Software Architectures

A Practical Guide to Designing Scalable Systems with Microservices, APIs, and Cloud-Native Technologies

Bîcu Andrei Ovidiu

Table of contents

Chapter 1: What Is Software Architecture and Why Does It Matter

What is Software Architecture?

Software architecture is the *high-level* structure of a software system. It defines the **components, their responsibilities and the patterns** of communication between them.

Architecture includes:

- Design decisions that are **hard to change later**.
- Considerations for **performance, security,** and **scalability.**
- How systems will evolve, scale and integrate over time.

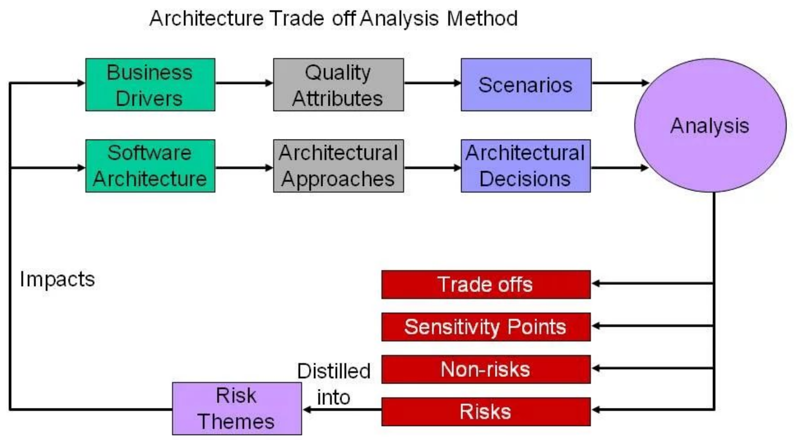
Why is Architecture crucial?

1. **Scalable:** can handle increasing users or data with minimal refactoring.
2. **Maintainable:** easy to modify and extend
3. **Reliable:** faults in one part don't crash the whole system.
4. **Secure:** enforces authentication, encryption, and safe data flow.
5. **Performant:** optimized for response times and throughput.
6. **Deployable:** can be deployed frequently with confidence.

Architecture is a Set of Trade-Offs

Every decision in architecture involves balancing:

- **Speed vs Maintainability -** Fast hacks vs clean modular design.
- **Simplicity vs Flexibility -** One-size-fits-all vs adaptable services
- **Coupling vs Autonomy -** Monolith vs Microservices
- **Cost vs Scalability** - Shared hosting vs serverless cloud



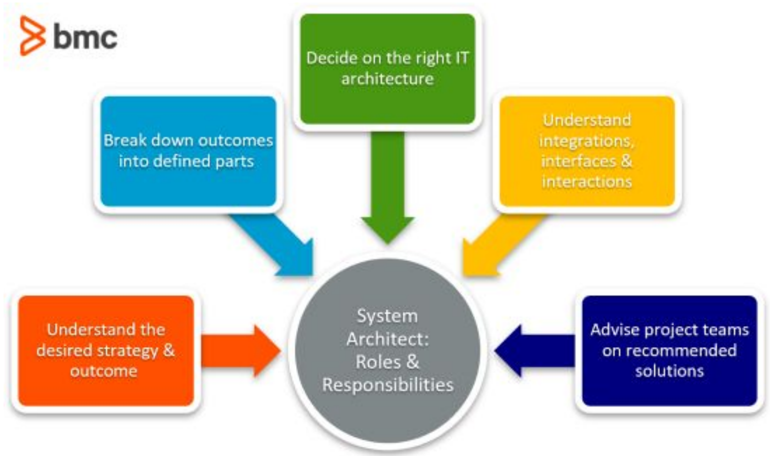
Key Responsibilities of a Software Architect & Real-World Analogy

Key Responsibilities:

- **Define system structure:** Components, services, data flows.
- **Choose technologies:** Frameworks, languages, and infrastructure.
- **Align with business needs:** Ensure tech supports business goals
- **Document decisions:** Use architecture decision records (ADRs)
- **Communicate:** Help devs, product, and ops align around structure

Real-World Analogy:

Imagine you're building a restaurant where the **code** is the <u>*recipes*</u>, the **design** is the <u>*kitchen layout*</u> and the <u>**architecture**</u> is the <u>whole restaurant's layout</u>: where guests sit, how food flows from the kitchen to the table, and where emergency exits are.


bmc
Decide on the right IT architecture
Break down outcomes into defined parts
Understand integrations, interfaces & interactions
Understand the desired strategy & outcome
System Architect: Roles & Responsibilities
Advise project teams on recommended solutions

Chapter 2: Architectural Styles - Monoliths, Microservices and Beyond.

1. Monolithic Architecture

A ***monolithic architecture*** *is a traditional way of structuring software as a single, unified unit.*

Characteristics:

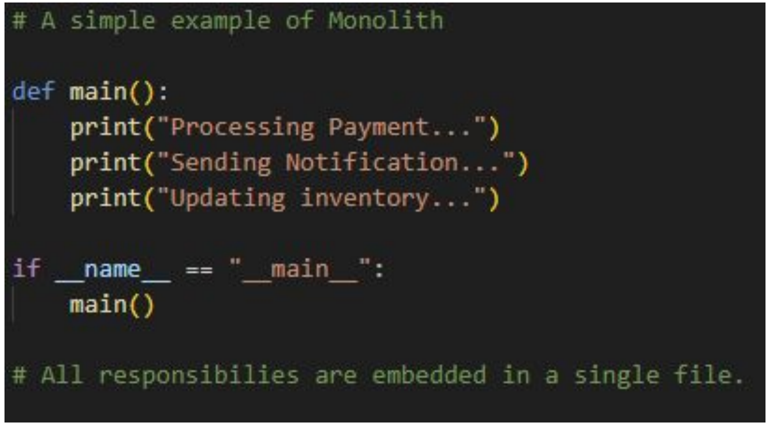
- All features bundled into one codebase
- Single deployment unit (often a .py, .jar or .war)
- Shared memory and resources

Advantages:

- Easy to develop at the start
- Simpler to test and deploy
- No need for complex infrastructure

Disadvantages:

- Hard to scale individual parts
- A small change can require redeploying the whole system
- Difficult to adopt new tech stacks or patterns incrementally



```python
# A simple example of Monolith

def main():
    print("Processing Payment...")
    print("Sending Notification...")
    print("Updating inventory...")

if __name__ == "__main__":
    main()

# All responsibilies are embedded in a single file.
```

2. Microservices Architecture

***Microservices** divide an application into small, independent services aligned to business capabilities.*

Characteristics:

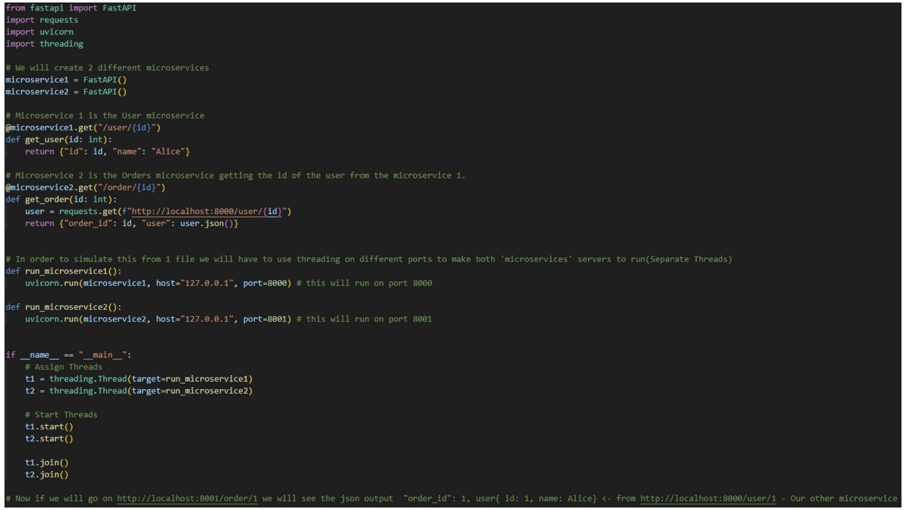
- Each service is independently deployable
- Services communicate via APIs
- Teams can own services end-to-end

Advantages:

- Scalability: Scale only the needed services
- Flexibility: Use different languages/tech per service
- Resilience: Failures are isolated

Disadvantages:

- Increased complexity in communication and deployment
- Requires DevOps maturity (monitoring, CI/CD, discovery)
- Debugging across services can be tricky



```python
from fastapi import FastAPI
import requests
import uvicorn
import threading

# We will create 2 different microservices
microservice1 = FastAPI()
microservice2 = FastAPI()

# Microservice 1 is the User microservice
@microservice1.get("/user/{id}")
def get_user(id: int):
    return {"id": id, "name": "Alice"}

# Microservice 2 is the Orders microservice getting the id of the user from the microservice 1.
@microservice2.get("/order/{id}")
def get_order(id: int):
    user = requests.get(f"http://localhost:8000/user/{id}")
    return {"order_id": id, "user": user.json()}


# In order to simulate this from 1 file we will have to use threading on different ports to make both 'microservices' servers to run(Separate Threads)
def run_microservice1():
    uvicorn.run(microservice1, host="127.0.0.1", port=8000) # this will run on port 8000

def run_microservice2():
    uvicorn.run(microservice2, host="127.0.0.1", port=8001) # this will run on port 8001


if __name__ == "__main__":
    # Assign Threads
    t1 = threading.Thread(target=run_microservice1)
    t2 = threading.Thread(target=run_microservice2)

    # Start Threads
    t1.start()
    t2.start()

    t1.join()
    t2.join()

# Now if we will go on http://localhost:8001/order/1 we will see the json output  "order_id": 1, user{ id: 1, name: Alice} <- from http://localhost:8000/user/1 - Our other microservice
```

3. Modular Monolith

*A **modular monolith** is a hybrid: it's a single deployment but internally organized into well-separated modules with clear boundaries.*

Characteristics:

- Single deployable application (e.g. one FastAPI app)
- Internal separation into independent modules with clear boundaries
- All modules share the same process/runtime and call each other directly.

Advantages:

- Easy to deploy and debug
- Faster than microservices (no inter-service latency)
- Simpler deployment (one artifact, one process)
- Easier to transition into microservices later

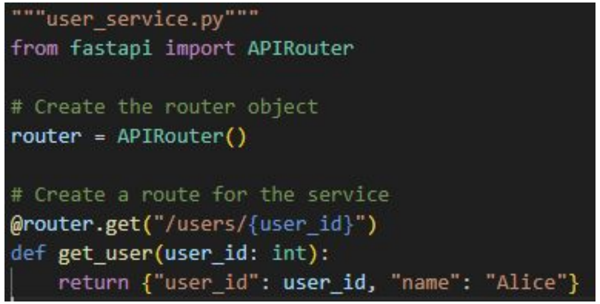
Disadvantages:

- Still has a single point of failure.
- Deployment is all-or-nothing
- A shared codebase can lead to tightly coupled modules without discipline.
- Doesn't scale as independently as microservices

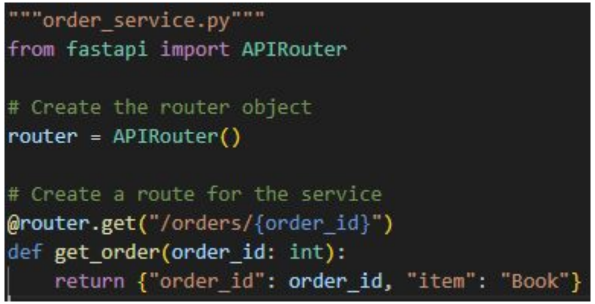
Think of it as "microservices within a monolith".

