

Advanced Topics of Software Engineering (ASE)

Chapter 4. Software Deployment Alternatives

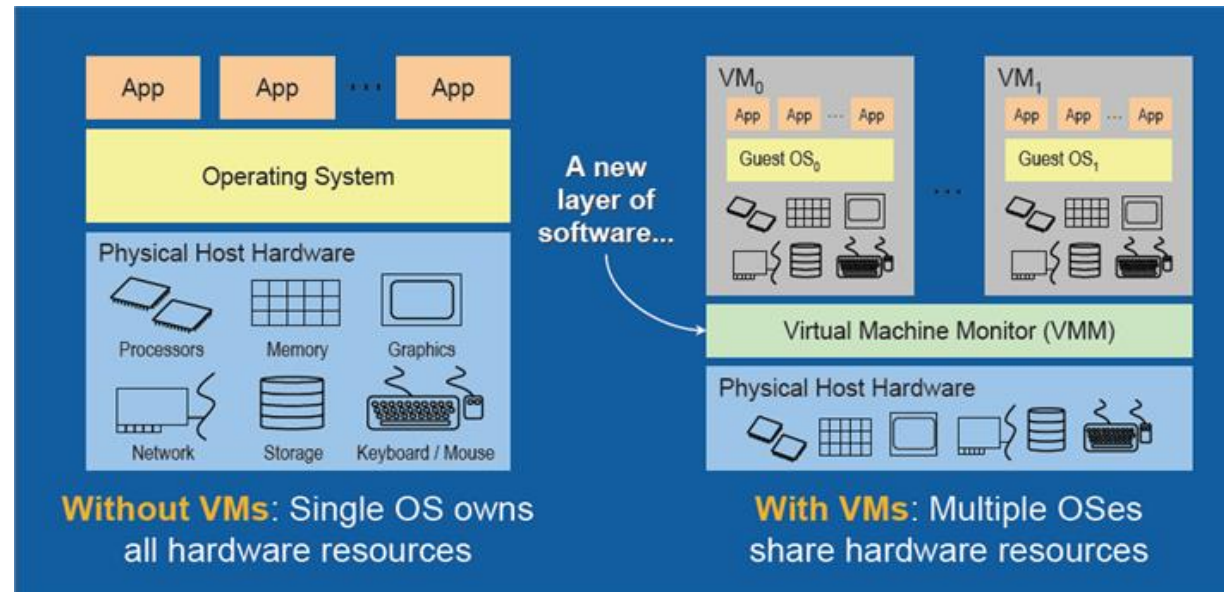
Prof. Dr. Florian Matthes, Prof. Dr. Alexander Pretschner

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
www.matthes.in.tum.de

4.1. Virtual machines and containers

4.2. Software architectures for the cloud

Virtualization is a combination of software and hardware engineering that creates Virtual Machines (VMs) - an abstraction of the computer hardware that allows a single machine to act as if it were many machines.



The term *virtual machine* initially described a 1960s operating system concept: a software abstraction with the looks of a computer system's hardware (real machine). Forty years later, the term encompasses a large range of abstractions—for example, Java virtual machines that don't match an existing real machine.

[The Reincarnation of Virtual Machines, Mendel Rosenblum (2004)]

- **Emulation** involves emulating the physical machine's hardware and architecture.
- **Virtualization**, on the other hand, involves simply isolating the virtual machine within memory.

[*"Virtual Machines: Virtualization vs. Emulation"*. Griffin (2006)]

Software compatibility

- The virtual machine provides a compatible abstraction so that all software written for it will run on it.
- A **hardware-level virtual machine** will run all the software, operating systems, and applications written for the hardware.
- An **operating system–level virtual machine** will run applications for that particular operating system,
- A **high-level virtual machine** will run programs written in the high-level language.
- The virtual machine abstraction frequently can mask differences in the hardware and software layers below the virtual machine. One example is Java's claim that you can “write once, run anywhere.”

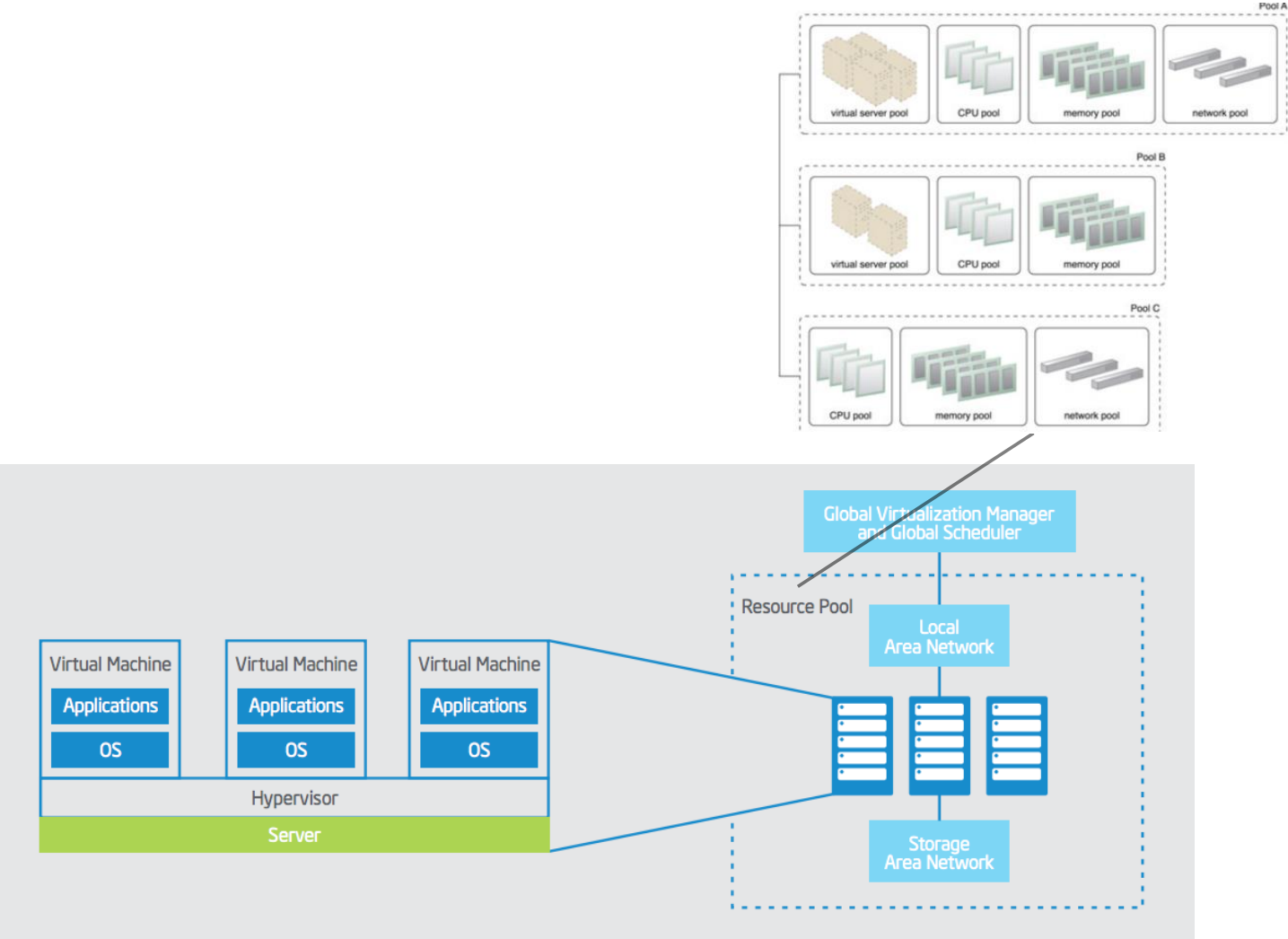
Isolation

- The virtual machine abstraction isolates the software running in the virtual machine from other virtual machines and real machines.
- Isolation provides that bugs or hackers can be contained within the virtual machine and thus not adversely affect other parts of the system.
- Virtualization layer can execute performance isolation so that resources consumed by one virtual machine do not necessarily harm the performance of other virtual machines.
- Operating systems are not as fair in performing resource balancing and starvation prevention as virtual machine environments tend to be.

Virtualization enables the cloud

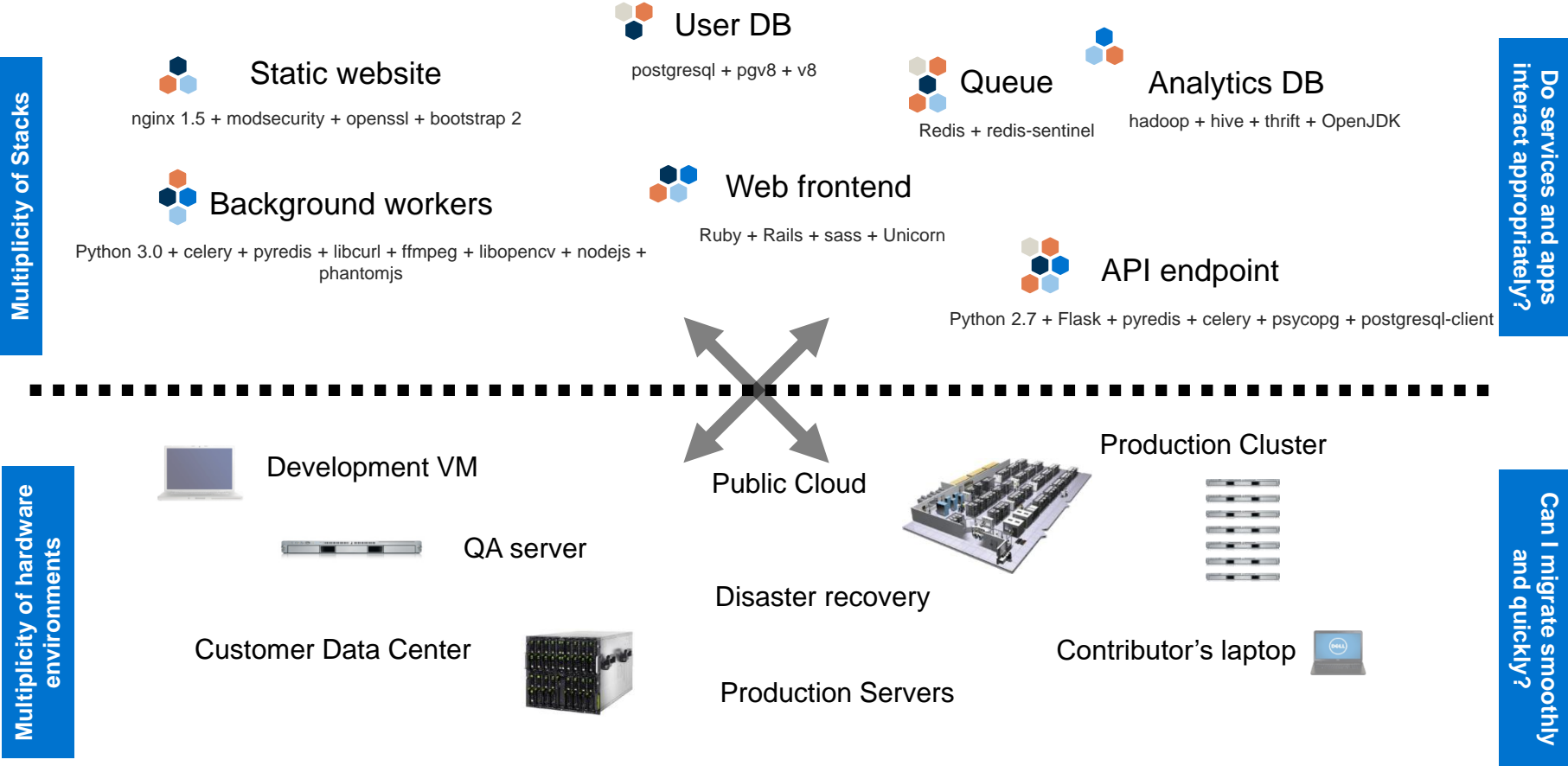
- The underpinning for the majority of high-performing clouds is a *virtualized* infrastructure.
- Virtualization has been in data centers for several years as a successful IT strategy for consolidating servers.
- Used more broadly to pool infrastructure resources, virtualization can also provide the basic building blocks for your cloud environment to enhance agility and flexibility.
- Today, the primary focus for virtualization continues to be on servers.
- Virtualizing storage and networks is emerging as a general strategy.

