Cloud Application Architecture

Foundations Workshop

Understanding how cloud systems are designed — before tools and vendors

January 31st, 2026

Workshop Overview

What You'll Learn

- Cloud fundamentals and core building blocks
- Application architecture patterns
- Modern deployment concepts and workflows
- Hands-on labs to reinforce concepts
- Foundation for Azure & GCP workshops

Focus: How the pieces connect — not tool mastery

Your Two-Day Journey

Day 1

FOUNDATIONS

Cloud & Application basics

- Fundamentals
- Building Blocks
- Architecture
- Principles

Day 2

DELIVERY

Tools & deployment concepts

- Dev Workflow
- Containers
- DevOps & CI/CD
- IaC

Your Instructor

Ahmed Bedair

Managing Delivery Architect

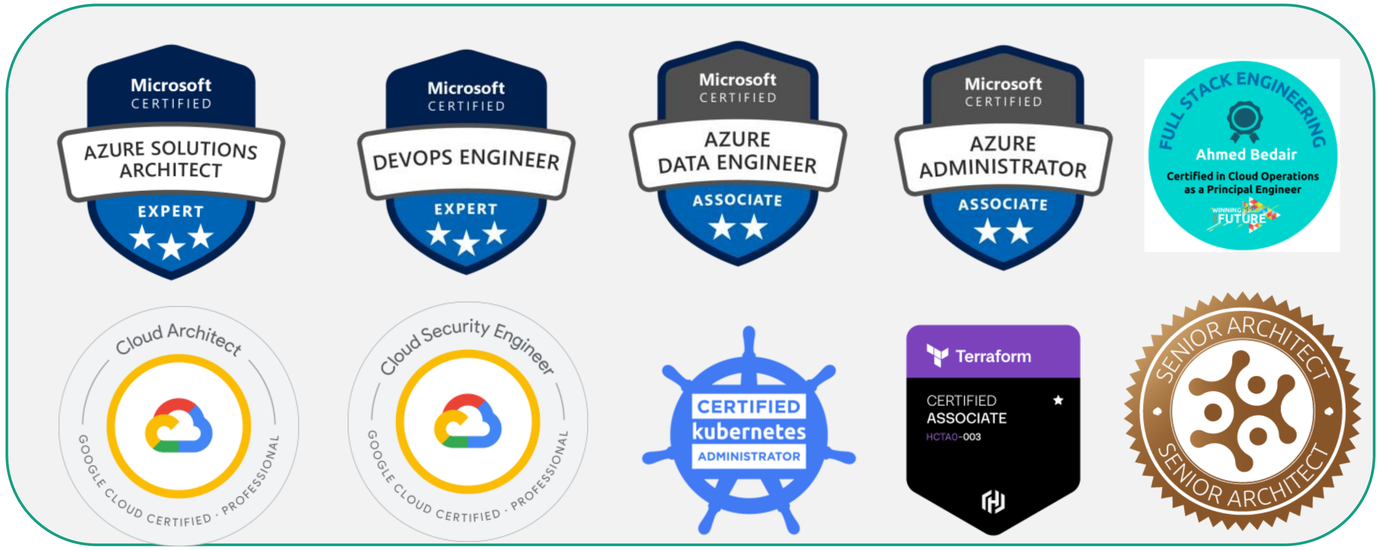
Helping organizations migrate to and build on the cloud.

Email: abedair@gmail.com
LinkedIn: https://linkedin.com/in/abedair/

Experience Areas

- 25+ years in IT Infrastructure
- Career journey:
  - Technical Trainer
  - IT Manager
  - IT Consultant
- Focus:
  - IT Solutions Architecture
  - Cloud Platform
  - Application Modernization
  - DevOps, IaC and automation.

Certifications


Microsoft CERTIFIED AZURE SOLUTIONS ARCHITECT EXPERT
Microsoft CERTIFIED DEVOPS ENGINEER EXPERT
Microsoft CERTIFIED AZURE DATA ENGINEER ASSOCIATE
Microsoft CERTIFIED AZURE ADMINISTRATOR ASSOCIATE
FULL STACK ENGINEERING Ahmed Bedair Certified in Cloud Operations as a Principal Engineer
Cloud Architect GOOGLE CLOUD CERTIFIED · PROFESSIONAL
Cloud Security Engineer GOOGLE CLOUD CERTIFIED · PROFESSIONAL
CERTIFIED kubernetes ADMINISTRATOR
Terraform CERTIFIED ASSOCIATE HCTA0-003
SENIOR ARCHITECT SENIOR ARCHITECT

How This Course Is Designed

What We Will Do	What We Won't Do
✓ Build mental models for cloud thinking	✗ Deep dive into every service option
✓ Understand patterns and when to use them	✗ Production-ready configurations
✓ Learn concepts that apply to any cloud	✗ Vendor-specific certifications
✓ Get hands-on practice with real tools	✗ Exhaustive tool comparisons

Each topic could be its own workshop — today we build the foundation to connect them all.

Our Approach:

Big Picture View

See how everything connects

not

Deep Dive

Every technical detail

What You'll Walk Away With

✓ Understand Cloud Concepts

Know the core building blocks and how they connect

✓ Recognize Patterns

Identify common architecture designs and when to use them

✓ Confidence to Explore

Foundation ready for Azure & GCP workshops

This workshop is a starting point — not the finish line.

By the end, you'll have a solid foundation to

continue your cloud learning journey with confidence.

Agenda

DAY 1 — Architecture

1. **Cloud Fundamentals**
   Why cloud, service models
2. **Core Building Blocks**
   Compute, storage, network, identity
3. **Application Architecture**
   Patterns and design decisions
4. **Architecture Principles**
   Security, reliability, cost, ops
5. **Day 1 Lab**
   Hands-on practice

DAY 2 — Delivery

6. **Developer Workflow**
   Local to cloud journey
7. **Containers**
   Docker fundamentals
8. **DevOps & CI/CD**
   Automation pipelines
9. **Infrastructure as Code**
   Terraform basics

★ **End-to-End + Day 2 Lab**
Put it all together

This workshop is the foundation for the Azure & GCP hands-on workshops

Workshop Resources

GitHub Repository

`github.com/bedairahmed/cloudworkshop`

Labs, cheatsheets, samples, and documentation

Azure Portal

`portal.azure.com`

Login: studentXX@ml.cloud-people.net

Cheatsheets

Git | Azure CLI | GCP CLI | Docker
Terraform | Kubernetes | Helm
Linux | PowerShell | Python
YAML | Pipelines | GitHub | VS Code
Ansible | README Guide

Hands-On Labs

Lab 00: Clone Repository & Setup
Lab 01: Azure Portal Login
Lab 02: Create a Virtual Machine

Lab 00 Clone Repository & Setup Environment

Objective: Clone the workshop repository and set up VS Code

Time: 5 minutes

Steps:

1. Open VS Code
2. Clone: github.com/bedairahmed/cloudworkshop
3. Open the cloned folder in VS Code

`labs/00-clone-repo.md`

Section 1

Cloud Fundamentals

Understanding the basics

What is Cloud Computing?