***Best for:** Startups, teams migrating to microservices gradually, projects needing modularity but not full distribution.*

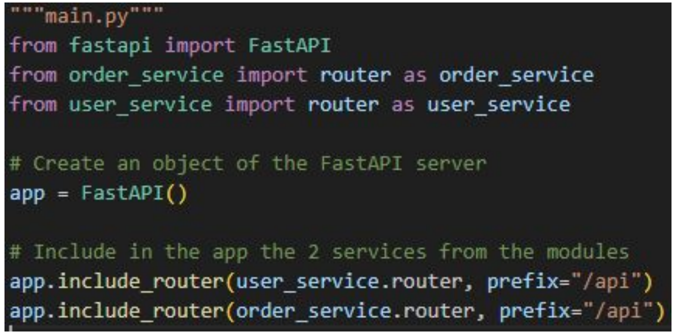
Bîcu Andrei Ovidiu



```python
"""user_service.py"""
from fastapi import APIRouter

# Create the router object
router = APIRouter()

# Create a route for the service
@router.get("/users/{user_id}")
def get_user(user_id: int):
    return {"user_id": user_id, "name": "Alice"}
```



```python
"""order_service.py"""
from fastapi import APIRouter

# Create the router object
router = APIRouter()

# Create a route for the service
@router.get("/orders/{order_id}")
def get_order(order_id: int):
    return {"order_id": order_id, "item": "Book"}
```



```python
"""main.py"""
from fastapi import FastAPI
from order_service import router as order_service
from user_service import router as user_service

# Create an object of the FastAPI server
app = FastAPI()

# Include in the app the 2 services from the modules
app.include_router(user_service.router, prefix="/api")
app.include_router(order_service.router, prefix="/api")
```

4. Event-Driven Architecture

*An **event-driven architecture (EDA)** reacts to system events asynchronously.*

Key Concepts:

- Components communicate asynchronously via events (e.g., “UserRegistered”)
- Publishers emit events; subscribers react to them
- Uses message brokers (RabbitMQ, Kafka, Redis Streams)
- Services are loosely coupled and unaware of who consumes their events

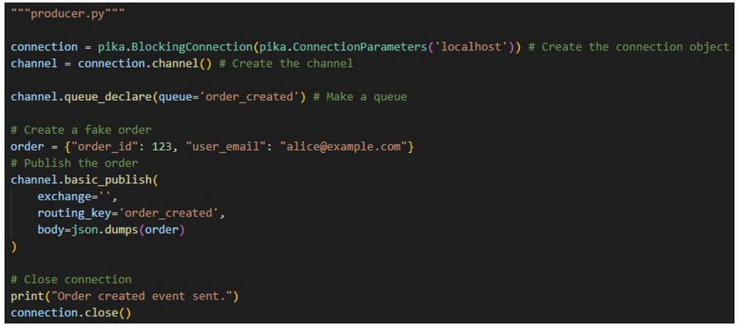
Advantages:

- Highly decoupled: systems don’t depend on each other’s availability
- Great for scalability and resilience
- Enables real-time behavior (e.g., live dashboards, alerts)
- Natural fit for domain-driven design and bounded contexts

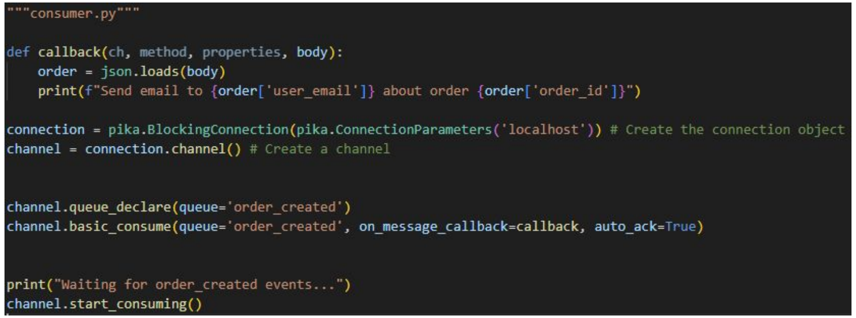
Disadvantages:

- Harder to trace and debug (no clear request flow)
- Requires careful event schema management
- Potential for message loss or duplication without idempotency
- Operational overhead: managing brokers, retries, and dead-letter queues.

Best for: Systems with many independent services, real-time analytics, and high throughput processing.



```python
"""producer.py"""

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost')) # Create the connection object
channel = connection.channel() # Create the channel

channel.queue_declare(queue='order_created') # Make a queue

# Create a fake order
order = {"order_id": 123, "user_email": "alice@example.com"}
# Publish the order
channel.basic_publish(
    exchange='',
    routing_key='order_created',
    body=json.dumps(order)
)

# Close connection
print("Order created event sent.")
connection.close()
```



```python
"""consumer.py"""

def callback(ch, method, properties, body):
    order = json.loads(body)
    print(f"Send email to {order['user_email']} about order {order['order_id']}")

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost')) # Create the connection object
channel = connection.channel() # Create a channel


channel.queue_declare(queue='order_created')
channel.basic_consume(queue='order_created', on_message_callback=callback, auto_ack=True)


print("Waiting for order_created events...")
channel.start_consuming()
```

5. Serverless Architecture

***Serverless** doesn't mean "no servers"; it means you don't manage them.*

Characteristics:

- Code is run as short-lived functions on demand
- Functions are triggered by events (HTTP, queue messages, file uploads, etc.)
- No server or container management required by developers
- Typically deployed on platforms like AWS Lambda, Azure Functions, or Google Cloud Functions

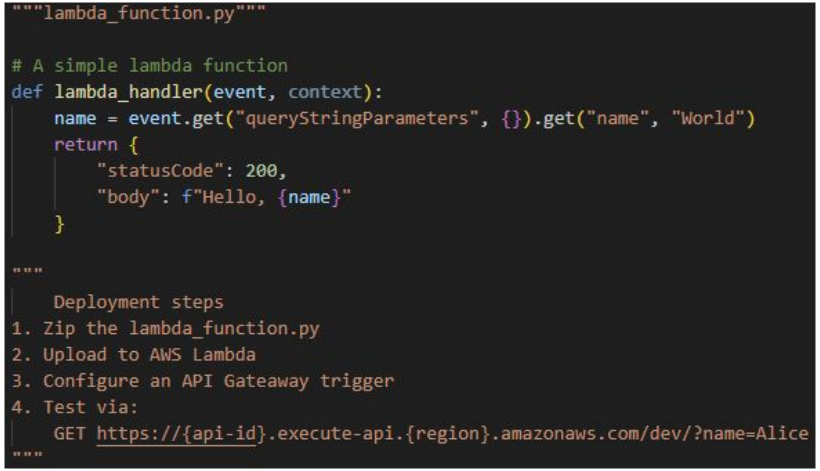
Advantages:

- No infrastructure to manage; focus on code
- Automatic scaling, from 0 to thousands of requests
- Code-effective; pay only for execution time
- Ideal for sporadic workloads, APIs, and automation tasks

Disadvantages:

- Cold starts may cause delays (esp. in high-latency runtimes like Python)
- Stateless; persistent data must be stored externally (e.g., S3, DB)
- Limited execution time (e.g., AWS Lambda: 15 min max)
- Vendor lock-in & less control over runtime

Best for: APIs, background jobs, IoT, automation scripts, and when you want minimal DevOps.



```python
"""lambda_function.py"""

# A simple lambda function
def lambda_handler(event, context):
    name = event.get("queryStringParameters", {}).get("name", "World")
    return {
        "statusCode": 200,
        "body": f"Hello, {name}"
    }

"""
    Deployment steps
1. Zip the lambda_function.py
2. Upload to AWS Lambda
3. Configure an API Gateway trigger
4. Test via:
    GET https://{api-id}.execute-api.{region}.amazonaws.com/dev/?name=Alice
"""
```

Bîcu Andrei Ovidiu

Chapter 3: Designing Microservices That Actually Work (with Python)

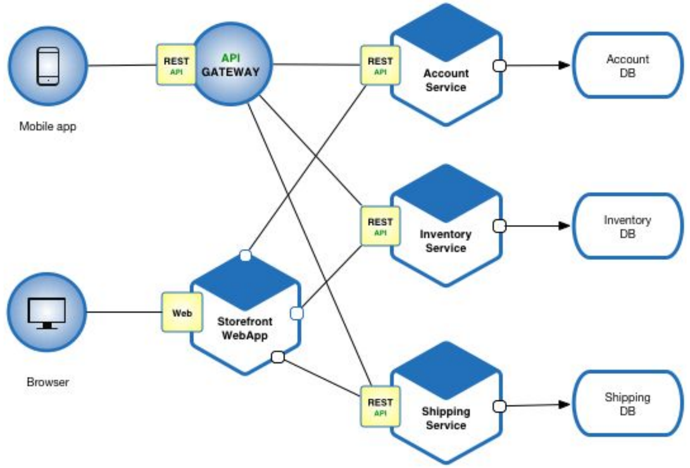
Characteristics of Good Microservices

- **Business-aligned**: Built around real-world business functions
- **Autonomous**: Deployable and scalable independently
- **Loosely coupled**: Doesn't rely heavily on other services' internals
- **Technology-agnostic**: Can use different stacks or Databases if needed
- **Observable**: Easy to monitor, trade and debug

Thinking in Domains, not Layers:

Traditional apps are organized into layers (UI, business logic, DB). Microservices flip that on its head:

*Each service contains **everything it needs**: routes, logic, data access and even its own database.*

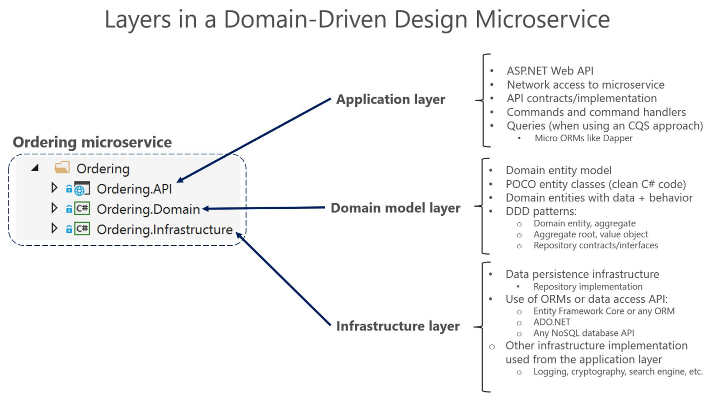


Domain-Driven Design (DDD) Basics

To define effective microservices, apply DDD principles:

- **Bounded context**: Each service is responsible for a specific business domain (e.g., Orders, Users, Payments)
- **Ubiquitous Language**: Use business terms consistently in code, APIs and discussions.

