Docker, Deployments, and CI/CD

Aditya Putra

me@adityaputra.com
https://github.com/adityaputra/praktisimengajar-cloudcomputing

What we'll be learning

- Module 1: Introduction to Docker
- Module 2: Docker Images and Containers
- Module 3: Docker Networking and Volumes
- Module 4: Docker Compose
- Module 5: Deploying Docker Containers to AWS
- Module 6: Alternative Deployment Methods and CI/CD with AWS CodeDeploy

Introduction to Docker

Aditya Putra

me@adityaputra.com
https://github.com/adityaputra/praktisimengajar-cloudcomputing

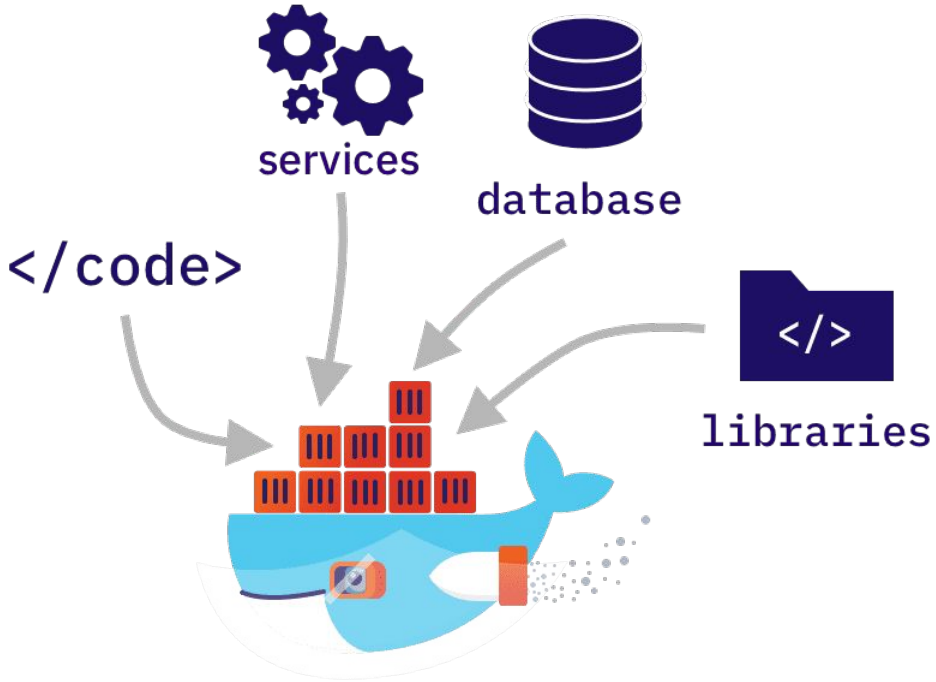
Module 1

Introduction to Docker

- What is Docker and why is it important?
- Key concepts: images, containers, Dockerfile, Docker Hub
- Installing Docker and running your first container

What is Docker?

Docker is an open-source platform that enables you to automate the deployment, scaling, and management of applications using containerization.


services
database
</code>
</>
libraries

Why is it important?

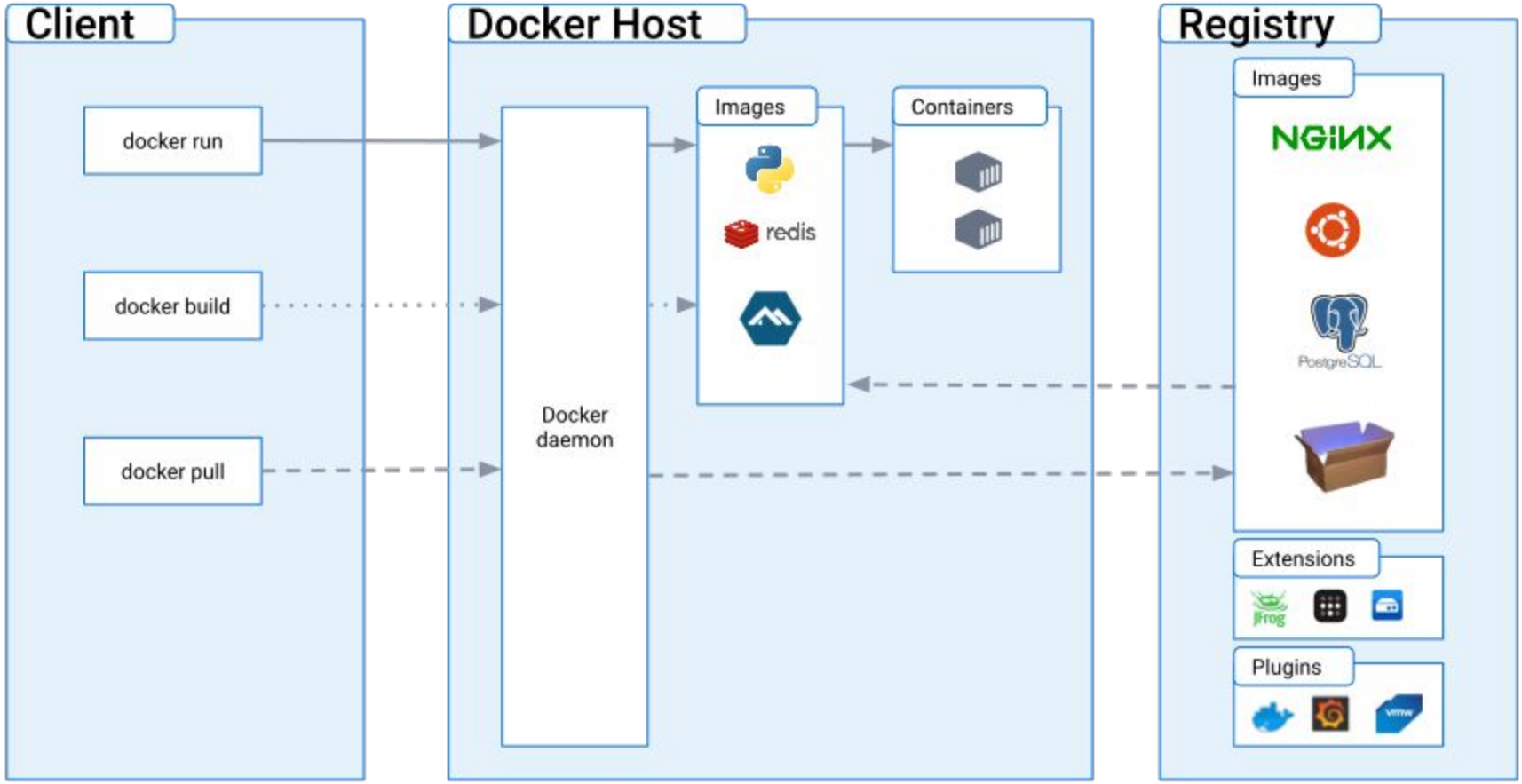
- **Consistent** environments: Containers provide a consistent and reproducible environment for applications, eliminating the "works on my machine" problem.
- Resource **efficiency**: Containers share the host system's OS kernel, resulting in lower resource usage compared to traditional virtualization.
- **Scalability**: Docker allows you to easily scale applications by running multiple containers across different hosts.
- **Isolation**: Containers provide isolation between applications, ensuring that they do not interfere with each other.

