L09 Software Deployment and Monitoring

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

Today’s learning goals

- **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

Outline

- **Part I: Deployment models in the cloud**
  - Baremetal, virtual machines, containers, and serverless
- **Part II:** Hello world in the cloud
- **Part III:** Orchestrating in the cloud
- **Part IV:** System monitoring

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	Application image

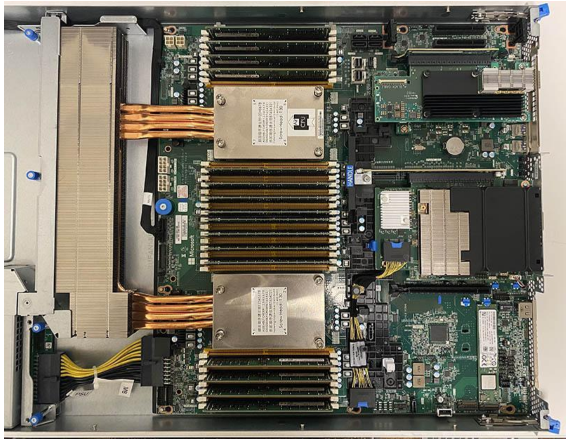
Higher flexibility

Less responsibilities

A. Baremetal

Baremetal: Introduction

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



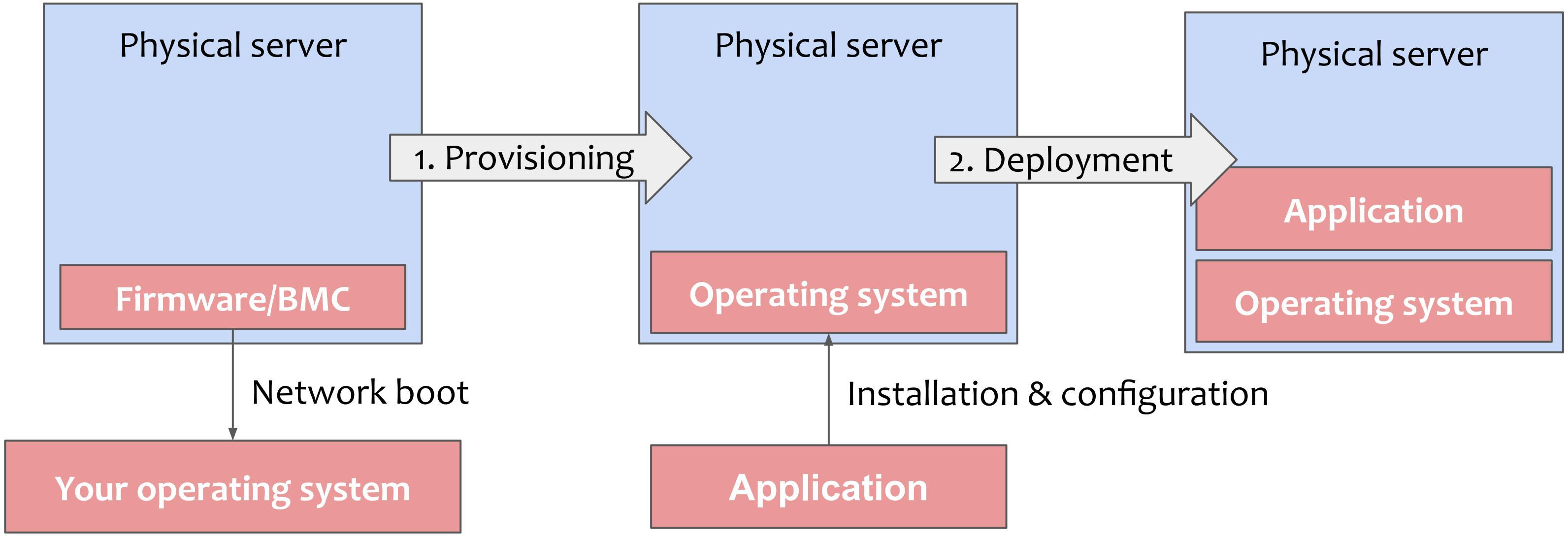
Server used by Microsoft Azure
Source: https://specbranch.com/posts/one-big-server/

Azure cloud server rack

Physical hardware are the foundation for all other deployments!

Baremetal: Workflow

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



Advantages of Baremetal deployments

- Full control over hardware configuration

- No overhead from virtualization layer -> Higher performance

- No “nosy neighbors”

- Better security due to isolation from other tenants

Challenges of Baremetal deployments

- More complex setup and maintenance compared to virtualization
- Limited scalability due to physical hardware constraints
- Difficult to implement disaster recovery solutions
- Requires more physical space and power compared to virtualization

Use-cases for Baremetal deployments

- High-performance computing (HPC)
- Data-intensive workloads (e.g., large databases)
- Latency-sensitive applications (e.g., gaming, financial trading)
- Legacy applications that require specific hardware configurations

Examples of Baremetal deployments

- Large cloud providers AWS and Azure offer baremetal instances
- Many HPC clusters use baremetal deployments
- Organizations with specific compliance requirements (e.g. healthcare, finance) may prefer baremetal deployments for security reasons

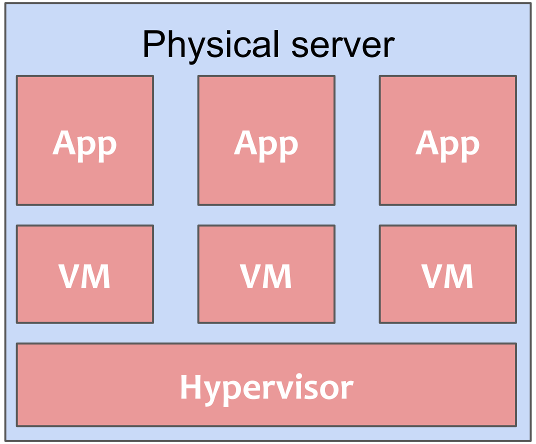
Best practices for Baremetal deployments

- Plan carefully to ensure hardware resources are utilized efficiently
- Use automation tools to simplify deployment and maintenance
- Implement monitoring and alerting to detect software and hardware issues
- Regularly test disaster recovery solutions

B. Virtual machines

Virtual machines: Introduction

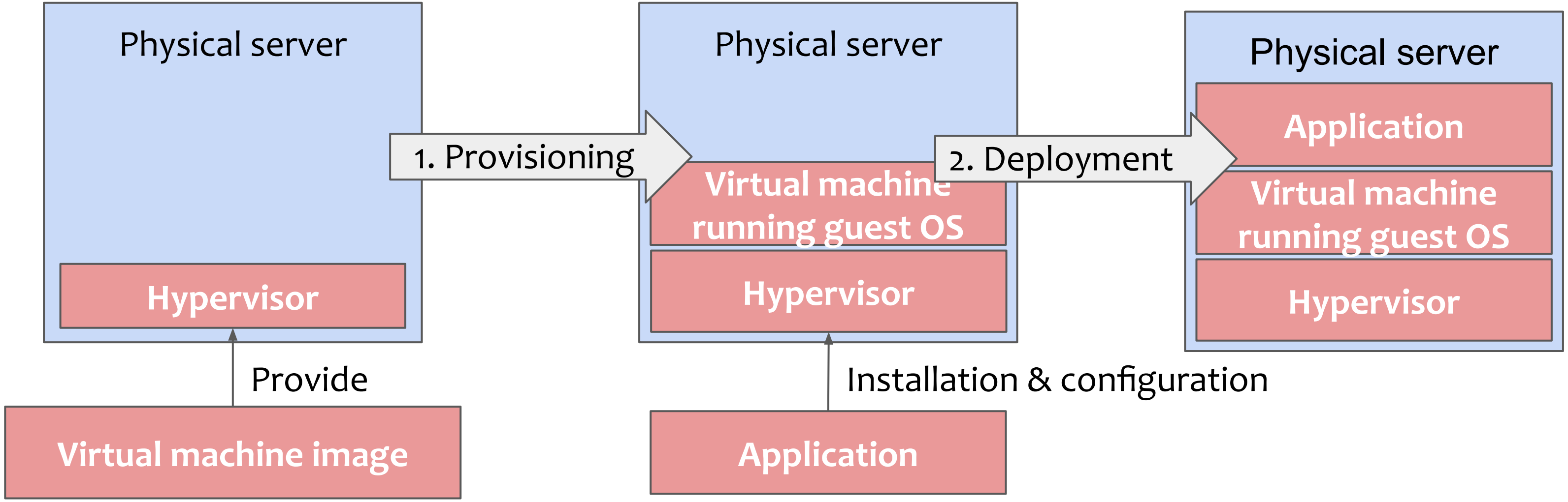
- Physical server gets shared between multiple virtual machines
- Each virtual machine runs its own operating system
- Resources are shared between different tenants



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

Virtual machines: Workflow

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



Advantages of Virtual Machine

- Ability to run multiple applications on a single physical server

- Easier disaster recovery and backup solutions

- Scalability through the use of cloud-based virtualization

Challenges of Virtual Machine

- Performance overhead from virtualization layer
- Resource contention between multiple VMs on a single physical server
- Complexity of configuring virtual networks and storage
- Security risks from sharing the same hardware

Use Cases for Virtual Machine

- Running legacy applications on modern hardware
- DevOps environments with consistent development and test environments
- Cloud-based hosting of scalable web applications
- Multi-tenant environments for software as a service (SaaS) providers

Examples of Virtual Machine

- Major cloud providers like AWS and Azure offer virtual machine instances
- Many organizations use virtual machines for testing and development environments