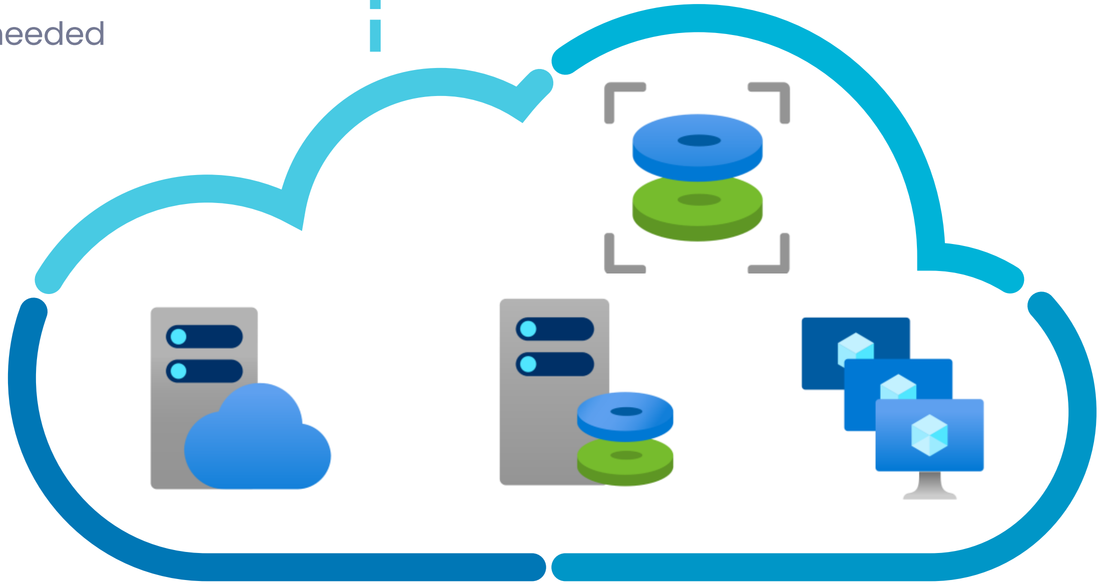
On-Demand Resources

Compute, storage, and networking available when needed

No Hardware to Manage

No servers to buy

No data centers to run



Pay for Usage

Costs scale with usage

No upfront investment

Focus on Applications

Build and deploy software

Infrastructure is handled for you

Why Organizations Use the Cloud?

Scalability

Scale up or down based on demand

Cost Efficiency

No upfront hardware
Pay for what you use

Speed & Agility

Faster development
Quicker experimentation

Reliability

Built-in redundancy High availability by design

Focus on Apps

Less infrastructure work
More application development

Cloud Service Models

🟩 Client Responsibility

🟦 Cloud Provider Responsibility

On-Premises	IaaS	PaaS	SaaS
🟩 Application	🟩 Application	🟩 Application	🟦 Application
🟩 Data	🟩 Data	🟩 Data	🟦 Data
🟩 Runtime	🟩 Runtime	🟦 Runtime	🟦 Runtime
🟩 Middleware	🟩 Middleware	🟦 Middleware	🟦 Middleware
🟩 OS	🟩 OS	🟦 OS	🟦 OS
🟩 Virtualization	🟦 Virtualization	🟦 Virtualization	🟦 Virtualization
🟩 Servers	🟦 Servers	🟦 Servers	🟦 Servers
🟩 Storage	🟦 Storage	🟦 Storage	🟦 Storage
🟩 Networking	🟦 Networking	🟦 Networking	🟦 Networking

Lab 01 Clone Repository & Setup Environment

Objective: Sign in to Azure Portal and set up Microsoft Authenticator

Time: 10 minutes

Steps:

1. Go to portal.azure.com
2. Sign in with your student credentials
3. Install Microsoft Authenticator on your phone
4. Scan the QR code to complete MFA setup

`labs/01-azure-login.md`

Section 2

Core Cloud Building Blocks

The essential components

The Four Cloud Building Blocks

Compute

Run applications and workloads

Virtual machines, containers, and serverless compute provide processing power on demand.

Storage & Data

Store and manage data

Databases, object storage, and file systems support structured and unstructured data.

Security

Control access and protect resources

Identity management, authentication, authorization, and security policies keep systems safe.

Networking

Connect users, apps, and services

Virtual networks, subnets, routing, and load balancing enable secure communication.

Compute

Run applications and workloads

Virtual Machines

- Full operating system
- Maximum control
- Traditional workloads

Cloud servers similar to on-prem machines

Containers

- App + dependencies
- Lightweight and fast
- Portable across environments

Standard way to package and run apps

Serverless

- No server management
- Auto scaling
- Pay per execution

Focus on code, platform handles the rest

Storage & Data

How applications store and move data?

Storage

Object Storage
- Files & objects
- Images, backups
- Blob, S3

Block Storage
- Disk for compute
- High performance
- OS, DB disks

File Storage
- Shared folders
- File system access
- Legacy apps

Databases

Relational
- Tables & rows
- Transactions
- Structured data

Non-Relational
- Flexible schema
- Large scale
- JSON / key-value

Platform Services (Data Handling)