Containerization and cloud

- Flexibility and portability
- Rapid deployment
- Scalability and resource efficiency
- DevOps practices

Key concepts

- **Images**: Docker images are read-only templates that define the application's code, runtime, libraries, and dependencies. They serve as the basis for creating containers.
- **Containers**: Docker containers are instances of Docker images. They encapsulate the application and its dependencies, enabling consistent execution across different environments.
- **Dockerfile**: A Dockerfile is a text file that contains instructions for building a Docker image. It defines the base image, required packages, environment variables, and commands to run during container creation.
- **Docker Hub**: Docker Hub is a cloud-based **registry** that hosts public and private Docker images. It allows you to easily share and distribute Docker images with others.


Client
docker run
docker build
docker pull
Docker Host
Docker daemon
Images
redis
Containers
Registry
Images
NGINX
PostgreSQL
Extensions
Plugins

Running your first container

- Pulling image from docker registry (default is docker hub)
  - docker pull nginx:latest
- Running a container (you can also run container directly and docker will automatically pull the image for you if it’s not in your local)
  - docker run -d -p 80:80 nginx
  - docker run hello-world

Installing docker

- Install: https://docs.docker.com/get-docker/
- Awesome collection of docker-related resources: https://awesome-docker.netlify.app/

Managing your containers

- docker ps
  - docker ps -a
- docker stop
- docker start
- docker rm

Demo

Conclusion

Docker Images and Containers

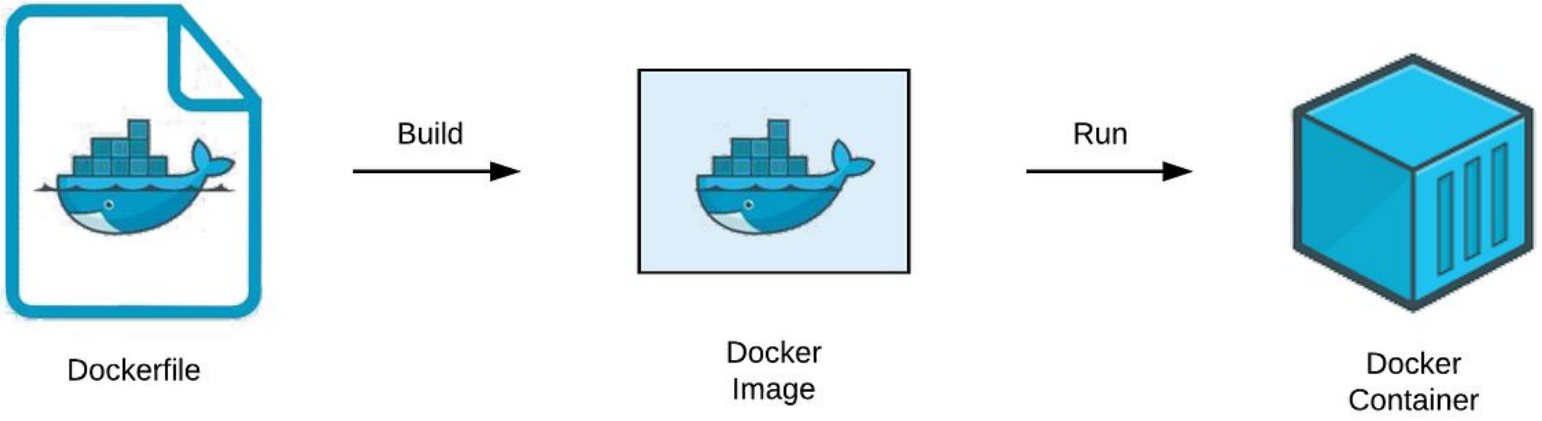
Aditya Putra

me@adityaputra.com
https://github.com/adityaputra/praktisimengajar-cloudcomputing

Module 2

- Creating Docker images using Dockerfiles
- Building images and tagging them
- Managing containers: starting, stopping, and removing containers
- Managing images: pulling, pushing, and deleting images from Docker Hub

Dockerfiles



```
#dependencies
FROM node:alpine

#working directory
WORKDIR /app

#copying dependency for our app into working directory
COPY src/package.json .

# command to run to install and configure app's dependencies
RUN npm install

# copy everything else into workdir
COPY src/. .

# exposing port for the app
EXPOSE 1000

# command to run our app
CMD ["node", "index.js"]
```

Building images and tagging them

- Building images from Dockerfiles using the **`docker build`** command.
- Tagging images with a specific version or label using the **`-t`** or **`--tag`** flag.
- Best practices for tagging images, such as using semantic versioning.

Demo

Managing containers

- Starting containers with the **docker run** command.
- Specifying container names, ports, and environment variables with flags.
- Stopping containers with the **docker stop** command.
- Removing stopped containers with the **docker rm** command.

Demo

Managing images

- Pulling images from Docker Hub using the **docker pull** command.
- Pushing images to Docker Hub using the **docker push** command.
- Managing images locally: listing images, inspecting image details, and deleting images.

Demo

Exploring docker hub and other registries

- https://hub.docker.com/
- https://aws.amazon.com/ecr/
- https://ghcr.io
- Many others

Demo

Assignment

- Create a docker image for a simple web app and publish the image on docker hub.
  - You can use any programming language of your choice
  - Be sure that people can simply run **docker pull** and **docker run** using your image name on docker hub
- Provide the docker image URL to us for submission

Docker Networking and Volumes

Aditya Putra

me@adityaputra.com
https://github.com/adityaputra/praktisimengajar-cloudcomputing

