DevOps: Software Engineering, Deployment, and Operation

W07 Microservices and Scalable Architectures

Stephan Krusche, Ingo Weber

6 June 2025

Schedule

#	Date	Subject
1	25.04.25	Introduction
2	02.05.25	DevOps Culture and Organization
3	09.05.25	Software Engineering Principles in DevOps
4	16.05.25	Virtualization and Containerization
5	23.05.25	Cloud Integration and Deployment Platforms
6	30.05.25	Continuous Delivery and Deployment Strategies
7	**06.06.25**	**Microservices and Scalable Architectures**
8	13.06.25	Monitoring and Observability
9	20.06.25	Post-Deployment Management
10	27.06.25	Security and Compliance in DevOps
11	04.07.25	Advanced DevOps Topics and Trends
12	11.07.25	DevOps for AI-based Systems
13	18.07.25	Guest Workshop
14	25.07.25	Course Review + Test Exam

Roadmap

- **Context**
  - You have a basic understanding of DevOps and its culture
  - You can apply agile SE, CI and CD
  - You have a good understanding of Docker (Compose) and Kubernetes
  - You understand cloud deployment strategies
- **Learning goals**
  - Explain the core principles of microservices and scalable architectures
  - Explain Conway’s law and its practical implications
  - Design an API using REST (OpenAPI) and gRPC (proto)
  - Differentiate between service discovery, API gateways, load balancers, and service meshes
  - Differentiate between horizontal and vertical scaling

Outline

- **Microservices design principles**
- Scaling and load balancing
- Service discovery and API gateways
- Service meshes

The evolution of software architecture

1990's

SPAGHETTI-ORIENTED
ARCHITECTURE
(aka Copy & Paste)

2000's

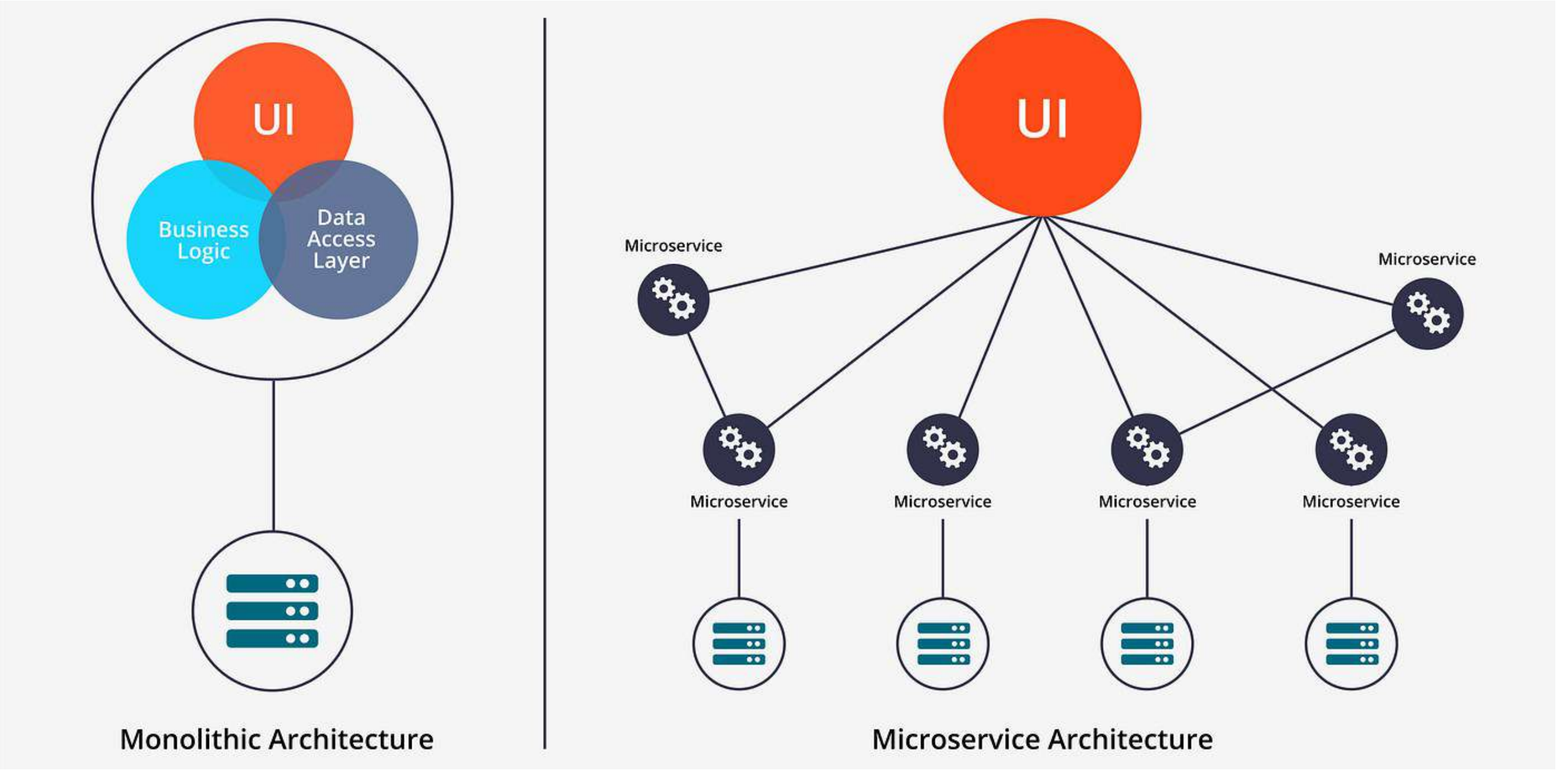
LASAGNA-ORIENTED
ARCHITECTURE
(aka Layered Monolith)

