T09 Software Deployment and Monitoring

Dr. Jörg Thalheim
Systems Research Group
https://dse.in.tum.de/

Tutorial outline

- **Part I: Lecture summary**
    - **Q&A for the lecture material**

- **Part II:** Programming basics

- **Part III:** Homework programming exercises (Artemis)

Lecture overview

- **Part I:** Deployment models in the cloud
  - Baremetal, virtual machines, containers, and serverless
- **Part II:** Hello world in the cloud
  - Development and deployment of a simple application in the cloud
- **Part III:** Orchestrating in the cloud
  - Deployment and orchestrating a microservice in the cloud
- **Part IV:** System monitoring
  - Background about monitoring and its importance
  - Metrics, alerting, logging, tracing

Software deployment models

- Software deployment
  - Process of delivering software from a development environment to a live environment
- Stages
  - Testing
  - Packaging
  - Installation
  - Configuration
  - Validation

Software deployment ensures that the software is delivered to users in a reliable and efficient manner while minimizing disruptions

Software deployment models

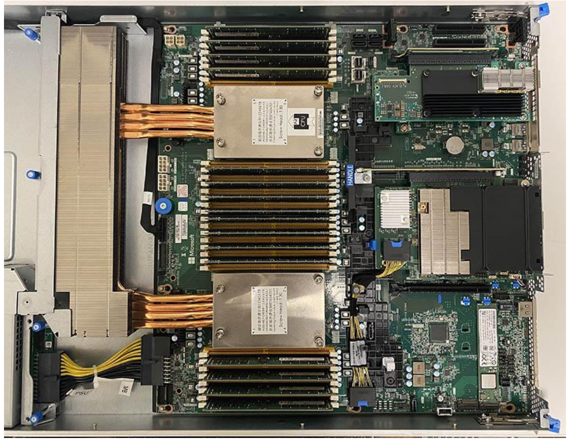
	A. Baremetal	B. Virtual machines	C. Containers	D. Serverless
You manage:	Physical machine + OS + Application	OS + Application	Application & Dependencies	Application Code

Higher flexibility

Less responsibilities

Baremetal: Introduction

- **Baremetal:** Installation and configuration of an operating system and other software directly onto physical hardware



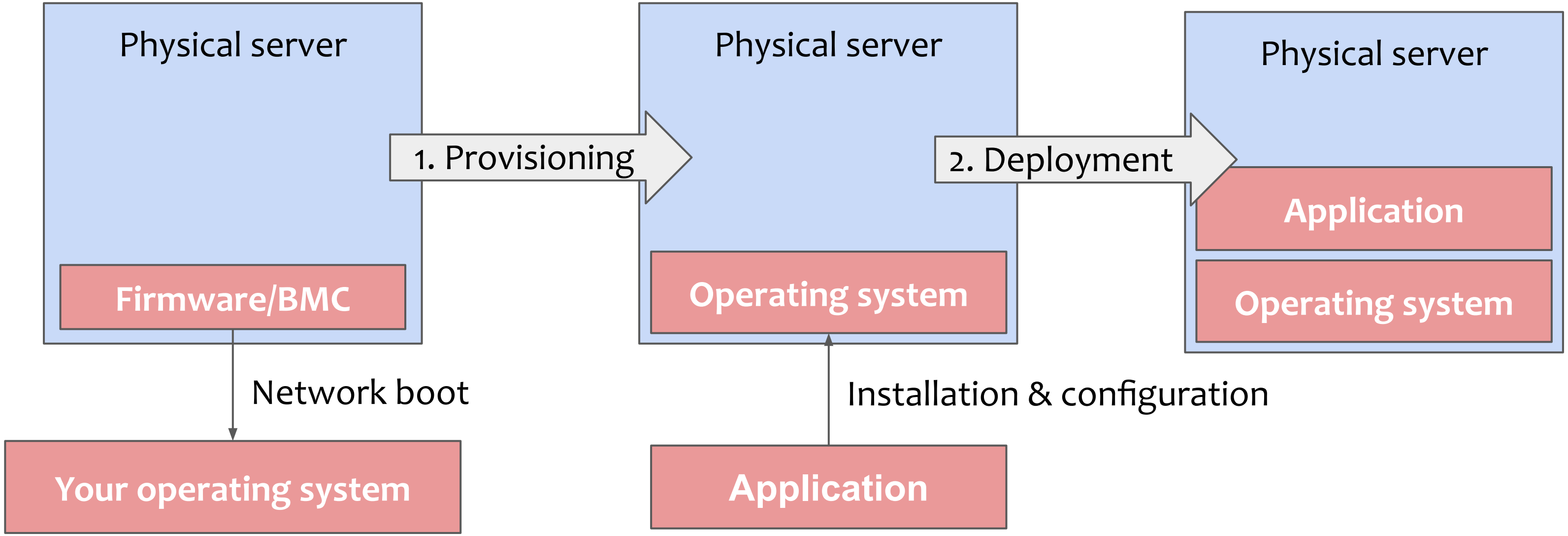
Server used by Microsoft Azure
Source: https://specbranch.com/posts/one-big-server/

Azure cloud server rack

Physical hardware is the foundation for all other deployments!

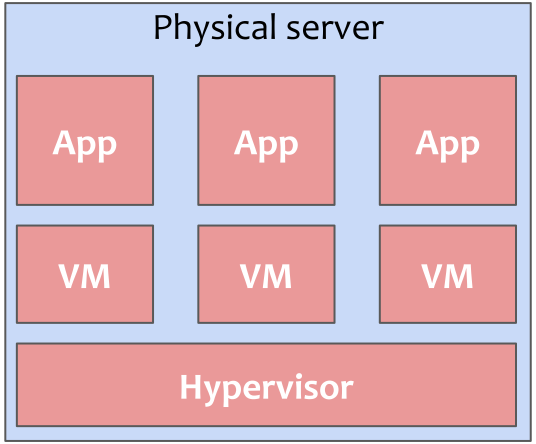
Baremetal deployment

- **Provisioning**
  - Installation through network boot
- **Deployment**
  - Install & configure application, i.e., using configuration management



Virtual machines: Introduction

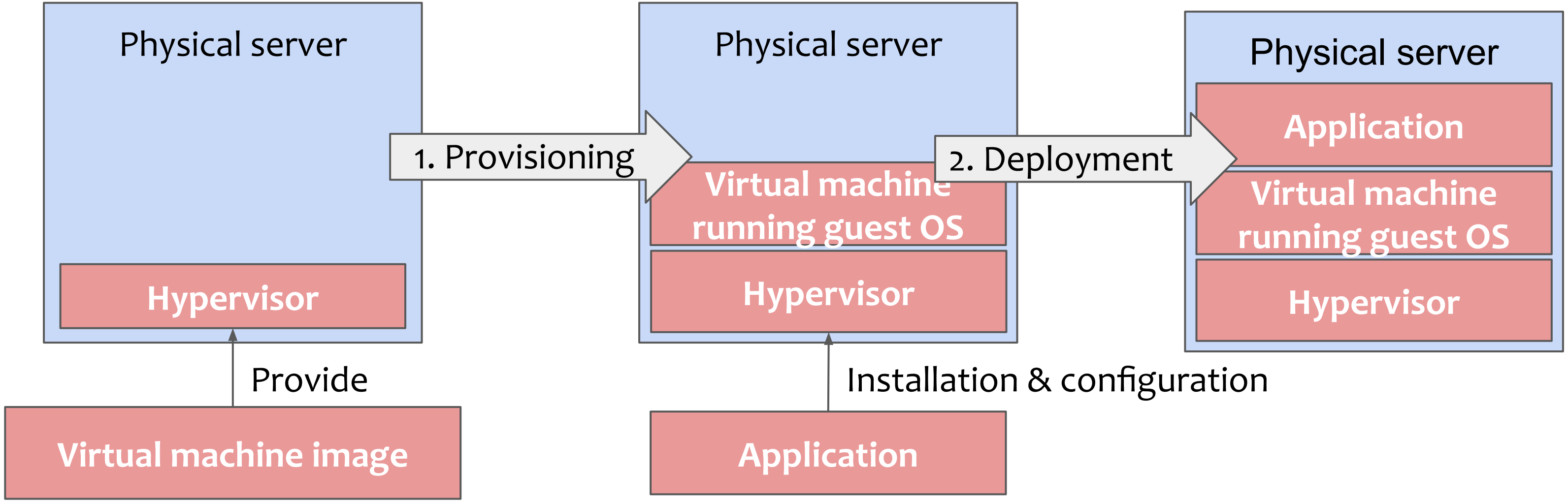
- Physical server gets shared between multiple virtual machines
- Each virtual machine runs its own operating system
- Resources are shared between different tenants