*Good microservice boundaries reflect **how the business works,** not just how data is stored.*



- ASP.NET Web API
- Network access to microservice
- API contracts/implementation
- Commands and command handlers
- Queries (when using an CQS approach)
  - Micro ORMs like Dapper

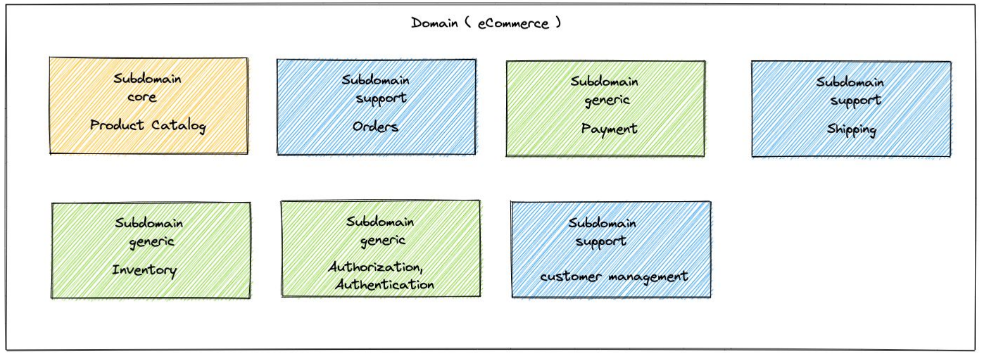
- Domain entity model
- POCO entity classes (clean C# code)
- Domain entities with data + behavior
- DDD patterns:
  - Domain entity, aggregate
  - Aggregate root, value object
  - Repository contracts/interfaces

- Data persistence infrastructure
  - Repository implementation
- Use of ORMs or data access API:
  - Entity Framework Core or any ORM
  - ADO.NET
  - Any NoSQL database API
- Other infrastructure implementation used from the application layer
  - Logging, cryptography, search engine, etc.

Example Domains for an E-Commerce Platform

1. **User Service –** Handles registration, login, profiles
2. **Order Service –** Manages order placement and history
3. **Product Service –** Manages catalog, inventory, pricing
4. **Payment Service –** Handles payment processing
5. **Notification Service –** Sends emails/SMS updates

Each of these would be an ***independent Python FastAPI service*** *with its own, repo, Dockerfile and DB.*


Domain (eCommerce)
Subdomain core Product Catalog
Subdomain support Orders
Subdomain generic Payment
Subdomain support Shipping
Subdomain generic Inventory
Subdomain generic Authorization, Authentication
Subdomain support customer management

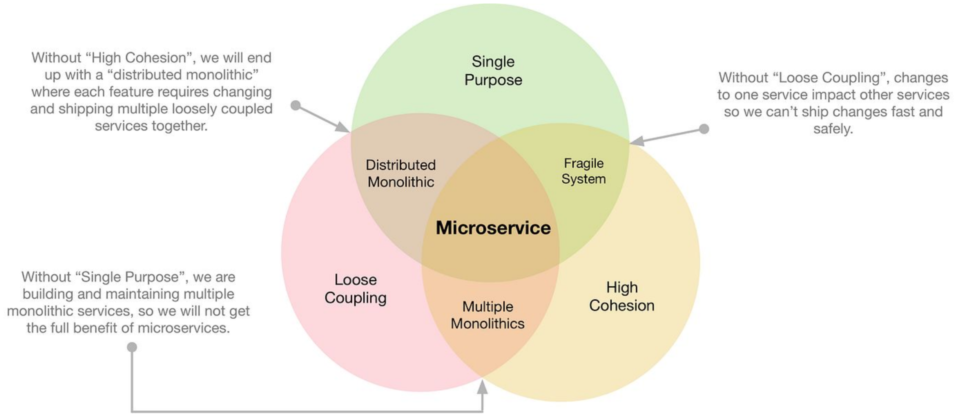
Principles of Designing Great Microservices & Common Mistakes

DO:

- **SRP (Single Responsibility Principle) -** each service should do one thing well
- **Own your data** - No shared DBs! Each service manages its own storage
- **API contracts are sacred -** Never break them. Use versioning (/v1/users) if needed
- **Fail fast and gracefully -** Use timeouts, retries and circuit breakers
- **Statelessness where possible -** Don't depend on local in-memory state

DON'T:

- Calling other services internal functions or databases
- Overcomplicating by splitting everything too soon
- Using microservices without DevOps and CI/CD pipelines
- Forgetting monitoring, logging and tracing


Without "High Cohesion", we will end up with a "distributed monolithic" where each feature requires changing and shipping multiple loosely coupled services together.
Single Purpose
Without "Loose Coupling", changes to one service impact other services so we can't ship changes fast and safely.
Distributed Monolithic
Fragile System
Microservice
Without "Single Purpose", we are building and maintaining multiple monolithic services, so we will not get the full benefit of microservices.
Loose Coupling
Multiple Monolithics
High Cohesion

Chapter 4: Communication Between Microservices

Why Communication Matters?

*In microservices, your code spends **as much time talking to other services as it does executing logic.***

Unlike monoliths, where functions call each other in-process, microservices require **network-based communication,** which introduces:

- **Latency**
- **Failure points**
- **Serialization overhead**
- **Security concerns**

Model	***Type***	***Characteristics***	***Use Case***
REST	Synchronous	HTTP-based, stateless	CRUD, APIs, simple queries
GraphQL	Synchronous	Flexible client queries over HTTP	Dynamic frontends, partial data
gRPC	Synchronous	High-Performance binary RPC protocol	Low-latency, high-throughput
Message Queue	Asynchronous	Event-driven, decoupled, eventual delivery	Notifications, background jobs

1. REST (Representational State Transfer)

Characteristics:

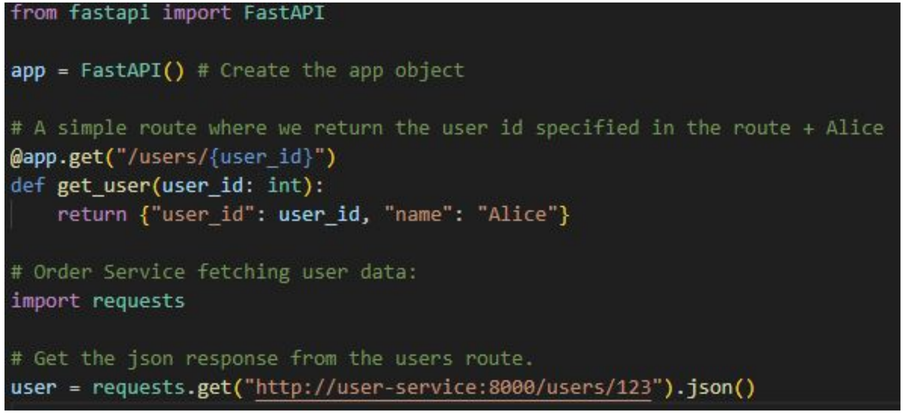
- Communicate using HTTP
- JSON over HTTP (GET, POST, PUT, DELETE)
- Stateless and language agnostic

Pros:

- Easy to test and debug
- Human readable
- Widely Supported

Cons:

- Limited to request-response
- Over-fetching or under-fetching of data



```python
from fastapi import FastAPI

app = FastAPI() # Create the app object

# A simple route where we return the user id specified in the route + Alice
@app.get("/users/{user_id}")
def get_user(user_id: int):
    return {"user_id": user_id, "name": "Alice"}

# Order Service fetching user data:
import requests

# Get the json response from the users route.
user = requests.get("http://user-service:8000/users/123").json()
```

2. GraphQL (Flexible Query Language)

Characteristics:

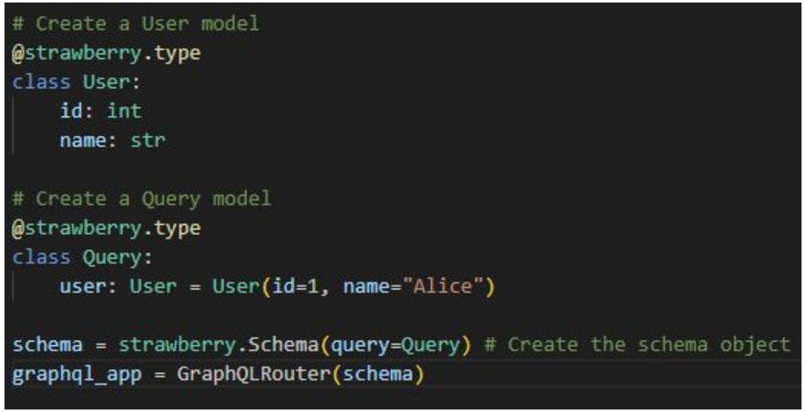
- Query exactly what you need
- Single endpoint, complex responses
- Strong schema via SDL

Pros:

- Avoids over-/under-fetching
- Powerful for front-end-driven development

Cons:

- Steeper learning curve
- Performance depends on query complexity



```python
# Create a User model
@strawberry.type
class User:
    id: int
    name: str

# Create a Query model
@strawberry.type
class Query:
    user: User = User(id=1, name="Alice")

schema = strawberry.Schema(query=Query) # Create the schema object
graphql_app = GraphQLRouter(schema)
```

3. gRPC (Google Remote Procedure Call)

Characteristics:

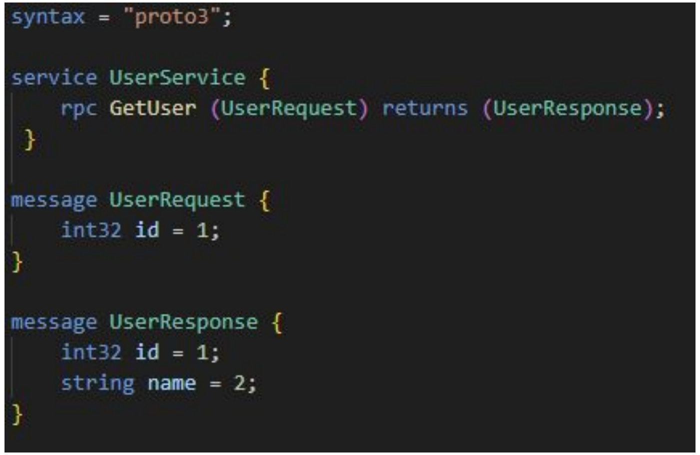
- Uses Protocol Buffers (protobuf) for data format
- Strongly typed, binary serialization
- Supports streaming and bi-directional comms

Pros:

- Fast, compact, efficient
- Ideal for internal service-to-service calls

Cons:

- Harder to debug (not human readable)
- Browser support requires extra tooling (e.g., gRPC-Web)



```
syntax = "proto3";

service UserService {
    rpc GetUser (UserRequest) returns (UserResponse);
 }

message UserRequest {
    int32 id = 1;
}

message UserResponse {
    int32 id = 1;
    string name = 2;
}
```

4. Message Queues (Asynchronous Communication)

Characteristics:

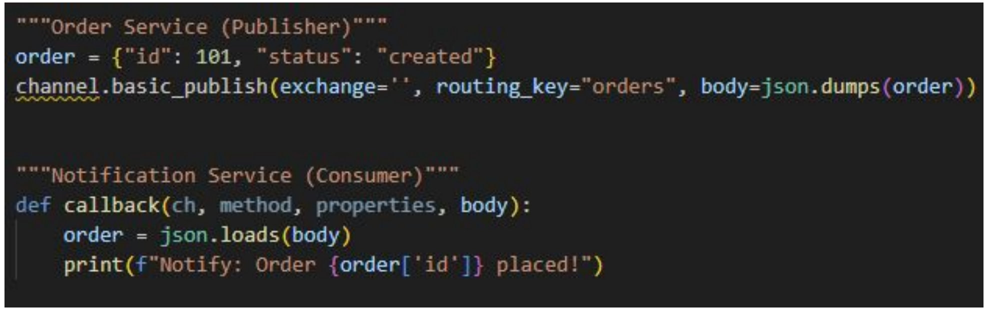
- Services **publish** and **subscribe** to events
- Queues buffer messages until the consumer processes them
- Enables loose coupling and non-blocking logic

