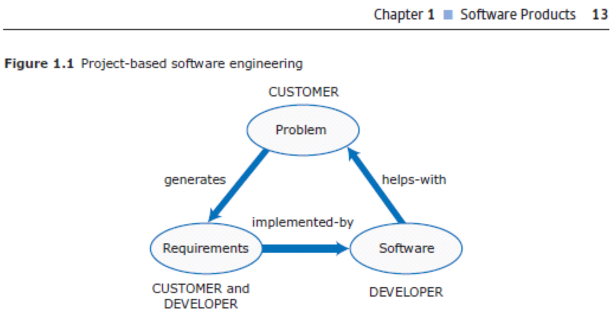
*What is product-based software engineering?



Chapter 1 ■ Software Products 13

Figure 1.1 Project-based software engineering

Customers decide system functionalities
Business changes → requirements change → software must change

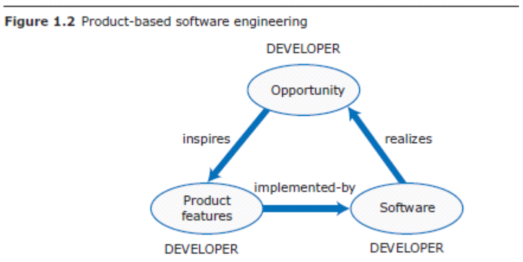


Figure 1.2 Product-based software engineering

Developer decides product features and evolution

Customers reluctant to change product after investing on it
→ Getting product to customers **quickly** is critical
→ Need rapid software developments technique (**Agile** methods)

. Customers defined their requirements and worked with the development team to specify, in detail, the software's functionality and its critical attributes.Project-based techniques are not suited to product development because of fundamental differences between project-based and product-based software engineering. These differences are illustrated in Figures 1.1 and 1.2.
Software projects involve an external client or customer who decides on the functionality of the system and enters into a legal contract with the software development company. The customer's problem and current processes are used as a basis for creating the software requirements, which specify the software to be implemented. As the business changes, the supporting software has to change. The company using the software decides on and pays for the changes. Software often has a long lifetime, and the costs of changing large systems after delivery usually exceed the initial software development costs.
Software products are specified and developed in a different way. There is no external customer who creates requirements that define what the software

must do. The software developer decides on the features of the product, when new releases are to be made available, the platforms on which the software

will be implemented, and so on. The needs of potential customers for the software are obviously considered, but customers can’t insist that the software includes particular features or attributes. The development company chooses when changes will be made to the software and when they will be released to users.
As development costs are spread over a much larger customer base, product-based software is usually cheaper, for each customer, than custom software. However, buyers of the software have to adapt their ways of working to the software, since it has not been developed with their specific needs in mind. As the developer rather than the user is in control of changes, there is a risk that the developer will stop supporting the software. Then the product customers will need to find an alternative product.

, there are two other important differences between project-based and product-based software engineering:
1. Product companies can decide when to change their product or take their product off the market. If a product is not selling well, the company can cut costs by stopping its development. Custom software developed in a software project usually has a long lifetime and has to be supported throughout that lifetime. The customer pays for the support and decides when and if it should end.
2. For most products, getting the product to customers quickly is critical. Excellent products often fail because an inferior product reaches the market first and customers buy that product. In practice, buyers are reluctant to change products after they have invested time and money in their initial choice.

Bringing the product to the market quickly is important for all types of products, from small-scale mobile apps to enterprise products such as Microsoft Word. This means that engineering techniques geared to rapid software development (agile methods) are universally used for product development

*What is the incremental development and delivery advocated by Agile?

All agile methods are based on incremental development and delivery.
The best way to understand incremental development is to think of a software product as a set of features. Each feature does something for the software user. There might be a feature that allows data to be entered, a feature to search the entered data, and a feature to format and display the data. Each software increment should implement a small number of product features.
With incremental development, you delay decisions until you really need to make them. You start by prioritizing the features so that the most important features are implemented first. You don’t worry about the details of all the features—you define only the details of the feature that you plan to include in an increment. That feature is then implemented and delivered.
Users or surrogate users can try it out and provide feedback to the development team. You then go on to define and implement the next feature of the system.

Figure 2.1 Incremental development

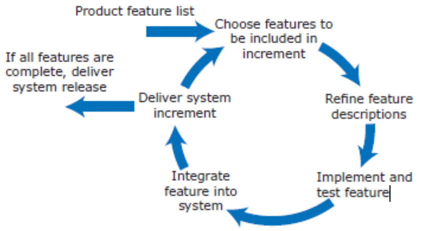
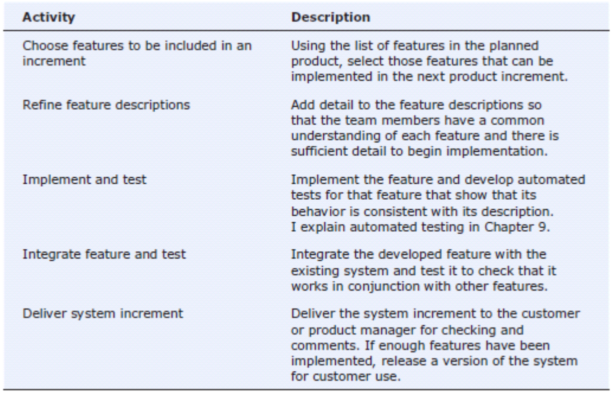

Product feature list
Choose features to be included in increment
Refine feature descriptions
Implement and test feature
Integrate feature into system
Deliver system increment
If all features are complete, deliver system release

Table 2.2 Incremental development activities



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

*** Which are the key Scrum practices?**

Scrum in practice

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

Key Scrum practices

- ***Product backlog***

This is a to-do list of items to be implemented that is reviewed and updated before each sprint.

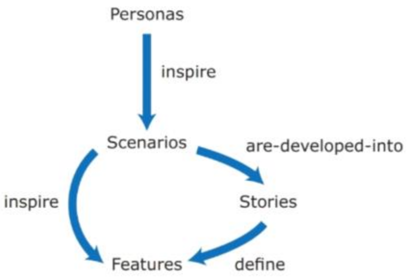
- ***Timeboxed sprints***

Fixed-time (2-4 week) periods in which items from the product backlog are implemented,

- ***Self-organizing teams***

Self-organizing teams make their own decisions and work by discussing issues and making decisions by consensus.

*What are personas, scenarios, user stories and features?



Personas

One of the first questions you should ask when developing a software product is “Who are the target users for my product?” You need to have some understanding of your potential users in order to design features that they are likely to find useful and to design a user interface that is suited to them.

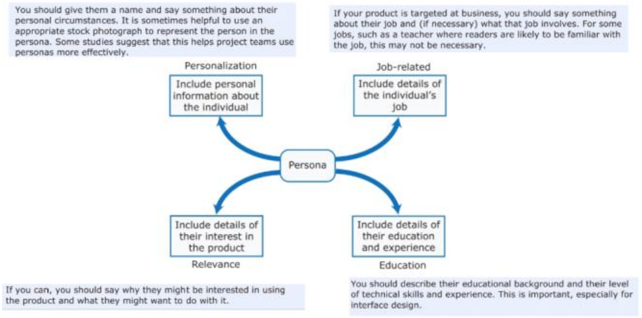
Who are the target users for our product?

- Need to understand potential users to design features useful for them
- Background, skills and experience of potential users are important

Personas are (imagined) types of product users

- e.g. for a dentist agenda: *dentist*, *receptionist*, *patient*
- Generally we need a few (1-2, max 5) personas to identify key product features

Aspects of persona descriptions


You should give them a name and say something about their personal circumstances. It is sometimes helpful to use an appropriate stock photograph to represent the person in the persona. Some studies suggest that this helps project teams use personas more effectively.

If your product is targeted at business, you should say something about their job and (if necessary) what that job involves. For some jobs, such as a teacher where readers are likely to be familiar with the job, this may not be necessary.

If you can, you should say why they might be interested in using the product and what they might want to do with it.

You should describe their educational background and their level of technical skills and experience. This is important, especially for interface design.

Personalization
Include personal information about the individual
Job-related
Include details of the individual’s job
Persona
Include details of their interest in the product
Relevance
Include details of their education and experience
Education

2 Scenarios

As a product developer, your aim should be to discover the product features that will tempt users to adopt your product rather than competing software. There is no easy way to define the “best” set of product features. You have to use your own judgement about what to include in your product. To help select and design features, I recommend that you invent scenarios to imagine how users could interact with the product that you are designing.

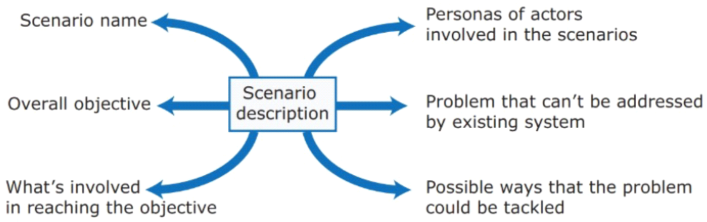
A scenario is a narrative that describes a situation in which a user is using your product's features to do something that they want to do. The scenario should briefly explain the user's problem and present an imagined way that

the problem might be solved. There is no need to include everything in the scenario; it isn't a detailed system specification

To discover product features, we can define scenarios of user interactions with the product
Scenario = narrative describing a situation in which a user is using our product's features to do something she wants to do

Writing scenarios

Several scenarios (e.g. 3-4) for each persona, covering main responsabilities of persona
Written from user's perspective
Each team member should (individually) create some scenarios,and then discuss them with rest of team and with users (if possible)

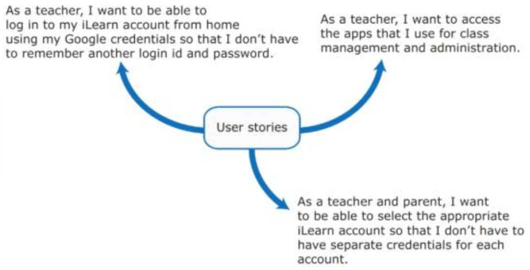

Scenario description
Scenario name
Overall objective
What's involved in reaching the objective
Personas of actors involved in the scenarios
Problem that can't be addressed by existing system
Possible ways that the problem could be tackled

3.3 User stories

I explained in Section 3.2 that scenarios are descriptions of situations in which a user is trying to do something with a software system. Scenarios are highlevel stories of system use. They should describe a sequence of interactions with the system but should not include details of these interactions.
User stories are finer-grain narratives that set out in a more detailed and structured way a single thing that a user wants from a software system. I presented a user story at the beginning of the chapter:
As an author I need a way to organize the book that I'm writing into chapters and sections.
This story reflects what has become the standard format of a user story:
As a <role>, I <want / need> to <do something>
Another example of a user story taken from Emma's scenario might be:
As a teacher, I want to tell all members of my group when new information is available.
A variant of this standard format adds a justification for the action:
As a <role> I <want / need> to <do something> so that <reason>
For example:
As a teacher, I need to be able to report who is attending a class trip so that the school maintains the required health and safety records.

- Scenarios = high-level stories of product use
- User stories = finer-grained narratives

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



As a teacher, I want to be able to log in to my iLearn account from home using my Google credentials so that I don't have to remember another login id and password.

As a teacher, I want to access the apps that I use for class management and administration.

As a teacher and parent, I want to be able to select the appropriate iLearn account so that I don't have to have separate credentials for each account.

Feature identification

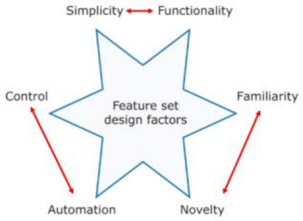
- Aim: Getting a list of features that define our product
- Properties

• ***Independence*** - A feature should not depend on how other system features are implemented and should not be affected by the order of activation of other features.

• ***Coherence*** - Features should be linked to a single item of functionality. They should not do more than one thing, and they should never have side effects.

• ***Relevance*** - System features should reflect the way users normally carry out some task. They should not offer obscure functionality that is rarely required.

Factors in feature set design



Q2

What is the effect of a git add

The `git add` command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit. However, `git add` doesn't really affect the repository in any significant way—changes are not actually recorded until you run `git commit`.

*Git branch

Creating a New Branch

What happens when you create a new branch? Well, doing so creates a new pointer for you to move around. Let’s say you want to create a new branch called `testing`. You do this with the `git branch` command:

```
$ git branch testing
```

This creates a new pointer to the same commit you’re currently on.

*Git clone

The `git clone` command is used to create a copy of a specific repository or branch within a repository.

When you clone a repository, you don't get one file, like you may in other centralized version control systems. By cloning with Git, you get the entire repository - all files, all branches, and all commits.

Cloning a repository is typically only done once, at the beginning of your interaction with a project. Once a repository already exists on a remote, like on GitHub, then you would clone that repository so you could interact with it locally. Once you have cloned a repository, you won't need to clone it again to do regular development.

*Git checkout

The `git checkout` command lets you navigate between the branches created by `git branch`. Checking out a branch updates the files in the working directory to match the version stored in that branch, and it tells Git to record all new commits on that branch. Think of it as a way to select which line of development you’re working on.