Module 3

Docker Networking and Volumes

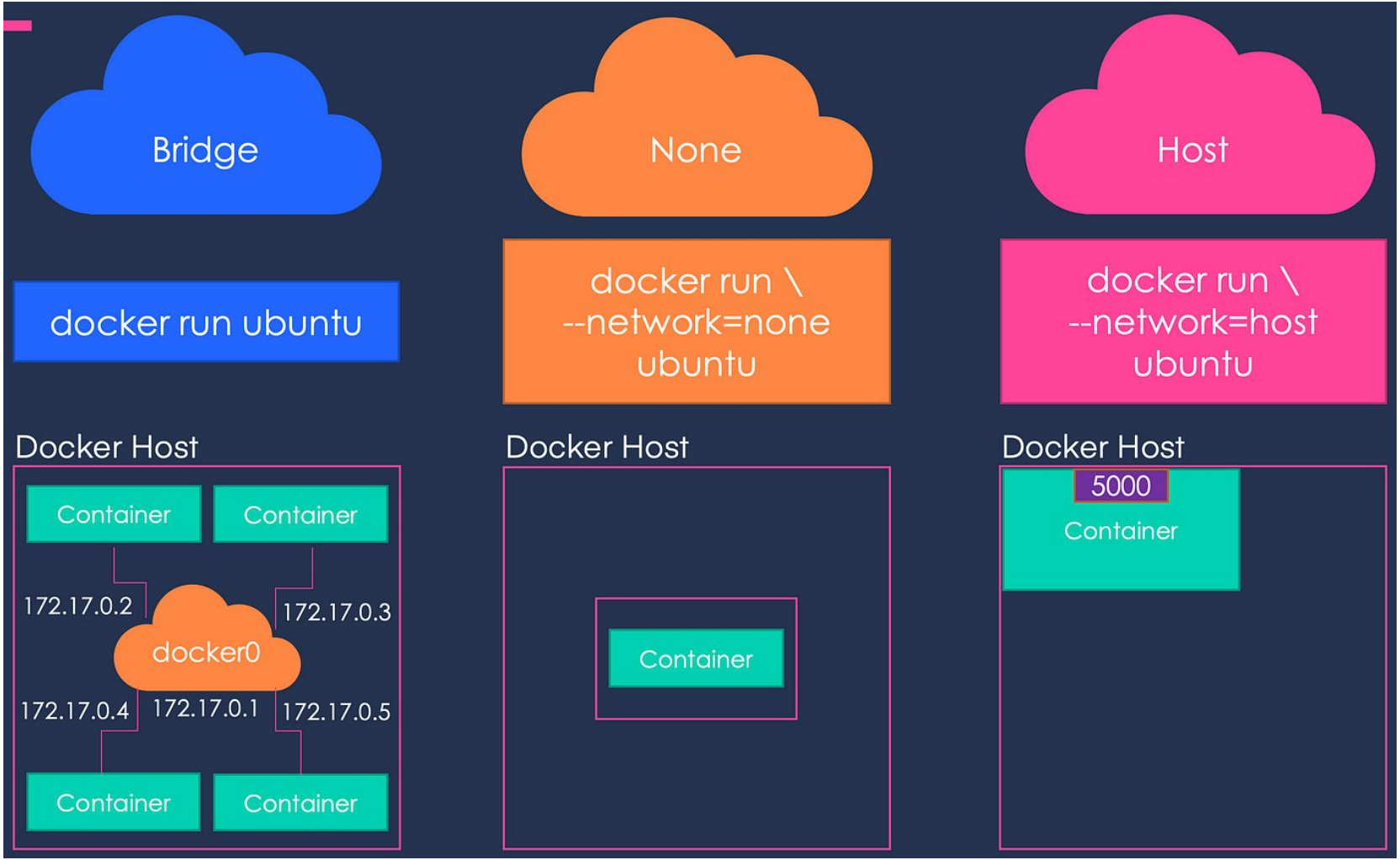
- Docker networking basics
- Docker network types
- Creating and managing Docker networks
- Docker volumes basics
- Creating and managing Docker volumes
- Using volumes for data persistence

Docker networking basics

- Default network: bridge
- Container IP addresses
- Container DNS resolution

Docker network types

- Bridge network: The default network that allows containers on the same host to communicate with each other using IP addresses.
- Host network: Containers use the host's network stack, sharing the host's IP address and network interfaces.
- Overlay network: Used for multi-host communication, allowing containers on different hosts to communicate securely.
- None: network is disabled


Bridge
None
Host
docker run ubuntu
docker run \
--network=none
ubuntu
docker run \
--network=host
ubuntu
Docker Host
Container
Container
172.17.0.2
172.17.0.3
docker0
172.17.0.4
172.17.0.1
172.17.0.5
Container
Container
Docker Host
Container
Docker Host
5000
Container

Creating and managing Docker networks

- Creating user-defined networks using the **docker network create** command.
- Connecting containers to networks using the **--network** flag when running containers.
- Inspecting network details using the **docker network inspect** command.
- Removing unused networks with the **docker network prune** command.

Demo

Docker volumes

- Volumes: Persistent data storage mechanisms in Docker that can be shared between containers and survive container restarts.
- Volume types:
  - named volumes
  - bind-mounts / host-mounted volumes
  - anonymous volumes, etc

Creating and managing Docker volumes

- Creating named volumes using the **docker volume create** command.
- Mounting volumes to containers using the **--volume** or **-v** flag when running containers.
- Inspecting volume details using the **docker volume inspect** command.
- Listing and removing unused volumes with the **docker volume prune** command.

Using volumes for data persistence

- Mapping directories from the host to containers using host-mounted volumes.
- Sharing volumes between containers in the same Docker network.
- Backing up and restoring Docker volumes.

Demo

Assignment

- Create a Docker container for a simple web server.
- Use any web server image of your choice (e.g., nginx, httpd, or apache).
- Configure the web server container to listen on port 80.
- Create a Docker network to connect the container.
- Implement a volume to persist the web server's data.
- Add a basic HTML page to the web server's root directory to display "Hello, Docker!"
- Test the application locally by running the container and accessing the web server via a web browser.
- Document the steps required to set up the application using Docker networking and volumes.

Docker Compose

Aditya Putra

me@adityaputra.com
https://github.com/adityaputra/praktisimengajar-cloudcomputing

Module 4

- Introduction to Docker Compose
- Writing a Docker Compose file
- Docker Compose commands
- Scaling services with Docker Compose
- Environment-specific configurations with Docker Compose
- Docker context