CACHE
- In-memory data
- Very fast access
- Reduce DB load

Event Communication
- Events & queues
- Async processing
- Decouple services

Secrets Manager
- Secrets & credentials
- Security-focused storage

Container Registry
- container images
- Deployment artifacts

Networking

Connect users, applications, and services

Virtual Network

- Private cloud network
- Isolates resources

Subnets

- Divide the network
- Organize workloads

Routing

- Control traffic paths
- Internal & internet flow

Security Rules

- Allow or block traffic
- Protect resources

Load Balancing

- Distribute traffic
- High availability

Connectivity

- Connect to on-prem & other networks
- Hybrid and private access

Security

Protect access, traffic, and data

Identity & Access

- Who can access resources
- Users, roles, permissions

Authentication

- Verify identity
- User and service login

Authorization

- Control allowed actions
- Least privilege access

Secrets Management

- Store passwords and keys
- Secure runtime access

Network Security

- Control network access
- Firewalls, IDS / IPS

Data Protection

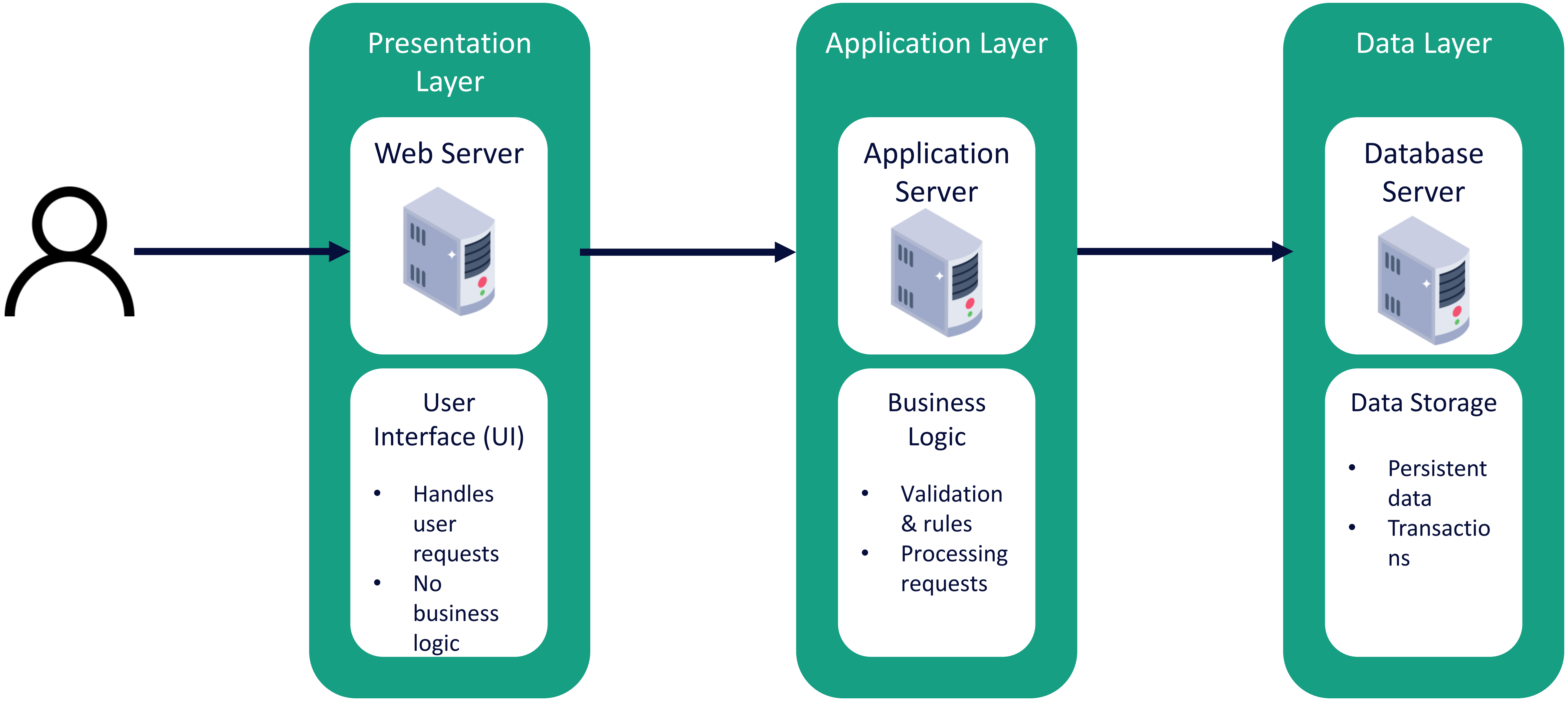
- Encrypt stored data
- Encrypt data in transit

Section 3

Application Architecture

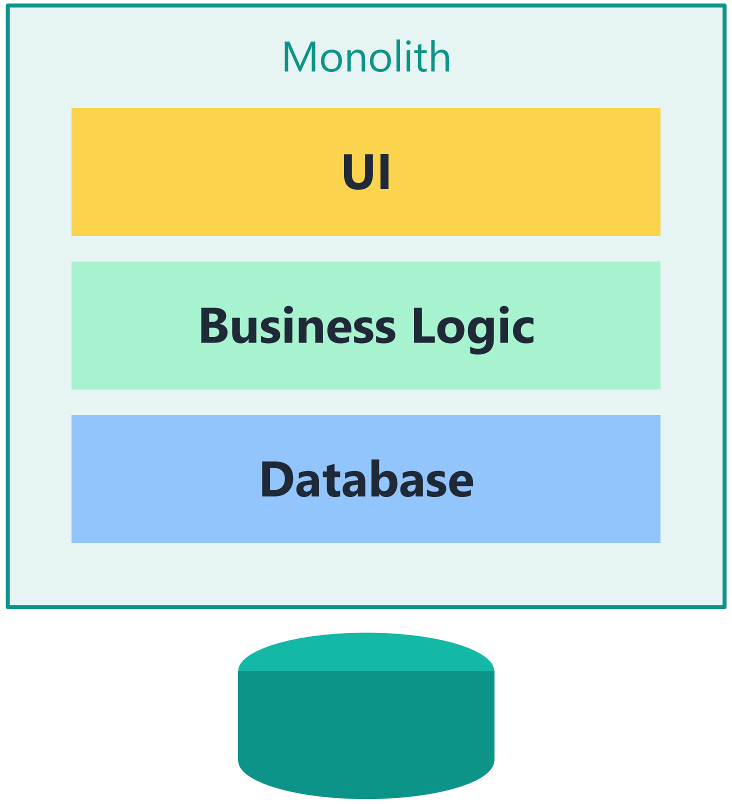
How are applications designed?