*Git push

he `git push` command is used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repo.

*Git commit

The `git commit` command captures a snapshot of the project's currently staged changes. Committed snapshots can be thought of as “safe” versions of a project—Git will never change them unless you explicitly ask it to. Prior to the execution of `git commit`, The `git add` command is used to promote or 'stage' changes to the project that will be stored in a commit.

*Git pull

The `git pull` command is used to fetch and download content from a remote repository and immediately update the local repository to match that content. Merging remote upstream changes into your local repository is a common task in Git-based collaboration work flows. The `git pull` command is actually a combination of two other commands, `git fetch` followed by `git merge`. In the first stage of operation `git pull` will execute a `git fetch` scoped to the local branch that `HEAD` is pointed at. Once the content is downloaded, `git pull` will enter a merge workflow. A new merge commit will be-created and `HEAD` updated to point at the new commit.

*How does GitHub flow work?

The GitHub flow is a workflow designed to work well with Git and GitHub.

It focuses on branching and makes it possible for teams to experiment freely, and make deployments regularly.

The GitHub flow works like this:

- Create a new Branch
- Make changes and add Commits
- Open a Pull Request
- Review
- Deploy
- Merge

Create a New Branch

Branching is the key concept in Git. And it works around the rule that the master branch is ALWAYS deployable.

That means, if you want to try something new or experiment, you create a new branch! Branching gives you an environment where you can make changes without affecting the main branch.

When your new branch is ready, it can be reviewed, discussed, and merged with the main branch when ready.

When you make a new branch, you will (almost always) want to make it from the master branch.

Make Changes and Add Commits

After the new branch is created, it is time to get to work. Make changes by adding, editing and deleting files. Whenever you reach a small milestone, add the changes to your branch by commit.

Adding commits keeps track of your work. Each commit should have a message explaining what has changed and why. Each commit becomes a part of the history of the branch, and a point you can revert back to if you need to.

Open a Pull Request

Pull requests are a key part of GitHub. A Pull Request notifies people you have changes ready for them to consider or review.

You can ask others to review your changes or pull your contribution and merge it into their branch.

Review

When a Pull Request is made, it can be reviewed by whoever has the proper access to the branch. This is where good discussions and review of the changes happen.

Pull Requests are designed to allow people to work together easily and produce better results together!

If you receive feedback and continue to improve your changes, you can push your changes with new commits, making further reviews possible.

Deploy

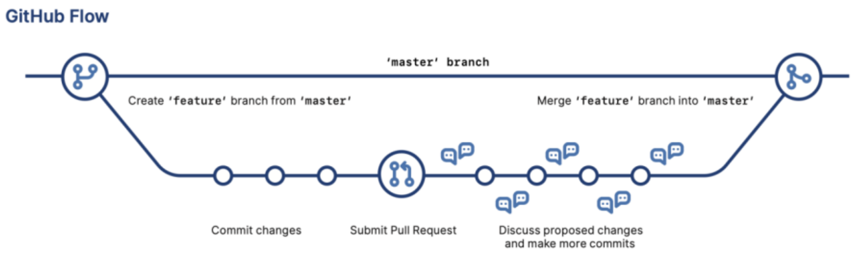
When the pull request has been reviewed and everything looks good, it is time for the final testing. GitHub allows you to deploy from a branch for final testing in production before merging with the master branch.

If any issues arise, you can undo the changes by deploying the master branch into production again!

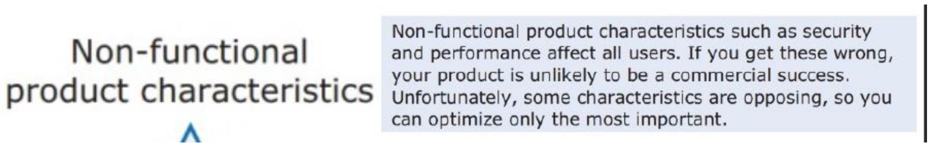
Merge

After exhaustive testing, you can merge the code into the master branch!

Pull Requests keep records of changes to your code, and if you commented and named changes well, you can go back and understand why changes and decisions were made.

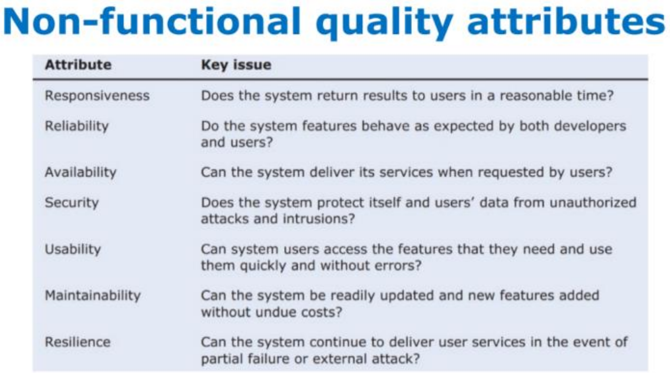

GitHub Flow
'master' branch
Create 'feature' branch from 'master'
Merge 'feature' branch into 'master'
Commit changes
Submit Pull Request
Discuss proposed changes and make more commits

* What is the role of non-functional quality attributes?



Non-functional product characteristics

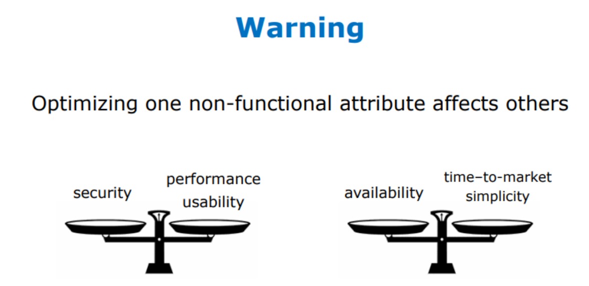
Non-functional product characteristics such as security and performance affect all users. If you get these wrong, your product is unlikely to be a commercial success. Unfortunately, some characteristics are opposing, so you can optimize only the most important.



Non-functional quality attributes

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

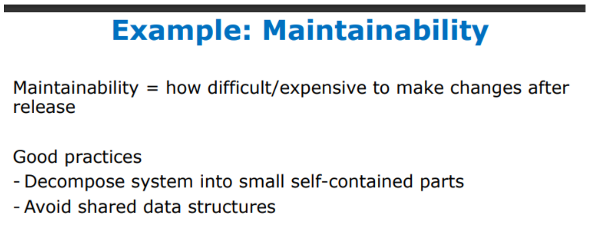
(Important for final product - not for prototype)



Warning

Optimizing one non-functional attribute affects others

security
performance
usability
availability
time–to-market
simplicity



Example: Maintainability

Maintainability = how difficult/expensive to make changes after release

Good practices
- Decompose system into small self-contained parts
- Avoid shared data structures

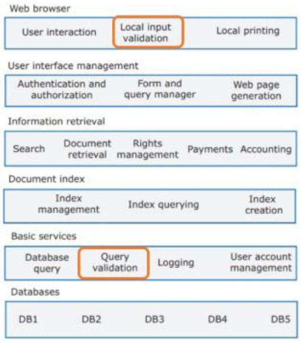
*decomposition in a software architecture?

The idea of abstraction is fundamental to all software design. Abstraction in software design means that you focus on the essential elements of a system or software component without concern for its details. At the architectural level, your concern should be on large-scale architectural components. Decomposition involves analyzing these large-scale components and representing them as a set of finer-grain components.

For example, Figure 4.6 is a diagram of the architecture of a product that I was involved with some years ago. This system was designed for use in libraries and gave users access to documents that were stored in a number of private databases, such as legal and patent databases. Payment was required for access to these documents. The system had to manage the rights to these documents and collect and account for access payments.
In this diagram, each layer in the system includes a number of logically related components. Informal layered models, like Figure 4.6, are widely used to show how a system is decomposed into components, with each component providing significant system functionality.

Layered architectures

- Each layer is an area of concern and is considered separately from other layers
- Within each layer, the components are independent and do not overlap in functionality
- The architectural model is a high-level model that does not include implementation information



Web browser

User interaction | Local input validation | Local printing

User interface management

Authentication and authorization | Form and query manager | Web page generation

Information retrieval

Search | Document retrieval | Rights management | Payments | Accounting

Document index

Index management | Index querying | Index creation

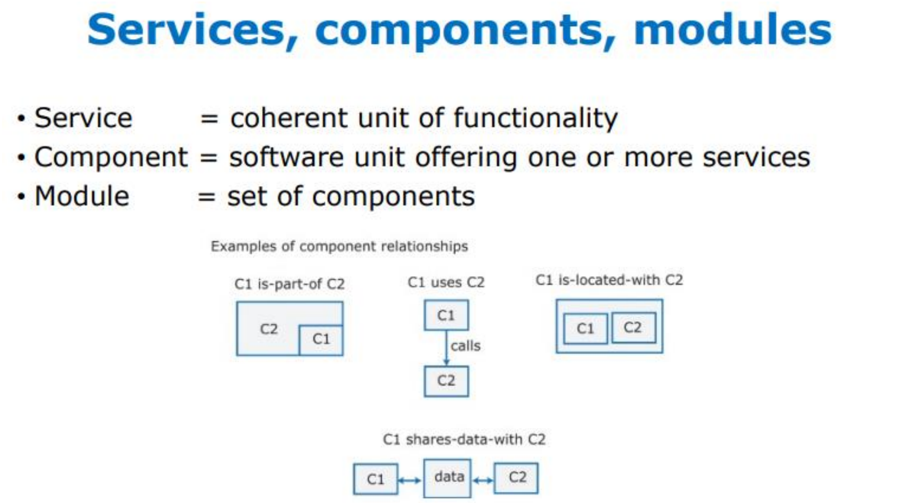
Basic services

Database query | Query validation | Logging | User account management

Databases

DB1 | DB2 | DB3 | DB4 | DB5

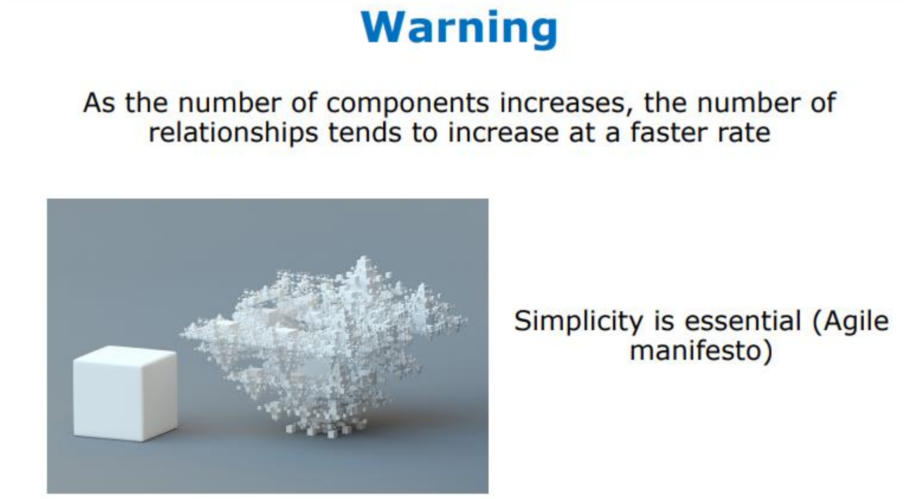
(Concerns may not be always 100% separated in practice)



Services, components, modules

- Service = coherent unit of functionality
- Component = software unit offering one or more services
- Module = set of components

Examples of component relationships



Warning

As the number of components increases, the number of relationships tends to increase at a faster rate

Simplicity is essential (Agile manifesto)

* What is a distribution architecture?

The majority of software products are now web-based products, so they have a client–server architecture. In this architecture, the user interface is implemented on the user's own computer or mobile device. Functionality is distributed between the client and one or more server computers. During the architectural design process, you have to decide on the "distribution architecture" of the system. This defines the servers in the system and the allocation of components to these servers.
Client–server architectures are a type of distribution architecture that is suited to applications in which clients access a shared database and business

logic operations on those data.

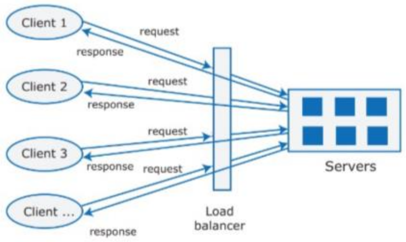
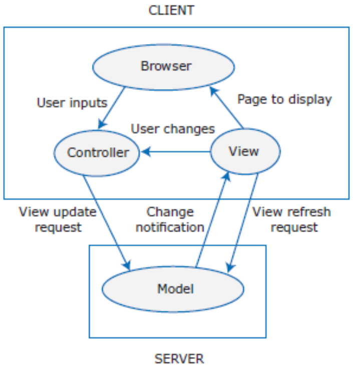


Figure 4.12 shows a logical view of a client–
server architecture that is widely used in web-based and mobile software products. These applications include several servers, such as web servers and database servers. Access to the server set is usually mediated by a load balancer, which distributes requests to servers. It is designed to ensure that the computing load is evenly shared by the set of servers.