Pros:

- High decoupling
- Excellent for workflows and async processing

Cons:

- Harder to trace the flow
- Eventual consistency may cause confusion



```python
"""Order Service (Publisher)"""
order = {"id": 101, "status": "created"}
channel.basic_publish(exchange='', routing_key="orders", body=json.dumps(order))


"""Notification Service (Consumer)"""
def callback(ch, method, properties, body):
    order = json.loads(body)
    print(f"Notify: Order {order['id']} placed!")
```

Bîcu Andrei Ovidiu

Tips for Microservices Communication

- Always set **timeouts** for outbound calls
- Use **circuit breakers** to avoid cascading failures
- Implement **retry logic** for transient errors
- Standardize error responses
- Add **tracing headers** (like OpenTelemetry) for observability

Choosing the Right Protocol

Requirement	***Recommended***
Simple web CRUD	**REST**
Frontend needs flexibility	**GraphQL**
Internal high-performance	**gRPC**
Decoupled workflows	**Message Queue (RabbitMQ, Kafka)**

Chapter 5: Tooling for Microservices – Containers, CI/CD, Monitoring & More

Key Tooling Categories

*Microservices aren't just about writing services; they're about running them **reliably, independently,** and **at scale**. To make that work, you need a robust toolchain.*

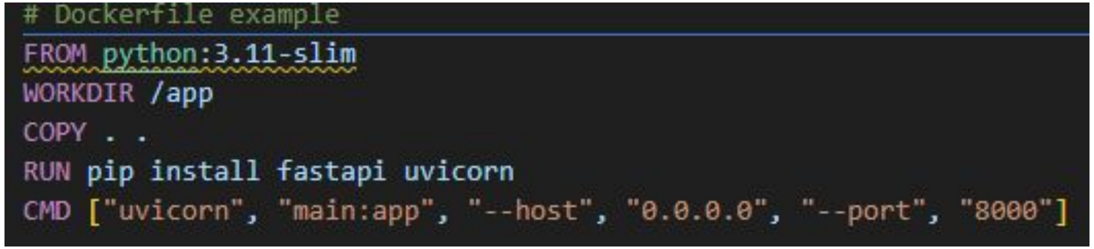
Category	***Purpose***	***Tools***
Packaging & Runtime	Isolate and run services	Docker, Kubernetes
Deployment Automation	Automate builds, tests, deployments	GitHub Actions, ArgoCD, CircleCI
Service Networking	Route traffic, expose APIs	API Gateway, NGINX, Traefik
Observability	Monitor, log and trace services	Prometheus, Grafana, OpenTelemetry
Configuration & Secrets	Manage sensitive settings	Env Vars, HashiCorp Vault, AWS Secrets Manager
Service Discovery	Dynamically locate services	Kubernetes DNS, Consul

1. Containers with Docker

Containers package your app with all its dependencies, so it runs ***the same everywhere.***

Benefits:

- Lightweight and fast to start
- Environment-independent
- Ideal for isolating microservices



```dockerfile
# Dockerfile example
FROM python:3.11-slim
WORKDIR /app
COPY . .
RUN pip install fastapi uvicorn
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Each service should have its ***own Dockerfile*** *and container image.*

Bîcu Andrei Ovidiu

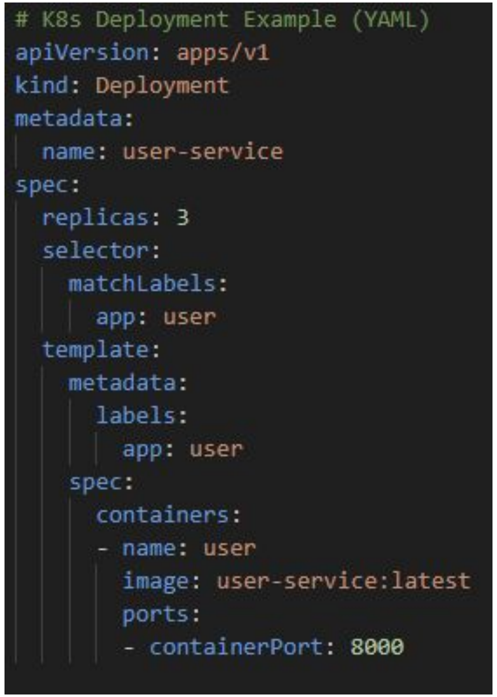
2. Orchestration with Kubernetes (K8s)

Kubernetes automates the deployment, scaling and management of containerized applications.

Benefits:

- Self-healing containers
- Auto-scaling
- Built-in **service discovery, ingress routing,** and **config maps**.

<u>*Deploy one microservice per pod or deployment group.*</u>



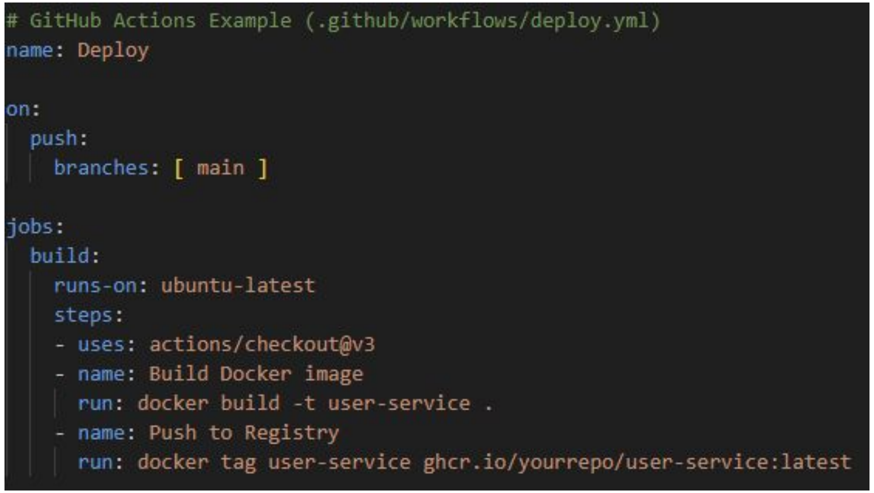
```yaml
# K8s Deployment Example (YAML)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: user-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: user
  template:
    metadata:
      labels:
        app: user
    spec:
      containers:
      - name: user
        image: user-service:latest
        ports:
        - containerPort: 8000
```

3. CI/CD with GitHub Actions

CI/CD pipelines ensure you can ship code quickly, consistently and safely.

Benefits:

- Automates testing, builds and deployment
- Encourages small, frequent releases
- Can include linting, security scans, Docker pushes



```yaml
# GitHub Actions Example (.github/workflows/deploy.yml)
name: Deploy

on:
  push:
    branches: [ main ]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - name: Build Docker image
      run: docker build -t user-service .
    - name: Push to Registry
      run: docker tag user-service ghcr.io/yourrepo/user-service:latest
```

4. API Gateway

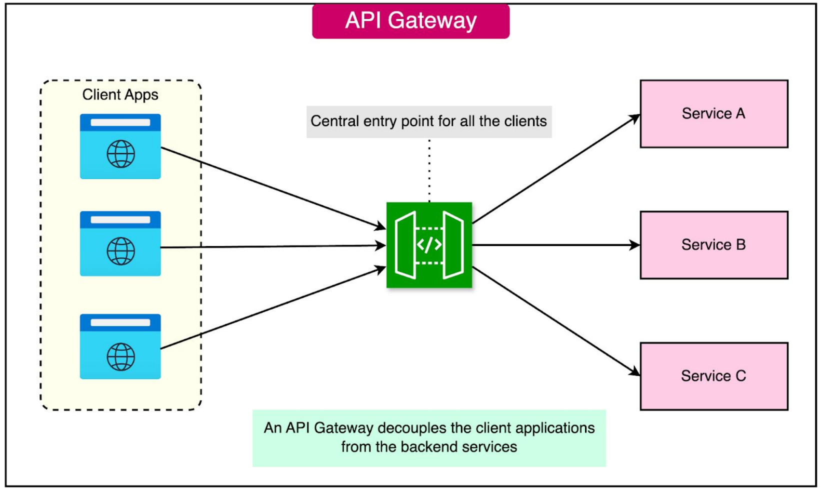
*An API Gateway acts as a **reverse proxy**, managing access to all microservices.*

Benefits:

- Centralized **authentication** and **rate limiting**
- Route traffic to appropriate backend service
- Handles CORS, logging, and response transformation

Examples:

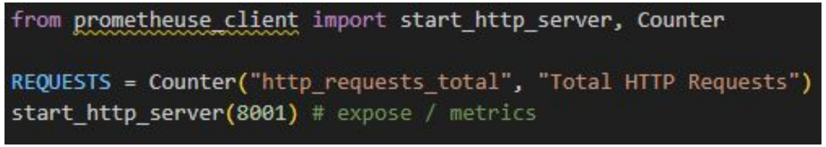
- **AWS API Gateway** (for serverless)
- **Kong** (open-source)
- **NGINX / Traefik** (self-hosted reverse proxy)


API Gateway
Client Apps
Central entry point for all the clients
Service A
Service B
Service C
An API Gateway decouples the client applications from the backend services

5. Observability Stack

You need full visibility into services to debug, monitor and optimize.

Layer	***Tool***	***Function***
Metrics	*Prometheus*	Collects time-series metrics
Logs	*Fluentd / Loki*	Centralized structured logs
Tracing	*Jaeger / OpenTelemetry*	Visualize request paths across services