2010's

RAVIOLI-ORIENTED
ARCHITECTURE
(aka Microservices)

Microservice vs. monolithic architecture

- **Definition**: “Microservices are a software architectural pattern where applications are built as a collection of small, independently deployable services, each focused on a specific business capability”

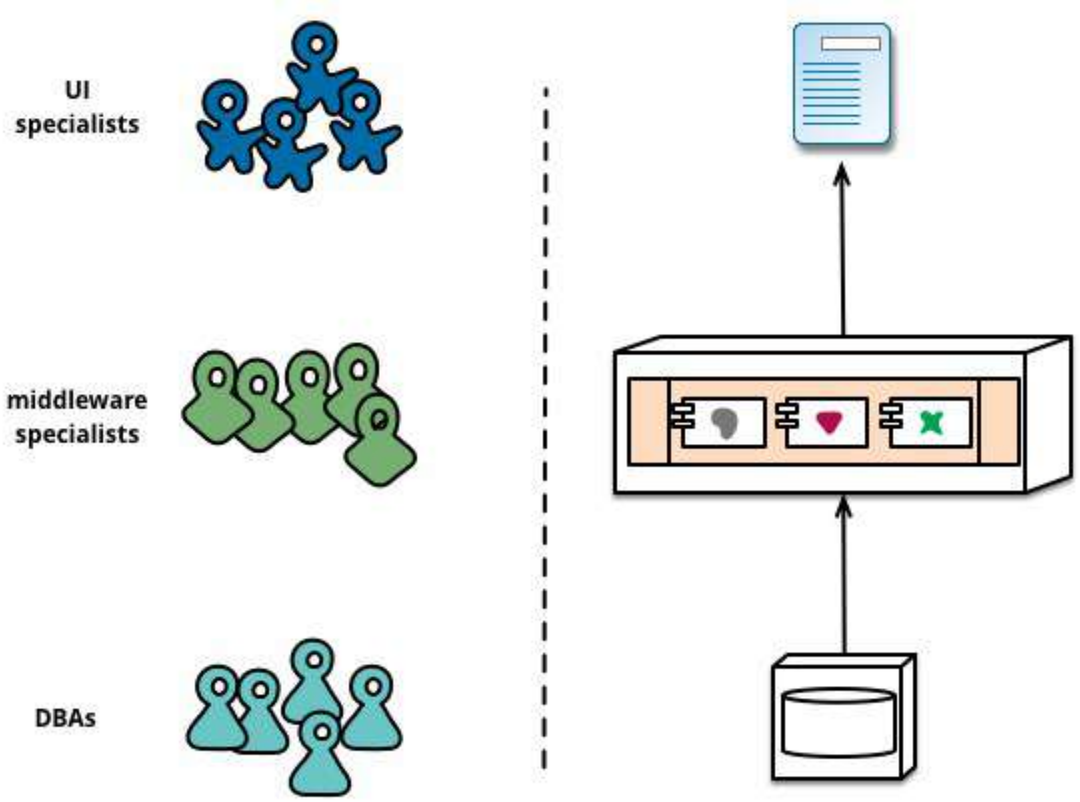


Source: https://blog.flexbase.in/digital-transformation-to-microservices-the-approach

