution makes sharing user context harder
- Security responsibilities distributed among different teams

7.1.1 Smells and refactoring for microservices security

Review of white and grye literature (*58 references*) to answer two research questions:

1. W*hich are the most recognised smells indicating possible security violations in microservice-based applications?*
2. *How to refactor a microservice-based application to mitigate the effect of a security smell?*

The analysis result in a taxonomi with **10 refactoring** associated with **10 smells** potentially affect **3 security properties**. The properties are taken by *ISO/IEC 25010* software quality standard:

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

Here we analyze some potential properties violation by identify a smell and propose a refactoring:

- *Property:* **Confidentiality**, *Smell:* **Insufficient access control** *Refactoring*: **Use OAuth 2.0**
  Microservice-based application does not enact access control in some microservices. Potential "confused deputy problem" with attacker getting data it shouldn't be able to get so there is a potential violation of confidentiality of data (and business functions).Client permissions need to be verified at request time so client identity should be verified without introducing extra latency and contention with frequent calls to a centralized service.
  Most suggested refactoring are to employ OAuth 2.0 by using token-based security framework for delegated access control and by allowing resource owner grant client access to a resource on its behalf. The access is granted for limited time and with limited scope.

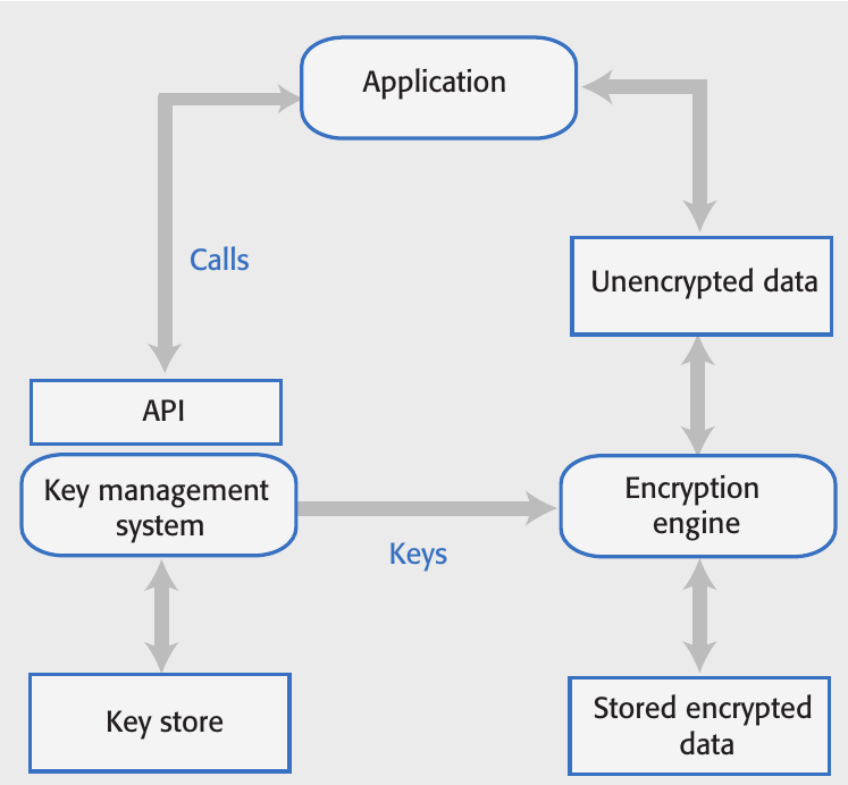


Figure 7.11: Using a KMS for encryption management

- *Property:* **Confidentiality**, *Smell:* **Publicly accessible microservices** *Refactoring*: **Add API gateway**
  Some microservices directly accessible by external clients: each microservice must check authentication and authorization for each request. This increase the exposure of credentials so increase the likelihood of confidentiality violations with an higher cost of application maintenance.
  Most cited refactoring is to add an API gateway: microservices are not directly accesible by external client and can enforce authentication, authorization and message content validation by allowing the microservice to stay behing a firewall.

- *Property:* **Confidentiality, Integrity**, *Smell:* **Unnecessary privileges to microservices** *Refactoring*: **Follow the least privilege principle**

  Microservices are granted unnecessary access levels, permissions, or functionalities that are not needed to deliver their business functions so resources are unnecessarily exposed: this increase the attack surface against confidentiality and integrity. The refactoring propose to use the **Least privilege principle** for which all running code have the permissions only needed to complete the required thask and no more.

- *Property:* **Confidentiality, Integrity, Authenticty**, *Smell:* **"Home-made" crypto code** *Refactoring*: **Use established encription techniques**

  Use of "home-made" crypto code can cause confidentiality, integrity and authenticity issues: this can get even worse than no encryption (false sense of security). The solution is to exploit encryption libraries heavily tested and patched and recommend to avoid experimental encryption algorithms.

- *Property:* **Confidentiality, Integrity, Authenticty**, *Smell:* **Non-encrypted data exposure** *Refactoring*: **Encypt all sensitive data at rest**

  Microservice-based application accidentally exposes sensitive data so intruders can access or modify data, including credentials. The solution is to encrypt all sensitive data at rest and decrypt only when needed by exploiting DBMSs support for automatic encryption or OSs support for disk-level encryption. Other solution are to use application-level encryption or to encrypt also cached data.

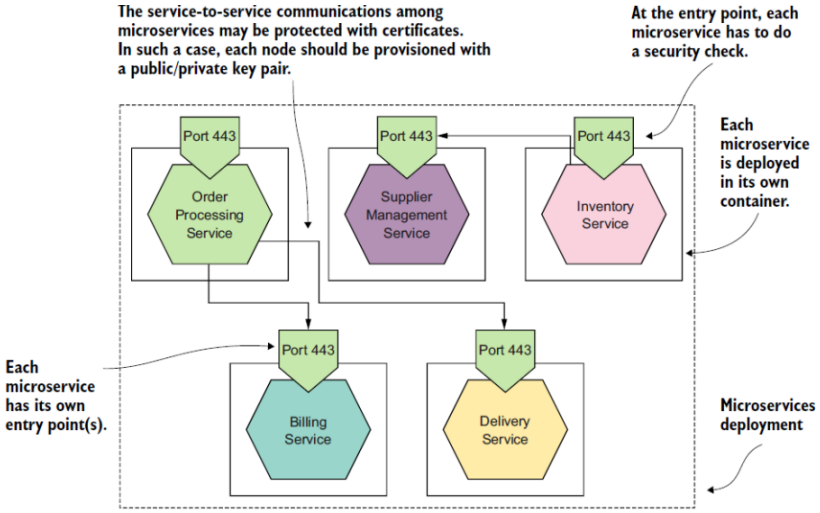


Figure 7.12: Broader attack surface: an example

- *Property:* **Confidentiality, Integrity, Authenticty**, *Smell:* **Hardcoded secrets** *Refactoring*: **Encrypt secrets at rest**

  Hardcoded secrets (e.g. API keys, client secrets, credentials) contained in microservice source code or in application deployment scripts (e.g. environment variables passing secrets in a Docker Compose spec) represent a security flaw. Never store secrets in environment variables because they could be accidentaly exposed. The solution is to encrypt secret at rest by do not store credentials alongside application source code but use effective mechanism to pass secrets at runtime from a secured source.

- *Property:* **Confidentiality, Integrity, Authenticty**, *Smell:* **Non-secured service-to-service communications** *Refactoring*: **Use mutual TLS**

  Microservices interacting without enacting a secure communication channel so data is exposed to man-in-the-middle, eavesdropping, and tampering attacks. Solutions are to use TLS by encrypting data in transit and ensuring intergrity. Mutual TLS also allows to legitimately identify a microservice that is talking by mutual authentication.

- *Property:* **Authenticty**, *Smell:* **Unauthenticated traffic** *Refactoring*: **Use mutual TLS and Use OpenID connect**

  If traffic not authenticated then microservices are exposed to security attacks like tampering with data, denial of service, or elevation of privileges. Proposed refactorings are to use Mutual TLS or to use OpenID Connect by using JWT containing authenticated user info so microservices can verify user identity with authorization server.

- *Property:* **Authenticty**, *Smell:* **Multiple user authentication** *Refactoring*: **Add API Gateway, use OpenID connect and use single sign-on**

  Application provides multiple access points to handle user authentication: each access point constitutes a potential attack vector for intruder to authenticate as end user. Use a Single Sign-On approach as a single entry point to handle user authentication and to enforce security for all user requests thus facilitating log storage and auditing.
  Single sign-on can be achieved by adding API gateway (if not there), and employing OpenID Connect (to share user contexts).

- *Property:* **Authenticty**, *Smell:* **Centralised authorization** *Refactoring*: **Use decentralised authorization**

Authorization can be enforced at the edge of the application (API gateway) and/or by each microservice of the application: if application only handles authorization at the edge, "central" authorization point becomes bottleneck reducing performance. Solution is represented by enacting a decentralized authorization approach: transmit an access token (e.g. JWT) together with each request to a microservice and access to microservice is granted to caller only if a known and correct token is passed.

A summary is shown in 7.13.

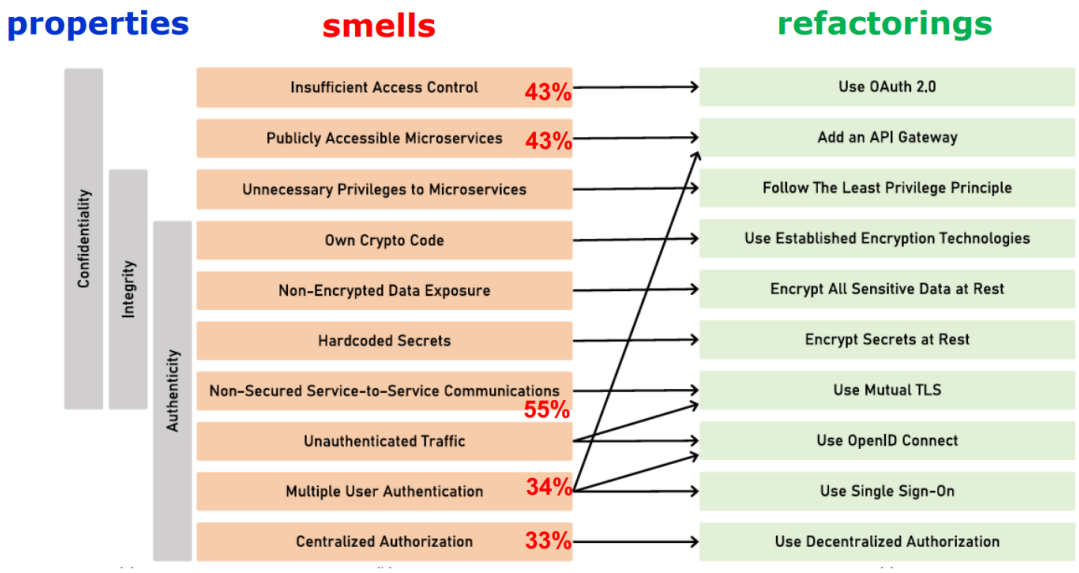


Figure 7.13: Smells and refactoring: summary

To decide if refactor or not refactor we need to have a complete overview of *securirty properties, design principles, performance efficiency and mantainability*: modelling softgoal interdependencies as graphs can help visualization and automated trade-off analysis, as shown in 7.14.