```
from prometheuse_client import start_http_server, Counter

REQUESTS = Counter("http_requests_total", "Total HTTP Requests")
start_http_server(8001) # expose / metrics
```

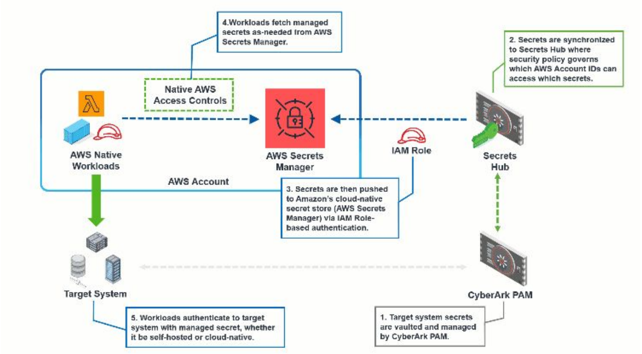
Each service should expose a /metrics endpoint for Prometheus to scrape.

6. Secrets Management & Service Discovery

*Do **not** hardcode credentials or secrets.*

Best Practices:

- Use **environment variables**
- Integrate with **Secrets Managers** (AWS, Azure, Vault)
- Encrypt secrets at rest and in transit


4.Workloads fetch managed secrets as-needed from AWS Secrets Manager.
2. Secrets are synchronized to Secrets Hub where security policy governs which AWS Account IDs can access which secrets.
Native AWS Access Controls
AWS Native Workloads
AWS Secrets Manager
IAM Role
Secrets Hub
AWS Account
3. Secrets are then pushed to Amazon's cloud-native secret store (AWS Secrets Manager) via IAM Role-based authentication.
Target System
CyberArk PAM
5. Workloads authenticate to target system with managed secret, whether it be self-hosted or cloud-native.
1. Target system secrets are vaulted and managed by CyberArk PAM.

When services scale dynamically, you can't hardcode their IPs or ports

Common Methods:

- **Kubernetes DNS**: Services can find each other by name (e.g., http://user-service)
- **Consul**: A third-party tool for dynamic registration and discovery
- **Envoy + Service Mesh**: Advanced routing, retries and observability

Chapter 6: Case Studies & Real-World Architecture Patterns

This chapter focuses on how leading tech companies use microservices at scale and explores the design patterns that help solve common architecture problems

Bîcu Andrei Ovidiu

Netflix

Context:

Originally a Java-based monolith

Transitioned to microservices to support global video streaming

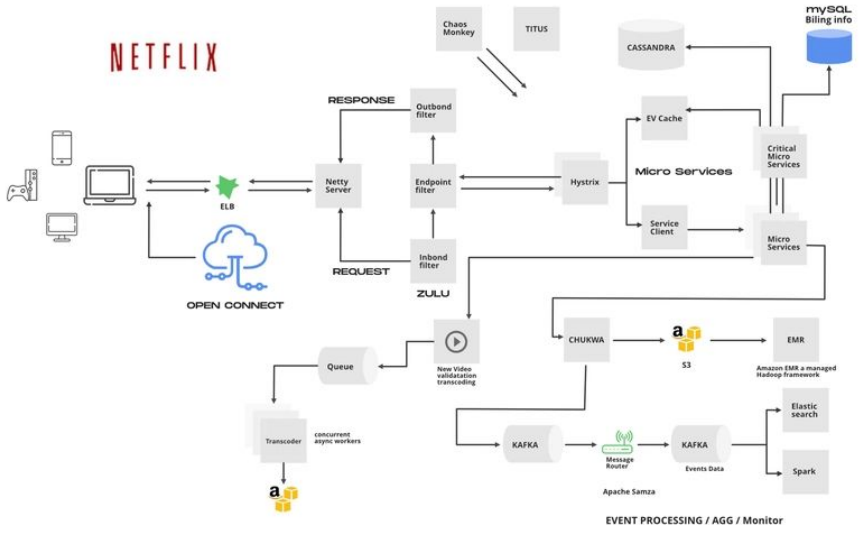
Hundreds of microservices run independently

Key Practices:

- Each service owns its own lifecycle and data
- Uses **Hystrix** for circuit breaking (now part of resilience engineering legacy)
- Relies heavily on **asynchronous communication, service discovery** and **observability**

Lesson:

Microservices can scale globally, but require investment in automation, observability and resilience.



Bîcu Andrei Ovidiu

Amazon

Context:

Migrated from a “two-tier” monolith to microservices

Catalog, payments and delivery systems are separate services

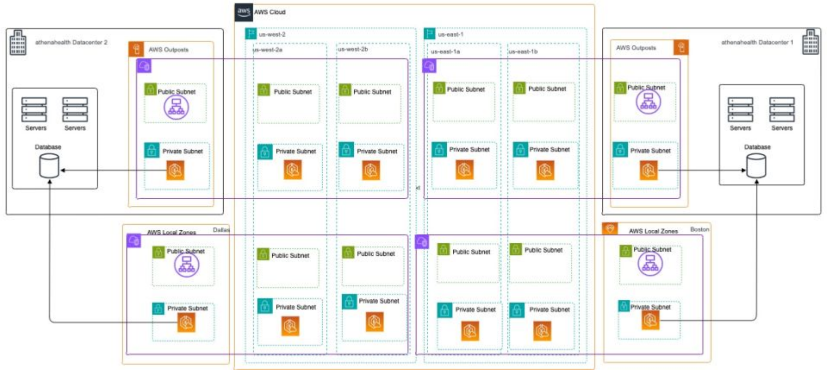
Extreme emphasis on **autonomous teams** (2-pizza rule)

Key Practices:

- Services talk via APIs (REST, internal RPC)
- Each team owns its **service** + **database**
- API-first design with **versioned endpoints**

Lesson:

Microservices support fast-moving, independent teams but only when boundaries are clearly drawn.



Bîcu Andrei Ovidiu

Uber

Context:

Originally a monolith, then aggressively modularized

Grew to over 2000 microservices

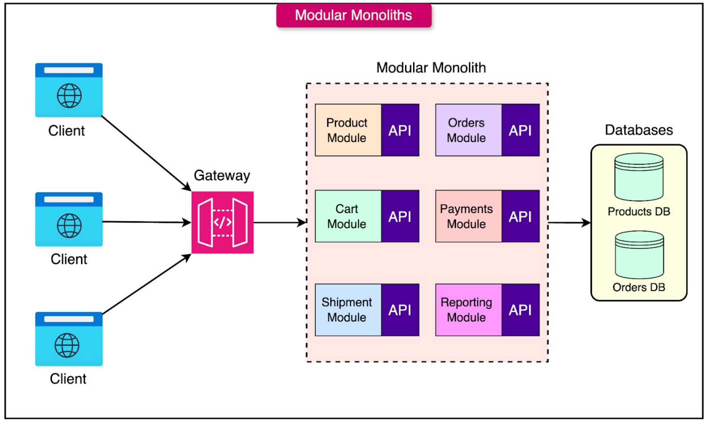
Faced “microservice sprawl” without solid governance

Key Practices:

- Heavily used **gRPC, Kafka, service mesh**
- Shifted toward a **modular monolith** hybrid to regain control

Lesson:

Over-fragmentation can backfire, use microservices ***only when the problem demands it****.*



1. Strangler Fig Pattern & Backends for Frontends (BFF)

Strangler fig: *Wraps a legacy monolith with microservices, gradually migrating features.*

Use When:

- Modernizing a monolithic system incrementally

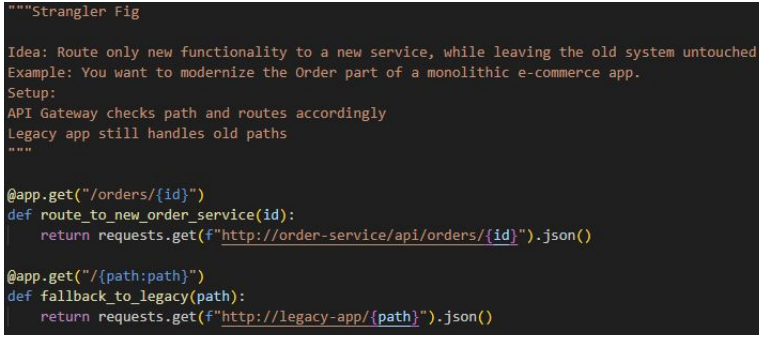
BFF: *Custom backends for each frontend (e.g., web, mobile), aggregating data from multiple services*

Use When:

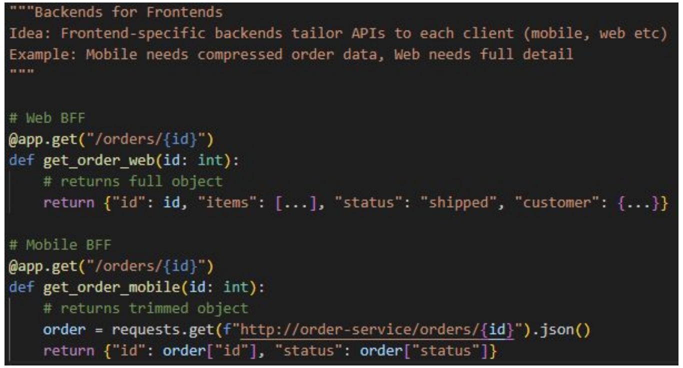
- Different clients need different views or formats of the same data

Example:

- Mobile app gets compressed data from Mobile BFF
- Web app gets the full model from Web BFF



```python
"""Strangler Fig

Idea: Route only new functionality to a new service, while leaving the old system untouched
Example: You want to modernize the Order part of a monolithic e-commerce app.
Setup:
API Gateway checks path and routes accordingly
Legacy app still handles old paths
"""

@app.get("/orders/{id}")
def route_to_new_order_service(id):
    return requests.get(f"http://order-service/api/orders/{id}").json()

@app.get("/{path:path}")
def fallback_to_legacy(path):
    return requests.get(f"http://legacy-app/{path}").json()
```



```python
"""Backends for Frontends
Idea: Frontend-specific backends tailor APIs to each client (mobile, web etc)
Example: Mobile needs compressed order data, Web needs full detail
"""

# Web BFF
@app.get("/orders/{id}")
def get_order_web(id: int):
    # returns full object
    return {"id": id, "items": [...], "status": "shipped", "customer": {...}}

# Mobile BFF
@app.get("/orders/{id}")
def get_order_mobile(id: int):
    # returns trimmed object
    order = requests.get(f"http://order-service/orders/{id}").json()
    return {"id": order["id"], "status": order["status"]}
```

2. Saga Pattern & Circuit Breaker

Saga: *Manages distributed transactions via coordinated steps and compensations*

Use When:

- A business process spans multiple microservices (e.g., Order -> Payment -> Inventory)

Example:

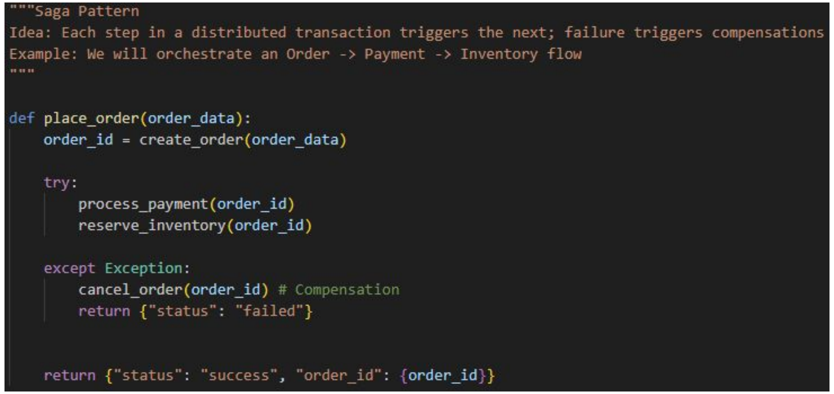
If payment fails, the saga triggers a rollback to cancel the order

Circuit*: Prevents one failing service from cascading failure through the system*

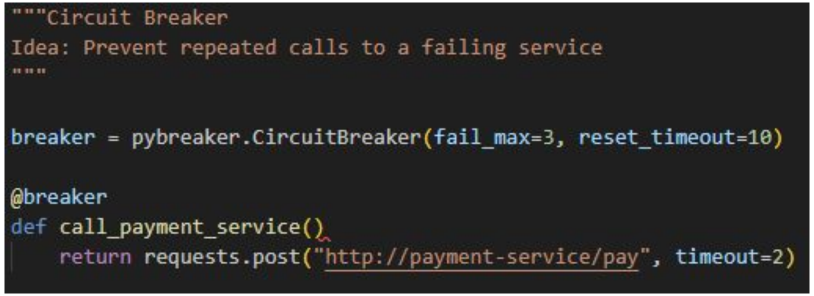
Use When:

- You have API dependencies that may intermittently fail

Tools: pybreaker, Hystrix (Netflix), Resilience4j (Java)



```python
"""Saga Pattern
Idea: Each step in a distributed transaction triggers the next; failure triggers compensations
Example: We will orchestrate an Order -> Payment -> Inventory flow
"""

def place_order(order_data):
    order_id = create_order(order_data)

    try:
        process_payment(order_id)
        reserve_inventory(order_id)

    except Exception:
        cancel_order(order_id) # Compensation
        return {"status": "failed"}


    return {"status": "success", "order_id": {order_id}}
```



```python
"""Circuit Breaker
Idea: Prevent repeated calls to a failing service
"""

breaker = pybreaker.CircuitBreaker(fail_max=3, reset_timeout=10)