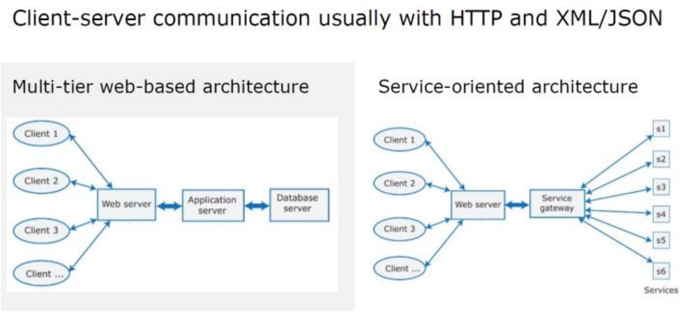
Figure 4.13 The Model-View-Controller pattern



Client–server interaction is usually organized using what is called the Model-View-Controller (MVC) pattern. This architectural pattern is used so that client interfaces can be updated when data on the server change (Figure 4.13).

Model-View-Controller pattern

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


Client-server communication usually with HTTP and XML/JSON

Multi-tier web-based architecture

Service-oriented architecture

Sometimes a multi-tier architecture may use additional specialized servers. For example, in a theater booking system, the user's payments may be handled by a credit card payment server provided by a company that specializes in credit card payments. This makes sense for most e-commerce applications, as a high cost is involved in developing a trusted payment system. Another type of specialized server that is commonly used is an authentication server. This checks users' credentials when they log in to the system.

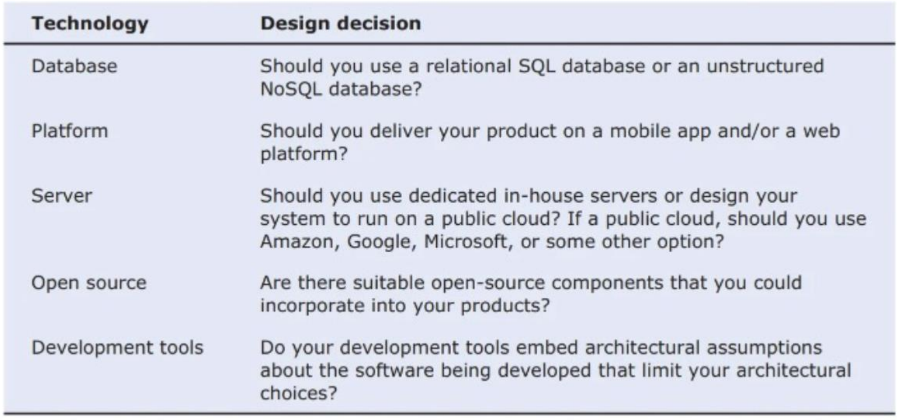
service-oriented architecture (Figure 4.15) where many servers may be involved in providing services. Services in a service-oriented architecture are stateless components, which means that they can be replicated and can migrate from one computer to another. A service-oriented architecture is usually easier to scale as demand increases and is resilient to failure.

Choosing distribution architecture

You have to decide which of these to choose for your software product. The issues that you must consider are:
1. *Data type and data updates* If you are mostly using structured data that may be updated by different system features, it is usually best to have a single shared database that provides locking and transaction management. If data are distributed across services, you need a way to keep them consistent, and this adds overhead to your system.
2. *Frequency of change* If you anticipate that system components will regularly be changed or replaced, then isolating these components as separate services simplifies those changes.
3. *The system execution platform* If you plan to run your system on the cloud with users accessing it over the Internet, it is usually best to implement it as a service-oriented architecture because scaling the system is simpler. However, if your product is a business system that runs on local servers, a multi-tier architecture may be more appropriate.

*Which are the technology choices that affect a software architecture?

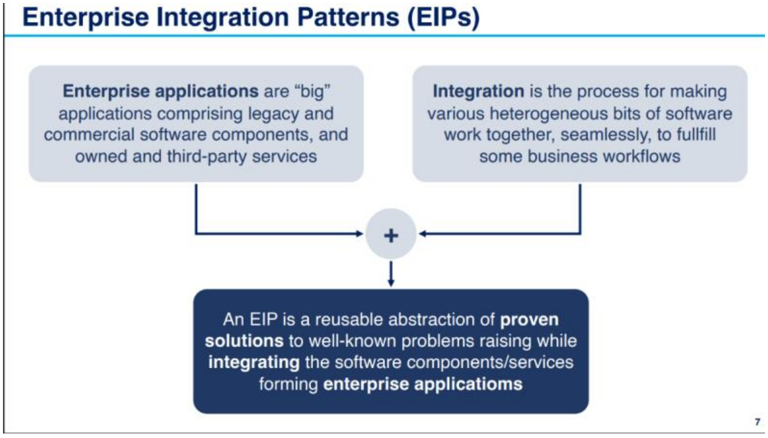
An important part of the process of designing software architecture is to make decisions about the technologies you will use in your product. The technologies that you choose affect and constrain the overall architecture of your system. It is difficult and expensive to change these during development, so it is important that you carefully consider your technology choices.



Technology	Design decision
Database	Should you use a relational SQL database or an unstructured NoSQL database?
Platform	Should you deliver your product on a mobile app and/or a web platform?
Server	Should you use dedicated in-house servers or design your system to run on a public cloud? If a public cloud, should you use Amazon, Google, Microsoft, or some other option?
Open source	Are there suitable open-source components that you could incorporate into your products?
Development tools	Do your development tools embed architectural assumptions about the software being developed that limit your architectural choices?

Book:119-123

* Which are the main features of Enterprise Integration Patterns?



Enterprise Integration Patterns (EIPs)

Enterprise applications are "big" applications comprising legacy and commercial software components, and owned and third-party services

Integration is the process for making various heterogeneous bits of software work together, seamlessly, to fullfill some business workflows

An EIP is a reusable abstraction of **proven solutions** to well-known problems raising while **integrating** the software components/services forming **enterprise applicatioms**

*What is a Camel route (for)?

ROUTES

In Apache Camel, a *route* is a set of processing steps that are applied to a message as it travels from a source to a destination. A route typically consists of a series of processing steps that are connected in a linear sequence.
A Camel *route* is where the integration flow is defined. For example, you can write a Camel route to specify how two systems can be integrated. You can also specify how the data can be manipulated, routed, or mediated between the systems.

The routes are typically defined using a simple, declarative syntax that is easy to read and understand.

For instance, you could write a *route* to consume files from an FTP server and send them to an ActiveMQ messaging system. A *route* to do so, using Java DSL, would look like this:

```
from("ftp:myserver/folder")
  .to("activemq:queue:cheese");
```

Camel *routes* can be defined using a variety of domain-specific languages (DSLs), such as Java, Spring XML, or YAML. For example, you could write the *route* described above using XML:

```
<route>
  <from uri="ftp:myserver/folder"/>
  <to uri="activemq:queue:cheese"/>
</route>
```

Q3

*Which are the differences between multi-tenant and multi-instance SaaS systems?

5.4.1 Multi-tenant systems

In a multi-tenant database, a single database schema, defined by the SaaS provider, is shared by all of the system's users. Items in the database are tagged with a tenant identifier, representing a company that has stored data in the system. The database access software uses this tenant identifier to provide "logical isolation," which means that users seem to be working with their own database.

Mid-size and large businesses that are buying software as a service rarely want to use generic multi-tenant software. They want a version of the software that is adapted to their own requirements and that presents their staff with a customized version of the software.

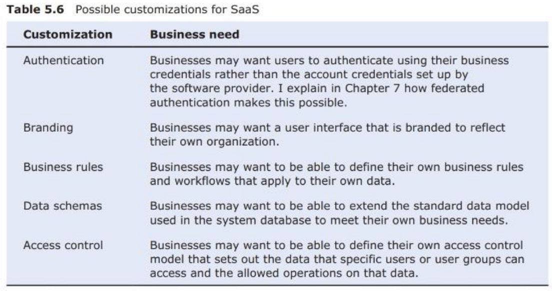


Table 5.6 Possible customizations for SaaS

Customization	Business need
Authentication	Businesses may want users to authenticate using their business credentials rather than the account credentials set up by the software provider. I explain in Chapter 7 how federated authentication makes this possible.
Branding	Businesses may want a user interface that is branded to reflect their own organization.
Business rules	Businesses may want to be able to define their own business rules and workflows that apply to their own data.
Data schemas	Businesses may want to be able to extend the standard data model used in the system database to meet their own business needs.
Access control	Businesses may want to be able to define their own access control model that sets out the data that specific users or user groups can access and the allowed operations on that data.

5.4.2 Multi-instance systems

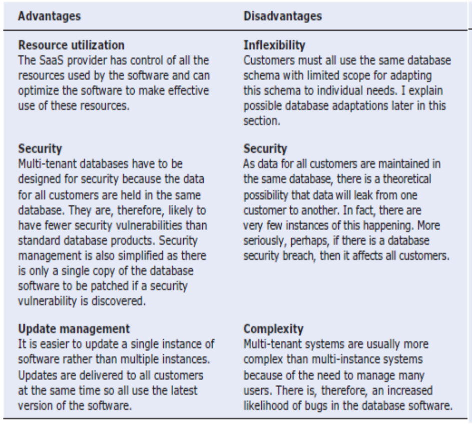
Multi-instance systems are SaaS systems where each customer has its own system that is adapted to its needs, including its own database and security controls. Multi-instance, cloud-based systems are conceptually simpler than multi-tenant systems and avoid security concerns such as data leakage from one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

There are two types of multi-instance system:

1. *VM-based multi-instance systems* In these systems, the software instance and database for each customer run in its own virtual machine. This may appear to be an expensive option, but it makes sense when your product is aimed at corporate customers who require 24/7 access to their software and data. All users from the same customer may access the shared system database.

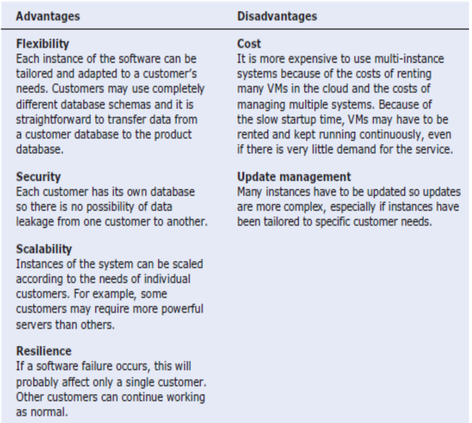
2. *Container-based multi-instance systems* In these systems, each user has an isolated version of the software and database running in a set of containers. Generally, the software uses a microservices architecture, with each service running in a container and managing its own database. This approach is suited to products in which users mostly work independently, with relatively little data sharing. Therefore, it is most suited for products that serve individuals rather than business customers or for business products that are not data intensive.

Table 5.5 Advantages and disadvantages of multi-tenant databases



Advantages	Disadvantages
Resource utilization The SaaS provider has control of all the resources used by the software and can optimize the software to make effective use of these resources.	**Inflexibility** Customers must all use the same database schema with limited scope for adapting this schema to individual needs. I explain possible database adaptations later in this section.
Security Multi-tenant databases have to be designed for security because the data for all customers are held in the same database. They are, therefore, likely to have fewer security vulnerabilities than standard database products. Security management is also simplified as there is only a single copy of the database software to be patched if a security vulnerability is discovered.	**Security** As data for all customers are maintained in the same database, there is a theoretical possibility that data will leak from one customer to another. In fact, there are very few instances of this happening. More seriously, perhaps, if there is a database security breach, then it affects all customers.
Update management It is easier to update a single instance of software rather than multiple instances. Updates are delivered to all customers at the same time so all use the latest version of the software.	**Complexity** Multi-tenant systems are usually more complex than multi-instance systems because of the need to manage many users. There is, therefore, an increased likelihood of bugs in the database software.

Table 5.7 Advantages and disadvantages of multi-instance databases



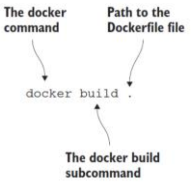
Advantages	Disadvantages
Flexibility Each instance of the software can be tailored and adapted to a customer's needs. Customers may use completely different database schemas and it is straightforward to transfer data from a customer database to the product database.	**Cost** It is more expensive to use multi-instance systems because of the costs of renting many VMs in the cloud and the costs of managing multiple systems. Because of the slow startup time, VMs may have to be rented and kept running continuously, even if there is very little demand for the service.
Security Each customer has its own database so there is no possibility of data leakage from one customer to another.	**Update management** Many instances have to be updated so updates are more complex, especially if instances have been tailored to specific customer needs.
Scalability Instances of the system can be scaled according to the needs of individual customers. For example, some customers may require more powerful servers than others.	
Resilience If a software failure occurs, this will probably affect only a single customer. Other customers can continue working as normal.	

