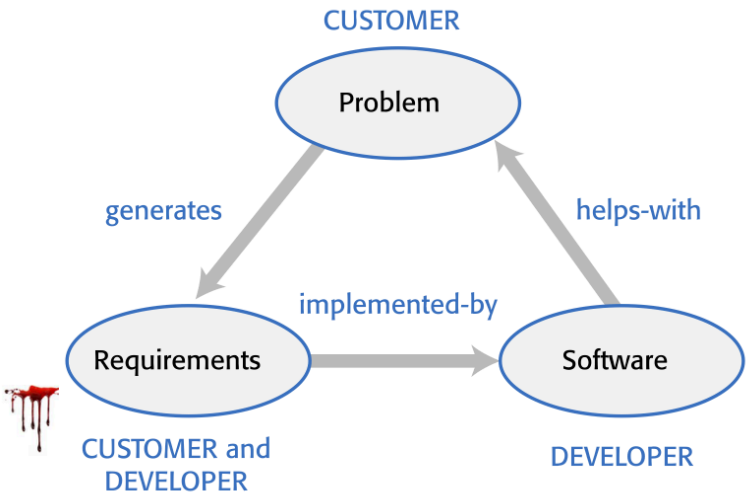
Advanced Software Engineering

✅ 1. Software Products

In **Project-Based Software Engineering**, there is a loop that often weakens software from the early stages of development. This happens because requirements are changeable and often get updated over time, along with the features added to the software.

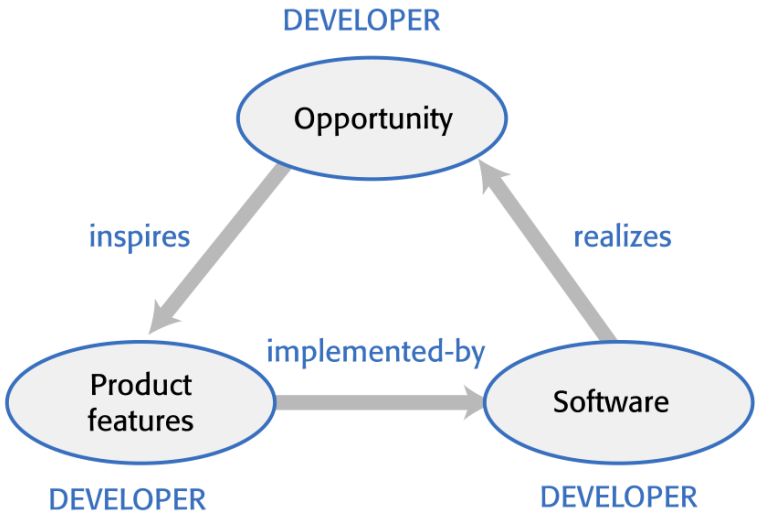
Customer decide system functionalities

Business changes → Requirements change → Software must change



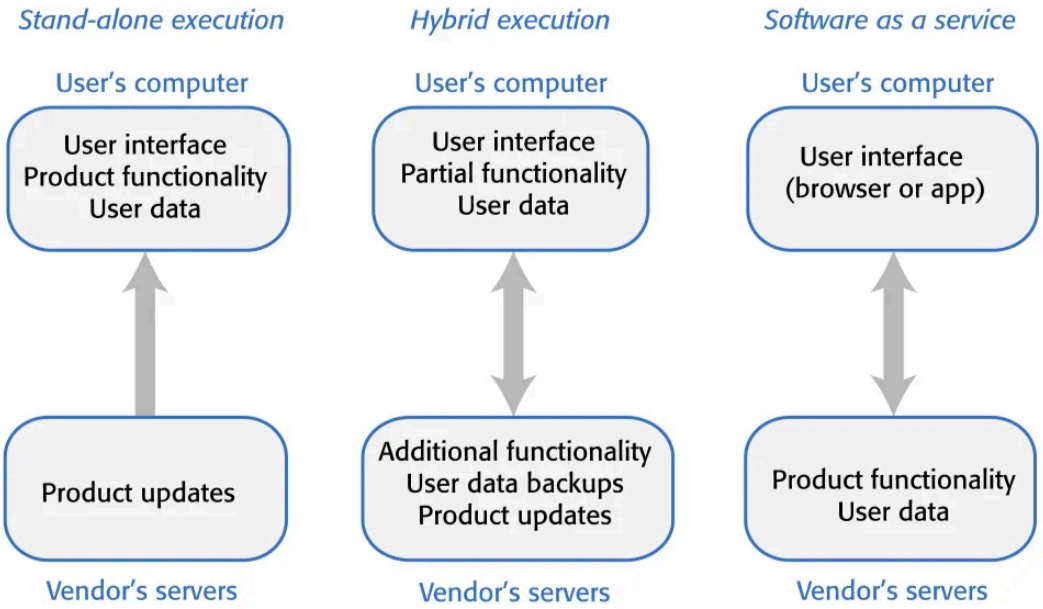
Product-Based Software Engineering:

Developer decides product features and evolution



- Getting product to customers quickly is critical.
- For this we need rapid software development techniques (Agile methods)

Software Execution Models:



The Product Vision - Starting Point. Three Fundamental questions:

1. **WHO** are the targeted customers ?
2. **WHAT** is the product to be developed ?

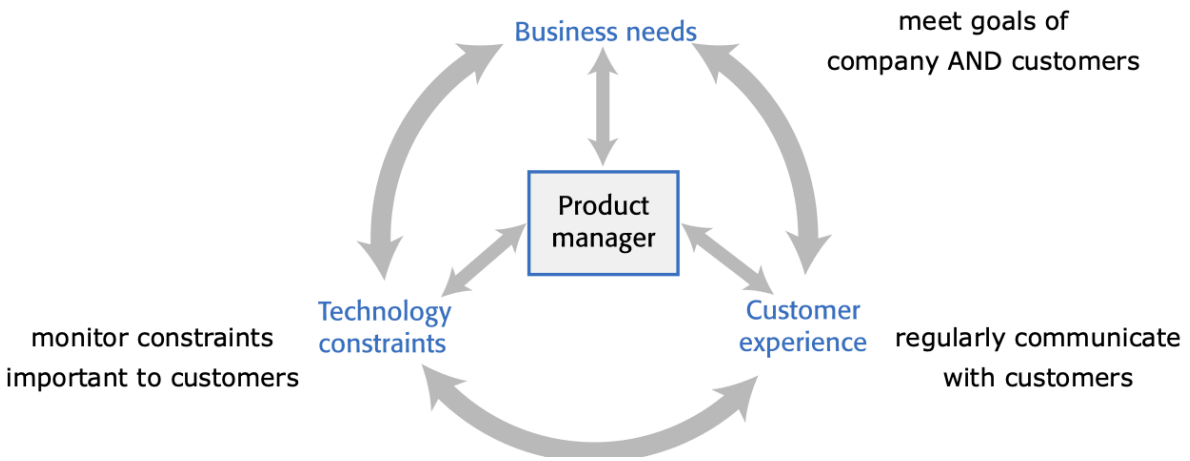
3. **WHY** should customers buy the product ?

- FOR (target customer)
- WHO (statement of the need or opportunity)
- THE (product name) is a (product category)
- THAT (key benefit, compelling reason to buy)
- UNLIKE (primary competitive alternative)
- OUR PRODUCT (statement of primary differentiation)

Product vision:

- **Domain experience** - Product developers may work in a specific area (for example, marketing or sales) and understand what kind of software support they need. They might feel frustrated with the weaknesses of the software they currently use and see chances to create a better system.
- **Product experience** - People who use existing software (like word processors) might think of easier and better ways to offer the same features and suggest creating a new system that does this. New products can also use modern technologies, such as voice or speech interfaces.
- **Customer experience** - Software developers may talk in detail with potential customers to understand the problems they face and the limits, like interoperability, that reduce their ability to buy new software. They also try to learn what important qualities the customers want in the software.
- **Prototyping and experimenting** - Developers may come up with an idea for new software but need to understand it better before turning it into a full product. They might build a small prototype as a test and try out different ideas and versions using that prototype as a base.

Software Product Management - PM must ensure that development team implements features that deliver real value to customer.



Technical interactions of PMs

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)

In Product Prototyping - Its also critically important to demonstrate software to potential customers and funders, they can help revising design. Aim at having first reduced prototype quickly (around 6 weeks)

✅ 2. Agile Software Engineering

Plan-driven development (twentieth century) involves significant overhead in planning, designing and documenting a system.

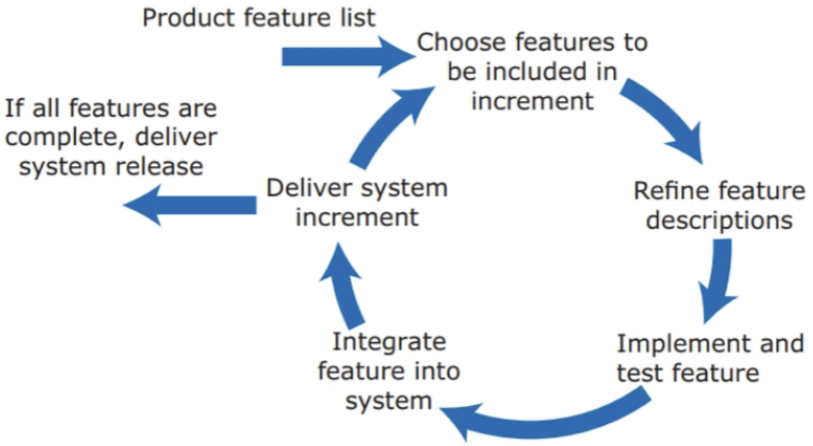
Agile is a set of principles and methods used in software development. Unlike project-based software engineering, in Agile the client explains what they want

in terms of features, not technical details.

Agile SE focus on the software to:

- deliver functionalities quickly
- respond quickly to changes
- minimise development overheads

Agile uses an ***incremental development and delivery model***, where the software is built and improved step by step. Software product is thought as a set of features. Incremental Development:



12 AGILE PRINCIPLES:

1. Our main goal is to make the customer happy by **delivering useful software early and regularly**.
2. **Accept changes in requirements**, even late in the project. Agile uses change to help the customer stay competitive.
3. **Deliver working software often**, every few weeks or months, preferably in shorter time periods.
4. Business people and developers should **work together** every day during the project.
5. Build projects around motivated people. Give them the right environment and support, and trust them to do their work well.

6. The best and fastest way to share information within a development team is through face-to-face communication.
7. **Working software is the main measure of progress**.
8. Agile supports sustainable development. Sponsors, developers, and users should be able to keep a steady work pace indefinitely.
9. Constant focus on technical quality and good design makes a team more flexible and effective.
10. **Simplicity**—doing only what is necessary and avoiding extra work—is very important.
11. The best architectures, requirements, and designs come from **self-organizing teams**.
12. The team should regularly look back on its work to find ways to improve and adjust how it operates.

Extreme Programming was proposed as part of the agile methodology. Extreme Programming practices:

Practice	Description
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 and 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.
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.

Scrum is a framework used in Agile project management that organizes how the team works. It includes specific roles for people who act as a link between

the development team and the organization.

Because requirements often change, long-term plans are not reliable. For this reason, software engineering focuses on short-term planning.

Scrum is based on **empiricism and lean thinking**. It means that knowledge comes from experience, and decisions should be made based on what is observed.

Other key ideas in Scrum are transparency (everyone knows what is going on in the project), inspection (regularly checking the code and other work), and adaptation (adjusting to new features or changing requirements).

The **Scrum Team** small self-organizing team (4-8 people) includes:

1. **Product Owner:** who is responsible for identifying product features and attributes, reviews work done and helps to test the product
2. **Scrum Master:** a Scrum coach who helps the team correctly follow the Scrum process.
3. **Developers:** the people who actually write the code and build the software.

In Scrum, software is developed in **sprints** — short, fixed time periods with a clear goal to complete.

- **Product backlog:** a list of all tasks and features that team has not completed and need to be developed.
- **Timeboxed sprints:** Short period when a product increment is developed. Sprints have a fixed duration (2-4 weeks) that cannot be extended.
- **Self-organizing teams:** the team manages its own work and decides how to achieve the sprint goals.
- **Scrum:** Daily team meeting where progress is reviewed and work to be done that day is discussed and agreed.

Product Backlog

The product backlog is a key part of the Scrum method. It is a to-do list, and each item on it is called a Product Backlog Item (PBI). The list is ordered by priority, so the most important tasks that should be done first are placed at the top.

In a sprint, team implements as many sprint backlog items as possible in the given time period. Incomplete items are returned to product backlog.

Product Backlog: states of items

Ready for consideration: 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.

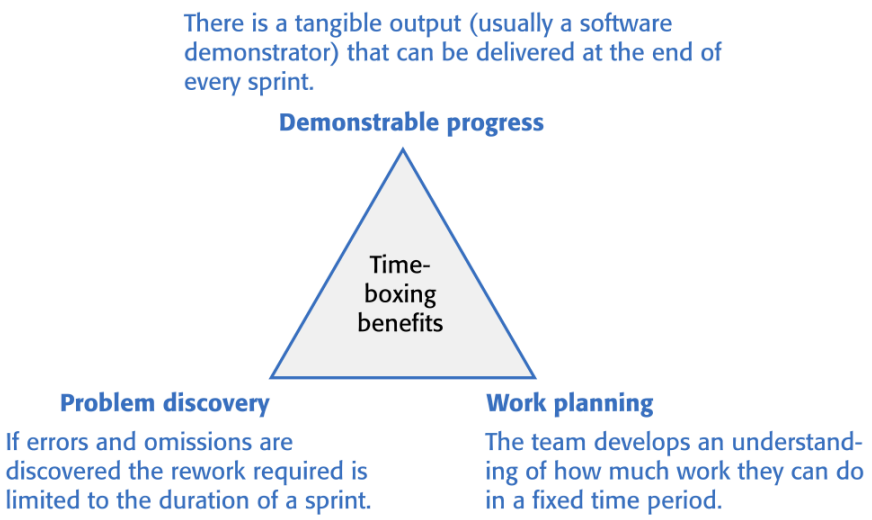
Product Backlog: activities:

- **Refinement:** Existing PBIs are reviewed and improved to make them more detailed. This process can also create new PBIs.
- **Estimation:** The team estimates how much work each PBI will take and records this information for each item.
- **Creation:** New items are added to the backlog. These can include new features suggested by the product owner, updates to existing features, technical improvements, or tasks like checking new development tools.
- **Prioritization:** The backlog is reordered to reflect new information or changes in the project's situation.

Time-boxed Sprints

Sprints have a fixed duration. Even if the goal is not fully completed by the end of a sprint, the work stops, and a new sprint begins. The unfinished tasks are moved to the next sprint. During a sprint, the team works on the items from the product
backlog.

Timeboxed sprints: benefits



Time-boxed sprints: activities

1. ***Sprint planning***
2. ***Sprint execution***
3. ***Sprint reviewing***

Scrum Meetings

During each sprint, the team holds short daily meetings (called daily scrums) to check progress and update the list of tasks that are still incomplete.

Agile Suggested Techniques. While Scrum does not prescribe the technical agile activities to be used, two practices should always be used in a sprint:

- **Test automation:** As much testing as possible should be automated. Developers should create a set of automatic tests that can be run at any time.
- **Continuous integration:** When a developer changes part of the software, the new code should be immediately combined with the rest of the system and tested to find any unexpected issues between components.

Sprint Reviews

At the end of every sprint, there is a review meeting with the whole team. The product owner decides whether the sprint goal has been reached. The review

should also include a discussion about how the team can improve their work process for the next sprint.

Self-organizing teams

Scrum teams have to tackle diverse tasks and so usually require people with different skills and different levels of experience. A team of 5-8 people is large enough to be diverse.

The developers of Scrum assumed that teams would be co-located. However, the use of daily scrums as a coordination mechanism is based on two assumptions that do not always hold:

- 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.
- Scrum assumes that all team members can attend a morning meeting to coordinate the work for the day. In reality, some team members may work flexible hours or may work on several projects at the same time.

Self-organizing teams: external interactions

- External interactions are interactions that team members have with people outside of the team.
- Only the ScrumMaster and Product Owner should be involved in external interactions. Developers should focus on development.

In all product development companies, there is a need for development teams to report on progress to company management. The developers of Scrum did not envisage that the ScrumMaster should also have project management responsibilities.

✅ 3. Features, Scenarios, Stories

The design of software products is driven by several factors:

- Inspiration
- Business or consumer needs that are not met by existing products
- Dissatisfaction with current products
- New technologies that make new types of products possible

The main focus is on features (parts of the software's functionality). To understand which features are needed, we first need to identify potential users through interviews, surveys, and informal user analysis or discussions.

Flow-chart

User representations — called **personas** — and natural language descriptions — such as **scenarios and stories** — help identify and define the product features:

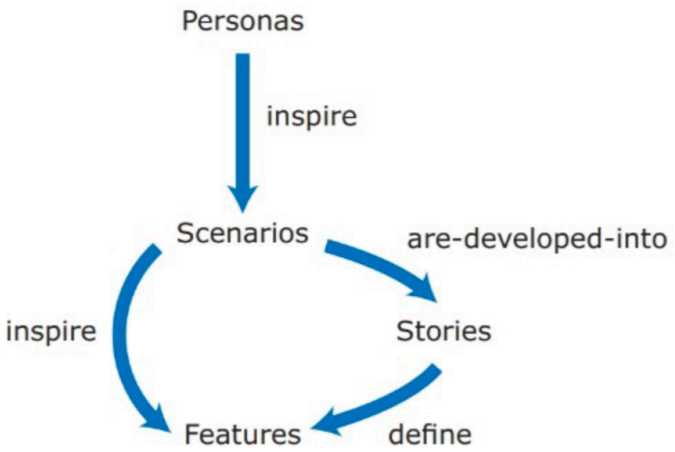
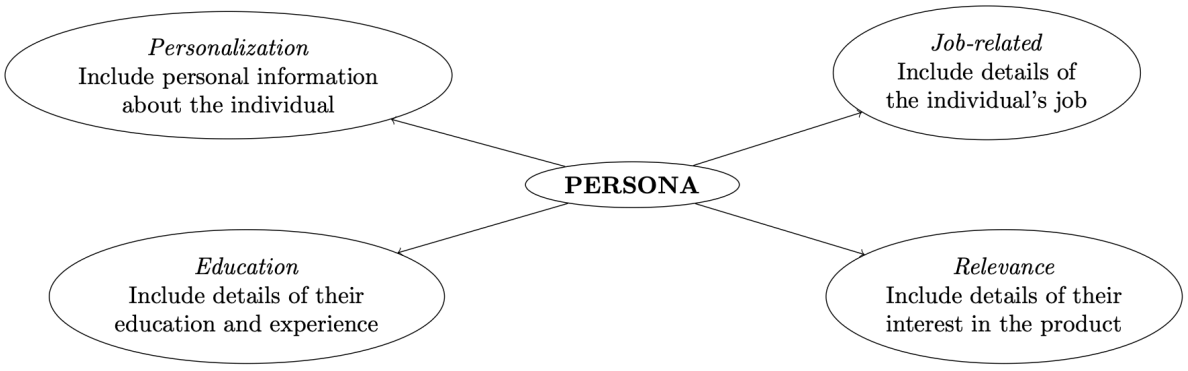


Figure 2.1: Features flowchart

Personas represent the main types of users who will use the product. Each persona should show the background, skills, and experience of potential users. Usually, only a few personas (no more than five) are enough to identify the main product features.

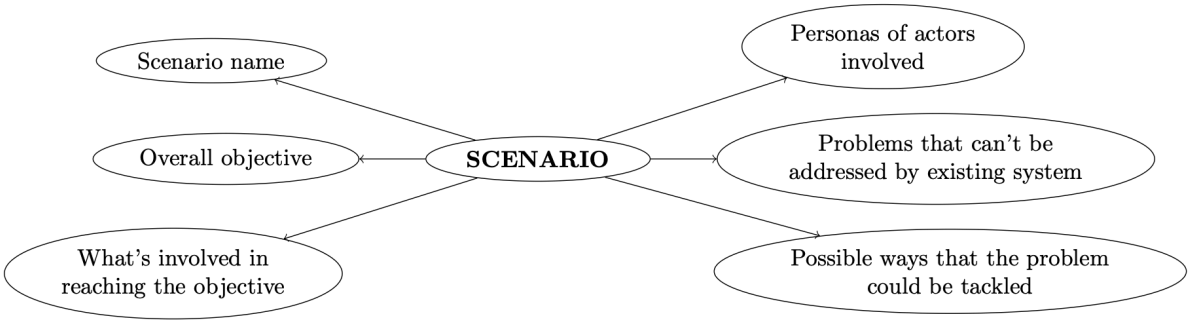


There are different opinions about whether personas should include photos or not.

Photos can sometimes give the wrong impression because “personas are not about how users look, but about what they do” (Steve Cable). According to Sara Wachter-Boettcher, “detailed personas made the team think that demographic details were the main reason behind user motivations,” which can be misleading.

Scenario is a short story written from the user’s point of view that explains a situation where the user uses the product’s features to achieve something they want to do.

Scenarios are not technical specifications. They do not include all details and may be incomplete.



Usually, it is enough to create 3–4 scenarios for each persona to cover their main tasks and responsibilities. Every team member should write some scenarios and discuss them with the rest of the team and, if possible, with real users.

User Stories

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

While scenarios are general stories about how a product is used, *user stories are more detailed and specific*. They help organize work into small tasks that

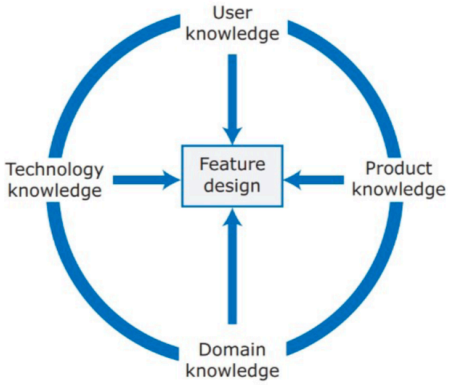
bring real value to the customer, allowing the software to be built step by step from the user's point of view. Longer stories can be divided into shorter ones and then prioritized.

Even though all parts of a scenario can be expressed as user stories, scenarios are often easier to read, give better understanding, and provide more context.

Knowledge sources for feature design

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

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



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

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

Feature identification

The goal is to create a list of features that define the product, keeping in mind these key properties:

- **Independence** → a feature should not rely on how other features are built and should not be affected by the order in which other features are used.
- **Coherence** → a feature should focus on one clear function. It should not do multiple unrelated things or cause unwanted side effects.
- **Relevance** → features should match the way users normally do their tasks. They should not include rare or unnecessary functions.

To identify features from scenarios and user stories, the development team should discuss them together and start prototyping to show the most important and new features.

Feature Creep

Feature creep happens when more and more features are added as new possible users are considered. To avoid this problem, four questions should be asked:

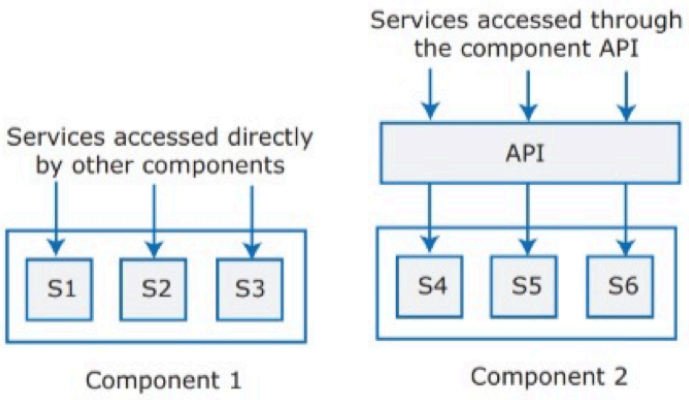
1. Does this feature add something truly new, or is it just another way to do something that already exists?
2. Can this feature be added by improving an existing one instead of creating a new one?
3. Is this feature important and useful for most users?
4. Does this feature provide general functionality, or is it too specific?