Virtual machines allow to multiplex physical hardware by simulating virtual hardware for each customer

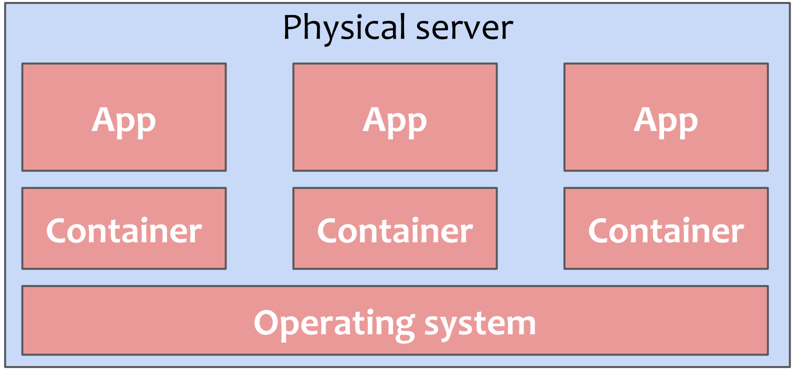
Virtual machines deployment

- **Provisioning**
  - Create virtual machine via cloud provider API based on VM image
- **Deployment**
  - Install & configure application i.e. using configuration management



Containers: Introduction

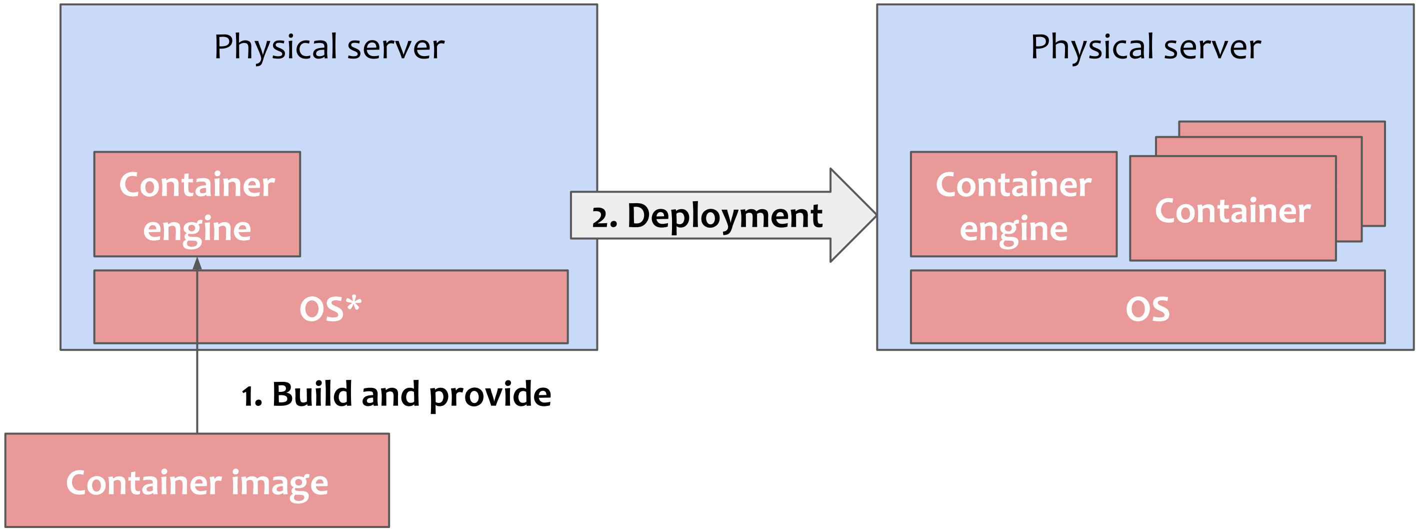
- Linux containers are a lightweight means of virtualizing an operating system and applications
- They are designed to run on a host operating system and share the host's resources
- Linux containers provide isolation between applications and their dependencies, making it easier to manage software deployments



Containers isolate applications purely in software in the operating system without any hardware support

Containers deployment

- **Provisioning**
  - Pull image from registry
- **Deployment**
  - Start container via orchestrator, apply config

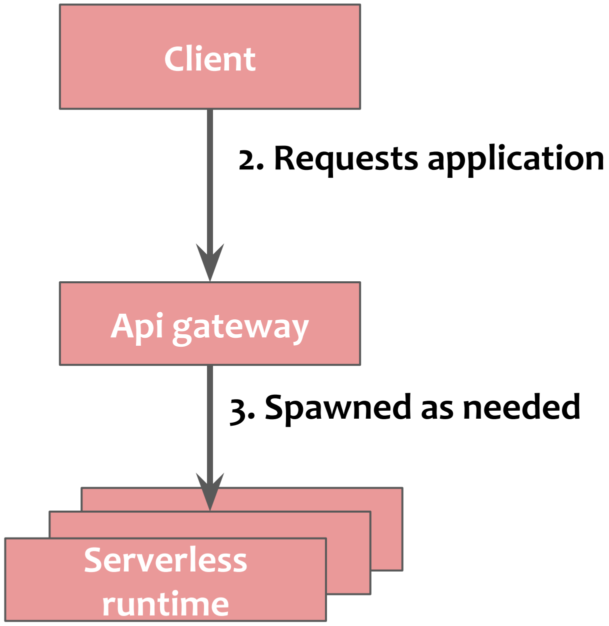


Serverless: Introduction

- Serverless computing is a cloud computing architecture where the cloud provider manages the server infrastructure, allowing developers to focus on their applications

- Developers don't have to worry about server maintenance, scaling or provisioning as it is all taken care of automatically

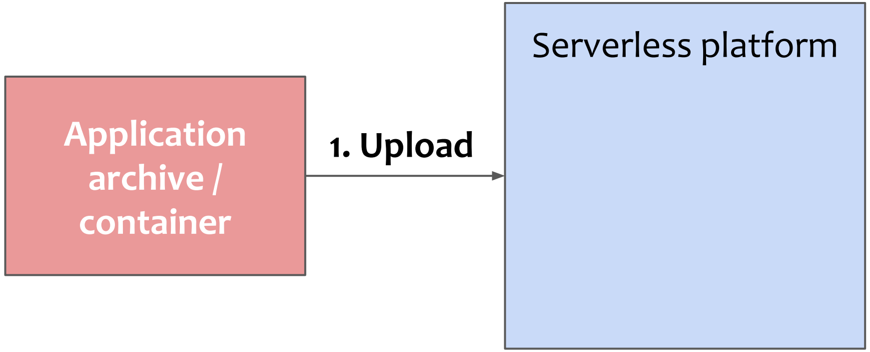
- Serverless is a pay-per-use model where developers only pay for the exact amount of resources their application requires



Provider manages serverless runtimes i.e., using container/VMs for the customer

Serverless deployment

- **Provisioning**
  - Upload function code to serverless provider
- **Deployment**
  - Configure requests triggers & environment
  - The serverless platform executes code in a scalable manner



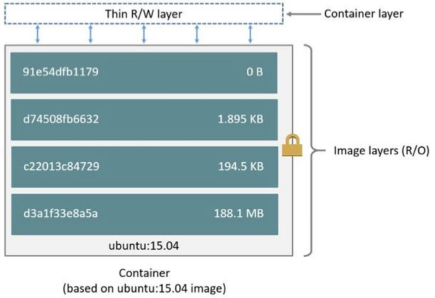
Outline

- ~~**Part I:** Deployment models in the cloud~~
  - ~~Baremetal, virtual machines, containers, and serverless~~
- **Part II:** Hello world in the cloud
  - Development and deployment of a simple application in the cloud
- **Part III:** Orchestrating in the cloud
  - Deployment and orchestrating a microservice in the cloud
- **Part IV:** System monitoring
  - Background about monitoring and its importance
  - Metrics, alerting, logging, tracing

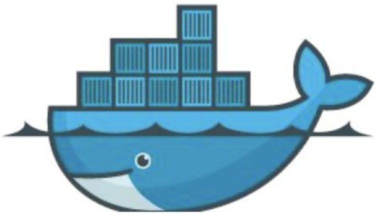
Why docker?

