Cloud-based software

Antonio Brogi

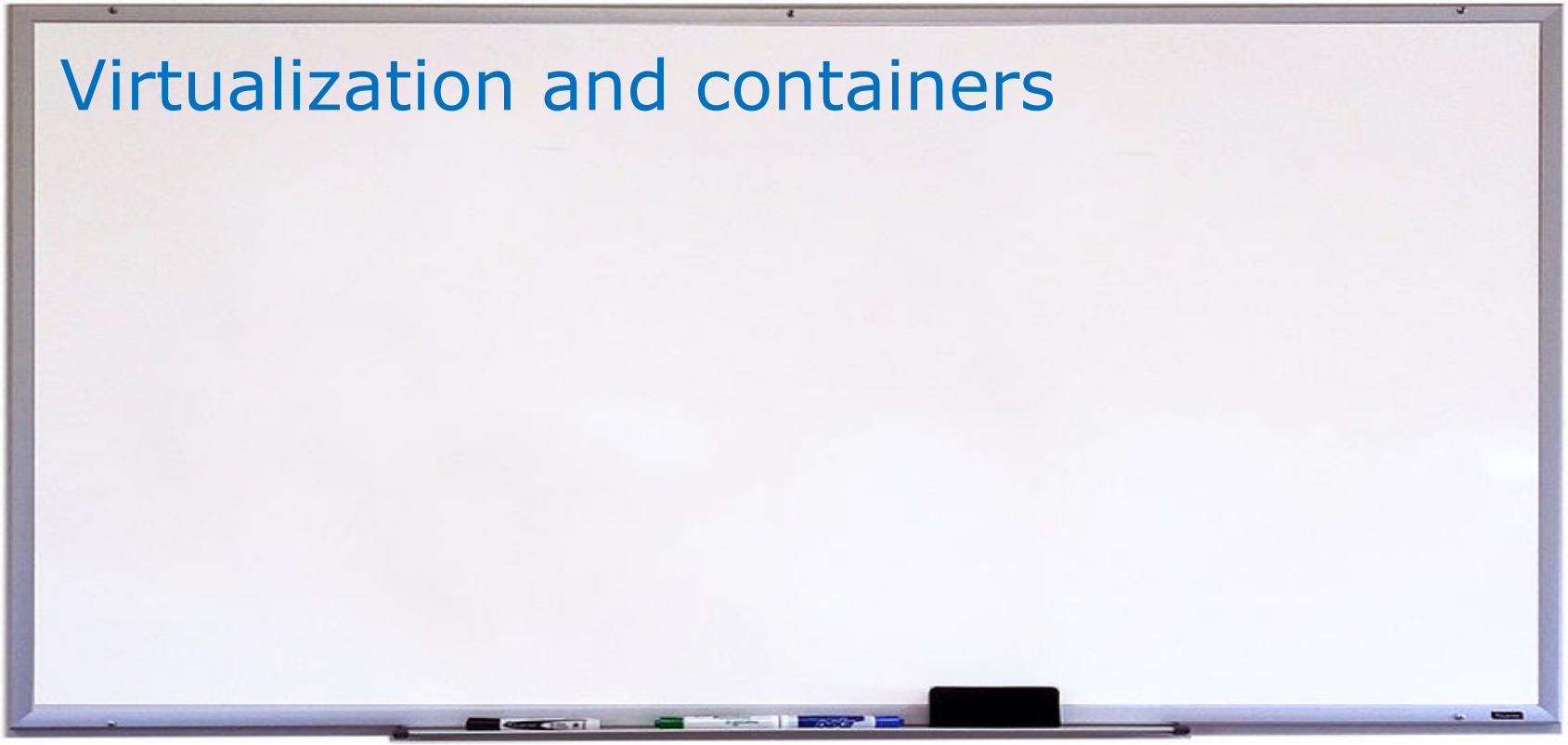
Department of Computer Science
University of Pisa

Cloud computing

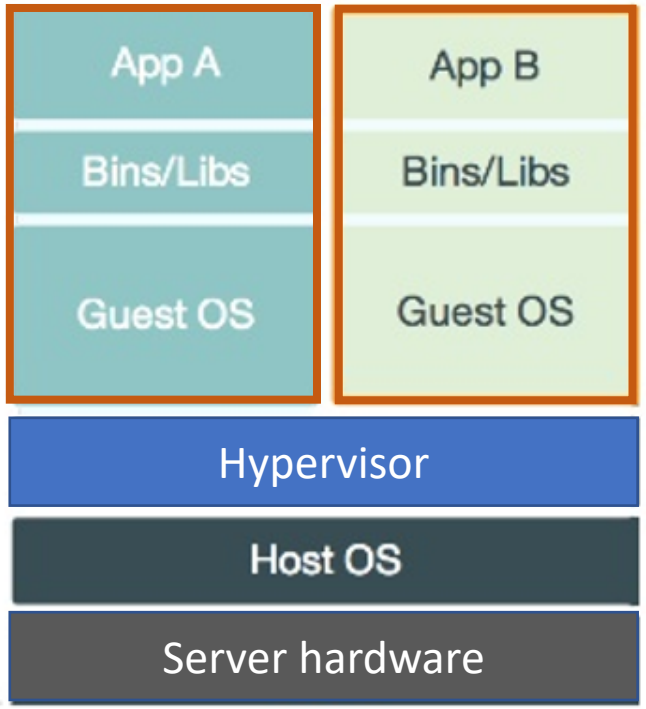
- Powerful computer hardware + high-speed networking have led to the development of cloud computing
- Virtualised resources (e.g. compute, storage, platform) accessible on demand

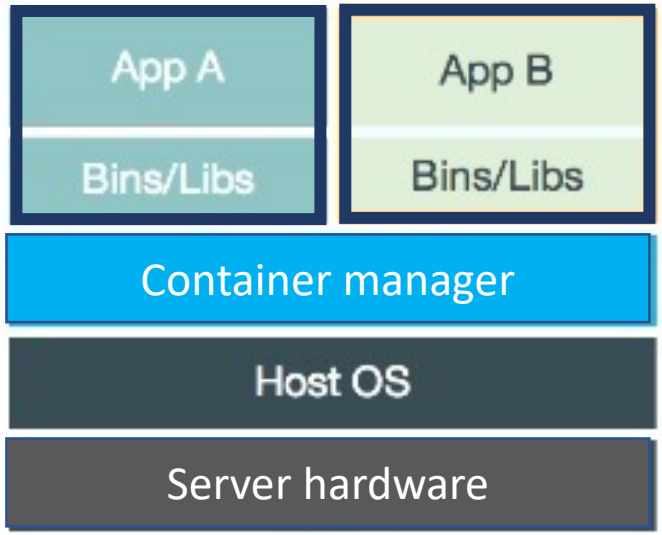
- Main advantages
  - Scalability (maintain performance as load increases)
  - Elasticity (adapt server configuration to changing demands)
  - Resilience (maintain service in the event of server failures)
  - Cost (CapEx reduction, pay-per-use)