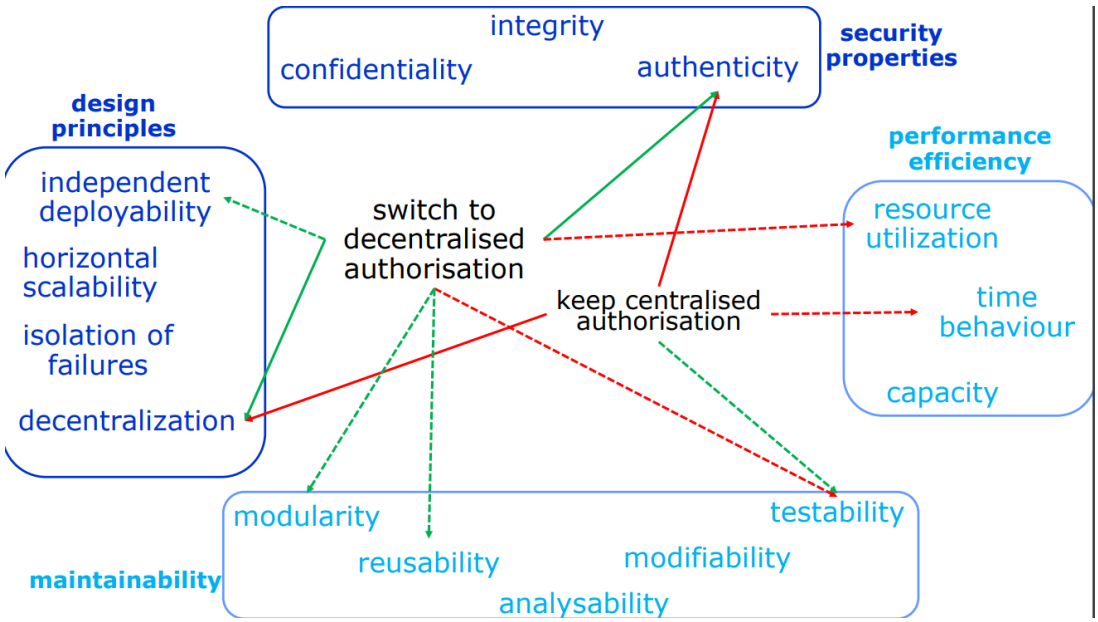


Figure 7.14: Graph visualization

7.2 OWASP API Security

Complexity of source code leads to more security vulnerabilities so exponential increase of defects as number of lines of code increases. Functional and security testing is utterly important: we distinguish two types of testing:

- Static: *Bandit* is a static analysis tool designed to find common security issues in Python code, by exploiting known patterns (plugins).
- Dynamic: see next

First, we discuss about **API-based app** in a context where client devices are getting more powerful: the logic moves from backend to frontend, for example servers are used more as data proxies and client render the data. Client also can consume raw data and mantain/monitor user's status. This implies that the HTTP request a have a lots of parameter and API disclose these information on implementation side. API expose microservices to consumers: it's therefore important to make them secure and avoid known pitfalls. The **top 10 API Vulnerabilities** are:

1. **Broken object-level authorization**: Attacker substitutes ID of their resource in API call with an ID of a resource belonging to another user. Lack of proper authorization checks allows access. This attack is also known as IDOR (Insecure Direct Object Reference). The **Use cases** are:
   - API call parameters use IDs of resourced accessed by the API: `/api/shop1/financial_details`
   - Attackers replace the IDs of their resources with different ones, which they guessed: `/api/shop2/financial_details`
   - The API does not check permissions and lets the call through
   - Problem is aggravated if IDs can be enumerated: `/api/123/financial_details`

   There are several way to prevent his type of vulnerability:
   - Implement authorization checks with user policies and hierarchy
   - Don't rely on IDs sent from client. Use IDs stored in the session object instead.
   - Check authorization each time there is a client request to access database
   - Use random non-guessable IDs (UUIDs)
2. **BROKEN AUTHENTICATION**: Poorly implemented API authentication allowing attackers to assume other users' identities.The **Use cases** are:
   - Unprotected APIs that are considered "internal"
   - Weak authentication not following industry best practices
   - Weak, not rotating API keys
   - Weak, plain text, encrypted, poorly hashed, shared/default passwords
   - Susceptible to brute force attacks and credential stuffing
   - Credentials and keys in URL
   - Lack of access token validation (including JWT validation)
   - Unsigned, weakly signed, non-expiring JWTs

   There are several way to prevent his type of vulnerability:
   - Check all possible ways to authenticate to all APIs Password reset APIs and one-time links also allow users to get authenticated and should be protected just as seriously
   - Use standard authentication, token generation, password storage, Multi-factor authentication
   - Use short-lived access tokens
   - Authenticate your apps (so you know who is talking to you)
   - Use stricter rate-limiting for authentication, implement lockout policies and weak password checks
3. **Excessive data exposure**: API exposing a lot more data than the client legitimately needs, relying on the client to do the filtering. Attacker goes directly to the API and has it all. The **Use cases** are:

- APIs return full data objects as they are stored by the database
- Client application shows only the data that user needs to see
- Attacker calls the API directly and gets sensitive data

There are several way to prevent his type of vulnerability:

- Never rely on client to filter data
- Review all responses and adapt responses to what the API consumers really need
- Define schemas of all the API responses
- Don't forget about error responses
- Identify all the sensitive or PII info and justify its use
- Enforce response checks to prevent accidental data and exception leaks

4. **LACK OF RESOURCES & RATE LIMITING**: API is not protected against an excessive amount of calls or payload sizes. Attackers use that for DoS and brute force attacks. The **Use cases** are:

   - Attacker overloading the API
   - Excessive rate of requests
   - Request or field sizes
   - "Zip bombs"

   There are several way to prevent his type of vulnerability:

   - Rate limiting
   - Payload size limits
   - Rate limits specific to API methods, clients, addresses
   - Checks on compression ratios
   - Limits on container resources leaks

5. **BROKEN FUNCTION LEVEL AUTHORIZATION**: API relies on client to use user level or admin level APIs. Attacker figures out the "hidden" admin API methods and invokes them directly. The **Use cases** are:

   - Some administrative functions are exposed as APIs
   - Non-privileged users can access these functions if they know how
   - Can be a matter of knowing the URL, using a different verb or parameter

   There are several way to prevent his type of vulnerability:

   - Don't rely on app to enforce admin access
   - Deny all access by default
   - Grant access based on specific roles
   - Properly design and test authorization

6. **SECURITY MISCONFIGURATION**:Poor configuration of the API servers allows attackers to exploit them. The **Use cases** are:

   - Unpatched systems
   - Unprotected files and directories
   - Unhardened images
   - Missing, outdated, misconfigured TLS
   - Exposed storage or server management panels
   - Missing CORS policy or security headers
   - Error messages with stack traces
   - Unnecessary features enabled

There are several way to prevent his type of vulnerability:

- Repeatable hardening and patching processes
- Automated process to locate configuration flaws
- Disable unnecessary features
- Restrict administrative access
- Define and enforce all outputs including errors

7. **MASS ASSIGNMENT**: The **Use cases** are:

   - API working with the data structures
   - Received payload is blindly transformed into an object and stored, like:

```
NodeJS:
var user = new User(req.body);
user.save();

Rails:
@user = User.new(params[:user])
```

   - Attackers can guess the fields by looking at the GET request data

   There are several way to prevent his type of vulnerability:

   - Don't automatically bind incoming data and internal objects
   - Explicitly define all the parameters and payloads you are expecting
   - For object schemas, use the readOnly set to true for all properties that can be retrieved via APIs but should never be modified
   - Precisely define at design time the schemas, types, patterns you will accept in requests and enforce them at runtime

8. **INJECTION**: Attacker constructs API calls that include SQL-, NoSQL-, LDAP-, OS- and other commands that the API or backend behind it blindly executes.The **Use cases** are:

   - Attackers send malicious input to be forwarded to an internal interpreter: SQL, NoSQL, LDAP, OS commands, XML parsers, Object-Relational Mapping (ORM).

   There are several way to prevent his type of vulnerability:

   - Never trust your API consumers, even if internal
   - Strictly define all input data: schemas, types, string patterns - and enforce them at runtime
   - Validate, filter, sanitize all incoming data
   - Define, limit, and enforce API outputs to prevent data leaks

Chapter 8

DevOps and Code Management

Traditionally, separate teams were responsible software development, software release and software support. The development team passed over a 'final' version of the software to a release team. This team then built a release version, tested this and prepared release documentation before releasing the software to customers and a third team was responsible for providing customer support. The original development team were sometimes also responsible for implementing software changes.
Alternatively, the software may have been maintained by a separate 'maintenance team'. This inevitable introduce delays and overheads in the traditional support model: some issue were communication delays between teams, separate teams use different tolls, have different skills and often don't uderstand other's problems and this generally needed days to fix urgent bug or security vulnerabilities. To speed up the release and support processes, an alternative approach is called **DevOps** or **Dvelopment+Operations**. Three factors enabling a change:

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

So DevOps integrates development, deployment and support in a single team. The DevOps philosophy follows some principles:

- *Everyone is responsible for everything*: All team members have joint 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

This also allows some **benefits** like:

- *Faster deployment*: Dramatic reduction of human communication delays → faster deployment to production
- *Reduced risk*: Small functionality increment in each release → less chance of feature interactions and system failures/outages
- *Faster repair*: No need to discover which team is responsible for fixing problem and to wait for them to fix it
- *More productive teams*: DevOps teams more productive than teams involved in separate activities

So the DevOps culture means creating team bringing together a number of different skillset like *software eng, UX design, security engineering, infrastructure engineering, customer interaction.* To create a successfull team should be present a culture of mutual respect and sharing in which everyone on the team should be involved in Scrums and other team meetings by encouraging team members to share their expertise with other and to learn new skills. Developers should also support software services that they

have developed so if a service fails, deeveloper is responsible for getting it up and running again or if the developer who created it it's unavailable other member's team can take over the task. The team priority is to *fix failures as quickly as possible.*

8.0.1 Code management

During the development of a software product, the development team will probably create tens of thousands of lines of code and automated tests. These will be organized into hundreds of files. Dozens of libraries may be used, and several, different programs may be involved in creating and running the code. **Code management** is a set of software-supported practices that is used to manage an evolving codebase. You need code management to ensure that changes made by different developers do not interfere with each other, and to create different product versions. Code management tools make it easy to create an executable product from its source code files and to run automated tests on that product, as shown in 8.1.
A example here:
Alice and Bob worked for a company called FinanceMadeSimple and were team members involved in developing a personal finance product. Alice discovered a bug in a module called TaxReturnPreparation. The bug was that a tax return was reported as filed but, sometimes, it was not actually sent to the tax office. She edited the module to fix the bug. Bob was working on the user interface for the system and was also working on TaxReturnPreparation. Unfortunately, he took a copy before Alice had fixed the bug and, after making his changes, he saved the module. This overwrote Alice's changes but she was not aware of this. The product tests did not reveal the bug as it was an intermittent failure that depended on the sections of the tax return form that had been completed. The product was launched with the bug. For most users, everything worked OK. However, for a small number of users, their tax returns were not filed and they were fined by the revenue service. The subsequent investigation showed the software company was negligent. This was widely publicised and, as well as a fine from the tax authorities, users lost confidence in the software. Many switched to a rival product. FinanceMade Simple failed and both Bob and Alice lost their jobs.

The main **objective** of code management is to manage evolving project codebase to allow different versions of components and entire systems to be stored and retrieved so developers can work in parallel without interfering with each other and integrate their work with that from other developers.
This allows benefits also in terms of **feature** like:

- *Code transfer*: Developers download code into personal file store, work on it, return it to shared code management system
- *Version storage and retrieval*: Files may be stored in several different versions, specific versions can be retrieved
- *Branching and merging*: Parallel development branches can be created for concurrent working. Changes made in different branches may be merged
- *Version information*: Information on different versions may be stored and retrieved

So a **source code management system** is usually formed by a **shared repository** so all source code files and file versions are stored in the repository, together with other artifacts (*e.g. configuration files, build scripts, shared libraries, versions of tools*). The repository includes a database of information about the stored files (e.g. version information, author of changes, time of changes). Files can be transferred to/from the repository, information about different versions of files and their relationships can be updated: specific versions of files and information about these versions can always be retrieved from the repository.
The main **features** are:

- Version and release identification: Managed versions of a code file are uniquely identified when submitted to the system so 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, submitter must add a string explaining the reasons of the change. This helps developers understand why new version was created.
- Independent development: Several developers can work on the same code file at the same time When this is submitted to the code management system, a new version is created so that files are never overwritten by later changes. This avoids the file overwriting problem of previous example.