✅ 4. Software Architecture

Architecture is the basic structure of a software system, shown through its components, how they connect with each other and with the environment, and the main principles that guide its design and future development.

Component

A component is a part of the system that provides a specific set of features. It can be seen as a group of services that other components may use, either directly or through an API.



Architectural design issues

- **Non-functional product characteristics:** aspects like security, reliability, and availability are just as important as the system's functions for building a successful product.
- **Product lifetime:** if the product is expected to last for a long time, its architecture should be flexible and easy to change. For example, using microservices allows the system to scale more easily and extend its lifetime.
- **Software compatibility:** some systems may include older (legacy) modules, so ensuring compatibility is important. This can limit some architectural choices.
- **Number of users:** when software is released online, it is hard to predict how many users it will have. The number may change a lot, so the architecture must be able to scale up or down as needed.
- **Software reuse:** using existing components from other products or open-source software can save time and effort, but it can also restrict the possible architectural designs.

Non-functional quality attributes

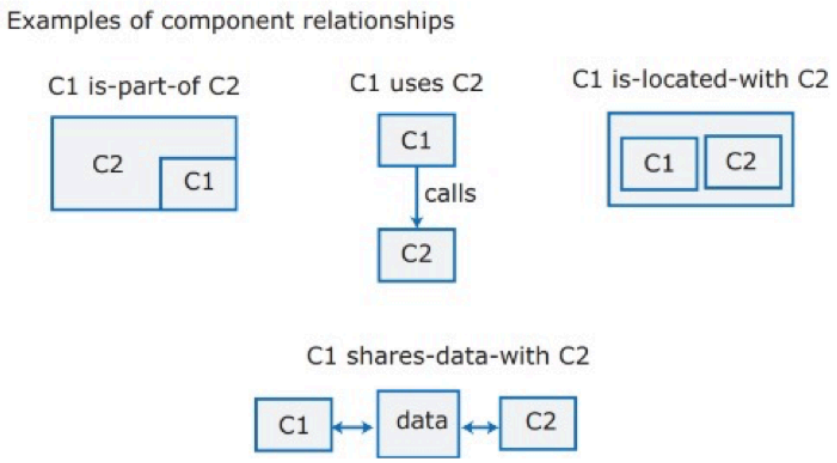
1. **Responsiveness** – Does the system give results in a reasonable amount of time?
2. **Reliability** – Do the features work as expected?
3. **Availability** – Can the system provide services whenever users need them?
4. **Security** – Does the system protect itself and user data from attacks or unauthorized access?
5. **Usability** – Can users easily and quickly access the features they need?
6. **Maintainability** – Can the system be updated or changed easily without high costs?
7. **Resilience** – Can the system recover after a failure or attack?

These attributes are usually not included in early prototypes because they belong to the final product. Adding them too early would make prototype development take much longer. Also, **improving one non-functional attribute may affect another**, so depending on the product and available resources, it is important to decide which attributes to prioritize.

System Decomposition

Let's look more closely at system decomposition by defining a few key terms:

- **Service:** a clear and focused unit of functionality.
- **Component:** a software part that provides one or more services.
- **Module:** a set of components.



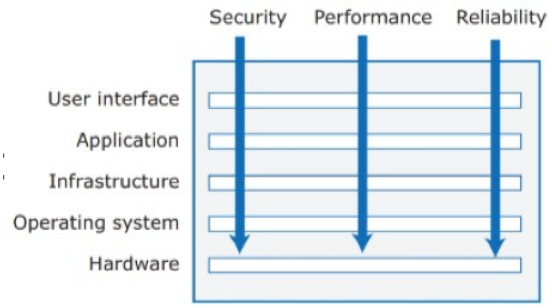
The Agile Manifesto says that simplicity is essential. In system decomposition, this is especially true because as the number of components grows, the relationships between them grow even faster, making the system more complex.

Therefore, it is important to **control complexity**. Some techniques to do this are:

- **Separation of concerns:** Organize your architecture into components that focus on a single concern
- **Stable interfaces:** Design component interfaces that are coherent and that change slowly
- **Implement once:** Avoid duplicating functionality at different places in your architecture

Layered architecture is where each layer deals with a specific concern. The components in the same layer are independent and do not duplicate each other's functions.

Some concerns, such as security, performance, and reliability, are called "**cross-cutting**" because they affect the entire system across all layers. These concerns define how the layers interact with each other.



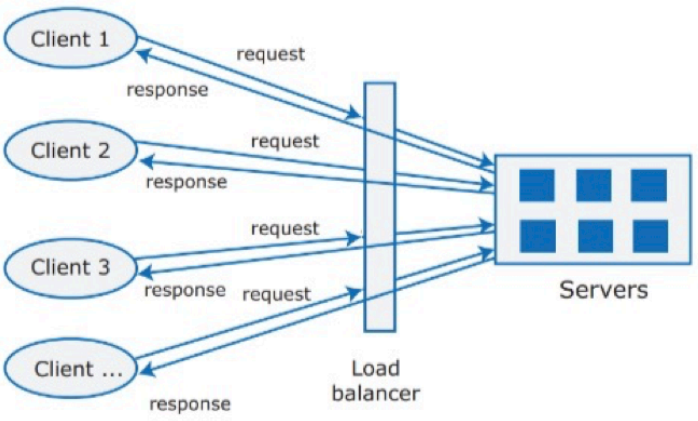
System decomposition must be done together with the selection of technologies for the system. Even though this may seem like mixing design and implementation, it is necessary. For example, choosing a specific type of database can influence how higher-level components are built. Similarly, deciding to support mobile device interfaces means the system will need mobile UI toolkits.

Distribution Architecture

How do we decide on servers and how to assign components to them?

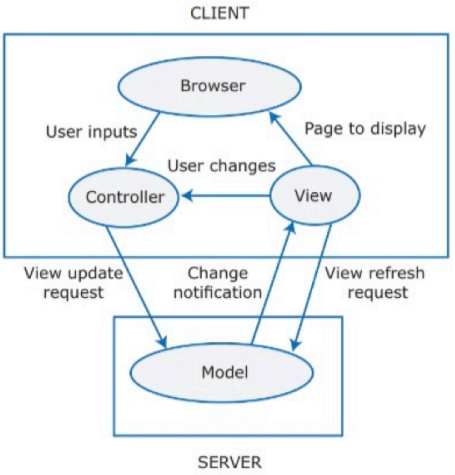
A common approach is the **Client-Server architecture**, also known as Model-View-Controller (MVC). Several servers, load balanced.

Suited to applications in which clients access a shared database and business logic operations on those data



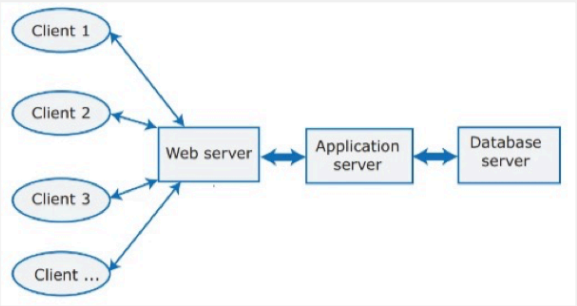
Model-View-Controller pattern

- Architectural pattern for client-server interaction
- Model decoupled from its presentation
- Each view registers with model so that if model changes all views can be updated

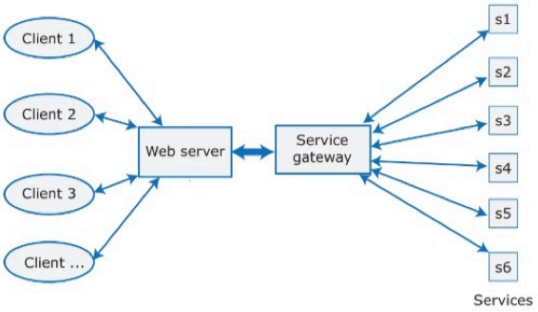


Usually, the client and server communicate using **HTTP and JSON (or XML).**

Multi-tier client-server architecture



Service-oriented architecture



Some important decisions when designing the **distribution architecture** include:

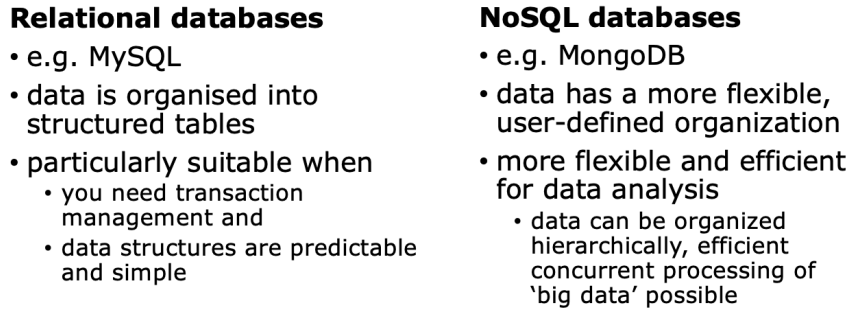
- **Data type and data updates**: should data be centralized or distributed and synchronized later?
- **Change frequency:** if changes happen often, it is better to separate components into individual services to make updates easier and cheaper.
- **System execution platform:** whether the service is online over the internet or part of a business system affects the architecture design.

Technologies Choices

Changing technologies during development is difficult and costly, so it is important to choose them carefully in advance. Some aspects of architecture

are closely linked to technology:

- **Database:** SQL or NoSQL?



- **Platform:** mobile app or web platform?
- **Server:** dedicated in-house servers or cloud?
- **Open-source:** are there suitable open-source solutions to use?
- **Development tools:** do the chosen tools limit or influence the architecture?

✅ 5. Cloud Based Software

Cloud computing provides virtual resources that can be accessed whenever needed, offering many advantages over traditional scaling methods, like buying more physical resources.

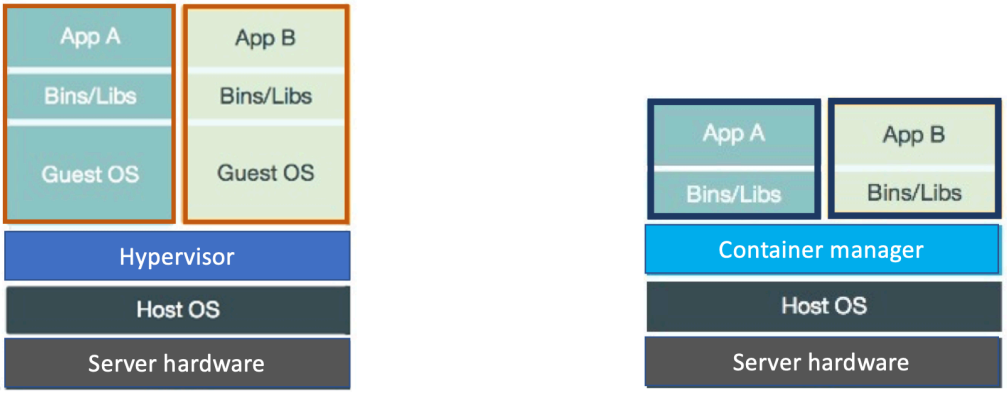
- *Scalability* – the system can grow to handle more users or data.
- *Elasticity* – resources can be added or removed quickly as needed.
- *Resilience* – the system can recover from failures.
- *Cost* – it can be cheaper than buying and maintaining physical hardware.

Virtualization and Containers

Virtual machines on a single physical machine are managed by hypervisor. Many virtual machines (each with its OS) running on same physical server

Containers instead exclude one layer of abstraction, saving up a lot of resources. + lighter and faster to start - share same OS

From VMs to containers



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

Docker uses container-based virtualization to run several separate instances on the same operating system. Software is packaged into **images**, which are read-only templates used to create and run containers. External storage **volumes** can be added to keep data safe when it is shared between containers or used by the host machine.

Multiple Docker images can be combined, and a new image can be created by stacking other images together.

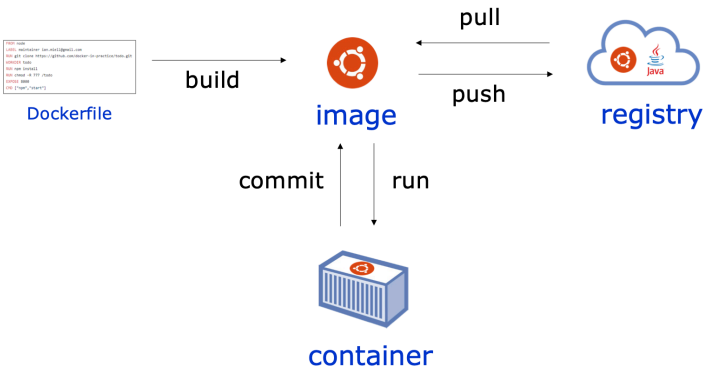
Docker Images

Read-only templates used to create containers.

Stored in a Docker registry, which can be private or public.

- Registries are organized into repositories.
- Each repository holds a set of images for different versions of the software.
- Images are identified by a pair: repository name and tag.

Docker commands



Everything as a Service

In traditional distributed software, customers had to install the software themselves and manage updates, while producers had to maintain multiple versions of the product. In SaaS (Software as a Service), the product is provided as a service, so there is no need to install anything. Users usually pay through a monthly subscription or based on usage to access the service.

- SaaS provides software on-demand for use, accessible via thin clients or APIs
- SaaS provider manages infrastructure + OS + app
- Client responsible for nothing
- Example: salesforce.com

- PaaS provides whole platform as a service (VMs, OS, services, SDKs,…)
- PaaS provider manages infrastructure + OS + enabling SW
- Client responsible for installing and managing app
- Examples: Heroku, Azure, GAE

- IaaS provides (virtualized) servers, storage, networking
- IaaS provider manages all infrastructure
- Client responsible for all other aspects of the deployment (e.g., OS, app)
- Example: EC2, S3

Benefits of SaaS:

Producer point of view:

Pros

- Steady and predictable cash flow.
- Easier and cheaper to manage updates.
- Continuous deployment: a new software version can be released as soon as it is tested.
- Flexible payment options, allowing different subscription plans to attract more users.
- Try-before-you-buy options are easy to offer without worrying about piracy, and they make the product more attractive to new customers.
- Collecting telemetry and usage data is easier and cannot easily be blocked by customers.

Pros and Cons for customers:

Consumer point of view:

Pros

- Access from mobile devices, laptops, and desktops.
- No upfront costs for software or servers.
- Software updates are immediate and automatic.
- Lower costs for managing and maintaining the software.

Cons:

- Must follow privacy regulations.
- Security risks and concerns.
- Software use can be limited by network speed or reliability.
- Sharing data with other services can be difficult if the service does not have a proper API.
- Users have no control over software updates.
- Risk of being locked into a specific service or provider.

Design issues

When designing SaaS, developers must make important decisions:

- **Authentication** method: personal login, federated login, or using Google/LinkedIn credentials.
- **Local vs Remote processing**: Whether some features should be available offline.
- Risk of **information leakage**.
- **Database management**: multi-tenant vs multi-instance, meaning a single shared database or separate copies for each customer.

Multi-tenant

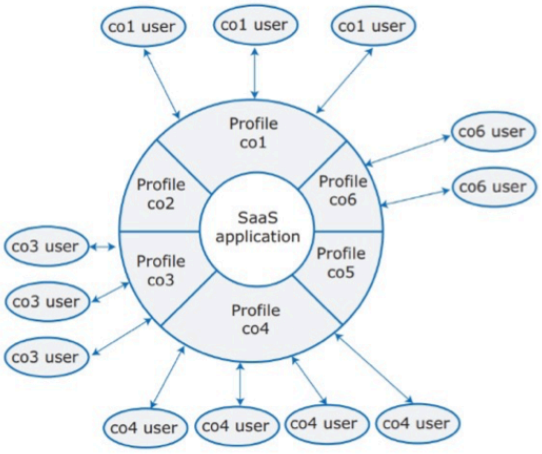
Multi-tenant systems use a single database schema shared by all users. Each database item has a tenant identifier to provide some level of "logical isolation."

Medium and large businesses usually prefer a customized version of the software rather than a generic multi-tenant system. They may want a custom UI and optional fields for certain users. Expanding the database schema for these customizations can lead to **wasted storage**.

Multi-tenant systems: UI configurability

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



Multi-tenant systems: DB schema adaptation

Corporate users may wish to extend or adapt DB schema to meet their specific business needs.

Solution I: Add a number of extra fields to each table and allow customers to use these fields as they wish

Issues:

- Difficult to know how many extra columns to include (too few will be insufficient, too many will lead to wasted space)
- Different customers are likely to need different types of columns

Solution II: Add a field to each table that identifies a separate "extension table" and allow customers to create these extension tables to reflect their needs.

Issue: added complexity

Security is a major concern in multi-tenant systems because a centralized database can be a single point of failure for data leaks or damage. A common solution is **multi-level access control**, checking access at both the organization and individual level. **Encryption of data** can help but may slow down performance. Usually only sensitive data are encrypted.

Multi-instance

Multi-instance systems can use **virtual machines (VMs) or containers.**

With containers, each user has an isolated version of the software and database, which is good for users working independently. Since there is no shared data structure, security is easier to manage.

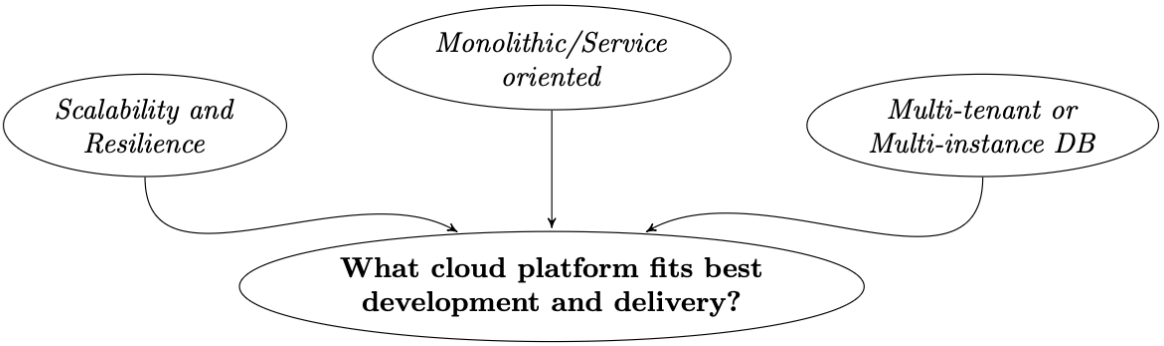
With VM-based solutions, each customer has a VM running their own database instance accessible only to their users.

Multi-instance databases offer flexibility, security, scalability, and resilience. However, updates are harder to manage, and renting cloud VMs can be expensive.

In summary, there are three ways to provide customer **databases in a cloud system**:

1. **Multi-tenant:** a single database shared by all customers, usually hosted on large cloud servers.
2. **Multi-instance on VMs:** each customer has their own database running on a virtual machine.
3. **Multi-instance on containers:** each customer's database runs in its own container and may be spread across multiple containers.

Architectural Decisions

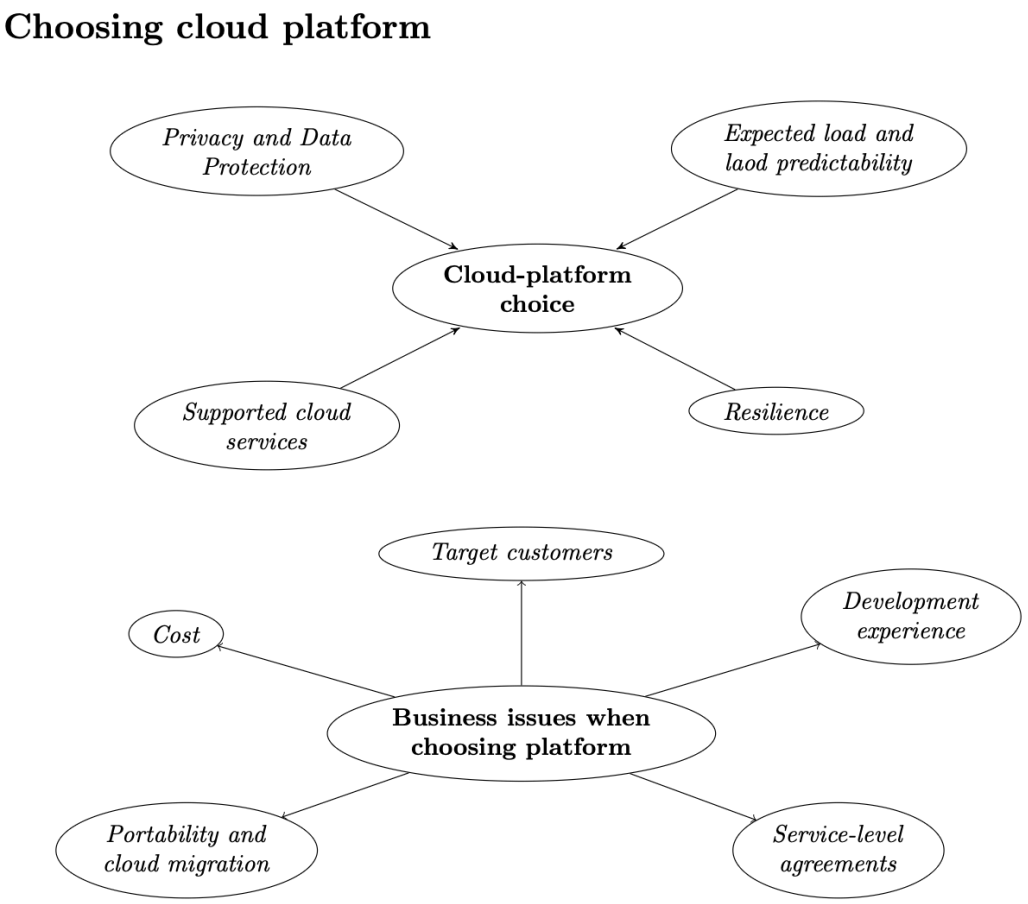


Scalability

To make cloud-based systems scalable, the software must be designed so that individual components can be copied and run in parallel. In addition, a load-balancing system must be used to distribute work evenly.

Resilience

Resilience can be achieved using **hot or cold standby** systems. Cold standby means the system is restarted using backup data after a failure. Hot standby means a duplicate system with a mirrored database runs at the same time as the main system. If the main system fails, the backup can immediately take over while the main system recovers. Hot standby is more expensive but more effective.



✅ 6. Microservices Architecture

Components can be developed at the same time by different teams, and they can be reused, replaced, or run on multiple computers.

To be effective, components must be easy to copy, run in parallel, and move between systems. This allows cloud-based scalability, reliability, and elasticity. A simple way to design such components is to use **stateless services** that store persistent data in a local database.

Software Service

- ***Definition*** – a software component that can be accessed over the Internet.
- Given an input, a service produces an output without side effects.
- Services are accessed through a public interface, and the implementation details are hidden.
- Services do not keep internal state.

- State information is stored in a database or managed by the requester of the service.
- State information can also be included in the service request, and updated state can be returned in the service response.
- Services can be moved between virtual servers, improving scalability.