Introduction

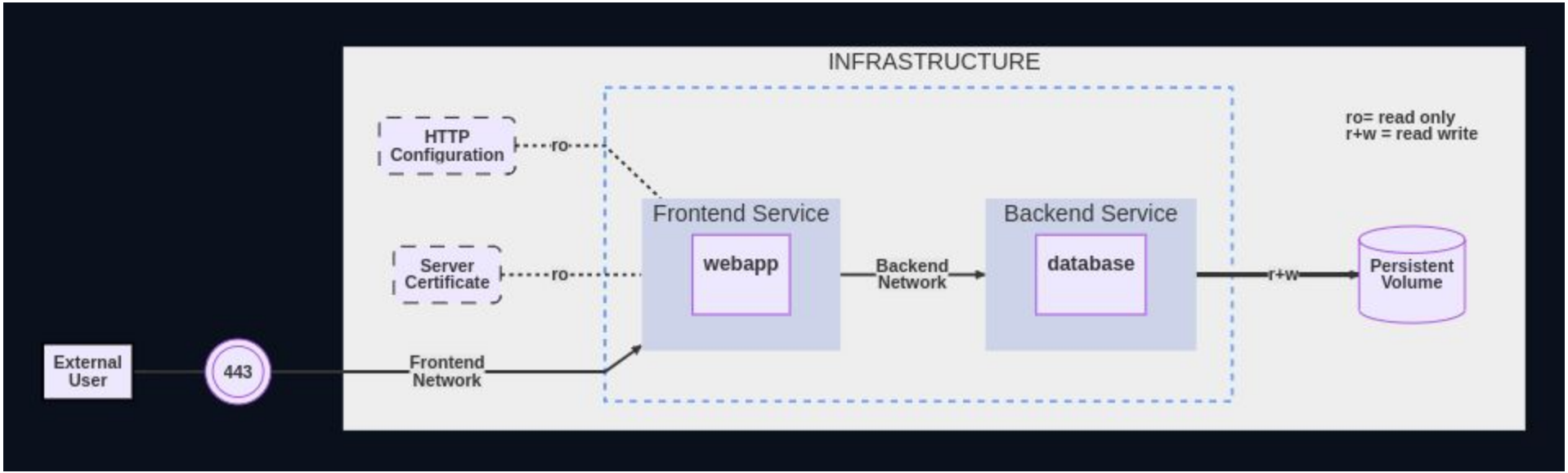
- Docker Compose: A tool for defining and running multi-container Docker applications using a YAML configuration file.
- Benefits of Docker Compose: Simplifies container management, automates container setup, and allows defining complex multi-container environments.

Installing

https://docs.docker.com/compose/install/

Docker compose file

```yaml
services:
  frontend:
    image: awesome/webapp
    ports:
      - "443:8043"
    networks:
      - front-tier
      - back-tier
    configs:
      - httpd-config
    secrets:
      - server-certificate

  backend:
    image: awesome/database
    volumes:
      - db-data:/etc/data
    networks:
      - back-tier

volumes:
  db-data:
    driver: flocker
    driver_opts:
      size: "10GiB"

configs:
  httpd-config:
    external: true

secrets:
  server-certificate:
    external: true

networks:
  # The presence of these objects is sufficient to define them
  front-tier: {}
  back-tier: {}
```


INFRASTRUCTURE
HTTP Configuration
ro
Server Certificate
ro
Frontend Service
webapp
Backend Network
Backend Service
database
r+w
Persistent Volume
ro= read only
r+w = read write
External User
443
Frontend Network

Docker compose commands

- **docker-compose up**: Starts containers defined in the Compose file.
- **docker-compose down**: Stops and removes containers created by the Compose file.
- **docker-compose build**: Builds or rebuilds container images defined in the Compose file.
- **docker-compose logs**: Displays logs from containers.
- **docker-compose exec**: Runs a command in a running container.

Scaling services with Docker Compose

- Scaling services horizontally using the **--scale** flag.
- Load balancing across replicated containers.

Environment-specific configs with Docker Compose

- Using environment variables in the Compose file for flexible configuration management.
- Managing different environment configurations for development, staging, and production.

Demo

Bonus

Some good self-hosted apps that you can simply docker-compose

- https://github.com/awesome-selfhosted/awesome-selfhosted
- https://github.com/Haxxnet/Compose-Examples
- https://github.com/docker/awesome-compose

Assignment

- Write a Docker Compose file (docker-compose.yml) to define a simple multi-container application.
- The application should consist of two services: a web server and a database.
- Use the official nginx image for the web server and the official mysql image for the database.
- Configure the web server to listen on port 80 and connect to the database.
- Specify environment variables, such as the database credentials, in the Docker Compose file.
- Ensure that the services are connected within a shared network.
- Test the application locally by running docker-compose up and accessing the web server via http://localhost.
- Document the steps required to deploy the application using Docker Compose.