- Package app + deploys once → run **anywhere**
- Isolated, reproducible environments — no “works on my laptop”
- Tiny, versioned images enable rapid roll-back & scaling
- Single CLI → consistent dev → prod workflow

`Dockerfile` →



→



→

Build & test locally

Dockerfile

```
FROM python:3.11-alpine
# This is the directory the app will start from
WORKDIR /app
RUN addgroup -S app && adduser -S app -G app
USER app
# Copy in the dependencies and install them
COPY requirements.txt .
RUN pip install --user --no-cache-dir -r
requirements.txt
# Copy the rest of the source code
COPY . .
# Our application will listen on TCP port 5000 for HTTP
requests
EXPOSE 5000
# Set's the command that gets run when the container
starts
CMD ["python", "app.py"]
```

CLI Commands

```
$ docker build -t hello:v1 .
$ docker run -p 5000:5000 hello:v1
```

- Browse to **localhost:5000**
- You should see “Hello World”

Ship to the cloud

CLI Commands

```
# tag & push to a registry
$ docker tag hello:v1 ghcr.io/<org>/hello:v1
$ docker push ghcr.io/<org>/hello:v1
```

fly.toml

```
app = "tum-greets-the-world"
[[services]]
  internal_port = 5000
  protocol = "tcp"
```

Result: container is scheduled close to users, health-checked, and ready to scale.

Outline

- ~~**Part I:** Deployment models in the cloud~~
  - ~~Baremetal, virtual machines, containers, and serverless~~
- ~~**Part II:** Hello world in the cloud~~
  - ~~Development and deployment of a simple application in the cloud~~
- **Part III:** Orchestrating in the cloud
  - Deployment and orchestrating a microservice in the cloud
- **Part IV:** System monitoring
  - Background about monitoring and its importance
  - Metrics, alerting, logging, tracing

Deployment models

Baremetal

✗ Hard to scale on-demand
✗ Lack of isolation between services

Function-as-a-Service (FaaS)

✔ Scales on-demand
✔ Cost efficient
✗ Limited control of the environment

Containers

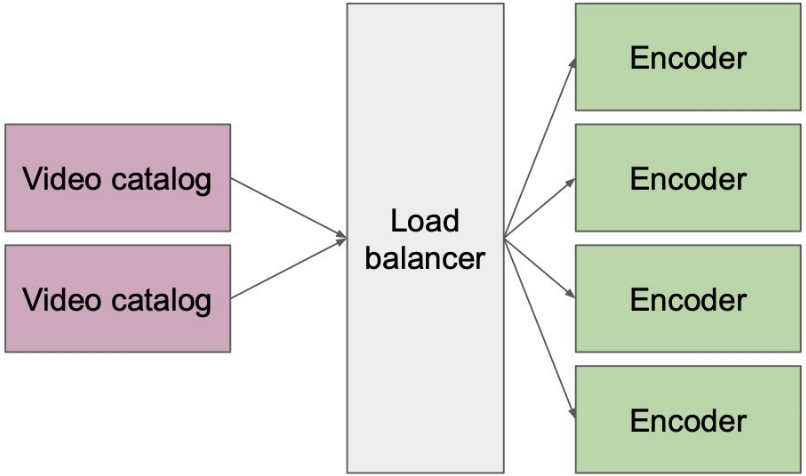
✔ Scales on-demand
✔ Isolation between services (+)
✔ Easy environment packaging

Virtual machines

✔ Scales on-demand
✔ Isolation between services (++)
✔ Easy environment packaging
✗ Cost of virtualization

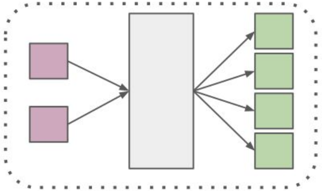
From design to reality

- Logical microservice architecture is tech-agnostic; real clusters aren't.
- Deployment questions to solve:
  1. How many **instances** of each service?
  2. Where do we run them (single host, cluster, multi-region)?
  3. What about databases and stateful components?
  4. Who (or what) automates scheduling & scaling?

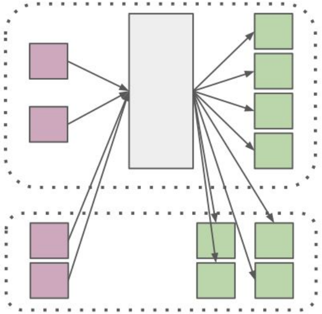


Scaling compute and data

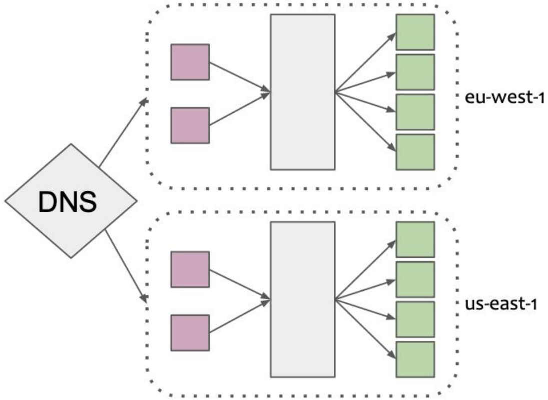
- **Service tier:** add more replicas behind a **load balancer** ⇒ horizontal scaling & fault-tolerance.
- **Database tier:** single primary can bottleneck; add **read replicas** → writes go to primary, reads fan out, replication keeps data fresh.
- Works on one machine, a cluster, or multiple regions (DNS routes traffic).



Single machine



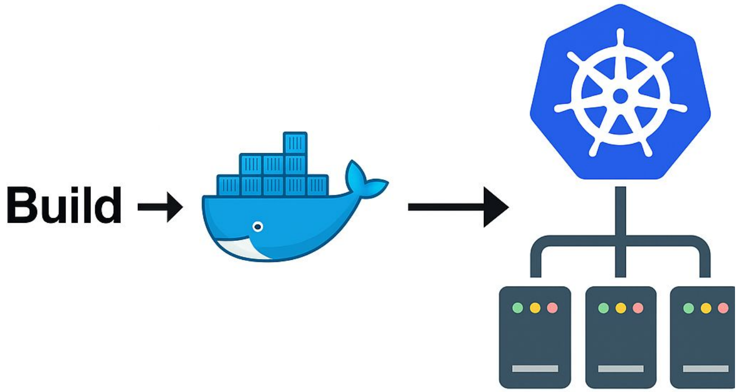
Multiple machines



DNS-based regional balancing

Scaling compute and data

- **Package** each microservice in a container image (Dockerfile lists deps)
- **Ship**: docker build → push → pull from a registry on any host
- **Run & manage** with an **orchestrator** (Kubernetes, Docker Swarm, Fly.io, etc.):
    1. schedules containers on available nodes
    2. auto-scales replicas
    3. restarts on failure & handles rolling updates