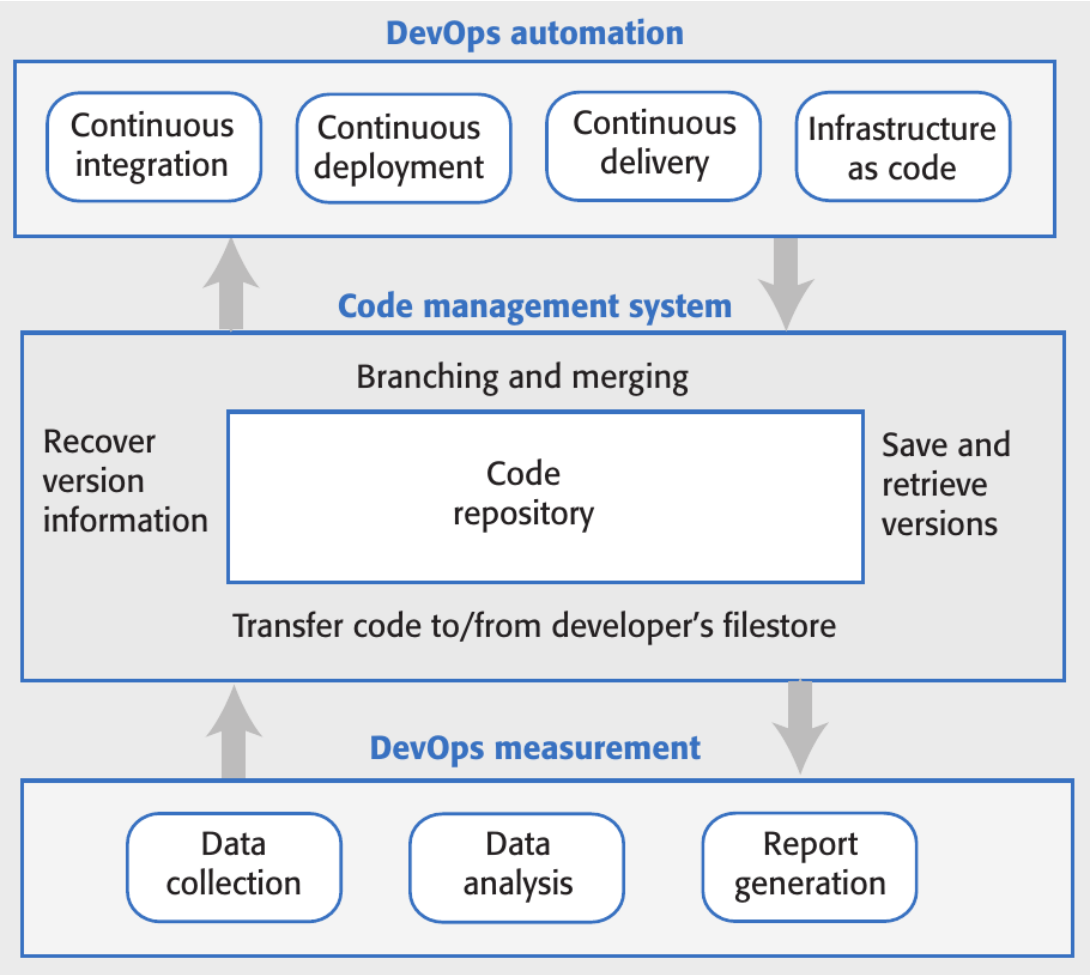


Figure 8.1: Code management and DevOps interaction

- Project support: All files associated with a project may be checked out at the same time
- Storage management: Code management system includes efficient storage mechanisms not to keep multiple copies of files that have only small differences (less dramatic today with cheap storage).

There are two types of source code management:

- **Centralized**: users check in and check out files, the system issues warnings when checking out file already in use and usually at check in new version of file is created, so system ensure that file copies do not conflict.
- **Decentralized**: The most common used nowadays is Git. It's able to upport large-scale open-source development (originally to manage Linux kernel's code). Allows to taking advantage of low storage costs, Git maintains a clone of the repository on every user's computer. The *master branch* is the current master version of software that the team is working on: new versions can be created by creating new branches. When users request a repository clone, they get a copy of the master branch that they can work on independently.

Some advantages of **decentralized repository architecure** are:

- Resilience: Everyone working on a project has own copy of repository. If shared repository damaged/attacked, work can continue, clones can be used to restore shared repository. People can work offline
- Speed: Committing changes to the repository is a fast, local operation. It does not need data to be transferred over the network.
- Flexibility: Local experimentation is much simpler. Developers can safely try different approaches without exposing experiments to others.

Referring to the scenario in 8.2, for example:

- 4 project repositories on Github (RP1–RP4)
- Each developer on each project identified by a letter
- Each developer has individual copy of project repository
- Each developer may work on more than one project at a time

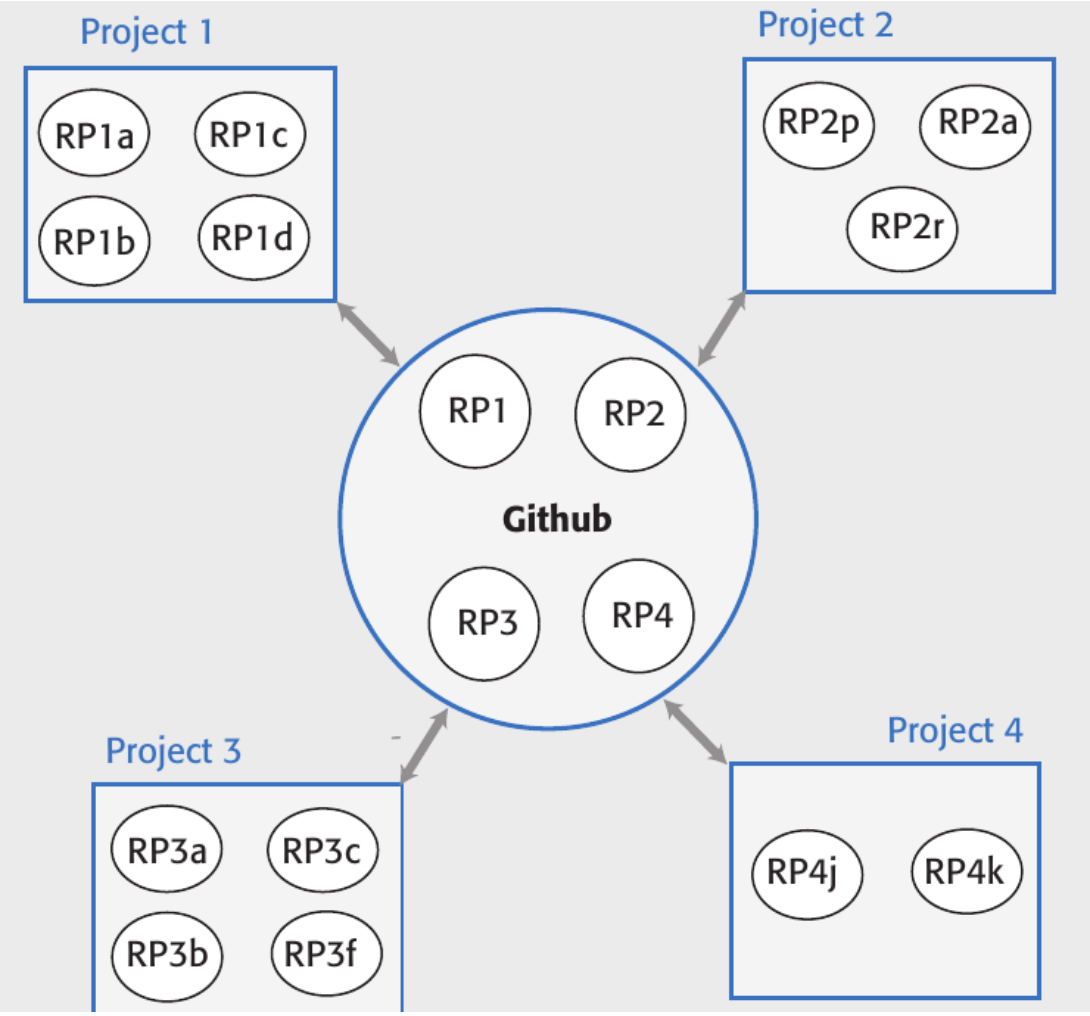


Figure 8.2: Git repository example

The principle for which **"everything that can should be automated"** implies taking advantages of **Continous integration** by producing an exectable version of the system at every commits on master branch, **Continous delivery** by testing the software in a simulated environment, **Continous deployment** by releasing new version to users every time a chainge is made to the master branch and **Infrastructure as Code (IaC)** by designing a machine-readable models of network, server and routers on which the product executes are used by configuration management tolls to build software's execution platform.

8.0.2 Continous Integration

Continuous integration simply means that an integrated version of the system is created and tested every time a change is pushed to the system's shared repository. On completion of the push operation, the repository sends a message to an integration server to build a new version of the product. The advantage of continuous integration compared to less frequent integration is that it is faster to find and fix bugs in the system. If you make a small change and some system tests then fail, the problem almost certainly lies in the new code that you have pushed to the project repo.
System integration involves other activity (8.3) other than compiling, like:

- install db software and set up db with appropriate schema
- load test data into db
- compile the files that make up the product
- link compiled code with libraries and other components used
- check that external services used are operational
- move configuration files to correct locations (and delete old ones)
- run system tests to check that integration has been successful

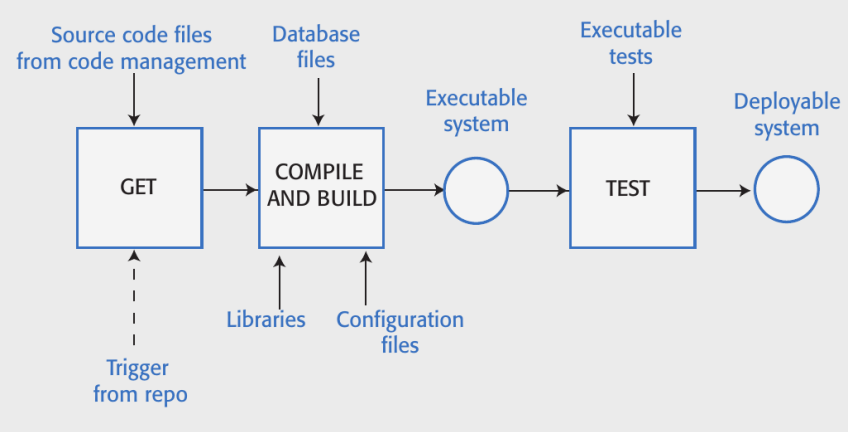


Figure 8.3: Continous integration process

To avoid to *"breaking the build"* it's better to integrate twice by integrate and test the code on local machine and then push the code to project repository to trigger integration server, as pictured in 8.4. Some **advantages** of CI are:

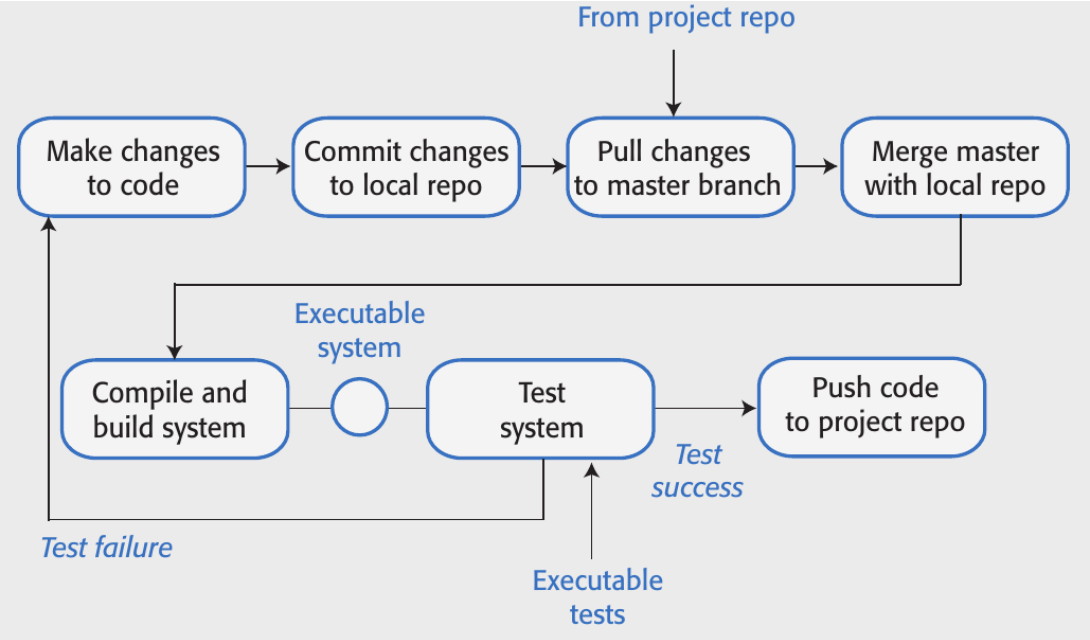


Figure 8.4: Local integration process

- It is faster to find and fix bugs in the system: if you make a small change and some system test fails, the problem almost certainly lies in the new code that you pushed

- A working system is always available to the whole team: It can be used to test ideas and to demo system to management and customers *Q*uality culture in development team: No team members wants the stigma of breaking the build. Everybody checks her work carefully before pushing it to project repo

Code integration tools only repeat actions if dependent files have been changed, for example:

- At first integration, all source code files and executable test files are compiled
- Subsequently, object code files are created only for modified tests and for modified source code files

Some usefull tools for system building are *Unix make, Ant, Maven, Rake (Ruby)* or for CI tools we have *Jenkins, Travis, Bamboo, etc.*
It's necessary to **exploit compositionality** to differentially analyse a large scale system by mainly **focussing on the largest changes** introduced in the system and **re-using preiously computed result** as much as possible. To this aim, it's usually used the **separation logic** that is an extension of *Hoare logic*:

$$\{\texttt{precondition}\}\ \texttt{code}\ \{\texttt{postcondition}\}$$

This allows to model in-place update of memory during execution in terms of precodition sand postconditions on the heap. For example:

$$\begin{gathered}\{x \to 0 * y \to 0\}\\ x = y;\\ y = x;\\ \{x \to y * y \to x\}\end{gathered}$$

The logic for reasoning about threads is called **Concurrenct separation logic**.

8.0.3 Continous Delivery

Continuous Integration creates an executable version of a software system by building system and running tests on your computer or project integration server. Real production environment will differ from development environment because production server may have different filesystem organization, access permissions, installed applications and new bugs may show up.
Continuous delivery ensures that changed system is ready for delivery to customers: feature tests in production environment to make sure that environment does not cause system failures and load tests are executed to check how software behaves as number of users increases.Containers are the simplest way to create a replica of a production environment. The entire process is pictured in 8.5.

It's recommended to deliver after initial integration testing by configure staged test environment (replica of production environment). Also system acceptane tests are executed (functionality, load, performance). To **deploy** software and data are transferred to production server: new service requests stopped, older version processes outstanding transactions. Switch to new system version and restart processing. Some benefits of CD 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: You can deploy new features when they are ready for customer use Users' feedback can be used to to identify improvements
- A/B testing: If you have many customers and several servers you can deploy new software version only on some servers, use load balancer to divert some customers to the new version and measure and assess how new features are used.

Surely it's better to avoid deploying every single change because you may have incomplete features that could be deployed, do not want to disclose this to competitors until implementation is complete or customers may be irritated by continually changing software, especially if this affects UI. Generally you want to synchronize your releases with known business cycles.
Some usefull tools are *Jenkins and Travis*: these tools can integrate with infrastructure configuration management tools such as Chef and Puppet and Ansible to implement software deployment. For cloud-based software, it is often simpler to use containers in conjunction with CI tools rather than use infrastructure configuration management software.

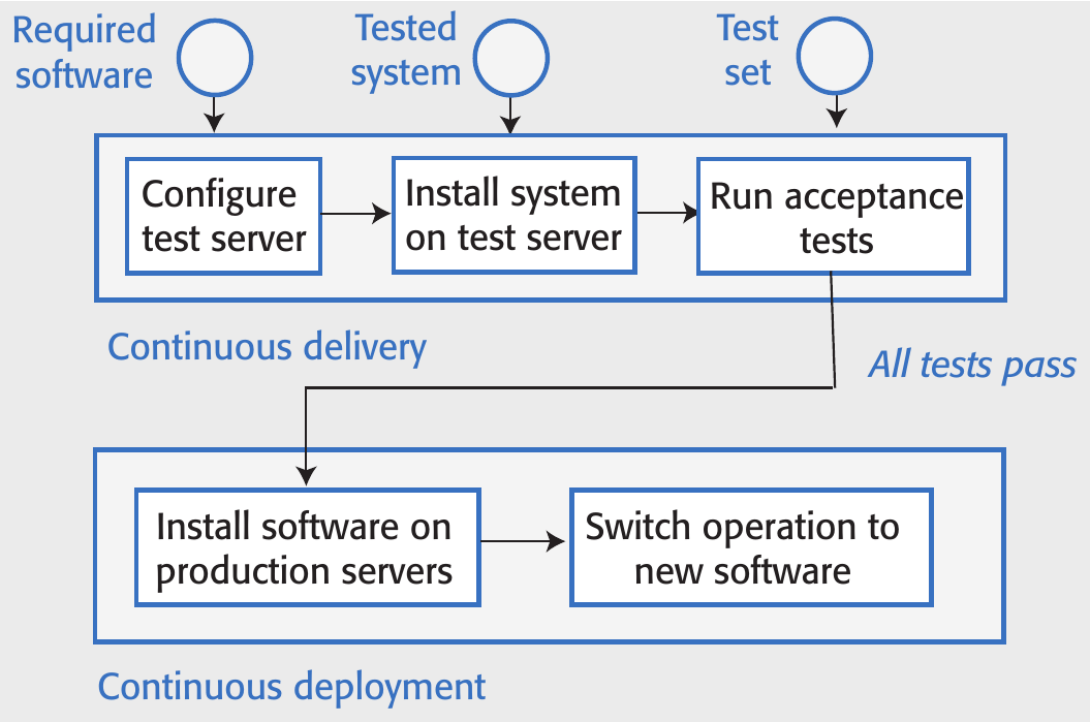


Figure 8.5: Continous delivery and deployment process

8.0.4 Infrastructure as Code (IaC)

Manually maintaining a computing infrastructure with tens or hundreds of servers is expensive and error-prone because there are many different physical/virtual servers that have different configurations and run different software packages and it's difficult to manually keep track of software installed on each server. Here IaC pop-up: it allows to automate the process of updating software on servers, by using a machine readable model of the infrastructure.Configuration Management (CM) tools, like Puppet and Chef and Ansible, can automatically install software and services on servers according to the infrastructure definition. When changes have to be made, infrastructure model is updated and CM tool makes the changes to all servers.
Some **advantages** of IaC are:

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

8.0.5 DevOps measurement

You should try to continuously improve your DevOps process to achieve faster deployment of better quality software: this implies to measure and analyse product and process data. We distingiuish in different types of measures:

- Process measurements: to collect & analyze data on development/testing/deployment processes
- Service measurements: to collect & analyze data on software's performance/reliability/acceptability to customers

- Usage measurements: to collect & analyze data on how customers use your product (help you identify issues and problems with the software itself)
- Business success measurements: to collect & analyze data on how your product contributes to overall business success (hard to isolate contribution of DevOps to business success - that may be due e.g. to DevOps introduction or to better managers)

Measuring software and its development is a **complex process** because need to identify suitable metrics that give you useful insights, need to find reliable ways of collecting and analyzing metrics data and some measures (e.g. customer satisfaction) must be inferred from other metrics (e.g. number of returning customers).
Also **software measurement** should be automated as far as possible by instrument your software to collect data about itself and use monitoring system to collect data about software's performance and availability.
Process measurement is not simple becayse different people record information differently (e.g. to record lead time for implementing a change). Is the "lead time" the overall elapsed time or the time spent by developers? How to take into account higher priority changes whose implementation slowed down other changes? The overall system to capture metrics is shown in 8.6
In 8.7 an example of measuring the achievement of the goal of shipping cloud-based code frequently without causing customer outages. The collected data

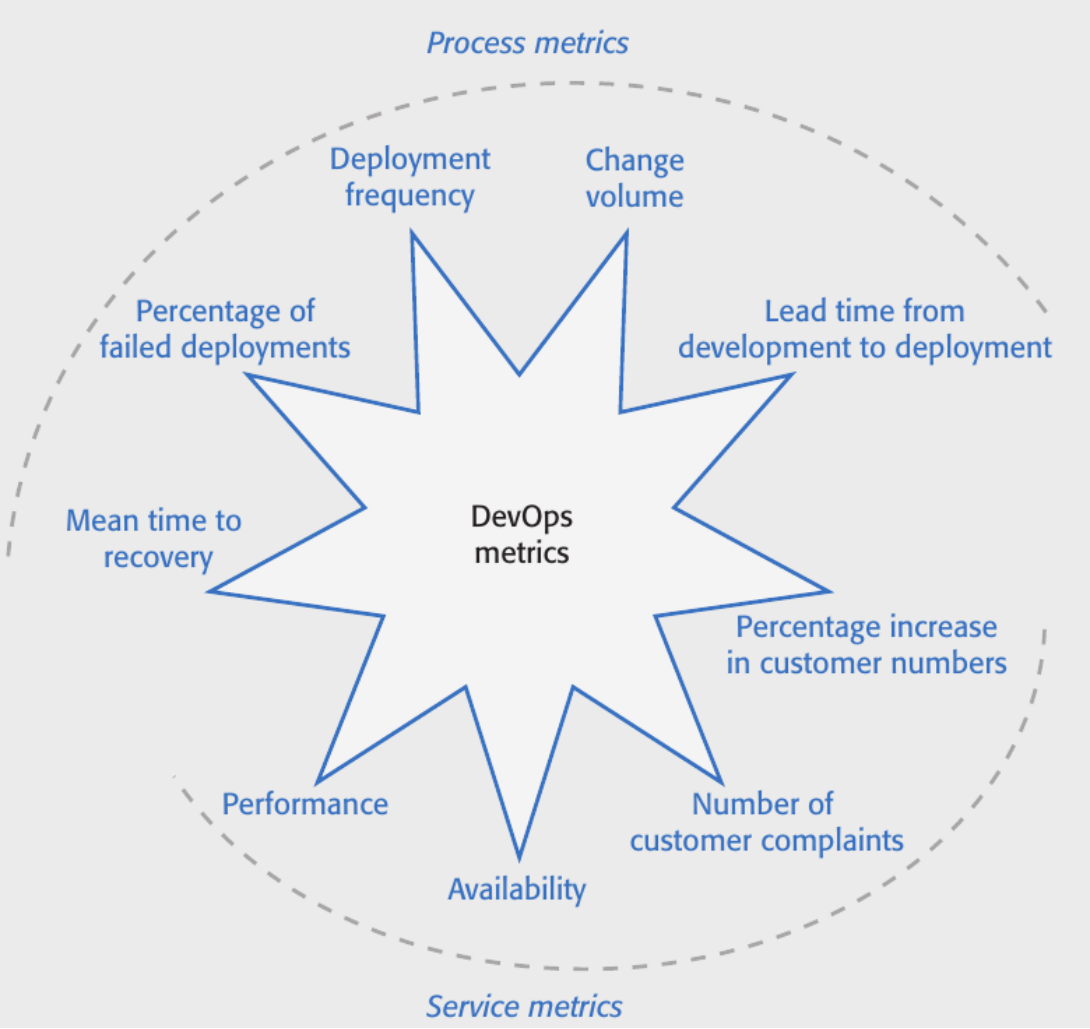


Figure 8.6: Metrics used in DevOps scorecard

To collect data you may use continuous integration tools like Jenkins to collect data on deployments, successful tests, etc. Also you may use monitoring services featured by cloud providers like Amazon's Cloudwatch to collect data on availability and performance or you may collect customer-supplied data from issue management systems. It's also recommended to add instrumentation to your product to