Three-Tier Application Model



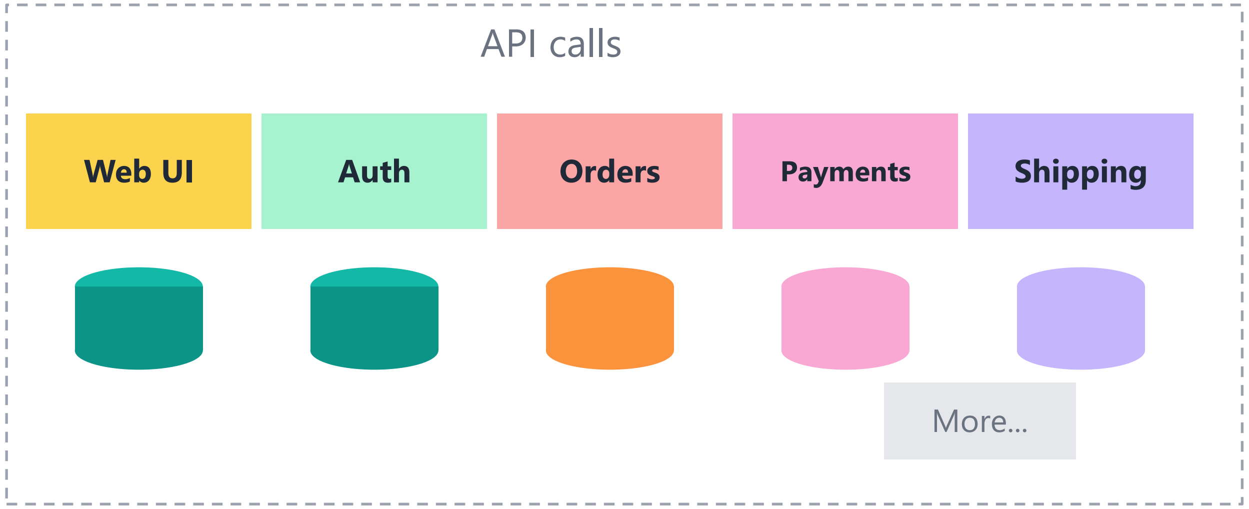
Monolith vs Microservices

Monolith



- Single unit
- Simple to start
- Harder to scale and change

Microservices

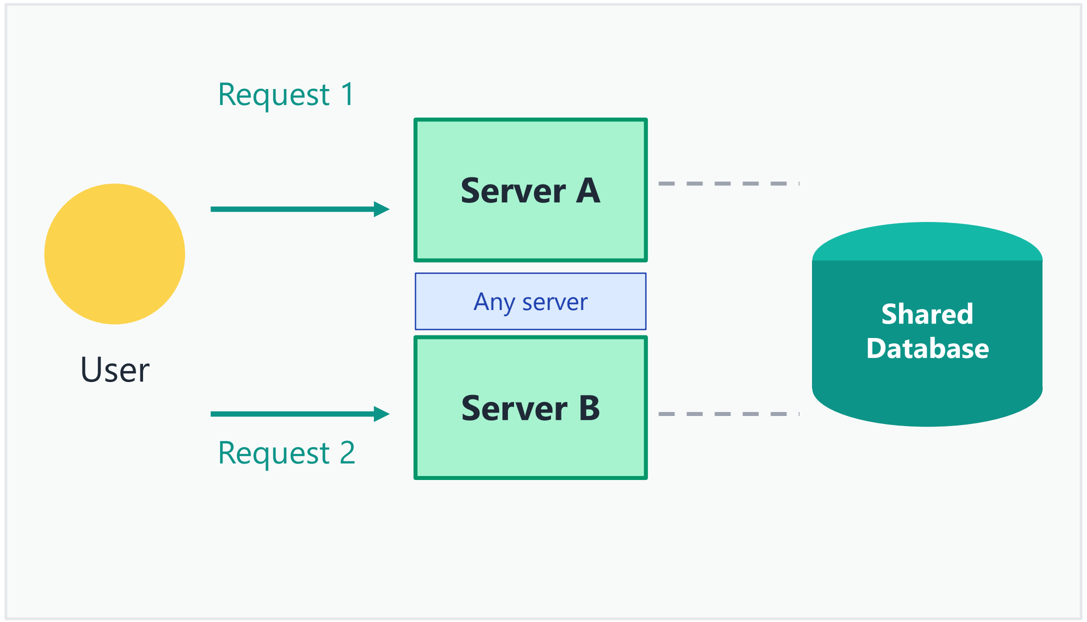


- Small, independent services
- Scales independently
- Faster innovation

Stateless vs Stateful Applications

Stateless

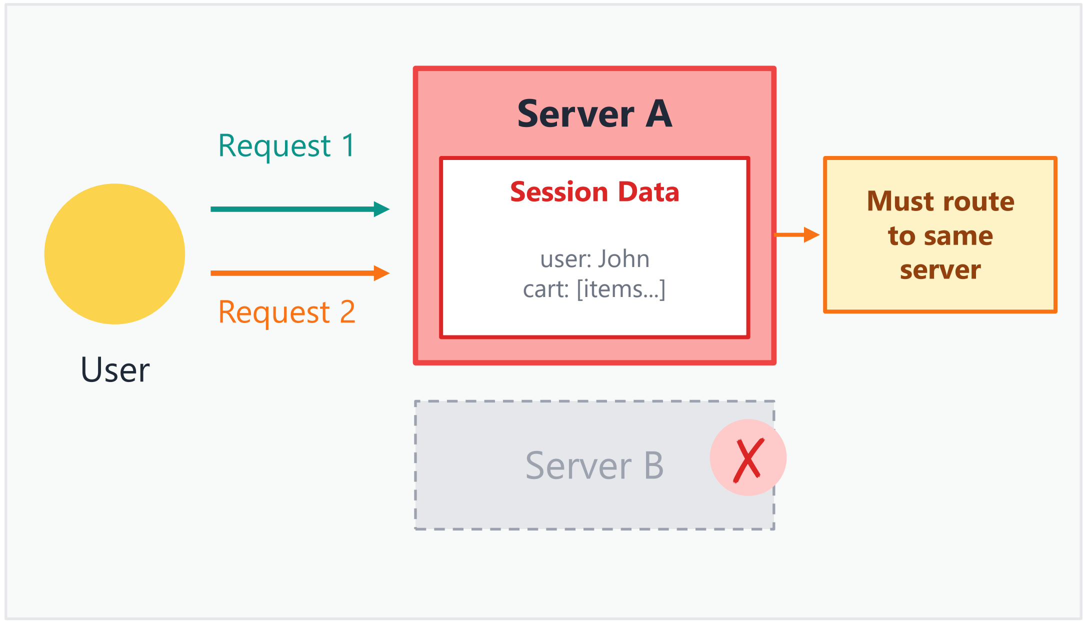
"Each request is independent"



- Easy to scale horizontally
- Any server can handle any request
- State stored externally

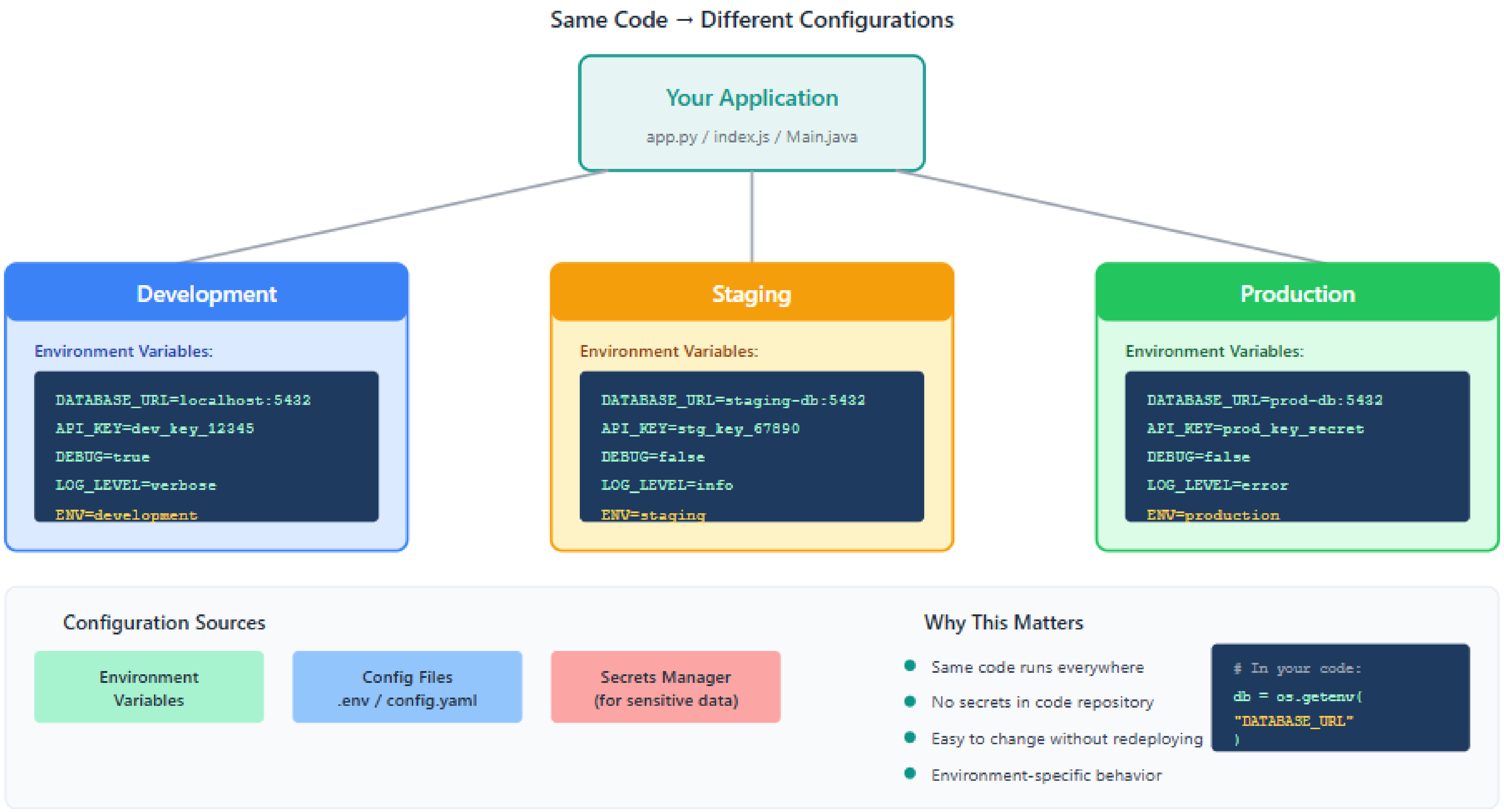
Stateful

"Server remembers previous requests"