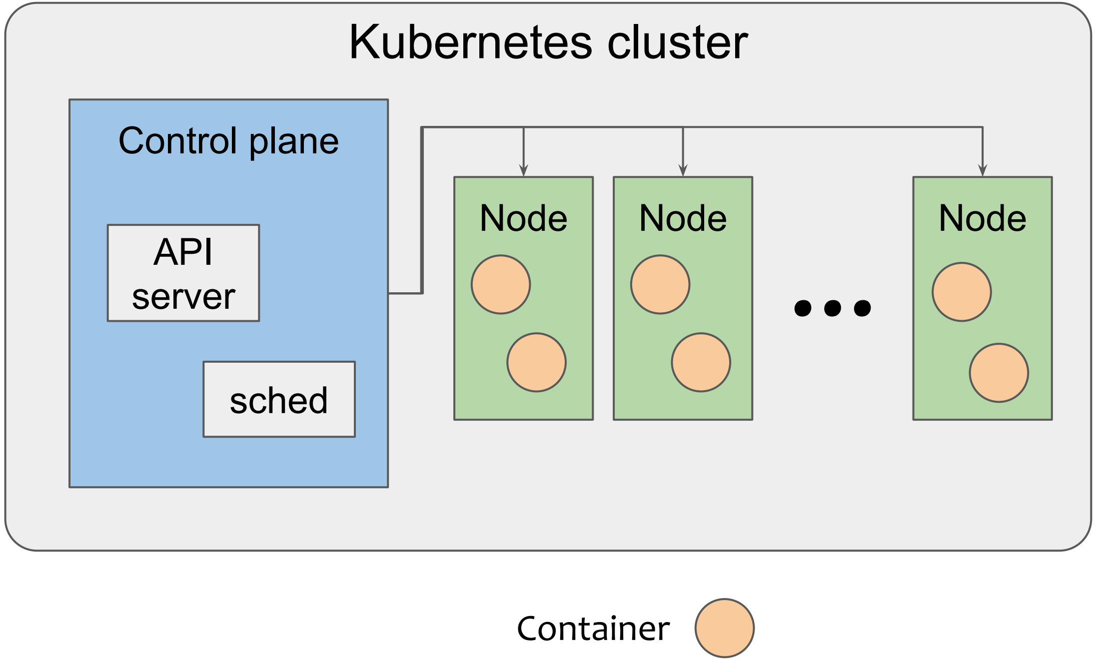
Container orchestration with Kubernetes

Kubernetes is a container orchestrator from Google

- Container deployment
    - Map containers on physical machines
    - Schedule containers
- Network management
    - Service discovery
    - Load balancing
- Scaling up/down
    - Replicate/destroy containers to scale
    - Load balancing

Kubernetes architecture

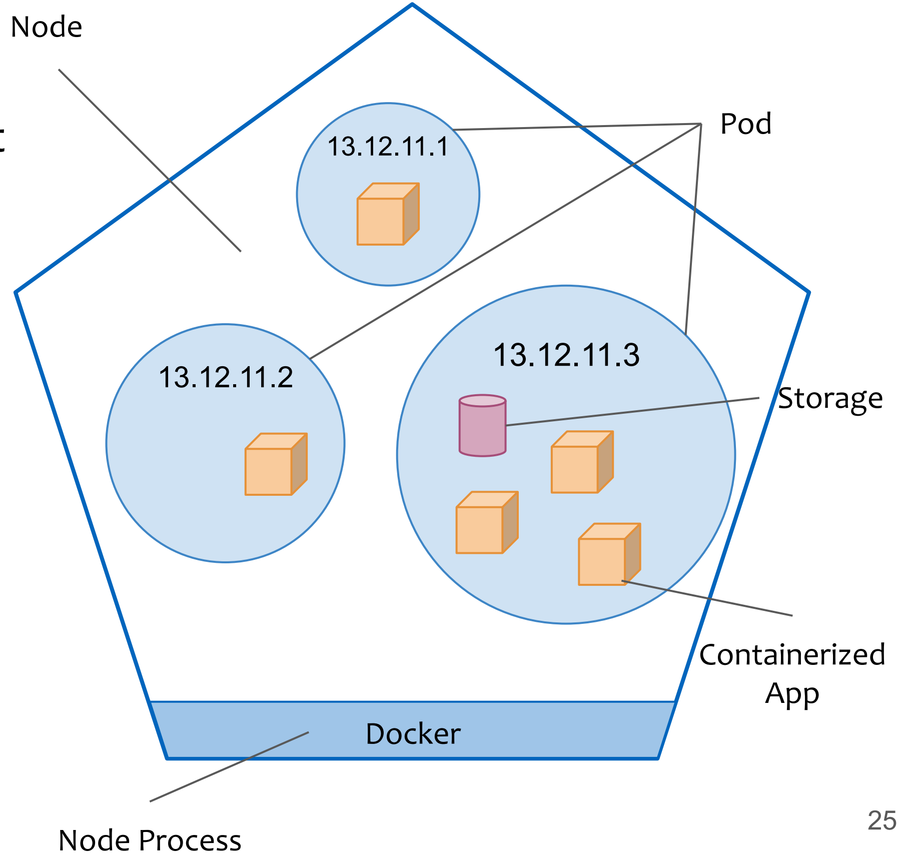
- **Control plane**
  - Manages containers
  - Exposes the control API
  - Schedules containers
- A **node** is a worker machine that runs the containers



Kubernetes pods And nodes

- The Kubernetes' atomic unit of deployment is called a **pod** and includes:
  - One or more application containers
  - Shared storage volume(s)
  - Shared networking, *i.e. IP address*

- A **node** is a physical worker machine that manages pods and containers running on it



Outline

- ~~**Part I:** Deployment models in the cloud~~
    - ~~Baremetal, virtual machines, containers, and serverless~~
- ~~**Part II:** Hello world in the cloud~~
    - ~~Development and deployment of a simple application in the cloud~~
- **Part III:** Orchestrating in the cloud
    - Deployment and orchestrating a microservice in the cloud
- **Part IV:** System monitoring
    - Background about monitoring and its importance
    - Metrics, alerting, logging, tracing

System monitoring

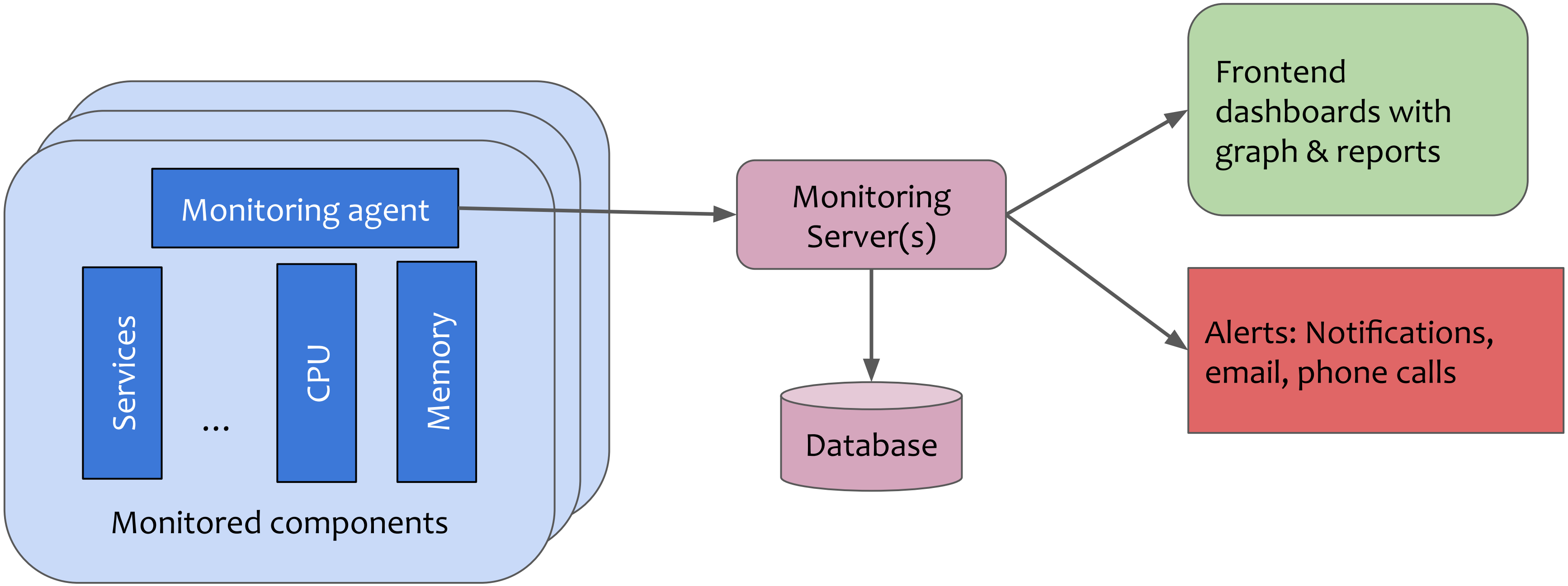
What is Monitoring?