Deploying Docker Containers to AWS

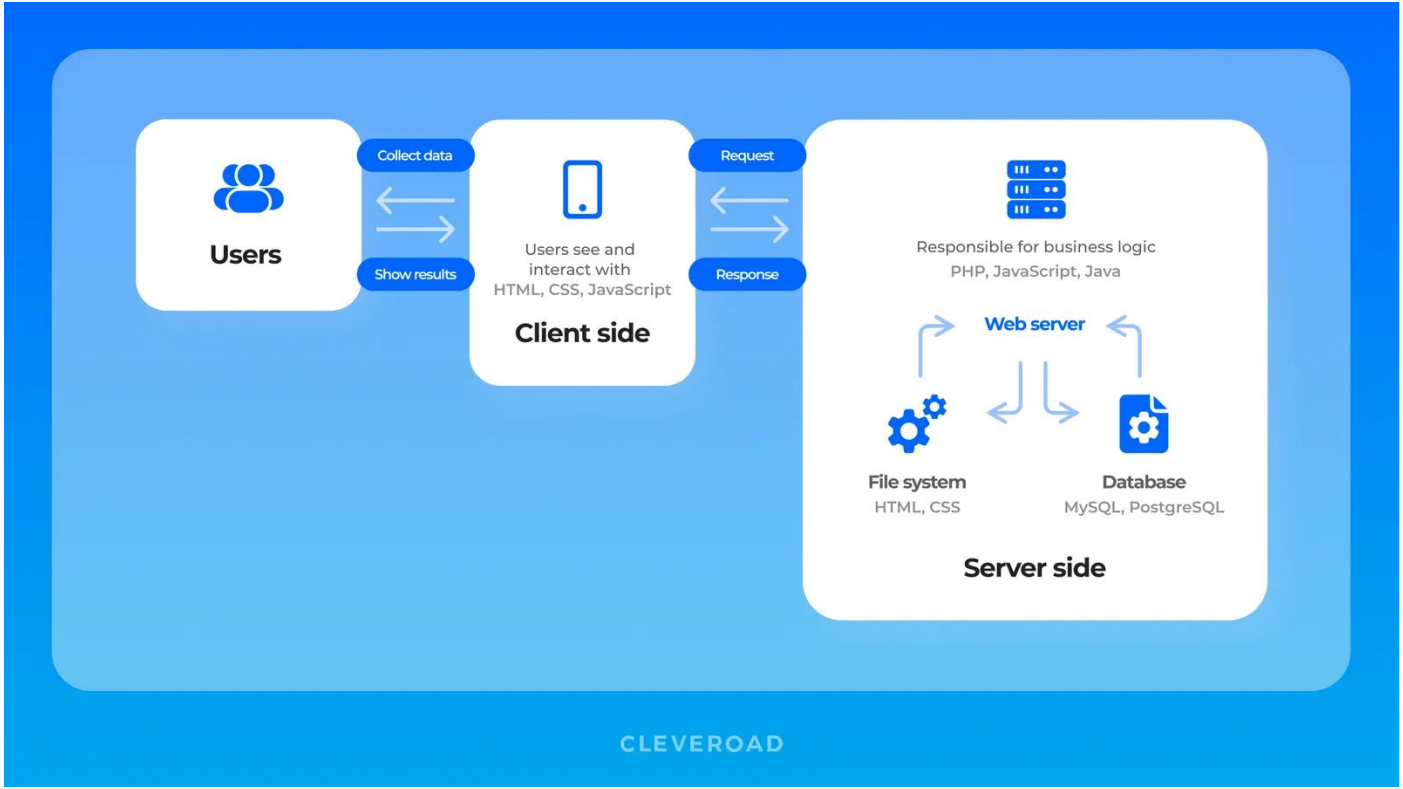
Aditya Putra

me@adityaputra.com
https://github.com/adityaputra/praktisimengajar-cloudcomputing

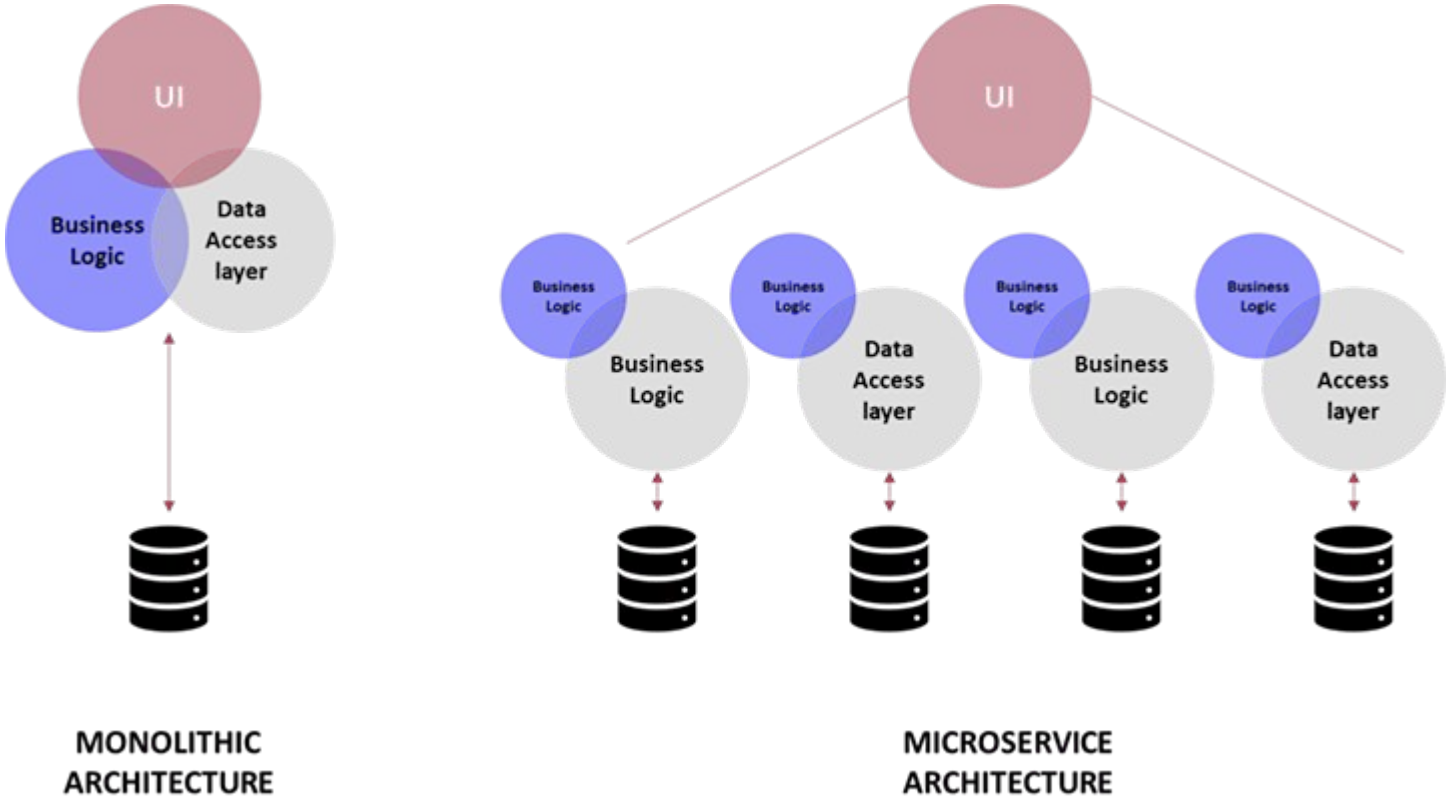
Module 5

- Common application architecture
- Traditional deployment techniques
- Container deployment on production