VAGRANT

- Many SaaS providers use virtual machines for their multi-tenant platforms

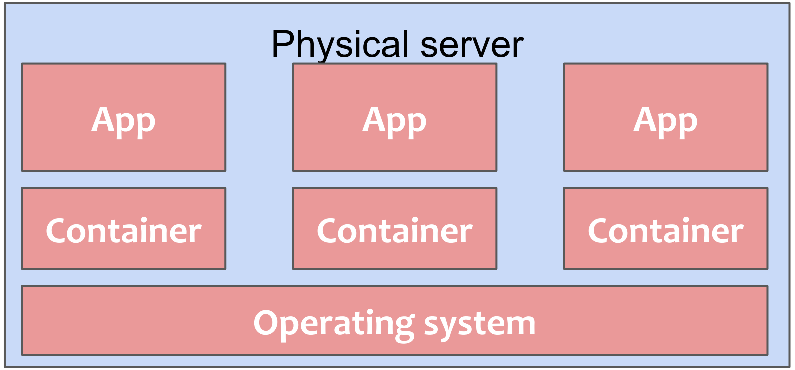
Best practices for Virtual Machine

- Optimize virtual machine configurations for maximum performance and resource utilization
- Automate provisioning and configuration management to minimize manual effort
- Monitor performance and capacity regularly to ensure availability and responsiveness
- Implement backup and disaster recovery solutions to protect against data loss

C. Containers

Containers: Introduction

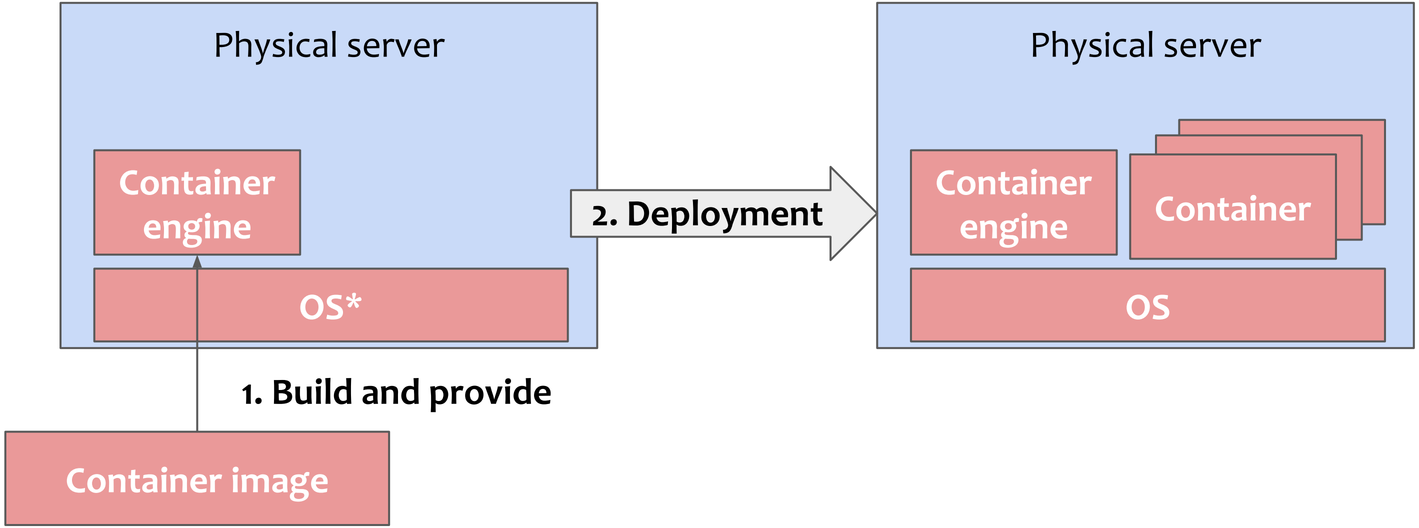
- Linux containers are a lightweight means of virtualizing an operating system and applications
- They are designed to run on a host operating system and share the host's resources
- Linux containers provide isolation between applications and their dependencies, making it easier to manage software deployments



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

Containers: Workflow

1. Build and provide container image
2. Container engine starts and manage containers



*in practice the container host often in virtual machines

Advantages of Container

- Lightweight and portable container images
- Consistent application behavior across different environments
- High resource utilization due to shared host OS
- Rapid deployment and scaling capabilities

Challenges of Container

- Complexity of container orchestration and networking
- Security risks from shared kernel and potential container escape attacks
- Limited access to host resources and system configurations
- Compatibility issues between different container platforms

Use-cases for Container

- Microservices architectures for web applications
- Testing and development environments with consistent configurations
- Deployments of complex distributed systems
- Hybrid cloud deployments with a mix of on-premises and cloud environments

Examples of Container

- Container platforms like Docker and Kubernetes are widely used for container deployments
- Many cloud providers offer container services for scalable deployments
- Many organizations use containers for development and testing environments

Best practices for Container

- Use lightweight base images and minimize container size for optimal performance
- Implement container orchestration and service discovery tools for easier management
- Monitor container performance and capacity regularly to ensure scalability and availability
- Implement security measures such as container isolation and network segmentation

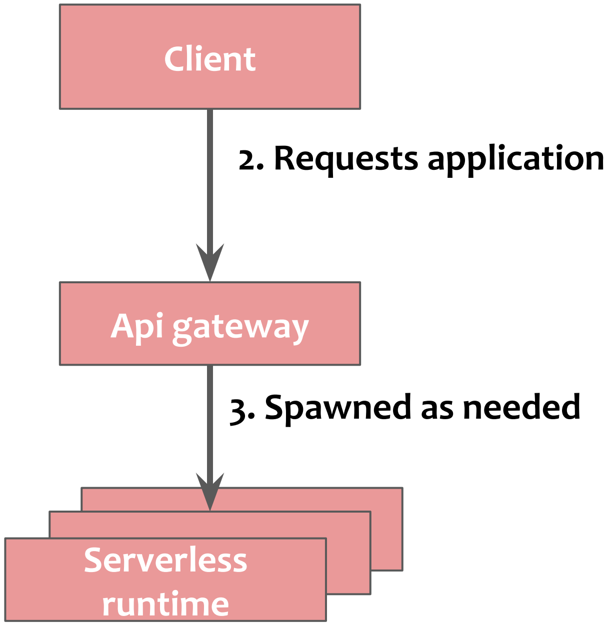
D. Serverless

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

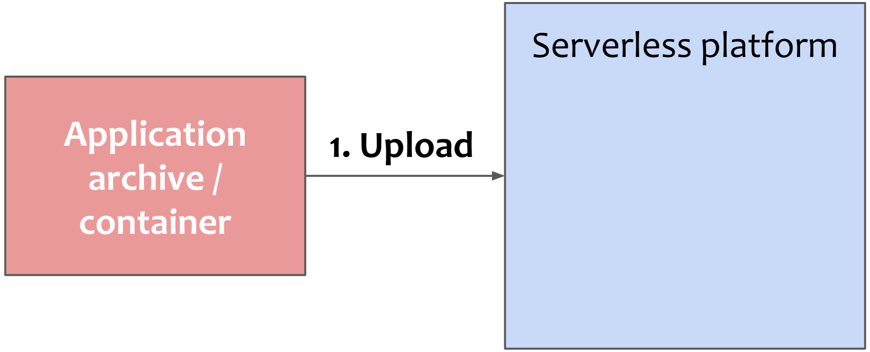
- Serverless is a pay-per-use model where developers only pay for the exact amount of resources their application requires



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

Serverless: Workflow

1. Build and provide application image
2. External event i.e. an http requests triggers the application
3. Serverless platform scales up serverless runtimes running the application



Advantages of Serverless

- No need to manage infrastructure or servers
- Auto-scaling capabilities for optimal resource utilization
- Lower operational costs due to pay-as-you-go pricing model especially for sporadic workloads

Challenges of Serverless

- Limited control over underlying infrastructure
- Cold start issues and variable performance
- Increased complexity of application architecture and development
- Compatibility issues with third-party services and libraries

Use-cases for Serverless

- Event-driven applications with sporadic or unpredictable workloads
- Microservices architectures for web applications
- Serverless functions as part of a larger application stack
- Hybrid cloud deployments with a mix of on-premises and cloud environments

Examples of Serverless

- Major cloud providers like AWS and Azure offer serverless services such as AWS Lambda and Azure Functions
- Many organizations use serverless functions as part of their web applications
- Many IoT applications use serverless functions for data processing and event handling

Best practices for Serverless

- Optimize serverless functions for performance and resource utilization
- Use monitoring and alerting tools to detect performance issues and potential errors
- Implement security measures such as access control and data encryption
- Minimize external dependencies and avoid vendor lock-in

Outline

- ~~**Part I:** Deployment models in the cloud~~
- **Part II:** **Hello world in the cloud**
  - Development and deployment of a simple application in the cloud
- **Part III:** Orchestrating in the cloud
- **Part IV:** System monitoring

Hello world in the cloud: The code

- Let’s package and deploy a simple python app!
- Create a directory with the following files:

Source code

app.py

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
   return 'Hello, World!'

if __name__ == '__main__':
   app.run(host='0.0.0.0')
```

Dependencies

requirements.txt

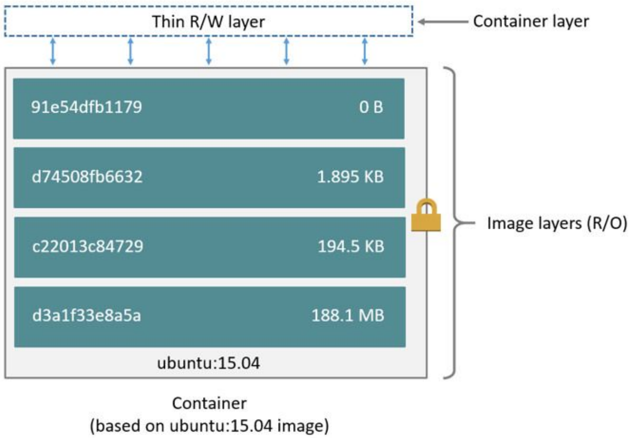
```
flask
```

Hello world in the cloud: Docker packaging

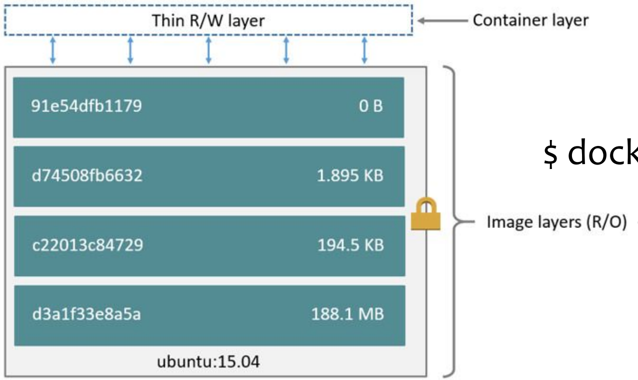
- Docker is a platform for building, shipping, and running applications in containers
- Docker packages apps and dependencies into a standardized unit for easy development
- Docker simplifies moving apps between environments and ensures consistency
- Docker reduces conflicts between dependencies