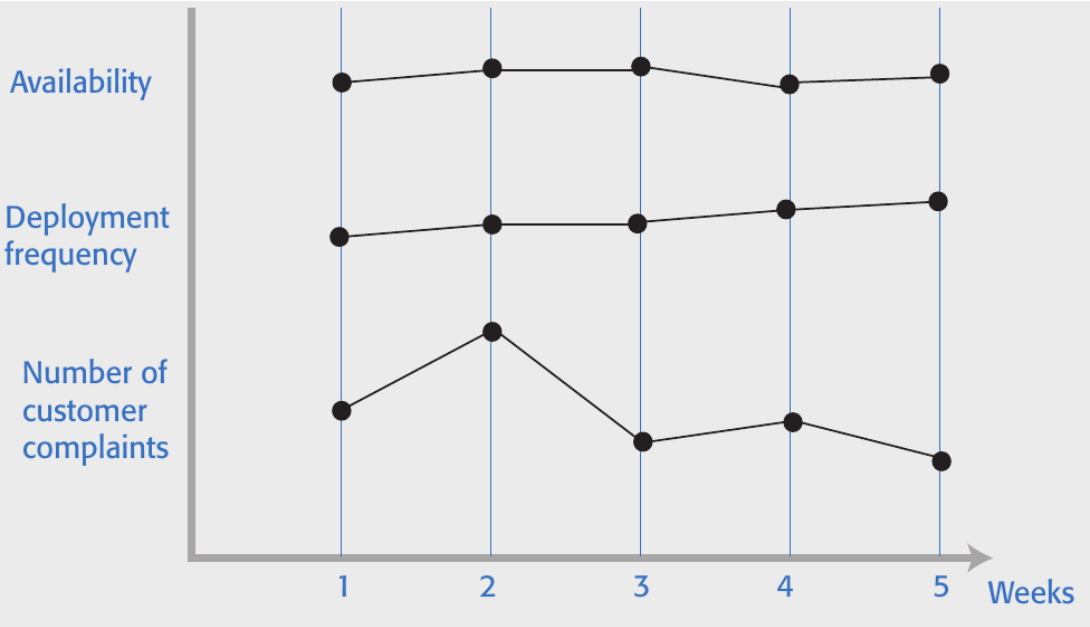


Figure 8.7: Example of metric trends

gather data on its performance and how it is used by customers by using log files, where log entries are timestamped events reflecting customer actions and/or software responses. It's better to log as many events as possible: use available log analysis tools to extract useful information about on how your software is used.

Chapter 9

Business Process Modelling

Business process management is the systematic method of examining your organization's existing business processes and implementing improvements to make your workflow more effective and more efficient. A **business process** is a set of business activities that represent the required steps to achieve a business objective. So a **business process model (BPM)** consists of a set of activity models and execution constraints among them while **business process instance (BPI)** represents a concrete case in the operational business of a company, consisting of activity instances. Here an example:
"When we receive a new order, an invoice should be sent to the customer.
The order should be archived only after receiving the payment.
The requested products must be shipped to the customers."
So we can summarize the activities as *Receive Order, Send Invoce, Archive Order, Receive Payment, Ship Products*. In figure 9.1 are pictures two example of business process model.

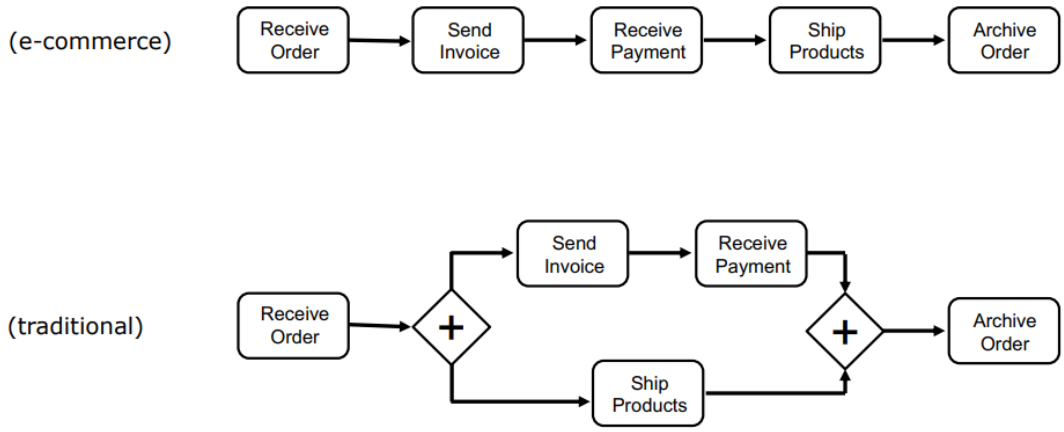


Figure 9.1: E-commerce vs Traditional BPM

9.1 Business Process Model and Notation - BPMN

BPMN is constrained to support only the concepts of modeling applicable to business processes. Other types of modeling done by organizations for non-process purposes are out of scope for BPMN. Examples of modeling excluded from BPMN are:

- Organizational structures
- Functional breakdowns
- Data models

Let's introduce the notation used in the diagrams.
BPMN models are expressed by simple diagrams constructed from a limited set of graphical elements. For both business users and developers, they simplify understanding of business activities' flow and process. BPMN's four basic element categories are:

- **Flow objects**: Events, activities, gateways
- **Connecting objects**: Sequence flow, message flow, association
- **Swim lanes**: Pool, lane
- **Artifacts**: Data object, group, annotation

The most basic are pictures in 9.2. Note that pool defines a group of participants or external entity, lane defines participant role whitin the process. In pictures 9.3, 9.4, 9.5 we show two complex example with

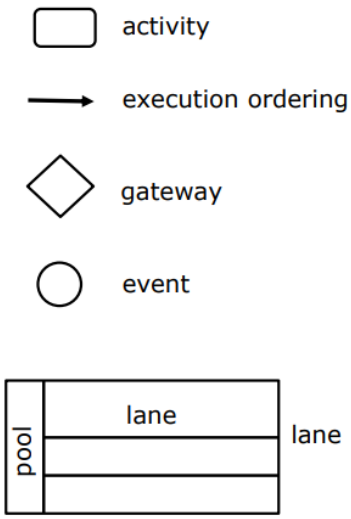


Figure 9.2: BPMN notation

different types of gateway and events. More information about different types of object can be find at BPMN - Wikipedia.

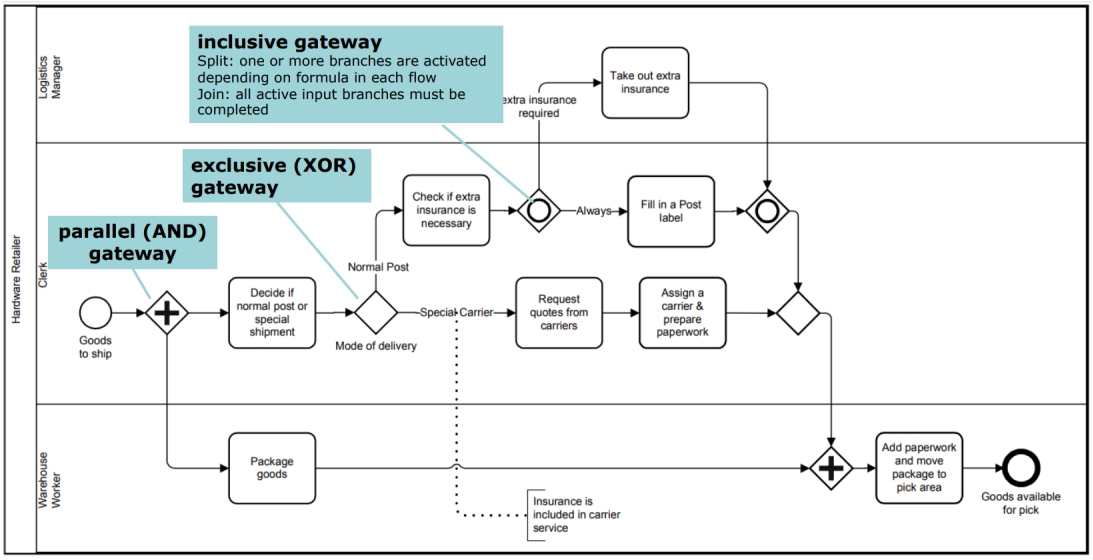


Figure 9.3

9.1.1 Camunda

Camunda is a framework supporting BPMN for workflow and process automation. It provides a RESTful API which allows you to use your language of choice as shown in 9.6.

Workflows are defined in BPMN which can be graphically modeled using the Camunda Modeler. Camunda has 2 *"usage pattern"* **A** and **B**:

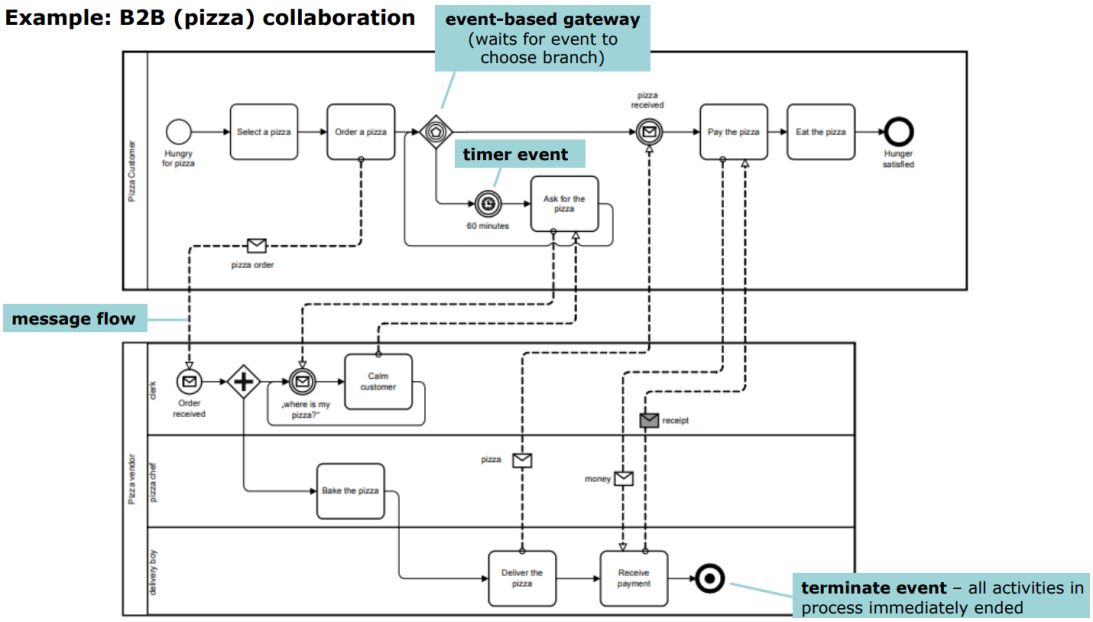


Figure 9.4

- **A - Endpoint-based integration**:After defining a BPMN process, Camunda can directly call services via *built-in connectors*.It supports both RESTful and SOAP services in this way.The main drawback is that it only alows scaling on process instance, NOT on microservices as pictured in 9.7.
- **B: Queue-based integration**: A more interesting pattern is known as **External Task**: Units of work (Tasks) are provided in a **Topic Queue** that can be polled by RESTful workers, possibly interacting with microservices. This pattern allows scaling of **process instances** so allows to scale workers and microservices as shown in 9.8. The *"worker philosophy"* is that they call microservices and acquire tasks, not viceversa.