Virtualization and containers



From VMs to containers





- Server virtualization
- Many virtual machines (each with its OS) running on same physical server
- ("Type 1" hypervisor loaded directly on HW)

Key idea of containers: Exploit OS kernel's capability of allowing multiple isolated user-space instances

+ lighter (require less resources, from Gb to Mb)

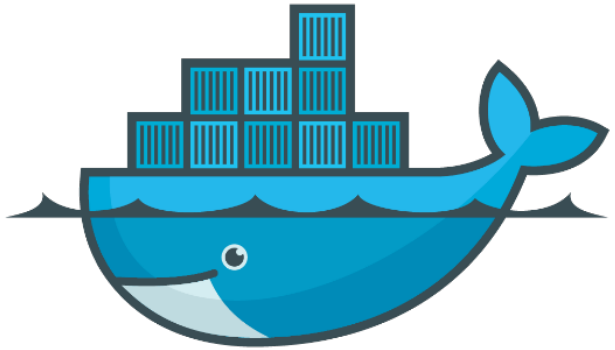
+ faster to start (from mins to secs)

- share same OS

What is Docker?

Docker is a platform that allows us
to run applications in an isolated environment

Docker allows us to develop and run portable applications
by exploiting *containers*



How old are containers?

- For decades, UNIX *chroot* command provided a simple form of filesystem isolation
- 1998 – FreeBSD *jail* utility extended *chroot* sandboxing to processes
- 2005 – Google started developing *CGroups* for Linux kernel and began moving its infrastructure to containers
- 2008 – Linux Containers (LXC) provided a complete containerization solution

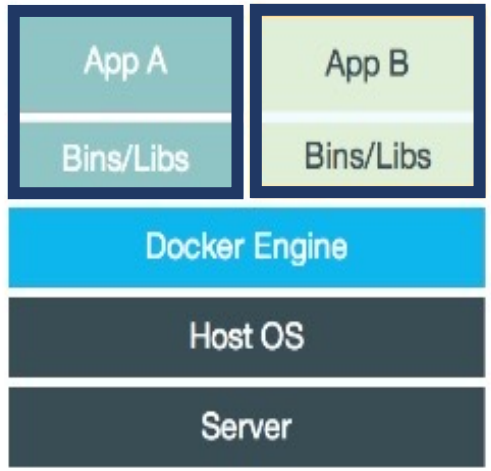
2013 - Docker added the missing pieces - *portable images* and *friendly UI* – to the containerization puzzle, and containers entered the mainstream

The Docker platform consisted of:

- **Docker Engine** (for creating and running containers)
- **Docker Hub** (for distributing containers)

Docker

- Docker exploits container-based virtualization to run multiple isolated guest instances on the (same) OS



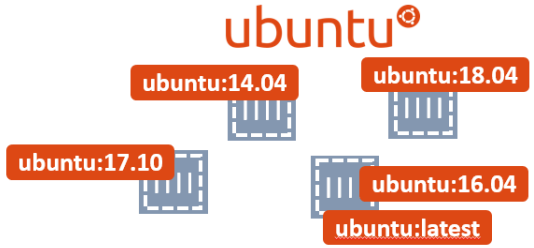
- Software components are packaged into ***images***, which are exploited as read-only templates to create and run ***containers***

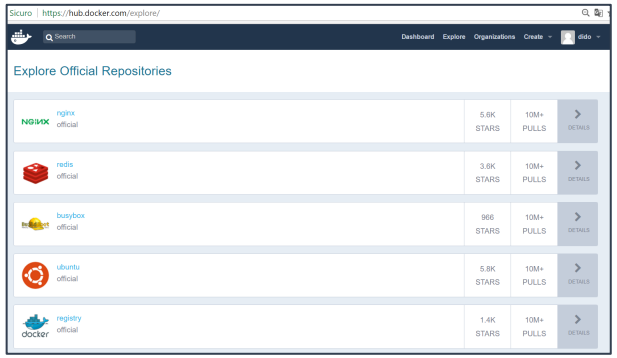
- External ***volumes*** can be mounted to ensure data persistence

"Build, ship, and run any app, anywhere"

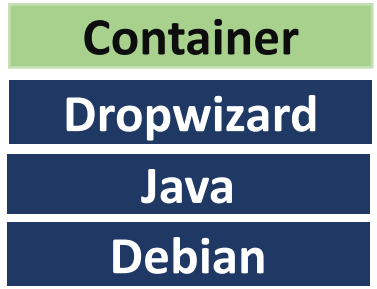
Docker images

- (read-only) templates used to create containers
- stored in a (private or public) Docker **registry**
  - registry structured in repositories
  - each repository contains a set of immages, for different versions of a software
  - images identified by pairs **repository:tag**
  - **Docker Hub**