- Harder to scale
- Requires sticky sessions
- State stored in server memory

Configuration & Environments


Same Code → Different Configurations
Your Application
app.py / index.js / Main.java
Development
Staging
Production
Environment Variables:
Configuration Sources
Environment Variables
Config Files .env / config.yaml
Secrets Manager (for sensitive data)
Why This Matters

Development — Environment Variables:

```
DATABASE_URL=localhost:5432
API_KEY=dev_key_12345
DEBUG=true
LOG_LEVEL=verbose
ENV=development
```

Staging — Environment Variables:

```
DATABASE_URL=staging-db:5432
API_KEY=stg_key_67890
DEBUG=false
LOG_LEVEL=info
ENV=staging
```

Production — Environment Variables:

```
DATABASE_URL=prod-db:5432
API_KEY=prod_key_secret
DEBUG=false
LOG_LEVEL=error
ENV=production
```

Why This Matters

- Same code runs everywhere
- No secrets in code repository
- Easy to change without redeploying
- Environment-specific behavior

```
# In your code:
db = os.getenv(
"DATABASE_URL"
)
```

Section 4

Cloud Architecture Principles

Designing for the cloud

Cloud Architecture Principles

Different clouds, same foundations

Every major cloud provider has a Well-Architected Framework:

AWS **Azure** **Google Cloud**

They all cover the same core pillars:

Security	**Reliability**	**Performance**	**Cost**	**Operations**
Protect data and systems	Keep running when things fail	Scale to meet demand	Don't waste money	Monitor and improve

Key Insight:

Learn the principles once → Apply them anywhere

Security by Design

"Security is not an afterthought — it's built in from day one"

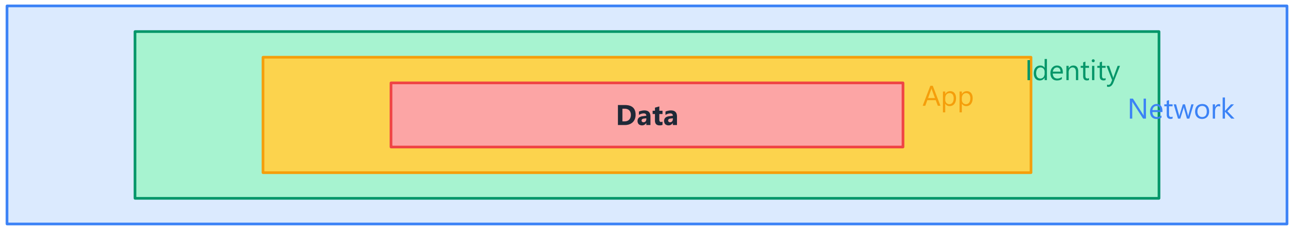
Core Concepts:

Least Privilege	Give only the access needed, nothing more
Defense in Depth	Multiple layers of protection
Zero Trust	Verify everything, trust nothing
Encrypt Everything	Data at rest and in transit

Security Layers:

Network	Firewalls, private subnets
Identity	Authentication, authorization
Application	Input validation, secrets
Data	Encryption, backups

Defense in Depth:



Reliability & Availability

"Plan for failure — because things will fail"

Key Concepts:

Availability	System is up and accessible
Redundancy	No single point of failure
Fault Tolerance	Keep working when components fail
Disaster Recovery	Recover from major outages

Availability Targets:

Target	Downtime/Year	Use Case
99%	3.65 days	Internal tools
99.9%	8.7 hours	Business apps
99.99%	52 minutes	Critical systems
99.999%	5 minutes	Mission critical

Design Question:

"What happens if this server / region / service goes down?"

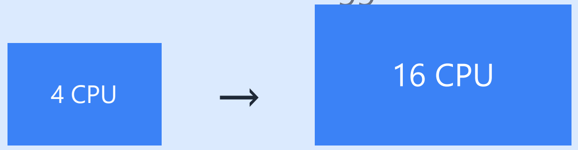
Performance & Scalability

"Handle growth without rewriting everything"

Two Types of Scaling:

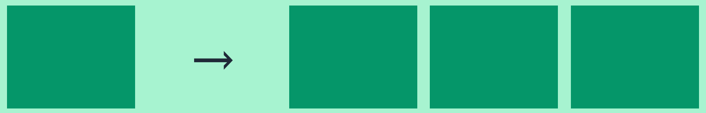
Vertical Scaling

"Bigger machine"



Horizontal Scaling

"More machines"



Key Concepts:

Auto-scaling
Add/remove resources based on demand

Load Balancing
Distribute traffic across servers

Caching
Store frequently accessed data closer

CDN
Serve content from edge locations

Design Question:
"What happens when traffic increases 10x?"

Cost Awareness

"Cloud is pay-as-you-go — but costs can surprise you"

Cost Drivers:

Compute	Running time, instance size
Storage	Amount stored, access frequency
Network	Data transfer out, between regions
Services	API calls, managed service fees

Optimization Strategies:

Right-sizing	Don't over-provision resources
Reserved	Commit for 1-3 years for discounts
Spot/Preemptible	Use spare capacity cheaply
Auto-shutdown	Turn off dev/test when not in use

Key Habit:

Set budgets and alerts from day one

Operations & Observability

"You can't fix what you can't see"

Three Pillars of Observability:

Logs

What happened

Error messages, events

Metrics

How it's performing

CPU, memory, latency

Traces

Request journey

End-to-end flow

Operational Practices:

Monitoring

Watch for problems

Alerting

Get notified when things break

Dashboards

Visualize system health

Runbooks

Document how to respond

DevOps Mindset:

"Build it, run it, own it"

Lab 01 Create a Linux Virtual Machine

1. **Go to Virtual Machines**
   Click + Create → Azure virtual machine
2. **Configure Basics**
   Set RG, name, region, image, size
3. **Configure Networking**
   Select VNet and Subnet
4. **Review + Create**
   Wait 2-3 minutes for deployment
5. **Connect via SSH**
   ssh studentXX@<public-ip>
6. **Clean Up**
   DELETE VM when done!