Common application architecture



Monolith vs Microservices

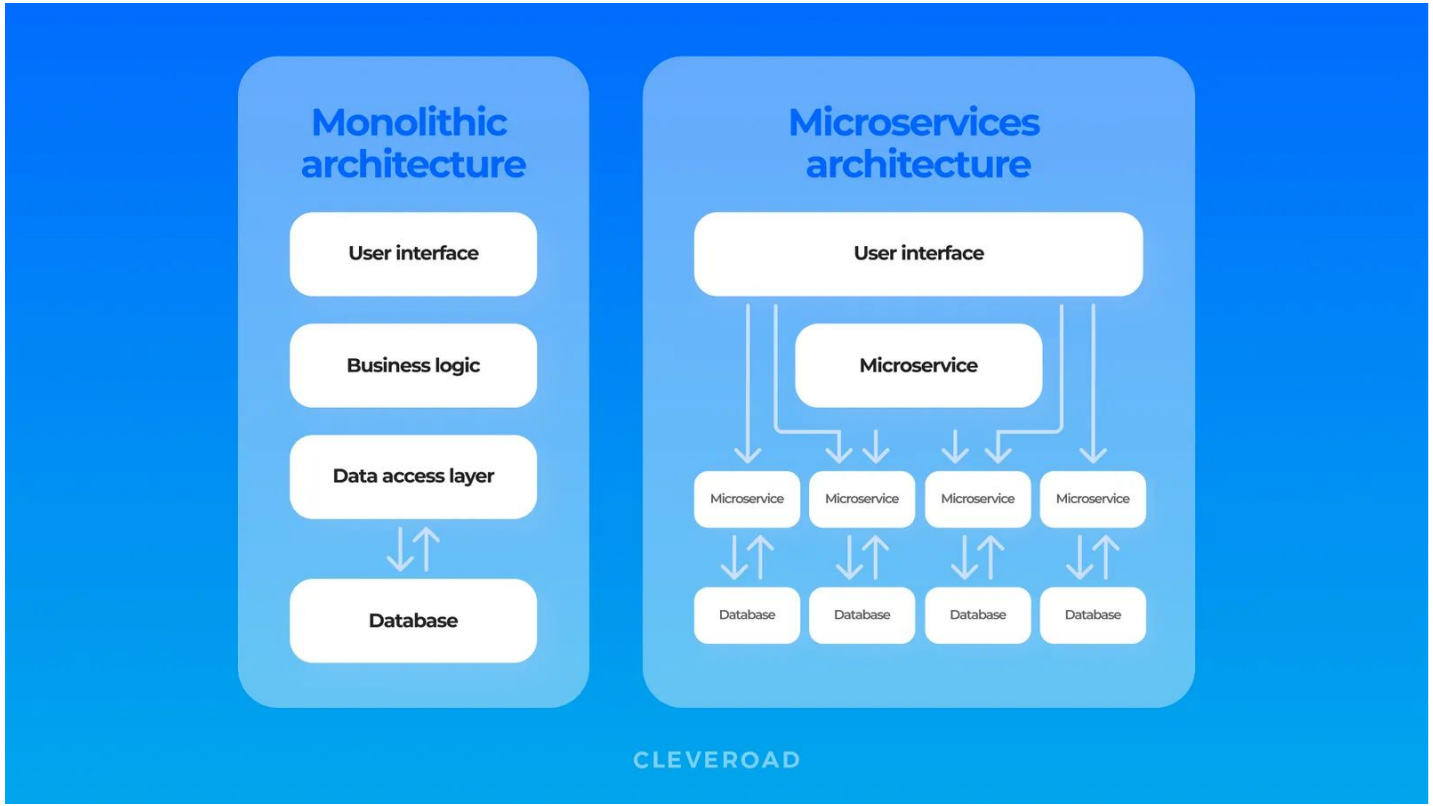


Traditional deployment techniques

1. Initial setup:
    a. Set up a version control system (e.g., git)
    b. Configure a web server
    c. Set up a database
    d. Prepare the deployment environment
    e. Configure environment-specific settings
    f. Set up cron jobs (if applicable)
2. Installing / updating app:
    a. Transfer files to the deployment target
    b. Set file permissions
    c. Install dependencies
3. Test the deployment

Showcase & demo

Containers deployment



Containerized deployment techniques - v1

1. Integrated DevOps flow setup:
    a. Setup docker image containing all necessary configuration
2. Initial deployment setup (on the deployment target):
    a. Set up a container engine (e.g., docker)
3. Installing / updating app:
    a. Pull the updated container image and run
4. Test the deployment

Monolith (left) vs Microservices (right)

- Pros:
  - Simplicity
  - Easier Testing
  - Reduced Complexity

- Cons:
  - Scalability
  - Limited Technology Flexibility
  - Team Collaboration
  - Deployment Speed

- Pros:
  - Scalability
  - Technology Flexibility
  - Team Autonomy
  - Fault Isolation

- Cons:
  - Complexity
  - Infrastructure Overhead
  - Distributed System Challenges

Which one to choose?

It's important to note that the choice between monolithic and microservices deployment depends on various factors, including the complexity of the application, scalability requirements, team size, and long-term goals. Both approaches have their pros and cons, and **the decision should align with the specific needs of the project and organization**.

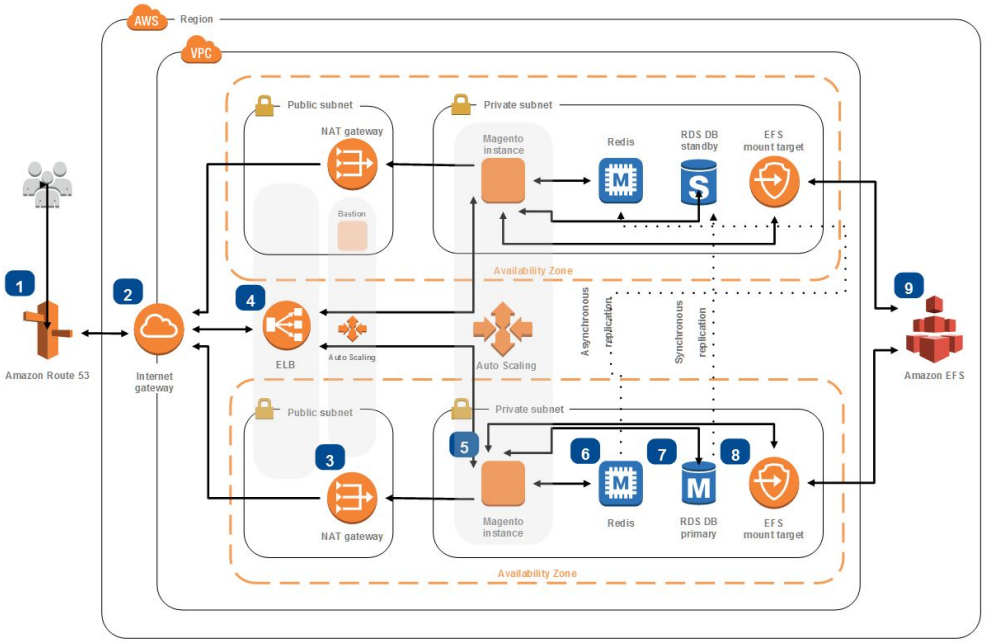
Showcase & demo

Deploying advanced apps

Magento CE Hosting

Running Magento Community Edition (CE) on AWS

Magento Community Edition (CE) is a flexible, open-source commerce platform for developers and small businesses. This reference architecture simplifies the complexity of deploying a scalable and highly available Magento CE commerce platform on AWS.