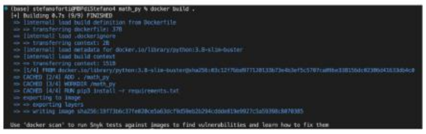
It is also possible to run containers on a virtual machine
a business could have its own VM-based system and run containers on top of this for individual users

* *What is the effect of docker build?*

The `docker build` command builds Docker images from a Dockerfile and a "context". A build's context is the set of files located in the specified `PATH` or `URL`

As easy as `docker build .`





```
(base) stefanofortiu@MBPdiStefano4 math_py % docker build .
[+] Building 0.7s (9/9) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 37B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/python:3.8-slim-buster
 => [internal] load build context
 => => transferring context: 151B
 => [1/4] FROM docker.io/library/python:3.8-slim-buster@sha256:[illegible]
 => CACHED [2/4] ADD . /math_py
 => CACHED [3/4] WORKDIR /math_py
 => CACHED [4/4] RUN pip3 install -r requirements.txt
 => exporting to image
 => => exporting layers
 => => writing image sha256:[illegible]

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
```

0

though Docker images and containers have a similar purpose, they have different uses. An image is a snapshot of an environment, and a container runs the software.

Both containers and images allow users to specify application dependencies and configurations and to describe everything necessary for a machine to run that application. However, containers and images have different lifecycles. For example, you can use containers, but not images, on container-based systems like Pivotal Cloud Foundry. Likewise, you can use images, but not containers, in non-container systems like Heroku or OpenShift.

An image is a snapshot of an environment, and a container runs the software. Both containers and images allow users to specify application dependencies and configurations and to describe everything necessary for a machine to run that application. However, containers and images have different lifecycles.

Is Dockerfile same as image?

The differences between images and containers

A Dockerfile is more like a recipe: instructions for how to build something. A compose file is more like a manual: instructions for how to use something. You use a Dockerfile to build an image. This image is the combination of an OS and your source code.

What is a Docker image vs container?

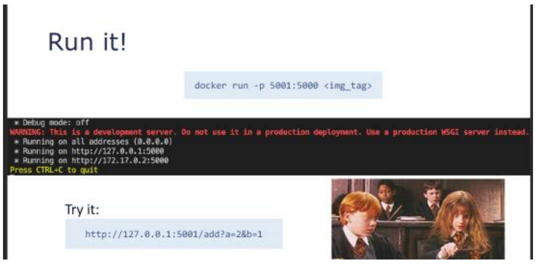
The key difference between a Docker image Vs a container is that a Docker image is a read-only immutable template that defines how a container will be realized. A Docker container is a runtime instance of a Docker image that gets created when the $ docker run command is implemented

*Docker Run

The `docker run` command must specify an *IMAGE* to derive the container from.

You can run containers from locally stored Docker images. If you use an image that is not on your system, the software pulls it from the online registry.

As an example, we used a Dockerfile to create a sample Docker image with the task to echo the message `Hello World`. For us, the image has the ID `e98b6ec72f51`. Your image name will differ depending on the container you want to run.



Run it!

```
docker run -p 5001:5000 <img_tag>
```

```
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://172.17.0.2:5000
Press CTRL+C to quit
```

Try it:

```
http://127.0.0.1:5001/add?a=2&b=1
```

Port local host : port container

*Docker tag

Docker tags are just **an alias for an image ID**

The Docker tag helps maintain the build version to push the image to the Docker Hub. **The Docker Hub allows us to group images together based on name and tag.**

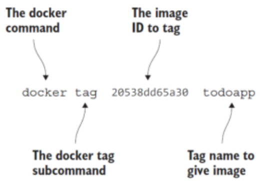
*Docker Commit

create a new image by committing the changes .

It can be useful to commit a container's file changes or settings into a new image. This allows you to debug a container by running an interactive shell, or to export a working dataset to another server.

Generally, it is better to use Dockerfiles to manage your images in a documented and maintainable way.

Tag an image



```
docker tag 20538dd65a30 todoapp
```

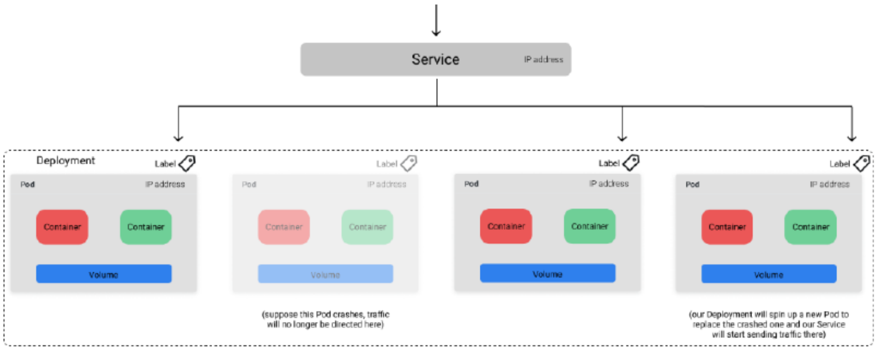
* *What is Docker Compose?*

Docker Compose is a tool that was developed to help define and share multi-container applications. With Compose, we can create a YAML file to define the services and with a single command, can spin everything up or tear it all down.

The *big* advantage of using Compose is you can define your application stack in a file, keep it at the root of your project repo (it's now version controlled), and easily enable someone else to contribute to your project. Someone would only need to clone your repo and start the compose app. In fact, you might see quite a few projects on GitHub/GitLab doing exactly this now.

*How does K8s control plane work?

Power



(suppose this Pod crashes, traffic will no longer be directed here)

(our Deployment will spin up a new Pod to replace the crashed one and our Service will start sending traffic there)

Two types of machines in a cluster:

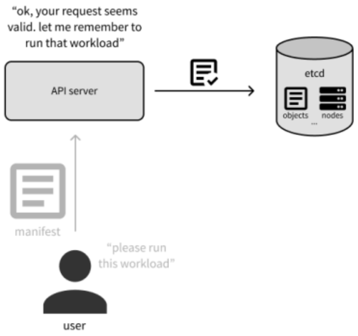
- **master node** - (often single) machine that contains most of the control plane components
- **worker node** - machine that runs the application workloads

(master node)

User provides new/updated object specification to **API server** of master node

- API server validates update requests and acts as unified interface for questions about cluster's current state
- State of cluster* is stored in a distributed key-value store **etcd**, a distributed keyvalue store

(*cluster configuration, object specifications, object statuses, nodes on the cluster, object-node assignments etc

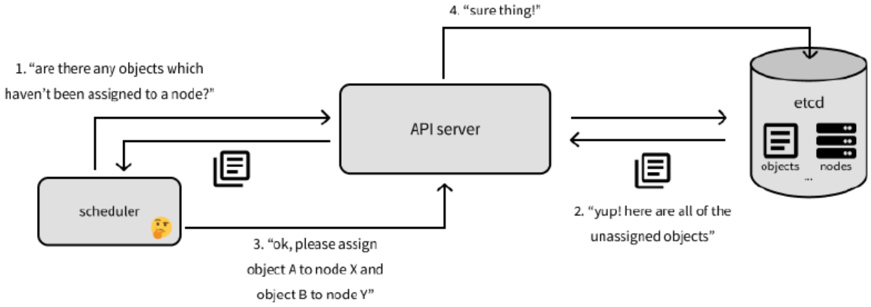

"ok, your request seems valid. let me remember to run that workload"
API server
etcd
objects
nodes
manifest
"please run this workload"
user

The **scheduler** determines where objects should be run

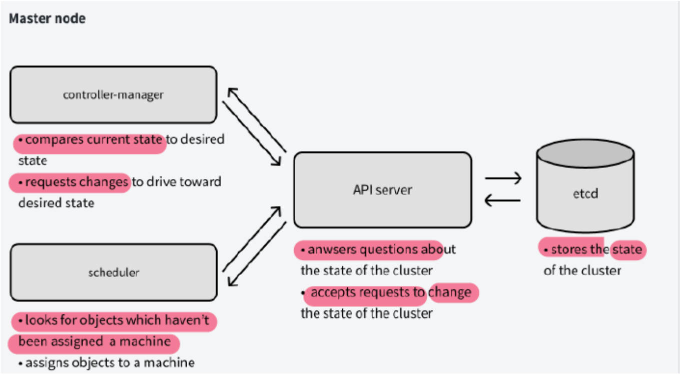
The scheduler

devote objects to the nodes(machine or workers)
in server store state of objects(cluster) we can found which obj(container) isn't allocated to nodes

- asks the API server which objects haven't been assigned to a machine
- determines which machines those objects should be assigned to
- replies back to the API server to reflect this assignment


4. "sure thing!"
1. "are there any objects which haven't been assigned to a node?"
API server
etcd
objects
nodes
scheduler
2. "yup! here are all of the unassigned objects"
3. "ok, please assign object A to node X and object B to node Y"

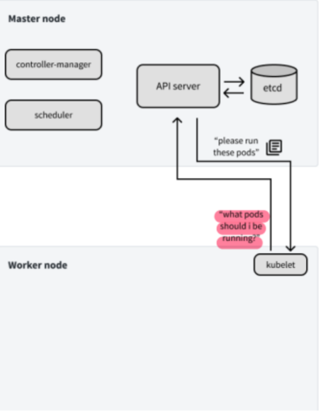
The **controller-manager** monitors cluster state through the API server
If actual state differs from desired state, the controller-manager will make changes via the API server to drive the cluster towards the desired state

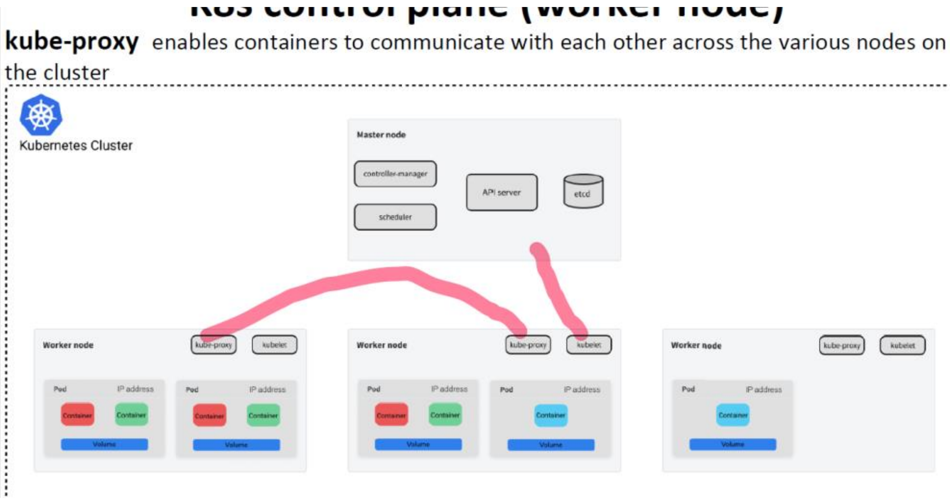

Master node
controller-manager
• compares current state to desired state
• requests changes to drive toward desired state
scheduler
• looks for objects which haven't been assigned a machine
• assigns objects to a machine
API server
• anwsers questions about the state of the cluster
• accepts requests to change the state of the cluster
etcd
• stores the state of the cluster

(worker node)

kubelet

- acts as a node's "agent" which communicates with the API server to see which container workloads have been assigned to the node
- responsible for spinning up pods to run these assigned workloads
- when a node first joins the cluster, kubelet announces the node's existence to the API server so the scheduler can assign pods to it





K8s control plane (worker node)

kube-proxy enables containers to communicate with each other across the various nodes on the cluster

** What is Minikube?*

Minikube is a lightweight Kubernetes implementation that creates a VM on your local machine and deploys a simple cluster containing only one node.

minikube quickly sets up a local Kubernetes cluster on macOS, Linux, and Windows.

Minikube is a tool that sets up a Kubernetes environment on a local PC or laptop. It's technically a Kubernetes distribution, but because it addresses a different type of use case than most other distributions (like Rancher, OpenShift, and EKS), it's more common to hear folks refer to it as a tool rather than a distribution.

Minikube supports all of the major operating systems – Windows, Linux, and macOS. (It doesn't run on mobile devices, alas.)

Commented [SG1]: How Does Minikube Work?
Minikube works by setting up a Kubernetes cluster on a local machine. By default, it creates a one-node cluster, but as we explain below, you can create a multi-node cluster with a Minikube environment if desired.
Minikube supports a flexible deployment mode based on the concept of "drivers." Depending on the driver you choose for deployment, your Kubernetes node, or nodes, can run in one of several ways:

- Inside a Docker container.
- Inside a virtual machine. Several VM platforms are supported, such as VirtualBox, KVM, Hyper-V, and Parallels, depending on which host operating system you use.
- Directly on bare metal, if your host OS supports it.

- The drivers available for your installation depend on which system you are using to run Minikube. You can only use Hyper-V VMs on Windows, for example, because Hyper-V only supports Windows. Likewise, the bare-metal driver option is only supported on Linux, since Kubernetes master nodes can run natively only on a Linux system.
- Which driver to choose depends on what level of performance and isolation you want from Minikube. You'll get the best performance if you deploy Minikube on bare metal. You'll get the next-best level of performance if you deploy it inside a container. Deploying in a VM gives you the lowest level of performance, with the upside being that your Minikube environment will be rigidly isolated from the host system.