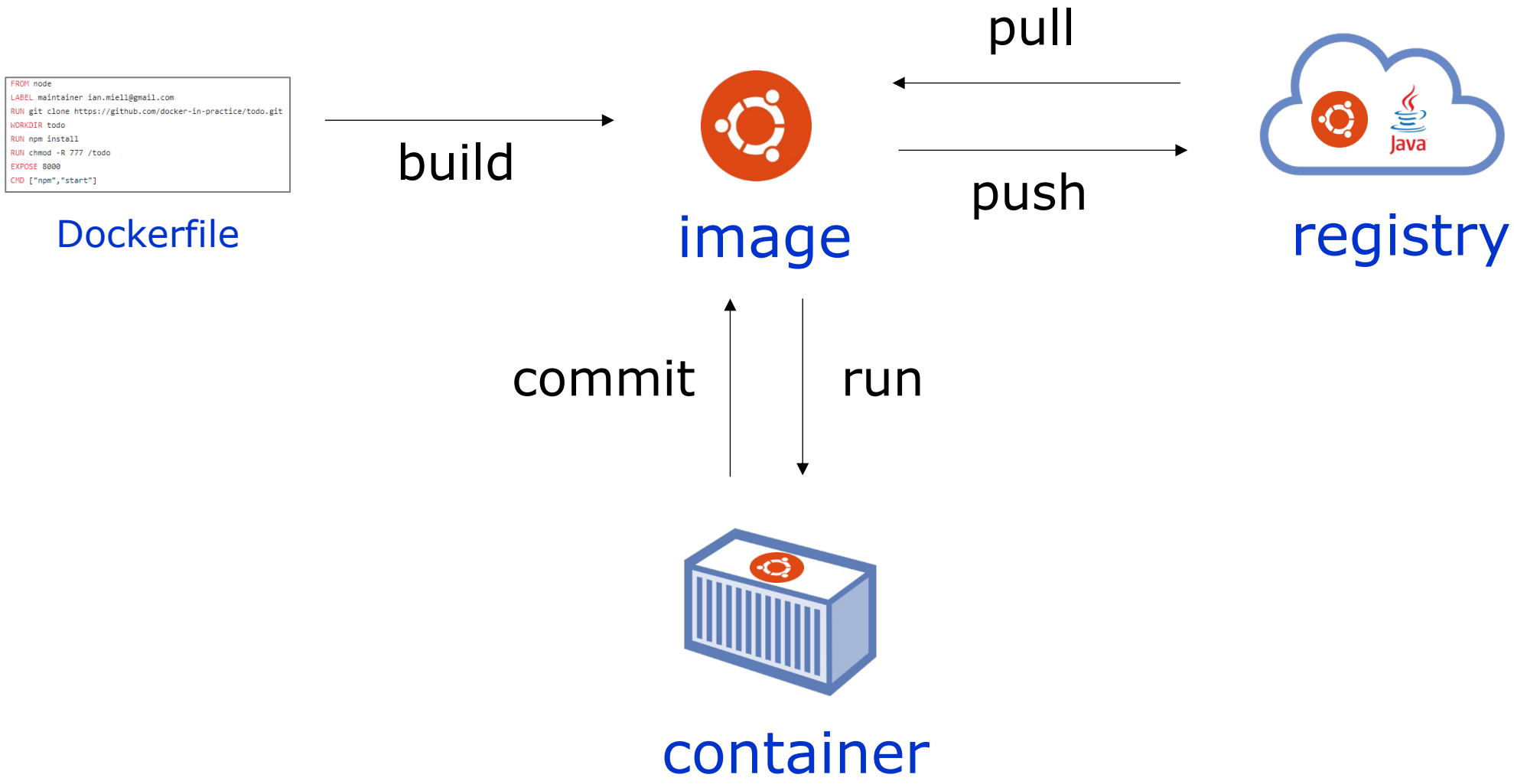




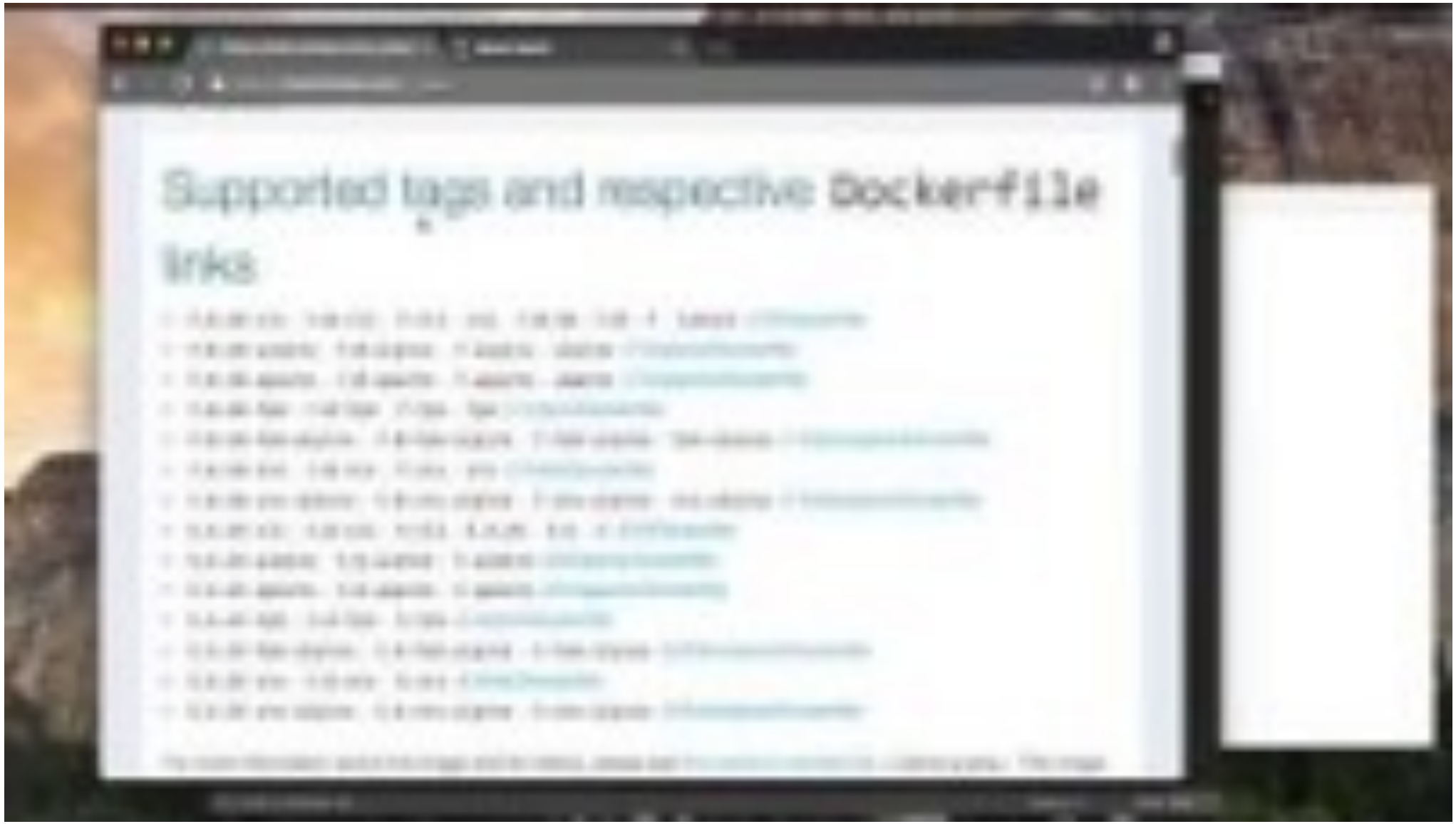
- images structured into (read-only) layers
- each layer is in turn an image, lowest layer called base image
- running container can write in new top layer, changes can be committed into new image



Docker commands



Learn Docker in 12 min



https://www.youtube.com/watch?v=YFl2mCHdv24

Advantages of Docker

- same environment
- sandbox projects
- it just works

Containers

- lighter and faster to start than VMs
- share OS
- images defined via Dockerfile
- build and run