The second pattern follows these steps, involving the relative components:

- Process Engine: Creation of an external task instance
- External Worker: Fetch and lock external tasks
- External Worker & Process Engine: Complete external task instance

9.2 Workflow nets

They're an extesion of **Petri nets**: it represent one of the best kwnown techniques for specifying business processes in a formal and abstract way because graphical representation eases communications between different stakeholders, process properties can be formally analysed and various supporting tools are available[1]
Petri nets consists of **places, transitions** and **direct arcs** connecting places and transitions. Transitions model **activities**, places and arcs model **execution constraints**. More on **Petri nets** can be found at Petri nets in one page. System dynamics represented by **tokens**, whose distribution over the places determines the state of modelled system. A **transition** *can fire* if there is a token in each of its input places (9.10). If a transition *fires*, one token is removed from each input place and one token is added to each output place (9.11). The idea behind the workflow nets are to enhance petri news with concepts and notations that ease the representation of business processes. Like petri nets, workflow nets focus on the control flow behaviour of a process:

- transitions represent activities
- places represent conditions

[1] *W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers, 8(1):21-66, 1998.*

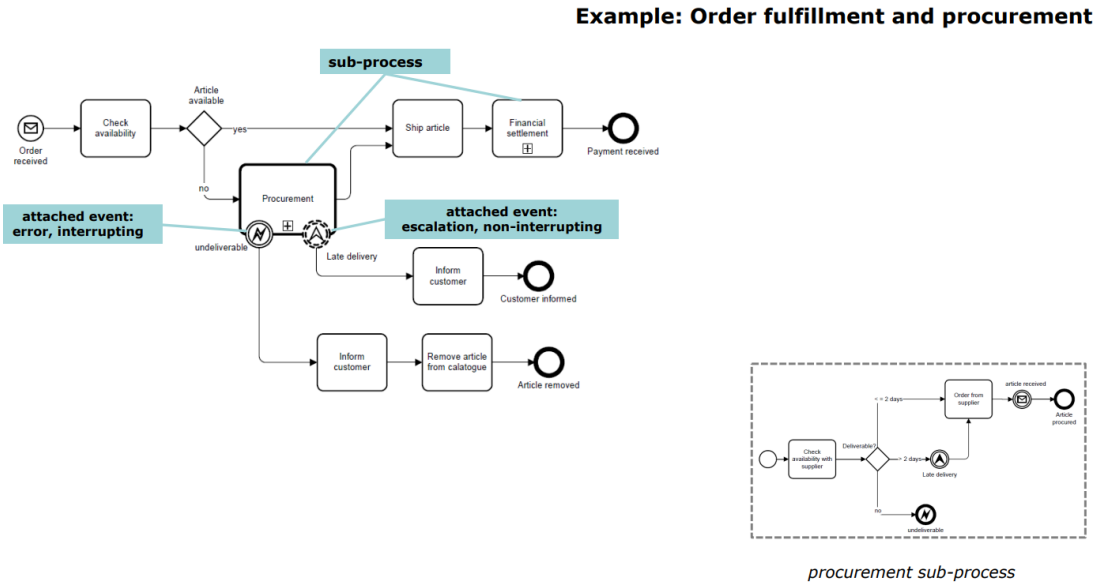


Figure 9.5

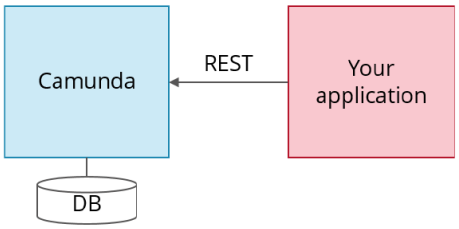


Figure 9.6: Camunda workflow

- tokens represent process instances

A petri net is a **workflow net** if and only if:

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

In figure 9.12 are shown some **Composition pattern**. In picture 9.13 are shown equivalent "sugared" representation of **AND-split** and **AND-join** transitions. A workflow net is **sound** if and only if:

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

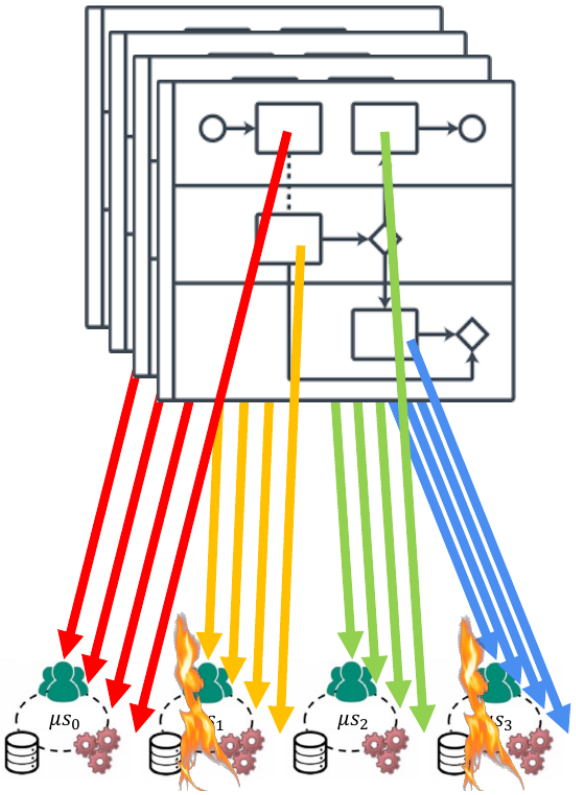


Figure 9.7: Camunda pattern A

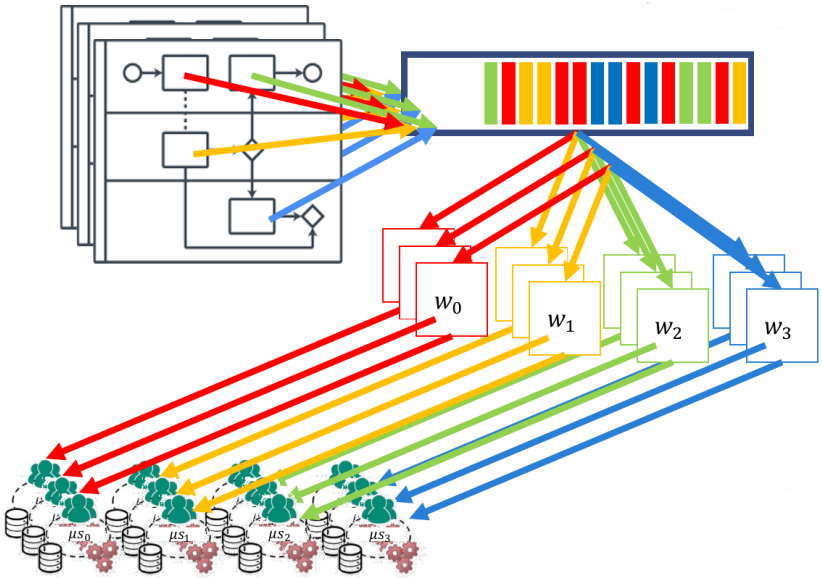


Figure 9.8: Camunda pattern B

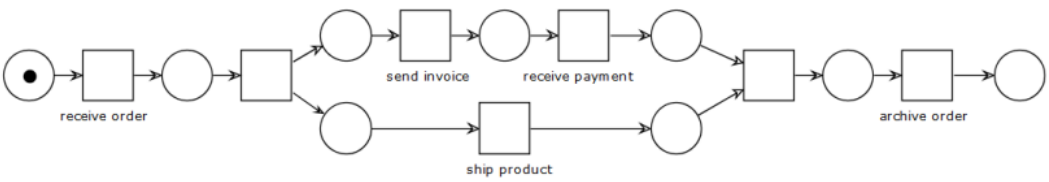
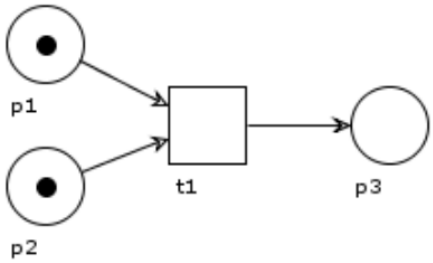
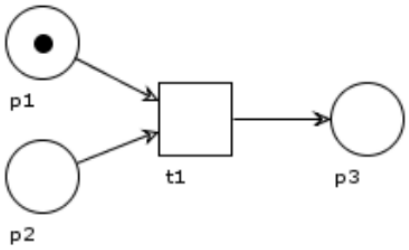


Figure 9.9



t1 can fire



t1 cannot fire

Figure 9.10

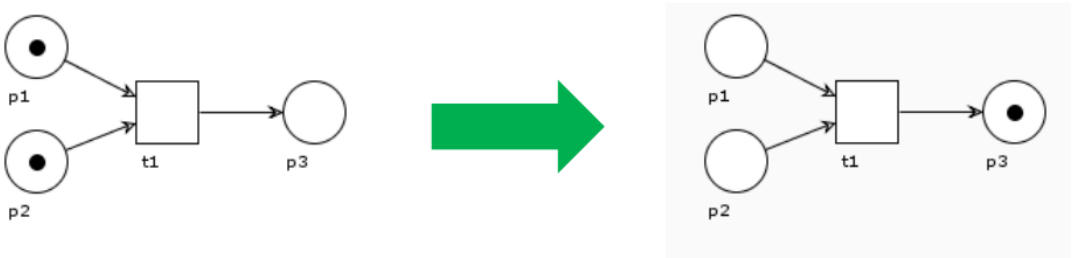


Figure 9.11

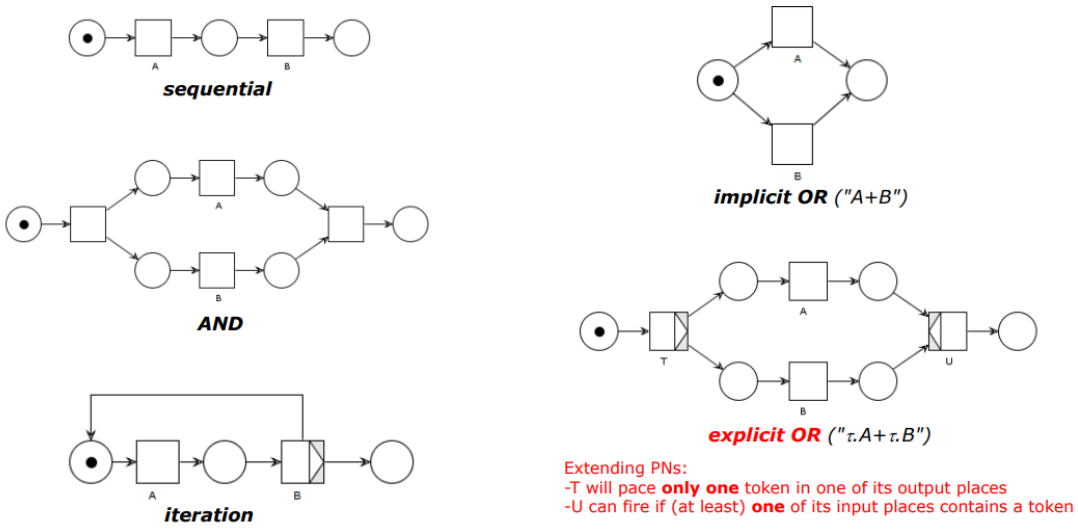


Figure 9.12: Composition pattern

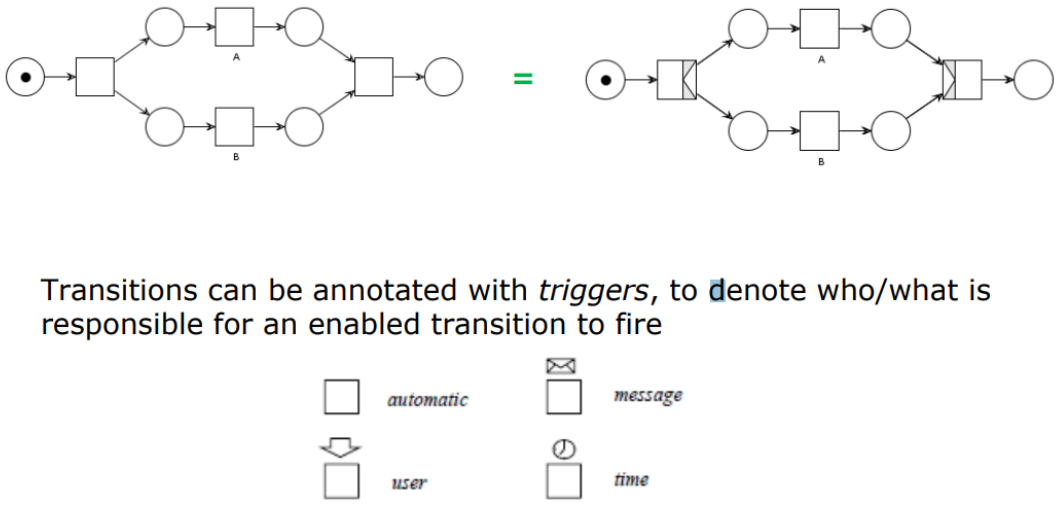


Figure 9.13: Equivalent "sugared" representation of **AND-split** and **AND-join** transitions