Amazon changed the way services are designed:

- Each service should handle a single business function.
- Services should be fully independent, with their own database.
- Each service should manage its own user interface.
- Services should be easy to replace or copy without affecting other services.

This approach leads to microservices: small, stateless services that focus on a single responsibility.

Microservices

- **Small services** that can be combined to build applications.
- **Independent** – changes to one service should not affect others.
- Can be modified and redeployed **without stopping or changing** other services.

More specific characteristics of microservices:

- **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 in their implementation.
- **Deployable independently**: 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.

When considering the “size” of a microservice, two measures are useful:

1. **Coupling** – the number of relationships with other services (inter-component).
    - ***Low coupling*** → services are independent and can be updated independently.
    - Idea → if two services interact too much, they should be merged into a single service.
2. **Cohesion** – the number of internal relationships within the service (intra-component).
    - ***High cohesion*** → less communication needed with other services.
    - Idea → if a service communicates too much internally, it should be split into smaller services.

These measures follow the “**Single Responsibility Principle**,” which says that each service should focus on doing one thing well.

Responsibility does not mean a single functional activity.

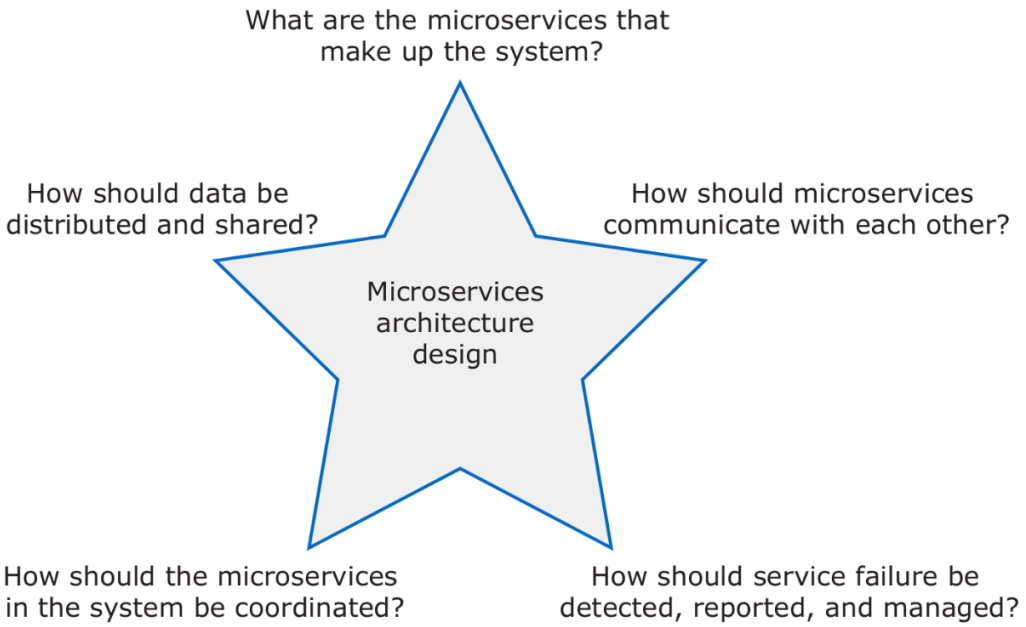
Another way to decide the size of a microservice is the “**Rule of Twos**”: a service should be developed, tested, and deployed in two weeks by a team that can be fed with two large pizzas (8–10 people). Services need enough people not just for development and testing but also for decoupling, support, and maintenance after deployment.

Motivations

1. Faster rebuilding and redeployment, which shortens the time needed for new features and updates.
2. Efficient scaling of the system.

Microservices architecture supports both of these points well. Each microservice can run in its own container, allowing it to be stopped, restarted, or scaled independently without affecting other services. This also makes it easy to deploy additional copies of a service quickly.

Design Decisions



Decomposing a system into microservices is not simple. There should not be too many services (which increases communication overhead) and not too few (which reduces independence for updates, deployment, and scaling). Key points for decomposition:

- Balance small, focused functionality with overall system performance.
- Follow the “common closure principle”: elements likely to change together should stay in the same service.
- Align services with business capabilities.
- Services should only access the data they need and use proper mechanisms to share data when required.

Service Communications

Ideally, each microservice should manage its own data

Challenge of handling possible **data dependencies. To deal with data dependencies:**

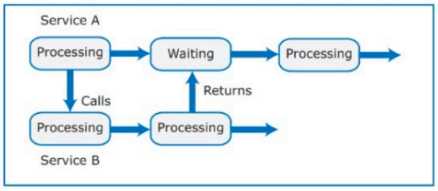
1. Data sharing should be minimized.

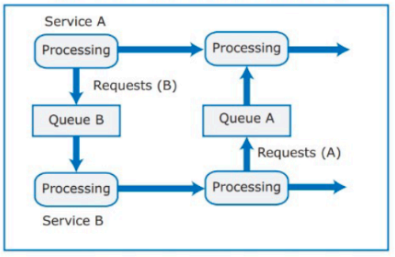
2. Most sharing should be read-only, few services should be responsible for updating data.
3. It is also recommended to have a system that keeps **database copies consistent** for replicated services.

Shared database systems use **ACID transactions** to order updates and avoid inconsistencies. In distributed microservices systems, there is a trade-off between data consistency and performance. Therefore, microservices must be designed to handle some level of temporary data inconsistency.

1. ***Dependent data inconsistency*** – Actions or failures in one service can make the data managed by another service inconsistent.
2. ***Replica inconsistency*** – Multiple replicas of the same service may run at the same time, each updating its own database copy. These databases need to become "eventually consistent."

Synchronous vs. asynchronous service interaction

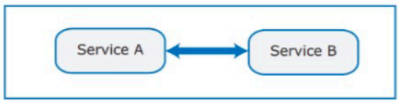


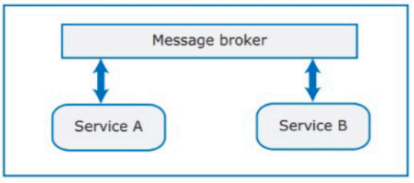


+ easier to write and undestand

+ looser coupling, more efficient
- more difficult to write to write and understand

Direct vs. indirect service communication





Messages directly sent to service address

Messages addressed to service name
Messages sent to message broker, which
finds address of requested service
can handle message translation
Example of message broker: RabbitMQ

+ Simpler
+ Faster
- Requester must know recipient's URI

+ Can support synchronous & asynchr. interactions
+ Easier to modify/replace services
- More complex
- Slower

CAP & Saga

CAP Theorem

A web service cannot guarantee Consistency, Availability, and Partition-tolerance all at the same time. When there is a network partition, it is impossible to have both Availability and Consistency.

1. **Consistency** – Each service returns the «right» response to each request.
2. **Availability** – Each request must get a response.
3. **Partition-tolerance** – The system can be split into groups and the network can lose or delay many messages between services.

Practical implications:

Solution 1: (A ∧ P): Guarantee availability, and provide best-effort consistency

Solution 2: (C ∧ P): Guarantee strong consistency, and provide best-effort availability

Saga Pattern

The Saga pattern helps manage consistency across transactions. Each business transaction that involves multiple services is implemented as a saga, which is a series of local transactions. Each local transaction updates its database and triggers the next transaction. If a local transaction fails, the saga runs compensating transactions to fix the problem.

Coordinating Sagas two ways:

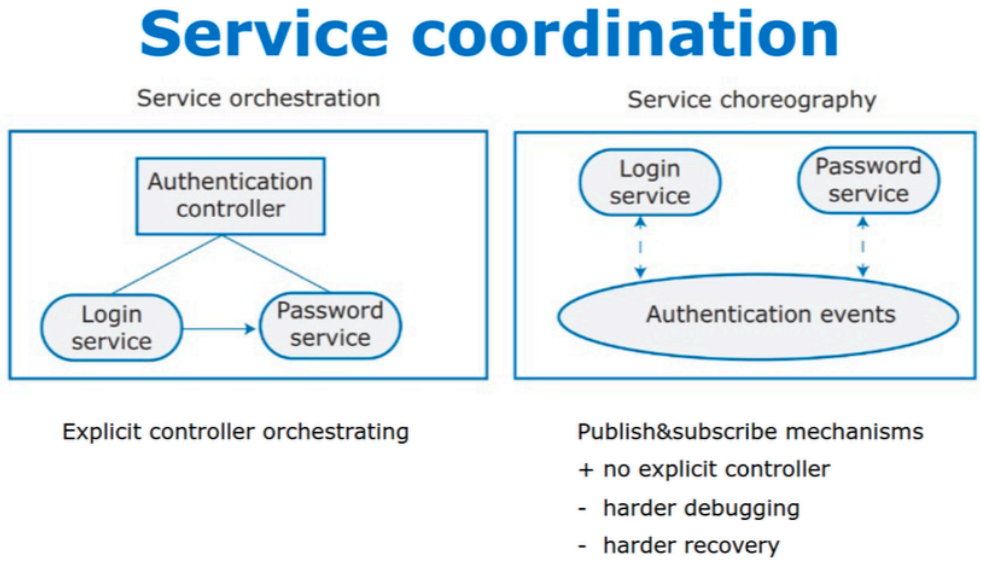
1. ***Choreography*** – Each local transaction publishes an event that triggers the next transaction(s).
2. ***Orchestration*** – An orchestrator tells participants which local transactions to run.2

Compensating Transactions:

1. Backward model – Undo the changes made by previous local transactions.
2. Forward model – Retry the transaction later.

Netflix Approach

Netflix ensures eventual consistency using Apache Cassandra. To replicate data across n nodes: “write to the nodes you can reach, then fix the rest later.” A quorum is used as a threshold, meaning at least (n/2 + 1) replicas must respond.



Tip: start with orchestration, switch to choreography only if product inflexible/hard to update

Failure Management

Problems will always happen, so a system must be able to handle failures. For example, if a service S calls two other services, A and B, each with 99% availability:

Downtime(S) = 30 minutes per day. 30 minutes of downtime per day is significant!

Types of failures:

1. **Internal failure** – A problem detected by a service that can be reported to the requesting service, for example, an invalid link or resource not found.
2. **External failure** – A problem caused by something outside the service, which can make it unavailable or unresponsive.
3. **Performance failure** – When the system becomes too slow or its performance drops to an unacceptable level.

Ways to handle failures:

- Circuit breakers – These use timeouts to handle unresponsive services and prevent cascading failures.
- Chaos Monkey – This testing approach deliberately causes random failures in the system VM instances and containers at a certain rate to check how well it can cope.

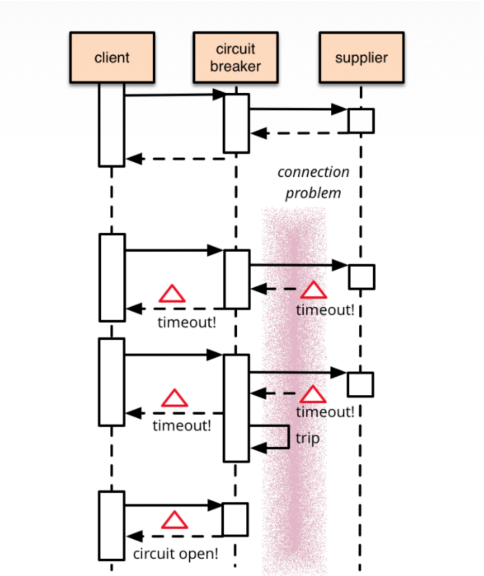


Figure 10.4: Circuit breaker

DevOps: The same team is responsible for developing, deploying, and managing services.

Monitoring: Testing alone cannot catch all problems, so monitoring deployed services is essential. Monitoring should be enough to detect issues but not so heavy that it slows down the system or affects usability.

Monitoring helps detect service failures and allows rollback if needed. When releasing a new service version, the old version is kept, and only the "current version link" is updated to point to the new service. This way, it is possible to revert to the old version if necessary.

Continuous Deployment pipeline:

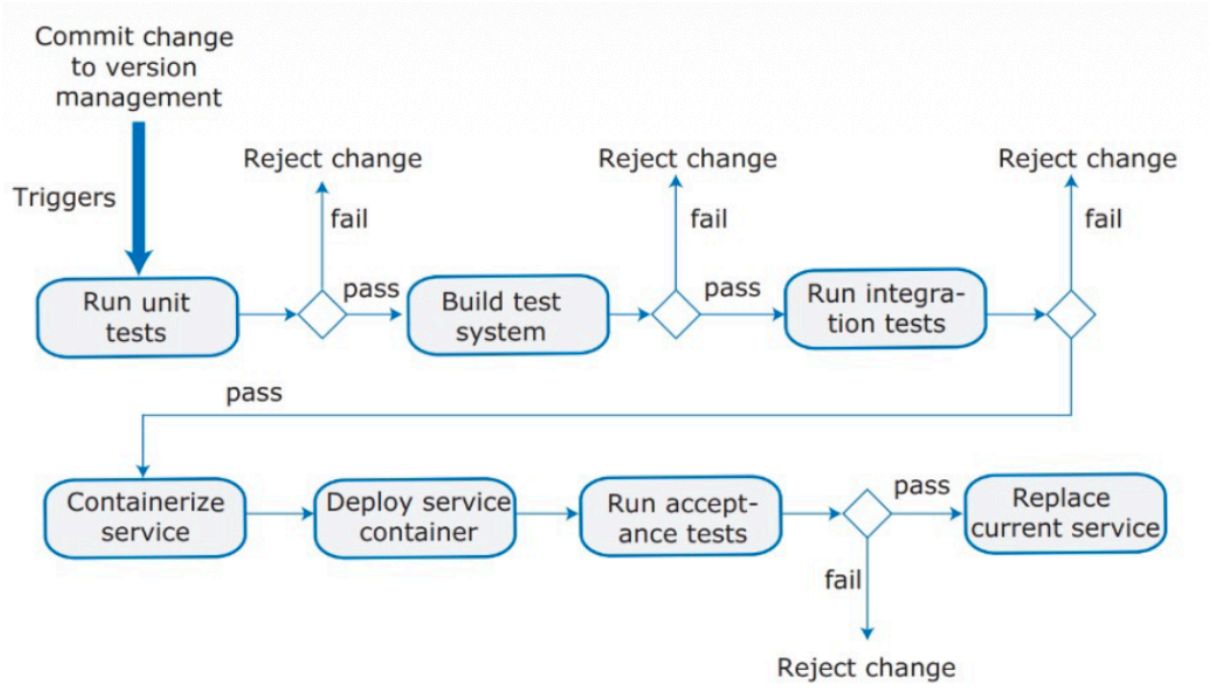


Figure 10.5: Continuous deployment pipeline

Concluding Remarks

Microservices Pros:

- Shorter lead time
- Effective scaling

Microservices Cons:

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

> **“Dont even consider microservices unless you have a system that’s too complex to manage as a monolith”** - Fowler

✅ 7. Architectural Smells and Refactoring

Definition

A review of white and grey literature was done to find the most common architectural problems **("smells")** in microservices and the best ways **("refactorings")** to fix them. In simple terms, the goal is to help a system designer check if their system follows microservices principles, and if not, understand how to improve it.

An **architectural smell** is a commonly used architectural decision that negatively impacts system lifecycle qualities.

Design Principles

1. **Independent deployability:** Each microservice in an application should be deployable on its own.
2. **Horizontal scalability:** Microservices should be able to scale horizontally, meaning more instances can be added easily when needed.
3. **Isolation of failures:** When one service fails, it should not cause other services to fail too.
4. **Decentralization:** Control and management should be spread out across all parts of the system, including data and governance, rather than being centralized.

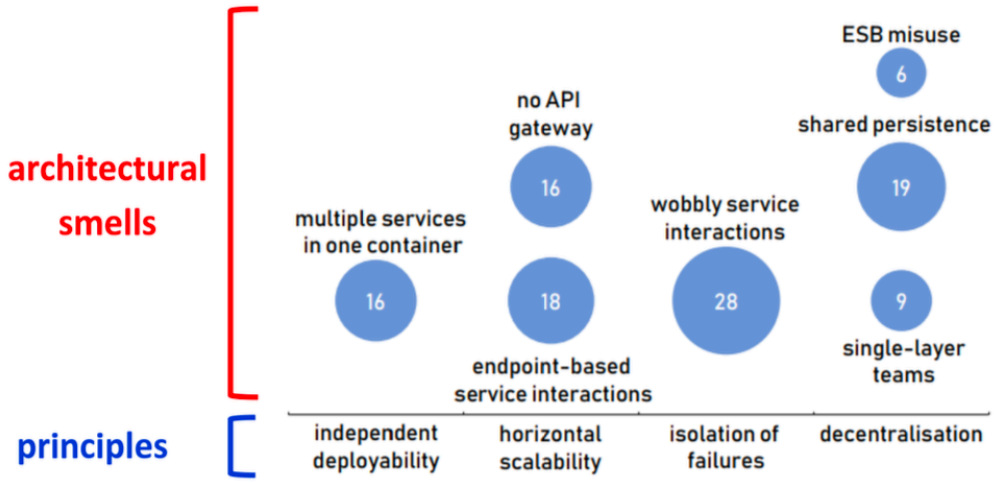
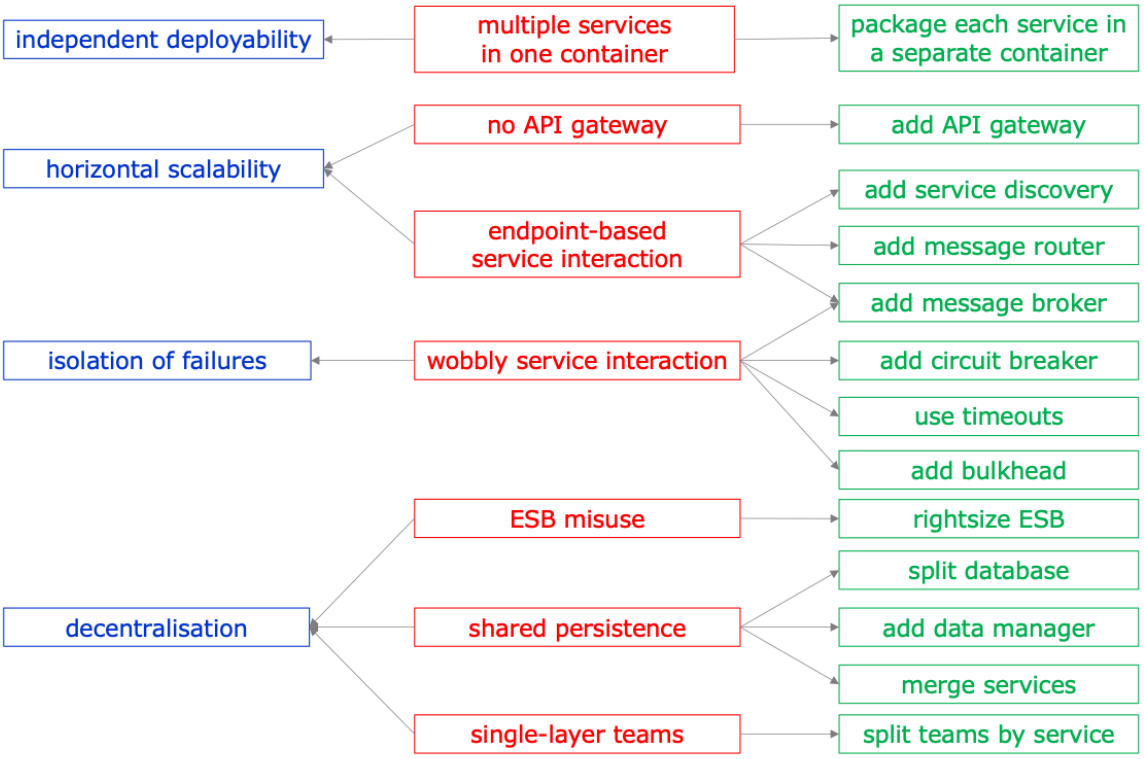


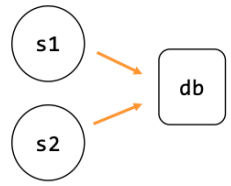
Figure 11.1: Smells Principles
Here we can see smells, i.e. wrong choices, and the principle they violate



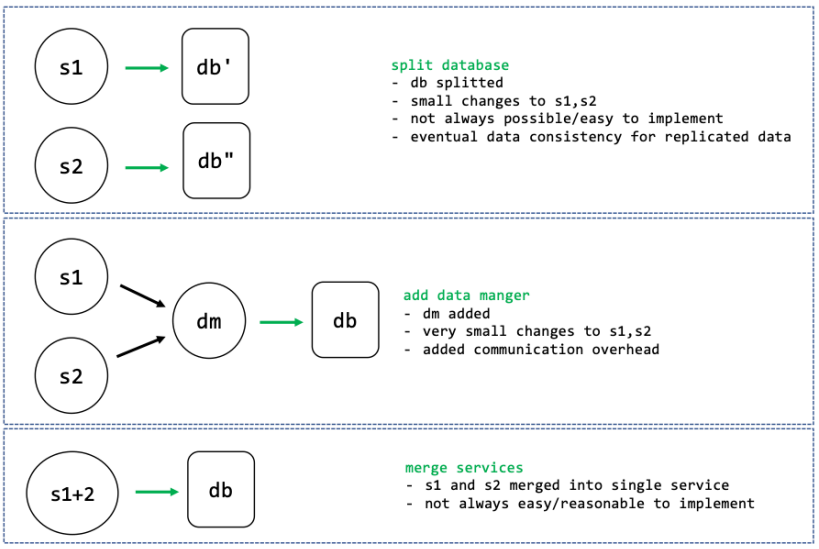
The interaction of m1 with m2 is ***wobbly*** when a failure of m2 can trigger a failure of m1.

> ## Multiple service access same DB is BAD, because:

- **Different services managed by different teams, teams must work independently**
- **Services must be developed, deployed and scaled independently**



shared persistence



MicroFreshener is a tool for:

- Editing app specifications
- Automatically identifying architectural smells
- Applying architectural refactorings to resolve the identified smells

Architecture level ≠ Implementation level

Applying a refactoring in MicroFreshener does not change the implementation of the application. Concrete implementation of refactoring left to application manager

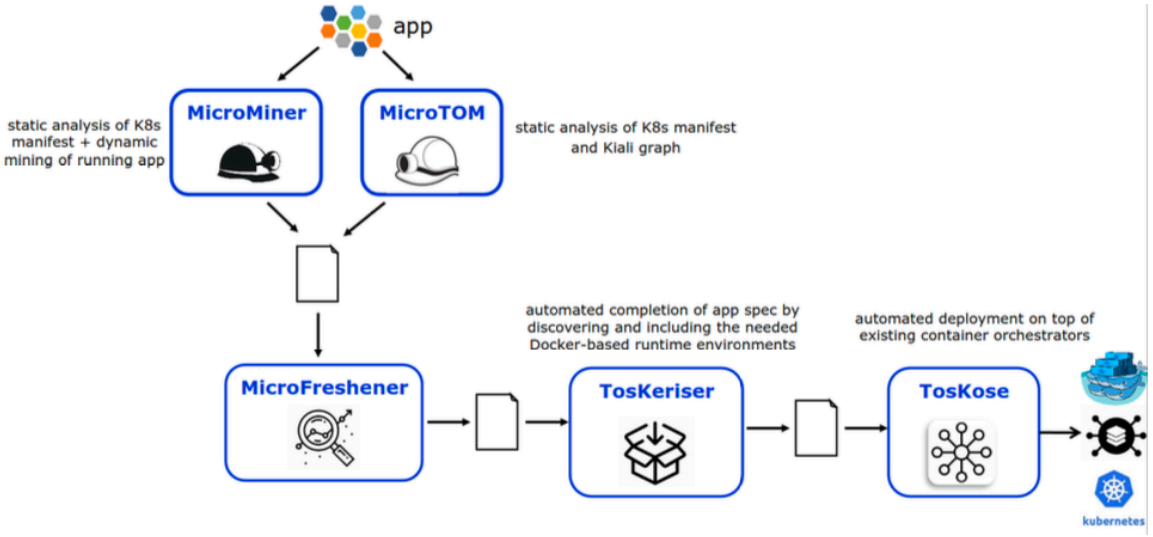


Figure 11.3: Microservices toolchain

✅ 8. Splitting The Monolith

The monolith grows over time

- New features and code are constantly added.
- **Low cohesion:** Unrelated parts of code are often kept together.
- **Tight coupling:** A small change can affect other parts of the system, and the whole monolith must be redeployed.