Demo

- install Docker
- simple php “hello world” app
  - index.php

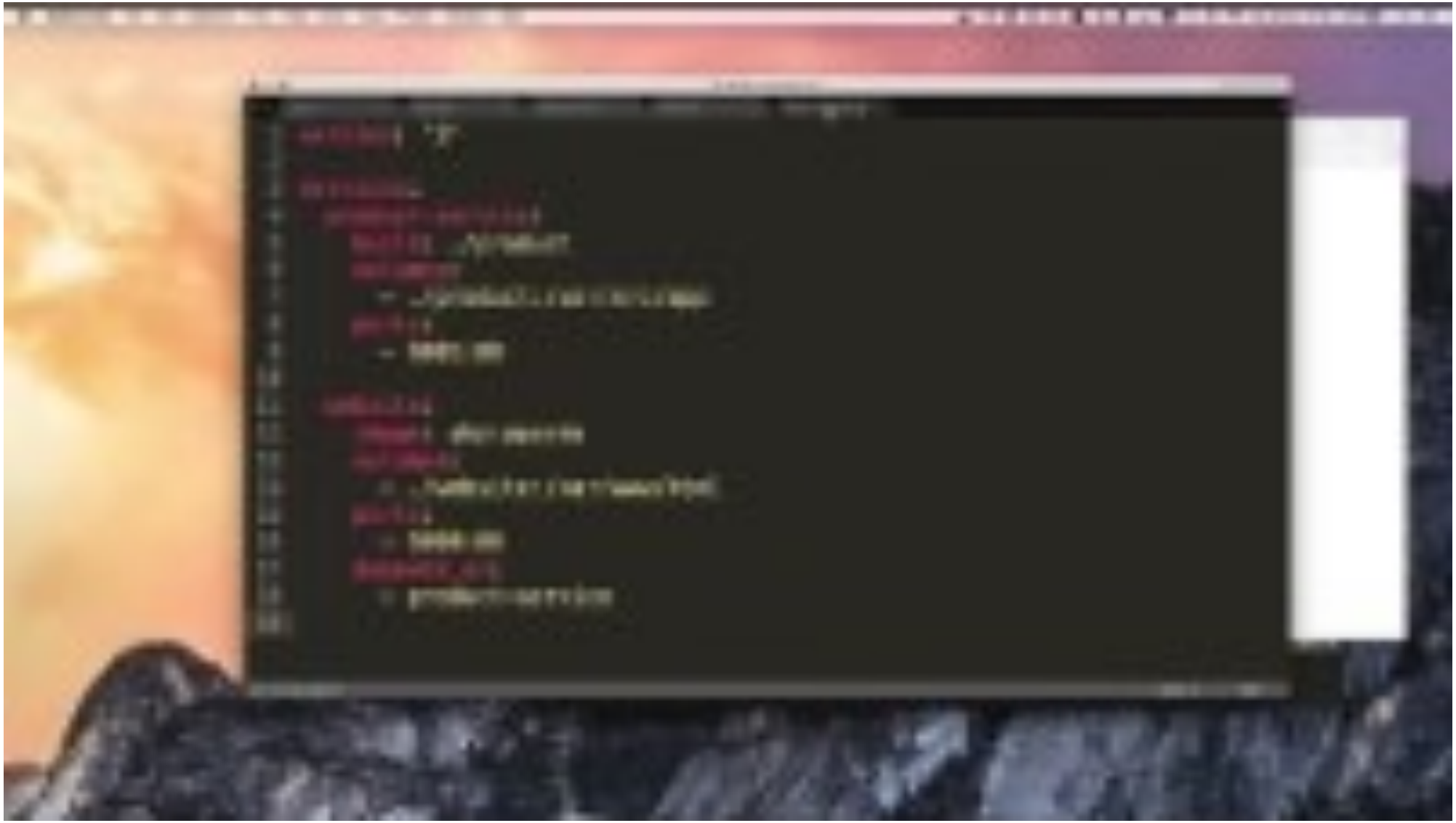
```
1 <?php
2
3 echo "Hello, World"
```

  - Dockerfile
    - find php image from Docker Hub

```
1 FROM php:7.0-apache
2 COPY src/ /var/www/html
3 EXPOSE 80
```

  - >docker build –t hello-world .
  - >docker run –p 80:80 hello-world
  - using a volume to share folder between host and container
  - one process per container

Docker Compose in 12 min



https://www.youtube.com/watch?v=Qw9zlE3t8Ko

//product service

api.py

```python
from flask import Flask
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

class Product(Resource):
    def get(self):
        return {
            'products': ['Ice cream',
                         'Chocolate',
                         'Fruit',
                         'Eggs']
        }

api.add_resource(Product, '/')

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=80, debug=True)
```

requirements.txt

```
Flask==0.12
flask-restful==0.3.5
```

Dockerfile

```
FROM python:3-onbuild
COPY . /usr/src/app
CMD ["python", "api.py"]
```

docker-compose.yml

```yaml
version: '3'

services:
  product-service:
    build: ./product
    volumes:
      - ./product:/usr/src/app
    ports:
      - 5001:80

  website:
    image: php:apache
    volumes:
      - ./website:/var/www/html
    ports:
      - 5000:80
    depends_on:
      - product-service
```

>docker-compose up
>docker ps
>docker-compose stop

//website

index.php

```php
<html>
    <head>
        <title>My Shop</title>
    </head>

    <body>
        <h1>Welcome to my shop</h1>
        <ul>
            <?php
                $json = file_get_contents('http://product-service');
                $obj = json_decode($json);

                $products = $obj->products;
                foreach ($products as $product) {
                    echo "<li>$product</li>";
                }
            ?>
        </ul>
    </body>
</html>
```


Virtualization and containers
Everything as a service

Pizza as a Service

Infrastructure as a Service (IaaS)	Platform as a Service (PaaS)	Software as a Service (SaaS)
Dining Table	Dining Table	Dining Table
Soda	Soda	Soda
Electric / Gas	Electric / Gas	Electric / Gas
Oven	Oven	Oven
Fire	Fire	Fire
Pizza Dough	Pizza Dough	Pizza Dough
Tomato Sauce	Tomato Sauce	Tomato Sauce
Toppings	Toppings	Toppings
Cheese	Cheese	Cheese

You Manage

Vendor Manages

*aaS

+ FaaS …

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

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

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


Virtualization and containers
Everything as a service
SaaS

SaaS

Software products were initially installed on customers' computers

- Customers had to configure software
- Customers had to deal with software updates
- Software product company had to maintain different product versions

SaaS

- Product delivered as a service
- Customers do not have to install software
- They pay a subscription and access product from remote

Benefits of SaaS for product providers

Cash flow

Regular cash flow: Customers pay periodic subscription or pay-per-usage

Update management

You control product updates, customers receive update at same time
No need of simulateneously maintaining several versions
Reduced costs

Continuous deployment

You can deploy new software versions as soon as changes have been made and tested

Payment flexibility

Different payment options can attract wider range of customers
e.g. small companies or individuals can avoid paying large upfront software costs

Try before you buy

You can make early free/low-cost product available and get customer feedback

Data collection

You can easily collect data on product usage and customers

Pros & cons for customers

- **Mobile, laptop, and desktop access**
- **No upfront costs for software or servers**
- **Immediate software updates**
- **Reduced software management costs**

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

Design issues for SaaS

Local vs. remote processing

Some features can be executed locally
+ reduced network traffic
+ increased response speed
- increased power consumption for battery-powered devices