* **Which are the main pros, cons and characteristics of microservices?**

Microservices are small-scale services that may be combined to create applications. They should be independent, so that the service interface is not affected by changes to other services. It should be possible to modify the service and re-deploy it without changing or stopping other services in the system.

microservices =
small-scale stateless services that have a single responsibility

- small-scale services that can be combined to create applications
- independent (service interface not affected by changes to other services)
- possible to modify and re-deploy service without changing/stopping other services

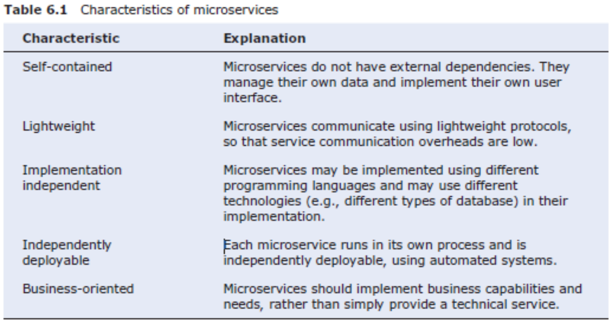


Table 6.1 Characteristics of microservices

Characteristic	Explanation
Self-contained	Microservices do not have external dependencies. They manage their own data and implement their own user interface.
Lightweight	Microservices communicate using lightweight protocols, so that service communication overheads are low.
Implementation independent	Microservices may be implemented using different programming languages and may use different technologies (e.g., different types of database) in their implementation.
Independently deployable	Each microservice runs in its own process and is independently deployable, using automated systems.
Business-oriented	Microservices should implement business capabilities and needs, rather than simply provide a technical service.

Motivations

(1) Shorten lead time for new features and updates

(2) Scale, effectively
Each microservice can be deployed in a separate container

⟶ quick stop/restart without affecting other services

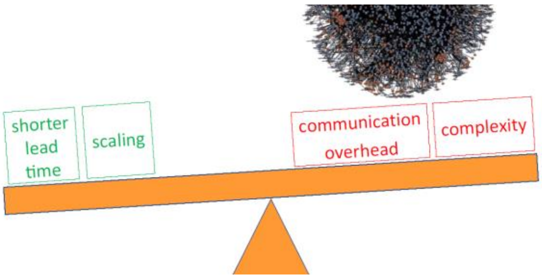
⟶ service replicas can be quickly deployed

- Many pros of microservices, including

- shorter lead time
- effective scaling

- Cons

- communication overhead
- complexity
- “wrong cuts”
- “avoiding data duplication as much as possible while keeping microservices in isolation is one of the biggest challenges”



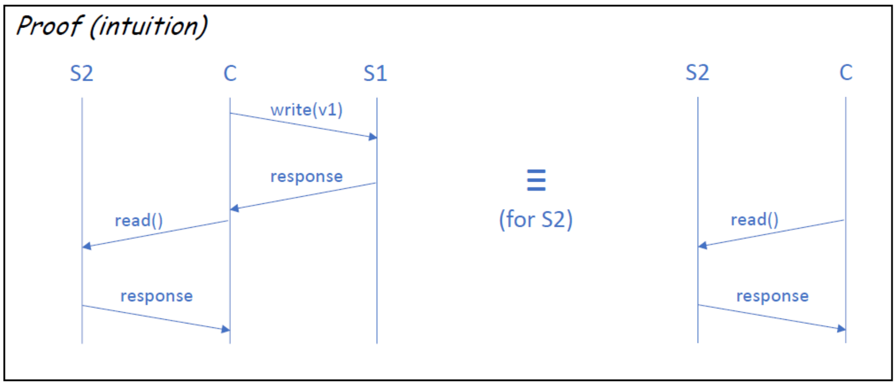
*What does the CAP theorem tell us?

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

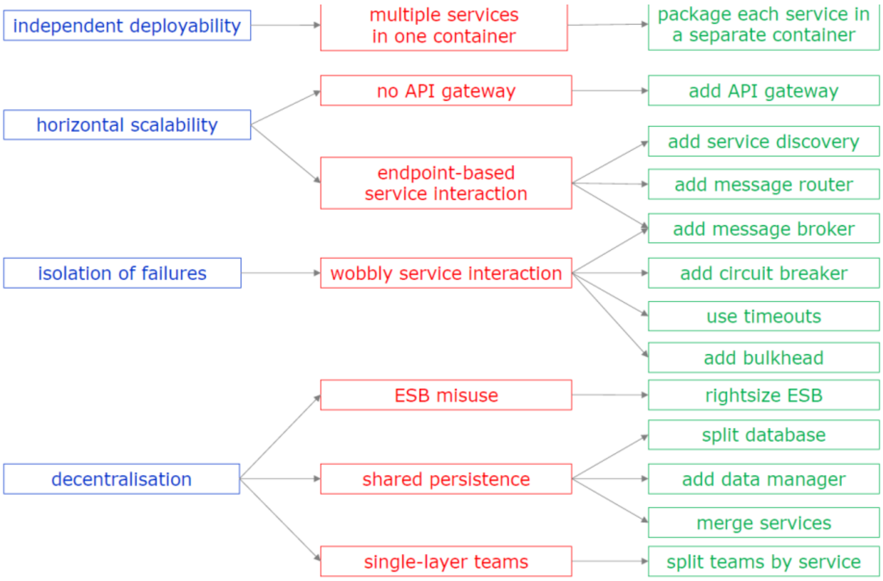
- *Consistency*: any read operation that begins after a write operation must return that value, or the
result of a later write operation

Commented [SG2]: Coupling measures number of inter-component relationships
Low coupling → independent services, independent updates
Cohesion measures number of intra-component relationships
High cohesion → less inter-service communication overhead
“Single responsibility principle”
• Each service should do one thing only and it should do it well
• Responsibility ≠ single functional activity
How big should a microservice be? “Rule of twos”
• Service can be developed, tested, and deployed by a team in two weeks
• Team can be fed with two large pizzas (8-10 people)
Why so many people for a microservice?
• implement service functionality
• develop code that makes service completely independent
• processing incoming and outgoing messages
• manage failures (service/interactions failures)
• manage data consistency when data are used by other services
• maintain service own interface
• test service and service interactions
• support service after deployment

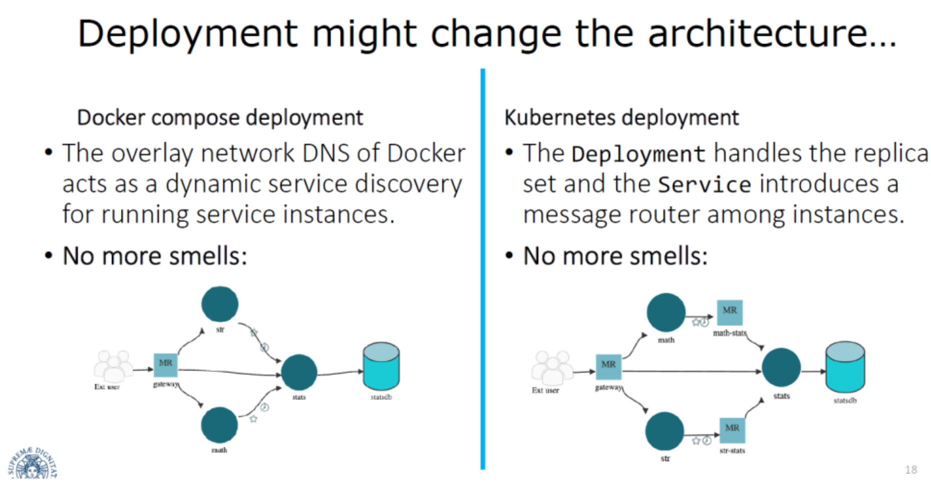
- *Availability*: every request received from a non-failing node must result in a response
- *Network partition*: network can lose arbitrarily many messages sent from one group to another



*** Which refactoring can be applied to resolve architectural smell X?**



****How can K8s or Swarm resolve architectural smells?***



Deployment might change the architecture…

Docker compose deployment

- The overlay network DNS of Docker acts as a dynamic service discovery for running service instances.
- No more smells:

Kubernetes deployment

- The `Deployment` handles the replica set and the `Service` introduces a message router among instances.
- No more smells:

*How can we feature authentication and authorization in a software product?

Authentication is the process of ensuring that a user of your system is who they claim to be. You need authentication in all software products that maintain user information so that only the providers of that information can access and change it. You also use authentication to learn about your users so that you can personalize their experience of using your product. Authentication in software products is based on one or more of three approaches—namely, user knowledge, user possession, and user attributes (Figure 7.4).

Knowledge-based authentication relies on users providing secret, personal information when registering to use the system. Each time a user logs on, the system asks for some or all of this information. If the information provided matches the registered information, the authentication is successful. Passwords are the most widely used method of knowledge-based authentication. An alternative, which is often used with passwords, is personal questions that the authenticating user must answer, such as “name of first school” or “favorite film.”

Possession-based authentication relies on the user having a physical device that can be linked to the authenticating system. This device can generate or display information that is known to the authenticating system. The user then inputs this information to confirm that they possess the authenticating device. The most commonly used version of this type of authentication relies on the user providing their mobile phone number when registering for an account. The authenticating system sends a code to the user’s phone number. The user has to input this code to complete the authentication. An alternative approach, which is used by some banks, is based on a special-purpose device that can generate one-time codes. The device calculates a code based on some aspect of the user input. The user inputs this code and it is compared with the code generated by the authenticating system, using the same algorithm as that encoded in the device.

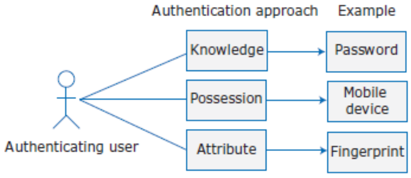
Attribute-based authentication is based on a unique biometric attribute of the user, such as a fingerprint, which is registered with the system. Some

Commented [SG3]: Knowledge-based authentication often employed for products delivered as cloud services
Weaknesses of password-based authentication
- Users choose ***insecure passwords*** that are easy to remember
- Users click on email link pointing to fake site that collects login and password → ***phishing attack***
- Users use the ***same password*** for several sites (if there is a security breach at one site …)
- Users regularly ***forget passwords*** → password recovery mechanism needed → potential vulnerability if credentials have been stolen

Defences
- Force users to set strong passwords
- Add knowledge-based authentication (e.g. user must answer questions)

mobile phones can authenticate in this way; others use face recognition for authentication. In principle, this is a very secure approach to authentication, but there are still reliability issues with the hardware and recognition software. For example, fingerprint readers often don't work if the user has hot, damp hands.

Figure 7.4 Authentication approaches



- The level of authentication that you need depends on your product
- No need to store confidential user information → knowledge-based authentication enough
- Need to store confidential user information → use two-stage authentication
- Implementing a secure and reliable authentication system is expensive and timeconsuming
- Even if using available toolkits and libraries (e.g. OAuth), there is still a lot of programming effort involved
- Authentication often outsourced with a federated identity system (see next)

7.3 Authorization

Authentication involves a user proving their identity to a software system. Authorization is a complementary process in which that identity is used to control access to software system resources. For example, if you use a shared folder on Dropbox, the folder's owner may authorize you to read the contents of that folder but not to add new files or overwrite files in the folder.
When a business wants to define the type of access that users get to resources, this is based on an access control policy. This policy is a set of rules that define what information (data and programs) is controlled, who has access to that information, and the type of access that is allowed (

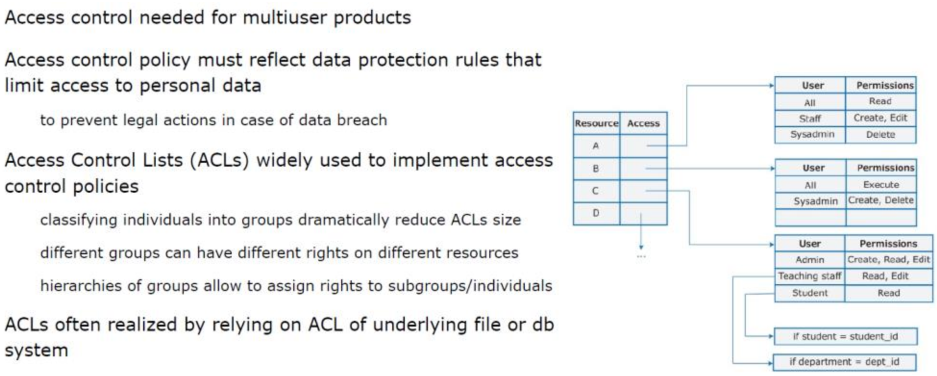
Commented [SG4]: For example, an access control policy may specify that nurses and doctors have access to all medical records stored on the system. Doctors may modify information on a record, but nurses may only add new information. Patients may read their own records and may issue a request for correction if they find what they believe is an error.