@breaker
def call_payment_service()
    return requests.post("http://payment-service/pay", timeout=2)
```

Bîcu Andrei Ovidiu

3. Event Sourcing & CQRS

Event Sourcing: *State is stored as a sequence of events*

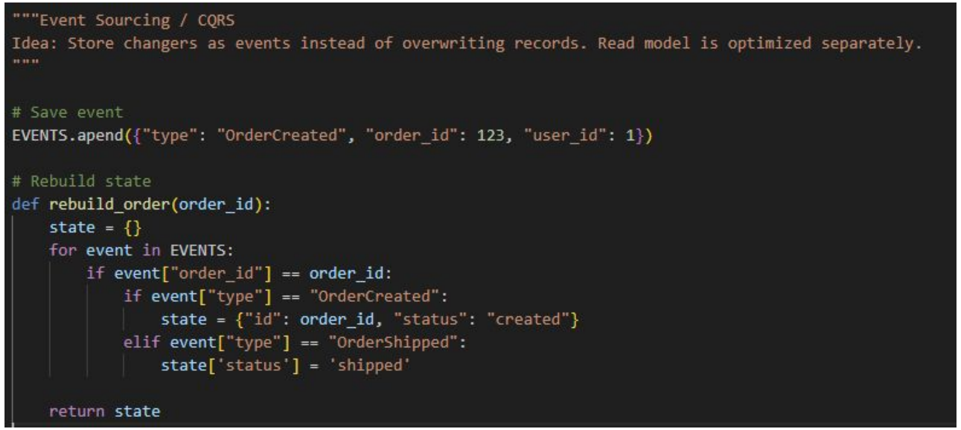
CQRS: *Separate read and write models*

Use When:

- You need auditability, eventual consistency or real-time projections

Example:

Orders written as events, read from a denormalized read model



```python
"""Event Sourcing / CQRS
Idea: Store changers as events instead of overwriting records. Read model is optimized separately.
"""

# Save event
EVENTS.apend({"type": "OrderCreated", "order_id": 123, "user_id": 1})

# Rebuild state
def rebuild_order(order_id):
    state = {}
    for event in EVENTS:
        if event["order_id"] == order_id:
            if event["type"] == "OrderCreated":
                state = {"id": order_id, "status": "created"}
            elif event["type"] == "OrderShipped":
                state['status'] = 'shipped'

    return state
```

Pattern	Pitfall
Strangler Fig	Never finishing the migration
BFF	Too many backends lead to code duplication
Saga	Complex coordination logic
Circuit Breaker	Silent failures if improperly configured
Event Sourcing	Event schema evolution is difficult

Bîcu Andrei Ovidiu

Chapter 7: Security in Microservices - APIs, mTLS, OAuth2 and Hardening

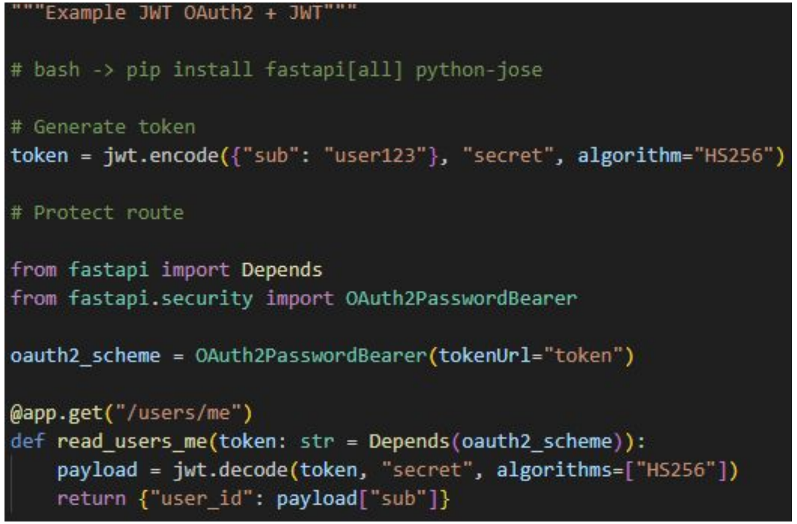
Bîcu Andrei Ovidiu

1. Authentication & Authorization

JWTs are digitally signed tokens used to securely transmit identity information.

Best Practices with JWT

Rule	***Why It Matters***
User **short expiration** times	Limits token abuse if stolen
Always **validate the signature**	Prevents forgery
Don't store sensitive data	JWTs are base64, not encrypted
Rotate secrets periodically	Reduces blast radios



```python
"""Example JWT OAuth2 + JWT"""

# bash -> pip install fastapi[all] python-jose

# Generate token
token = jwt.encode({"sub": "user123"}, "secret", algorithm="HS256")

# Protect route

from fastapi import Depends
from fastapi.security import OAuth2PasswordBearer

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

@app.get("/users/me")
def read_users_me(token: str = Depends(oauth2_scheme)):
    payload = jwt.decode(token, "secret", algorithms=["HS256"])
    return {"user_id": payload["sub"]}
```

JWTs are digitally signed tokens used to securely transmit identity information.

2. Secure Service-to-Service Communication

In microservices, service A shouldn't blindly trust requests from service B.

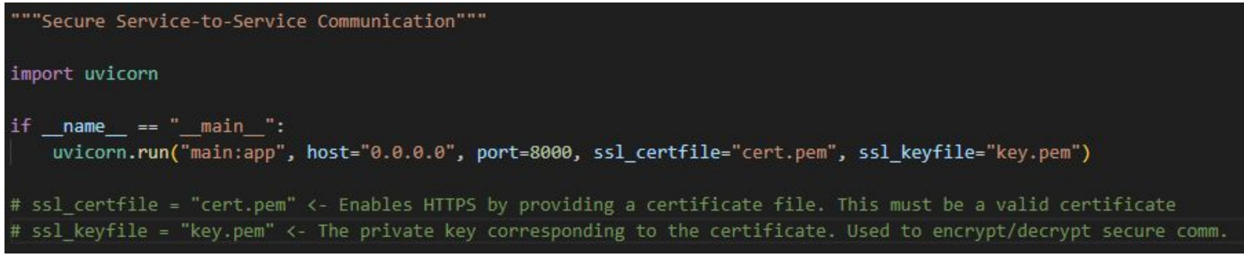
Option 1: Mutual TLS (mTLS)

- Both client and server present certificates
- Ideal for internal communication between services
- Can be managed by **service mesh** (e.g., Istio, Linkerd)

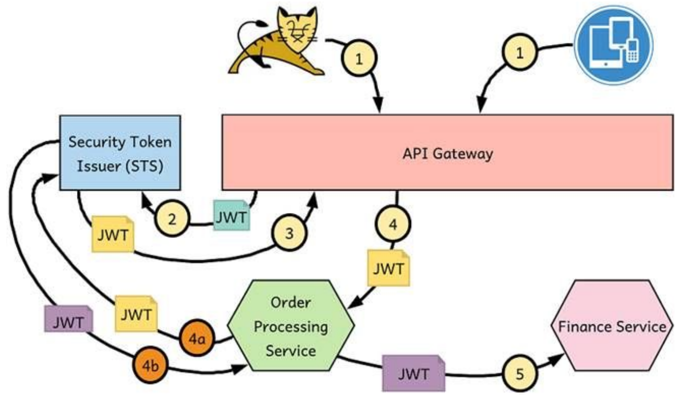
Option 2: OAuth2 + Client Credentials Flow

- Each service has its **own client ID/secret** to get a token.

Preferred for public-facing services, APIs and B2B integrations



```python
"""Secure Service-to-Service Communication"""

import uvicorn

if __name__ == "__main__":
    uvicorn.run("main:app", host="0.0.0.0", port=8000, ssl_certfile="cert.pem", ssl_keyfile="key.pem")