Virtual machines in the cloud
















Containers

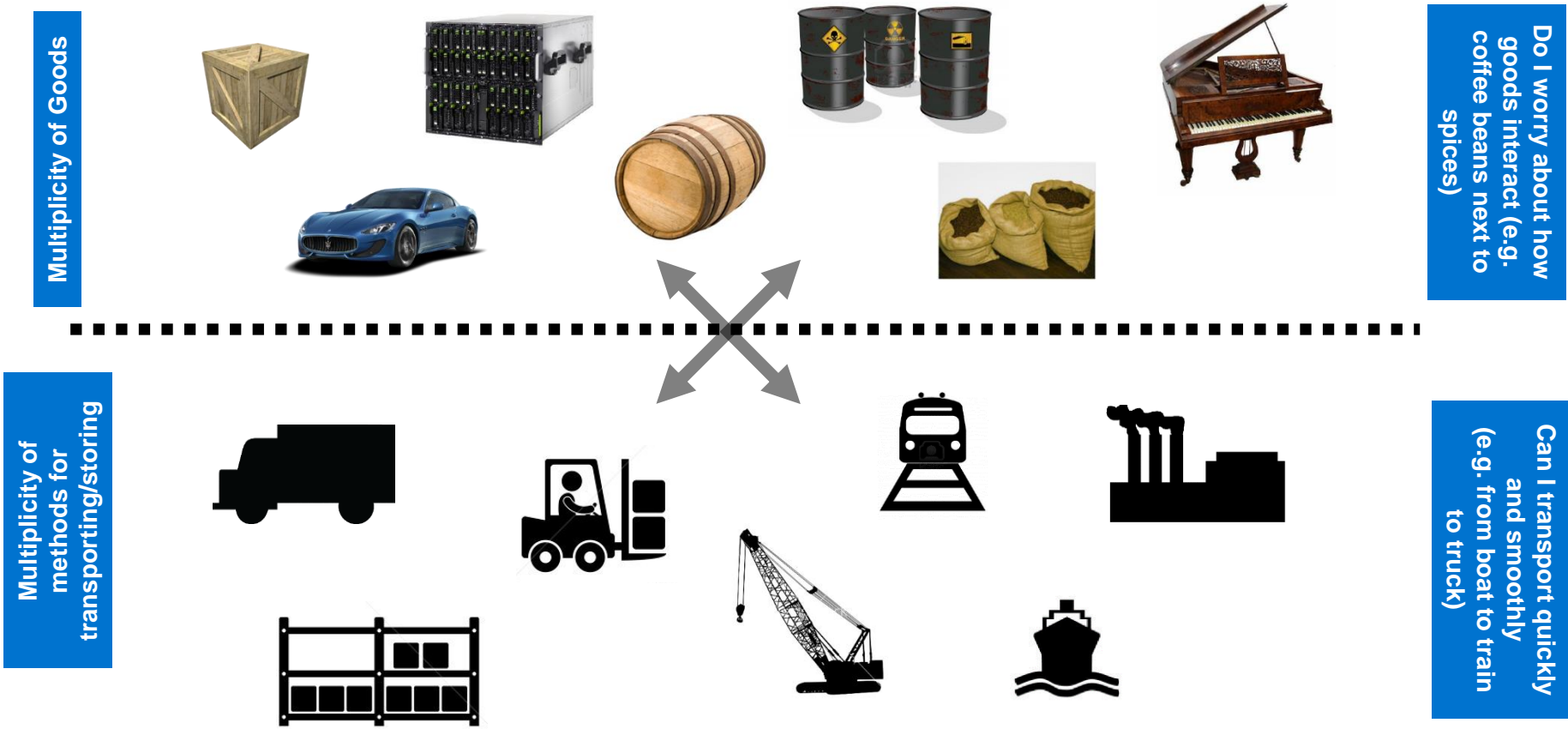
The challenge



N X N compatibility nightmare

	Static website	?	?	?	?	?	?	?
	Web frontend	?	?	?	?	?	?	?
	Background workers	?	?	?	?	?	?	?
	User DB	?	?	?	?	?	?	?
	Analytics DB	?	?	?	?	?	?	?
	Queue	?	?	?	?	?	?	?
		Development VM	QA Server	Single Prod Server	Onsite Cluster	Public Cloud	Contributor's laptop	Customer Servers
								

Cargo transport pre-1960 analogy














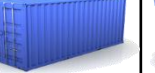



































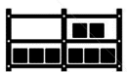







Intermodal shipping container



Shipping containers

A solution to the NXN problem

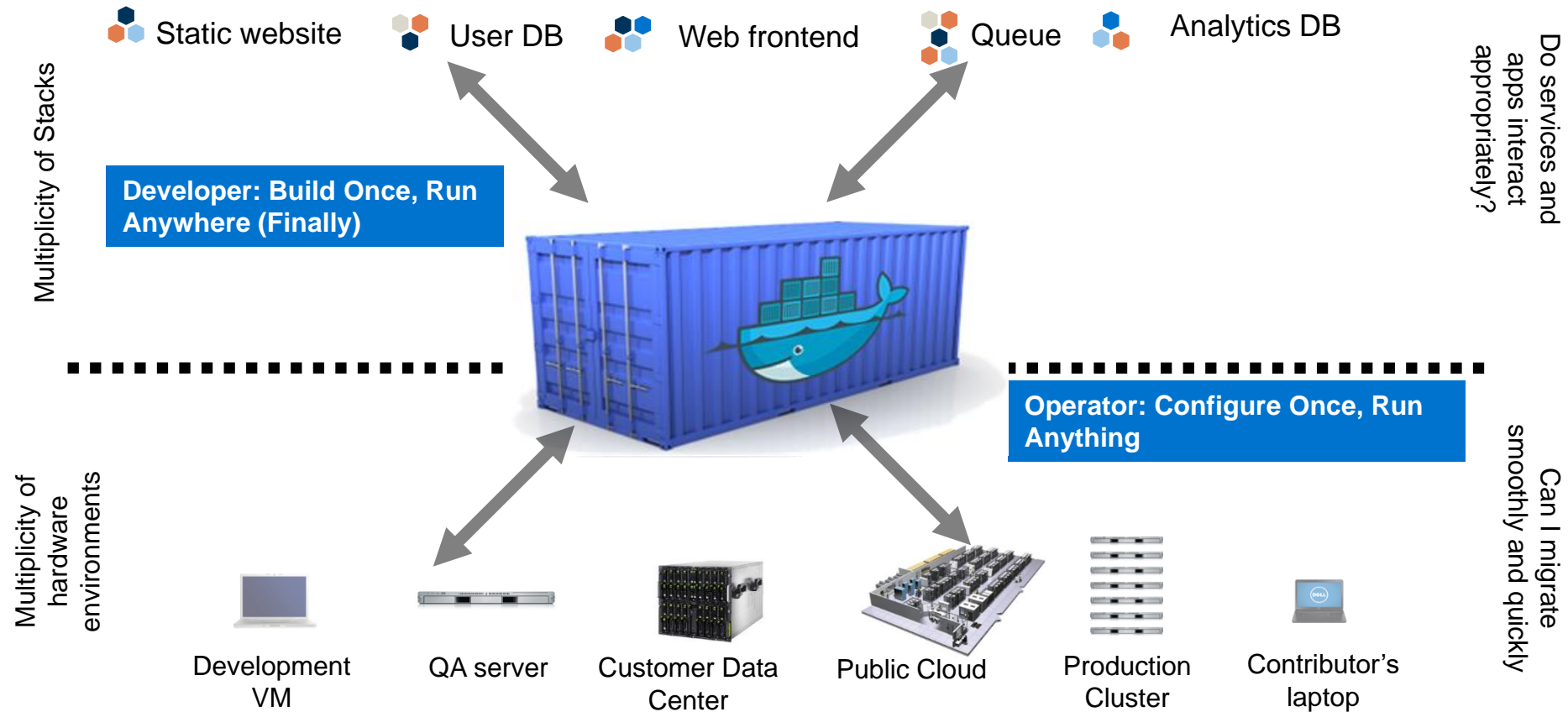
Intermodal shipping container ecosystem



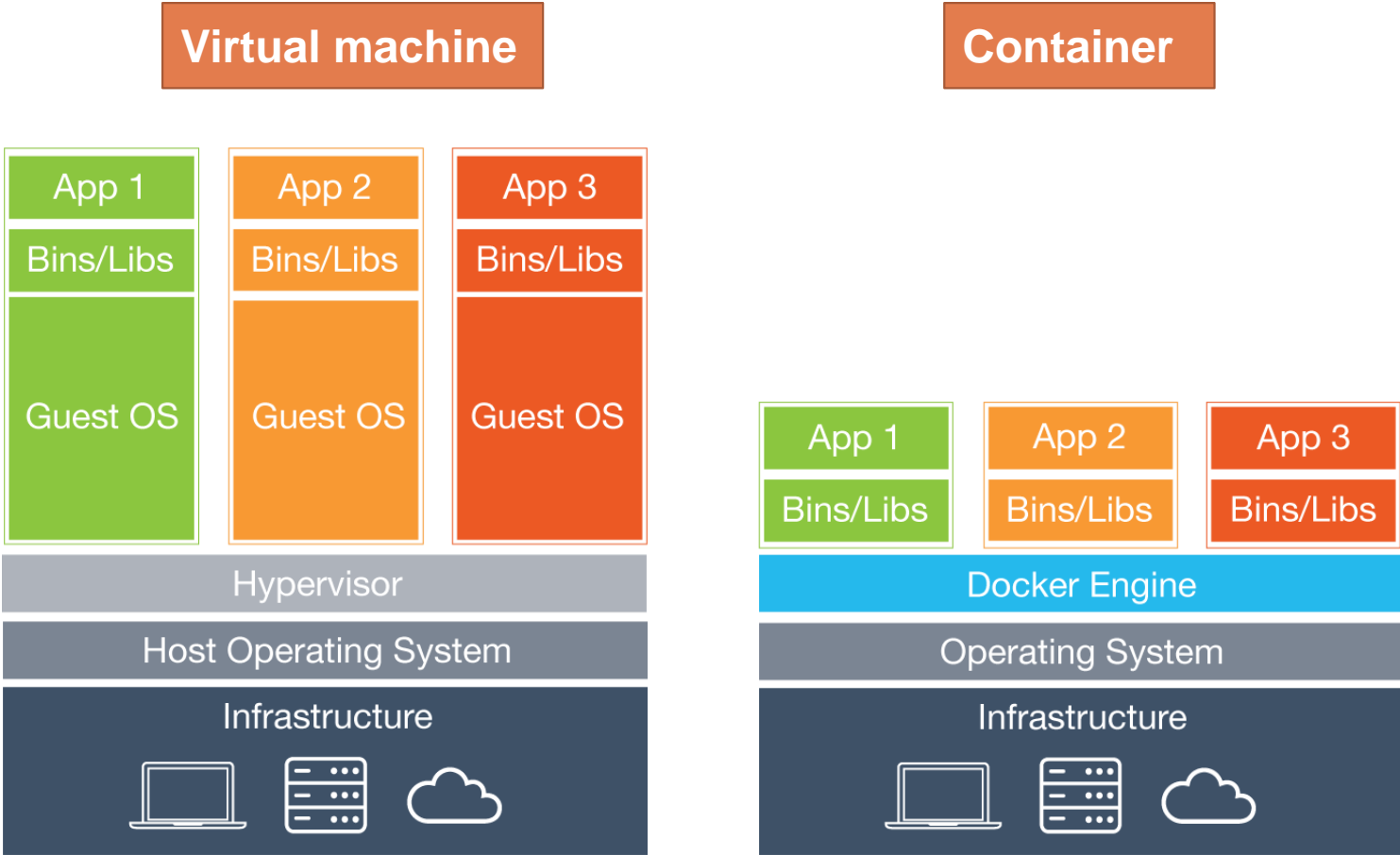
- 90% of all cargo now shipped in a standard container
- Order of magnitude reduction in cost and time to load and unload ships
- Massive reduction in losses due to theft or damage
- Huge reduction in freight cost as percent of final goods (from >25% to <3%)
- Massive globalization
- 5000 ships deliver 200M containers per year