```
# Dockerfile
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

$ docker build

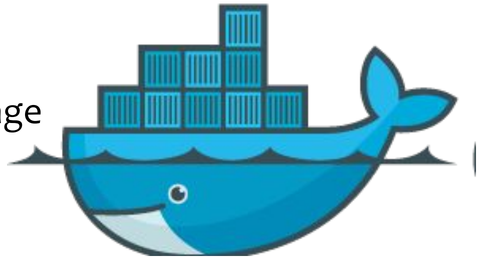


Hello world in the cloud: Docker registry

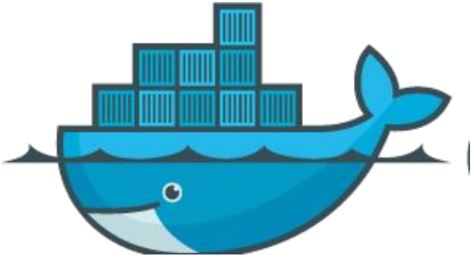


Container
(based on ubuntu:15.04 image)

$ docker push myregistry.com/myorg/myimage

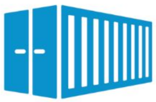


Docker registry



Docker registry

$ docker run myregistry.com/myorg/myimage



Container-instance running an application process

Hello world in the cloud: Dockerfile

- This is what a Dockerfile looks like for our toy application:

```
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"]
```

- The base image that are container starts with.
- This one is using alpine linux with python pre-installed

Best practice: Don't let container apps run as the root user!

Hello world in the cloud: docker build

- Open a terminal window, navigate to the project directory, and run the following command to build the Docker image:

```
Step 7/9 : COPY . .
 ---> 6ed3d5f0d947
Step 8/9 : EXPOSE 5000
 ---> Running in 5d0e03cb3704
Removing intermediate container 5d0e03cb3704
 ---> 02dd61143abe
Step 9/9 : CMD ["python", "app.py"]
 ---> Running in 6d83a8d41c0a
Removing intermediate container 6d83a8d41c0a
 ---> 2a59a5c98a72
Successfully built 2a59a5c98a72
Successfully tagged hello-world:latest
```

Hello world in the cloud: docker run

- Once the build is complete, run the following command to start a container using the newly created image:

```
$ docker run -p 5000:5000 hello-world
 * Serving Flask app 'app'
…
 * Running on http://localhost:5000
Press CTRL+C to quit
```

- Finally, open a web browser and navigate to http://localhost:5000
- You should see a "Hello, World!" message displayed in the browser, indicating that the container is running and serving the application

Application works; Let's ship it!

Hello world in the cloud: Deployment example

- In this example we are going to use fly.io (serverless platform provider)

fly.toml (our deployment configuration)

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

```
$ flyctl apps create --name tum-greets-the-world
$ flyctl deploy
....
 1 desired, 1 placed, 1 healthy, 0 unhealthy
--> v0 deployed successfully
```

Container deployed; Mission completed

Outline

- ~~**Part I:** Deployment models in the cloud~~

- ~~**Part II:** Hello world in the cloud~~

- **Part III: Orchestrating in the cloud**
  - Deployment and orchestrating a microservice in the cloud

- **Part IV:** System monitoring

Deployment models in the cloud

Already covered in lecture 02, just a quick recap:

- Baremetal
- Virtual machines
- Containers
- Serverless

Deployment models for microservices

Baremetal

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

Deployment models for microservices

Baremetal

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

Virtual machines

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

Deployment models for microservices

Baremetal

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

Containers

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

Virtual machines

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

Deployment models for microservices

Baremetal

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

Containers

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

Virtual machines

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

Function-as-a-Service (FaaS)

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

Deploying microservices

Until now, we used a logical view of microservices, agnostic of physical machines

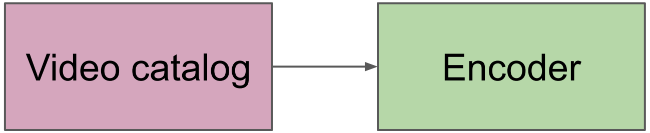
Deploying on physical machines can be done in various ways, with pros and cons

We will now discuss:

- How to manage multiple instances of the same service?
- How to deploy databases?
- What are the different deployment models?
- How to orchestrate the deployment and scaling of microservices?

Managing multiple instances of microservices

With one instance per service, we can map instances together easily

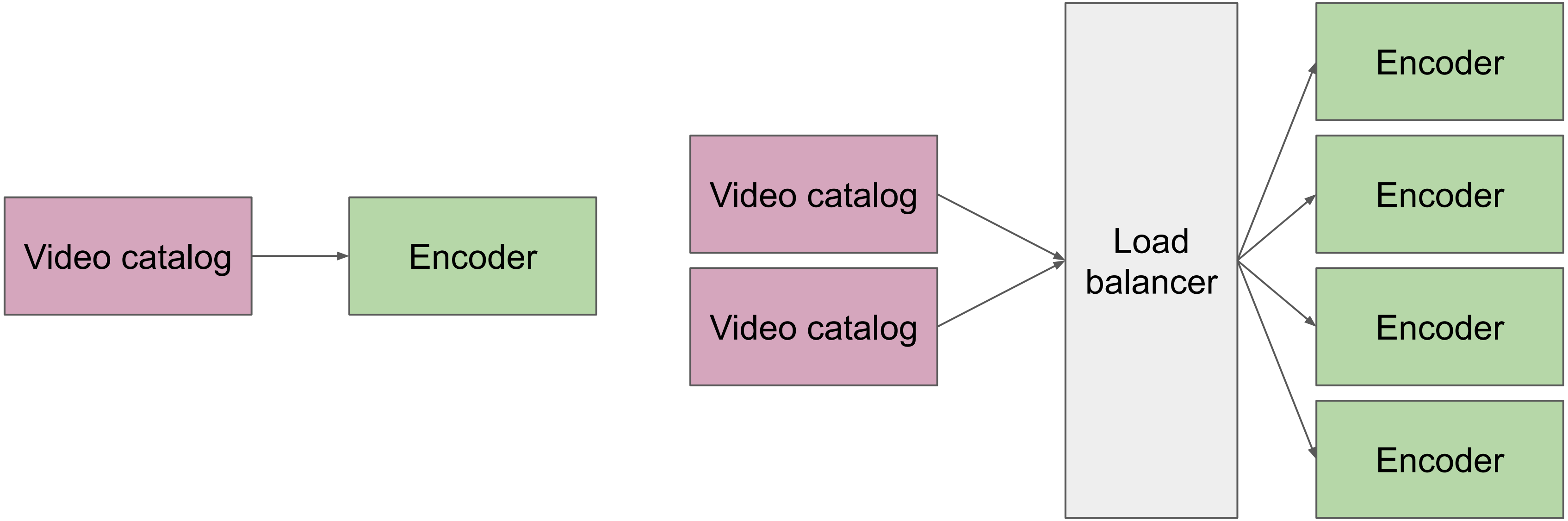


Managing multiple instances of microservices

With one instance per service, we can map instances together easily

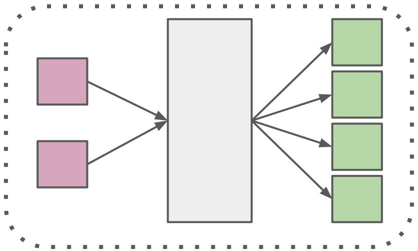
If we have multiple instances of the same service, we need to route requests

We can use a **load balancer**

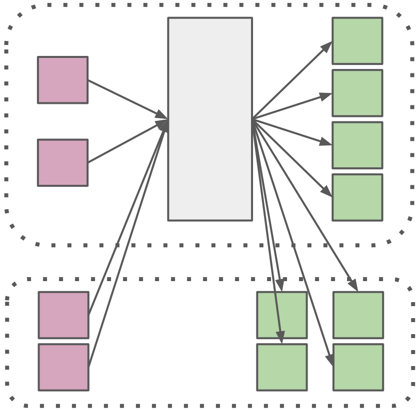


Managing multiple instances of microservices (2)

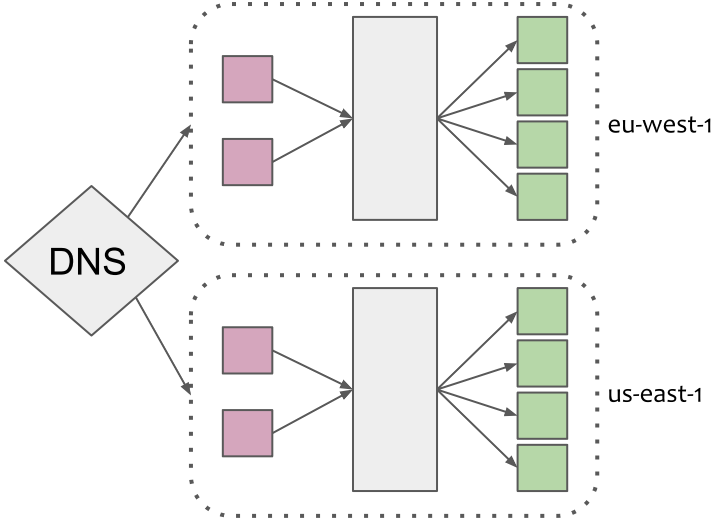
This topology can be mapped on physical hardware on a single or multiple machines