> **"Split the monolith when it becomes a problem."**

"Seam"-driven splitting

Find the seams: A *seam* is a part of the code that can be worked on separately without affecting the rest of the system.

Identify seams that can become **service boundaries.** We should organize the code around these seams:

Create **packages** that represent specific contexts and move existing code into these packages.

- Do **incremental refactoring and testing** step by step.
- Any leftover code may show new bounded contexts.

- Use code analysis to understand dependencies between packages.

Where to start breaking?

Start where you can get the **most benefit.**

Typical drivers:

- **Pace of Change:** Some parts change more often than others.
- **Team Structure:** Teams can own different services.
- **Security:** Separate areas with different access levels.
- **Technology:** Different parts may need different tech stacks.
- **Tangled Dependencies:** Start with the seam that has the fewest dependencies.

Databases – the main source of tangled dependencies

Databases often connect multiple services together. We need to find **seams inside the database** to separate them cleanly.

When splitting a database, we need to:

- Understand which parts of the code read from or write to it.
- Find database-level constraints.
- Notice if the same table is used by different bounded contexts.

Databases example: **Breaking foreign key relationships**

Catalog code uses *Line items* table to store info on an album

Finance code uses *Ledger* table to track financial info

To generate monthly reports like “We sold 600 copies of Tones and I’s Dance monkey and made 2,000 USD”

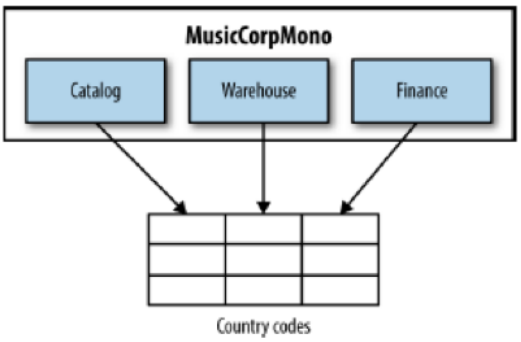
Finance code accesses *Line items table*, and a foreign-key relationship from *Ledger* table to *Line items table* exists

How to stop Finance code accessing the *Line items* table?

Solution: Expose data via API in the *Catalog* package

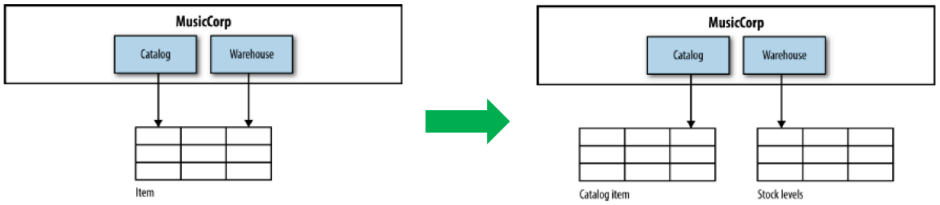
- overhead introduced (yes)
- foreign key relationship lost
  - constraints will have to be managed in the services rather at the database level
  - may* need to implement consistency check across services, or to trigger actions to clean up related data

Databases example: **Shared *static* data**



Solutions:
(1) Duplicate table for each package: potential consistency challenge
(2) Treat this shared static data as code: consistency issues remain
(3) Push static data into a separate service: overkill?

Databases example: **Shared tables**



- *Catalog* needs to store name and price of record we sell
- *Warehouse* needs to keep electronic record of inventory
- Two concepts stored in same *Item table*

Solution: Store the two concepts separately, split table in two

---

What to split first: schemas or services?

Splitting schemas first can:

- Increase the number of database calls.
- Break transactional integrity.

However, if you split schemas but keep the application code together, it's easier to roll back if something goes wrong.

Transactions

Transactions make sure that all related changes happen together, keeping data consistent.

Example problem:

If inserting an order record succeeds, but inserting a picking record fails, what happens?

Bad solutions:

1. Abort everything (undo) and use **compensating transactions.**
2. Use **distributed transactions.** *Transaction manager* orchestrates the various transactions being done. *Two-phase commit*
    - Voting phase: each participant tells manager whether its local transaction can go ahead
    - If transaction manager gets a *yes* vote from all participants, it tells them all to go ahead and perform their commits – otherwise it sends a

rollback to all parties.

Vulnerabilities: If transaction manager goes down, pending transactions never complete. If one participant fails to respond during voting, everything blocks

Better solution:

Use **eventual consistency** — retry later until everything succeeds.

The Reporting Database

Reporting usually needs data from many parts of the system to generate useful output. In monolithic systems, reports often run on a **read replica** of the main database for performance.

What to do if database has been split? When reporting involves multiple systems, Solutions:

1. **Data retrieval via service calls** – ask each service for needed data via API calls. Doesn't scale well with large volumes of data. Exposed API may not be designed for reporting.
2. **Data pumps** – copy data into a reporting database. Data is pushed to the reporting system. Data pump maps service db to reporting schema. Coupling worth to pay to make reporting easier
3. **Event data pumps** – use events to keep reporting data up to date. Microservices can emit events based on the state change of entities that they manage. Write event subscriber that pumps data into reporting database

✅ 9. Kubernetes

Kubernetes answers key questions such as:

- What happens if a container crashes?
- What happens if the machine running a container fails?
- How can containers communicate with each other through a network?
- In a system with many machines, which machine should run a container?

The solution is **container orchestration**, and the main tool for it is **Kubernetes (K8s)**.

Kubernetes manages the full lifecycle of containers. It starts and stops containers as needed.

For example, if a container crashes unexpectedly, Kubernetes automatically creates a new one to replace it.

Kubernetes also provides a way for applications to communicate with each other, even when containers are being created or destroyed in the background.

When you have a set of container workloads and a set of machines in a cluster, the orchestrator decides the **best machine** for each container to run on.

Design Principles

A key idea in Kubernetes is **declarativeness** — you describe what you *want* your system to look like, and Kubernetes makes sure it stays that way.

The **desired state** of your system is defined using a set of **objects**.

- Each object has:
  - a **specification** (the desired state), and
  - a **status** (the current state).
- Kubernetes constantly checks each object to make sure the status matches the specification.
- If an object stops responding, Kubernetes will create a new one to replace it.
- If the status is different from what is desired, Kubernetes will take action to bring it back to the correct state.

Kubernetes works very well with **microservice architectures** and supports the idea of **decoupling** — breaking a system into smaller, independent services that can be scaled or updated separately.

Kubernetes is built around the idea of **immutable infrastructure**.

“Immutable” means that containers should not be changed while running.

You should not log into a container and manually change files, code, or libraries.

Instead, since containers are **temporary** by design, when an update is needed, you should:

- build a **new container image**,
- deploy it, and
- replace the old one.

This also makes **rollbacks** easy — you can simply go back to a previous container image if something goes wrong.

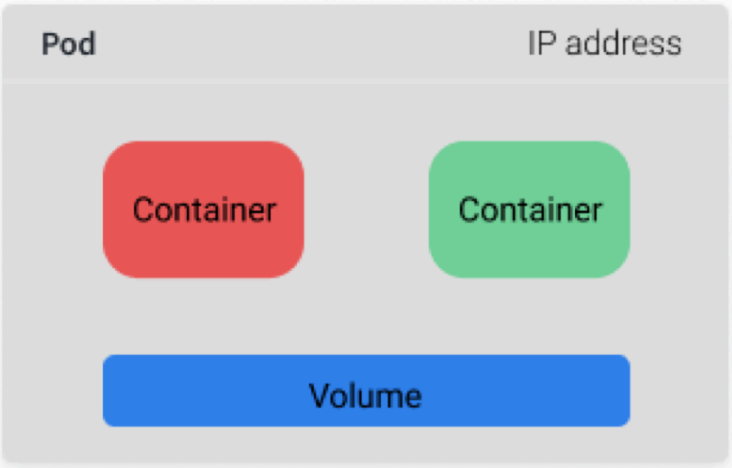
K8s Objects

Pod

A **Pod** is the smallest unit in Kubernetes. It contains one or more **closely related containers**, which share:

- the same **network**, and
- the same **file storage volumes**.

The containers inside a Pod usually work together and need to communicate easily.



Deployment

A **Deployment** object manages a group of Pods. It defines:

- a **template** (the Pod design), and
- a **replica count** (how many copies of that Pod should run).

Kubernetes always tries to keep the defined number of Pods running.

For example, if a Deployment has **10 replicas** and **3 Pods crash**, Kubernetes will automatically start **3 new Pods** on other machines in the cluster to replace them.

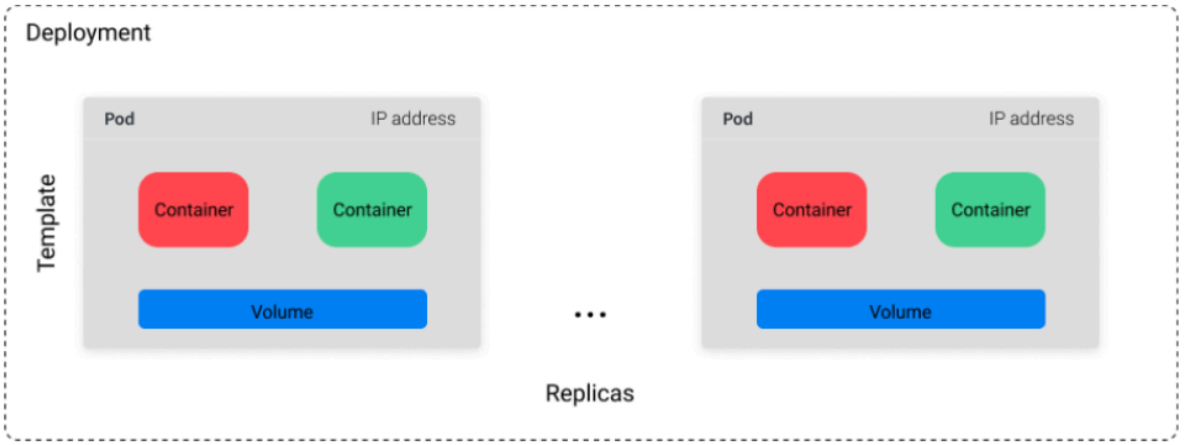


Figure 9.1: Deployment Object

Service

Each **Pod** has its own unique **IP address**, which can be used to communicate with it.

But there is a problem: the set of Pods in a **Deployment** can change at any time — some Pods may stop, be replaced, or move to other machines.

So, a valid question is: **How can we communicate with Pods if their IP addresses keep changing?**

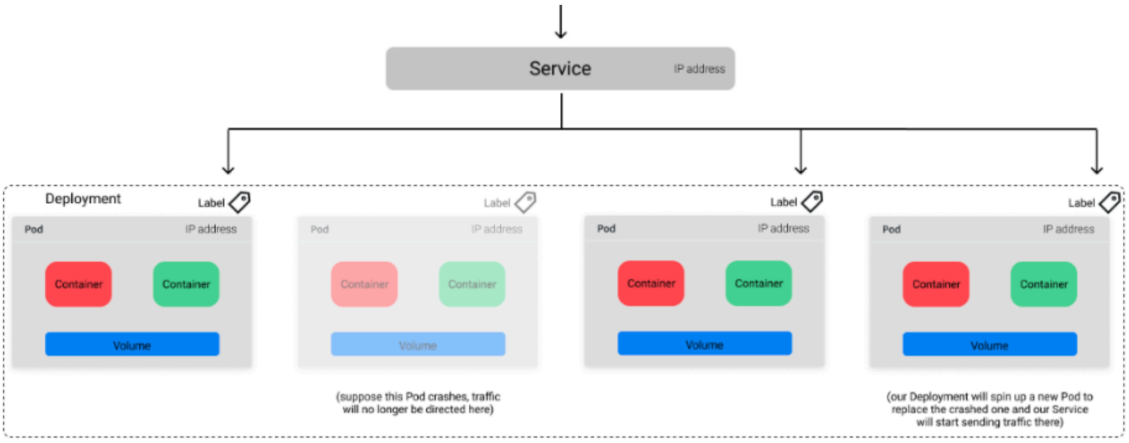


Figure 9.2: Service Object

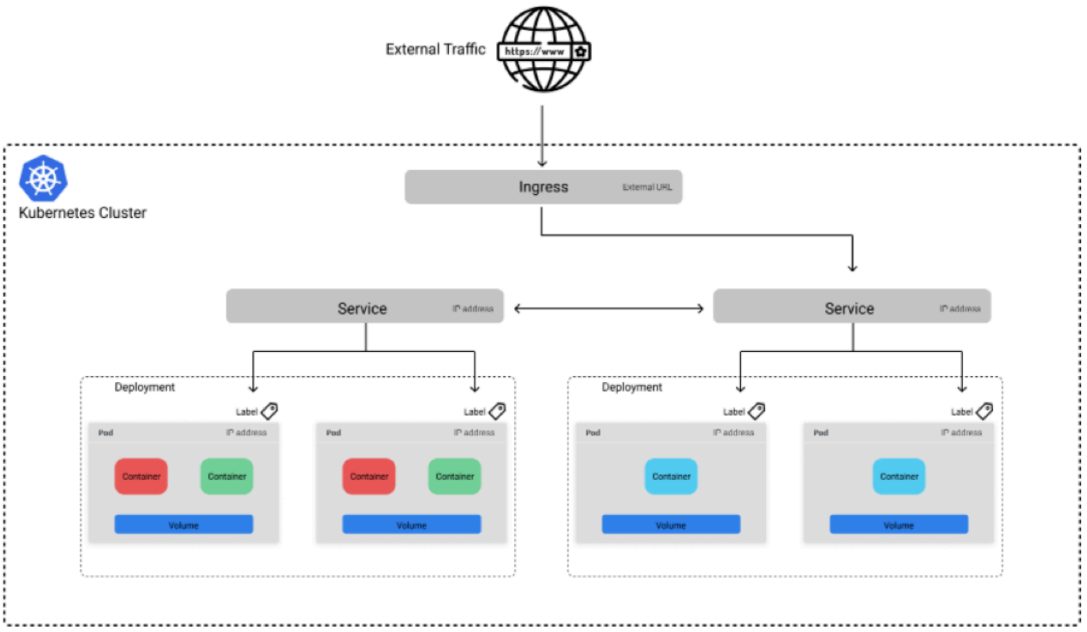
The **K8s Service** object provides a **stable endpoint** that directs traffic to the correct Pods, even when the actual Pods change because of updates, scaling, or failures.

A **Service** knows which Pods to send traffic to by using **labels** (key-value pairs) that are defined in the **Pod metadata**.

Ingress

Service objects let us expose applications through a stable endpoint, but only for **internal cluster traffic**.

To make an application available to **external traffic** (outside the cluster), we need to create an **Ingress** object. The **Ingress** defines which **Services** should be made publicly accessible.



Control Plane

A Kubernetes cluster has two main types of machines:

1. Master node – usually one machine that contains most of the **control plane components**.
2. Worker node – machines that actually run the **application workloads**.

Let's look at the components of both types.

Master Node

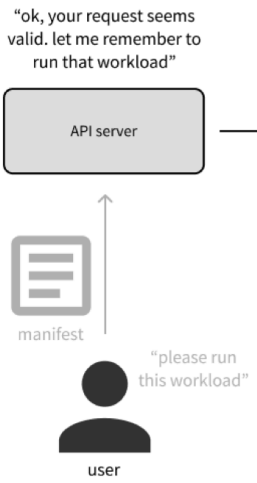
The **master node** manages and controls the cluster.

When a user sends a new or updated **object specification**, it goes to the **API server** on the master node.

The API server:

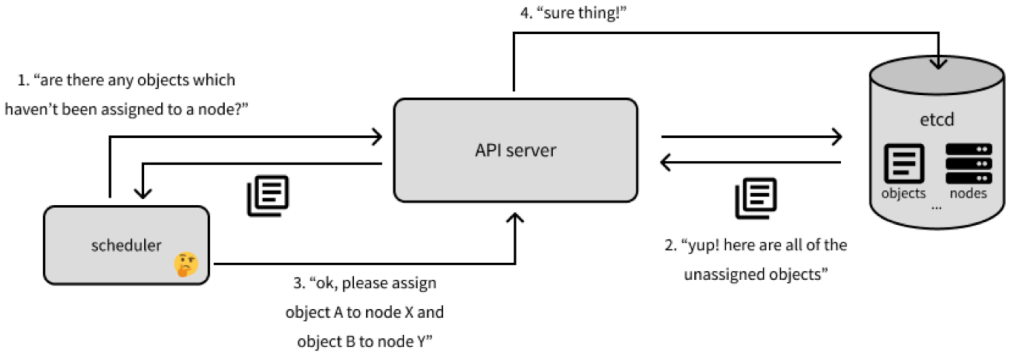
- checks and validates the request, and
- acts as the main interface for any information about the cluster's current state.

  This state information is stored in a distributed key-value database called **etcd**.



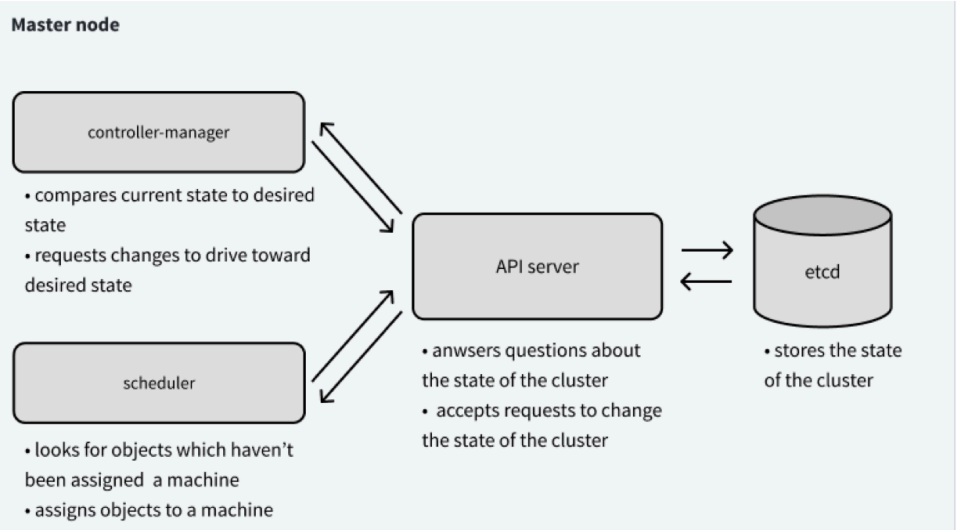
The **scheduler** decides where (on which worker node) the objects should run. It:

- asks the API server which objects have not yet been assigned to a machine,
- chooses which machines they should go to, and
- updates the API server with these assignments.



The **controller-manager** keeps watching the cluster's state through the API server.

If the **actual state** of the cluster is different from the **desired state**, the controller-manager makes the needed changes (through the API server) to bring the cluster back to the desired state.

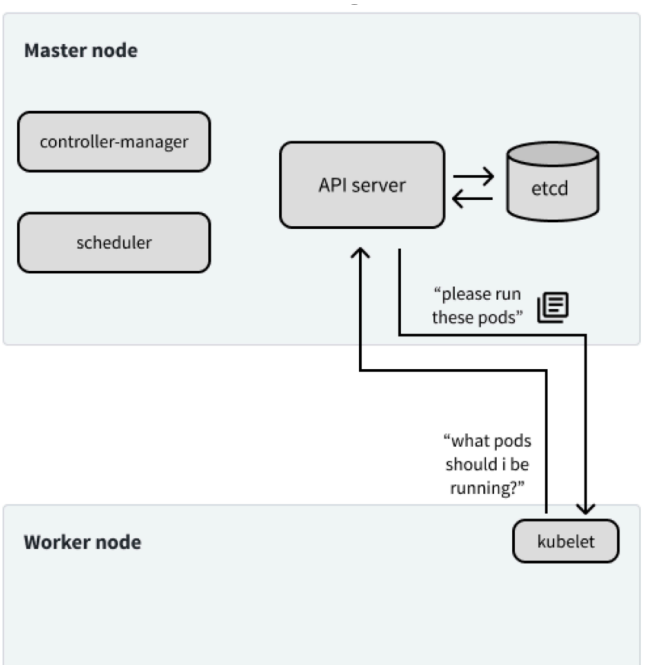


Worker Node

The **kubelet** is the "agent" running on each worker node. It communicates with the API server to find out which container workloads have been assigned to that node.

The kubelet is responsible for:

- starting the Pods that should run on the node, and
- registering the node with the API server when it first joins the cluster, so that the scheduler can assign Pods to it.



The **kube-proxy** allows containers to communicate with each other across different nodes in the cluster.

Apart from these two components, the rest of the work on the node is done by the **Pods**, which were discussed earlier.

Concluding Remarks

When should you not use Kubernetes?

⋄ When your workload can easily run on a **single machine**.

⋄ When your **computing needs are small**.

⋄ When you **don’t need high availability** and can handle some downtime.

⋄ When you **don’t plan to update or change** your deployed services often.

⋄ When you have a **monolithic application** and don’t plan to split it into microservices.

In these cases, a simpler and easier alternative is **Docker Swarm**, which is usually chosen when **simplicity** and **fast development** are more important than advanced features.

✅ 10. Testing

Types of Testing

There are different types of testing, depending on their purpose:

1. **Functional Testing:** Checks that the system's functions work as expected and helps find possible bugs.
2. **User Testing:** Checks that the product is useful and easy to use for end users.
    - **Alpha testing:** "Do you really want these features?" (tests early design and feature choices).
    - **Beta testing:** An early version of the product is given to users to test how well it works and how it interacts with other systems.
3. **Performance and Load Testing:** Checks that the system responds quickly to requests and can handle different levels of usage without problems as the load increases.
4. **Security Testing:** Looks for weaknesses or vulnerabilities that attackers could exploit.

There are two important things to remember about testing:

- As Edsger W. Dijkstra said:

  "Program testing can be used to show the presence of bugs, but never to show their absence." This means testing can find errors, but it can't prove that there are none.

- **Software testing** is not the same as **software verification.**

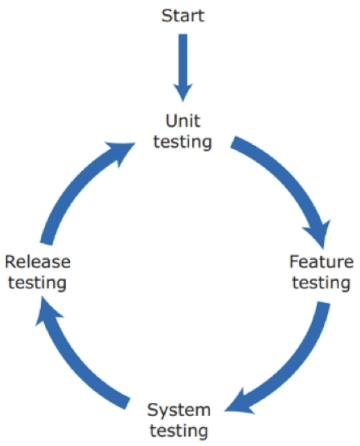
  Verification means using models or formal methods to prove that the software has certain properties.