Docker

A shipping container for the code



Virtual machines vs application containers



[“[Docker](#) - How is this different from virtual machines?”]

The difference between hypervisors and containers

- A **hypervisor** works by having the host operating system emulate machine hardware and then bringing up other virtual machines (VMs) as guest operating systems on top of that hardware.
 - This means that the communication between the guest and host operating systems must follow a hardware paradigm.
- A **container** virtualization is the virtualization at the operating system level, instead of the hardware level.
 - (+) Each of the guest operating systems share the same kernel and sometimes parts of the operating system, with the host.
 - (+) Leaner and smaller than hypervisor guests.
 - (-) Obviously cannot run different kernels in the system.
 - (-) You cannot install a Windows container on a Linux host or vice-versa.
 - (-) Isolation and security - the isolation between the host and the container is not as strong as hypervisor-based virtualization since all containers share the same kernel of the host and there have been cases in the past where a process in the container has managed to escape into the kernel space of the host.

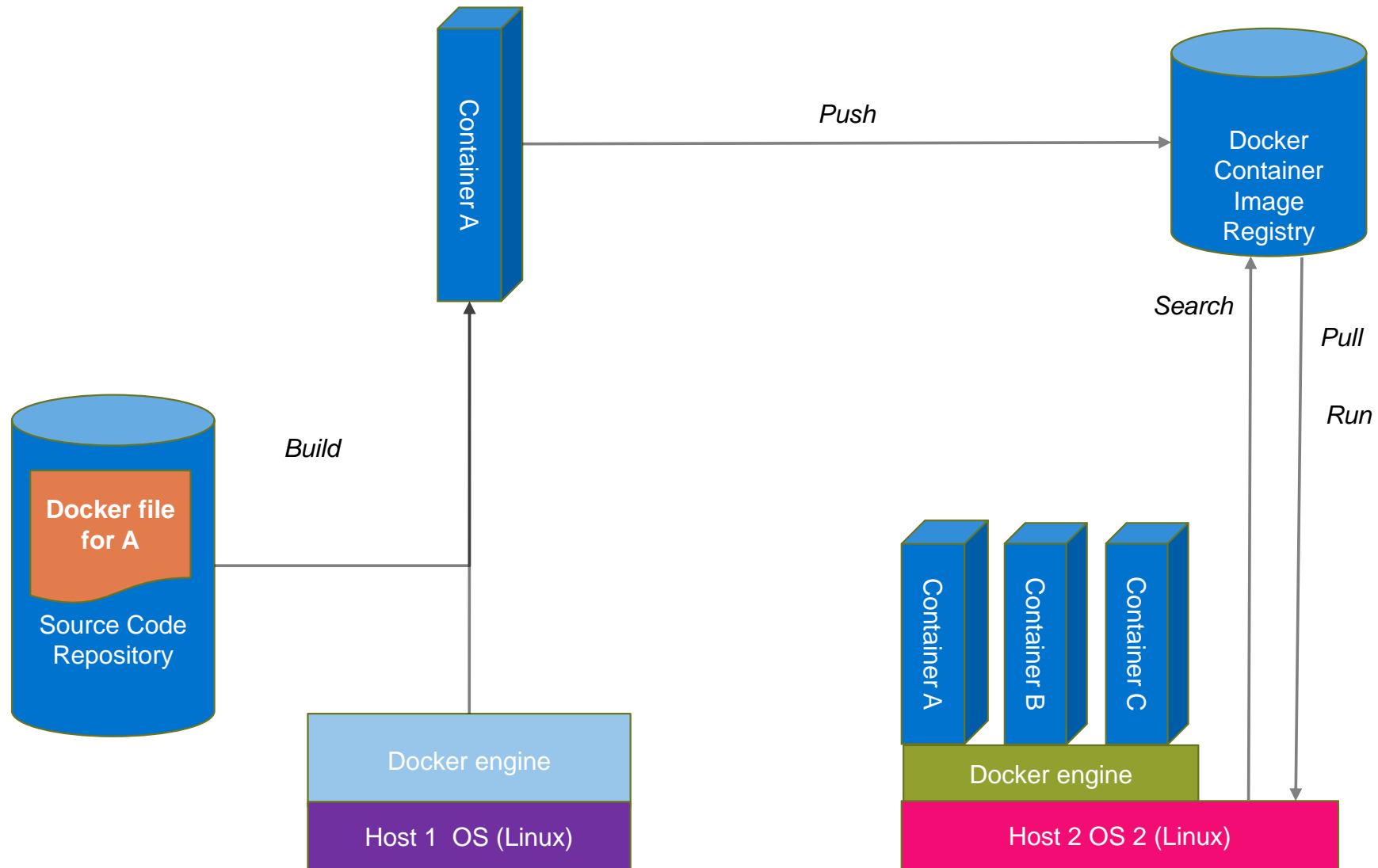
Docker system

- Virtualization software
- The open-source application container engine
- Apache license, version 2.0.
- First release: march 2013
- GitHub stars: 27.000+

“Docker interests me because it allows simple environment isolation and repeatability. I can create a run-time environment once, package it up, then run it again on any other machine. Furthermore, everything that runs in that environment is isolated from the underlying host (much like a virtual machine). And best of all, everything is fast and simple.”

[“Using Docker to Build Firefox.” Gregory Szorc (2013)]

Docker system



- Dockerfile defines what goes on in the environment inside your container.

```
$ docker build -t "syncPipesImage"
```

```
$ docker run -p 80:9000 syncPipesImage
```



Sharing your image on **hub.docker.com**

```
$ docker tag syncPipesImage  
username/sebisChair:syncPipes
```

```
$ docker push username/sebisChair:syncPipes
```



```
$ docker pull sebisChair:syncPipes
```

Use Java

Set up environment variables

Copy files from your file system
into the container

Build if necessary

Run the application

Expose the container's ports

Example Dockerfile- running a Java Play application:

```
FROM anapsix/alpine-java:8_jdk

ENV SBT_URL=https://dl.bintray.com/sbt/native-packages/sbt
ENV SBT_VERSION 0.13.15
ENV INSTALL_DIR /usr/local
ENV SBT_HOME /usr/local/sbt
ENV PATH ${PATH}:${SBT_HOME}/bin

ENV PROJECT_HOME /usr/src
COPY dist/akre-1.0.zip ${PROJECT_HOME}/akre-1.0.zip

RUN cd ${PROJECT_HOME} && unzip akre-1.0.zip && \
    chmod +x ${PROJECT_HOME}/akre-1.0/bin/akre

RUN mkdir -p ${PROJECT_HOME}/akre-1.0/app/resources
COPY app/resources/ ${PROJECT_HOME}/akre-1.0/app/resources/

CMD ["/usr/src/akre-1.0/bin/akre", "-Dhttp.port=9000"]

# Expose port 9000
EXPOSE 9000
```

Basic docker commands

Services with docker compose



- Services are really just “containers in production.”
- A service only runs one image, but it codifies the way that image runs - what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on.

docker-compose.yml

Start your services with
\$ docker-compose up

Get the image from dockerHub →

Define dependencies on other services →

Link internal configuration names
to other services →

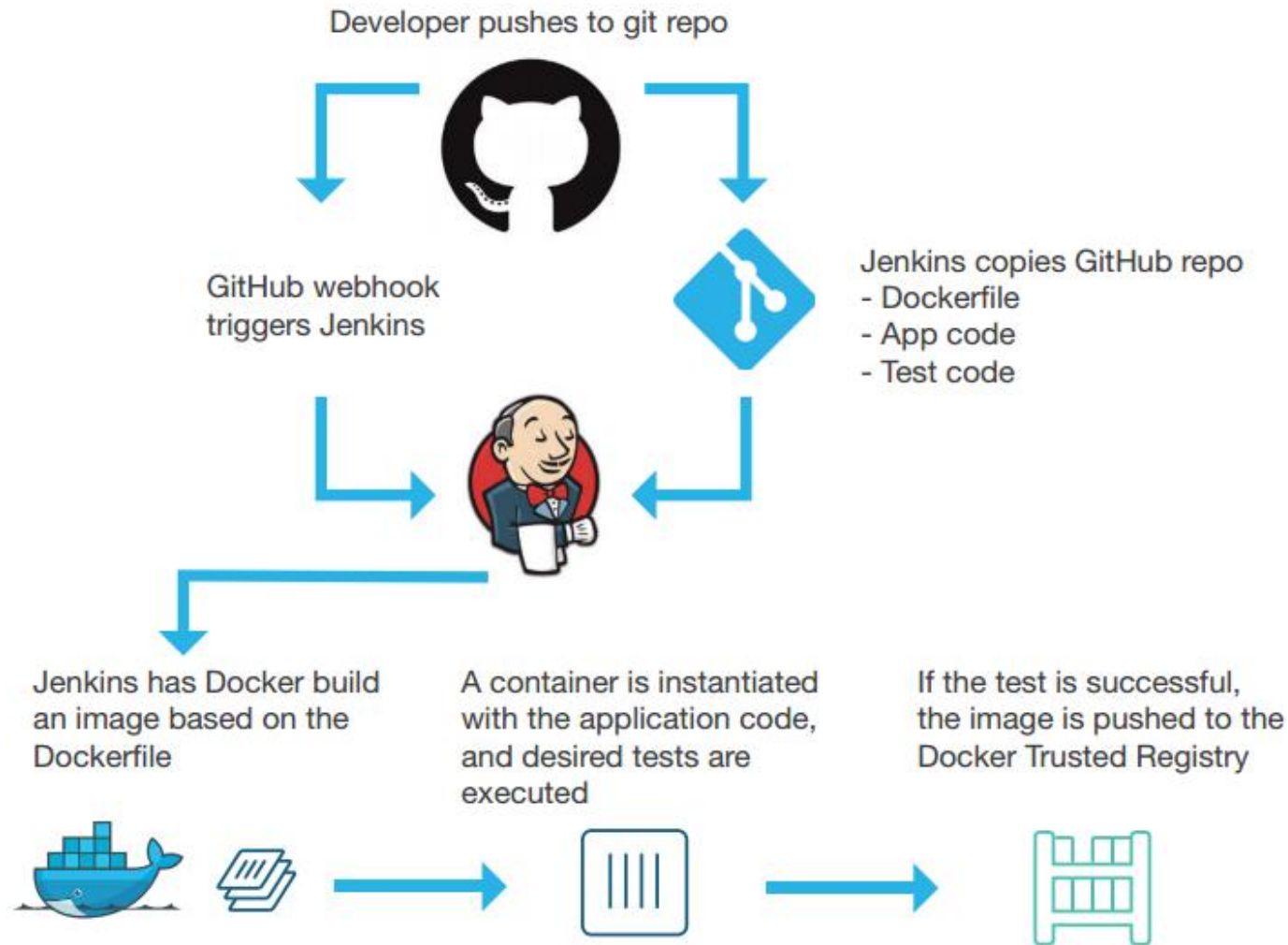
List of other services →

```
version: '3'
services:
  akre:
    image: 'sebischair/amelie:akre'
    ports:
      - "9000:9000"
    depends_on:
      - docclustering
      - docclassification
      - syncpipes-server
    links:
      - mongo:mongo
      - docclustering:docclustering
      - docclassification:docclassification
      - syncpipes-server:syncpipes-server
  amelie: <3 keys>
  syncpipes-client: <3 keys>
  syncpipes-server: <5 keys>
  docclassification: <4 keys>
  docclustering: <4 keys>
  mongoforamelie: <3 keys>
  mongo:
    image: 'mongo:3.6.2'
    ports:
      - 27017:27017
  rabbit: <3 keys>
```