Authentication ≠ Authorization

ensure that user is who she claims to be

control that user can access resources

Events –rwxr–r--
Blog –rwxrwxrwx



Access control needed for multiuser products

Access control policy must reflect data protection rules that limit access to personal data

- to prevent legal actions in case of data breach

Access Control Lists (ACLs) widely used to implement access control policies

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

ACLs often realized by relying on ACL of underlying file or db system

Resource	Access
A	
B	
C	
D	

User	Permissions
All	Read
Staff	Create, Edit
Sysadmin	Delete

User	Permissions
All	Execute
Sysadmin	Create, Delete

User	Permissions
Admin	Create, Read, Edit
Teaching staff	Read, Edit
Student	Read

if student = student_id

if department = dept_id

* Which are the main challenges in securing microservices?

1 **The broader the attack surface, the higher the risk**

Monolith: inter-component communications within single process
Microservices:
- inter-service communication via remote calls
- large number of entry points, **attack surface broadens**
- security of app = strength of weakest link

2 **Distributed security screening affects performance**

Monolith: security screening done once, request dispatched to corresponding component
Each microservice has to carry out independent security screening
- may need to connect to a remote security token service
- repeated, distributed security checks affect **performance**

Work around: trust-the-network (and skip security checks at each microservice)
Industry trend: trust-the-network zero-trust networking principles
Still, overall performance must be taken into consideration

3 **Bootstrapping trust among microservices needs automation**

Service-to-service communication must take place on protected channels
Suppose that you use certificates:
- Each microservice must be provisioned with a certificate (and corresponding private key) to authenticate itself to another microservice during service-to-service interactions
- Recipient microservice must know how to validate the certificate associated with calling microservice
- Need a way to **bootstrap trust** between microservices
- (Need also to revoke and rotate certificates)

®To manage large-scale deployments of hundreds of microservices, **automation** is needed

4 **Tracing requests spanning multiple microservices is challenging**

A *log* is the recording of an event of a service
Logs can be aggregated to produce *metrics* that reflect the system state (e.g. average invalid access requests per hour) and that may trigger alerts
Traces help you track a request from the point where it enters the system to the point where it leaves the system

Unlike in monolithic applications, a request to a microservices deployment
may span multiple microservices

→ **correlating requests among microservices is challenging**

Tools

- e.g. Prometheus and Grafana to monitor incoming requests
- e.g. Jaeger and Zipkin for distributed tracing

5 Containers complicate credentials/policies handling

Containers are **immutable servers**, they don't change state after spin up

great to simplify deployment and to achieve horizontal-scalability

But for each service we need to maintain a dynamic list of allowed clients and a dynamic set of access-control policies

e.g. get updated policies from some policy admin endpoint (with a push or pull model)

Each service must also maintain its own credentials, which need to be rotated periodically

e.g. keep credentials in container filesystem and inject them at boot time

6 Distribution makes sharing user context harder

User context has to be passed explicitly from one microservice to another

Challenge: Build trust between microservices so that receiving microservice accepts user context passed from calling microservice

Popular solution: Using JSON Web Token (JWT) to share user context among microservices

Idea: messages carrying user attributes in a cryptographically safe way

7 Security responsibilities distributed among different teams

Polyglot architecture: Different squads can use different technology stacks

Different teams can use different security best practices, and different tools for static code analysis and dynamic testing

Security responsibilities distributed across different teams

Organizations often adopt hybrid approach, with centralized security team and security experts in the teams

**What is static/dynamic vulnerability analysis?*

Two types of testing:

- Static (bandit-like)
- Dynamic (owasp ☺)

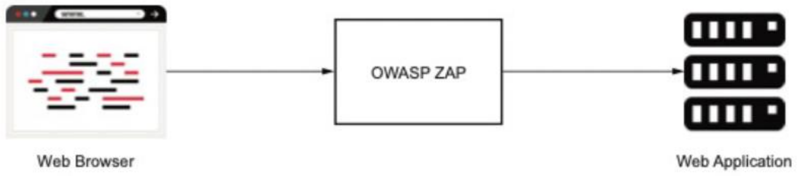
Bandit => Secure Programming

- Bandit is a static analysis tool designed to find common security issues in Python code, by exploiting known patterns (plugins).
- Bandit was originally developed within the OpenStack Security Project and later re-homed to PyCQA

- It checks code while it executes.
- It generates various types of input parameters to trigger as

many execution flows as
possible.

- OWASP ZAP is a tool for dynamic analysis that helps finding vulnerabilities in running web apps.
- ZAP acts as a proxy between the client app and the server.



**Which are the most frequent API security vulnerabilities and how can we prevent them?*

Power :owasp-api-security-top-10-cheat-sheet-a4

Q4

*What is DevOps automation?

Historically, the processes of integrating a system from independently developed parts, deploying that system in a realistic testing environment, and releasing it were time-consuming and expensive. By using DevOps with automated support, however, you can dramatically reduce the time and costs for integration, deployment, and delivery.
“Everything that can be should be automated” is a fundamental principle of DevOps. In addition to reducing the costs and time required for integration, deployment, and delivery, automation makes these processes more reliable and reproducible.

“Everything that can be should be automated”

Continuous integration

Each time a developer commits a change to the project’s master branch, an executable version of the system is built and tested.

Continuous delivery

Executable software is tested in a simulated product’s operating environment.

Continuous deployment

New release of system made available to users every time a change is made to the master branch of the software.

Infrastructure as code

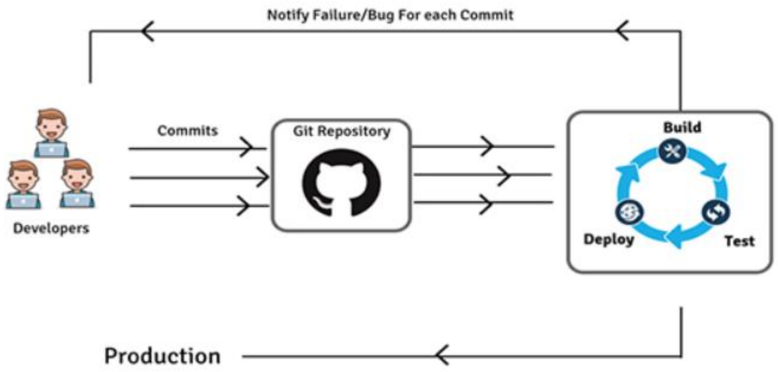
Machine-readable models of infrastructure (network, servers, routers, etc.) on which the product executes are used by configuration management tools to build the software’s execution platform. The software to be installed, such as compilers and libraries and a DBMS, are included in the infrastructure model.

** What is Jenkins?*

Book :Continuous integrationtools such as Jenkins can collect data about deployments, successful tests, and so on.
CI tools such as Jenkins and Travis may also be used to support continuous
delivery and deployment. These tools can integrate with infrastructure configuration
management tools such as Chef and Puppet to implement software
deployment
Power:
CI tools such as Jenkins and Travis may also be used to support
continuous delivery and deployment.
These tools can integrate with infrastructure configuration
management tools such as Chef and Puppet and Ansible to
implement software deployment.
For cloud-based software, it is often simpler to use containers in
conjunction with CI tools rather than use infrastructure
configuration management software.
search:
The leading open source automation server, Jenkins provides hundreds of plugins to support building, deploying and automating any project
jenkins is an open source automation server. It helps automate the parts of software development related to building, testing, and deploying, facilitating continuous integration and continuous delivery.
Continuous Integration and Continuous Delivery :As an extensible automation server, Jenkins can be used as a simple CI server or turned into the continuous delivery hub for any project.

به منظور ادغام مداوم نوشته شده است. جنکینز برای جنکینز ابزاری برای اتوماسیون است که در زبان برنامه نویسی جاوا ساخت و آزمایش پروژه های نرم افزاری شما بطور مداوم مورد استفاده قرار می گیرد و باعث می شود تا برنامه نویسان بتوانند تغییرات را در پروژه اعمال کند و همچنین دستیابی به بیلد جدید را برای کاربران آسان تر کند. همچنین به توسعه و استقرار، نرم افزار خود را به طور مداوم تحویل تست دهندگان امکان می دهد تا با ادغام با تعداد زیادی از فناوری های بر روی Jenkins. استفاده می شود DevOps است که برای ادغام مراحل مختلف DevOps دهید. ادغام مداوم مهمترین بخش سروری نصب می شود که ساخت مرکزی در آن انجام خواهد شد.

ادغام Jenkins .، سازمان ها می توانند فرایند توسعه نرم افزار را از طریق اتوماسیون به سرعت افزایش دهندJenkins با فرآیندهای چرخه عمر توسعه هر نوعی را که شامل انواع ساخت، سند، تست، بسته، مرحله، استقرار، تجزیه و تحلیل استاتیک و غیره می شود را انجام می دهد. ادغام مداوم را به کمک پلاگین ها انجام می دهد. اگر می خواهید یک ابزار خاص را ادغام کنید، باید پلاگین هایی برای این ابزار نصب کنید. به عنوان مثال: گیت، Maven 2 project ،Amazon EC2، HTML publisher و غیره.



جنکینز با کمک بیش از 1000 پلاگین، به ادغام مداوم دست یافته است، که به شما این امکان را می‌دهد بطور مداوم ساخت، آزمایش و استقرار انجام دهید. جنکینز می‌تواند به مؤسساتی که روند توسعه سریع و چرخه زندگی را دنبال می‌کنند، کمک کند. اتوماسیون در اصل شامل یک تجزیه و تحلیل استاتیک می‌باشد و نیاز به تعمیر و نگهداری کمی دارد به خاطر خودکار شدن عملیات، و دارای یک رابط کاربری گرافیکی داخلی برای بروزرسانی آسان است. این گونه کیفیت نرم افزار بهبود یافته است

***’’’*How does Jenkins exploit Git?*

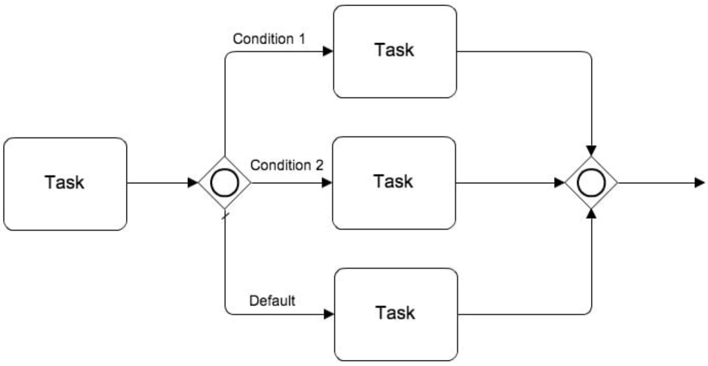
```
git clone https://github.com/jenkins-docs/simple-python-pyinstaller-app.git
git add .
git commit -m "Add initial Jenkinsfile"
git add .
git commit -m "Add ‘Test’ stage"
git add .
git commit -m "Add ‘Deliver’ stage"
```

*What is a parallel/exclusive/inclusive gateway in BPMN?

inclusive gateway

Split: one or more branches are activated depending on formula in each flow
Join: all active input branches must be Completed

When used as a diverging gateway (splitting the sequence flow into many paths), an inclusive gateway can have 2 or more outgoing paths. It is used to direct the sequence flow along all paths where the condition evaluates to “True”. When using an inclusive gateway ONE, SEVERAL, OR ALL paths can be taken for a given instance of a process. Each condition is checked and if it evaluates to true a token is sent down the path. With an inclusive gateway, even after a condition evaluates to “True” the rest of the conditions are checked to see if other tokens should be released along the other paths.



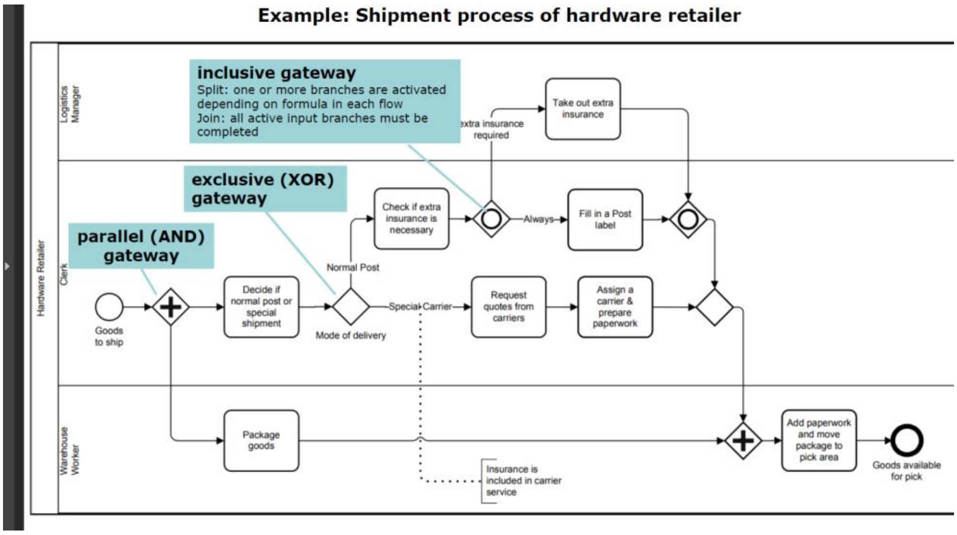
Often, there is one path which is identified as the default path. This means that if no other path condition evaluates to true then the default path will automatically proceed. If the process modeler fails to define a default path and none of the path conditions evaluate to true then a runtime exception occurs.