Functional Testing

The first important point in functional testing is **testing coverage** — every part of the code should be executed at least once during testing.

Testing should start **as soon as you begin writing code** and should be **automated** as much as possible.

Automation makes the development and testing cycle faster and allows functional testing to be done in clear, repeatable stages.



Unit Testing

Unit testing means testing small, individual parts of a program (such as a function or method) **in isolation**.

Unit Testing Principle (Theorem 14.1.1): If a program unit behaves correctly for a group of input values that share similar characteristics, it should also behave correctly for all other inputs that share those same characteristics.

Example: If your program works correctly for the inputs 1, 5, 17, 45, and 99, you can assume it will also work correctly for other numbers between 1 and 99.

To apply this principle, it's important to identify **equivalence partitions** — groups of inputs that are treated the same by the program. You only need to test a few inputs from each partition to cover all possible cases. Also, remember that **errors often occur at boundaries**.

So, when defining partitions, test the inputs that are **at or near the edges** of these boundaries.

Unit testing: Guidelines

Guideline	Explanation
Test edge cases	If your partition has upper and lower bounds (e.g., length of strings, numbers, etc.), choose inputs at the edges of the range.
Force errors	Choose test inputs that force the system to generate all error messages. Choose test inputs that should generate invalid outputs.
Fill buffers	Choose test inputs that cause all input buffers to overflow.
Repeat yourself	Repeat the same test input or series of inputs several times.
Overflow and underflow	If your program 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 transformations, keep count of the number of elements in each list and check that these are consistent after each transformation.
One is different	If your program deals with sequences, always test with sequences that have a single value.

Feature Testing

Feature testing checks that a feature works as expected and that it satisfies real user needs.

There are two main types of feature testing:

1. **Interaction Testing:** Tests how multiple units or components work together, especially when they were developed by different teams.
2. **Usefulness Testing:** Focuses on whether the feature is something users actually want or find valuable. The **product manager** usually helps with this type of testing.

From user stories to feature tests

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.

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.

System Testing

Besides testing individual features, the **entire system** should also be tested as a whole.

To do this, it is recommended to use **scenarios** or **user stories** that represent real end-to-end user actions. This helps test the system from the user's point of view.

Testing the whole system is important for several reasons:

1. To find **unexpected or unwanted interactions** between different features.
2. To check if **all features work well together** to support what users actually need.
3. To make sure the system **works correctly in different environments** where it will be used.
4. To test important **quality attributes** such as speed, capacity, and security.

Release testing focuses on testing the **final version** of the system before it is given to customers.

It is done in the **real operational environment**, not just in a testing environment. The goal of release testing is to decide whether the system is **ready to be released**, not to find new bugs.

Preparing a system for release includes:

- packaging the system for deployment,
- installing all needed software and libraries, and
- configuring the right parameters.

Many mistakes can happen during this process, so release testing is necessary to make sure everything works correctly. If the system is deployed **on the cloud**, an **automated continuous release process** can be used to make deployment faster and more reliable.

Test Automation

Automated testing is commonly used in software development. There are many **testing frameworks** available for all major programming languages to make this process easier.

Automated (executable) tests check whether the software gives the expected results for given input data.

A good way to organize automated tests is to follow three steps:

1. **Arrange:** Set up everything needed for the test — define test parameters, create mock objects, and prepare the environment.
2. **Action:** Run or call the part of the program (unit) that you want to test using the prepared inputs.
3. **Assert:** Check that the result matches what you expected if the test ran successfully.

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

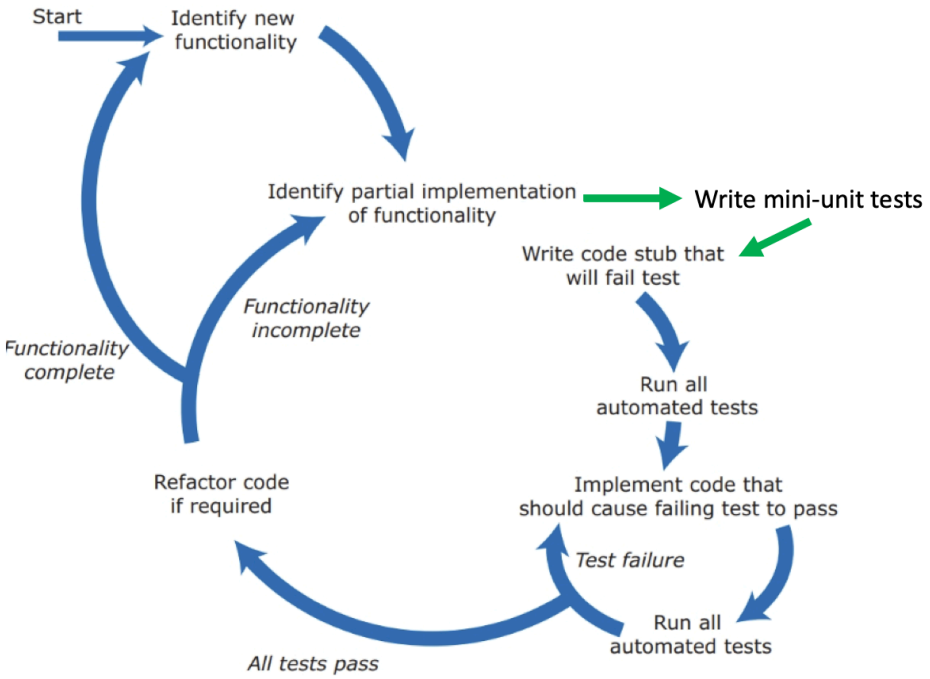
*Keep in mind that **test code can also contain bugs**.* To avoid this, tests should be **as simple as possible** and should be **reviewed together with the code** they test.

Unit tests are the easiest type to automate. If done well, they can reduce (but not completely remove) the need for feature tests.

GUI-based testing is expensive and complex to automate because it requires many checks to confirm that a feature works as expected. For this reason, **API-based testing** is usually preferred since it is simpler and faster to automate.

System testing checks the system by simulating a **real user**, performing a sequence of user actions. However, manual system testing is usually **boring and error-prone**, so it is better to use **testing tools** that can record user actions and **automatically replay** them.

Test-driven development



Theorem 14.4.1 (Extreme Programming)

"First write the executable test, then write the code." This idea is the basis of **Test-Driven Development (TDD)**.

Pros of Test-Driven Development:

1. This systematic method makes sure that every part of the code is linked to a test, reducing the chance of having untested code.
2. Tests help you better understand what the program should do.
3. Debugging becomes easier and can be done step by step.
4. It often results in simpler, cleaner code.

Cons of Test-Driven Development:

1. It is difficult to apply TDD to **system-level testing**.
2. TDD can **discourage major changes** to the program design.
3. It can make you **focus more on writing tests** than on solving the real problem.
4. It can make you **think too much about implementation details** instead of the overall structure of the program.
5. It is hard to write tests for **invalid or "bad" data** cases.

Security Testing

The two main goals of security testing are:

1. To find **vulnerabilities** that attackers could exploit.
2. To provide **evidence** that the system is secure enough.

Finding vulnerabilities is harder than finding regular bugs because it means testing for things that the software **should not do**, which can lead to an almost unlimited number of possible tests.

Regular functional tests often do not show security issues.

Also, the software your product depends on — such as the **operating system**, **libraries**, or **databases** — may contain vulnerabilities. Some security issues may even appear when these components are combined.

Through security testing needs **specialist knowledge** and usually follows a **risk-based approach**:

1. Identify the main **security risks** for your product.
2. Create tests to show that the product can **protect itself** from those risks.

When testing security, you must **think like an attacker**, not like a normal user.

Some security tests can be automated, but others need **manual inspection** of system behavior or files.

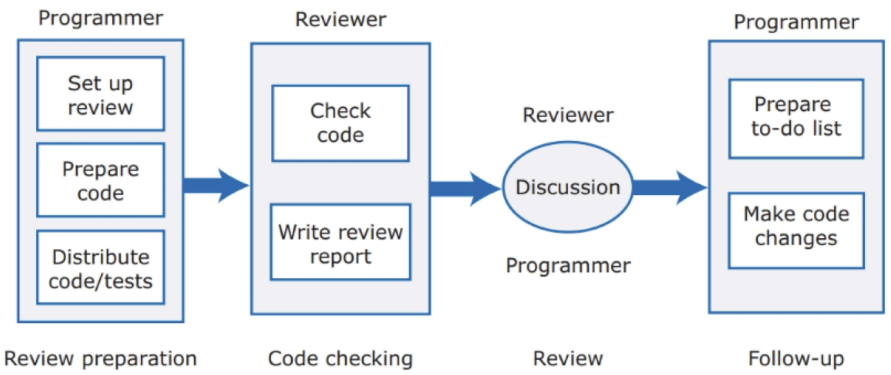
Limitations of Testing

1. You test code based on **your understanding** of what it should do. If your understanding is wrong, both the code and the tests will be wrong.
2. Testing might not cover **all parts of the code**. Even with Test-Driven Development, some parts of the program may still be missing or incomplete.
3. Testing cannot measure certain qualities of code, such as **readability**, **structure**, or **maintainability**.

Because of these limits, **code reviews** are often used.

Code reviews

Code reviews complement testing



A reviewer — sometimes from the same DevOps team — checks the code and can comment on things like readability and clarity, which testing cannot evaluate.

A single review session usually focuses on **200–400 lines of code** and often follows a **checklist**, for example:

1. Are variable and function names clear and meaningful?
2. Have all possible data errors been considered and tested?
3. Are all exceptions handled properly?
4. Are default function parameters used correctly? (Python)
5. Are data types used consistently? (Python)
6. Is the indentation correct? (Python)

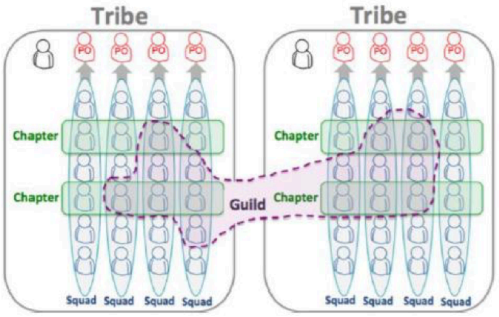
Reviews can be **triggered automatically** by commits to shared repositories and supported by **code review tools** or messaging platforms such as Slack.

Takeaway Quotes

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

“Testers don’t break software — the software is already broken.”

Code reviews (example)



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

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

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

✅ 11. DevOps

In **Project-based Software Engineering**, there used to be two separate teams:

- A **development team**, which worked on everything from design, requirements, and prototypes to testing and delivering the finished software.
- An **operations team**, which was responsible for deploying and maintaining the system, supporting users, and sometimes making small software updates.

This model had several **problems**:

- **Communication delays** between teams.
- Teams used **different tools** and had **different skills**.
- Teams often **did not understand each other's issues**.
- Fixing urgent bugs or security problems could take **days**, because the operations team didn't build the software themselves.

Three main factors led to change:

1. **Agile software engineering** made development faster, so the traditional release process became too slow.
2. **Amazon** reorganized its system into small **microservices**, where each team handled both **development** and **support** of its own service.
3. **SaaS (Software as a Service)** and **cloud computing** made it possible to release and update software continuously.

DevOps (Development + Operations)

DevOps combines **development**, **deployment**, and **support** into a **single team**.

DevOps principles:

1. **Everyone is responsible for everything:** All team members share responsibility for developing, releasing, and maintaining the software.
2. **Automate everything possible:** Most tasks related to testing, deployment, and support should be automated.
3. **Measure first, change later:** Decisions should be based on real data about system performance and operations.

Benefits of DevOps:

1. **Faster deployment:** Communication delays are reduced, so software can move from development to production much faster — from **days or weeks to just hours**.
2. **Reduced risk:** Smaller and more frequent updates make it less likely that new features will cause system failures.
3. **Faster repair:** There is no need to wait for another team to fix problems — the same team that built the software also supports it.

4. **More productive teams:** DevOps teams are generally more efficient and work better together than traditional separated teams.

Building a DevOps Team

A DevOps team brings together people with **different skills**, including:

- Software engineering
- UX (User Experience) design
- Security engineering
- Infrastructure engineering
- Customer support

The success of a DevOps team depends on a **culture of respect, teamwork, and shared learning**.

- Everyone participates in **Scrums** and team meetings.
- Members are encouraged to **share knowledge** and **learn new skills**.
- Developers also take **responsibility for their code after release** — if something breaks (even on weekends), the team fixes it quickly instead of blaming each other.

The main focus of DevOps teams is to **solve problems fast** and **keep systems running smoothly**.

Code Management

During software development, thousands of lines of code are written, and many automated tests are created. These are organized into hundreds of files and use dozens of external libraries and tools. Because of this complexity, it is impossible to manually keep track of all changes made to the software — this requires **automated support**.

A **code management system** is software that helps manage a constantly changing codebase. It ensures that:

- changes from different developers do not interfere with each other,
- different product versions can be created easily, and
- the process of building and testing the product is simplified.

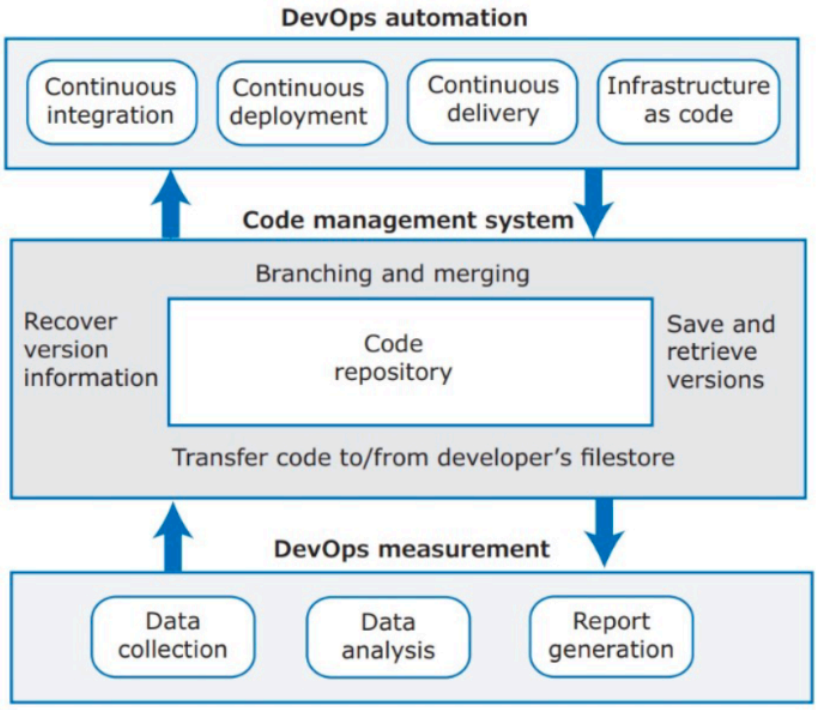


Figure 10.1: DevOps system structure

Main Features of Code Management Systems

1. **Version and release identification:** Each version of a file is uniquely identified when saved in the system. Old versions are never overwritten. Files can be retrieved using their version ID or other details.
2. **Change history recording:** When a developer changes a file, they must add a note explaining *why* the change was made. This helps others understand the reason behind updates.
3. **Independent development:** Multiple developers can work on the same file at the same time. When each submits their work, a new version is created — this prevents one developer's changes from overwriting another's.
4. **Project support:** Developers can download project files, work on them locally, and then upload their changes.
    - Multiple **branches** can be created for parallel work.
    - Changes from different branches can later be **merged**.
    - The entire set of files for a project can be checked out together.
5. **Storage management:** The system saves storage space by keeping only the differences between versions of a file instead of storing full copies.

(This is less critical today since storage is cheaper.)

Centralized vs Decentralized Systems

In the past, **centralized systems** were used to manage projects.
However, **decentralized systems** (like **Git**) have become more popular because they offer important advantages:

1. **Resilience:** Everyone has a full copy of the repository and can work offline. If the shared repository is damaged or attacked, the project can be restored from any developer's clone.
2. **Speed:** Saving changes (committing) is fast because it can be done locally without sending data over the network.
3. **Flexibility:** Developers can safely experiment and test changes in their local copy before sharing them with others.

Example: Git

Git is a well-known **decentralized version control system**. Each developer has:

- a **local clone** (a full copy of the shared repository) on their own computer, and
- access to a **shared repository** stored on a company server or a cloud platform like **GitHub** or **GitLab**.

Figure 10.2 shows some Git commands and how Git branches work.

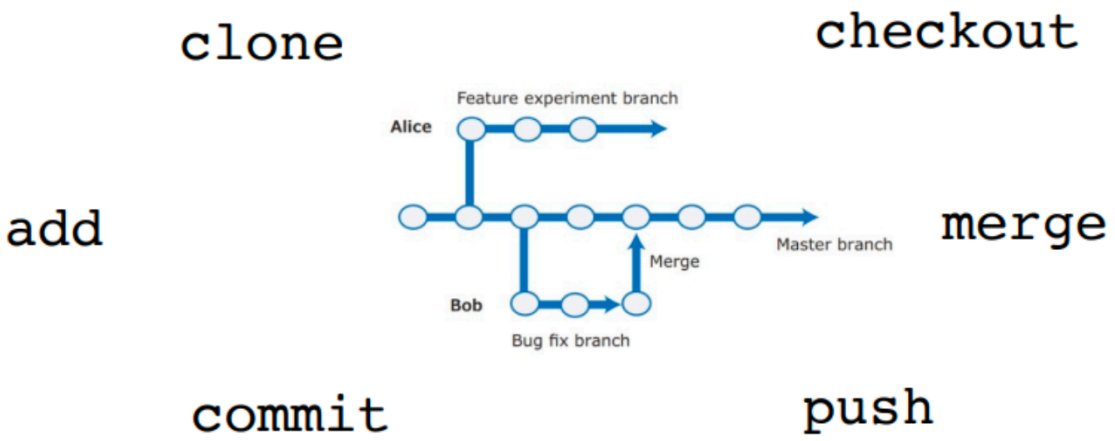


Figure 10.2: Git branch example

When talking about open-source software development with Git, there is usually a group of people who decide which changes should be added to the project. GitHub uses a feature called **Webhooks** to automatically trigger **DevOps tools** whenever updates are made to the project repository.

DevOps Automation

Everything that can be automated should be automated — this is one of the main DevOps principles. This section explains how automation is used in DevOps through four main ideas:

- **Continuous integration:** Every time a developer commits a change to the project's main branch, an executable version of the system is automatically built and tested.
- **Continuous delivery:** After building the new version, the executable software is tested in an environment that simulates the real operating conditions of the product.
- **Continuous deployment:** A new release of the system is automatically made available to users every time a change is made to the main branch.
- **Infrastructure as code:** Models describing the system's infrastructure (network, servers, routers, etc.) are written in a machine-readable format. Configuration tools then use these models to create the platform where the software runs. This includes installing required software like compilers, libraries, and databases.

Continuous Integration

Continuous integration (CI) is important because when integration happens rarely, it becomes harder to find and fix problems, which slows down development.

System integration (or system building) involves more than just compiling code. It also includes:

1. Installing database software and creating the correct database structure.
2. Loading test data into the database.
3. Linking compiled code with libraries and other components.
4. Making sure external services used by the system are running.
5. Moving configuration files to the right locations and removing old ones.

6. Running system tests to check if the integration was successful.

As shown in Figure 10.4, every time a change is pushed to the shared code repository, a new integrated version of the system is built and tested. When the push happens, the repository sends a message to the integration server, which automatically builds the new version of the product.

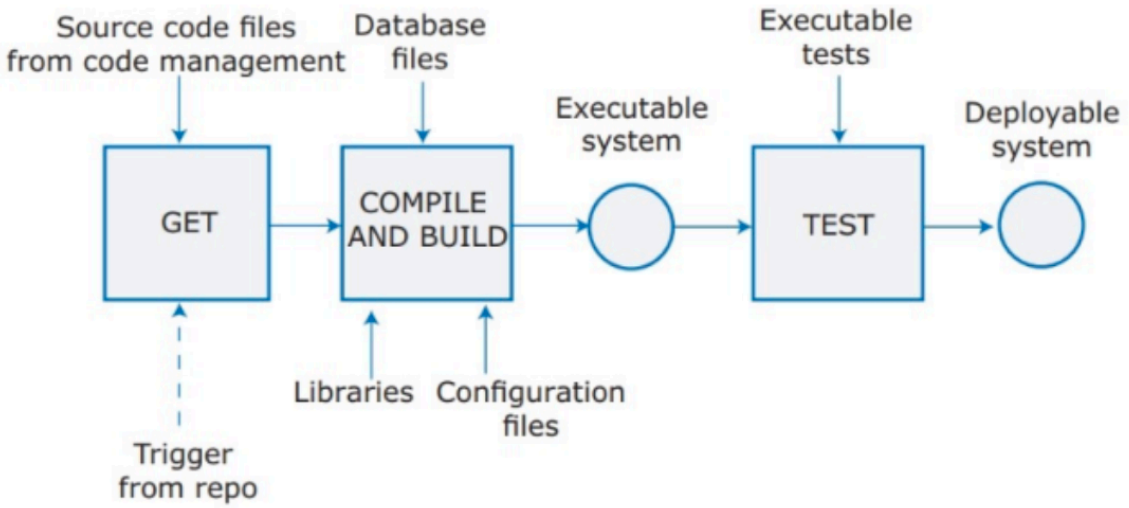


Figure 10.3: Continuous integration pipeline

It is a good practice to use the **"integrate twice"** approach for system integration.

This means first integrating the code **locally on the developer's machine**, and then doing a **second integration** by pushing the code to the project repository, which triggers the integration server (as shown in Figure 10.4).

This method helps prevent broken or faulty builds from being added to the shared project repository.

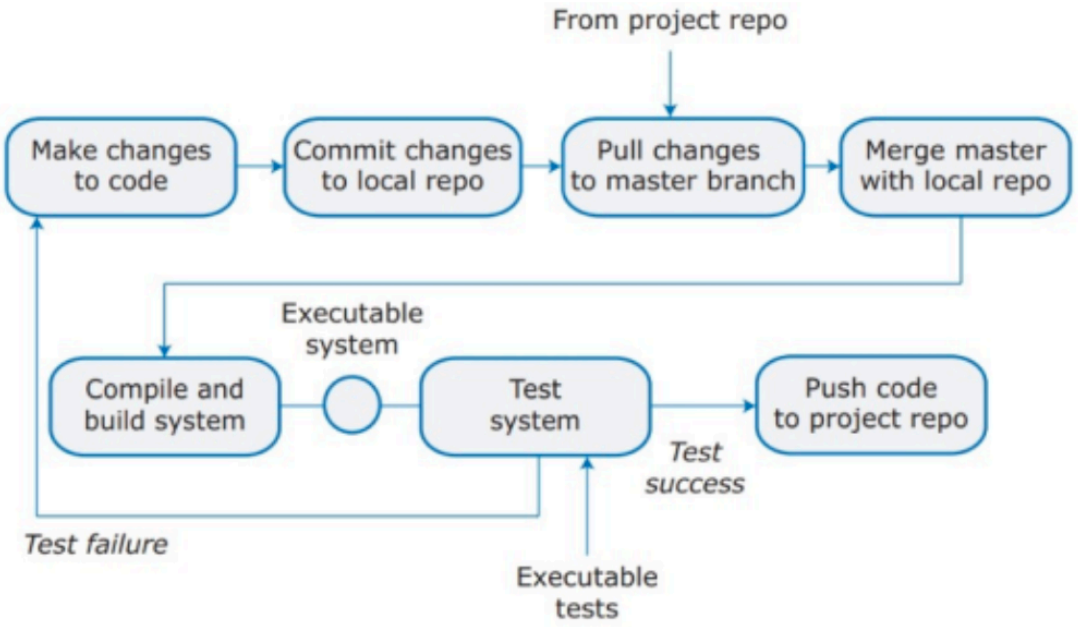


Figure 10.4: Integrate twice pipeline

Advantages of Continuous Integration

The main benefits of continuous integration are:

- **Faster bug detection and fixing:** When you make a small change and a test fails, the issue is almost always caused by the new code you just pushed.
- **Always having a working system:** The whole team can use the system at any time to test new ideas or show demos to managers and customers.
- **Better quality culture:** Developers take more care before pushing changes, because no one wants to be the person who breaks the build for everyone.