[For more check: <https://docs.docker.com/get-started>]

Continuous deployment pipeline with Docker

Also check automated build options using web hooks in Docker cloud build



Docker Compose is a tool for defining and running multi-container Docker applications.

It uses **YAML** files to configure the application's services and performs the creation and start-up process of all the containers with a single command.

Sample 3-tier application: docker-compose.yaml

```
version: "3"
services:
  frontend:
    image: app-frontend
    ports:
      - "8080:80"
  backend:
    image: app-backend
    ports:
      - "8081:8081"
  db:
    container_name: 'Postgres'
    image: postgres:9.6
    ports:
      - "5432:5432"
    environment:
      - POSTGRES_DB=test_db
    volumes:
      - /tmp/db:/var/lib/postgresql/
```

To start an orchestration of the services run:

\$ docker-compose up

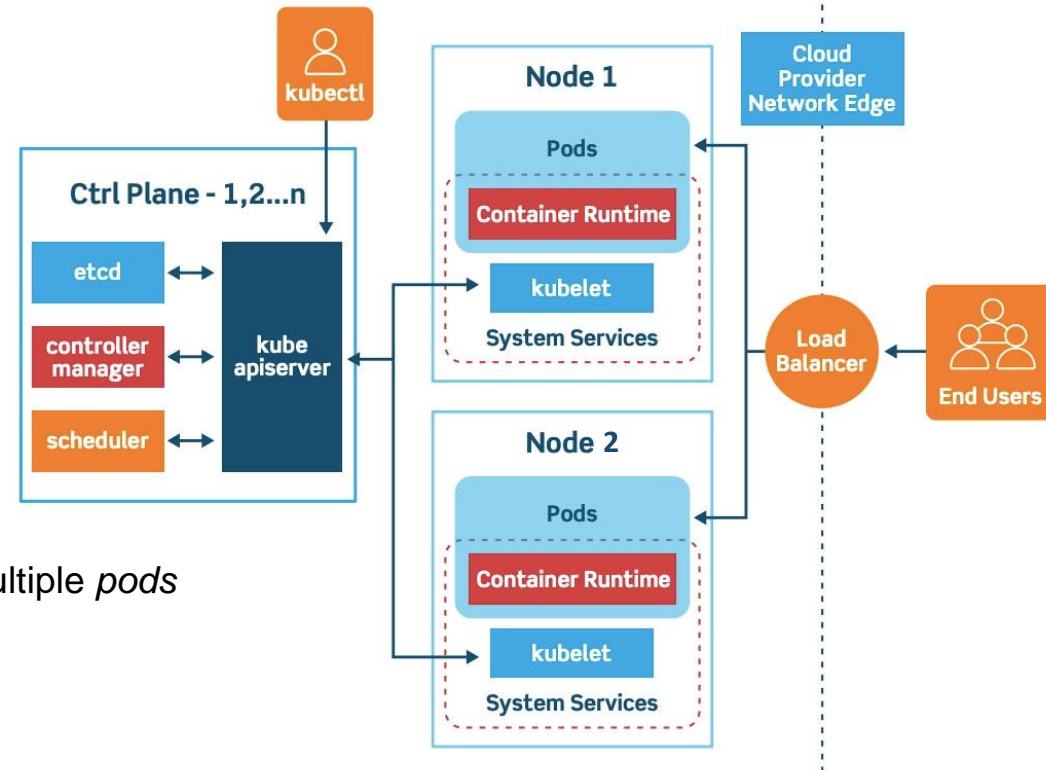
The orchestration can be deleted with:

\$ docker-compose down

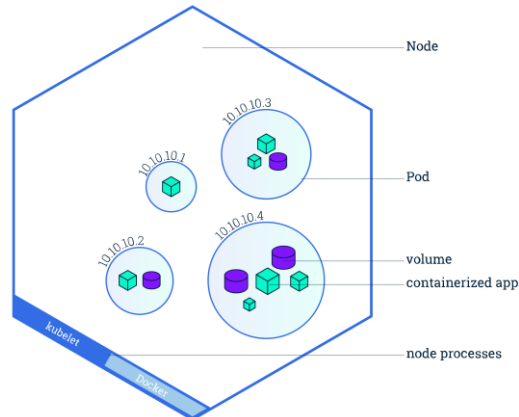
Container Orchestration with Kubernetes (K8s)

Kubernetes environment (high-level):

- Control plane (*master*)
- Distributed storage system (*etcd*)
- Cluster nodes (*kubelets*)



A Kubernetes Node (*kubelet*) consists of multiple *pods* which contain Docker *containers*



[Platform 9, 2019]
[Kubernetes, 2019]

4.1. Virtual machines and containers

4.2. Software architectures for the cloud

“...a style of computing in which scalable and elastic IT-enabled capabilities are delivered as a service to external customers using Internet technologies.”

[Gartner report, 2008]

“...a standardized IT capability (services, software, or infrastructure) delivered via Internet technologies in a pay-per-use, self-service way.”

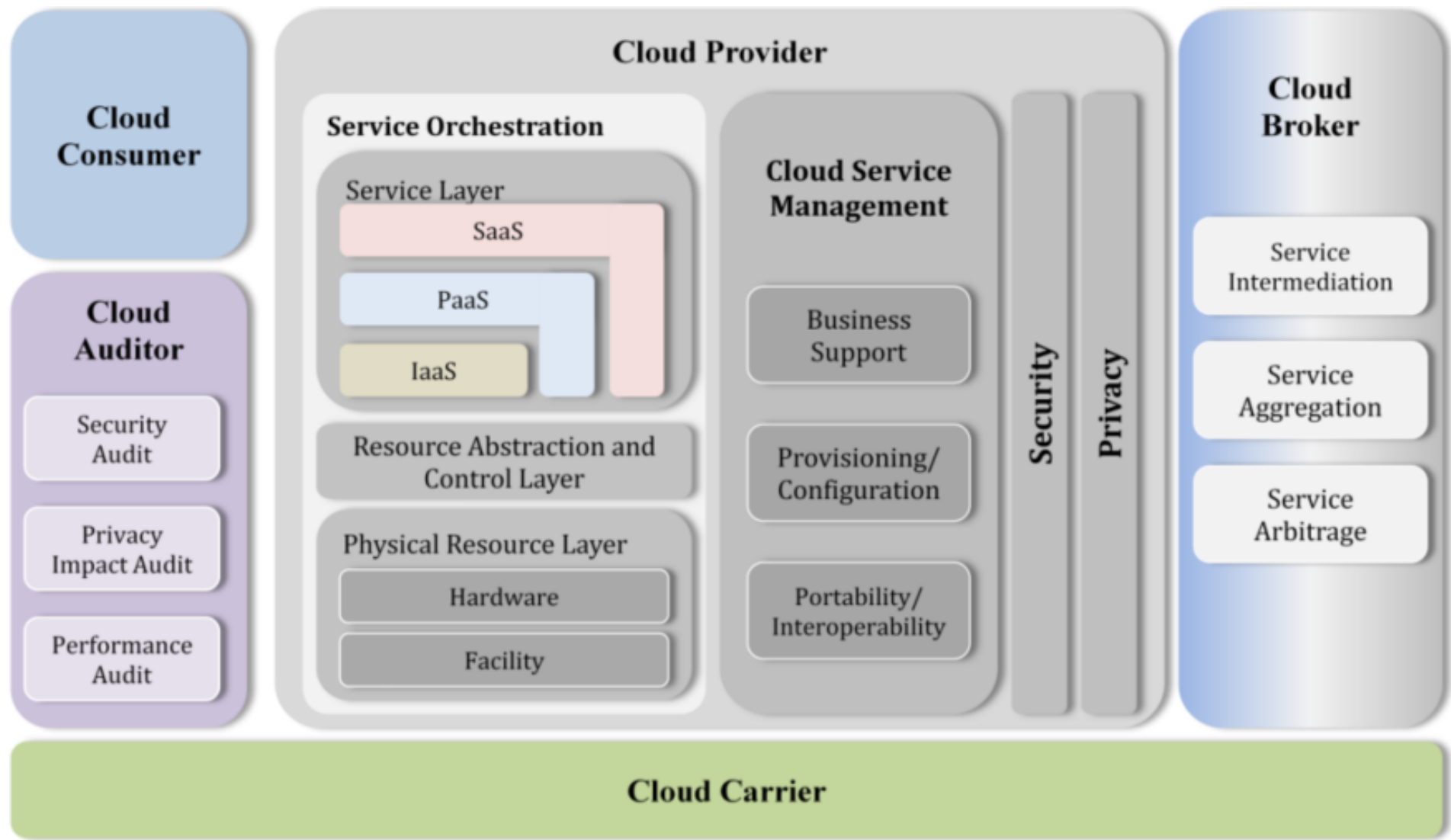
[Forrester research]

“Cloud computing is a specialized form of distributed computing that introduces utilization models for remotely provisioning scalable and measured resources.”

[“Cloud computing - concepts, technology & architecture.” Thomas Erl (2013)]

- **Increased availability and reliability**
 - An IT resource with increased availability is accessible for longer periods of time.
 - The modular architecture of cloud environments provides extensive failover support that increases reliability.
- **Reduced investments and proportional costs**
 - On-demand access to pay-as-you-go computing resources (on a short-term basis such as processors by the hour), and the ability to release these computing resources when they are no longer needed
- **Increased scalability**
 - By providing pools of IT resources, along with tools and technologies designed to leverage them collectively, clouds can instantly and dynamically allocate IT resources to cloud consumers, on-demand or via the cloud consumer's direct configuration.

The cloud reference architecture



["NIST cloud computing reference architecture." Liu, Fang, et al. (2011)]

- **Cloud consumer**

A person or organization that maintains a business relationship with, and uses service from, cloud providers.