Authentication

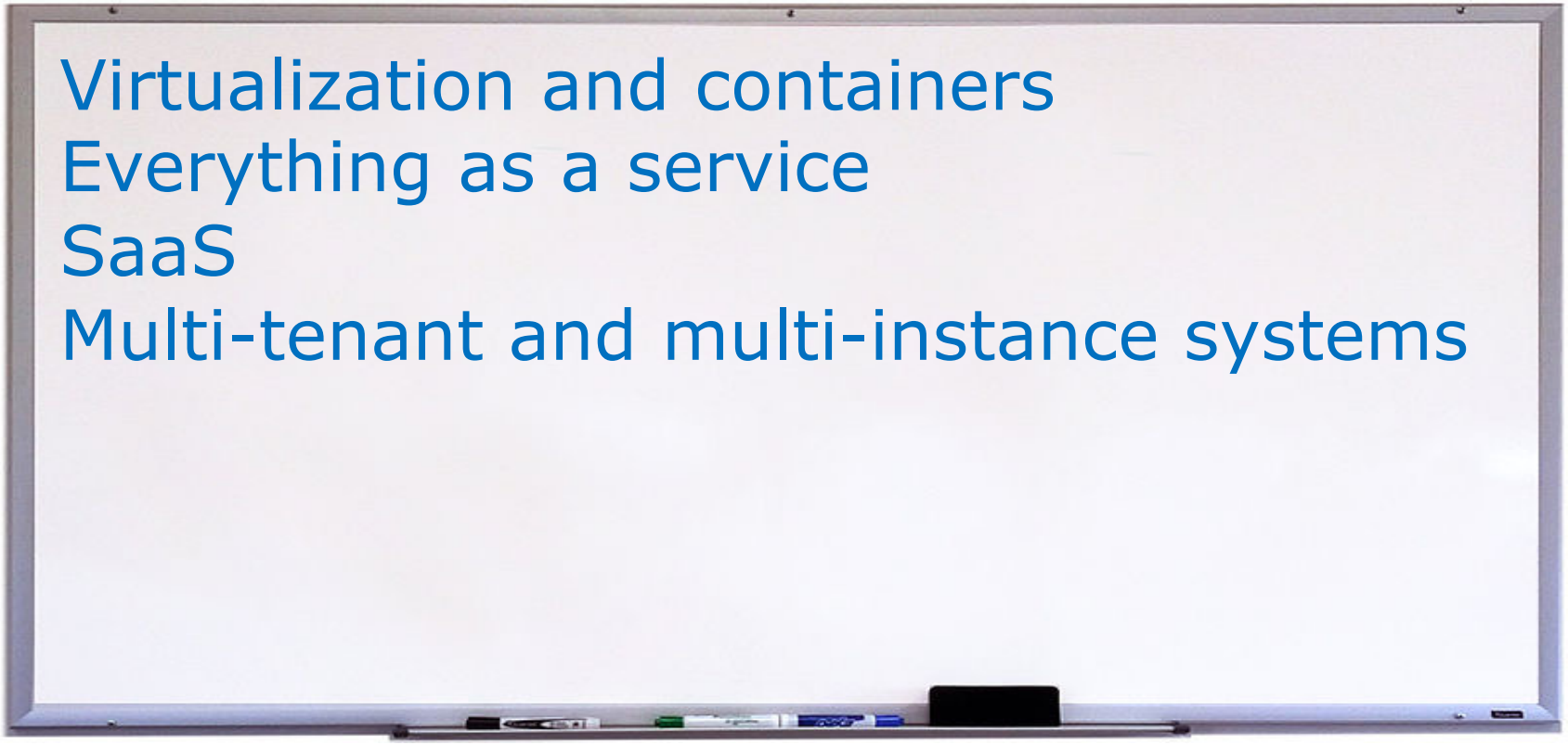
Product own authentication system vs.
Federated authentication vs.
Individual Google/LinkeIn/… credentials

Information leakage

Security risks with multiple users from multiple organizations

Multi-tenant vs. multi-instance database management

Single repository vs. separate copies of system and database


Virtualization and containers
Everything as a service
SaaS
Multi-tenant and multi-instance systems

Multi-tenant systems

- Single database schema shared by all system's users
- Items in database tagged with tenant identifier to provide "logical isolation"

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

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.

Multi-tenant systems

Mid-size and large businesses

- rarely want to use generic multi-tenant software
- often prefer a customized version adapted to their own requirements

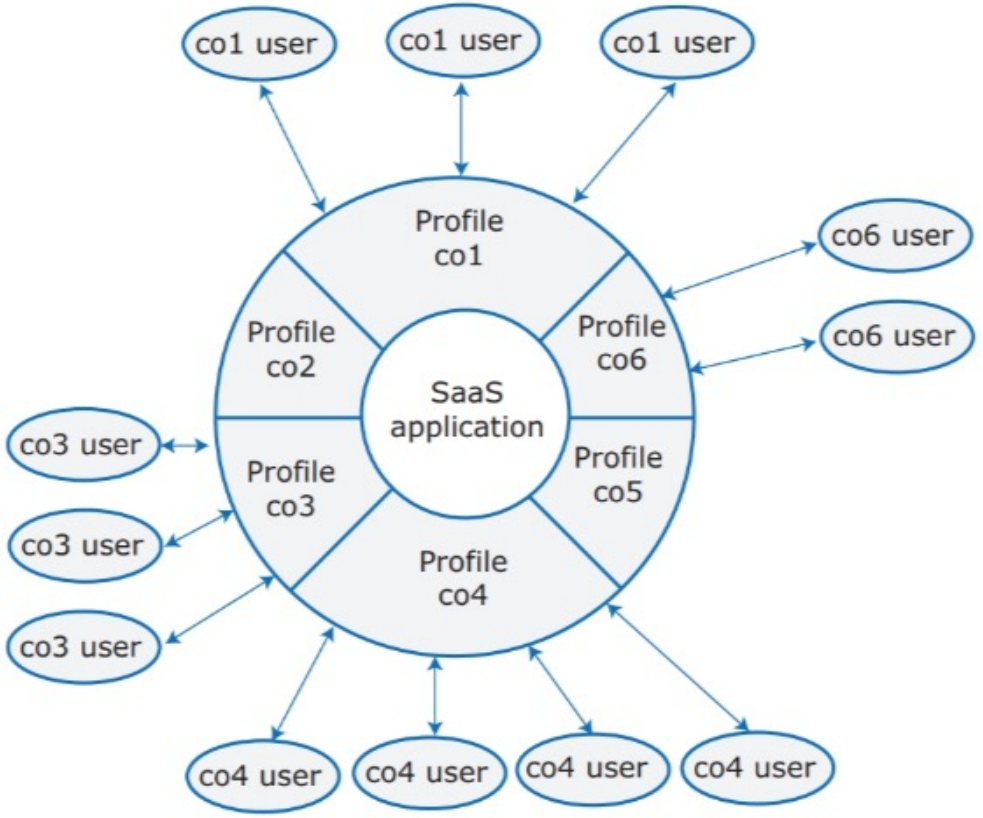
Table 5.6 Possible customizations for SaaS

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.