Single machine



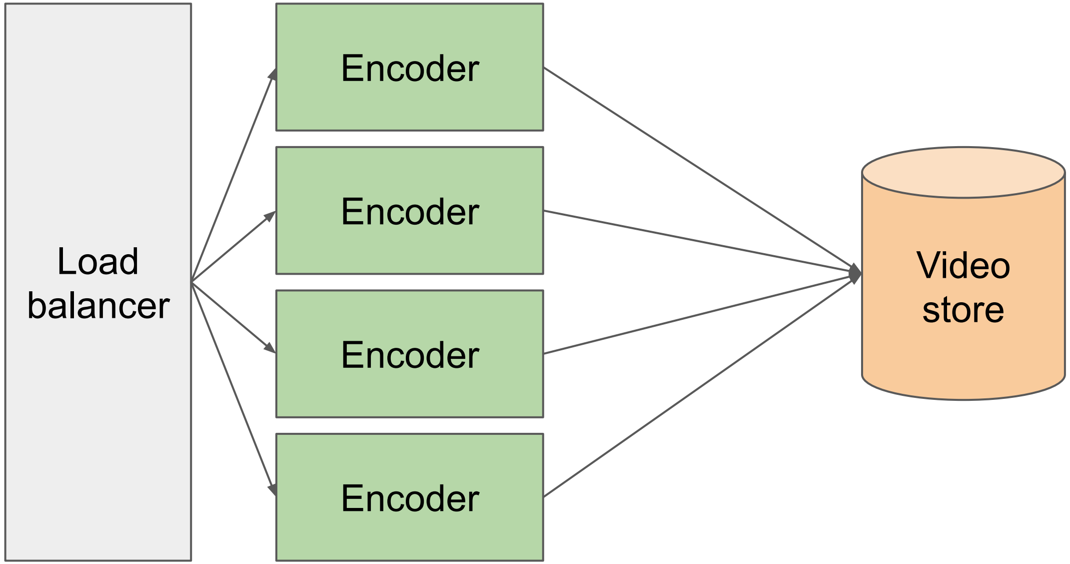
Multiple machines



DNS-based regional balancing

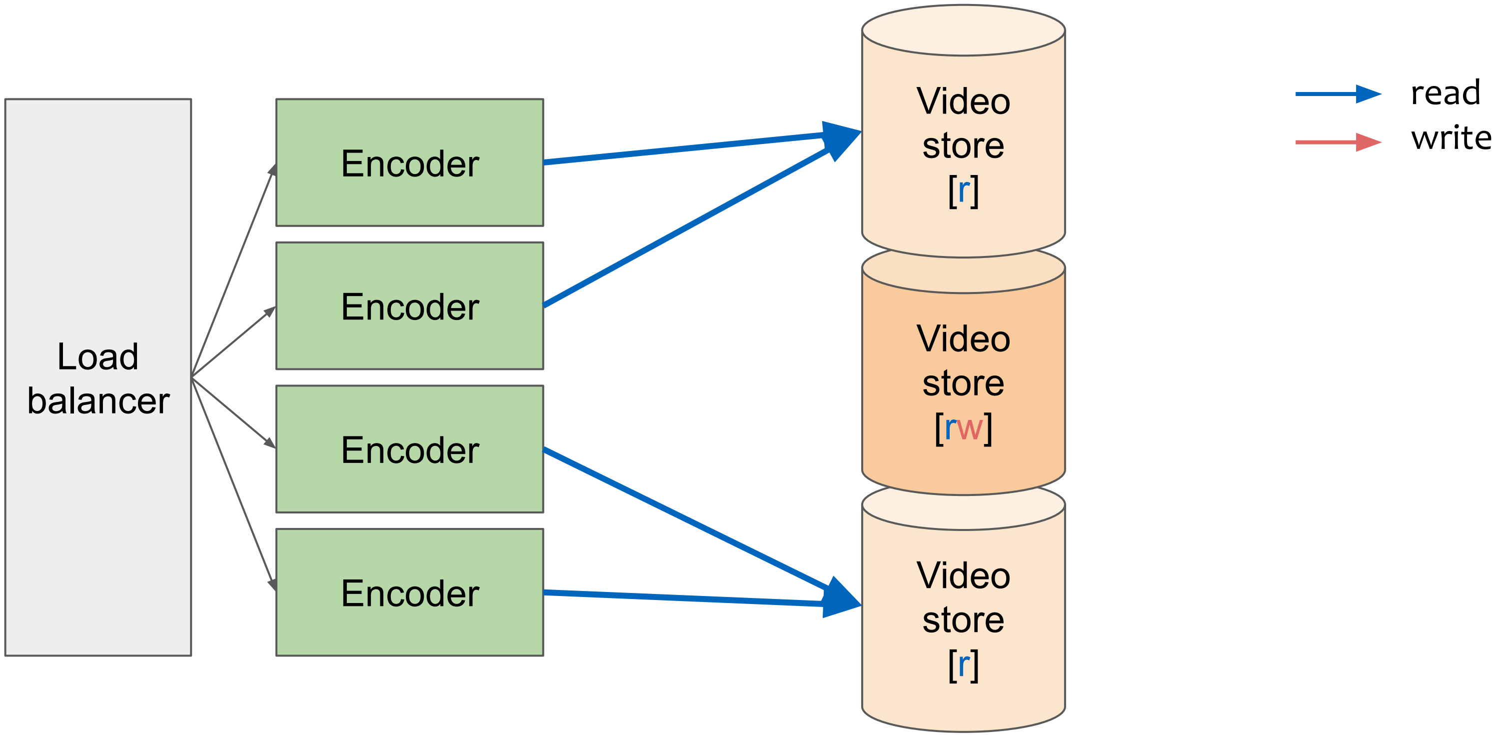
Database deployment

We may have similar contention issues with databases



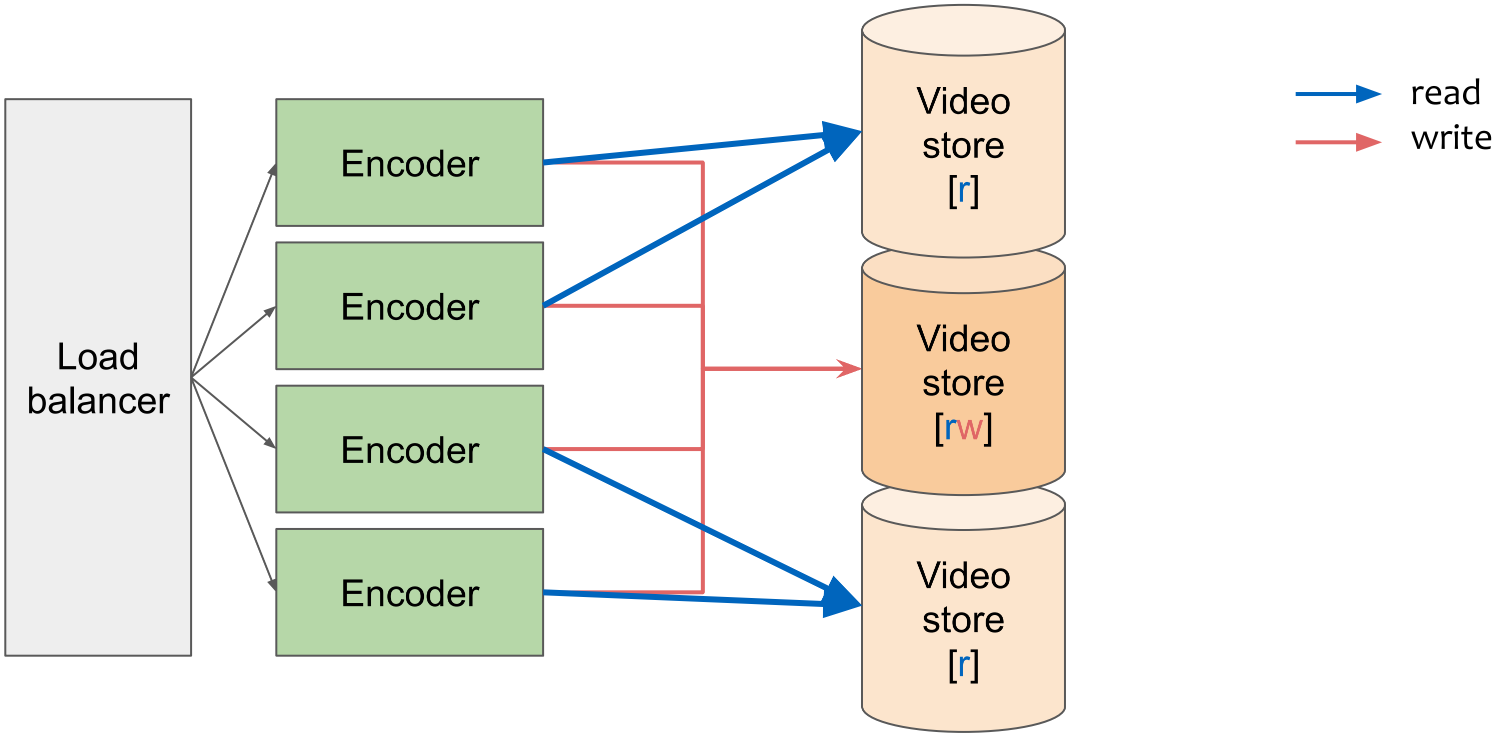
Database deployment (2)

Example optimization: If we have read-heavy workloads, we can create read replicas