- **Cloud provider**

A person, organization, or entity responsible for making a service available to interested parties.

- **Cloud auditor**

A party that can conduct independent assessment of cloud services, information system operations, performance and security of the cloud implementation.

- **Cloud broker**

An entity that manages the use, performance and delivery of cloud services, and negotiates relationships between cloud providers and cloud consumers.

- **Cloud carrier**

An intermediary that provides connectivity and transport of cloud services from cloud providers to cloud consumers.

On-demand self-service

- Get computing capabilities as needed automatically
- A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider

Broad network access

- Capabilities are available over the network using mobile phones, tablets, laptops, workstations etc.

Resource pooling

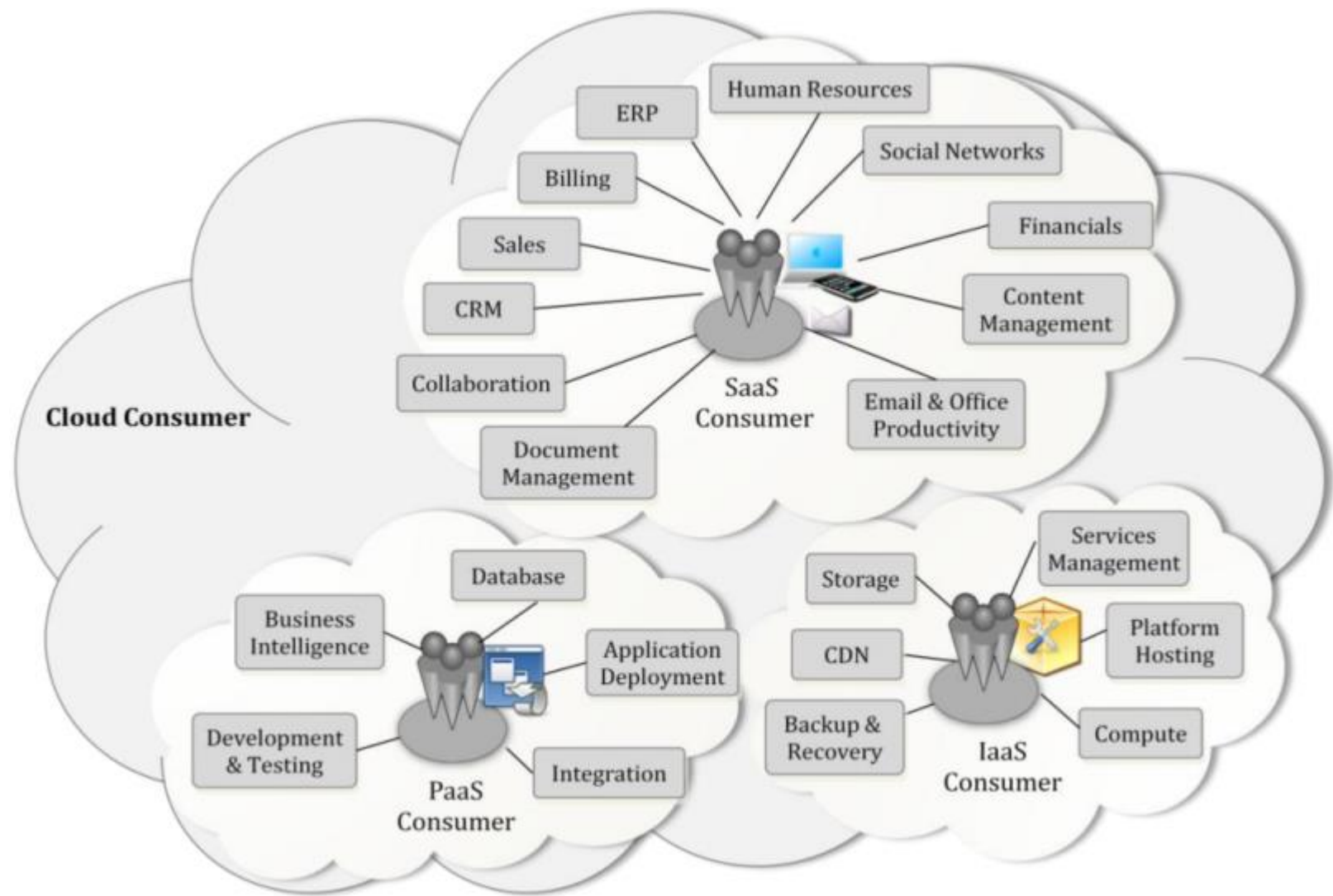
- The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model
- Location independence
- Examples of resources include storage, processing, memory, and network bandwidth

Rapid elasticity

- Ability to quickly scale in/out service

Measured service

- Cloud systems automatically *control* and *optimize* resources based on metering at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts).



[“NIST cloud computing reference architecture.” Liu, Fang, et al. (2011)]

Software as a Service (SaaS)

The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure

The applications are accessible from various client devices through either

- A thin client interface, such as a web browser (e.g., web-based email)
- A program interface

The consumer does not manage or control the underlying cloud infrastructure including

- Network
- Servers
- Operating systems,
- Storage

Examples: Google Docs, Microsoft office online ...

The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider.

The consumer does not manage the underlying cloud infrastructure including

- Network
- Servers
- Operating systems
- Storage
- But has control over the deployed applications and possibly configuration settings for the application-hosting environment.

Example: Google AppEngine - provides a programmable platform that can scale easily

Infrastructure as a Service (IaaS)

The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications

The consumer does not manage or control the underlying cloud infrastructure but has control over

- Operating systems
- Storage
- Deployed applications
- Possibly limited control of select networking components (e.g., host firewalls)

Example: Amazon EC2 – consumers can rent virtualized hardware, can control the software stack on the rented machines

The capability provided to the consumer is to delegate the operational logic of managing a server to a provider who handles provisioning, maintaining, and scaling of the server using an event-driven computing model that only charges for the execution time of the deployed functions

- Consumer has **no concern** for
 - Managing a server host (deploy, run, monitor & maintain)
 - Scaling
 - Provision
 - Load Balancing
 - Logging
- Consumer **benefits** from
 - Focusing on business logic
 - Auto-scaling based on load (CPU/RAM/...)
 - Auto-provision
 - Demand-based cost

Example: AWS Lambda, Microsoft Azure with Azure Functions, Google Cloud Functions, IBM Cloud Functions

[“The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms.” Eyk, Erwin et. al. (2019)]

Infrastructure as a Service (IaaS)

- Consumer prepurchase a cloud provider for always-on server components
- Server capacity scaled up and down by the consumer based on demand
- Cloud infrastructure continues to run even when the app is not used

Function as a Service (FaaS)

- Consumer pays only for the utilized functions/backend (e.g. when a function deployed on the cloud is called, the consumer pays only for the execution duration of the code)
- The provider automatically allocates resources when an event is triggered
- Automatically scaled based on demand
- Consumer does not pay for the unused infrastructure

Use stateless architecture for

- Multimedia processing - *apply a transformation to an uploaded file*
- Event streaming – *triggered from pub/sub topics, scales based on event stream load*
- Batch file processing - *high-performance, parallel computation*
- REST API – *auto-scale based on demand*
- CI/CD – *automate processes (e.g. Pull Requests triggering automated tests to run)*

Popular applications

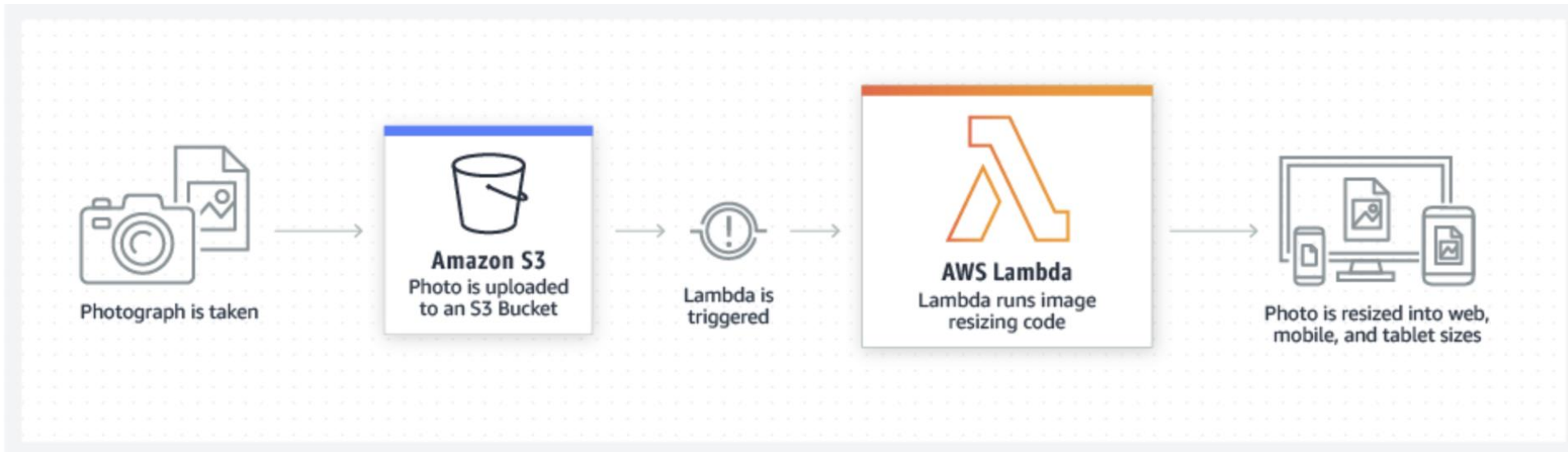
- Web and mobile apps
- Big Retailers (Saturn/Media Markt in Black Friday)
- IoT
- Chatbots
- Payment systems

...

Function as a Service (FaaS) – Serverless Computing

Deploying a REST API using Serverless Framework, AWS Lambda, and DynamoDB

- **Serverless Framework** makes it simpler to deploy and manage a serverless code base by offering full automation.
- Without a framework, one has to manually manage their services through the respective provider's management console. (e.g. AWS console)
- **AWS Lambda** is an event-driven, serverless computing service offered by AWS. It can be integrated with more than 200 AWS services.
- **DynamoDB** is a NoSQL database that can be considered as AWS's version of MongoDB. Like Lambda, it does not require any manual maintenance or scaling.