Multi-tenant systems: UI configurability

UI configurability can be implemented by employing user profiles

- Users asked to select their organization or provide their business email address
- Product uses profile information to create personalized version of interface
  - e.g. generic company name and logo replace with company's ones
  - e.g. some features of menus disabled



Multi-tenant systems: DB schema adaptation

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

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

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

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

Multi-tenant systems: DB schema adaptation

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

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

- e.g. The extension fields for tenant T516/ Item 100 are 'Location', 'Weight' and 'Fragile', with values 'A17/S6', '4', and 'False'
- e.g. The extension fields for T632/Item 100 are 'Delivered' and 'Place', and their values are '2017/1/15' and 'Dublin'

Issue: added complexity

Main database table

Tab1 — Stock management

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

Tab2 — Field names

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

Extension table showing the field names for each company that needs database extensions

Tab3 — Field values

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

Value table showing the value of extension fields for each record

Multi-tenant systems: Security issues

Security is the major concern of corporate customers with multi-tenant databases

- As information from all customers is stored in the same database, a software bug or an attack could lead to the data of some or all customers being exposed to others

Multilevel access control

- access to data must be controlled at both the organizational level and the individual level
  - organizational level access control to ensure that any database operations act on only that organization's data
  - individual users accessing the data should also have their own access permissions

Encryption of data

- To ensure corporate users that their data cannot be viewed by people from other companies if some kind of system failure occurs
- Encrypted data can be decrypted only when accessed with the appropriate key
- Encryption/decryption are computationally intensive operations → usually only sensitive data are encrypted

Multi-instance systems

Each customer has its own system that is adapted to its needs, including its own database and security controls

- conceptually simpler than multi-tenant systems
- avoid security concerns such as inter-organization data leakage

Multi-instance systems

Two types of multi-instance systems:

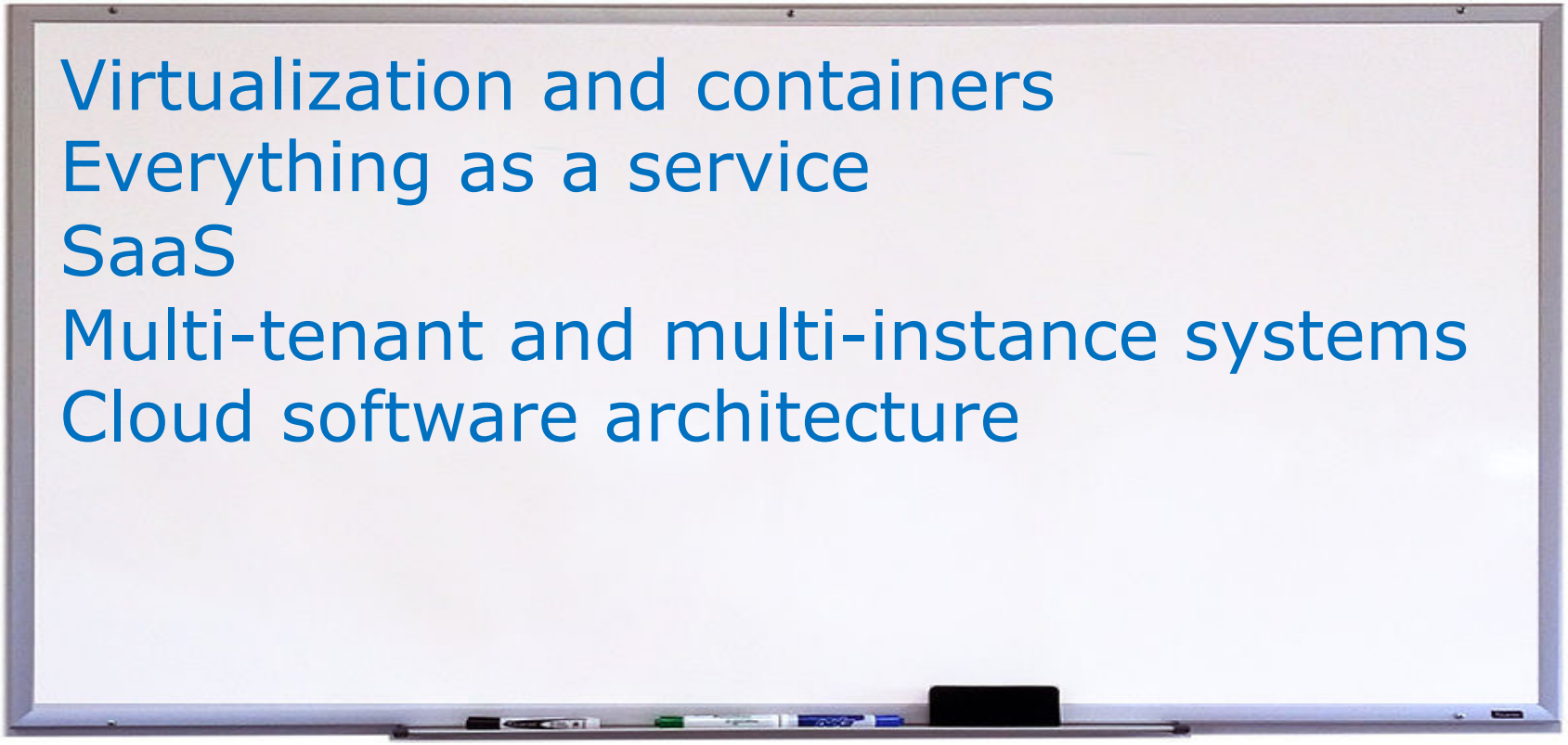
- ***VM-based***
  - software instance and database for each customer run in its own VM
  - all users from same customer may access the shared system database

- ***Container-based***
  - each user has an isolated version of software and database running in a set of containers
  - most suited to products in which users mostly work independently, with little data sharing (e.g. for individual users)