When used as a converging gateway, an inclusive gateway will wait for all tokens to arrive before merging and synchronizing the flow. Each converging inclusive gateway needs to have an associated diverging gateway earlier within the process such that it is aware of which paths were activated and the tokens it should be waiting for.

Exclusive Gateway

An exclusive gateway is used to express that exactly one alternative can be selected. Customers can pay either with credit card or direct debit, but not with both

A *parallel gateway* starts parallel work—two (or more) sequence flows that then progress at the same time, perhaps to be later joined back together by another parallel gateway.


Example: Shipment process of hardware retailer
inclusive gateway
Split: one or more branches are activated depending on formula in each flow
Join: all active input branches must be completed
exclusive (XOR) gateway
parallel (AND) gateway
Hardware Retailer
Logistics Manager
Clerk
Warehouse Worker
Goods to ship
Decide if normal post or special shipment
Mode of delivery
Normal Post
Special Carrier
Check if extra insurance is necessary
extra insurance required
Take out extra insurance
Always
Fill in a Post label
Request quotes from carriers
Assign a carrier & prepare paperwork
Package goods
Insurance is included in carrier service
Add paperwork and move package to pick area
Goods available for pick

*What is a workflow net?

Idea: Enhance Petri nets with concepts and notations that ease the representation of business processes

Like Petri nets, workflow nets focus on the *control flow* behaviour of a process:

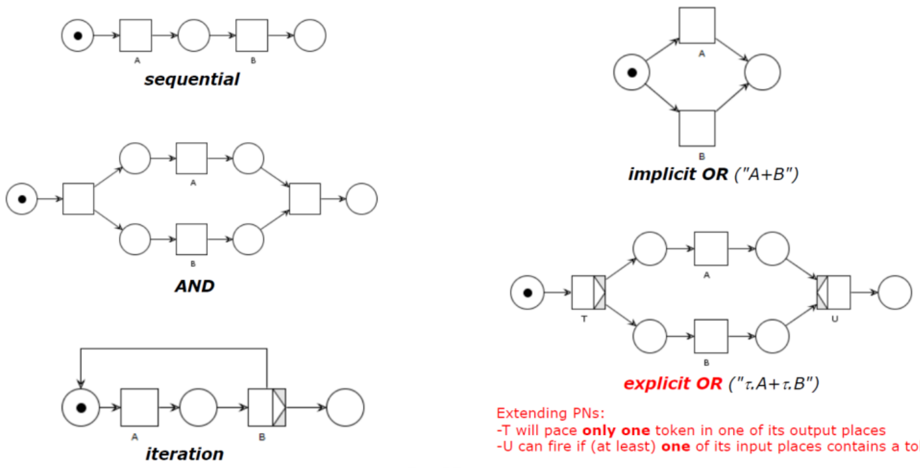
- transitions represent activities
- places represent conditions
- tokens represent process instances

A Petri net is a ***workflow net*** iff

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

Extension of **Petri nets** (see next)
One of the best known techniques for specifying business processes in a *formal* and *abstract* way
+ Graphical representation eases communications between different stakeholders
+ Process properties can be formally analysed
+ Various supporting tools are available

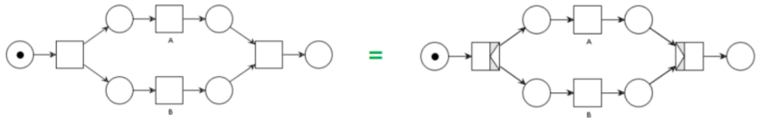
Composition patterms



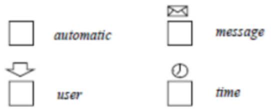
Extending PNs:
-T will pace **only one** token in one of its output places
-U can fire if (at least) **one** of its input places contains a token

Workflow nets

Equivalent "sugared" representation of AND-split and AND-join transitions



Transitions can be annotated with *triggers*, to denote who/what is responsible for an enabled transition to fire



*What is a sound/live/bounded net?

A workflow net is ***sound*** iff

(1)every net execution starting from the *initial state* (one token in the source place, no tokens elsewhere)

eventually leads to the *final state* (one token in the sink place, no tokens elsewhere), and
(2)every transition occurs in at least one net execution

Soundness (cont.)

Soundness is a notion of correctness of workflow nets

Situations to Avoid

Unboundness

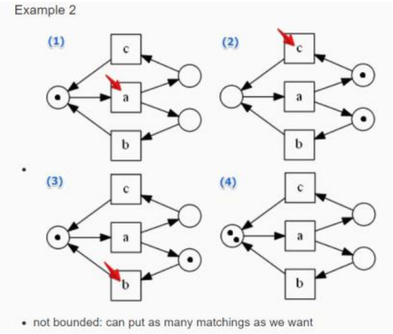
Unboundness means:

- there is no bound on the number of tokens that a place can hold
- this always means a problem

Liveness

In a *live* petri net there are no dead transitions

- a dead transition is a transition that can never fire in any marking reachable from the initial marking

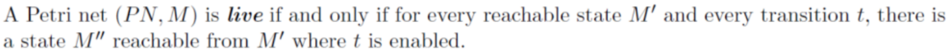


Example 2

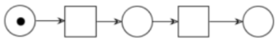
- not bounded: can put as many matchings as we want

http://www.mlwiki.org/index.php/Workflow_Soundness

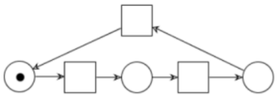
How to formally (and automatically) establish whether a net is sound?



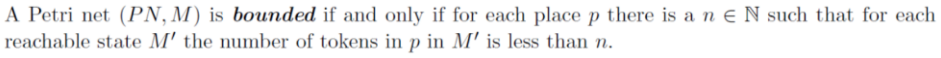
A Petri net (PN, M) is ***live*** if and only if for every reachable state M' and every transition t, there is a state M'' reachable from M' where t is enabled.



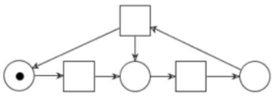
not live



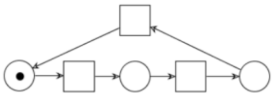
live



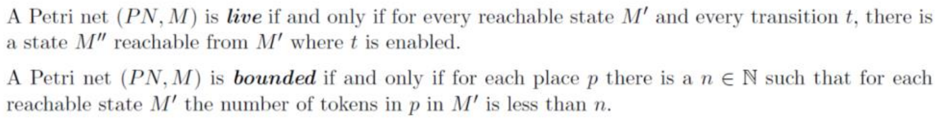
A Petri net (PN, M) is ***bounded*** if and only if for each place p there is a $n \in \mathbb{N}$ such that for each reachable state M' the number of tokens in p in M' is less than n.



not bounded

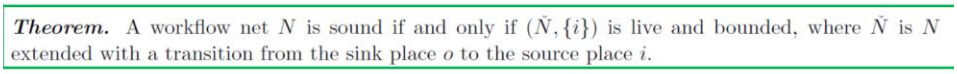


bounded



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

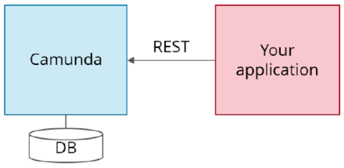
A Petri net (PN, M) is ***bounded*** if and only if for each place p there is a $n \in \mathbb{N}$ such that for each reachable state M' the number of tokens in p in M' is less than n.



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

**What is Camunda?*

- Camunda is a framework supporting BPMN for workflow and process automation.
- It provides a RESTful API which allows you to use your language of choice.
- Workflows are defined in BPMN which can be graphically modeled using the Camunda Modeler.



****Which are the two “usage patterns” of Camunda?***

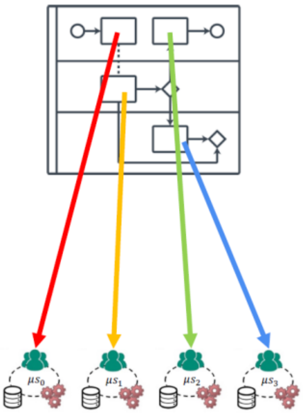
(A) aka endpoint-based integration
(B) aka queue-based integration

(A) endpoint-based integration

How does it work? (Pattern A)

After defining a BPMN process, Camunda can directly call services via built-in *connectors*.

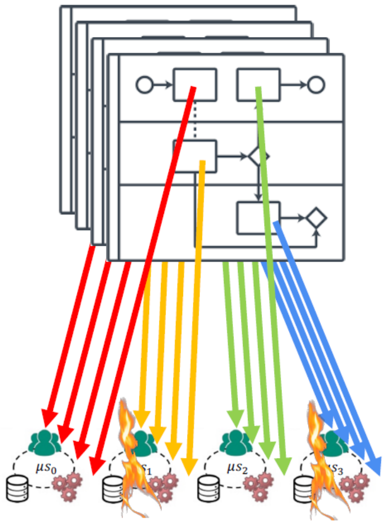
It supports both RESTful and SOAP services in this way.

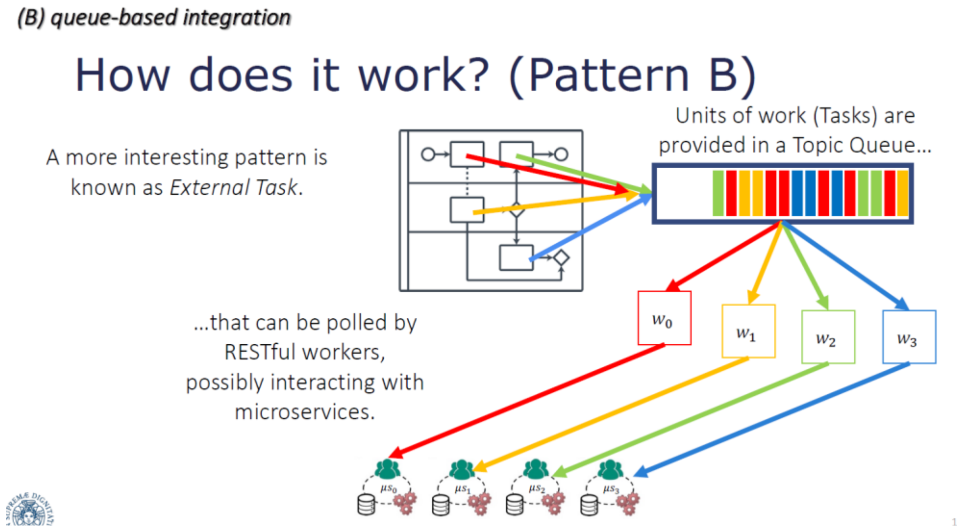


(A) endpoint-based integration

Scaling (Pattern A)

However, it only allows scaling on process instances, NOT on microservices.





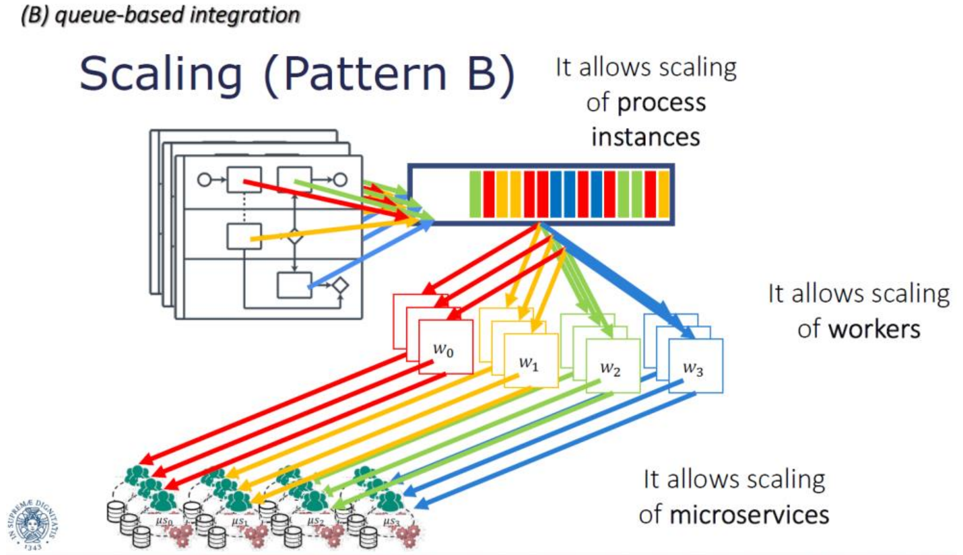
(B) queue-based integration

How does it work? (Pattern B)

A more interesting pattern is known as *External Task*.