Function as a Service (FaaS) – Serverless Computing



Deploying a REST API using Serverless Framework, AWS Lambda, and DynamoDB

- In this exercise, we will be deploying a REST API to AWS Lambda using the Serverless Framework.
 1. Setup development environment
 2. Create a route handler for handling business logic
 3. Create *serverless.yml* file which connects routes to endpoints and configures the runtime environment (used by Serverless)
 4. Deploy the endpoints
 5. Remove everything from AWS

[<https://medium.com/swlh/how-to-create-a-serverless-nodejs-rest-api-903e16d80fea>]

Setup development environment

1. Check whether NodeJS and npm are already installed
2. Install Serverless framework with `$ npm install -g serverless` command
3. Run `$ serverless` command in the terminal and fill out the prompted questions (choose *AWS Node.js* as project type)
4. Create an AWS account and access the set of keys for your IAM user
5. Setup AWS configuration using the Serverless framework (*Don't forget to use your own key and secret*)
`$ serverless config credentials --provider aws --key YOUR_KEY_HERE --secret YOUR_KEY_SECRET_HERE`
6. Once the command is run, a profile file will be created under `~/.aws/credentials`. You can modify this file later on when you need to update your AWS credentials
7. Finally, create your *package.json* where you will store your dependencies
`$ npm init -y # initialize package.json`
`$ npm install --save uuid # install uuid package`

Create a route handler for handling business logic

1. Create a file named *users.js* under *routes* folder
2. The following code snippet defines a handler for create user operation (you can find the rest of the operations in the source link)

```
1  const AWS = require("aws-sdk"); // import aws sdk
2  const uuid = require("uuid"); // import uuid generator library
3
4  const dynamoDb = new AWS.DynamoDB.DocumentClient(); // get a db client
5
6  module.exports.createUser = (event, context, callback) => {
7    // create user route handler
8    const data = JSON.parse(event.body);
9
10   const params = {
11     TableName: process.env.DYNAMODB_TABLE,
12     Item: {
13       id: uuid.v4(),
14       email: data.email,
15       firstName: data.firstName,
16       lastName: data.lastName,
17       createdAt: new Date().getTime(),
18       updatedAt: new Date().getTime(),
19     },
20   };
21   dynamoDb.put(params, (error) => {
```

```
22     if (error) {
23       console.error(error);
24
25       callback(null, {
26         statusCode: error.statusCode || 501,
27       });
28       return;
29     }
30
31     callback(null, {
32       statusCode: 200,
33       body: JSON.stringify(params.Item),
34     });
35   });
36 };
```

Create serverless.yml file

1. This file is responsible for connecting the route handlers to the endpoints. In other words, this is where the functions to be deployed are configured (along with other configurations related to the resources)

```
service: serverless-rest-api

frameworkVersion: "2"

provider:
  name: aws
  runtime: nodejs12.x
  environment:
    DYNAMODB_TABLE: ${self:service}-${opt:stage, self:provider.stage}
  iamRoleStatements:
    - Effect: Allow
      Action:
        - dynamodb:Query
        - dynamodb:Scan
        - dynamodb:GetItem
        - dynamodb:PutItem
        - dynamodb:UpdateItem
        - dynamodb>DeleteItem
      Resource: "arn:aws:dynamodb:${opt:region, self:provider.region}:*:table/${self:provider.environment.DYNAMODB_TABLE}"
```

Create serverless.yml file

1. This file is responsible for connecting the route handlers to the endpoints. In other words, this is where the functions to be deployed are configured (along with other configurations related to the resources)

```
functions:
  createUser:
    handler: routes/users.createUser
    events:
      - http:
          path: user
          method: post
          cors: true

  updateUser:
    handler: routes/users.updateUser
    events:
      - http:
          path: user/{id}
          method: put
          cors: true

  deleteUser:
    handler: routes/users.deleteUser
    events:
      - http:
          path: user/{id}
          method: delete
          cors: true
```

[<https://medium.com/swlh/how-to-create-a-serverless-nodejs-rest-api-903e16d80fea>]

Create `serverless.yml` file

1. This file is responsible for connecting the route handlers to the endpoints. In other words, this is where the functions to be deployed are configured (along with other configurations related to the resources)

```
resources:
  Resources:
    TodosDynamoDbTable:
      Type: "AWS::DynamoDB::Table"
      DeletionPolicy: Retain
      Properties:
        AttributeDefinitions:
          - AttributeName: id
            AttributeType: S
        KeySchema:
          - AttributeName: id
            KeyType: HASH
        ProvisionedThroughput:
          ReadCapacityUnits: 1
          WriteCapacityUnits: 1
        TableName: ${self:provider.environment.DYNAMODB_TABLE}
```

Deploy Endpoints

1. Run `$ serverless deploy` to deploy your functions to AWS
2. Once the deployment is done, you will get the locations for your endpoints

```
Service Information
service: serverless-rest-api
stage: dev
region: us-east-1
stack: serverless-rest-api-dev
resources: 35
api keys:
  None
endpoints:
  POST - https://oq9vkzfud9.execute-api.us-east-1.amazonaws.com/dev/user
  PUT - https://oq9vkzfud9.execute-api.us-east-1.amazonaws.com/dev/user/{id}
  DELETE - https://oq9vkzfud9.execute-api.us-east-1.amazonaws.com/dev/user/{id}
  GET - https://oq9vkzfud9.execute-api.us-east-1.amazonaws.com/dev/user/{id}
  GET - https://oq9vkzfud9.execute-api.us-east-1.amazonaws.com/dev/user
functions:
  createUser: serverless-rest-api-dev-createUser
  updateUser: serverless-rest-api-dev-updateUser
  deleteUser: serverless-rest-api-dev-deleteUser
  getUser: serverless-rest-api-dev-getUser
  getUsers: serverless-rest-api-dev-getUsers
layers:
  None
```

[<https://medium.com/swlh/how-to-create-a-serverless-nodejs-rest-api-903e16d80fea>]

Remove everything from AWS

1. When you decide to take down your deployed API, run `$ serverless remove`

Drawbacks

- Not fully controlling your own server-side logic, delegating it to a third-party, can limit the scope of your system
- **Vendor lock-in:** Changing your provider comes with the cost of adapting to the specifications of the new provider (e.g. providers may have different sets of available APIs)
- Difficult to collect data to debug a deployed function as serverless architecture always creates a new version of itself when something goes wrong (requires additional third-party tools to log events)
- Trusting a vendor to run your application's entire backend can raise security concerns when handling sensitive data
- Performance issues due to cold starts

Benefits and Drawbacks of FaaS

- Delegating boilerplate tasks of server management to a third-party
- Reduced operational costs (pay only for the cloud-compute time of your deployed functions)
- Auto-scaling based on demand
- Focus on the business logic

- Vendor lock-in (becoming dependent on services provided by a third-party and facing extra costs when switching to a new provider)
- Harder to debug
- Cold starts
- Security concerns

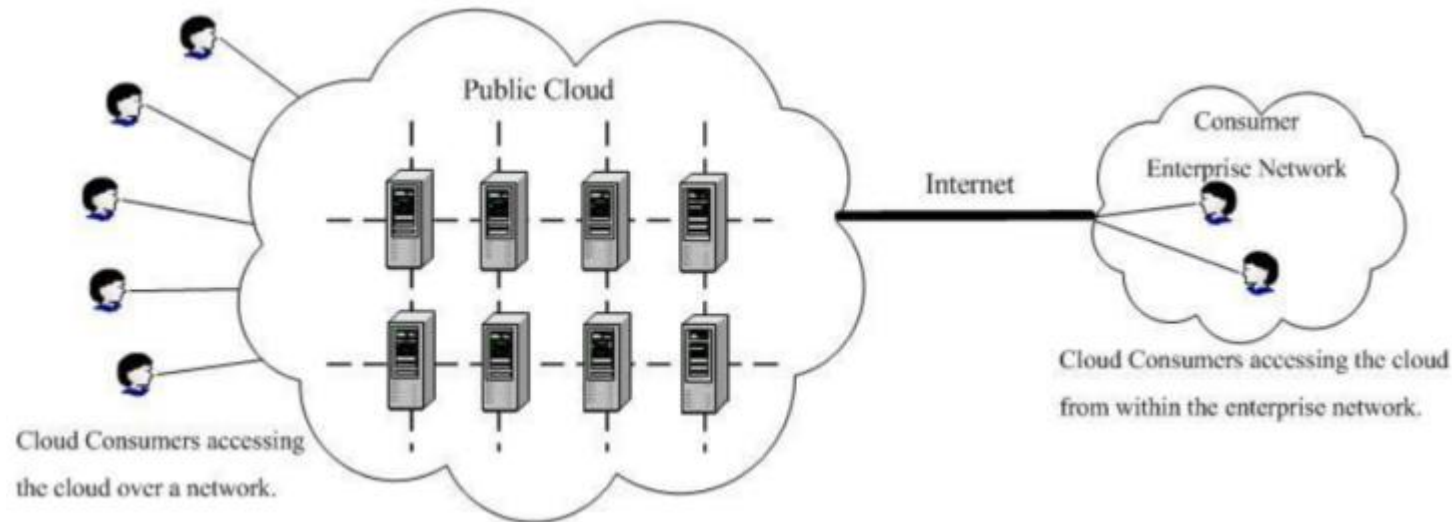
Additional service models

- Network as a service (NaaS)
- Storage as a service (STaaS)
- Security as a service (SECaaS)
- Backend as a service (BaaS)
- Desktop as a service (DaaS)
- Test environment as a service (TEaaS)
-

Deployment model

Public cloud

The cloud infrastructure is provisioned for open use by the *general public*. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider.

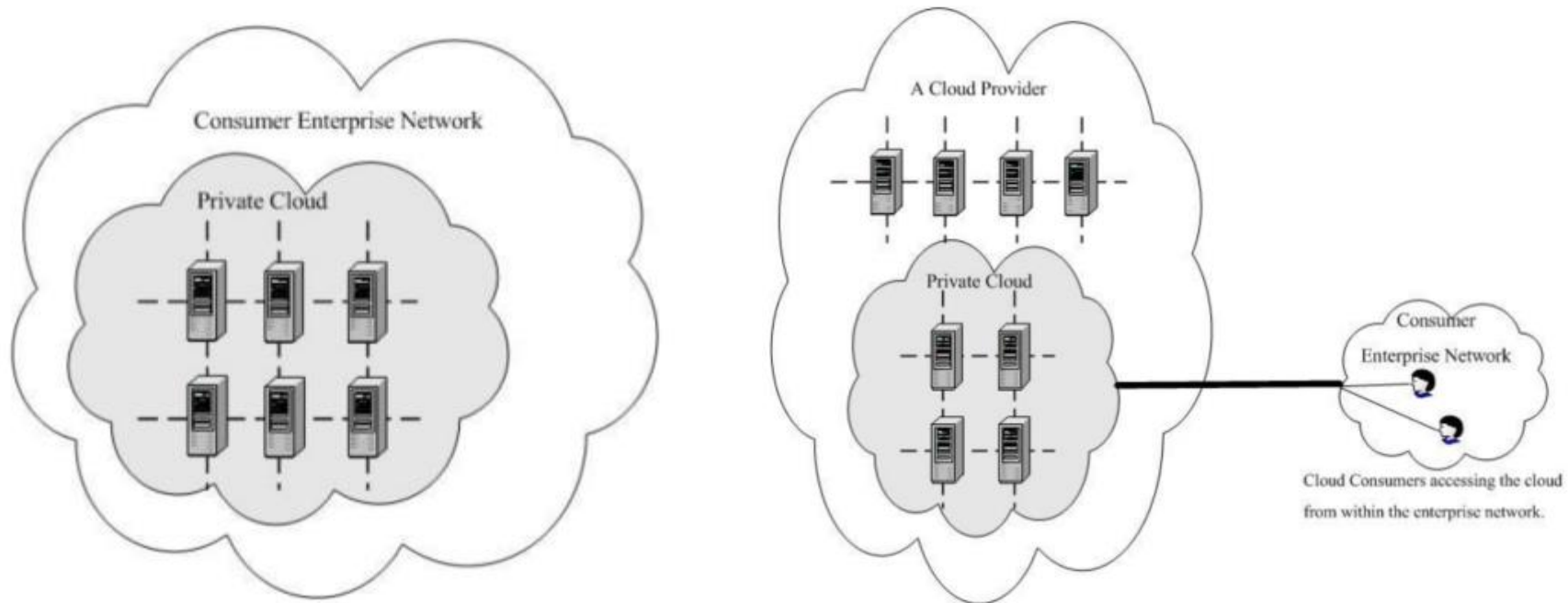


[“NIST cloud computing reference architecture.” Liu, Fang, et al. (2011)]

Deployment model

Private cloud and out-sourced private cloud

The cloud infrastructure is provisioned for exclusive use by a *single organization* comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises.