Basics Settings

Resource group:	`workshop-students-rg`
VM name:	`studentXX-vm`
Region:	`East US`
Image:	`Ubuntu 24.04 LTS`
Size:	`Standard_B1s`

Networking Settings

Virtual network:	`vnet-mlct-student-eus`
Subnet:	`snet-vm-student-eus`
NIC NSG:	`None`

⚠ **DELETE your VM after the lab to avoid charges!**

⏱ Time: 15 minutes

📄 Full guide: labs/02-create-vm.md

Day 1 Wrap-Up

Key takeaways

Section 5

Developer Workflow

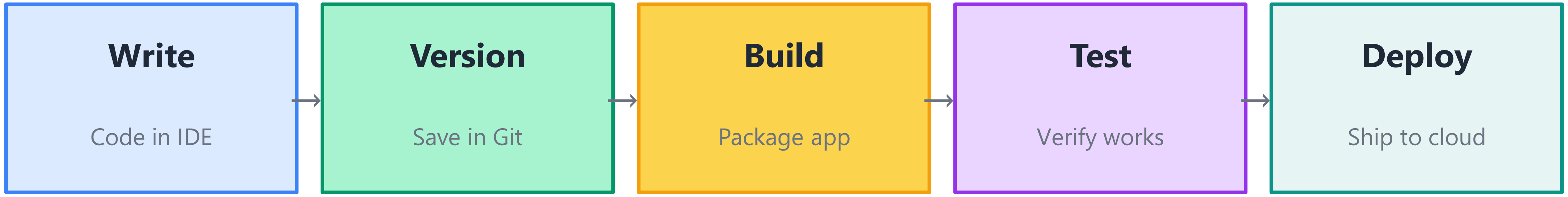
Developer tools and flow

Developer Workflow

What is Developer Workflow?

The process and tools developers use to write, test, and deliver code from their local machine to production.

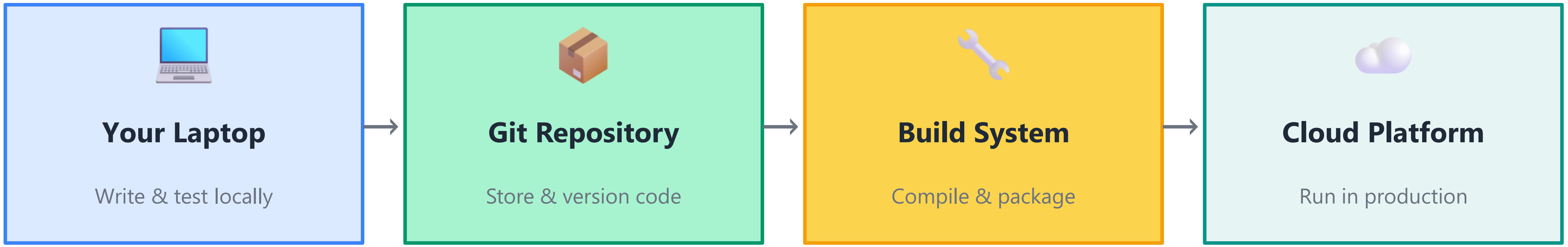
Key Stages:



A good workflow = faster development + fewer errors + easier collaboration

From Laptop to Cloud

The Journey Your Code Takes



Local vs Cloud:

Local Development

- Fast iteration
- Easy debugging
- Only you can access

Cloud Deployment

- Scalable & reliable
- Always available
- Anyone can access

Role of the IDE

IDE = Integrated Development Environment

Your "home base" for writing code — combines multiple tools in one place.

What an IDE Provides:

Code Editor

Syntax highlighting, autocomplete

Debugger

Find and fix errors

Terminal

Run commands

Git Integration

Version control built-in

Popular IDEs: VS Code (free) • Visual Studio • JetBrains (IntelliJ, PyCharm) • Eclipse

Git as the Source of Truth

Git = Version Control System

Tracks every change to your code. Like "save points" in a video game — you can always go back.

Why Git Matters:

● **History:**	See every change ever made
● **Collaboration:**	Multiple developers, same codebase
● **Branching:**	Work on features without breaking main code
● **Backup:**	Code is safe even if laptop dies

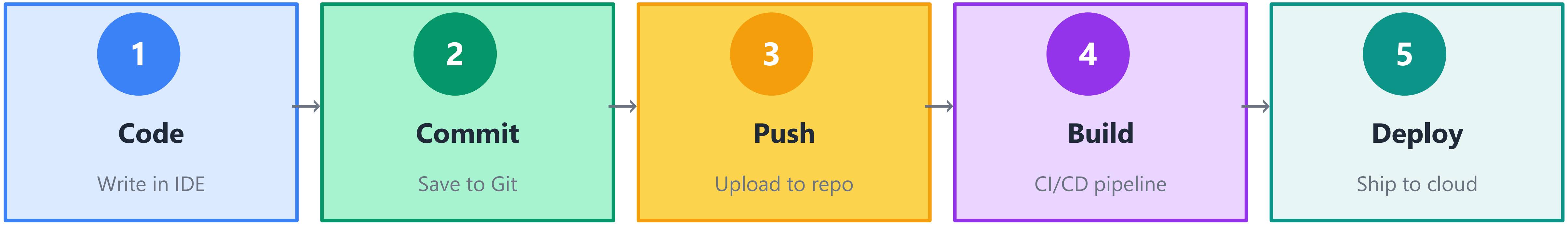
Git Platforms:

- GitHub
- GitLab
- Azure DevOps
- Bitbucket

"If it's not in Git, it doesn't exist" — The single source of truth for your code

How Code Moves Through the Workflow

The Development Flow:



Common Git Commands:

```
git add .            # Stage your changes
git commit -m "msg" # Save with a message
git push             # Upload to remote repo
```

Section 6

Containers

Packaging applications consistently

Containers

What is a Container?

A lightweight, standalone package that includes everything needed to run your application: code, runtime, libraries, and settings.

Analogy: Shipping Container

Just like shipping containers standardized global trade (any ship, any port), software containers standardize deployment (any server, any cloud).

What's Inside a Container:

Your Code	**Runtime**	**Libraries**	**Config**

"Works on my machine" → "Works on ANY machine"

Why Containers Exist

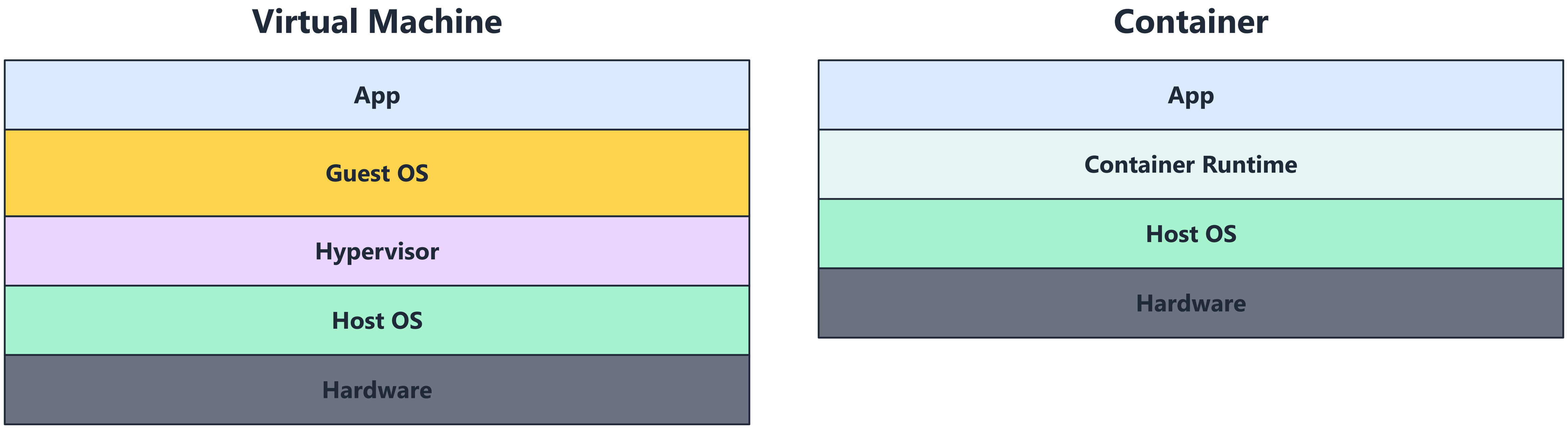
The Problem (Before Containers):