Chapter 10

Testing

Software testing is a process in which you execute your program using data that simulates user inputs. You observe its behaviour to see whether or not your program is doing what it is supposed to do: tests pass if the behaviour is what you expect, tests fail if the behaviour differs from that expected. If your program does what you expect, this shows that for the inputs used, the program behaves correctly. If these inputs are representative of a larger set of inputs, you can infer that the program will behave correctly for all members of this larger input set.
There are many types of testing:

- **Functional testing**: Test the functionality of the overall system. The goals of functional testing are to discover as many bugs as possible in the implementation of the system and to provide convincing evidence that the system is fit for its intended purpose.
- **User testing**: Test that the software product is useful to and usable by end-users. You need to show that the features of the system help users do what they want to do with the software. You should also show that users understand how to access the software's features and can use these features effectively.
- **Performance and load testing**: Test that the software works quickly and can handle the expected load placed on the system by its users. You need to show that the response and processing time of your system is acceptable to end-users. You also need to demonstrate that your system can handle different loads and scales gracefully as the load on the software increases.
- **Security testing**: Test that the software maintains its integrity and can protect user information from theft and damage.

Remember that **software testing** is different from **software verification**.

10.0.1 Functional testing

Functional testing involves developing a large set of program tests so that, ideally, all of a program's code is executed at least once. Functional testing is a staged activity in which you initially test individual units of code. You integrate code units with other units to create larger units then do more testing. The **functional testing processes** are the following:

- Unit testing: The aim of unit testing is to test program units in isolation. Tests should be designed to execute all of the code in a unit at least once. Individual code units are tested by the programmer as they are developed.
- Feature testing: Code units are integrated to create features. Feature tests should test all aspects of a feature. All of the programmers who contribute code units to a feature should be involved in its testing.
- System testing: Code units are integrated to create a working (perhaps incomplete) version of a system. The aim of system testing is to check that there are no unexpected interactions between the features in the system. System testing may also involve checking the responsiveness, reliability and security of the system. In large companies, a dedicated testing team may be responsible for system testing. In small companies, this is impractical, so product developers are also involved in system testing.

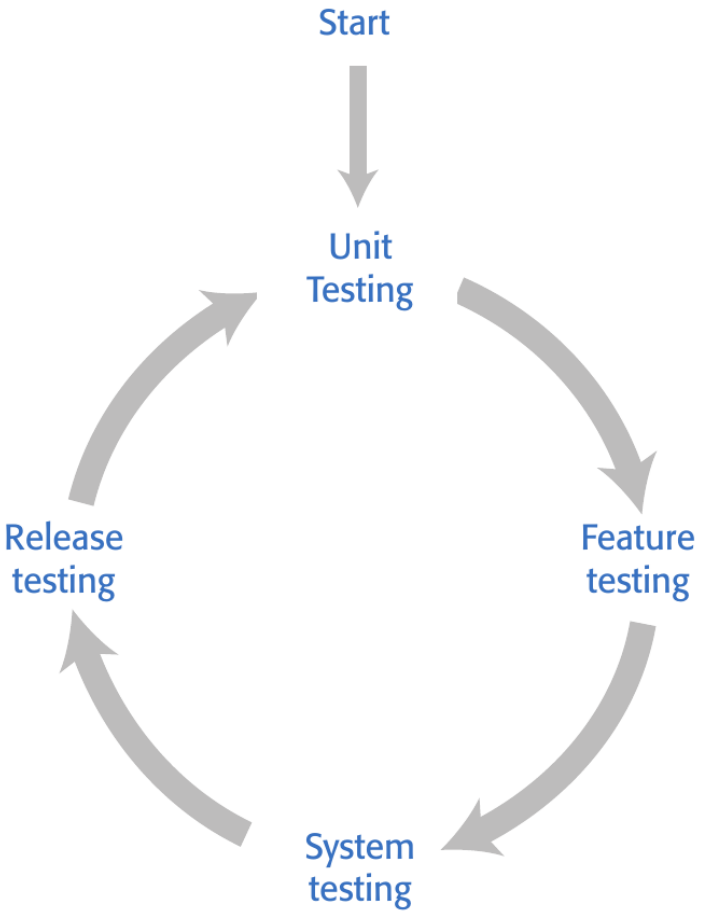


Figure 10.1: Functional testing cycle

- Release testing: The system is packaged for release to customers and the release is tested to check that it operates as expected. The software may be released as a cloud service or as a download to be installed on a customer's computer or mobile device. If DevOps is used, then the development team are responsible for release testing otherwise a separate team has that responsibility.

Regarding the **unit testing**, a code unit is anything that has a clearly defined responsibility. It is usually a function or class method but could be a module that includes a small number of other functions. Unit testing is based on a simple general principle:

If a program unit behaves as expected for a set of inputs that have some shared characteristics, it will behave in the same way for a larger set whose members share these characteristics.

So, for example, if your program behaves correctly on the input set (1, 5, 17, 45, 99), you may conclude that it will also process all other integers in the range 1 to 99 correctly.
To test a program efficiently, you should identify sets of inputs (**equivalence partitions**) that will be treated in the same way in your code. The equivalence partitions that you identify should not just include those containing inputs that produce the correct values. You should also identify *incorrectness partitions* where the inputs are deliberately incorrect. An example in 10.2.

There are some guidelines with a brief explanation for unit testing:

- 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.

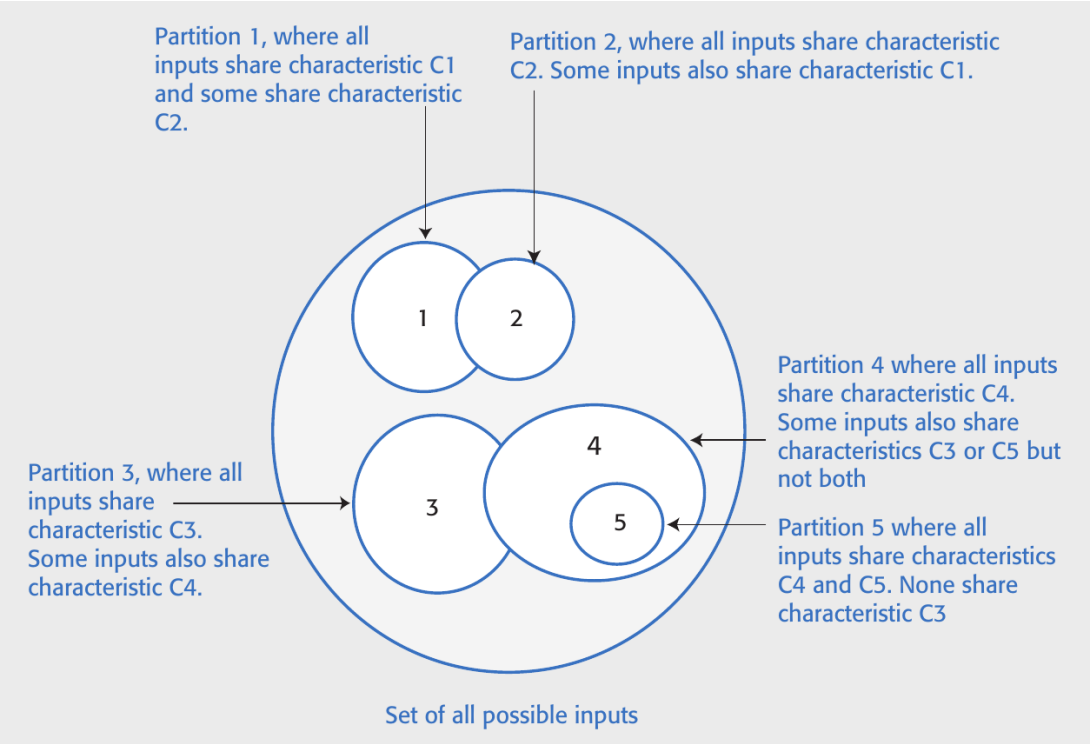


Figure 10.2: Example of equivalence partitions

- Overflow and underflow: If your program does numeric calculations, choose test inputs that 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 transformation, 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.

Feature testing

Features have to be tested to show that the functionality is implemented as expected and that the functionality meets the real needs of users. Normally, a feature that does several things is implemented by multiple, interacting, program units. There are two types of feature test:

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

As an example, refer the scenario pictures in 10.3.

System testing

System testing involves testing the system as a whole, rather than the individual system features. System testing should focus on four things:

1. Testing to discover if there are unexpected and unwanted interactions between the features in a system.
2. Testing to discover if the system features work together effectively to support what users really want to do with the system.

Story title	User story
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.
Information sharing	As a user, I want to know what information you will share with other companies. I want to be able to cancel my registration if I don't want to share this information.
Email choice	As a user, I want to be able to choose the types of email that I'll get from you when I register for an account.

From scenarios/user stories to feature tests

Test	Description
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.
Incorrect credentials	Test that the error message and retry screen are displayed if the user inputs incorrect Google credentials.
Shared information	Test that the information shared with Google is displayed, along with a cancel or confirm option. Test that the registration is canceled if the cancel option is chosen.
Email opt-in	Test that the user is offered a menu of options for email information and can choose multiple items to opt in to emails. Test that the user is not registered for any emails if no options are selected.

Figure 10.3: Feature testing example

3. Testing the system to make sure it operates in the expected way in the different environments where it will be used.

4. Testing the responsiveness, throughput, security and other quality attributes of the system.

The best way to systematically test a system is to start with a set of **scenarios** that describe possible uses of the system and then work through these scenarios each time a new version of the system is created. Using the scenario, you identify a set of end-to-end pathways that users might follow when using the system: an end-to-end pathway is a sequence of actions from starting to use the system for the task, through to completion of the task.

Release testing

Release testing is a type of system testing where a system that's intended for release to customers is tested. The fundamental differences between release testing and system testing are:

- Release testing tests the system in its real operational environment rather than in a test environment. Problems commonly arise with real user data, which is sometimes more complex and less reliable than test data.

- The aim of release testing is to decide if the system is good enough to release, not to detect bugs in the system. Therefore, some tests that 'fail' may be ignored if these have minimal consequences for most users.

Preparing a system for release involves packaging that system for deployment (e.g. in a container if it is a cloud service) and installing software and libraries that are used by your product. You must define configuration parameters such as the name of a root directory, the database size limit per user and so on.