Running all tests on the entire codebase every time a commit is made would be too heavy and slow. That's why integration tools only rebuild and retest the parts of the code that have changed or that depend on changed files. Modification timestamps help check which dependencies have been updated.

Continuous Delivery and Deployment

However, when deploying this version into the **real production environment**, problems can occur.

This is because the production environment might be different from the development one — for example, the production server might have a different file system, access permissions, or installed applications.

Continuous delivery ensures that the new system version is ready to be released to users.

It does this by running feature tests in a production-like environment (to make sure the environment itself doesn't cause errors), system tests, and load tests (to see how the system performs when more users are added).

Containers are often used to easily create a copy of the production environment for testing.

As shown in Figure 10.5, the delivery process includes:

1. Setting up a staged test environment after initial integration testing.
2. Running system acceptance tests (functionality, load, and performance).
3. Transferring the software and data to the production servers.
4. Switching to the new version of the system.
5. Restarting the process — while making sure that users still connected to the old version are properly handled during the switch.

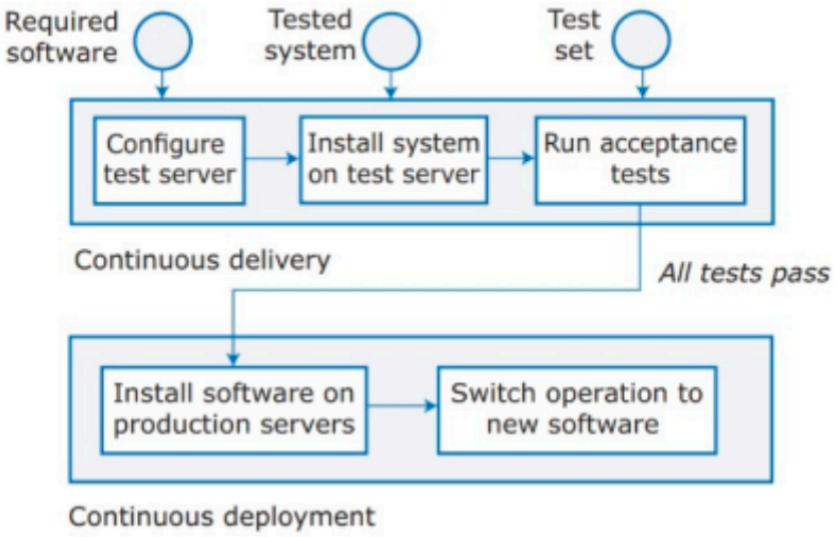


Figure 10.5: Continuous delivery and continuous deployment schema

Benefits of Continuous Deployment

The main advantages of continuous deployment are:

- **Lower costs:** The entire deployment process is fully automated, which reduces manual work and saves time.
- **Faster problem solving:** If a problem appears, it usually affects only a small part of the system, and it's easier to identify where it came from.
- **Quicker customer feedback:** New features can be released as soon as they are ready, and feedback from users can help improve the product.
- **A/B testing:** If there are many users and several servers, you can deploy a new version only on some servers. A load balancer can then direct some users to the new version so you can compare how it performs and how users react.

However, it's usually not a good idea to deploy every single small change immediately. Some changes may be incomplete or part of a larger feature that isn't ready yet. Releasing them early might also reveal information to competitors.

Another reason to avoid constant small releases is that frequent updates — especially those that change the user interface — can annoy customers.

Developers may also choose to align releases with business events, such as the start of the academic year for education software.

There are many **Continuous Integration (CI)** tools, such as **Jenkins** and **Travis**, that also support **Continuous Delivery and Deployment**.

These tools can work together with **infrastructure configuration management tools** to automate software deployment.

For cloud-based applications, it's often easier to use **containers** together with CI tools instead of relying on complex infrastructure configuration systems.

Infrastructure as Code

Manually managing an infrastructure with many servers (dozens or hundreds) is expensive and often leads to mistakes.

Each physical or virtual server can have different configurations and software versions. When new software updates are released, some servers may need to be updated while others cannot, because they depend on older versions.

Keeping track of what software is installed on each server is difficult, and in many cases, emergency updates are not properly documented.

To solve this, we can **automate infrastructure management** using a **machine-readable model**.

Configuration Management (CM) tools like **Puppet, Chef, and Ansible** can automatically install software and services on servers according to this model.

This model describes the entire infrastructure in data form, which is why it's called **Infrastructure as Code (IaC)**.

When a change is needed, you simply update the infrastructure definition, and the CM tool automatically applies those changes across all servers.

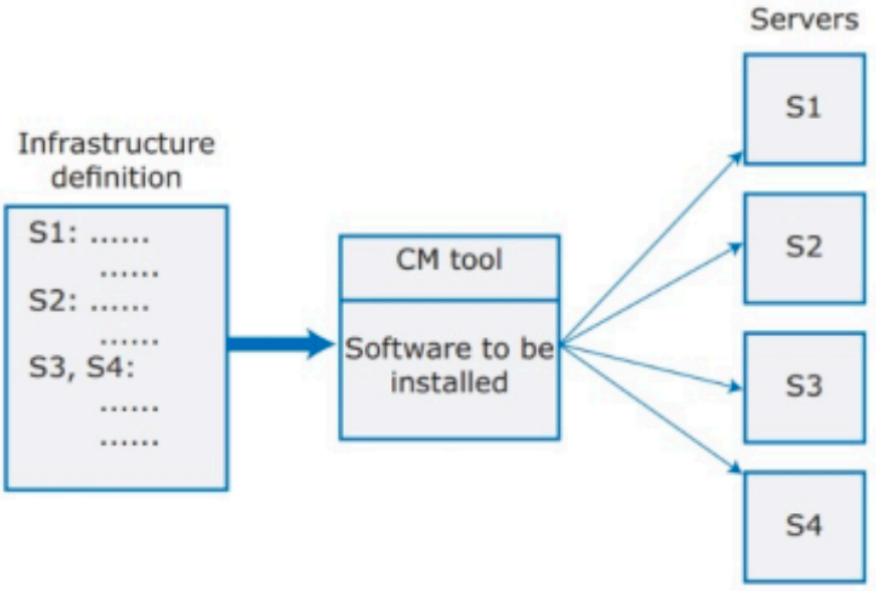


Figure 10.6: Infrastructure as code schema

Advantages of Infrastructure as Code:

- **Visibility:** The whole infrastructure is written as a model that everyone on the DevOps team can read, understand, and review.
- **Reproducibility:** The installation process always happens in the same order, creating identical environments every time. You don't have to rely on people remembering steps — computers ensure consistency.
- **Reliability:** Automation removes common human errors that occur when administrators manually update several servers.
- **Recovery:** The infrastructure model can be versioned and stored in a code management system. If something goes wrong, you can easily revert to an

older version and restore a stable environment.

DevOps Measurement

To keep improving your DevOps process and achieve faster deployment of better-quality software, it's important to **measure and analyze data** about both the product and the process.

There are several types of measurements:

- **Process measurements:** Collect and analyze data about development, testing, and deployment processes.
- **Service measurements:** Collect and analyze data about the software's performance, reliability, and customer satisfaction.
- **Usage measurements:** Collect and analyze data on how customers actually use your product, instead of just asking for feedback through pop-ups. This helps identify problems and improve the product.
- **Business success measurements:** Collect and analyze data on how your product helps the company succeed. It can be difficult to know if improvements come from DevOps practices or from other factors like better management.

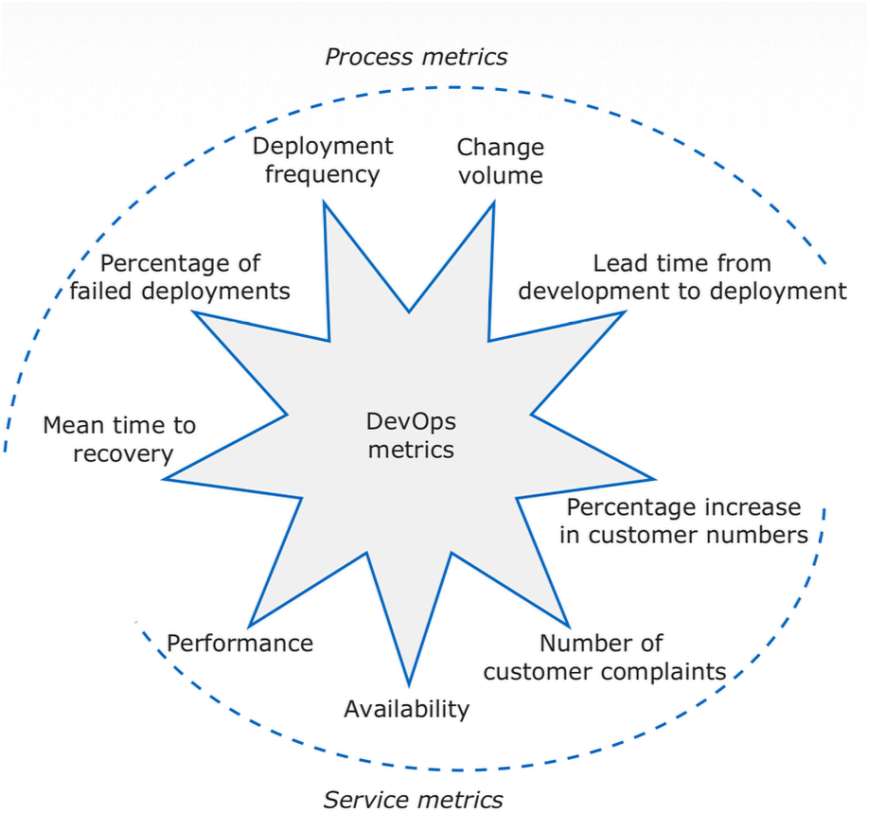


Figure 15.2: DevOps Metrics

Measuring software and its development can be complex. Developers need to choose **the right metrics** that provide useful information and find reliable ways to collect and analyze that data.

Some values, such as **customer satisfaction**, cannot be measured directly and must be estimated from other data (for example, the number of users who keep returning).

Software measurement should be **as automated as possible**. Developers should build tools into the software to collect data automatically and use monitoring systems to track performance and availability.

To collect data automatically, teams can:

- Use **continuous integration tools** (like Jenkins) to track deployments, test results, and build success.
- Use **cloud monitoring services** (like Amazon CloudWatch) to collect information on system performance and availability.
- Gather **customer feedback and reports** from issue management systems.

- Add **instrumentation** to the software to collect data about its performance and user behavior. This is usually done through **log files**, which record as many events as possible. Log analysis tools can then extract valuable insights about how the software is used.

✅ 12. Security & Privacy

Software security is a very important concern for both product developers and users, because cyberattacks can cause serious damage and financial loss for both sides.

Figure 7.1 shows the main types of security threats, although some attacks can mix different types of threats. For example, a **ransomware attack** that harms the integrity of system data also affects the **availability** of the system.

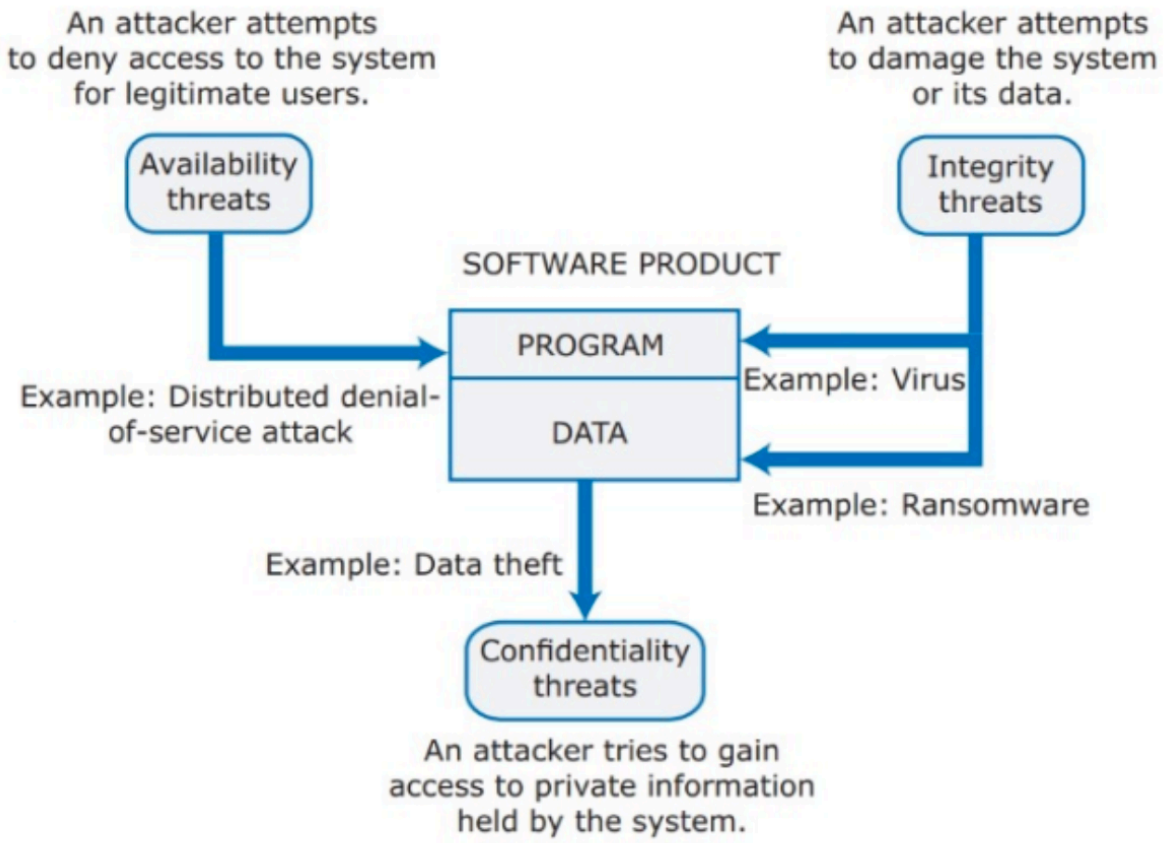


Figure 7.1: Main types of security threats

Security is an issue that affects the whole system. Application software depends on many layers — such as the operating system, web server, language runtime, database, frameworks, and tools. Attacks can target any part of this system stack, starting from the network level.

Some management activities that help keep a system secure are:

- **Authentication and authorization:** making sure all users have strong passwords and correct access permissions.
- **System infrastructure management:** keeping system software correctly configured and applying security updates quickly to fix vulnerabilities.
- **Monitoring attacks:** regularly checking for possible attacks to detect them early and reduce their impact.
- **Backup policies:** keeping safe copies of programs and data so they can be restored after an attack.

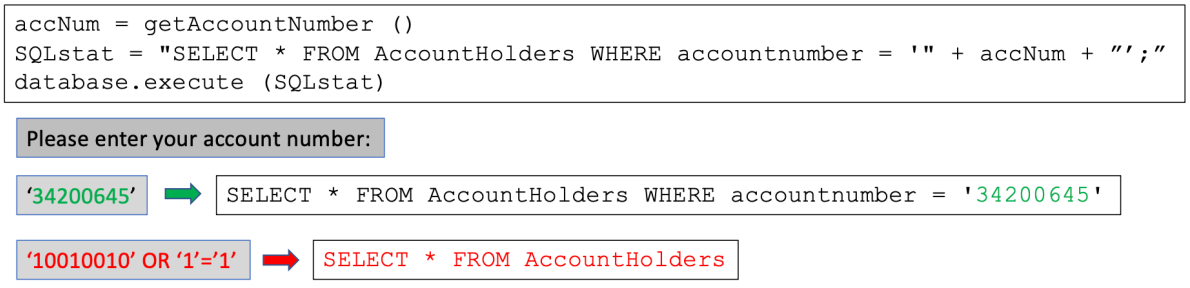
Attacks and defenses

1. Injection attacks

An **injection attack** happens when a malicious user enters harmful code or database commands into a valid input field to damage the system.

A common type is the **buffer overflow attack**, where an attacker sends input that overwrites memory and adds executable code. If the function's return address is changed, control can go to the malicious code. This often affects programs written in C/C++, which do not check if array assignments are within bounds.

Another common attack is **SQL injection**, where attackers insert harmful SQL commands through user input fields. A simple defense is to **check and validate all user input** before using it.



2. Session hijacking attacks

Many web applications use **sessions**, which are time periods during which a user stays logged in. This is done using **session cookies (tokens)** that are sent between the server and client in each request.

Session hijacking happens when an attacker steals a valid session cookie to pretend to be the real user. Attackers can steal cookies through **cross-site scripting (XSS)** or by monitoring unprotected network traffic, such as on public Wi-Fi.

Active vs. Passive Session Hijacking

- **Active:** The attacker takes control of a user's session and performs actions on the server as if they were the legitimate user (for example, making purchases or changing account settings).
- **Passive:** The attacker does not interact with the system directly but instead listens to and monitors the communication between the client and the server to collect sensitive information such as passwords or credit card numbers.

Defenses include:

- Encrypting traffic between client and server (for example, using **HTTPS**).
- Using **multifactor authentication** to confirm important actions.
- Setting **short session timeouts** to reduce the window of attack.

3. Cross-site scripting (XSS) attacks

A **cross-site scripting (XSS) attack** is another type of injection attack where an attacker adds malicious JavaScript code to a web page sent from the server to the client.

The code runs in the victim's browser when the page loads, and it can steal user data or redirect users to another website. Stolen cookies can also be used to perform **session hijacking attacks**.

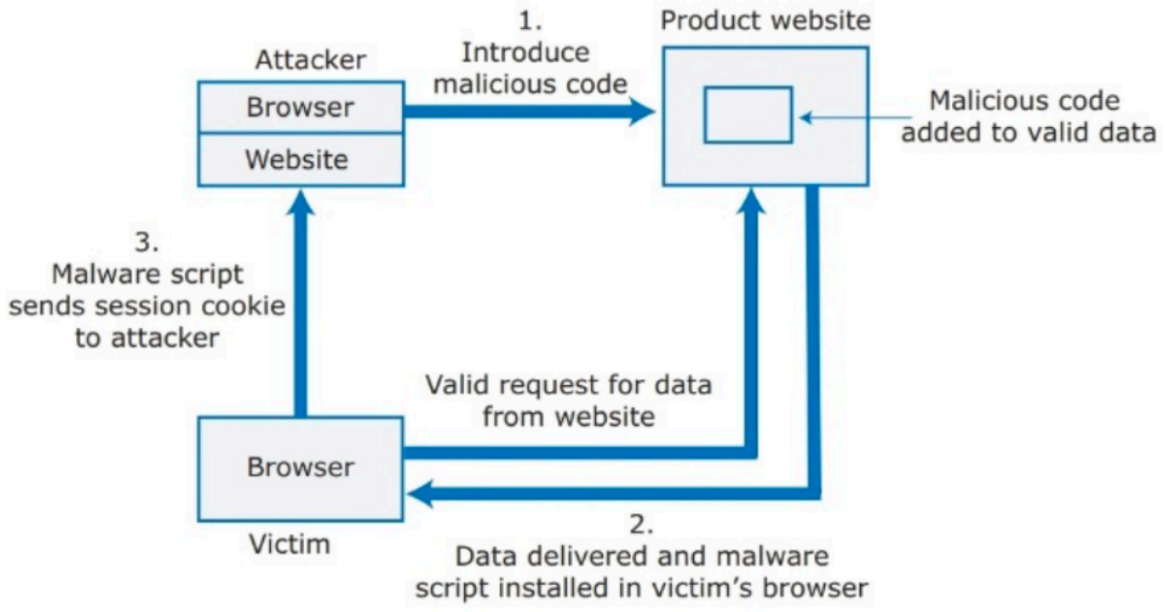


Figure 7.2: Cross-site scripting attack

Some defenses against this type of injection attack include validating form inputs, checking all input data before adding it to a web page, and using the HTML "encode" command so that information added to the page cannot be executed as code.

4. Denial-of-Service (DoS) attacks

Denial-of-Service (DoS) attacks aim to make a system unavailable for normal users. Attackers usually use many infected computers (called botnets) that send huge numbers of requests to a web application at the same time. This overloads the system, making it crash or slow down. These attacks are often done to harm the service provider or to demand ransom money.

There is special software that can detect and block suspicious traffic to stop these attacks. Other defenses include **temporary user lockouts** (for example, locking a user account after too many failed login attempts) and **IP address tracking**, which helps to identify unusual login attempts from unknown locations.

5. Brute force attacks

In a **brute force attack**, the attacker knows part of the login information (for example, the username) and tries to guess the password by testing many combinations of letters and numbers.

Defenses include requiring users to create **long and complex passwords** that are not real words and not found in a dictionary. Another layer of protection is **two-factor authentication (2FA)**, where the user must confirm their identity with a code sent to their phone or email.

Authentication and Authorization

Authentication

The goal of **authentication** is to make sure that users are really who they claim to be. There are several ways to do this:

- **Knowledge-based authentication:** The user provides secret information (like a password or security question) created during registration. This method has weaknesses, such as:

– weak or reused passwords,

– phishing attacks (fake websites that steal login data),

– forgotten passwords, which require a recovery process that could be insecure.

To make this method safer, users should be forced to create **strong passwords** and use **personal security questions** for recovery.

- **Possession-based authentication:** The user has a **physical device** connected to the system, such as a phone that receives a code or a device that generates one-time passwords.
- **Attribute-based authentication:** The system uses **biometric features** like fingerprints or facial recognition to verify identity.

When a system stores sensitive user data, **multi-factor authentication (MFA)** — such as using a password plus a mobile code — is considered best practice, since it adds extra security.

Usually, developers don't build authentication systems from scratch. Instead, they use **libraries and toolkits** such as **OAuth**. Even when using these, some programming work is still needed to make authentication secure and reliable.

For better security and less maintenance, many systems use **federated identity services**, meaning they use external providers like **Google or Facebook** for authentication. These services are already proven to be secure and reduce the need to store passwords locally. They also give the product provider access to some user information (if the user agrees).