- ✗ "It works on my machine!" — but fails in production
- ✗ Different versions of libraries on dev vs prod
- ✗ Setup takes hours/days for new developers
- ✗ Hard to replicate production environment locally

The Solution (With Containers):

- ✓ Same container runs everywhere — laptop, test, production
- ✓ All dependencies bundled inside the container
- ✓ New developer? Just run the container — minutes, not days
- ✓ Production environment runs locally too

Containers vs Virtual Machines



Comparison	VM	Container
Startup	Minutes	**Seconds**
Size	GBs	**MBs**
Isolation	Full OS	**Process-level**

Container Concepts:

Image

A blueprint/template for creating containers. Read-only.

Analogy: Like a recipe

Container

A running instance of an image. You can have many from one image.

Analogy: Like the dish you cook

Registry

A storage location for images. Like GitHub for containers.

Analogy: Like a recipe book library

Docker

The most popular tool for building and running containers.

Analogy: Like the kitchen

Popular Registries: Docker Hub • Azure Container Registry • Google Container Registry • Amazon ECR

Dockerfile Basics

Dockerfile = Instructions to build an image

A text file that tells Docker how to create your container image, step by step.

Example Dockerfile:

```
FROM node:18                          # Start with Node.js base

WORKDIR /app                          # Set working directory

COPY package*.json ./                 # Copy dependency files

RUN npm install                       # Install dependencies

COPY . .                              # Copy app code

EXPOSE 3000                           # Declare port

CMD ["npm", "start"]                  # Run command
```

Key Instructions:

FROM	Base image to start from
WORKDIR	Set the working directory
COPY	Copy files into image
RUN	Execute a command
EXPOSE	Document the port
CMD	Default command to run

```
Build: docker build -t myapp .      Run: docker run -p 3000:3000 myapp
```

Section 7

DevOps & CI/CD

Automating delivery

DevOps & CI/CD

The Big Picture

DevOps and CI/CD are practices and tools that help teams deliver software faster, more reliably, and with fewer errors.

DevOps

A culture and set of practices that brings development (Dev) and operations (Ops) teams together.

Focus: Collaboration & Communication

CI/CD

Continuous Integration and Continuous Deployment — automating the build, test, and deployment process.

Focus: Automation & Speed

Goal: Ship code faster with confidence — from commit to production in minutes, not weeks

What Is DevOps?

DevOps = Development + Operations

A culture shift that breaks down silos between teams who write code and teams who deploy/manage it.

Traditional vs DevOps:

Traditional (Siloed)	DevOps (Collaborative)
• Dev builds, "throws over the wall" to Ops	• Dev and Ops work together
• Ops deploys, blames Dev when it breaks	• Shared responsibility for uptime
• Long release cycles (months)	• Fast release cycles (daily/weekly)
• Manual processes	• Automated processes

Key DevOps Practices:

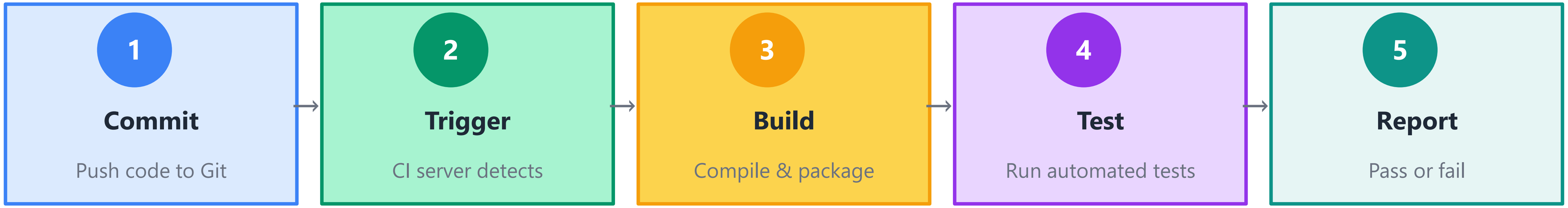
CI/CD	IaC	Monitoring	Feedback
Automate build & deploy	Infrastructure as Code	Observe everything	Fast iteration loops

Continuous Integration (CI)

CI = Automatically build and test code every time a developer commits

Catch bugs early by integrating code changes frequently — multiple times per day.

How CI Works:



Benefits of CI:

✓ Find bugs early ✓ Faster feedback ✓ Always have working code ✓ Reduce integration problems

Continuous Deployment (CD)

CD = Automatically deploy code to production after it passes all tests

Every commit that passes the pipeline goes straight to production — no manual steps.

Two Flavors of CD:

Continuous Delivery

Code is always ready to deploy, but deployment requires manual approval.

→ "Push button" deployment

Continuous Deployment

Code automatically deploys to production after passing tests. No human intervention.

→ Fully automated

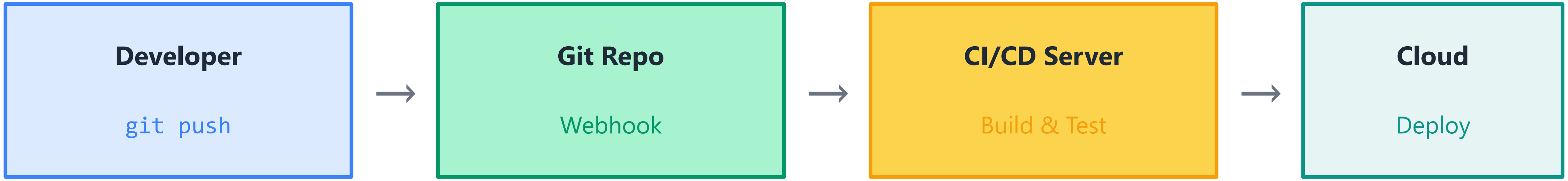
Full CI/CD Pipeline:

Commit → Build → Test → Stage → Deploy → Monitor

CI/CD Pipelines Triggered by Git

Git Push = Pipeline Trigger

When you push code to Git, the CI/CD pipeline automatically starts — no manual intervention needed.



Popular CI/CD Tools:

GitHub Actions	Built into GitHub
Azure DevOps	Microsoft ecosystem
GitLab CI	Built into GitLab
Jenkins	Self-hosted, flexible

The pipeline is defined in code (YAML) — version controlled, just like your app

Section 8

Infrastructure as Code

Infrastructure through code

Infrastructure as Code

What is Infrastructure as Code (IaC)?

Managing and provisioning infrastructure (servers, networks, databases) through code files instead of manual processes.

Analogy: Infrastructure as Code is like a recipe for your cloud environment

Write it once, and you can recreate the exact same environment anywhere, anytime.