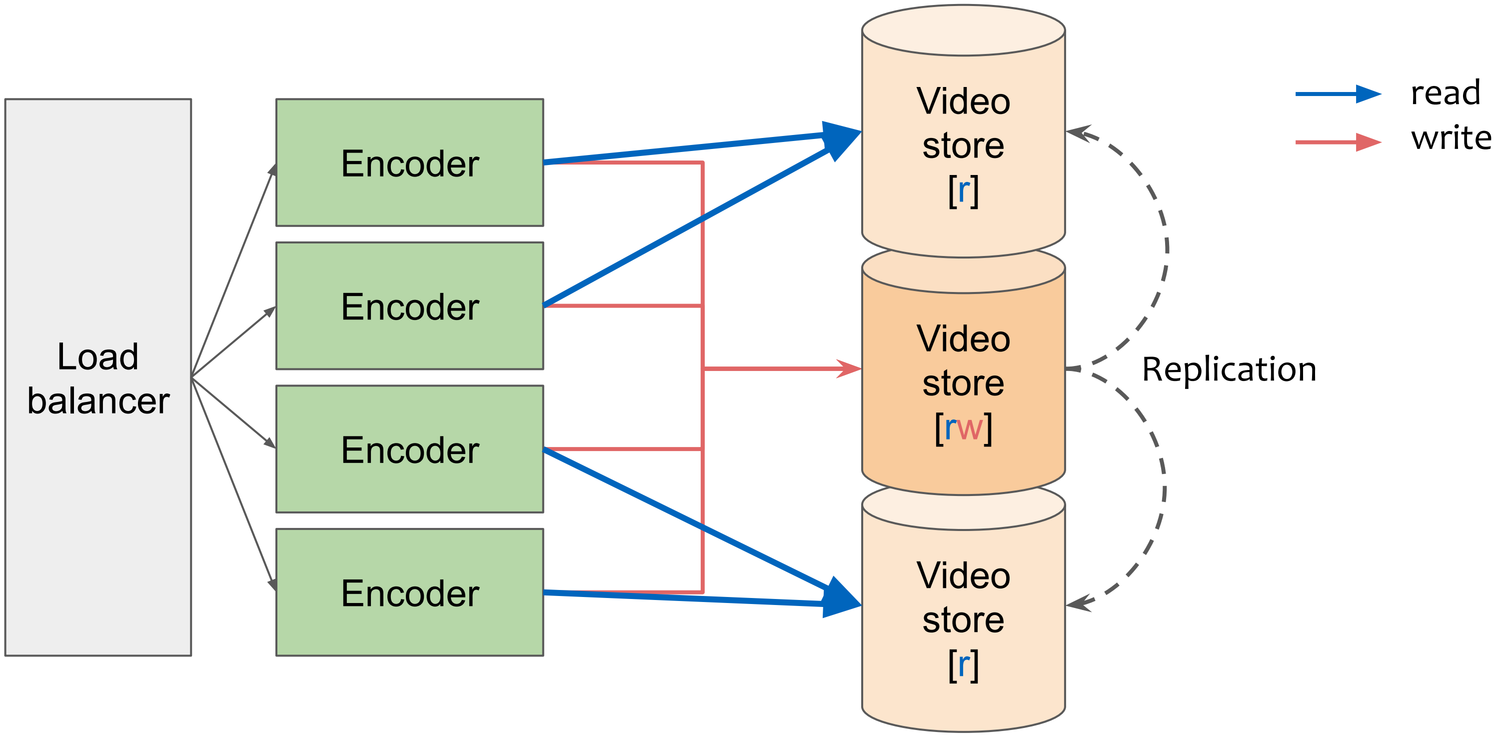
Database deployment (2)

Example optimization: If we have read-heavy workloads, we can create read replicas



Database deployment (2)

Example optimization: If we have read-heavy workloads, we can create read replicas



Deploying microservices with containers

Building the container environment

- List the dependencies (libraries, software)
- Create the container image, e.g., with a Dockerfile

```
FROM ubuntu

RUN apt-get update
RUN apt-get install –y nginx openssl
```

Orchestrate the deployment of the containers

- Where are the container instances deployed? (public cloud, private infrastructure)
- How are they scaled up/down?
- How are they scheduled?

  → Kubernetes, Docker Swarm

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

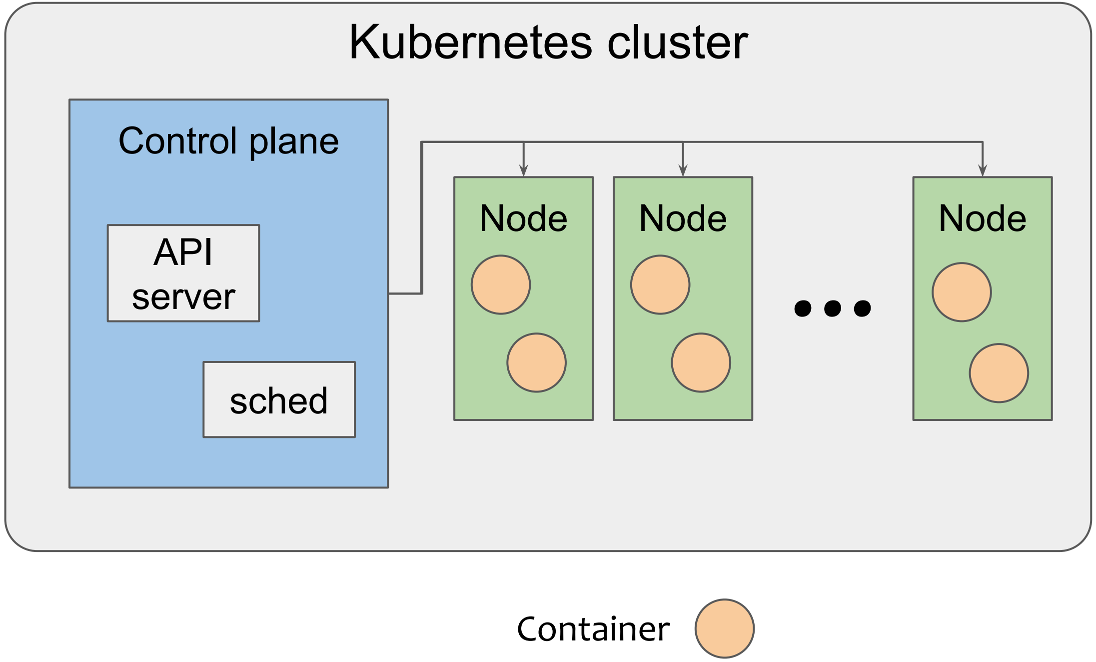
kubernetes

Kubernetes architecture

Control plane

- Manages containers
- Exposes the control API
- Schedules containers

A **node** is a worker machine that runs the containers

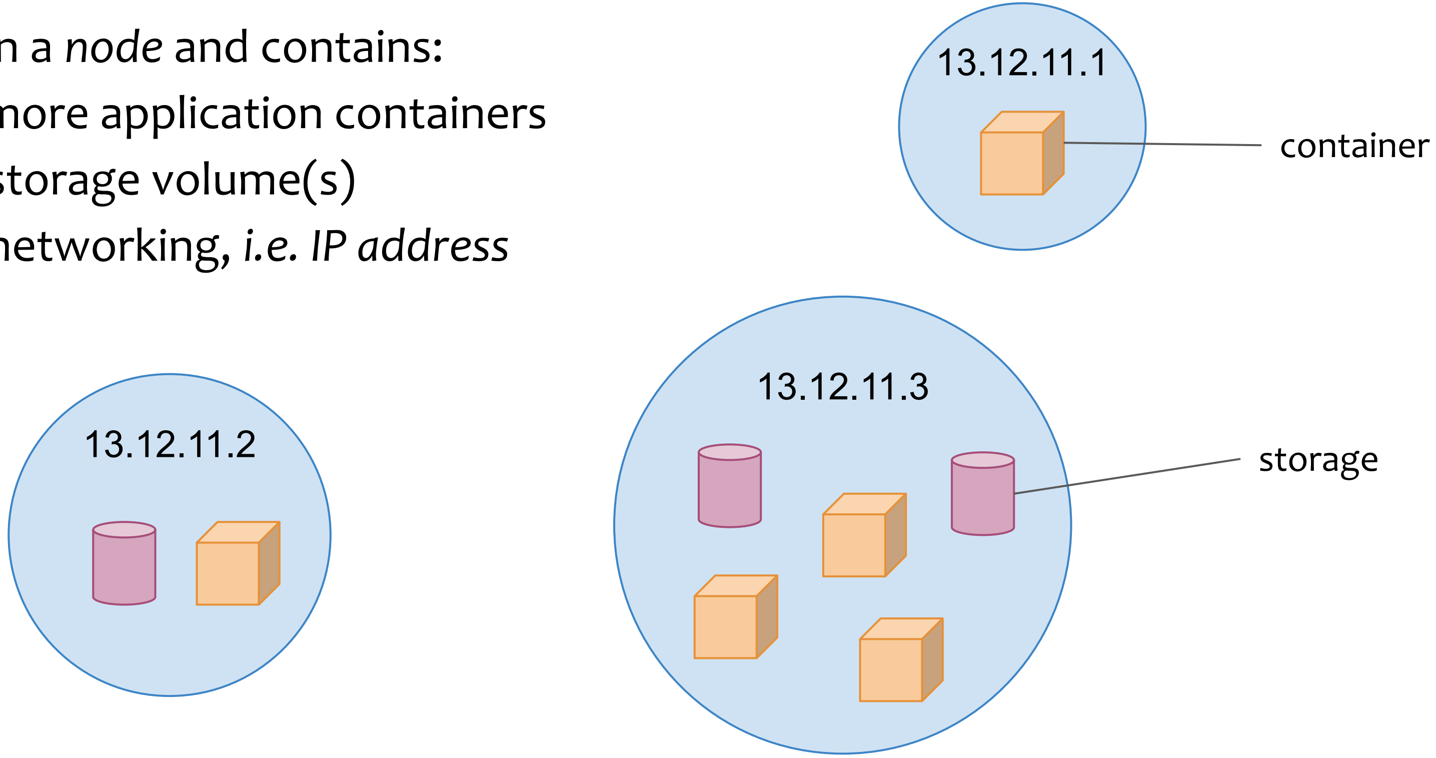


Kubernetes pods

Kubernetes’ atomic unit of deployment are called **pods**

A pod runs on a *node* and contains:

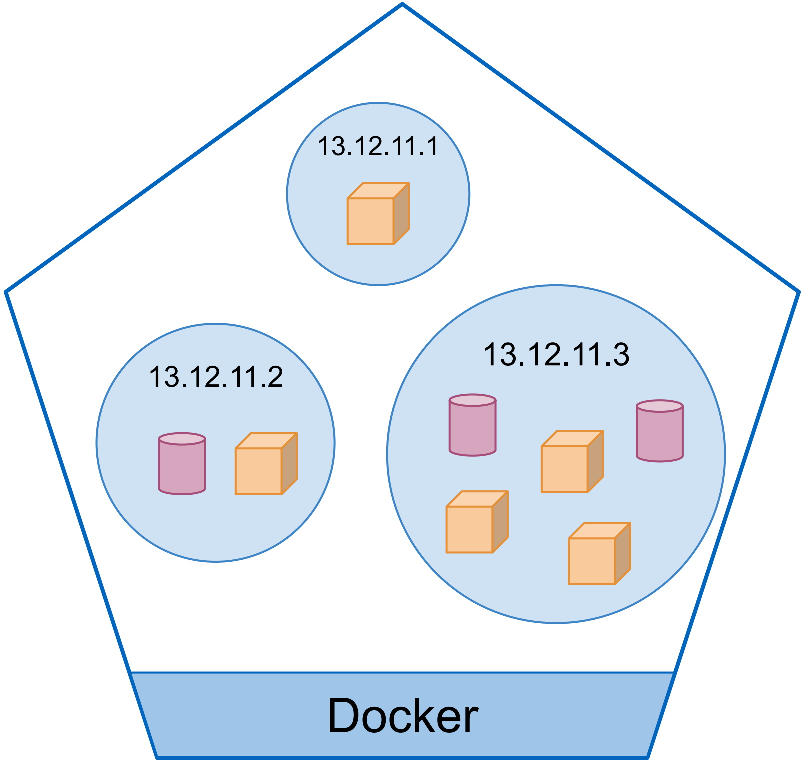
- One or more application containers
- Shared storage volume(s)
- Shared networking, *i.e. IP address*



Kubernetes nodes

A **node** is a physical worker machine that hosts *pods*:

- Runs the container runtime responsible for pulling container images and running containers
  e.g. Docker
- Handles communication with the control plane
- Manages pods and containers running on it



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?***

Outline

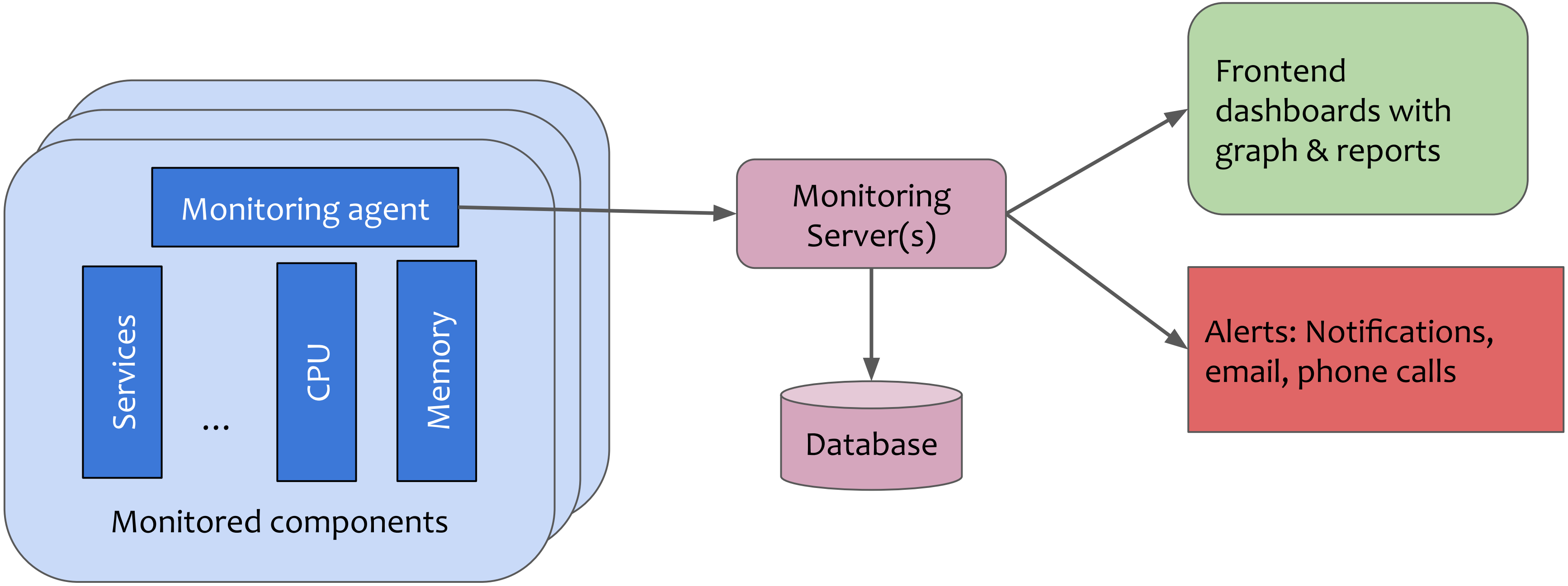
- ~~**Part I:** Deployment models in the cloud~~
- ~~**Part II:** Hello world in the cloud~~
- ~~**Part III:** Orchestrating in the cloud~~
- **Part IV: System monitoring**
    - Background about monitoring and its importance
    - Metrics, alerting, logging, tracing

Importance of system monitoring

1. Proactive problem solving
2. Performance optimization
3. Fault and issue detection
4. Insight into system behavior
5. Improved security
6. Cost management
7. Business insights

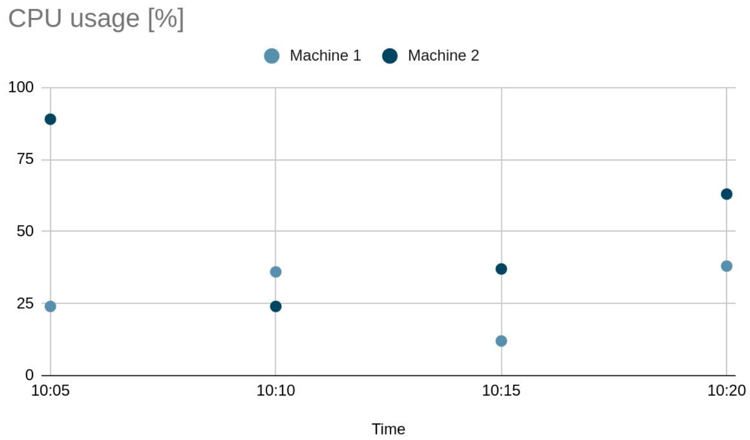
No matter if you are an operator, mobile app developer or product manager:
Monitoring is important to get insights from deployed software.

Common monitoring architecture



Metrics - What is happening?

What are Metrics?



Each metric point consists of

- Timestamp
- Value
- Metadata (Name, Origin)

- Use Metrics when you need a high-level overview of your system
- Metrics provide quantifiable data about system performance and usage
- Ideal for identifying trends and patterns, detecting anomalies, or determining whether your system is behaving normally

Type of metrics

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

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

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

Metrics monitoring with Prometheus

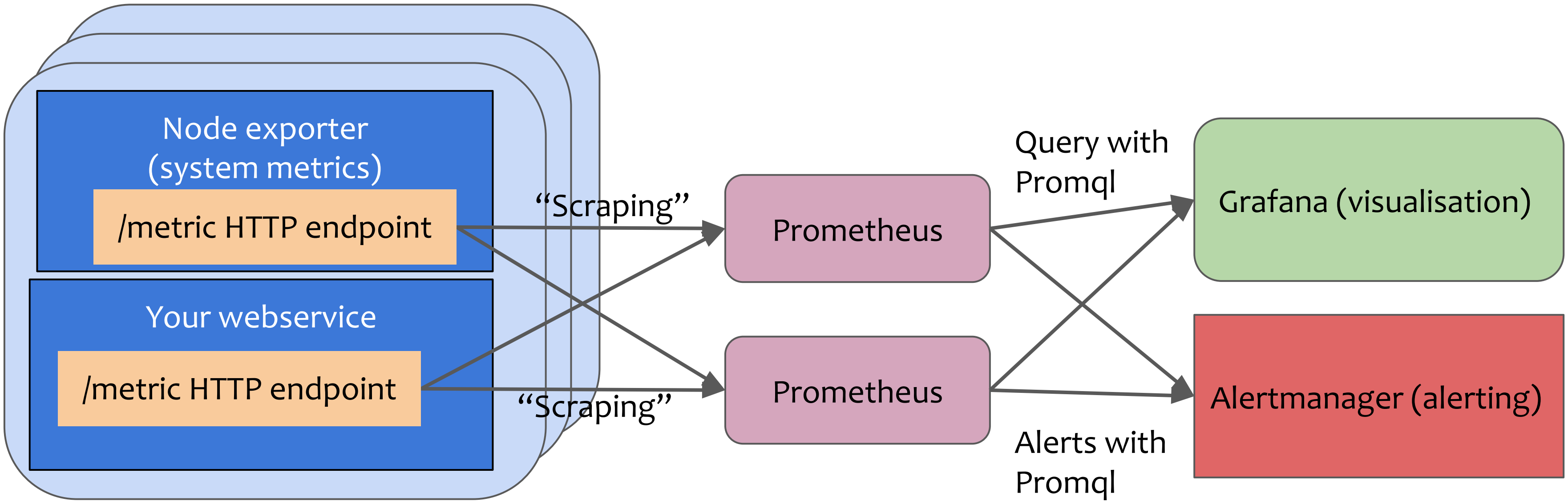
Prometheus is an open-source systems monitoring and alerting toolkit