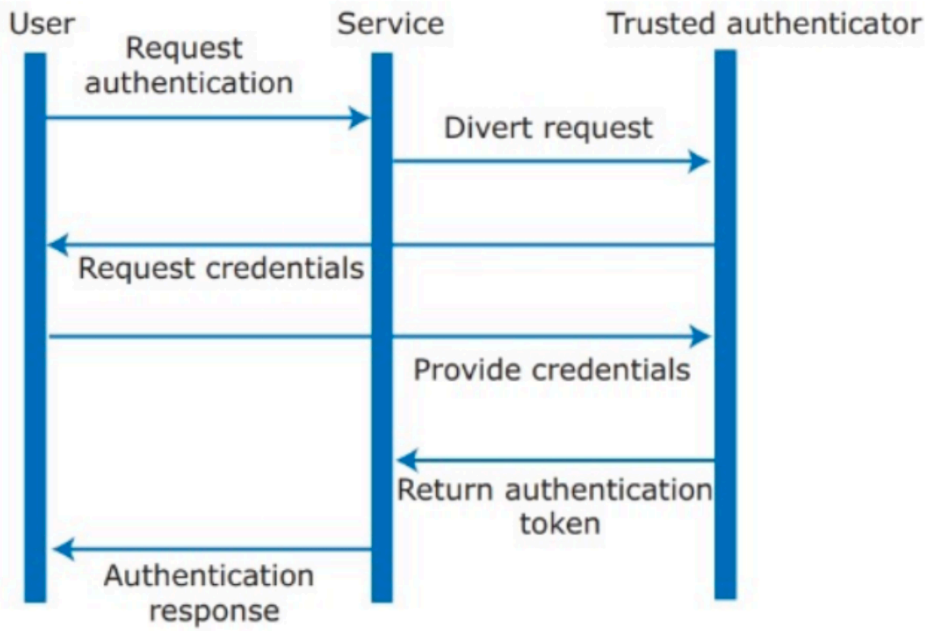


Figure 7.3: Federated identity sequence diagram

Mobile device authentication works a bit differently because typing passwords on mobile keyboards is uncomfortable. Instead of a written password, a token (a digital key) can be installed on the device to confirm identity. However, this has a weakness — if the phone is lost or stolen, someone else could use it to access the system. A safer option is to use digital certificates issued by trusted providers to identify each user.

Authorization

Authentication ≠ Authorization

ensure that user is who she claims to be

control that user can access resources

Authorization

While **authentication** checks who the user is, **authorization** ensures the user is allowed to access certain resources. Access control is especially important in systems with many users, where rules must protect personal and sensitive data (to avoid legal issues if data is leaked).

A common method for managing access is using **Access Control Lists (ACLs)**. In this approach:

- Users are divided into **groups**, which makes ACLs easier to manage.
- Different **groups** have different permissions for various resources.
- Group **hierarchies** can give rights to subgroups or individual users.

An example of an Access Control List is shown in Figure 7.4.

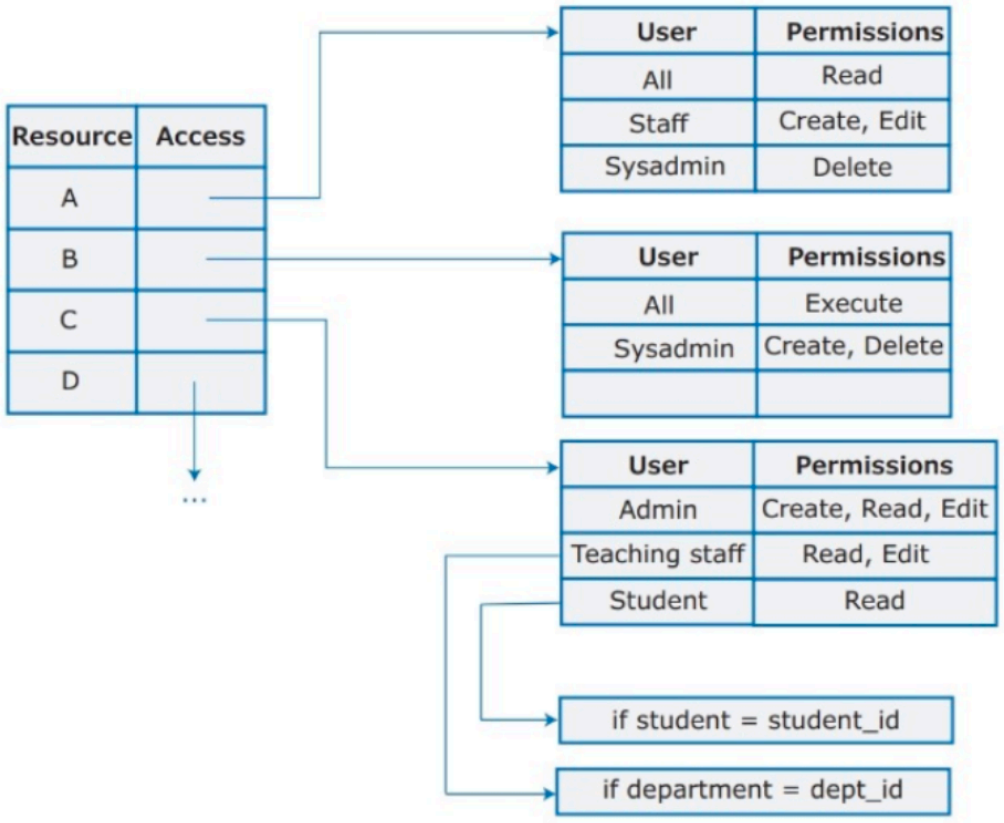
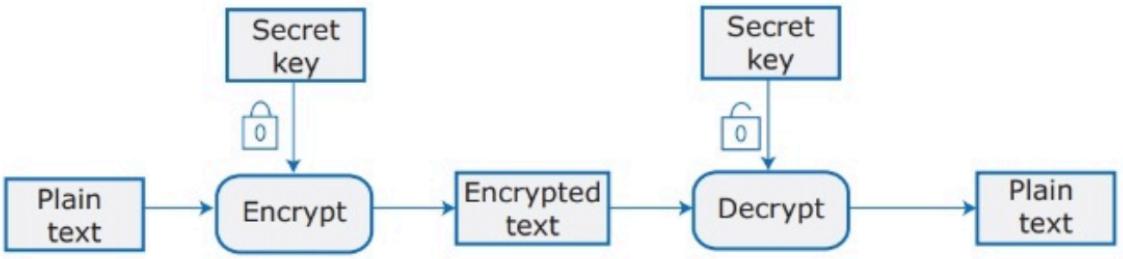


Figure 7.4: Example of Access Control Lists

Encryption

Encryption means turning readable data (plain text) into unreadable data by using a mathematical algorithm. The text is encrypted with a secret key before

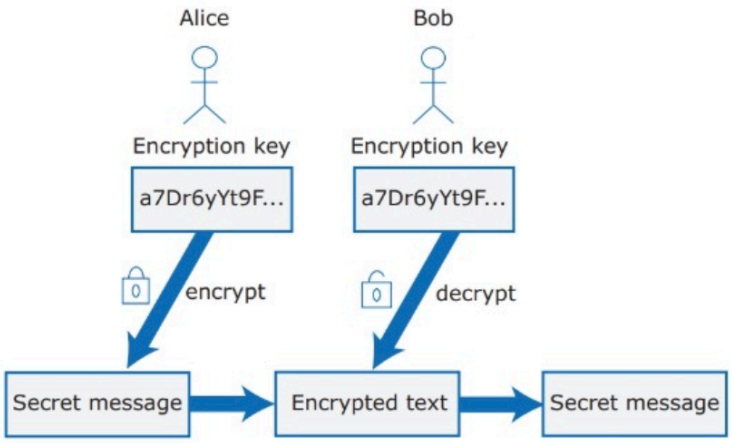
it is sent through an unsafe channel, and then decrypted with the same or another key when it reaches its destination.



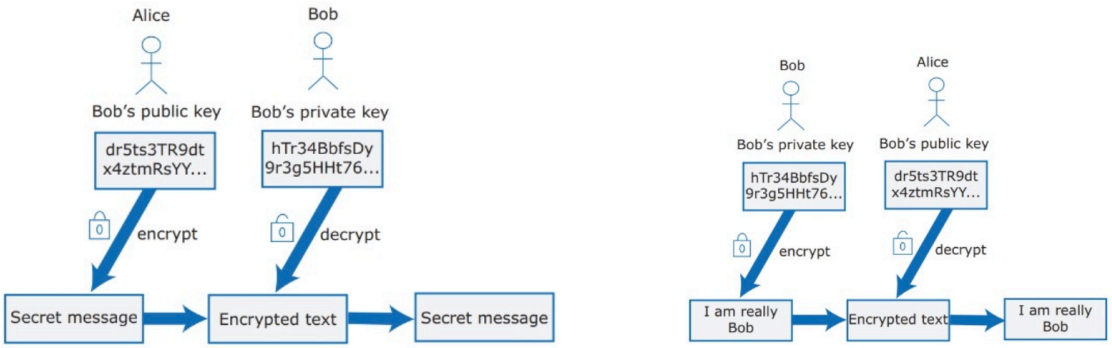
Modern encryption methods are considered “almost impossible to break” with today’s technology. However, new technology (like future quantum computers) could make it possible to break some encryption methods in the future.

There are two main types of encryption:

- **Symmetric encryption**: This is the older method, used for many centuries and still used today. It uses the **same key** for both encryption and decryption. The weakness of this method is that both parties need to share the same key safely, which can be risky.



- **Asymmetric encryption**: This method uses **two different keys** — one for encryption and another for decryption. It is more secure but requires more computer power to generate and manage the keys. Usually, asymmetric encryption is used at the start of communication to securely exchange a symmetric key, which is then used for the rest of the conversation.



Transport Layer Security (TLS) is a widely used security protocol that protects privacy and data during communication over the Internet. One of its main uses is encrypting communication between web browsers and web servers. When TLS is used with HTTP, it becomes **HTTPS**, which is now the standard for secure websites.

TLS also helps confirm the **identity** of web servers by using **digital certificates** — files sent from the server to the client. These certificates are issued by trusted organizations called **Certificate Authorities (CAs)**. A typical example of how this client-server "handshake" works is shown in Figure 7.5.

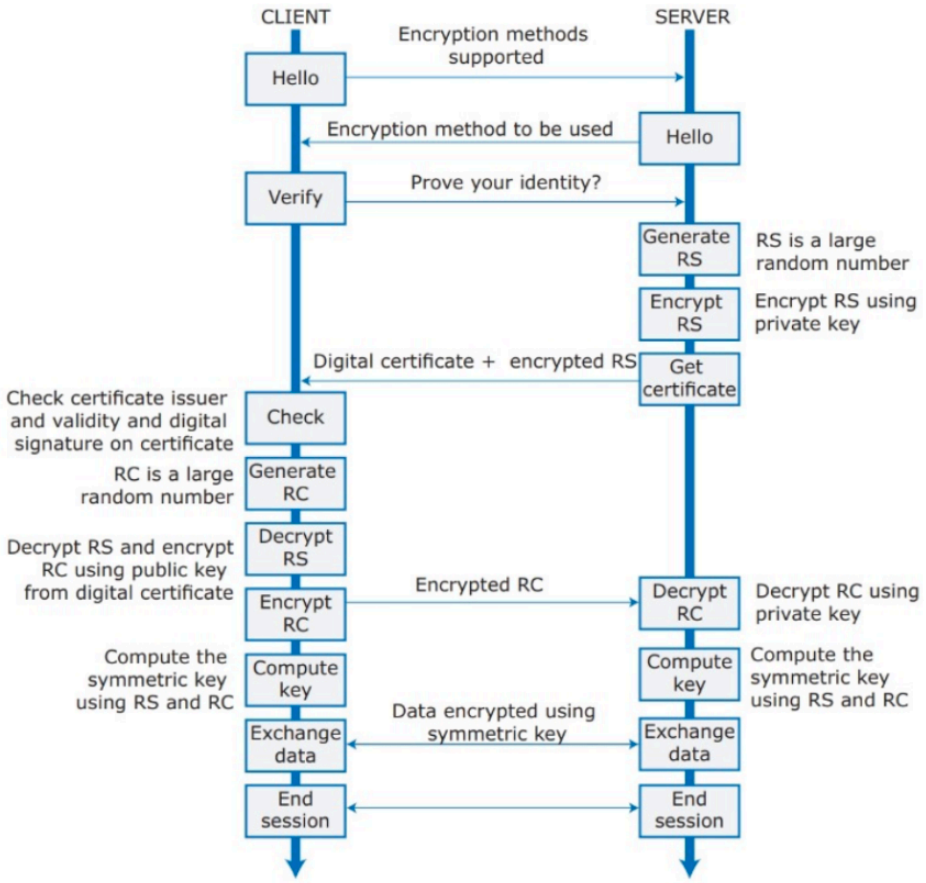


Figure 7.5: TLS client-server interaction to generate symmetric key for exchanging data

It's important to know **when user data should be encrypted**, and for that we can divide data into three types:

- **Data in transit**: This is data being sent between systems or users. It should **always be encrypted** to protect communication privacy.
- **Data at rest**: This is data stored on a disk or database. It should **always be encrypted** too, so that if attackers get access, they can't read the data in plain text.
- **Data in use**: This is data that the system is currently processing. Encrypting and decrypting this data can slow down the system's performance.

Encryption can happen at **four different levels** in a system:

- **Application level**: The application itself decides which data to encrypt and decrypts it just before use. This can cause slower performance and requires secure key management.

- **Database level**: The database system (DBMS) can encrypt the whole database when it's closed and decrypt it when reopened. It can also encrypt specific tables or columns.
- **File level**: The operating system can encrypt individual files when they are closed and decrypt them when opened again.
- **Media level**: The operating system can encrypt entire disks when they are unmounted and decrypt them when remounted. This is especially useful if a laptop or hard drive is lost or stolen.

All these encryption methods need **keys** to encrypt and decrypt the data.

However, data protection laws often require that copies of data be kept for years. This means that **if encryption keys are lost**, the encrypted data cannot be recovered.

To prevent this, **keys must be changed regularly**, and databases need to keep several versions of keys with timestamps.

To simplify this process, organizations use **Key Management Systems (KMS)**. A KMS securely generates, stores, and provides keys only to authorized users. Figure 7.6 shows how KMS are used in a system.

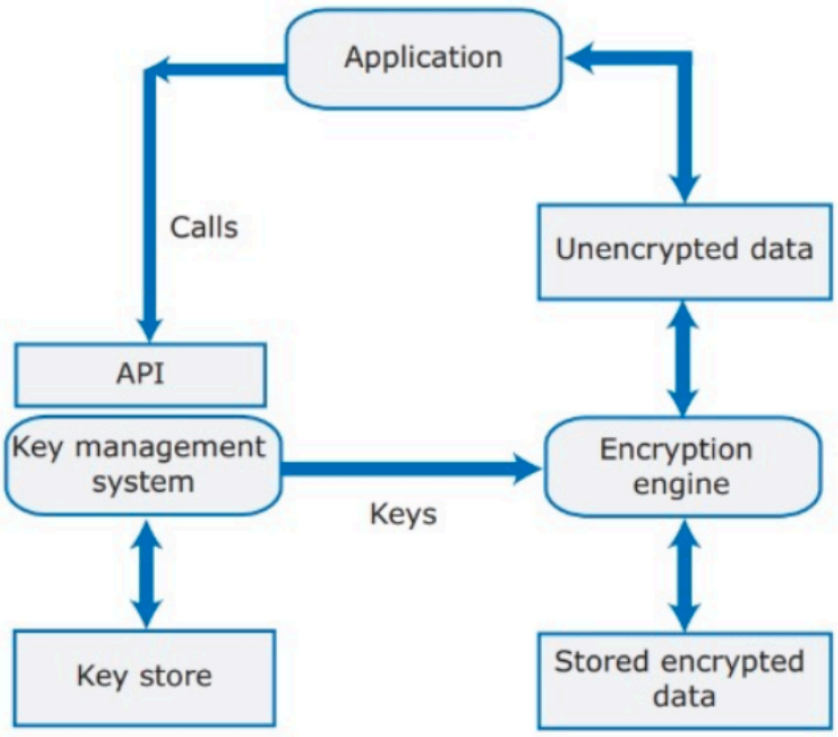


Figure 7.6: KMS usage schema

Privacy

Privacy is a social idea that deals with how personal information is collected, shared, and properly used by a third party.

Figure 7.7: Data protection laws

Some **main data protection principles** are:

- **Awareness and Control:** Users must know what data your product collects and must have control over their personal information.
- **Purpose:** You must clearly tell users why you are collecting their data and not use it for any other reason.
- **Consent:** You must always get a user's permission before sharing their data with others.
- **Data lifetime:** You must not keep data longer than needed. If a user deletes their account, you must also delete their related personal data.
- **Secure storage:** You must store data safely so that it cannot be changed or accessed by unauthorized people.
- **Discovery and error correction:** Users must be able to see what personal data you store and correct any mistakes.
- **Location:** You must not store data in countries with weak data protection laws unless there is an agreement to follow stronger rules.

There are also **business reasons** to care about privacy:

- If your system does not follow privacy regulations, your company may face legal problems or be unable to sell the product.
- If you sell to other businesses, they may require strong privacy protections to avoid legal or user issues.

- If private data leaks or is misused, it can seriously harm your company's reputation.

The amount of information a software product collects depends on its **functionality** and **business model**.

Here are some extra **tips for data privacy:**

- Do not collect personal information that is not necessary.
- Create a clear privacy policy explaining how you collect, store, and use personal or sensitive data.
- Be transparent if you use users' data for advertising or to offer services paid for by other companies.
- If your product has social features where users can share information, make sure users understand how to control what they share.

✅ 13. Security & Microservices

Challenges of Securing Microservices

This subsection describes the general attack and security issues related to microservices. The explanation is given as key points:

- ***A larger attack surface increases risk:***

Microservices constantly exchange data through remote calls. This creates many possible entry points for attackers, *increasing the attack surface*. The application's security depends on the weakest service: one weak entry point can compromise the entire system.

- ***Distributed security checks reduce performance:***

Each microservice must perform its own security checks. This often requires calling a remote token or identity service many times. These repeated checks slow down the system. One workaround is to trust the internal network and skip checks, but the real, modern solution is to use *zero-trust networking*, even though it may impact performance.

- ***Automating trust setup between microservices is necessary:***

Microservices must communicate over secure channels. If they use *certificates*, each service needs its own certificate and private key to prove its

identity to other services. The receiving service must be able to verify these certificates. This means we need a process to *establish trust*, rotate or revoke certificates, and keep everything up to date. For systems with many microservices, *automation* is required.

- ***Tracing requests across many microservices is difficult:***

Logs can be collected to create system metrics (e.g., invalid access attempts per hour) and send alerts. *Traces* allow us to follow a request from when it enters the system to when it leaves. In microservices architectures, a single request may pass through many services, making it harder to connect logs and traces together.

- ***Containers make handling credentials and policies more complex:***

Containers are immutable, meaning they do not change after they start. But each service still needs an updated list of allowed clients and access-control rules. Since the container cannot change itself after starting, the only way to update these lists is to store credentials inside the container's file system and *inject them at startup.*

- ***Distributed systems make sharing user context harder:***

When a request enters the system, it has useful information attached to it (for example, whether the user is premium or not). Microservices need to trust this information when it is passed from one service to another. Attackers may try to change this user information. A common solution is using *JSON Web Tokens* (JWT), which securely carry user context through the system.

- ***Security responsibilities spread across different teams:***

Different teams may use different technologies, tools, and security practices when building their services. This means security tasks are shared among many teams. Many organizations use a hybrid model: a central security team plus security specialists embedded inside development teams.

Smells and Refactoring:

A **security smell** is a common architectural choice that may cause security problems in microservice-based applications. Once the main microservice principles are defined, we need a way to detect these **security smells** that affect **security properties** and fix them through **refactoring**.

This section explains, based on a multivocal review (using research papers and technical websites), the most recognized security smells in microservices and how to refactor them.

The **security properties** considered are:

- **Confidentiality:** Ensuring that data can only be accessed by authorized users.
- **Integrity:** Ensuring that data and programs cannot be changed by unauthorized people.
- **Authenticity:** Ensuring that the identity of a user or resource is real and correctly verified.

Since some ***smells*** affect more than one security property, the list below shows each **smell**, which **security properties** it harms, and how to **refactor** it.

- **Insufficient Access Control:**

Some services do not enforce proper access control. This can cause a "***confused deputy***" problem, where an attacker gains access to data they should not see. This threatens the **confidentiality** of data and business functions. Client permissions must be checked when the request is made, but this should not add too much delay or create too many calls to a central service.

A common refactoring is to **use OAuth 2.0**, a token-based system that allows secure delegated access control.

- **Publicly Accessible Microservices:**

Some microservices are directly reachable by external clients. Each service must then check authentication and authorization for every request, which increases maintenance costs. It also increases exposure of credentials and can lead to **confidentiality** risks.

A common refactoring is to **use an API gateway**, placed behind a firewall. It can handle authentication, authorization, rate limits, and message validation before forwarding requests to internal microservices.

- **Unnecessary Privileges to Microservices:**

Sometimes microservices are given more access rights, permissions, or capabilities than they actually need. This often happens because developers reuse code patterns without adjusting them. Giving too many privileges increases the attack surface and may harm **confidentiality and integrity**.

The refactoring is to **follow the Least Privilege Principle**, giving each microservice only the minimum rights needed to do its job.

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

- **"Home-Made" Crypto Code:**

Using your own custom cryptography code can create serious **confidentiality, integrity, and authenticity** problems. It can even be more dangerous than not encrypting at all because it gives a false sense of security.

The solution is to **use well-tested and widely reviewed encryption libraries.** Avoid experimental or unproven encryption algorithms.

- **Non-Encrypted Data Exposure:**

A microservice application might accidentally expose sensitive data (for example, data stored without encryption or stored in a weak system). This allows attackers to read or change data, including credentials. This harms **confidentiality, integrity, and authenticity.**

The refactoring is to **encrypt all sensitive data at rest.** Sensitive data should always stay encrypted and be decrypted only when needed. Most databases support automatic encryption, but encryption can also be done at the application level, OS level, or cache level.

However, encryption uses system resources, so teams should identify the most critical data and encrypt only what is necessary.

- **Hardcoded Secrets:**

Secrets such as API keys, client secrets, and passwords should never be stored directly in environment variables or inside the code. They can be accidentally exposed, for example when error logs are sent to external logging systems. This affects **confidentiality, authenticity, and integrity.**

The solution is to **encrypt secrets at rest** when stored, avoid keeping them inside application code or repositories, and not rely on environment variables for storing sensitive secrets.

- **Non-Secured Service-To-Service Communications:**

If microservices talk to each other without secure channels, data can be stolen or altered through attacks like man-in-the-middle or eavesdropping. This threatens **confidentiality, integrity, and authenticity.**

A common solution is to use **mutual TLS - Transport Layer Security**, which encrypts data in transit, ensures data integrity, and allows both microservices to verify each other's identity.

- **Unauthenticated Traffic:**

Microservices must authenticate each other, especially when passing user information. If traffic is not authenticated, microservices are open to attacks such as data tampering, denial of service, or privilege escalation. This affects **authenticity.**

TLS supports mutual authentication. Another option is **OpenID Connect**, which uses JWTs containing verified user information. Microservices check these tokens with the authorization server.

- **Multiple User Authentication:**

Developers may allow users to authenticate from different points. Each entry point becomes a new possible attack path, increasing **authenticity** risks.

The recommended **solution is Single Sign-On (SSO)**, where users authenticate through one main entry point. This makes logging and auditing easier.