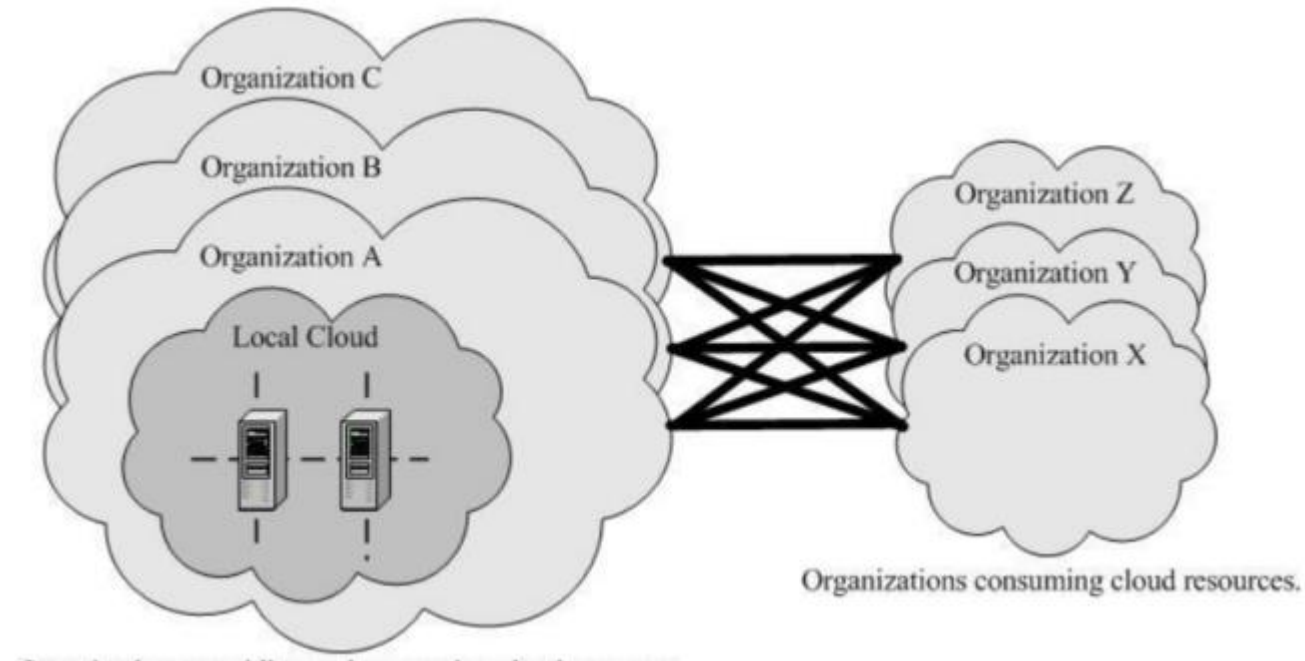


[“NIST cloud computing reference architecture.” Liu, Fang, et al. (2011)]

Deployment model

Community cloud

The cloud infrastructure is provisioned for exclusive use by a *specific community* of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.

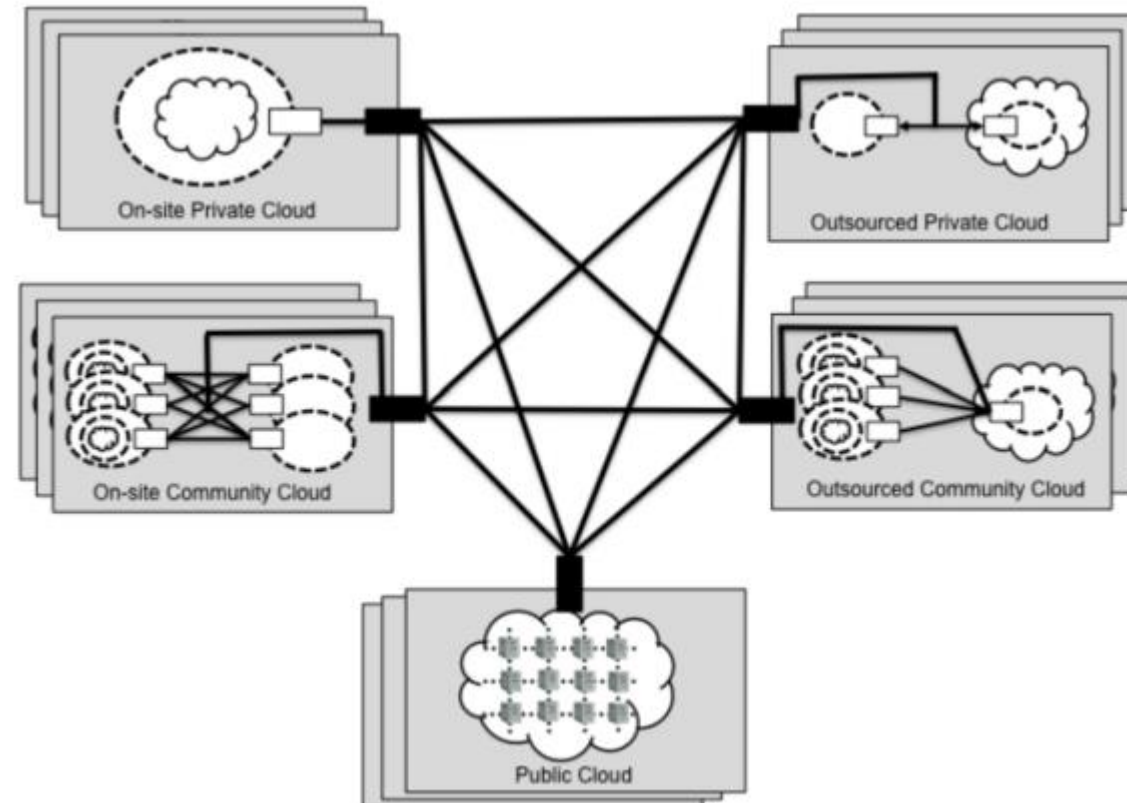


[“NIST cloud computing reference architecture.” Liu, Fang, et al. (2011)]

Deployment model

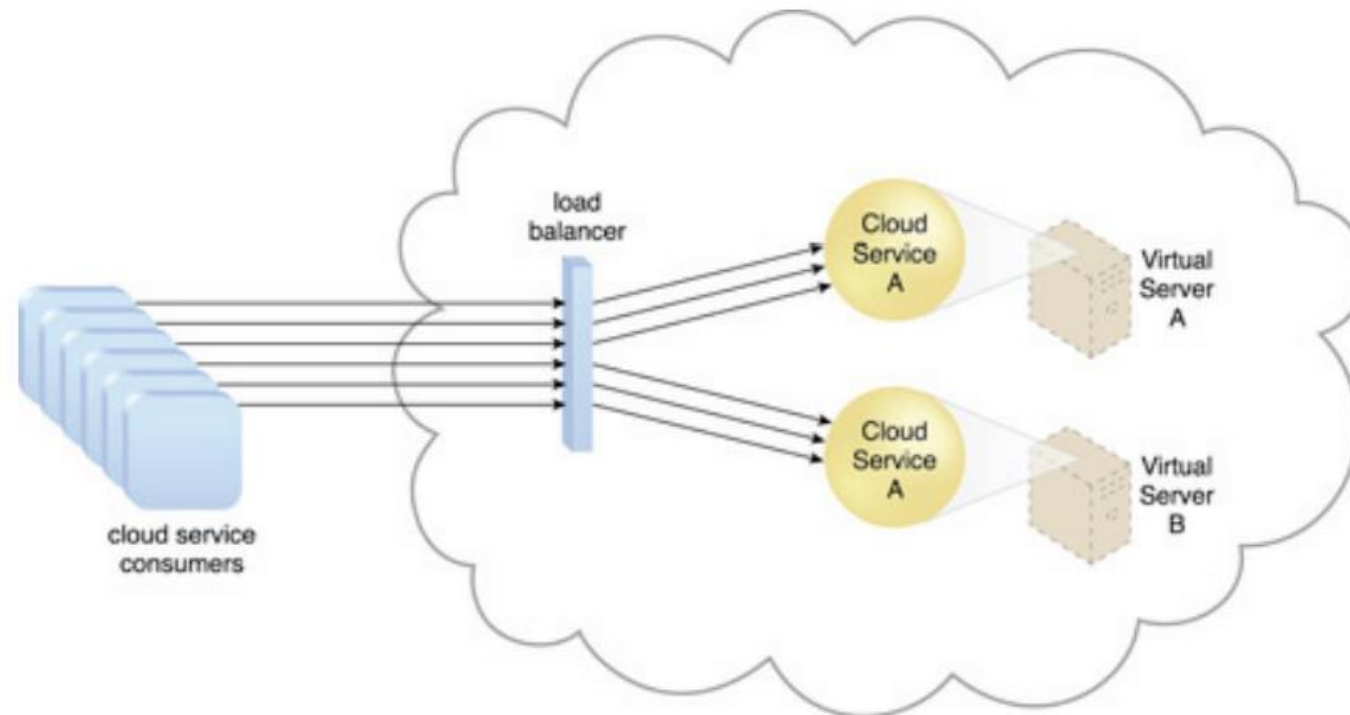
Hybrid model

The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds).



[“NIST cloud computing reference architecture.” Liu, Fang, et al. (2011)]

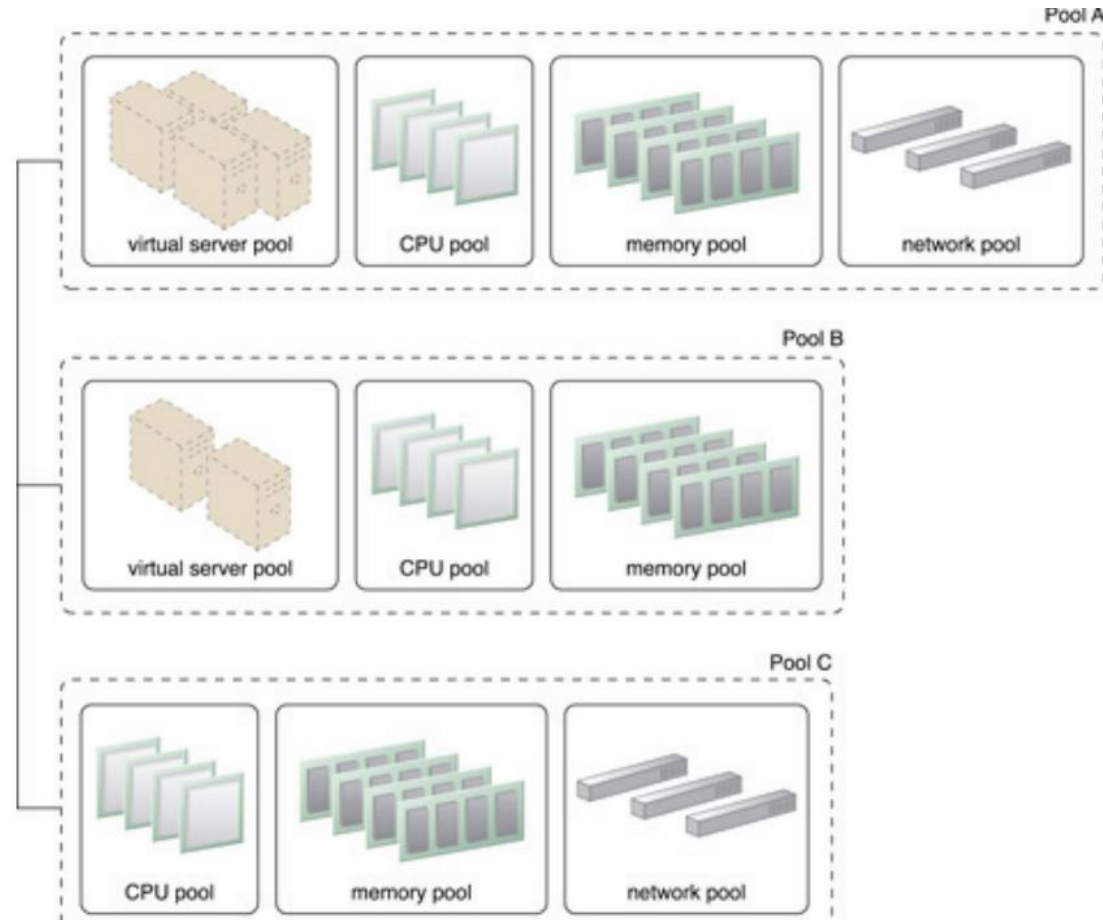
- IT resources can be horizontally scaled via the addition of one or more identical IT resources, and a load balancer that provides runtime logic capable of evenly distributing the workload among the available IT resources
- The resulting workload distribution architecture reduces both IT resource overutilization and under-utilization to an extent dependent upon the sophistication of the load balancing algorithms and runtime logic