- Monitoring is the process of continuously observing and analyzing the operations of an application or system to ensure it operates at peak performance

Types of Monitoring

- **Metrics:** Quantifiable measurements that provide insight into system behavior under different loads. Examples include CPU usage, memory consumption, disk I/O, etc. → **What is happening?**
- **Logging:** Records of events happening in the system, useful for debugging and understanding system behavior → ***What events occurred?***
- **Tracing:** Analyzing individual operations, such as user requests, as they flow through various components of a system → ***Where is the problem?***

Common monitoring architecture



Types of metrics

- **System metrics:** Include measurements related to CPU utilization, memory usage, network I/O, disk I/O, etc.

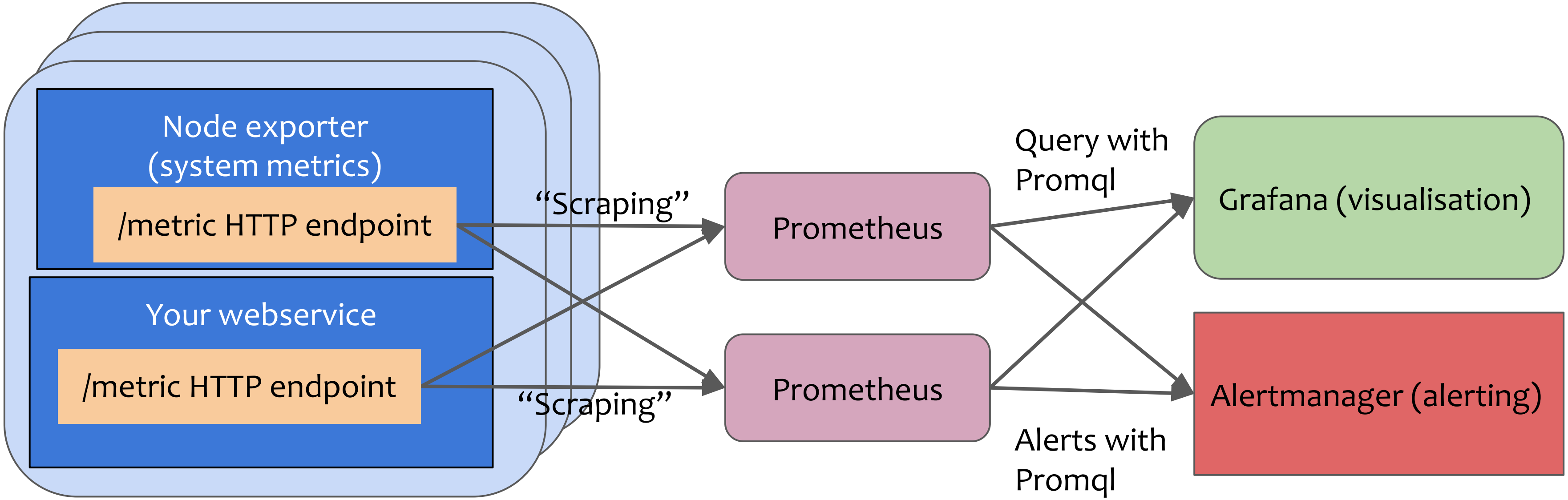
- **Application metrics:** These are specific to the application and include measures like response time, throughput, error rates, active users, etc.

- **Business metrics:** These focus on the business impact and include measurements like user engagement, conversion rate, customer acquisition cost, etc.

Metrics collection with Prometheus

Prometheus is an open-source systems monitoring and alerting toolkit

- Designed for **reliability**, handling multiple metrics **efficiently**



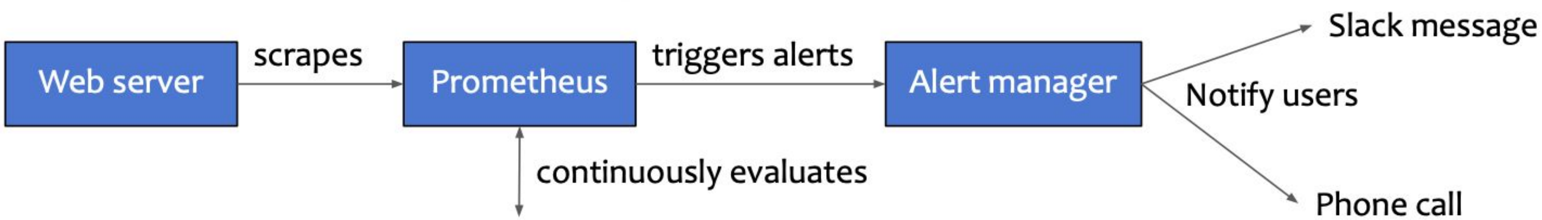
Metrics visualization

Grafana is the most popular software for observability

- Generates graphs based on (PromQL)-Queries
- Example showing system metrics from Node-Exporter

Metrics alerting

Example

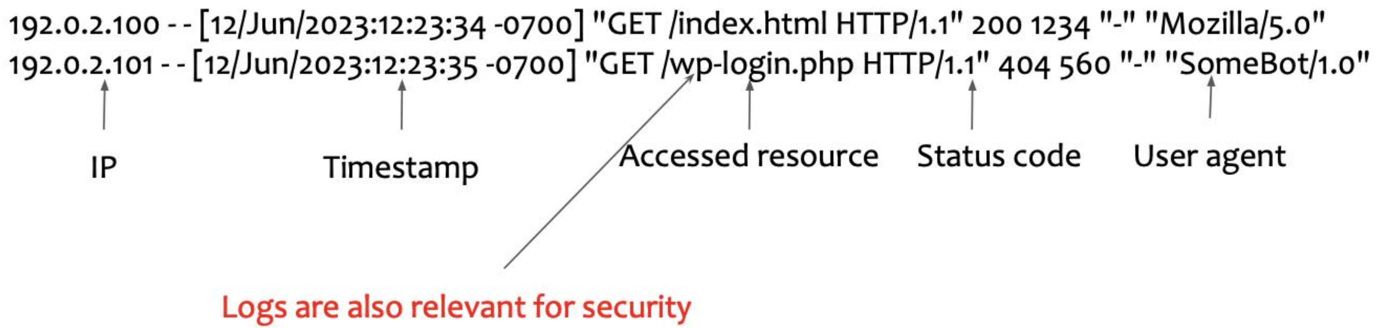


Alert rule:

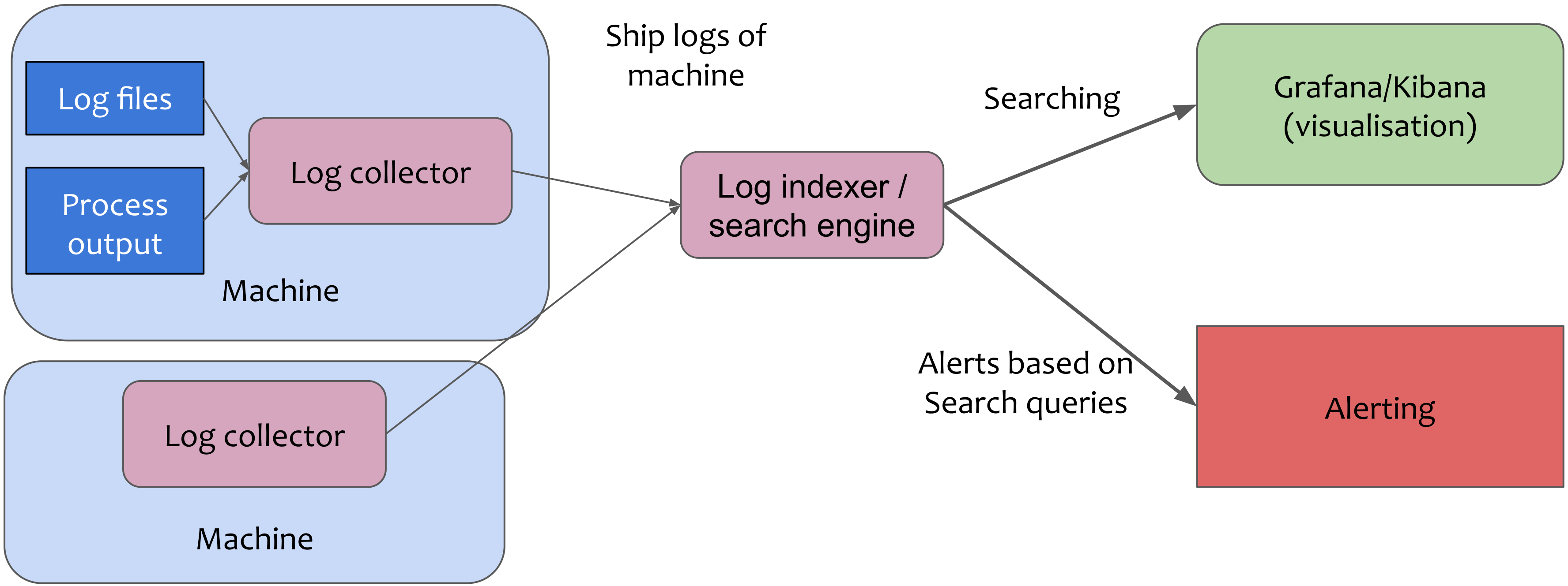
```
groups:
- name: webserver
  rules:
  - alert: HighServerErrorRate
    expr: rate(http_requests_total{status_code=~"5.."}[1m]) > 1
    for: 10m
    labels:
      severity: critical
    annotations:
      summary: "High server error rate detected"
```

Metrics logging

- Use logging when you need a **detailed chronological record of system events** for debugging or audit purposes
- Ideal for **troubleshooting issues**, understanding system behavior, or maintaining records for compliance purposes
- **Log aggregation systems**: Promtail/Grafana Loki, Elasticsearch/Kibana

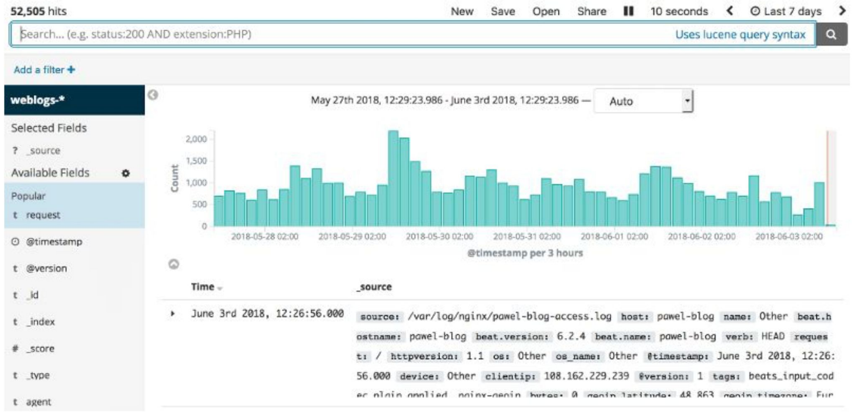


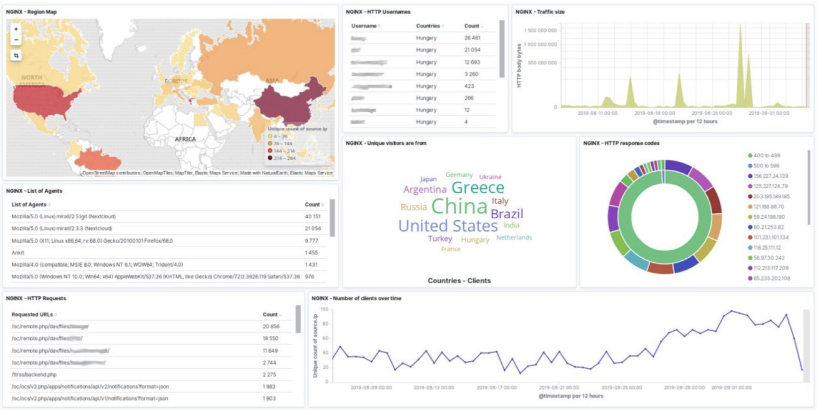
Metrics logging aggregation



Metrics logging visualization

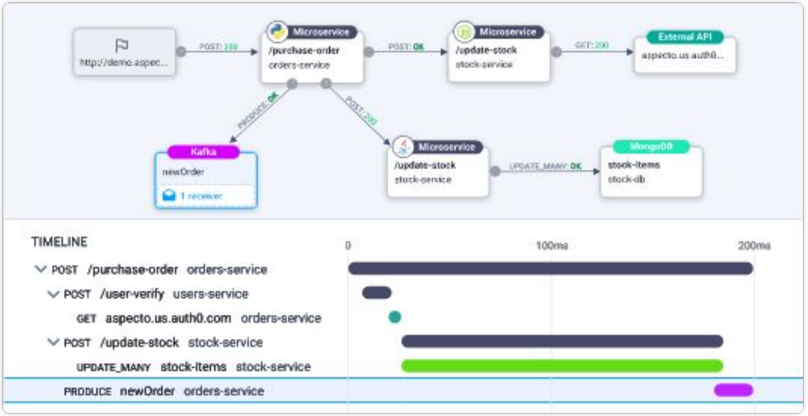
- Central log aggregation parses metadata from logs and allows to filter by it
- We can use the metadata also to derive metrics and visualisation

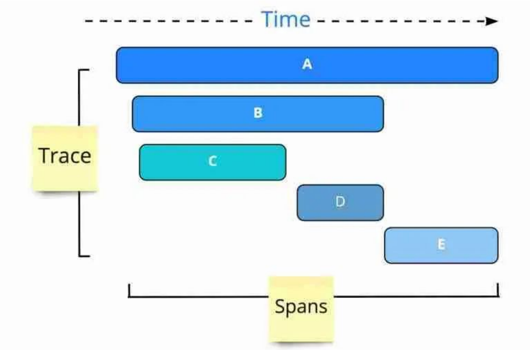




Tracing: Where is the problem?

- **Use tracing** when you need to track a request's journey through various system components to understand its behavior or locate issues
- **Tracing provides detailed visibility** into the lifecycle of a single operation
- **Ideal for debugging complex issues** in microservice architectures, optimizing performance, and improving user experience
- **Examples:** User request tracing, function calls, database queries, etc.
- **Tracing tools:** Jaeger, Sentry, Grafana Tempo





Tutorial outline

- ~~**Part I:** Lecture summary~~
    - ~~Q&A for the lecture material~~

- **Part II:** **Programming basics**

- **Part III:** Homework programming exercises (Artemis)

Programming Basics (PB) exercises

L09PB01 Digitalizing a Michelin Star Restaurant Business [Containers]

Start exercise

Not released | Optional | tutorial | Easy

L09PB02 Digitalizing a Michelin Star Restaurant Business (Part 2) [REST, Containers]

Start exercise

Not released | Optional | tutorial | Easy

L09PB01 Digitalizing a Michelin Star Restaurant Business [Containers]

Goal

- Implement a Client-Server Architecture with Spring Boot using REST APIs and Docker containers

Objectives

- REST API Development (Server-side): Implement robust endpoints for adding, modifying, and deleting customer data
- Client-side Integration: Develop a GUI-driven client to interact with the API
- Docker Deployment: prepare the application for deployment by creating a Dockerfile and a start script