Units of work (Tasks) are provided in a Topic Queue…

…that can be polled by RESTful workers, possibly interacting with microservices.



(B) queue-based integration

Scaling (Pattern B)

It allows scaling of process instances

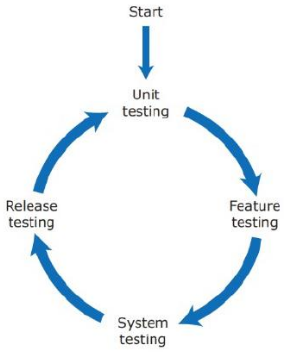
It allows scaling of workers

It allows scaling of microservices

---

* What is functional testing?

Large set of program tests so that all code is executed at least once
Testing should start on the day you start writing code
Develop/test cycle simplified by automated tests
Functional testing is a staged activity



Unit testing

Test program units (e.g. function, method) in isolation

Unit testing principle

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

Example

If your program behaves correctly on the input set {1, 5, 17, 45, 99}, you may conclude that it will also process all other integers in the range 1 to 99 correctly

Unit testing: Equivalence partitions

Identify “equivalence partitions”: sets of inputs that will be treated the same in your code

Test your program using several inputs from each equivalence partition

Programmers make mistakes at boundaries: Identify partition boundaries and choose inputs at these boundaries

Feature testing

A product feature implements some useful user functionality

Features must be tested to show that functionality

(1) is implemented as expected and

(2) meets the real needs of users

Feature normally implemented by multiple interacting program units

Two types of tests

(1) Interaction tests

Testing interactions between units (developed by different developers)

Can also reveal bugs in program units that were not exposed by unit testing

(2) Usefulness tests

Testing that feature implements what users are likely to want

Product manager should be involved in designing usefulness tests

System testing

Testing the system as a whole

- To discover unexpected/unwanted interactions between the features
- To discover if system features work together effectively to support what users really want to do
- To make sure system operates as expected in the different environments where it will be used

- To test responsiveness, throughput, security, and other quality attributes

Tip: Use a set of scenarios/user stories to identify users’ end-to-end
Pathways

Release testing

Testing a system that is intended for release to customers

- Release testing tests the system in its real operational environment (rather than in a test environment)

- The aim of is to decide if the system is good enough to release, not to detect bugs in the system

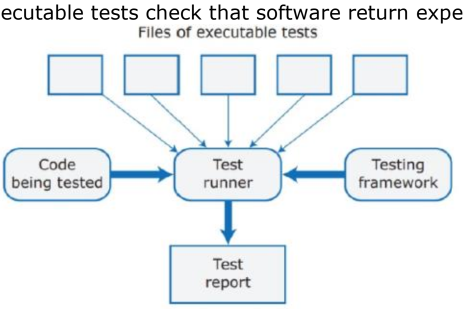
Needed since preparing a system for release involves packaging the system for deployment, installing used software and libraries, configure parameters – and mistakes can be made in that process
If you deploy on the cloud, an automated continuous release process can be used

* What is test automation?

One of the most significant innovations in agile software engineering is automated testing, which is now widely used in product development companies. Automated testing (Figure 9.4) is based on the idea that tests should be executable. An executable test includes the input data to the unit that is being tested, the expected result, and a check that the unit returns the expected result. You run the test and the test passes if the unit returns the expected result. Normally, you should develop hundreds or thousands of executable tests for a software product.

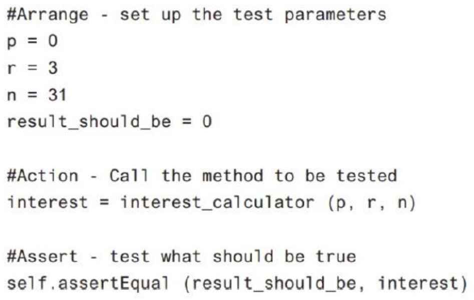
Test automation

Automated testing widely used in product development companies
Testing frameworks available for all widely used programming languages
Executable tests check that software return expected result for input data



Good practice: Structure automated tests in three parts

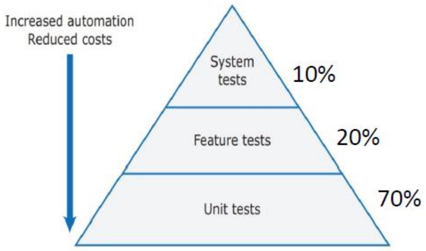
1. Arrange - Set up the system to run the test (define test parameters, mock objects if needed)
2. Action - Call the unit that is being tested with the test parameters
3. Assert – Assert what should hold if test executed successfully.

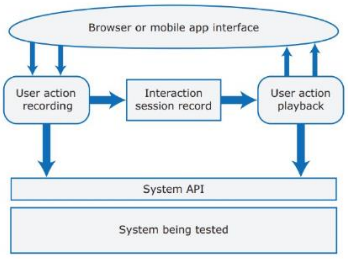


```
#Arrange - set up the test parameters
p = 0
r = 3
n = 31
result_should_be = 0

#Action - Call the method to be tested
interest = interest_calculator (p, r, n)

#Assert - test what should be true
self.assertEqual (result_should_be, interest)
```

Test code can include bugs!
Good practice to reduce the chances of test errors:
1. Make tests as simple as possible
2. Review all tests along with the code that they test

Unit tests are the easiest to automate
Good unit tests reduce (do not eliminate) need of feature tests
GUI-based testing expensive to automate, API-based testing preferrable
Need multiple assertions to check that feature executed as expected
System testing involves testing system as a surrogate user
Perform sequences of user actions
Manual system testing boring and error prone
Testing tools record series of actions and automatically replay them





*****Which are the patterns of unit/load tests?***

*What is Locust?

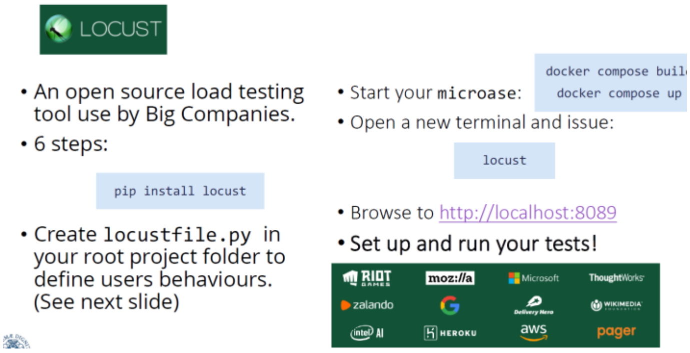
Locust is **a Python-based testing tool used for load testing and user behavior simulation**. Load testing is the practice of testing a software application with the primary purpose of stressing the application's capabilities. Locust is a tool that creates a set of testing functions that simulate a heavy number of users.١٠ مارس ٢٠٢٢

**How can you use it to identify bottlenecks?*

Locust is an easy-to-use, distributed, user load testing tool. It is intended for load-testing web sites (or other systems) and figuring out how many concurrent users a system can handle.

The idea is that during a test, a swarm of locust users will attack your website. The behavior of each user is defined by you using Python code, and the swarming process is monitored from a web UI in real-time. This will help you battle test and identify bottlenecks in your code before letting real users in.

Locust is completely event-based, and therefore it’s possible to support thousands of concurrent users on a single machine. In contrast to many other event-based apps it doesn’t use callbacks. Instead it uses light-weight processes, through gevent. Each locust swarming your site is actually running inside its own process (or greenlet, to be correct). This allows you to write very expressive scenarios in Python without complicating your code with callbacks.


LOCUST
RIOT GAMES
moz://a
Microsoft
ThoughtWorks
zalando
G
Delivery Hero
WIKIMEDIA
intel AI
HEROKU
aws
pager

- An open source load testing tool use by Big Companies.
- 6 steps:

```
pip install locust
```

- Create `locustfile.py` in your root project folder to define users behaviours. (See next slide)
- Start your `microase`:

```
docker compose buil
docker compose up
```

- Open a new terminal and issue:

```
locust
```

- Browse to http://localhost:8089
- Set up and run your tests!

***Which are the problems of and solutions for application management in the Cloud-Edge Continuum?**

Deploying composite applications in a QoS- and context-aware manner on the Cloud-Edge Continuum is challenging ...

App requirements

- Hardware
- Software
- QoS
- ...

Infrastructure

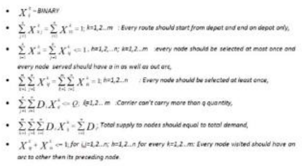
- Heterogeneous
- Large
- Dynamic*

Need **tools** helping to master orthogonal dimensions

Problem #1: How to suitably **place** a composite application on the Cloud-Edge Continuum

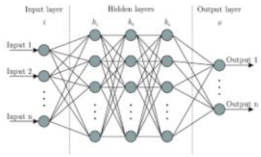
Different approaches

MILP



- X_{ij}^{k} - BINARY
- $\sum_{j=1}^{n} X_{1j}^{k} = \sum_{i=1}^{n} X_{i1}^{k} = 1$; k=1,2...m : Every route should start from depot and end on depot only,
- $\sum_{i=1}^{n} X_{ih}^{k} = \sum_{j=1}^{n} X_{hj}^{k} <= 1$, h=1,2,...n; k=1,2...m :every node should be selected at most once and every node served should have a in as well as out arc,
- $\sum_{k=1}^{m} \sum_{j=1}^{n} X_{ij}^{k} = \sum_{k=1}^{m} \sum_{i=1}^{n} X_{ih}^{k} = 1$; h=1,2...n : Every node should be selected at least once,
- $\sum_{j=1}^{n} \sum_{i=1}^{n} D_i X_{ij}^{k} <= Q$; k=1,2... m :Carrier can't carry more than q quantity,
- $\sum_{k=1}^{m} \sum_{j=1}^{n} \sum_{i=1}^{n} D_i X_{ij}^{k} = \sum_{i=1}^{n} D_i$; Total supply to nodes should equal to total demand,
- $X_{ij}^{k} + X_{ji}^{k} <= 1$; for i,j=1,2...n; h=1,2...n for every k=1,2...m: Every node visited should have an arc to other then its preceding node.

ML



Declarative

"service S can be placed on node N if ..."

Hard to read

Slow to run

Infrastructure is very dynamic

Explainability

Declarative approach

1) Declare what an eligibile placement is

service S can be placed on node N ***if***

the hardware reqs of S are met by N ***and***

the IoT connection reqs of S are met by N ***and***

the software reqs of S are met by N

2) Let the inference engine look for it!

services S1 ... Sm can be placed on nodes N1 ... Nm ***if***

service S1 can be placed on node N1 ***and***

... ***and***

service Sm can be placed on node Nm ... ***and***

the QoS reqs of S1 ... Sm are met

Problem #2: How to suitably **manage** application deployments in the Cloud-Edge Continuum (after first deployment)

Continuous Reasoning

Exploit compositionality to differentially analyse a large-scale system:
- by mainly **focussing on the latest changes** introduced in the system, and
- by **re-using previously computed results** as much as possible

Other aspects

+ probabilities to model **infrastructure dynamicity**
+ semirings to model (non-monotonic,
conditionally transitive) **trust** relations
among different stakeholders

+ **infrastructure monitoring** and **management enactment**

***** What is an explainable failure root cause analysis?

Questions on the syllabus1

1. What is product-based software engineering?
2. What is the incremental development and delivery advocated by Agile?
3. Which are the key Scrum practices?
4. What are personas, scenarios, user stories and features?
5. *What is the effect of a git add/branch/clone/checkout/push/commit/pull command?*
6. *How does GitHub flow work?*
7. What is the role of non-functional quality attributes and decomposition in a software architecture?
8. What is a distribution architecture?
9. Which are the technology choices that affect a software architecture?
10. Which are the main features of Enterprise Integration Patterns?
11. *What is a Camel route (for)?*
12. Which are the differences between multi-tenant and multi-instance SaaS systems?
13. *What is the effect of docker build/run/commit/tag? What is Docker Compose?*
14. How does K8s control plane work?
15. *What is Minikube?*
16. Which are the main pros, cons and characteristics of microservices?
17. What does the CAP theorem tell us?
18. Which refactoring can be applied to resolve architectural smell X?
19. *How can K8s or Swarm resolve architectural smells?*
20. How can we feature authentication and authorization in a software product?
21. Which are the main challenges in securing microservices?
22. *What is static/dynamic vulnerability analysis?*
23. *Which are the most frequent API security vulnerabilities and how can we prevent them?*
24. What is DevOps automation?
25. *What is Jenkins?*
26. *How does Jenkins exploit Git?*
27. What is a parallel/exclusive/inclusive gateway in BPMN?
28. What is a workflow net?
29. What is a sound/live/bounded net?
30. *What is Camunda?*
31. *Which are the two “usage patterns” of Camunda?*
32. What is functional testing?
33. What is test automation?
34. *Which are the patterns of unit/load tests?*
35. *What is Locust?*
36. *How can you use it to identify bottlenecks?*

37. Which are the problems of and solutions for application management in the Cloud-Edge Continuum?

38 .What is an explainable failure root cause analysis?