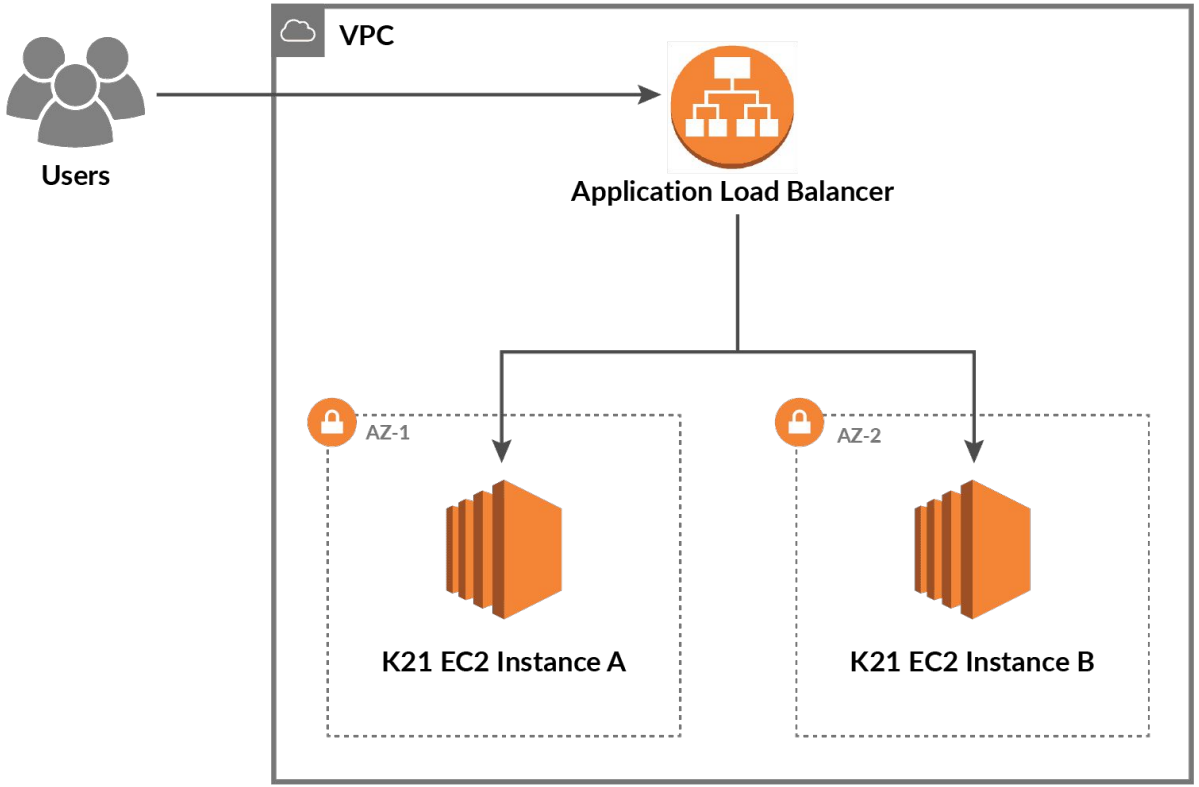


1. **Amazon Route 53** provides DNS configuration and routes traffic to Elastic Load Balancing (ELB) endpoints.
2. An **internet gateway** allows communication between instances in your VPC and the internet.
3. **NAT gateways** in each public subnet enable Amazon EC2 instances in private subnets to access the internet.
4. Use an **ELB Load Balancer** to distribute web traffic across an Auto Scaling group of Amazon EC2 instances in multiple Availability Zones.
5. Run your Magento commerce site using an **Auto Scaling group** of **Amazon EC2 instances**. Install the latest versions of Magento CE, Nginx web server, and PHP 7. Then, build an Amazon Machine Image (AMI) that the Auto Scaling group launch configuration can use to launch new instances in the group.
6. If database access patterns are read-heavy, consider using a caching layer like **Amazon ElastiCache for Redis** in front of the database layer to cache frequently accessed data.
7. Simplify your database administration by running your database layer in **Amazon RDS** using either Aurora or MySQL.
8. Amazon EC2 instances access the shared Magento data in an Amazon EFS file system using **mount targets** in each Availability Zone in your VPC.
9. Use an **Amazon EFS** network file system so that Magento instances can access your shared, unstructured Magento data such as images, media files etc.

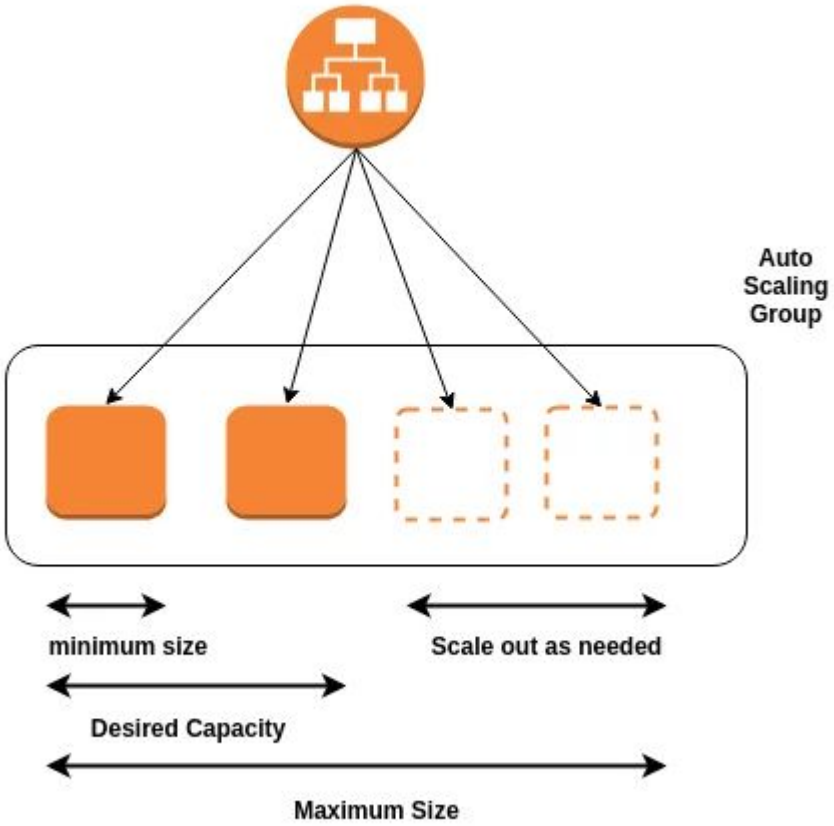
amazon web services

AWS Reference Architectures

Load balancing with EC2



Auto-scaling with EC2



Showcase & demo

Infrastructure as Code

Aditya Putra

me@adityaputra.com
https://github.com/adityaputra/praktisimengajar-cloudcomputing

Bonus

Infrastructure as Code

IaC: Infrastructure as Code

IaC is a methodology for managing and provisioning infrastructure resources through machine-readable definition files, rather than manual configuration.

It enables the automation and versioning of infrastructure deployment, making it repeatable, consistent, and easily manageable.