Before You Start - Reactive Web Programming

Traditional web applications can struggle with:

- **Scalability:** Handling many concurrent users or large data volumes
- **Responsiveness:** Keeping the UI fluid and fast, especially with complex data or slow network conditions
- **Fault Tolerance:** Preventing a single failure from bringing down the entire application

Reactive programming offers elegant solutions to these common challenges:

- **Scalability:** Reactive programming uses **non-blocking operations** to handle more concurrent users and data efficiently, freeing up server resources instead of waiting for I/O
- **Responsiveness:** It enables **real-time UI updates** by pushing data changes as they happen, ensuring a fluid user experience without constant polling
- **Fault Tolerance:** It improves stability with robust **error handling** within data streams and **backpressure mechanisms** that prevent system overload and cascading failures

Before You Start - Spring Webflux

Spring WebFlux is a reactive-stack web framework

Concepts in use:

- **Make HTTP requests non-blockingly:** By using *WebClient* application threads are free to do other work while waiting for network responses.
- **Handle asynchronous results:** *subscribe()* provides callbacks to process data once it arrives, rather than waiting for a direct return value.
- **Perform clear error handling:** *onErrorStop()* provides a simple way to manage errors in the reactive flow.
- **Process different response types:** *toBodilessEntity()* for no body, and *bodyToMono()* with ParameterizedTypeReference for complex JSON objects/lists.
- **Update application state:** Using the *Consumer* callback, we can notify other parts of your application about the updated data.

Before You Start - Spring Webflux - Mono

In Spring WebFlux, Mono is a core component for handling asynchronous data streams. It represents a single element (or no element) that may be available asynchronously.

Key aspects of Mono:

- **Single Value or Empty:** It either emits one item or completes without emitting anything.
- **Non-Blocking:** Operations do not block the thread, allowing better resource utilization.
- **Chaining Operations:** Provides operators like *map*, *flatMap*, and *filter* to transform or process data
- **Error Handling:** It provides mechanisms for handling errors that may occur during asynchronous operations

L09PB01 Digitalizing a Michelin Star Restaurant Business [Containers]

```
@PostMapping("customers")
public ResponseEntity<Customer> createCustomer(@RequestBody Customer customer) {
    if (customer.getId() != null) {
        return ResponseEntity.badRequest().build();
    }
    return ResponseEntity.ok(customerService.savePerson(customer));
}

@PutMapping("customers/{customerId}")
public ResponseEntity<Customer> updateCustomer(@RequestBody Customer updatedCustomer,
                                              @PathVariable("customerId") UUID personId) {
    if (!updatedCustomer.getId().equals(personId)) {
        return ResponseEntity.badRequest().build();
    }
    return ResponseEntity.ok(customerService.savePerson(updatedCustomer));
}

@DeleteMapping("customers/{customerId}")
public ResponseEntity<Void> deleteCustomer(@PathVariable("customerId") UUID personId) {
    customerService.deletePerson(personId);
    return ResponseEntity.noContent().build();
}
```

Add the responses for POST

Add the responses for PUT

Add the responses for DELETE

L09PB01 Digitalizing a Michelin Star Restaurant Business [Containers]

```java
public void addPerson(Customer customer, Consumer<List<Customer>> personsConsumer) {
    // TODO Part 2: Make an http post request to the server
    webClient.post()
            .uri("customers")
            .bodyValue(customer)
            .retrieve()
            .bodyToMono(Customer.class)
            .onErrorStop()
            .subscribe(newCustomer -> {
                customers.add(newCustomer);
                personsConsumer.accept(customers);
            });
}
public void updatePerson(Customer customer, Consumer<List<Customer>> personsConsumer) {
    // TODO Part 2: Make an http put request to the server
    webClient.put()
            .uri("customers/" + customer.getId())
            .bodyValue(customer)
            .retrieve()
            .bodyToMono(Customer.class)
            .onErrorStop()
            .subscribe(newCustomer -> {
                customers.replaceAll(oldCustomer ->
                    oldCustomer.getId().equals(newCustomer.getId()) ? newCustomer : oldCustomer);
                personsConsumer.accept(customers);
            });
}
```

Finish the POST request

Finish the PUT request

L09PB01 Digitalizing a Michelin Star Restaurant Business [Containers]

```
public void deletePerson(Customer customer, Consumer<List<Customer>> personsConsumer) {
    webClient.delete()
            .uri("customers/" + customer.getId())
            .retrieve()
            .toBodilessEntity()
            .onErrorStop()
            .subscribe(v -> {
                customers.remove(customer);
                personsConsumer.accept(customers);
            });
}
public void getAllPersons(CustomerSortingOptions sortingOptions, Consumer<List<Customer>> personsConsumer) {
    webClient.get()
            .uri(uriBuilder -> uriBuilder
                    .path("customers")
                    .queryParam("sortField", sortingOptions.getSortField())
                    .queryParam("sortingOrder", sortingOptions.getSortingOrder())
                    .build())
            .retrieve()
            .bodyToMono(new ParameterizedTypeReference<List<Customer>>() {})
            .onErrorStop()
            .subscribe(newPersons -> {
                customers.clear();
                customers.addAll(newPersons);
                personsConsumer.accept(customers);
            });
}
```

Implement the DEL request

Implement the GET request

L09PB01 Digitalizing a Michelin Star Restaurant Business [Containers]

The last part the Dockerfile is to be modified to package the server application

L09PB01 Digitalizing a Michelin Star Restaurant Business [Containers]

```
# File: Dockerfile

FROM openjdk:17-bullseye

WORKDIR /app

# TODO: Copy the compiled jar
COPY build/libs/L09PB01-1.0.0-plain.jar app.jar

# TODO: Copy the start.sh script
COPY start.sh start.sh

# TODO: Make start.sh executable
RUN chmod 770 start.sh

# TODO: Set the start command
CMD ./start.sh
```

L09PB02 Digitalizing a Michelin Star Restaurant Business (Part 2) [REST, Containers]

Goal

- Implement a Client-Server Architecture with Spring Boot using REST APIs and Docker containers.

Objectives

- REST API Development (Server-side): Implement robust endpoints for adding, modifying, and deleting order data.
- Client-side Integration: Develop a GUI-driven client to interact with the API
- Docker Deployment: prepare the application for deployment by creating a Dockerfile and a start script.

L09PB02 Digitalizing a Michelin Star Restaurant Business (Part 2) [REST, Containers]

```java
@PostMapping("orders")
public ResponseEntity<Order> createOrder(@RequestBody Order.OrderRequest orderRequest) {
   List<Customer> customers = customerService.getAllPersons(null);
   var optionalCustomer = customers.stream().filter(customer ->
customer.getId().equals(orderRequest.getCustomerId())).findFirst();
   if (optionalCustomer.isEmpty()) {
       return ResponseEntity.badRequest().build();
   }
   Order.MenuItem item = switch (orderRequest.getItem()) {
       case "Pizza" -> MenuItem.Pizza;
       case "Spaghetti" -> MenuItem.Spaghetti;
       case "Hamburger" -> MenuItem.Hamburger;
       default -> null;
   };
   if (item == null) {
       return ResponseEntity.badRequest().build();
   }
   Order order = new Order();
   order.setId(UUID.randomUUID());
   order.setCustomerId(orderRequest.getCustomerId());
   order.setItem(item);
   order.setOrderedOn(LocalDate.now());
   return ResponseEntity.ok(orderService.saveOrder(order));
}
```