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

Key features

1. **Multidimensional data model:** Stores data as time series, identified by a metric name and key-value pairs known as labels, which offer powerful, flexible queries.
2. **PromQL:** This is Prometheus's native query language that lets users select and aggregate time-series data in real-time.
3. **Pull model:** Unlike traditional systems that push metrics to a centralized database, Prometheus pulls metrics from monitored systems, simplifying architecture and providing better control over what gets ingested.
4. **Visualization:** No complex visualization layer, however promql api integrates well with tools like Grafana for advanced visualizations

Prometheus architecture



Prometheus and it's APIs have been widely adopted in industry. Many alternative implementations exists (VictoriaMetrics, Cortex, Thanos, M3DB and TimescaleDB, …)

Prometheus exporters

My calendar web application

Browser →

```
var requests_per_page
login() # requests_per_page{page=”/login”} += 1
scrape_metrics()
```

← Prometheus

Text-based metrics format

```
# HELP requests Page statistics metric
# TYPE requests counter
requests{application=”mycalendar”, page=”/login”} 1
requests{application=”mycalendar”, page=”/about”} 0
```

Metric name Label Value

Prometheus metrics datatypes

Prometheus works with the following datatypes:

- **Counter:** A cumulative metric that represents a single monotonically increasing counter, which can only increase or be reset to zero on restart
  - **Example:** number user logins
- **Gauge:** A metric that represents a single numerical value that can arbitrarily go up and down
  - **Example:** CPU usage
- **Histogram:** A metric that samples observations (usually things like request durations or response sizes) and counts them in configurable buckets
  - **Example:** HTTP latency
    - http_request_duration_seconds_bucket{le="0.1"} 12
    - http_request_duration_seconds_bucket{le="0.5"} 1
    - http_request_duration_seconds_bucket{le="1.0"} 0
    - http_request_duration_seconds_bucket{le="+Inf"} 0

Metrics: Promql

- **Functional query language** - think of SQL for time series
  - Designed to support **real-time monitoring** (visualisation) and alerting
- **Basic query:** http_requests_total{method="GET"}
  - Selects the time series with the **http_requests_total** metric name where the **method** label is equal to **GET**
- **Operators and functions:** rate(http_requests_total{method="GET"}[5m])
  - The **rate()** function calculates the per-second average rate of increase of the time series in the range vector, in this case over the last **5 minutes**
  - This is a typical use to examining how metrics change over time
- **Aggregations:** sum(rate(http_requests_total{method="GET"}[5m])) by (status)
  - In this query, sum is an aggregator function that sums the values across all selected time series. The by keyword denotes that the sum should be grouped by unique status labels

Metrics: Visualization

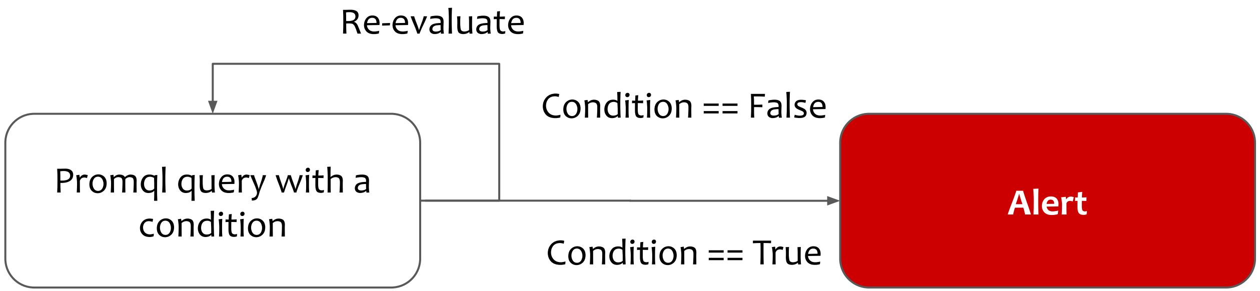
- Most popular software: **Grafana**
  - Generates graphs based on (PromQL)-Queries
  - Example showing system metrics from Node-Exporter

Metrics: Visualization

- Dashboards are always **very custom** to the applications/services
  - Different dashboards for developers/operators/security analysts
  - Allow to correlate different metrics over time (i.e. active user -> load on the system)
  - Example Freifunk Munich dashboard (https://stats.ffmuc.net)

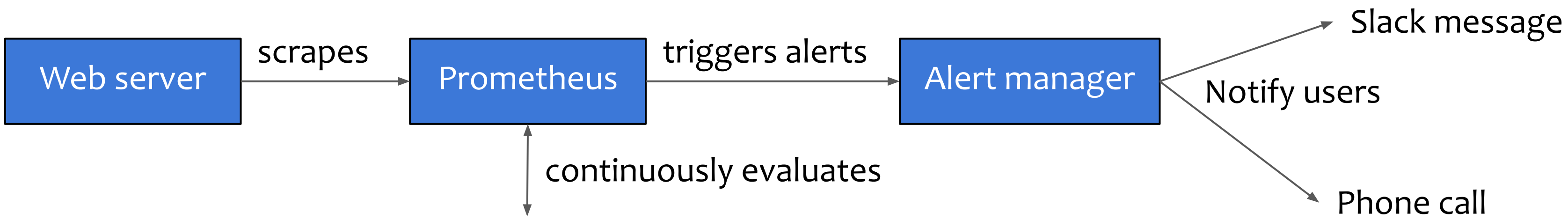
Metrics: Alerting

Basic algorithm



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: Best practices

- Identify meaningful metrics
- Monitor in real-time
- Establish alert systems
- Visualize your metrics
- Monitor across the system stack
- Update your metrics regularly

Logging - What events occurred?

Example simplified web server log:

192.0.2.100 - - [12/Jun/2023:12:23:34 -0700] "GET /index.html HTTP/1.1" 200 1234 "-" "Mozilla/5.0"
192.0.2.101 - - [12/Jun/2023:12:23:35 -0700] "GET /wp-login.php HTTP/1.1" 404 560 "-" "SomeBot/1.0"

IP | Timestamp | Accessed resource | Status code | User agent

Logs are also relevant for security

An imaginary home-automation software:

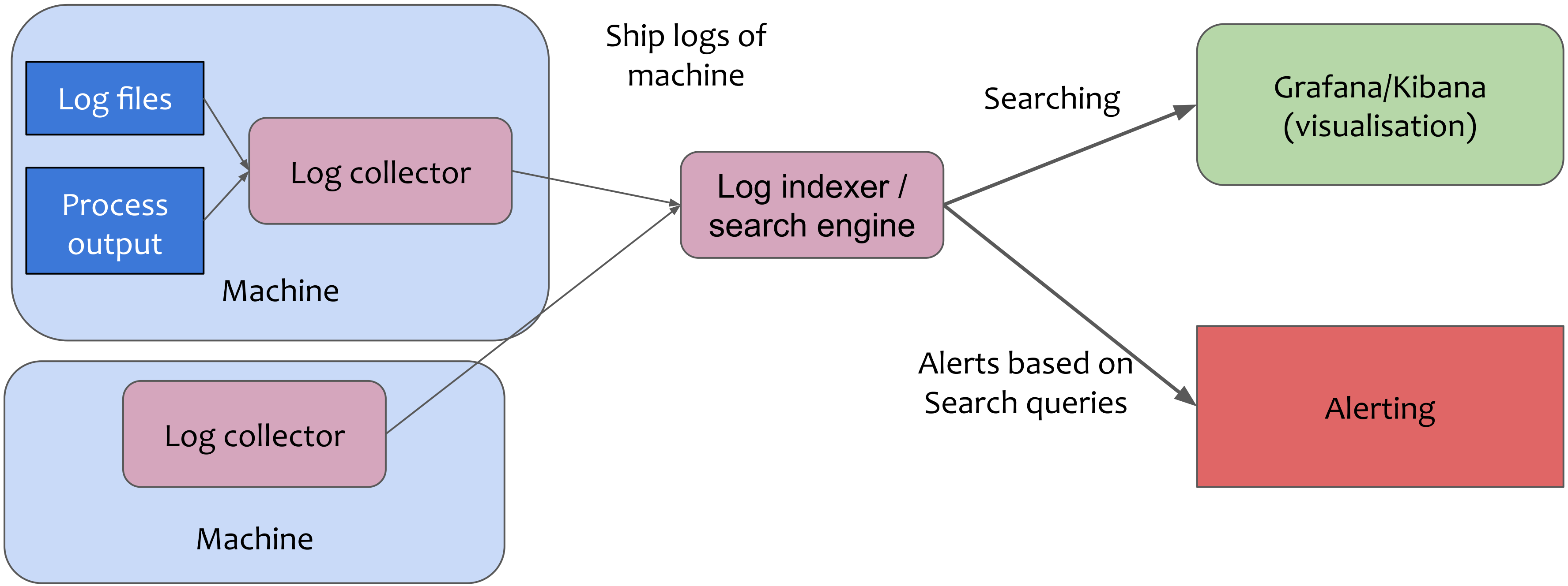
Jun 12 15:00:00 hub1 Thermostat[1001]: Temperature set to 72°F
Jun 12 15:05:22 hub1 Lights[1002]: Living room lights turned on
Jun 12 15:16:35 hub1 Lock[1003]: Front door unlocked

Logging - What events occurred?