Manual vs IaC:

Manual: Click through UI

Slow, error-prone, hard to replicate

IaC: Write code

Fast, consistent, repeatable

Popular IaC Tools: Terraform • AWS CloudFormation • Azure Bicep • Pulumi

Why Infrastructure as Code?

Problems with Manual Infrastructure:

- ✗ "Snowflake" servers — each one is unique, hard to reproduce
- ✗ Configuration drift — environments slowly become different
- ✗ No history — who changed what, and when?
- ✗ Slow setup — manual steps take hours/days

Benefits of IaC:

Consistency	Version Control	Speed	Reusable
Same infra every time	Track all changes in Git	Deploy in minutes	Use same code for dev/prod

Declarative Infrastructure

Declarative = Describe WHAT you want, not HOW to build it

You define the desired end state, and the tool figures out how to get there.

Imperative vs Declarative:

Imperative (How)

Step-by-step instructions:
1. Create VM
2. Install Node.js
3. Configure networking
4. Deploy app

Declarative (What)

Desired state:
"I want a VM with Node.js,
connected to a network,
running my app"

Terraform uses declarative approach — you describe the end state, Terraform figures out the steps

Terraform Core Concepts

Provider

Plugin that connects to a cloud (AWS, Azure, GCP). Tells Terraform how to talk to the cloud.

Resource

A single piece of infrastructure (VM, database, network). The building blocks.

State

Terraform's record of what exists. Tracks what it created so it knows what to change.

Plan

Preview of changes before applying. Shows what will be created, changed, or destroyed.

Basic Workflow:

```
terraform init  →  terraform plan  →  terraform apply  →  terraform destroy
```

Infrastructure Lifecycle

The Terraform Lifecycle:

1 **Write** — Define infrastructure in .tf files →
2 **Init** — Download providers →
3 **Plan** — Preview changes →
4 **Apply** — Create/update infra →
5 **Manage** — Update or destroy

State Management:

Terraform keeps track of what it created in a state file (terraform.tfstate)

• Local state — stored on your machine (simple but not for teams)
• Remote state — stored in cloud (Azure Blob, S3) for team collaboration

IaC + Git + CI/CD = Fully automated, version-controlled infrastructure

Section 9

End-to-End Cloud Flow

Putting it all together

End-to-End Cloud Flow

Bringing Everything Together

Now that we've covered the individual pieces, let's see how they all connect in a real cloud deployment.

What We've Covered:

Foundations
Cloud basics

Building Blocks
Compute, Data, Network

Architecture
Patterns & Principles

Dev Workflow
IDE, Git

Containers
Docker

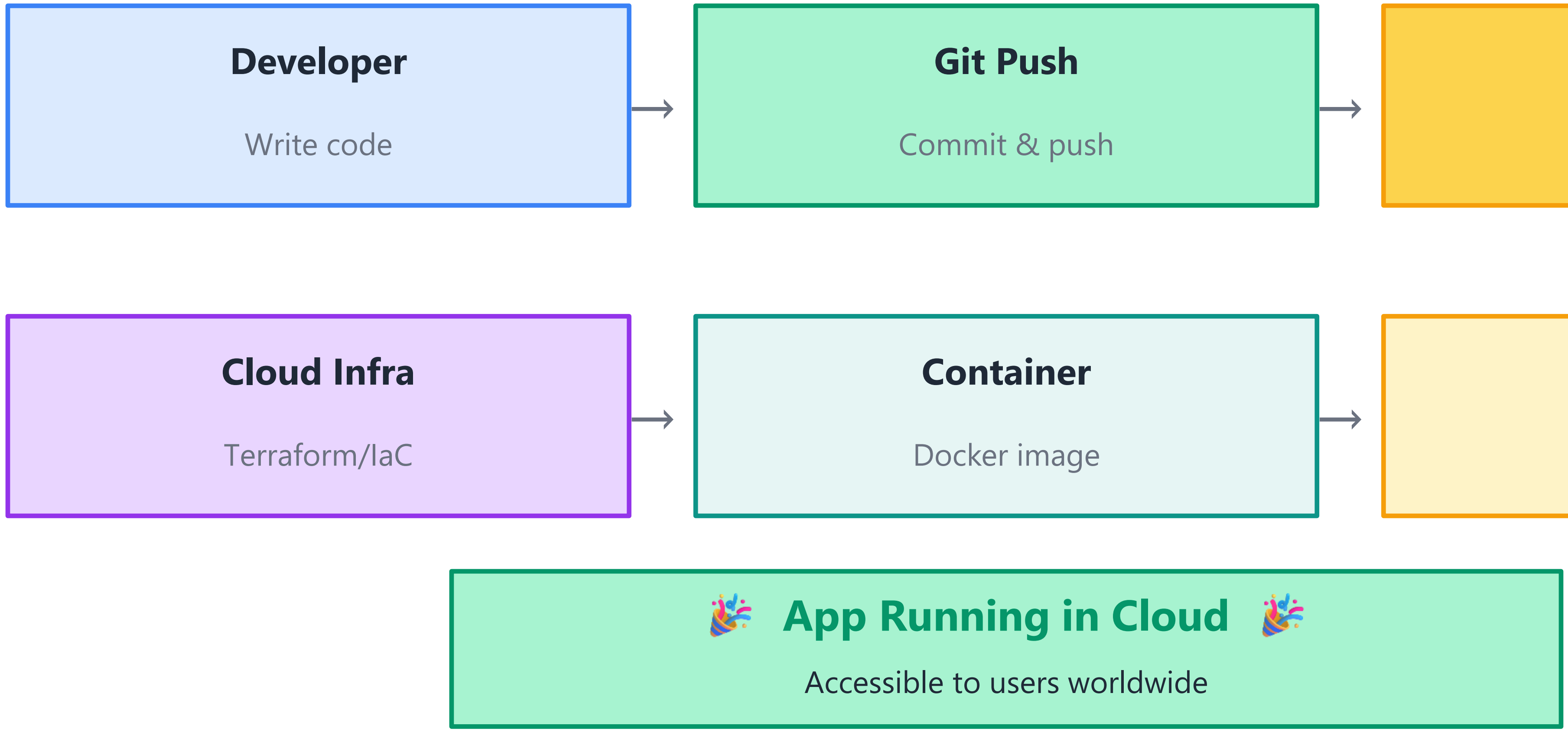
DevOps
CI/CD

IaC
Terraform

Now let's see how these pieces work together in a real deployment flow

End-to-End Cloud Application Flow

From Code to Cloud — The Complete Journey:



Mapping Concepts to Real Systems

How Workshop Concepts Translate to Real Tools:

Concept	Azure	GCP
Compute (VMs)	Virtual Machines	Compute Engine
Compute (Containers)	Azure Container Apps	Cloud Run
Compute (Serverless)	Azure Functions	Cloud Functions
Database (SQL)	Azure SQL	Cloud SQL
Database (NoSQL)	Cosmos DB	Firestore
Storage (Files)	Blob Storage	Cloud Storage
CI/CD	Azure DevOps	Cloud Build
Container Registry	ACR	Artifact Registry

Day 2 Lab

From Code to Cloud-Ready

??????????

Day 2 Wrap-Up

What comes next?