Add the responses for POST

L09PB02 Digitalizing a Michelin Star Restaurant Business (Part 2) [REST, Containers]

```
@PutMapping("orders/{orderId}")
public ResponseEntity<Order> updateOrder(@RequestBody Order.OrderRequest orderRequest, @PathVariable("orderId") UUID
orderId) {
    List<Order> orders = orderService.getAllOrders(null, null);
    var optionalOrder = orders.stream().filter(order -> order.getId().equals(orderId)).findFirst();
    if (optionalOrder.isEmpty()) {
        return ResponseEntity.badRequest().build();
    }
    List<Customer> customers = customerService.getAllPersons(null);
    var optionalCustomer = customers.stream().filter(customer ->
customer.getId().equals(orderRequest.getCustomerId())).findFirst();
    if (optionalCustomer.isEmpty()) {
        return ResponseEntity.badRequest().build();
    }
    Order.MenuItem item = switch (orderRequest.getItem()) {
        case "Pizza" -> MenuItem.Pizza;
        case "Spaghetti" -> MenuItem.Spaghetti;
        case "Hamburger" -> MenuItem.Hamburger;
        default -> null;
    };
    if (item == null) {
        return ResponseEntity.badRequest().build();
    }
    Order order = optionalOrder.get();
    order.setCustomerId(orderRequest.getCustomerId());
    order.setItem(item);
    return ResponseEntity.ok(order);
}
```

Add the responses for PUT

L09PB02 Digitalizing a Michelin Star Restaurant Business (Part 2) [REST, Containers]

```
@DeleteMapping("orders/{orderId}")
public ResponseEntity<Void> deleteOrder(@PathVariable("orderId") UUID orderId) {
    orderService.deleteOrder(orderId);
    return ResponseEntity.noContent().build();
}
```

Add the responses for DELETE

```
@GetMapping("customers")
public ResponseEntity<List<Customer>> getAllCustomers(
@RequestParam(value = "sortField", defaultValue = "ID") CustomerSortingOptions.SortField sortField,
@RequestParam(value = "sortingOrder", defaultValue = "ASCENDING") CustomerSortingOptions.SortingOrder sortingOrder
) {

    return ResponseEntity.ok(customerService.getAllPersons(new CustomerSortingOptions(sortingOrder, sortField)));
}

@GetMapping("orders")
public ResponseEntity<List<Order>> getAllOrders(
@RequestParam(value = "from", defaultValue = "") Long from,
@RequestParam(value = "to", defaultValue = "") Long to
) {
     return ResponseEntity.ok(orderService.getAllOrders(from, to));
}
```

Add the responses for GET

L09PB02 Digitalizing a Michelin Star Restaurant Business (Part 2) [REST, Containers]

```java
public void addOrder(Order.OrderRequest orderRequest, Consumer<List<Order>> ordersConsumer) {
    // TODO Part 2: Make an http post request to the server
    webClient.post()
        .uri("orders")
        .bodyValue(orderRequest)
        .retrieve()
        .bodyToMono(Order.class)
        .onErrorStop()
        .subscribe(newOrder -> {
            orders.add(newOrder);
            ordersConsumer.accept(orders);
        });
}
```

Implement the POST request

```java
public void updateOrder(UUID orderId, Order.OrderRequest orderRequest, Consumer<List<Order>> ordersConsumer) {
    // TODO Part 2: Make an http put request to the server
    webClient.put()
        .uri("orders/" + orderId)
        .bodyValue(orderRequest)
        .retrieve()
        .bodyToMono(Order.class)
        .onErrorStop()
        .subscribe(newOrder -> {
            orders.replaceAll(oldOrder -> oldOrder.getId().equals(newOrder.getId()) ? newOrder : oldOrder);
            ordersConsumer.accept(orders);
        });
}
```

Implement the PUT request

L09PB02 Digitalizing a Michelin Star Restaurant Business (Part 2) [REST, Containers]

```java
public void deleteOrder(UUID orderId, Consumer<List<Order>> ordersConsumer) {
    // TODO Part 2: Make an http delete request to the server
    webClient.delete()
        .uri("orders/" + orderId)
        .retrieve()
        .toBodilessEntity()
        .onErrorStop()
        .subscribe(v -> {
            orders.removeIf(order -> order.getId().equals(orderId));
            ordersConsumer.accept(orders);
        });
}
public void getAllOrders(Long from, Long to, Consumer<List<Order>> ordersConsumer) {
    // TODO Part 2: Make an https get request to the server
    webClient.get()
        .uri(uriBuilder -> uriBuilder
            .path("orders")
            .queryParam("from", from)
            .queryParam("to", to)
            .build())
        .retrieve()
        .bodyToMono(new ParameterizedTypeReference<List<Order>>() {})
        .onErrorStop()
        .subscribe(newOrders -> {
            orders.clear();
            orders.addAll(newOrders);
            ordersConsumer.accept(orders);
        });
}
```

Implement the DELETE request

Implement the GET request

L09PB02 Digitalizing a Michelin Star Restaurant Business (Part 2) [REST, Containers]

```
# File: Dockerfile

FROM openjdk:17-bullseye

WORKDIR /app

# TODO: Copy the compiled jar
COPY build/libs/L09PB02-1.0.0-plain.jar app.jar

# TODO: Copy the start.sh script
COPY start.sh start.sh

# TODO: Make start.sh executable
RUN chmod 770 start.sh

# TODO: Set the start command
CMD ./start.sh
```

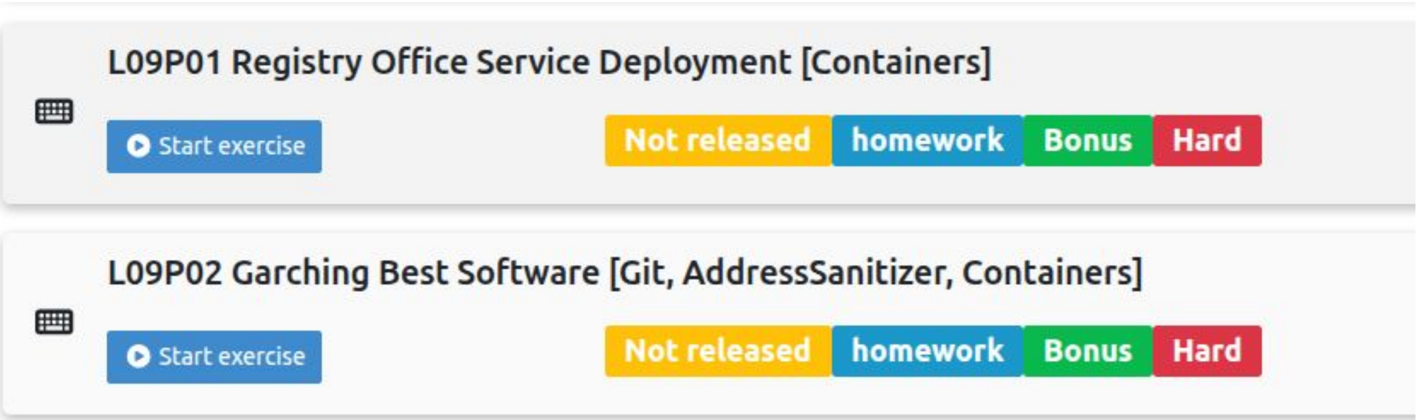
Tutorial outline

- ~~**Part I:** Lecture summary~~
  - ~~Q&A for the lecture material~~

- ~~**Part II:** Programming basics~~

- **Part III: Homework programming exercises (Artemis)**

Programming (P) exercises



L09P01 Registry Office Service Deployment [Containers]

Goal

- Provide hands-on experience in the **full lifecycle of a software application**, from development and containerization to deployment and feature enhancement, in a simulated junior software engineer role

Objectives

- Set up and configure the **ngrok** service to expose your local application to the internet
- Build and manage containers for a Spring Boot app, using a **Dockerfile** and **docker compose**
- Automate the build and deployment process and retrieve the application's public URL
- Implement a new feature within the application, focusing on business logic and persistence
- Write unit and integration tests to ensure its reliability and correctness

L09P02 Garching Best Software [Git, AddressSanitizer, Containers]

Goal

- Resolve existing issues in the codebase related to version control, memory safety, and containerization before further development

Objectives

- **Git:** Improve the repository's commit history by practicing various Git operations
- **Dynamic Analysis:** Identify and fix memory safety bugs in the C application
- **Containerization:** Improve the portability and deployment of the application using Docker and Docker Compose