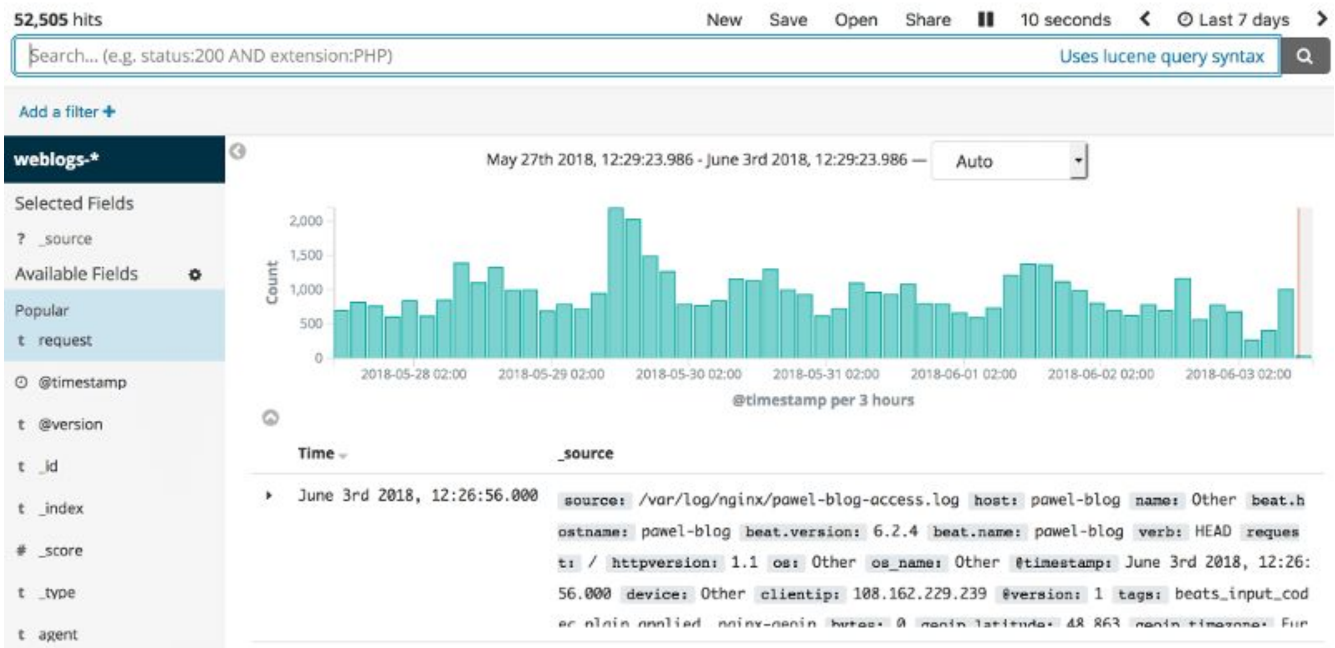
- Use logging when you need a **detailed chronological record of system events** for debugging or audit purposes
  - Logs provide information about specific events happening within your system.
- Ideal for **troubleshooting issues**, understanding system behavior, or maintaining records for compliance purposes
  - Examples: User login events, system errors, transaction records, etc.
- **Log aggregation systems:** Promtail/Grafana Loki, Elasticsearch/Kibana

Log aggregation allows us to correlate / aggregate logs from many services in one place

Logging aggregation architecture



Log visualization (Kibana)

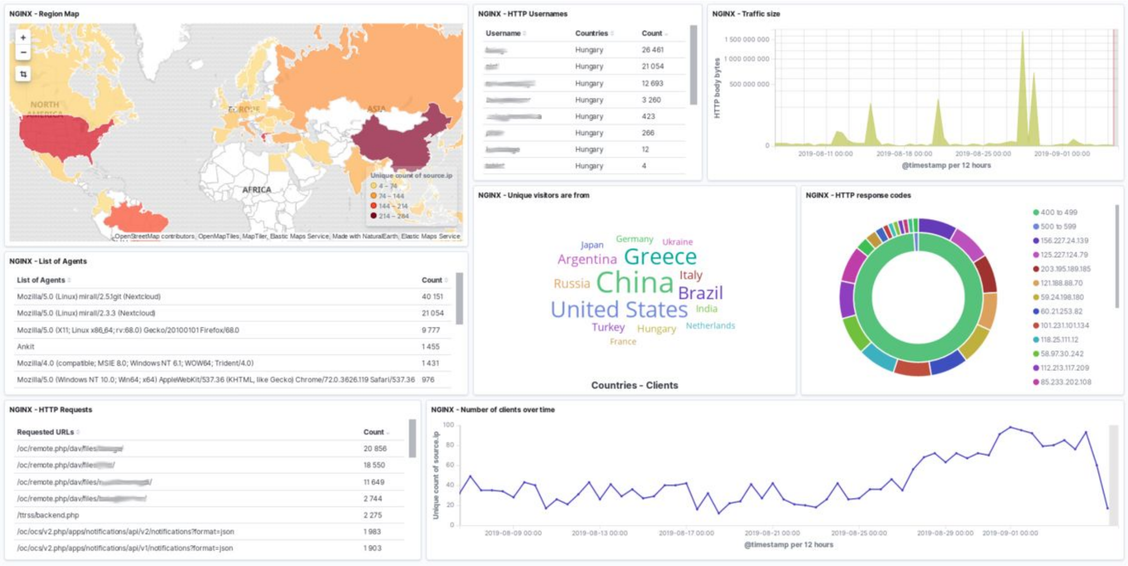


Kibana search interface

Central log aggregation parses metadata from logs and allows to filter by it

84

Log visualization (Kibana)



Kibana dashboard based on server logs

We can use the metadata also to derive metrics and visualisation

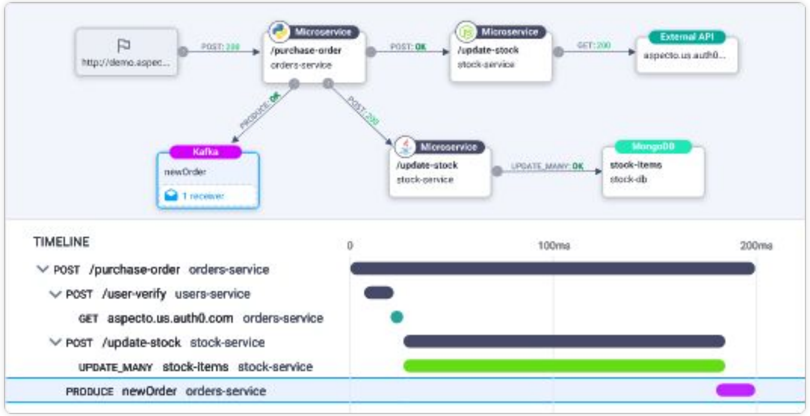
85

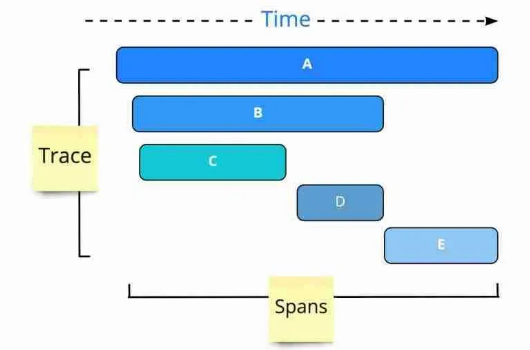
Logging: Best practices

- Determine the correct logging level
- Provide sufficient context
- Adopt structured logging
- Exclude sensitive data
- Capture full exception details
- Centralize your logs

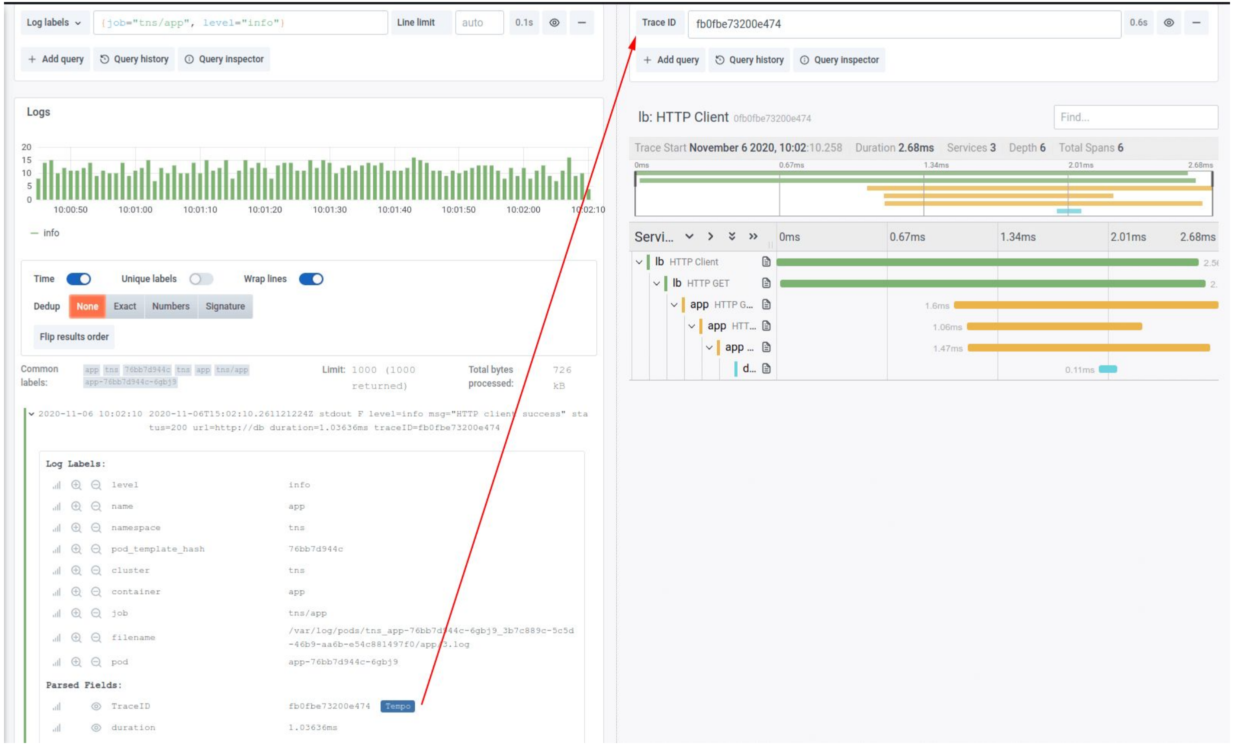
Tracing: Where is the problem?

- **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





Tracing: Log to trace correlation



Tracing: Best practices

- Enable distributed tracing
- Correlate traces with logs and metrics
- Include relevant information
- Implement context propagation
- Identify and analyze latency issues

Today’s learning goals

- **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