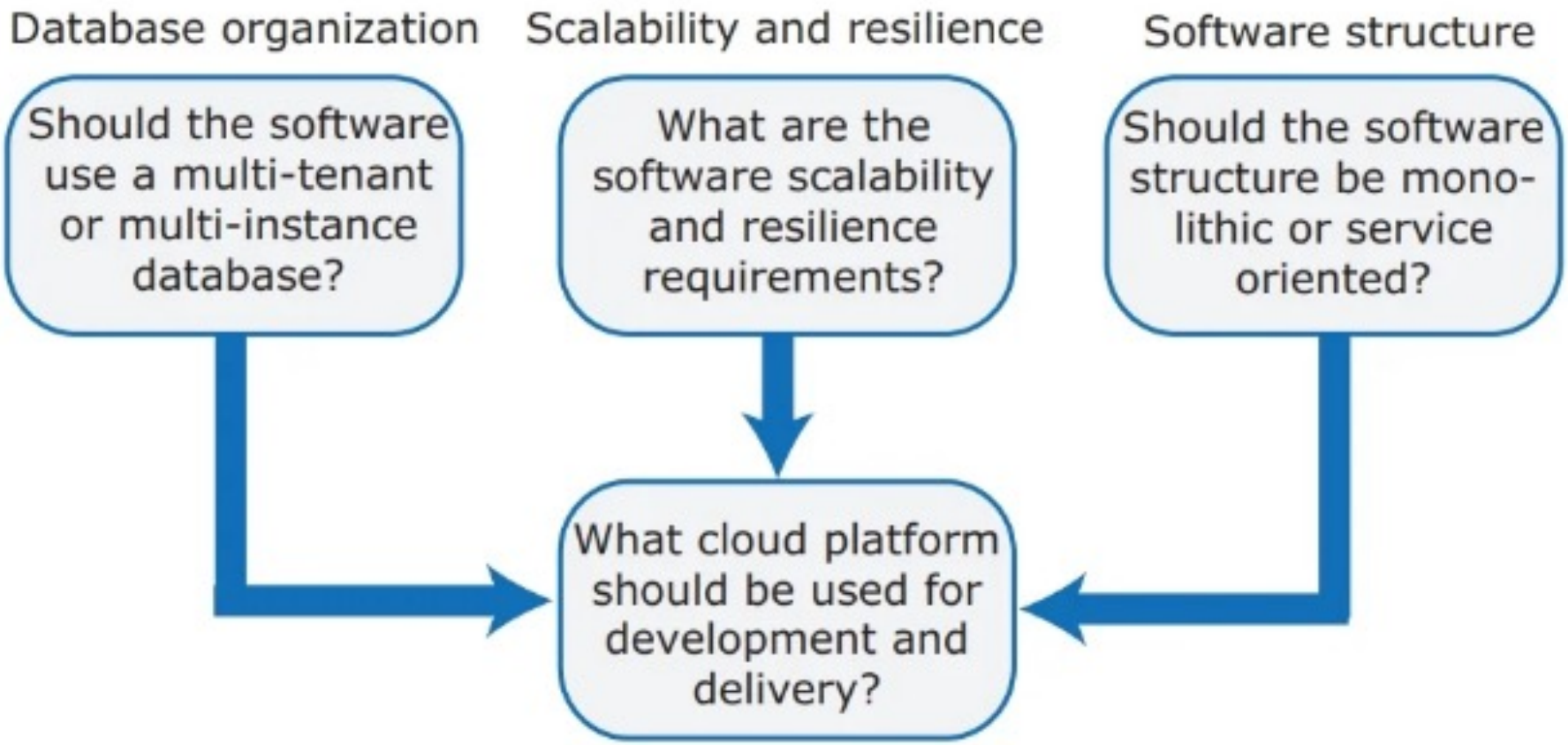
It is also possible to run containers on a virtual machine
a business could have its own VM-based system and run containers on top of this for individual users

Pros and cons 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.	


Virtualization and containers
Everything as a service
SaaS
Multi-tenant and multi-instance systems
Cloud software architecture

Architectural decisions



DB organization

There are three possible ways of providing a customer database in a cloud-based system:

1. As a multi-tenant system, shared by all customers for your product. This may be hosted in the cloud using large, powerful servers.
2. As a multi-instance system, with each customer database running on its own virtual machine.
3. As a multi-instance system, with each database running in its own container. The customer database may be distributed over several containers.

Factor	Key questions
Target customers	Do customers require different database schemas and database personalization? Do customers have security concerns about database sharing? If so, use a multi-instance database.
Transaction requirements	Is it critical that your products support ACID transactions where the data are guaranteed to be consistent at all times? If so, use a multi-tenant database or a VM-based multi-instance database.
Database size and connectivity	How large is the typical database used by customers? How many relationships are there between database items? A multi-tenant model is usually best for very large databases, as you can focus effort on optimizing performance.
Database interoperability	Will customers wish to transfer information from existing databases? What are the differences in schemas between these and a possible multi-tenant database? What software support will they expect to do the data transfer? If customers have many different schemas, a multi-instance database should be used.
System structure	Are you using a service-oriented architecture for your system? Can customer databases be split into a set of individual service databases? If so, use containerized, multi-instance databases.

DB organization

Different types of customers have different expectations about software products

Examples:

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

Scalability

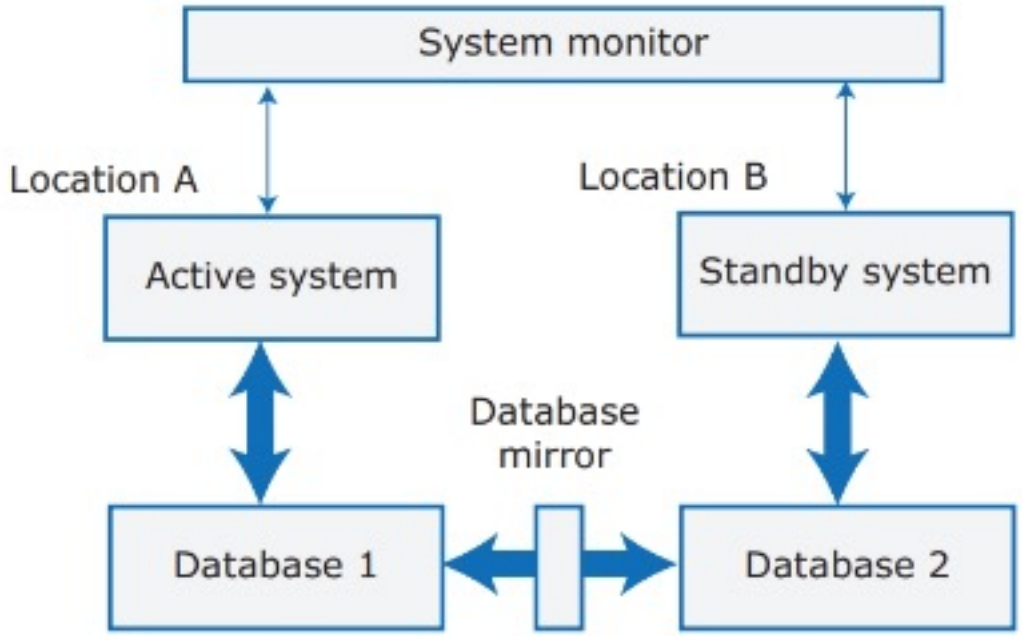
- Scalability = ability to adapt automatically to load changes
- Achieved by
  - By adding new virtual servers (scaling out)
  - Or by increasing the power of a system server (scaling up)

  in response to increasing load

- Scaling out typical of cloud-based systems
  - Product must be organized so that individual software components can be replicated and run in parallel
  - Load-balancing mechanisms direct requests to different instances of the components (e.g. with PaaS)

Resilience

Resilience = ability to continue to deliver critical services in the event of system failure or malicious system use



"Hot standby"

- Replicas of software and data maintained in different locations
- Database updates are mirrored
- System monitor continuously checks system status

Cheaper alternative: "cool stanby"