# ssl_certfile = "cert.pem" <- Enables HTTPS by providing a certificate file. This must be a valid certificate
# ssl_keyfile = "key.pem" <- The private key corresponding to the certificate. Used to encrypt/decrypt secure comm.
```



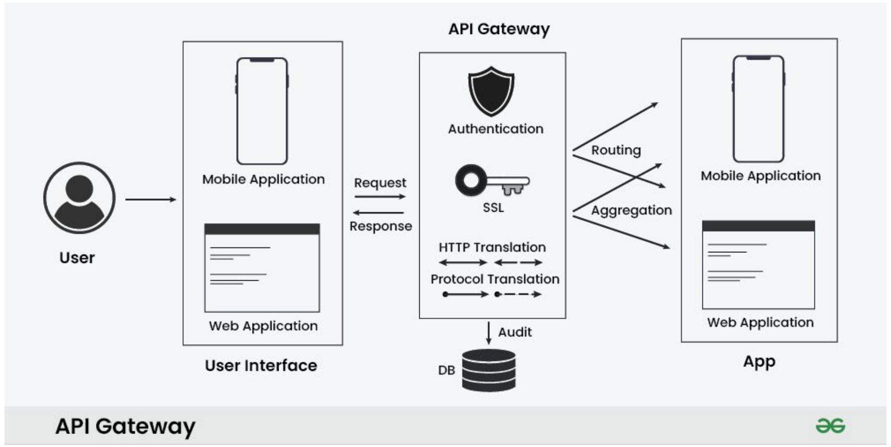
3. Secure APIs with an API Gateway

*Let the **API Gateway enforce:***

- *Authentication (JWT, OAuth2)*
- *Rate limiting (prevent abuse)*
- *Request logging and threat detection*
- *IP whitelisting/geo blocking*

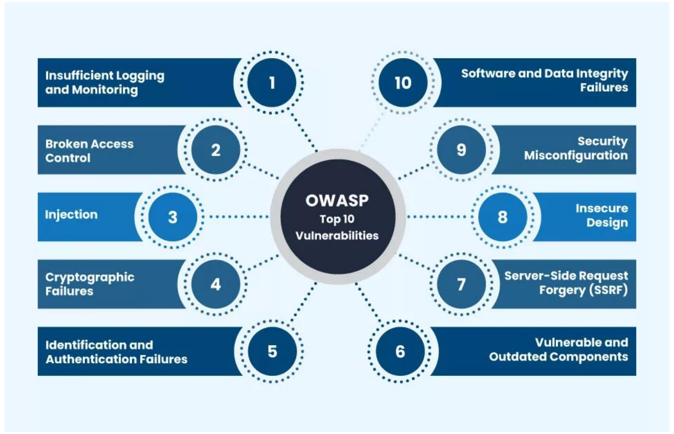
Examples:

- **Kong**
- **AWS API Gateway**
- **NGINX with JWT plugin**



4. Protect Against OWASP Top 10

Risk	*Mitigation*
Insufficient Logging and Monitoring	Implement centralized logging, monitor in real time, alert on suspicious activity and retain logs securely
Broken Access Control	Enforce least privilege, use secure access control mechanisms, deny access by default and test thoroughly
Injection	Use parameterized queries, input validation and stored procedures. Avoid dynamic SQL
Cryptographic Failures	Use strong, up-to-date encryption algorithms; avoid hardcoded secrets, enforce TLS/SSL
Identification and Authentication Failures	Implement MFA, strong password policies, session timeouts and secure credential storage
Vulnerable and Outdated Components	Maintain inventory of components, monitor for updates and apply security patches promptly
Server-Side Request Forgery (SSRF)	Validate and sanitize URLs, enforce network segmentation and use deny-by-default firewall rules
Insecure Design	Perform threat modeling, apply secure design patterns and review design architecture for flaws
Security Misconfiguration	Automate secure configurations, disable unnecessary features and conduct regular configuration reviews
Software and Data Integrity Failures	Use signed code, implement CI/CD integrity checks and protect against unauthorized code changes

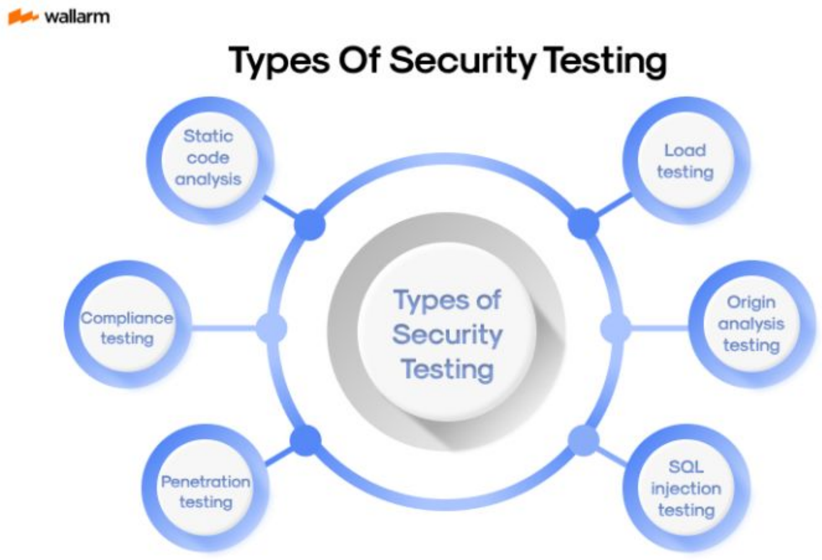

OWASP
Top 10
Vulnerabilities
Insufficient Logging and Monitoring
1
Broken Access Control
2
Injection
3
Cryptographic Failures
4
Identification and Authentication Failures
5
Vulnerable and Outdated Components
6
Server-Side Request Forgery (SSRF)
7
Insecure Design
8
Security Misconfiguration
9
Software and Data Integrity Failures
10

Bîcu Andrei Ovidiu

5. Security Testing Tools

Tools:

- **Bandit**: Python static code analysis
- **OWASP ZAP**: API penetration testing
- **Trivy:** Container image vulnerability scanner
- **Docker Bench:** Audits container configs



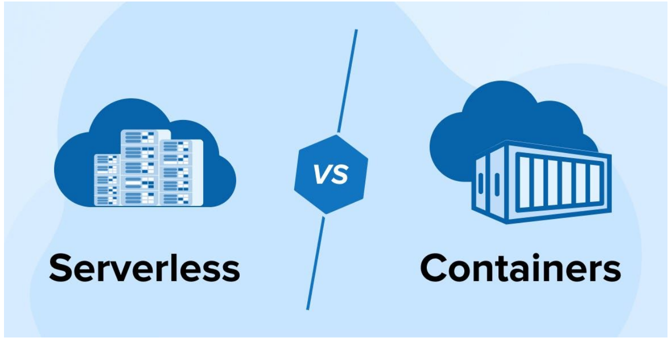
Chapter 8: Serverless vs Containers

Serverless vs Containers - Comparison Diagram

Feature	***Serverless (FaaS)***	***Containers (Docker/K8s)***
Deployment Unit	Individual Functions	Services packaged in containers
Startup Time	Slower (cold starts)	Moderate (depends on the image pull)
Statefulness	Stateless only	Stateful or stateless
Runtime Control	Minimal (provider-managed)	Full control over OS, libraries, and tools
Scaling	Auto-scaled per request	Auto/Manual (replica-based)
Billing	Pay-per-request	Pay-per-resource (CPU/RAM)
Max Execution Time	15 min (AWS Lambda)	Unlimited
Best For	Lightweight APIs, automation, triggers	Full-featured services, long processes

Serverless vs Containers - When to use what

Use Case	*Go Serverless If…*	*Go Containers If…*
Small, stateless APIs	Fast, low-traffic or bursty usage	Overhead isn't worth it
Long-running background tasks	Serverless timeouts	Use containers for queues, consumers
Real-time apps (chat, websockets)	Serverless is not designed for it	Containers handle persistent connections
Developer control	Limited runtimes and debugging	Full control of the stack
Startups / POCs	Rapid iteration, low cost	Containers if already using Docker locally



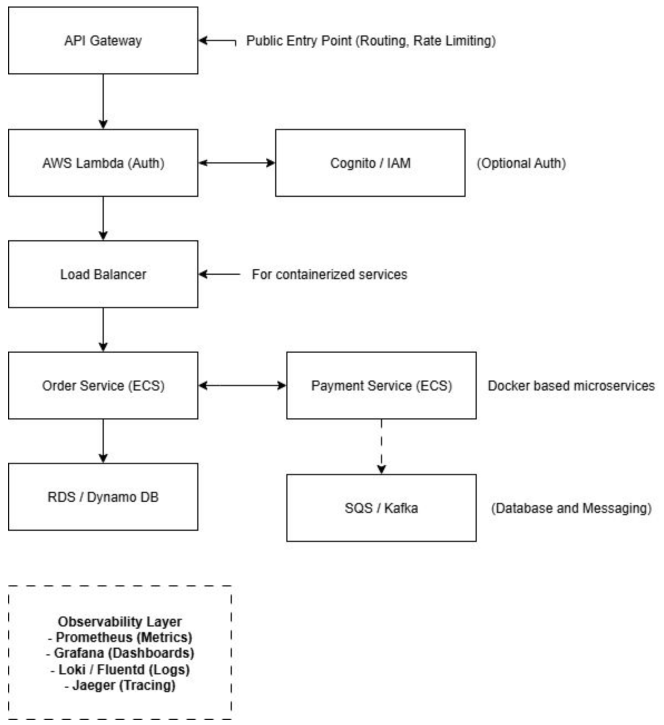
Option 3: Hybrid Architecture

Many modern systems use both:

- **Serverless** for lightweight tasks like auth, email, triggers
- **Containers** for core business logic, data handling, real-time APIs

Notes on the image:

- **API Gateway:** Entry point for clients (web/mobile). Routes traffic to Lambda or services
- **Lambda Functions**: Handle auth, webhooks or scheduled jobs with low overhead
- **ECS / EKS / GKE:** Run core microservices in containers, scale them based on usage
- **Message Broker (SQS, Kafka):** Handles async workflows (e.g., order events, emails)
- **Databases:** Each microservice owns its data (polyglot persistence is possible)
- **Observability:** Integrated with Prometheus, Grafana and tracing tools
- **Security:** Uses IAM roles, secrets manager, mTLS and JWT/OAuth2



Chapter 9: Microservices Cheat Sheet - What to Do and What to Avoid

What to Do

Design Smart

- Build services around **business domains**, not technical layers
- Use **bounded contexts** and **DDD** principles
- Define **clear ownership** per service and team

Keep Services Cohesive

- Each service should:
  - Do **one thing well**
  - Own its **data and logic**
  - Be **deployable independently**

Automate Everything

- Use **CI/CD pipelines** for every service
- Automate:
  - Testing
  - Docker builds
  - Deployments
  - Rollbacks

Stabilize APIs

- Define with **OpenAPI** or **Protobuf**
- Version all public APIs and events (/v1/orders, order_created_v2)
- Treat your contracts as **products**

Log, Monitor, Trace

- Add:
  - **Structured logs** (JSON)
  - **Metrics** (Prometheus)
  - **Tracing** (OpenTelemetry, Jaeger)
- Correlate logs with request/trace IDs

Secure Everything

- Use **JWT** or **OAuth2** for APIs
- Use **mTLS** for internal service traffic
- Store secrets in **Vault, AWS Secrets Manager,** or **Kubernetes secrets**
- Apply **rate limiting, CORS, API keys, audit logs**

Resilience First

- Always set **timeouts**
- Add **retries with backoff**
- Use **circuit breakers**
- Design for **graceful degradation**

Version & Document

- Document:
  - API contracts
  - Service ownership
  - Operational dependencies
- Version everything that others rely on

What to Avoid

Overengineering from Day one

- Don’t split into microservices too early
- Start with a **modular monolith**
- Add complexity only when it solves a real problem

Tight Coupling Between Services

- Don’t:
    - Share databases
    - Call internal functions
    - Assume other services are always up
- Use **APIs, queues,** or **events** instead

Breaking Contracts

- Don’t change APIs or event formats without versioning
- Avoid surprise deployments that break consumers

Ignoring Observability

- Don’t rely on print() or guesswork
- If you can’t trace an issue end-to-end, you’ll burn hours debugging it

Skipping Security Hygiene

- Don’t hardcode secrets or tokens
- Don’t run production without HTTPS
- Don’t allow unauthenticated internal requests

Neglecting Documentation

- If a service fails at 3 AM, will someone else know how to debug it?
- Write minimal but meaningful docs

Letting Everyone Touch Everything

- Avoid shared ownership across teams
- Set clear **service boundaries** and **on-call responsibilities**

Glossary (A-Z)

API Gateway - A server that acts as the single entry point into a system. It routes requests, applies security (auth, rate limiting), transforms responses and can hide service complexity from the client.

ASGI (Asynchronous Server Gateway Interface) - The standard interface between Python async web frameworks (e.g., FastAPI) and web servers like Uvicorn.

Authentication - The process of verifying the identity of a user or service. Common mechanisms include JWT, OAuth2 and API keys.

Authorization - Determining if a user/service has permission to perform a specific action. Often handled with role-based access control (RBAC) or scopes.

Autoscaling - Automatically increasing or decreasing the number of service instances based on load, traffic or CPU usage. Supported in Kubernetes, AWS Lambda, etc.

Backends for Frontends (BFF) - A design pattern where each client (web, mobile, etc) has its own tailored backend service that aggregates and formats data for that specific UI.

Bounded Context - A DDD concept referring to the clear boundary within which a particular domain model applies. Each microservice should map to one bounded context.

Broker (Message Broker) - Software that routes messages between services in asynchronous communication. Examples: RabbitMQ, Kafka, Redis Streams.

Circuit Breaker - A resilience pattern that prevents a service from repeatedly calling a failing dependency. Trips open after a failure threshold and reset after a cooldown.

CD/CD (Continuous Integration / Continuous Deployment) - A set of practices and tools to automate testing, building and deploying applications. Common tools: GitHub Actions, ArgoCD, GitLab CI.

Cold Start - The initial delay that occurs when a serverless function (e.g., AWS Lambda) is invoked after a period of inactivity. This includes booting the runtime and loading dependencies.

Container - A lightweight, portable package that includes everything needed to run an app: code, runtime, libraries and configs. Popular container: Docker.

CQRS (Command Query Responsibility Segregation) - A pattern that separates read and write operations into different models/services to improve performance and scalability.

CRUD - Short for Create, Read, Update, Delete - basic database operations.

Data Consistency - Refers to how synchronized data is across services. In microservices, strict consistency is often replaced by **eventual consistency** due to the distributed architecture.

Database per Service - A best practice where each microservice owns and manages its own database, avoiding shared state and tight coupling

DDD (Domain-Driven Design) - A design methodology that focuses on aligning software structure with business domains. Core concepts include **bounded contexts, aggregates** and **ubiquitous language**.

Deployment - The process of moving application code into a live or staging environment. Can be manual or automated via CI/CD pipelines.

DevOps - A set of practices that combines development and IT operations to shorten the system development lifecycle and improve deployment quality and frequency.

Distributed System - A system composed of multiple services running on different machines or containers, communicating over a network. Microservices are a common form.

Docker - An open-source platform used to package and run applications in **containers**. Enables consistent environments across dev, test and prod.

Downstream Service - A service that is **called by** another service. For example, in a payment workflow, the inventory service is a downstream from the order service.

Endpoint - A specific route or URL exposed by an API that clients can call to perform actions or retrieve data (e.g., /api/orders/{id}).

Environment Variables - Key-value pairs injected into running services to manage configuration and secrets without hardcoding. Used for posts, DB URIs, credentials, etc.

Event-Driven Architecture (EDA) - A communication pattern where services **emit** and **respond to events** (e.g., OrderCreated, PaymentReceived) instead of direct API calls.

Eventual Consistency - A model where data across services or nodes is **not immediately synchronized**, but will become consistent over time. Often used in distributed systems.

Fault Tolerance - The ability of a system to continue operating even when some components fail. Achieved through retries, circuit breakers, redundancy, etc.

FastAPI - A modern, async-friendly Python web framework for building APIs. Known for performance, built-in validation and OpenAPI docs.

gRPC - A high-performance binary RPC (Remote Procedure Call) protocol developed by Google. Uses Protocol Buffers (protobuf) and supports streaming and strict contracts.