Benefits of IaC

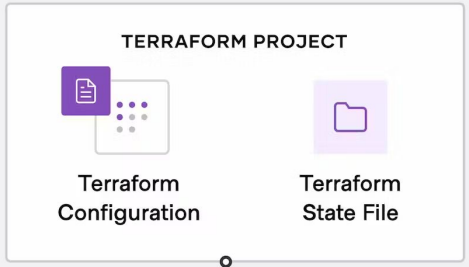
- Automates infrastructure provisioning
- Treats infrastructure as software
- Eliminates manual errors
- Enables version control and collaboration
- Tools: AWS CloudFormation, Terraform, Ansible
- Fast, consistent, and scalable deployments
- Benefits: speed, reproducibility, scalability
- Supports DevOps practices
- Improves infrastructure management

Common tools

- Terraform
- Ansible
- AWS CloudFormation

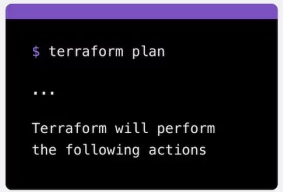
Write

Define infrastructure in configuration files



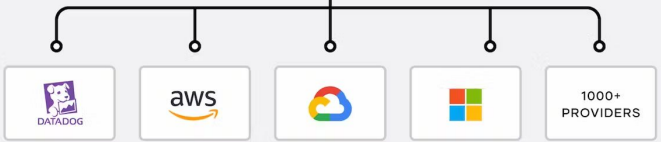
Plan

Review the changes Terraform will make to your infrastructure



Apply

Terraform provisions your infrastructure and updates the state file.



Showcase & Demo

Alternative Deployment Methods and CI/CD with AWS CodeDeploy

Aditya Putra

me@adityaputra.com
https://github.com/adityaputra/praktisimengajar-cloudcomputing

Module 6

.

DevOps, CI/CD, and Advanced Topics

Aditya Putra

me@adityaputra.com
https://github.com/adityaputra/praktisimengajar-cloudcomputing

Module 6

- DevOps
- CI/CD
- CDN
- Firewall

The hypothesis (or prediction)

What do you think will happen?

Research

Explain all of the research you've done about this issue/challenge.

What was the goal of your research? Be sure to explain how you found it and anyone who might have helped you!

My testing method

Each scientist uses different methods of experimentation

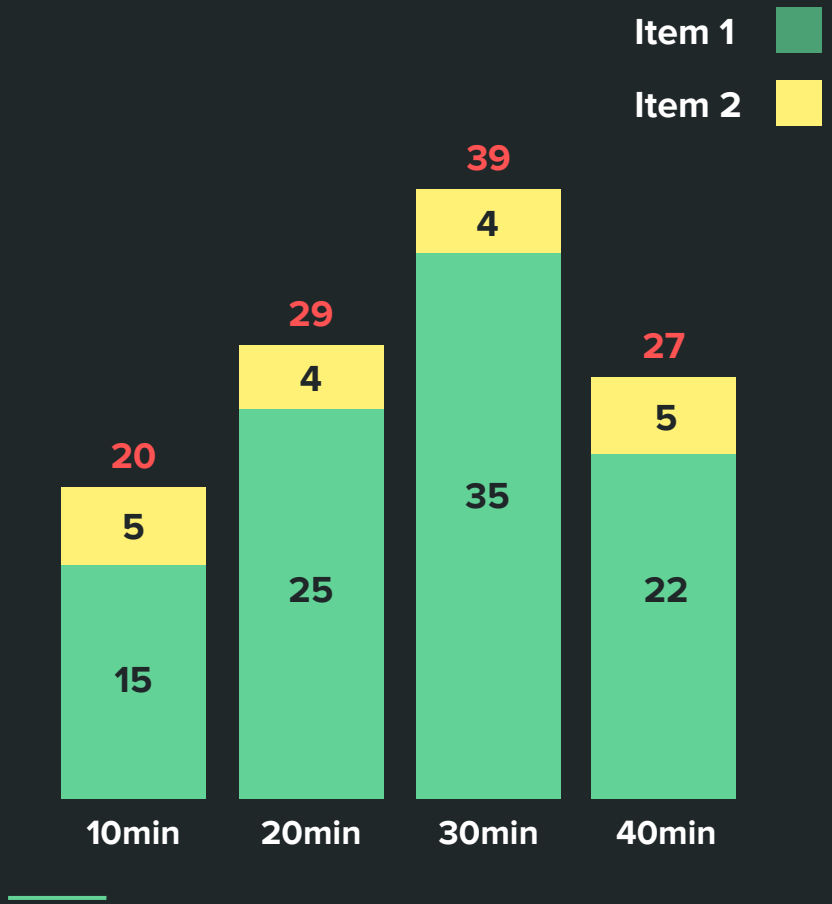
What methods did you use in your experiment?

- Lorem ipsum dolor sit amet, consectetur adipiscing elit
- Incididunt ut labore et dolore
- Consectetur adipiscing elit, sed do eiusmod tempor incididunt

Experiment data

Record the information you get from your experiment

Include a table or graph to display what you see

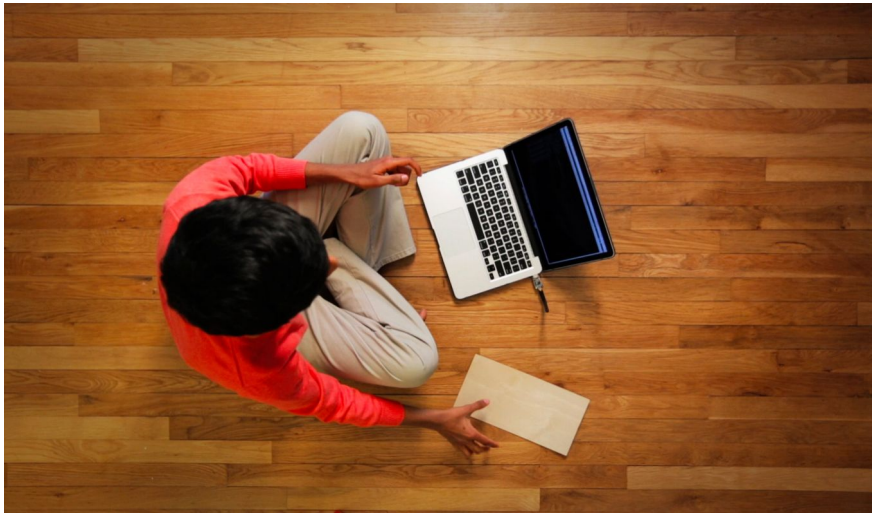


Aha!

My discoveries

What did you learn after testing?

1. Lorem ipsum dolor sit amet, consectetur adipiscing elit
2. Incididunt ut labore et dolore
3. Consectetur adipiscing elit, sed do eiusmod tempor incididunt



This is the most important takeaway that everyone has to remember.

Conclusion

What is the conclusion of your experiment? Did the results support your hypothesis or predicted outcome? How will your findings help the area of science you've researched?

What will I do next?

What will you do with your findings next? How will you further your research/findings?