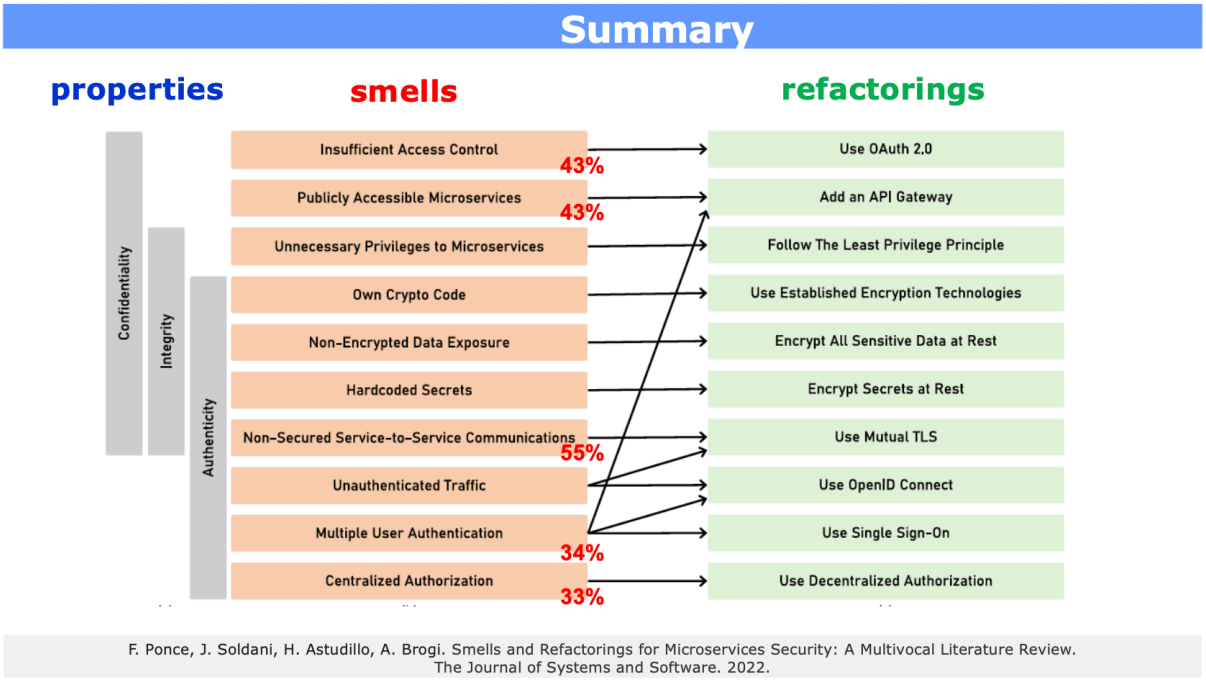
SSO can be implemented with an **API gateway** and **OpenID Connect** (to share user context).

- **Centralized Authorization:**

Authorization can be done at the system's edge (API gateway) or by each microservice. If only the gateway handles authorization, it becomes a

performance bottleneck. This can also lead to the “confused deputy problem,” where microservices trust the gateway simply because of its identity, potentially breaking **authenticity**.

The solution is **decentralized authorization**, where each request carries an **access token (such as a JWT)**. A microservice grants access only if it receives a valid, known token.



F. Ponce, J. Soldani, H. Astudillo, A. Brogi. Smells and Refactorings for Microservices Security: A Multivocal Literature Review. The Journal of Systems and Software. 2022.

How to Eliminate the Security Smells ?

1. **Detect** the Security Smells
2. **Triage** Security Smells. taking into account (Sort of Priority, From Red,Yellow,Green, where Red is critical smell)
   a. service relevance
   b. smell impact on quality attributes

   as well as:

   c. needed effort to refactor
   d. possible need to interact with other teams
3. Choose and implement **Refactoring**.

Refactor or Not Refactor?

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

14. ASE in Industry

ARTIFACTS: WHAT WE DELIVER. These are the units we build and ship in software projects:

1. **Libraries / Packages / Add-ons:** Reusable code that other applications can include.
2. **Standalone Applications:** Full programs that run on their own.
3. **Container Images:** Lightweight units that bundle an application with everything it needs to run.
4. **Virtual Machine Images:** Full operating-system images that include the application and its environment.

CONTAINERIZED APPLICATIONS: WHY WE USE THEM

Containers are popular because they provide several advantages:

1. **Portability:** They run the same way on any machine that supports containers.
2. **Performance:** They start fast and use fewer resources than virtual machines.
3. **Isolation:** Each container runs separately from others, improving security and stability.
4. **Deployment Speed:** Containers let teams release updates quickly and consistently.
5. **Microservice-Friendly:** Ideal for breaking systems into small, independent services.

AUTOMATING THE INFRASTRUCTURE

Modern infrastructure uses automation to ensure reliability and reduce manual work.

1. **Imperative Scripts (Not Recommended):** You tell the system *how* to do every step. This often becomes complex and error-prone.
2. **Declarative Approach (Recommended):** You describe *what* the final state should be, and tools handle the steps. Key properties:
    - **Idempotent:** Running the configuration multiple times gives the same result.
    - **Predictable:** Fewer surprises; the system always ends in the desired state.
    - **Drift Detection:** Detects when the real system differs from the desired configuration.
    - **Self-Healing:** Automatically corrects drift to return to the desired state.

INFRASTRUCTURE AS CODE (IaC)

Infrastructure as Code is a way to manage and provision servers, networks, and other computing resources using configuration files instead of manual setup.

Core Principles

1. **Declarative Approach:** You describe *what* the infrastructure should look like, and tools handle *how* to build it.
2. **System State:** The desired final condition of your infrastructure (e.g., which VMs, networks, services must exist).
3. **System Configuration:** The detailed settings and parameters that define how each part of the infrastructure should function.
4. **Consistency Enforcement:** IaC tools continually compare the actual system to the desired configuration and correct any drift, ensuring predictable and repeatable environments.

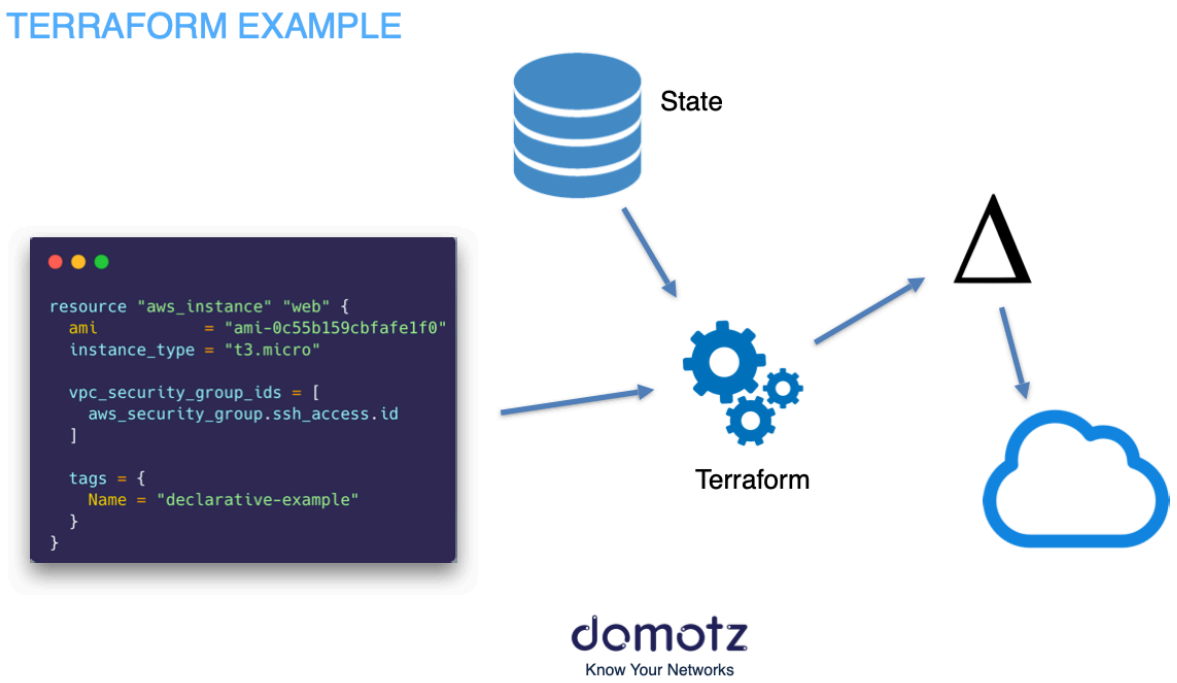
IaC: Tools: Ansible, SaltStack, OpenTofu, Terraform

SALTSTACK: ESSENTIAL CONCEPTS

1. **States (Declarative Configuration):** Files that describe *what* each system should look like (packages, services, files). Salt makes the server match this desired state.
2. **Pillars (Variables):** Secure, structured variables used to pass data such as passwords, paths, or environment-specific settings.
3. **Grains (Host Properties):** Auto-detected information about each machine (OS, IP, CPU, memory). Used to target specific hosts or apply conditional logic.

TERRAFORM: ESSENTIAL CONCEPTS

1. **Modules (Declarative Configuration):** Reusable building blocks. A module describes *what* resources should exist (VMs, networks, buckets). Helps organize and scale infrastructure code.
2. **Variables:** Input values that make modules flexible (region, instance size, environment).
3. **Outputs:** Values returned after Terraform creates the infrastructure (IP addresses, resource IDs). Useful for chaining with other modules or systems.
4. **State (Database):** Terraform's record of what resources exist and how they map to the configuration. Keeps deployments consistent and enables updates, differences, and drift detection.



Infrastructure Fails. And in those failure cases we need to use our Superpowers: IaC + Containers

Monitoring.

Many microservices use different languages and frameworks, often deployed dynamically across distributed locations with sync and async interactions, including third-party integrations. This complexity makes observability **critical** and monitoring solutions **mandatory**.

Effective monitoring involves **incident detection**, **exception handling**, and analyzing **metrics and patterns**. Tools like **Sentry** and **OpenTelemetry** complement each other rather than compete.

Sentry integrates easily with most languages/frameworks, automatically captures exceptions, groups errors to reduce noise, provides full stack traces and context, and sends alerts.

OpenTelemetry is an open standard for telemetry, covering traces, metrics, and logs, with vendor-agnostic instrumentation. Metrics are published by components, collected via a **collector**, and stored or visualized in a monitoring backend.

Common issues: include missing context in spans/traces, incorrect sampling rates, inconsistent naming, and over-alerting.

15. Cloud Edge Continuum

Traditional deployment models:

IoT + Edge: Data processed at the Edge

+ Low latencies

- Limited processing and storage capabilities

IoT + Cloud: Data processed in the Cloud

+ Unlimited processing and storage capabilities

- Very much data transferred

- Mandatory Internet connectivity. Example: Water flooding management must work in critical situations.

- High latencies. Example: Autonomous vehicles need to stop promptly

The **Cloud-Edge Continuum** combines the advantages of Edge Computing and Cloud Computing and extends cloud services to IoT devices. This creates a **distributed, heterogenous infrastructure** system made of different types of hardware working together. Its main benefits are:

- **Computing Power:** You can use both cloud servers and edge devices to process data more efficiently.
- **Connectivity:** Devices communicate more easily, allowing smoother data exchange.
- **Low Latency:** Since data is processed closer to where it is generated, delays are reduced and the system responds faster.

Future applications will mostly use **containers** and **microservices** and will run across both **cloud and edge environments**. However, managing these systems is difficult because everything keeps changing. This leads to several issues:

Infrastructure changes:

Workloads on different nodes can shift, network latency and bandwidth may vary, and nodes can join or leave the system at any time. Temporary network failures can also happen.

Application changes:

The application's code and requirements can change, which means updates must be applied quickly.

Because of all these factors, applications need continuous management even after they are deployed.

How to suitably place a composite application in the Cloud-Edge Continuum ? It's challenging and NP-hard

1. **Application requirements:** Need to consider different types of application requirements: Hardware, Software, QoS (Response time, Reliability, Inter-service latency)
2. **Data Gravity:** Large datasets.
    a. Tent to "attract" services using/producing such data.
    b. Are heavy to move
3. **Security:** Need to match application's security requirements with infrastructure's security capabilities
4. **Trust:** Need to model (non-monotonic, conditionally transitive) trust relations among different stakeholders
5. **Sustainability:** Take into account the environmental impact of software applications:
    a. energy consumption
    b. CO2 emissions
6. **Infrastructure:** Heterogenous, Large, Dynamic

So, Placing a composite application in the Cloud-Edge Continuum is challenging (and NP-hard)

Application placement Approaches:

1. **ML:** Infrastructure is very dynamic, Lack of explainability
2. **MILP:** Finds optimal solutions, Hard to read, hard to code non-numerical info, Slow to run
3. **Declarative programming:**In a declarative model you:

1. Describe the conditions that define a valid placement for an application (for example, required resources, latency limits, or hardware capabilities).

2. Let the inference engine automatically searches for nodes that meet these conditions.

Advantages:

- Simple to read and understand
- Easy to express non-numerical or high-level requirements
- Transparent and explainable, because the rules and decisions are clear

This approach shifts the focus from *how* to find a placement to *what* the placement must satisfy, allowing the system to handle the decision-making.

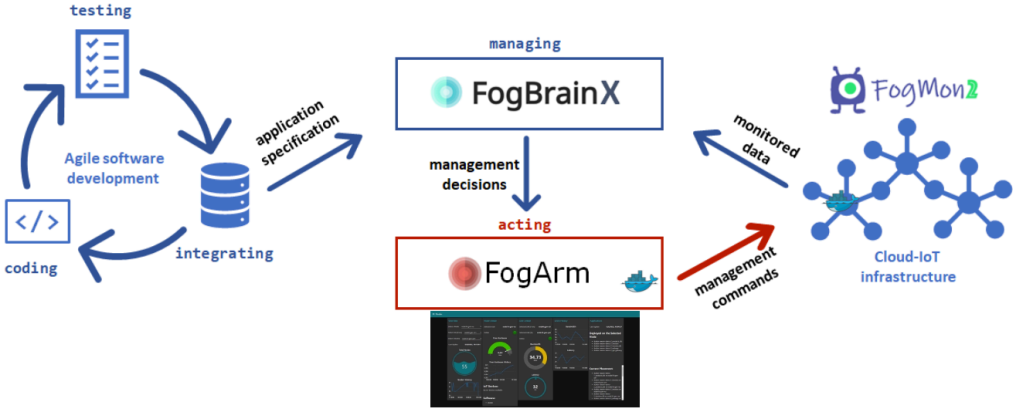
Effective monitoring is necessary to track both applications and the underlying infrastructure. The monitoring system must be lightweight, fault-tolerant, and able to adapt to changes.

One useful technique is **continuous reasoning**, which analyzes large systems by focusing only on recent updates and reusing previous results whenever possible. In a Cloud-Edge environment, this is important for deciding when to replace, migrate, restart, or scale services.

A recent example is **FogBrainX**, which compares changes in application specifications with real-time infrastructure data. It helps make placement and management decisions by:

- Reacting to infrastructure changes, such as available resources or network quality, which may require moving services.
- Responding to updates in service requirements (software, hardware, IoT needs) or communication patterns that may require reconfiguration.
- Managing additions or removals of services or communication rules defined in the application's specifications.

After FogBrainX decides what actions are needed, it sends these instructions to **FogArm**, which applies the changes in the Cloud-IoT environment.



15.1 ECLYPSE

WHY?

The Cloud-Edge continuum is very complex. It contains many different types of devices, changes over time, and requires strict quality-of-service levels.

Simple simulation tools are not enough. There are also no easy, user-friendly tools that can represent this complexity in a realistic way.

WHAT?w

ECLYPSE is a Python-based platform for creating simulated or emulated Cloud-Edge environments. It allows you to:

- Manage, monitor, and test complex Cloud-Edge setups
- Model real-world infrastructures
- Model application behaviour, including machine-learning workloads

It provides a practical way to study and experiment with Cloud-Edge systems before deploying them in reality.

ECLYPSE: Simplifying Cloud-Edge Simulations

Simulation steps:

1. Assets

You define the basic components of the system:

- **Nodes** (e.g., servers, edge devices)
- **Edges** (connections between them)

2. Requirements & Capabilities

You describe what each part needs and what it can do:

- **Services** and how they interact
- **Nodes** and their network links or resource limits

3. Events

You specify what happens during the simulation:

- **Place** services on nodes
- **Update** resources or conditions
- **Monitor** system behaviour
- **Report** results

4. Simulation Execution

You run and customize the scenario:

- Place components
- Update states
- Monitor performance
- Adjust the setup as needed

Emulation Overview:

1. **Remote Configuration:** Uses Ray for distributed setups.
2. **Remote Environment:** Supports logic with MPI/REST, and actors with delayed remote calls (RMI).
3. **Events:** Place, update, remotely monitor, and report on nodes/services.

ECLYPSE - Key Features:

- Public GitHub repo with examples and ready-to-use interface.
- Pre-built components: placement strategies, update policies, metrics, assets.
- Clear and updated documentation.
- Modular and easy to extend: customize events, metrics, strategies, policies.
- Works with popular Python libraries: NetworkX, Ray, PyTorch, Pandas.

Extra Info (short):

ECLYPSE helps test distributed cloud-edge systems realistically, letting you try algorithms, simulate failures, and monitor dynamic behaviors without real hardware.

How Would Your Algorithm Behave in the Real World?

Use **ECLYPSE** as a testing environment to understand how your algorithm performs under realistic conditions.

Test your algorithm directly

- Insert your own algorithm into the platform.
- Experiment with placement, scheduling, consensus, or replication strategies.
- Simulate real-world conditions such as changing resources, failures, latency, and node churn.

Prototype decentralized behaviour

- Deploy services using Ray actors that follow local policies and have only partial knowledge.
- Explore what happens when nodes take independent decisions.

Build meaningful benchmarks

- Contribute to designing benchmarks that are realistic, repeatable, and useful for evaluating distributed and parallel algorithms under stress.

16. Quantum Software Engineering

1. Why Quantum Computing?

Quantum computing is a different way of solving difficult problems by using principles from quantum physics.

Classical computers use bits that are either 0 or 1. Quantum computers use **qubits**, which can be 0, 1, or a mix of both at the same time. Qubits can also become **entangled**, meaning their states are linked even if they are far apart.

These properties allow some problems that take classical computers extremely long to solve to be solved much faster on a quantum computer.

Possible advantages

- Faster algorithms, such as:

– **Shor's algorithm** for breaking large-number factorization

– **Grover's algorithm** for speeding up search problems

Main application areas

- Chemistry and pharmaceuticals: simulate molecules
- Finance: optimize portfolios and manage risk
- Logistics: solve hard optimization problems
- Energy: discover new materials
- Cryptography: design post-quantum secure methods
- AI/ML: speed up certain high-dimensional tasks

2. What Are Quantum Computers?

A quantum computer uses quantum mechanics to process information.

Its basic unit, the **qubit**, can be a combination of 0 and 1 until it is measured, at which point it becomes one of the two based on probability.

Qubits can also be **entangled**, meaning the state of one instantly affects the state of another.

Implementation Technologies

There is no single standard for building qubits. Each technology has benefits and drawbacks:

Technology	Examples	Advantages	Disadvantages
Superconductors	IBM, Google	Fast, scalable	Very sensitive to temperature
Ions	IonQ, Honeywell	Very accurate, stable	Slower operations
Photons	PsiQuantum, Xanadu	Easy to scale, no cooling	Still developing
Neutral atoms	PASQAL, QuEra	Long coherence times	Requires cooling

Technology	Examples	Advantages	Disadvantages
Silicon	Intel	Uses existing chip tech	Still early-stage
Topological	Microsoft	Potentially very stable	Not demonstrated yet

3. Current State: The NISQ Era

Today's quantum computers are called **NISQ machines** (Noisy Intermediate-Scale Quantum):

- They have limited usable qubits (typically under 100–1000)
- They are very sensitive to noise, which causes errors
- They can only run short, simple circuits before noise becomes too high

4. Quantum Computing in the Cloud

Companies like IBM, IonQ, and Rigetti allow you to use quantum hardware through the cloud.

Because quantum computers are probabilistic, each program must be run many times (called **shots**) to get reliable results.

5. Quantum Software Engineering (QSE)

Quantum Software Engineering applies software engineering methods to quantum programming, which still relies heavily on low-level circuit design.

Quantum Software Engineering (QSE) focuses on applying solid engineering methods to build, operate, and maintain quantum software and its documentation. The main goal is to create quantum programs that are reliable, run efficiently on quantum computers, and remain cost-effective.

The Talavera Manifesto for QSE highlights several important principles, such as:

- QSE should work with different quantum programming languages and technologies.
- It should support the combination of classical and quantum computing. Quantum computers should mainly be used for tasks where they are especially strong, such as solving factorization problems.

Main goals (Talavera Manifesto)

1. Do not depend on any single quantum language or technology

2. Combine classical and quantum computing cleanly
3. Ensure quantum software quality, reuse, security, and maintainability
4. Manage the quantum software lifecycle like traditional projects

Current challenges

- Hard to test and debug because hardware is noisy
- Tools and IDEs are still immature
- Few high-level abstractions for developers
- Need for compilers that target many different hardware backends

---

6. New Techniques: Shot-wise Distribution & Cut&Shoot

6.1 Shot-wise Distribution

Each shot (one run of a quantum circuit) is independent.

We can split shots across multiple quantum processors (QPUs):

- Faster execution
- More resilience against hardware failures
- Ability to choose preferred or less noisy devices

A **Quantum Broker** handles:

- Compiling the circuit
- Selecting the available QPUs
- Assigning shots to each device
- Merging all results afterward

Quantum Broker

A practical example of QSE is the Quantum Broker, which answers the question: “Which quantum computer should I use for my algorithm?” This is useful for clients who may not fully understand the differences between quantum providers.

Even after choosing a provider and running algorithms, several problems can still occur:

- Availability: What if the chosen quantum computer becomes unavailable while the algorithm is running?

- Requirements Accuracy: How do I balance cost, speed, and accuracy based on my needs?
- Customization: How can I adjust how the quantum computer makes decisions so it fits my requirements?

The idea of **shot distribution** is important for getting the best performance in quantum computing. A “shot” is one execution of a quantum circuit. By spreading these shots across multiple quantum computers, we can improve performance and reliability. Running the same circuit many times gives better statistical results because quantum outputs are probabilistic. After collecting data from all runs, we combine the results to get a complete picture of how the circuit behaves.

Given a quantum circuit and certain requirements, the quantum broker chooses the best set of quantum computers to distribute the shots. For each selected computer, it decides which compiler to use and how many shots to run.

This method provides several benefits:

- **Improved Resilience:** Using multiple quantum systems makes the overall process more resistant to failures.
- **High Customization:** Users can fine-tune how the computation is managed.
- **Partial Distributions:** It allows creating partial results from different systems and merging them later to form a full view of the circuit’s execution. More advanced merging strategies are also possible.

6.2 Circuit Cutting

Circuit cutting allows large circuits to be split into smaller pieces that today’s NISQ machines can execute.

Advantages

- Lets us run circuits that would otherwise be too large
- Reduces circuit depth and therefore hardware noise

Disadvantages

- Classical pre-processing and reconstruction time grows quickly with circuit size and number of fragments

---

6.3 Cut&Shoot (Combined Approach)

This approach integrates both ideas:

1. Cut the large circuit into smaller fragments
2. Distribute the shots for each fragment across several QPUs
3. Combine the partial outputs
4. Reconstruct the final probability distribution of the original circuit

Simulator experiments show:

- Less error due to cutting
- More resilience thanks to shot distribution
- Only moderate overhead

---

7. Conclusions

Techniques like Shot-wise Distribution and Cut&Shoot help overcome the limits of today's NISQ devices. By using software intelligently, we can run larger, more accurate, and more resilient quantum programs even on noisy and varied hardware.

Quantum Software Engineering will play a central role in turning quantum computing into a practical technology used in industry.