10.0.2 Test automation

Automated testing is based on the idea that tests should be executable: an executable test includes the input data to the unit that is being tested,the expected result and a check that the unit returns the expected result. (See 10.4). You run the test and the test passes if the unit returns the expected result. Normally, you should develop hundreds or thousands of executable tests for a software product. A good practice is to **structure automated test in 3 parts**:

- **Arrange**: Set up the system to run the test (define test parameters, mock objects if needed)

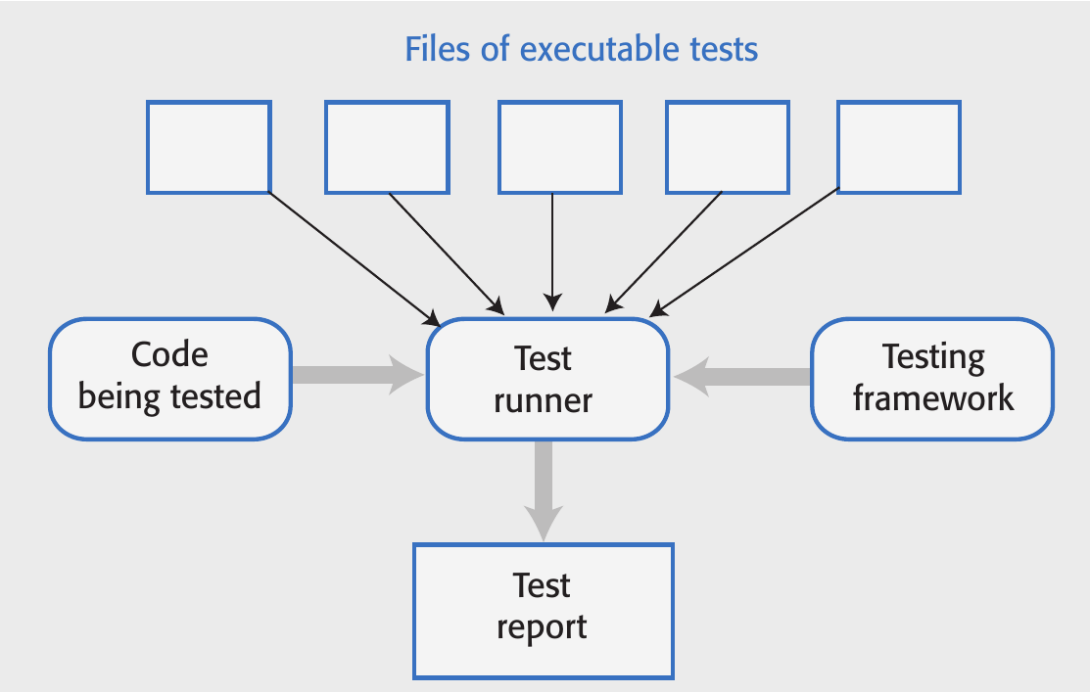


Figure 10.4: Automated testing interaction

- **Action**: Call the unit that is being tested with the test parameters
- **Assert**: Assert what should hold if test executed successfully.

Of courde test code can include bugs so it's a good practice to reduce the chances of test errors by making tests as simple as possible and review all test along with the code that they test.
Unit tests are the easiest to automate: good unit tests reduce (do not eliminate) need of feature tests thus GUI-based testing expensive to automate so API-based testing preferrable. Generally requires multiple assertions to check that feature executed as expected. Differently, **system testing** involves testing system as a surrogate user: perform sequences of user action, execute manual system testing. There are testng tools that record series of actions and automatically reply them to avoid human error in manual testing.

10.0.3 TDD - Test Driven Development

Test-driven development (TDD) is an approach to program development that is based around the general idea that you should write an executable test or tests for code that you are writing before you write the code, as shown in 10.5. It was introduced by early users of the Extreme Programming agile method, but it can be used with any incremental development approach. Test-driven development works best for the development of individual program units and it is more difficult to apply to system testing.

Here are some pros:

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

And some cons:

- Difficult to apply TDD to system testing
- TDD discourages radical program changea
- TDD leads you to focus on the tests rather than on the problem you are trying to solve

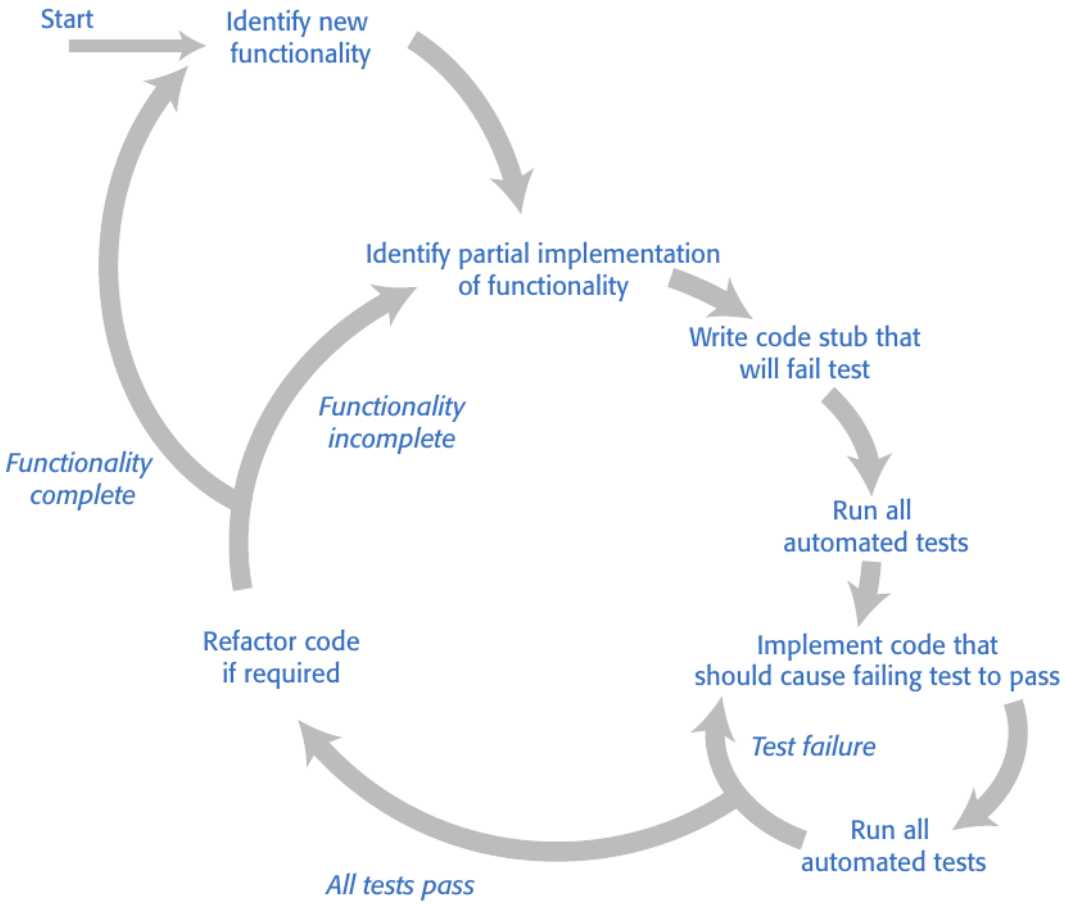


Figure 10.5: Test-driven development phases

- TDD leads you to think too much about implementation details rather than on overall program structure
- Hard to write "bad data" tests

10.0.4 Security Testing

Security testing aims to find vulnerabilities that may be exploited by an attacker and to provide convincing evidence that the system is sufficiently secure. Finding vulnerabilities it's harder than finding bugs because you nust test for something that software should not do and generally normal functional test may not reveal any vulnerabilities. Also the software stack on which your product depends may contain vulnerabilities. Comprehensive security testing requires specialist knowledge of software vulnerabilities and approaches to testing that can find these vulnerabilities.
A **risk-based approach** to security testing involves identifying common risks and developing tests to demonstrate that the system protects itself from these risks. You may also use automated tools that scan your system to check for known vulnerabilities, such as unused HTTP ports being left open. Based on the risks that have been identified, you then design tests and checks to see if the system is vulnerable. It may be possible to construct automated tests for some of these checks, but others inevitably involve manual checking of the system's behaviour and its files. Here some examples of security risks:

- Unauthorized attacker gains access to a system using authorized credentials
- Authorized individual accesses resources that are forbidden to that person
- Authentication system fails to detect unauthorized attacker
- Attacker gains access to database using SQL poisoning attack
- Improper management of HTTP sessions

- HTTP session cookies are revealed to an attacker
- Confidential data are unencrypted
- Encryption keys are leaked to potential attackers

10.0.5 Code reviews

Testing have some limitation like:

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

Code reviews involve one or more people examining the code to check for errors and anomalies and discussing issues with the developer. If problems are identified, it is the developer's responsibility to change the code to fix the problems. Code reviews complement testing: they are effective in finding bugs that arise through misunderstandings and bugs that may only arise when unusual sequences of code are executed. In 10.6 is represented the code reviews process setting. Generally one review session should

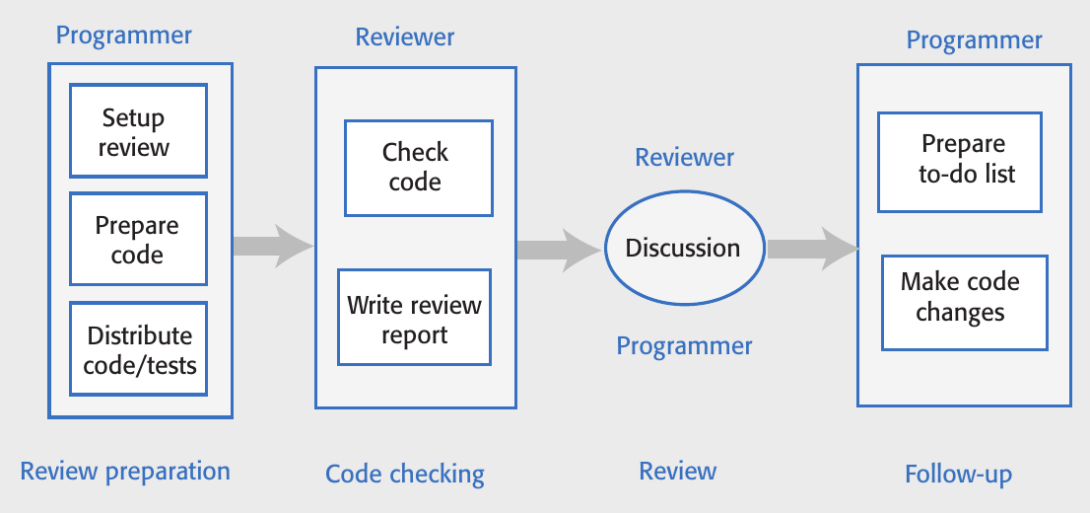


Figure 10.6: Code review process

focus on 200-400 lines of code and should include a checklist with main check point like:

1. Are meaningful variable and function names used? (General): Meaningful names make a program easier to read and understand.
2. Have all data errors been considered and tests written for them? (General); It is easy to write tests for the most common cases but it is equally important to check that the program won't fail when presented with incorrect data.
3. Are all exceptions explicitly handled? (General): Unhandled exceptions may cause a system to crash.
4. Are default function parameters used? (Python): Python allows default values to be set for function parameters when the function is defined. This often leads to errors when programmers forget about or misuse them.
5. Are types used consistently? (Python): Python does not have compile-time type checking so it it is possible to assign values of different types to the same variable. This is best avoided but, if used, it should be justified.

6. Is the indentation level correct? (Python): Python uses indentation rather than explicit brackets after conditional statements to indicate the code to be executed if the condition is true or false. If the code is not properly indented in nested conditionals this may mean that incorrect code is executed.

As a code review example, **Spotify** identify as **Chapters** the team members working within a special area (e.g., front office developers, back office developers, database admins, testers) and a **Guild** is a community of members across the organization with shared interests (e.g., web technology, test automation), who want to share knowledge, tools code and practices. Usually Chapters (sometimes guilds) do code reviews for squads: are required two *+1* to merge.