- data restored from backup
- system unavailable till data restore is complete

Software structure

Monolith vs. Fine-grain, stateless services

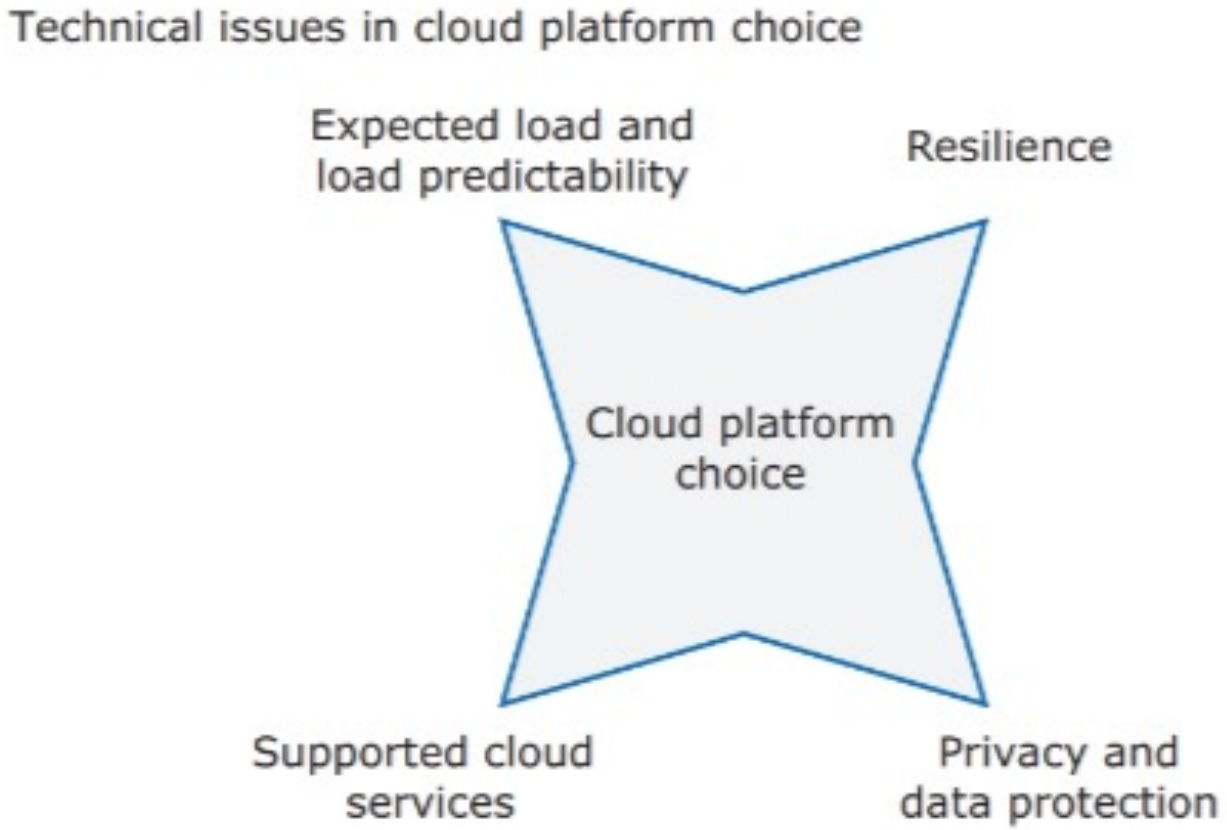
+ independent services that can be replicated, distributed, and migrated

+ particularly suitable for cloud-based software with services deployed in containers

Monolithic approach often employed to build prototype

When different parts have to be updated at different times and when only some parts have to scale → microservices

Cloud platform

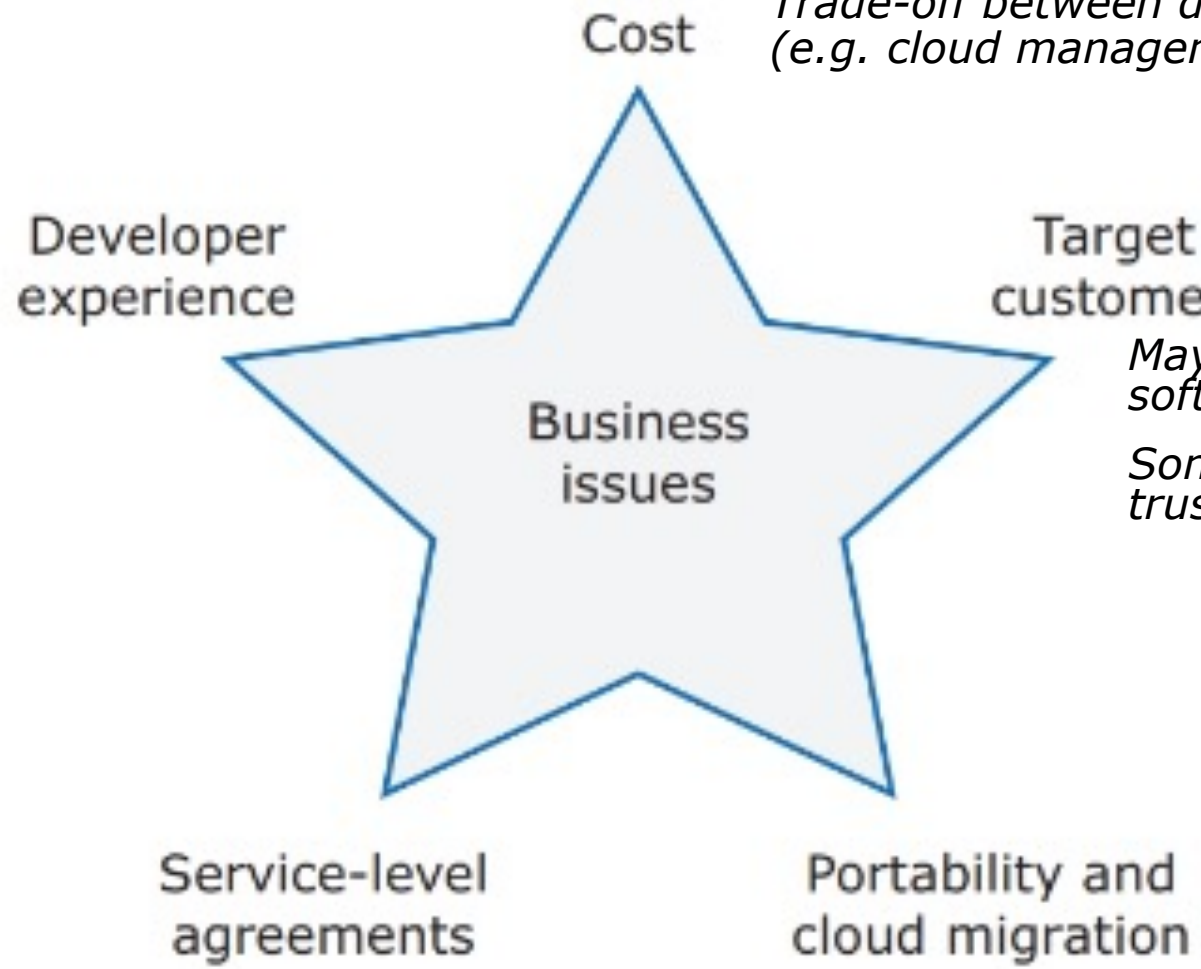


Both technical and business issues

e.g. EU countries have strict requirements on data protection and on where data are stored → need cloud provider providing guarantees on storage locations

Cloud platform

Business issues in cloud platform choice



Trade-off between direct and indirect costs (e.g. cloud management)

May need to interoperate with other software used by your customers

Some companies prefer to deal with trusted providers

"Guaranteed" QoS?

Both technical and business issues

Make an "exit plan" to mitigate vendor lock-in

Reference

Chapter 5 – Cloud-based software