Gateway (API Gateway) - A front-facing layer that routes traffic to services, handles security (auth, throttling) and aggregates responses. Examples: Kong, AWS API Gateway, NGINX.

GitOps - A DevOps workflow where **Git** is the source of truth for infrastructure and deployments. Tools like ArgoCD and Flux automate syncing state from Git to environments.

IAM (Identity and Access Manager) - Cloud-native security system (e.g., AWS IAM) that controls **who can access what** resource. Used for fine-grained permissions across services and infrastructure.

Idempotency - A property where repeating the same operation multiple times **produces the same result**. Important in APIs and messaging systems to avoid duplication.

Ingress Controller - A Kubernetes component that manages external access to services inside a cluster. Examples: **NGINX Ingress, Traefik, Istio Gateway**.

Infrastructure as Code (IaC) - Defining infrastructure (networks, servers, databases) using **declarative code**. Tools include Terraform, AWS CloudFormation and Pulumi.

Instance - A single running copy of a service or container. You scale services by running multiple instances.

Internal API - An API exposed **within your organization** or cluster, not publicly accessible. Often used for service-to-service communication.

Istio - A popular **server mesh** that provides advanced networking, observability, security (mTLS) and traffic control in Kubernetes.

JSON Web Token (JWT) - A compact, URL-safe token used to transmit identity and claims. Includes a signature to ensure integrity. Common for **stateless authentication**.

Kafka - A distributed, high-throughput messaging system. Used for event-driven systems and real-time streaming data pipelines.

Kubernetes (K8s) - An open-source platform for **orchestrating** containerized applications. Manages deployments, scaling, service discovery and health checks.

Latency - The time delay between a request and its response. High latency leads to sluggish performance and poor user experience.

Load Balancer - A networking component that distributes traffic across multiple service instances for performance and availability.

Logging - Capturing structured or unstructured data about what services are doing. Important for debugging, audits and monitoring.

Loki - A log aggregation tool from Grafana Labs. Works well with **Prometheus** and **Grafana** to enable full observability stacks.

Message Queue - A system that holds messages between producers and consumers. Enables **asynchronous communication**. Example: RabbitMQ, Kafka or SQS.

Microservices - An architectural style where an application is built as a suite of **small, independent services**, each responsible for a single business capability and communicating over APIs

Modular Monolith - A monolithic app structured with **clean, isolated modules**. A stepping stone to microservices that enables clear boundaries without distributed complexity.

Namespace - A logical grouping or isolation unit within Kubernetes (or other systems) used to separate resources, services and configurations within the same cluster.

Node - In Kubernetes, a **worker machine** (VM or physical) that runs containers. Nodes are managed by the Kubernetes control plane.

NoSQL - A type of database that doesn't follow traditional relational models. Ideal for flexible, scalable data storage. Examples: MongoDB, DynamoDB or Redis.

Observability - The ability to understand the internal state of a system based on its external outputs (logs, metrics, traces). Crucial for debugging and incident response in microservices.

OpenAPI - A specification for defining RESTful APIs. Enables automatic documentation, code generation and validation. Used by FastAPI, Swagger etc.

OpenTelemetry (OTel) - An open-source standard for collecting distributed **traces, metrics** and **logs**. Vendor-neutral and compatible with tools like Jaeger and Prometheus.

Orchestration - Automated coordination of containers, deployments and services in production environments. Kubernetes is the most common orchestrator.

Pagination - A technique to divide large datasets into manageable pages in API responses. Helps avoid over-fetching and performance issues.

Payload - The actual data sent in an API request or response (e.g., JSON body). It contains useful content excluding metadata like headers.

Pod - The smallest deployable unit in Kubernetes. A pod contains one or more containers that share the same network and storage.

Polyglot Persistence - The use of multiple types of databases (SQL, NoSQL, graph, etc) in a single system. Common in microservices where each service chooses the best-fit DB.

Prometheus - A time-series database and monitoring system. Used to collect metrics (e.g., CPU, latency) from microservices and expose them to dashboards or alerting tools.

Protobuf (Protocol Buffers) - A binary data format used by **gRPC** for efficient serialization. Enables fast and strongly-typed communication between services.

Rate Limiting - Controlling how many requests a client or IP can send in a given period. Helps prevent abuse, overloading or DoS attacks.

Replica - A duplicate instance of a container or pod used to scale a service for load balancing and high availability.

Request ID / Correlation ID - A unique identifier attached to each request and passed through services to trace its full journey across the system. Crucial for **distributed tracing**.

Resilience - The ability of a system to withstand and recover from failures without interrupting overall service. Include retries, timeouts and circuit breakers.

REST (Representational State Transfer) - A common architectural style for web APIs using HTTP methods (GET, POST, etc) and resources. Known for simplicity and compatibility.

Retry Pattern - A resilience technique that retries failed operations a limited number of times with backoff delays, often used for transient errors.

Rolling Deployment - A deployment strategy where updates are rolled out gradually to instances, minimizing downtime and risk.

Saga Pattern - A pattern to manage **distributed transactions** using a sequence of local transactions with compensating actions for rollback.

Protobuf (Protocol Buffers) - A binary data format used by **gRPC** for efficient serialization. Enables fast and strongly-typed communication between services.

Scalability - The system's ability to handle increased load by adding more resources. Can be **vertical** (bigger machines) or **horizontal** (more instances).

Secrets Management - A secure way to store API keys, passwords and tokens. Toos: AWS Secrets Manager, HashiCorp Vault, Kubernetes Secrets.

Service Discovery - The process by which services **find and communicate** with each other dynamically, without hardcoding addresses. Examples: Kubernetes DNS, Consul.

Service Mesh - An infrastructure layer that handles service-to-service communication, including mTLS, retries, routing and observability. Tools: Istio or Linkerd.

Sidecar Container - A helper container deployed alongside a main app container in the same pod. Commonly used for logging, metrics, proxies etc.

Tenant / Multi-Tenancy - A design where a single system serves **multiple independent users or organizations** (tenants). Often used in SaaS architectures.

Tenacity - A Python library used to implement **retry logic** with customizable backoff, stop conditions and error handling in microservices.

Test Pyramid - A testing strategy emphasizing: Many **unit tests**. Fewer **integration tests**. Minimal **end-to-end tests**. Helps maintain fast, reliable feedback loops.

Timeout - A maximum time to wait for a request to complete. Prevents services from hanging indefinitely during slow network or service issues.

Token (Auth Token) - A secure object (e.g., JWT, OAuth2 token) used to represent a user or client's authentication and/or authorization state in API requests.

Tracing - A technique to track a single request as it travels through multiple services. Enables **end-to-end debugging** in distributed systems.

Traffic Shaping - Controlling how traffic flows between services or versions. Examples: **canary releases, blue/green deployments, rate limiting.**

Uvicorn - A lightning-fast ASGI web server used with Python frameworks like FastAPI. Runs production-grade async applications.

Versioning (API) - Creating multiple versions of an API (/v1/users) to support backward compatibility as services evolve.

Vertical Scaling - Improving performance by upgrading resources (CPU, RAM) of a single instance. Contrast with **horizontal scaling** (adding instances).

Webhooks - Server-to-server callbacks triggered by events. For example, a payment provider sends a webhook to confirm a transaction status.

Zero Trust Architecture - A security model that **assumes no trust** between components; every request must be authenticated, even within internal networks.

Zipkin - An open-source tool for **distributed tracing,** similar to Jaeger. Visualizes how requests travel across services.

References

Books & Foundational Reading

Building Microservices - *Sam Newman*; A practical, opinionated guide for designing and evolving microservices in the real world.

Microservice Patterns - *Chris Richardson*; Covers services decomposition, the Saga pattern and architecture trade-offs in depth.

Software Architecture: The Hard Parts - *Neal Ford, Mark Richards, Pramod Sadalage, Zhamak Dehghani*; Focuses on architectural decision-making, trade-offs and distributed challenges.

Domain-Driven Design - *Eric Evans*; The foundational book of DDD, including concepts like bounded contexts and aggregates.

Implementing Domain-Driven Design - *Vaughn Vernon*; A more practical and code-focused companion to Eric Evans' work.

Site Reliability Engineering (SRE) - *Betsy Beyer et al* (Google); A critical read for reliability, monitoring and service-level objectives in cloud-native systems.

Official Documentation & Standards

FastAPI - https://fastapi.tiangolo.com/

Kubernetes - https://kubernetes.io/docs/

Docker - https://docs.docker.com/

Prometheus - https://prometheus.io/docs/

OpenTelemetry - https://opentelemetry.io/

OpenAPI - https://swagger.io/specification/

OAuth 2.0 Authorization Framework (RFC 6749) - https://datatracker.ietf.org/doc/html/rfc6749

JWT (RFC 7519) - https://datatracker.ietf.org/doc/html/rfc7519

OWASP Top 10 (2021 Edition) - https://owasp.org/Top10/

Tools and Projects Mentioned

Uvicorn - https://www.uvicorn.org/

PyBreaker (Circuit Breaker for Python) - https://pypi.org/project/pybreaker/

Tenacity (Retrying library) - https://tenacity.readthedocs.io/

Jaeger Tracing - https://www.jaegertracing.io/

Grafana - https://grafana.com/

Loki (Logging by Grafana Labs) - https://grafana.com/oss/loki/

RabbitMQ - https://www.rabbitmq.com/

Apache Kafka - https://kafka.apache.org/

AWS Lambda - https://docs.aws.amazon.com/lambda/

Serverless Framework - https://www.serverless.com/

GitHub Actions - https://docs.github.com/en/actions

Terraform by HashiCorp - https://www.terraform.io/

HashiCorp Vault - https://www.vaultproject.io/

AWS Secrets Manager - https://aws.amazon.com/secrets-manager/

Zappa (Serverless Python framework) - https://github.com/zappa/Zappa

Articles, Blogs & Real-World Cases

"The Netflix Tech Blog" - https://netflixtechblog.com/

"The Amazon Builder's Library" - https://aws.amazon.com/builders-library/

"Microservices at Uber" - https://eng.uber.com/microservice-architecture/

"Microservices" - *Martin Fowler;* https://martinfowler.com/articles/microservices.html

"A Guide to Service Mesh" - Buoyant (Linkerd) – https://linkerd.io/what-is-a-service-mesh/