[“Cloud computing - concepts, technology & architecture.” Thomas Erl (2013)]

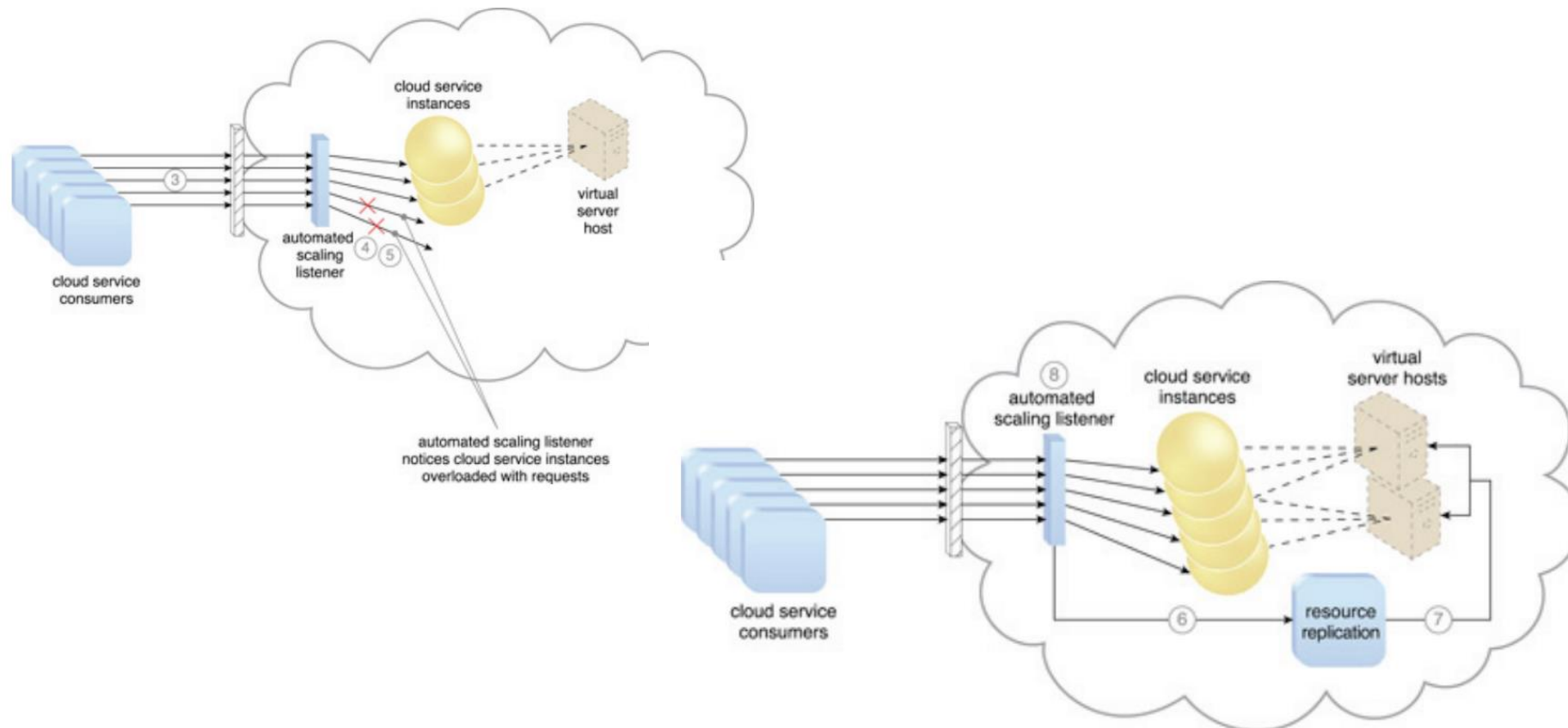
Resource pooling architecture

A resource pooling architecture is based on the use of one or more resource pools, in which identical IT resources are grouped and maintained by a system that automatically ensures that they remain synchronized.



[“Cloud computing - concepts, technology & architecture.” Thomas Erl (2013)]

The dynamic scalability architecture is an architectural model based on a system of predefined scaling conditions that trigger the dynamic allocation of IT resources from resource pools. Dynamic allocation enables variable utilization as dictated by usage demand fluctuations.



[“Cloud computing - concepts, technology & architecture.” Thomas Erl (2013)]

- **Dynamic horizontal scaling**

IT resource instances are scaled out and in to handle fluctuating workloads. The automatic scaling listener monitors requests and signals resource replication to initiate IT resource duplication, as per requirements and permissions.

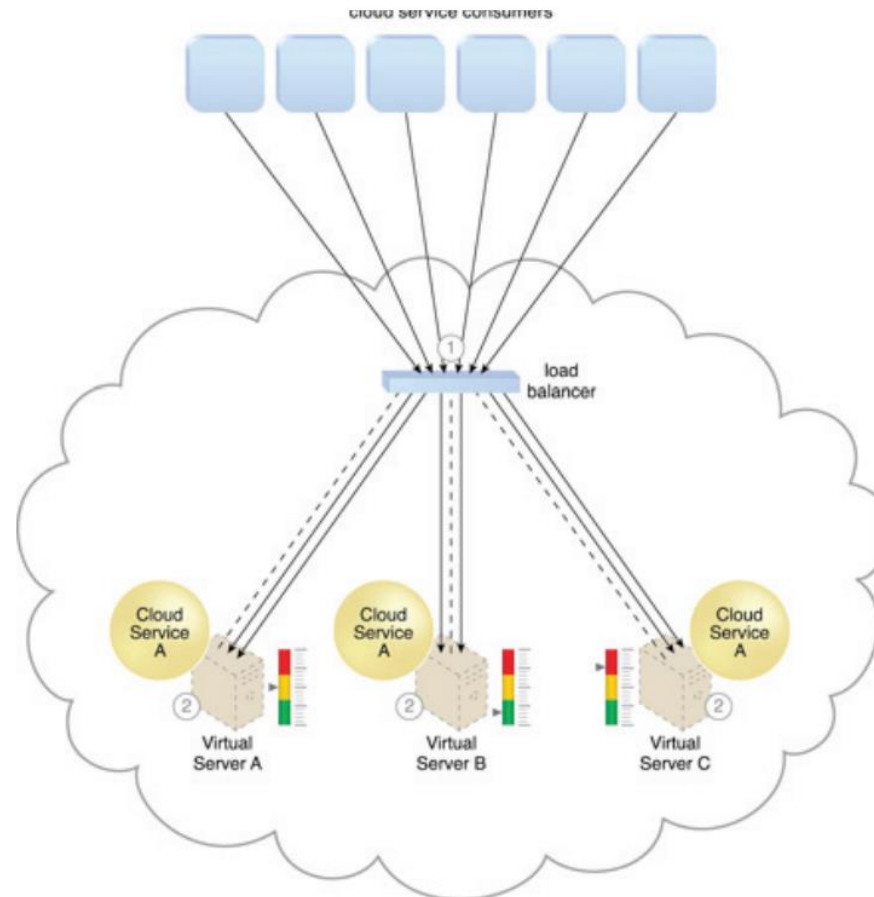
- **Dynamic vertical scaling**

IT resource instances are scaled up and down when there is a need to adjust the processing capacity of a single IT resource. For example, a virtual server that is being overloaded can have its memory dynamically increased or it may have a processing core added.

- **Dynamic relocation**

The IT resource is relocated to a host with more capacity. For example, a database may need to be moved from a tape-based SAN storage device with 4 GB per second I/O capacity to another disk-based SAN storage device with 8 GB per second I/O capacity.

The service load balancing architecture can be considered a specialized variation of the *workload distribution architecture* that is geared specifically for scaling cloud service implementations. Redundant deployments of cloud services are created, with a load balancing system added to dynamically distribute workloads.



[“Cloud computing - concepts, technology & architecture.” Thomas Erl (2013)]

The service load balancing architecture can involve the following mechanisms in addition to the load balancer

- **Cloud usage monitor** - Cloud usage monitors may be involved with monitoring cloud service instances and their respective IT resource consumption levels, as well as various runtime monitoring and usage data collection tasks.
- **Resource cluster** - Active-active cluster groups are incorporated in this architecture to help balance workloads across different members of the cluster.
- **Resource replication** - The resource replication mechanism is utilized to generate cloud service implementations in support of load balancing requirements.

The challenges in cloud architectures

- Clouds are still subject to traditional data confidentiality, integrity, availability, and privacy issues.
- **Confidentiality**
 - Will the sensitive data stored on a cloud remain confidential? Will cloud compromises leak confidential client data (i.e., fear of loss of control over data)
 - Will the cloud provider itself be honest and won't peek into the data?
- **Integrity**
 - How do I know that the cloud provider is doing the computations correctly?
 - How do I ensure that the cloud provider really stored my data without tampering it?
- **Availability**
 - Will critical systems go down at the client, if the provider is attacked in a Denial of Service attack?
 - What happens if cloud provider goes out of business?

The challenges in cloud architectures

- **Privacy issues** raised via massive data mining
 - Cloud stores data from a lot of clients, and can run data mining algorithms to get large amounts of information on clients
- **Increased attack surface**
 - Entity outside the organization now stores and computes data, and so
 - attackers can now target the communication link between cloud provider and client
 - cloud provider employees can be phished
- **Auditability and forensics**
 - Difficult to audit data held outside organization in a cloud
 - Forensics also made difficult since now clients don't maintain data locally
- **Legal and trust issues**
 - Who is responsible for complying with regulations (e.g., SOX, HIPAA, GLBA)?
 - If cloud provider subcontracts to third party clouds, will the data still be secure?

["Security and privacy in cloud computing." Ragib Hasan (2010)]

The challenges in cloud architectures

[Cloud computing] is a security nightmare and it can't be handled in traditional ways.

[John Chambers. CISCO CEO]