Conway’s Law

“Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure”

-- Melvin Conway, 1968



Siloed functional teams…

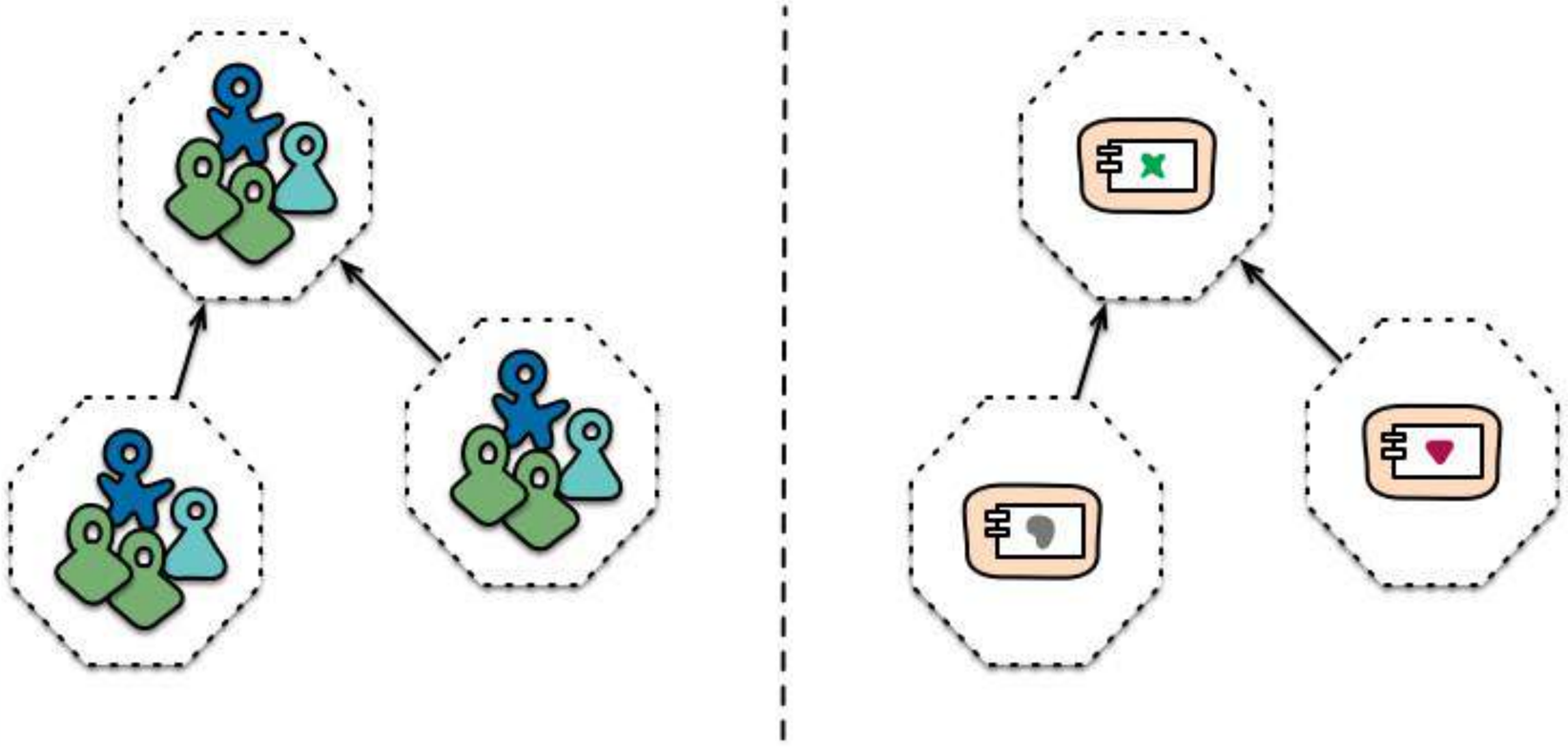
…lead to siloed application architectures

Source: https://martinfowler.com/articles/microservices.html

Conway’s Law

“Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure”

-- Melvin Conway, 1968

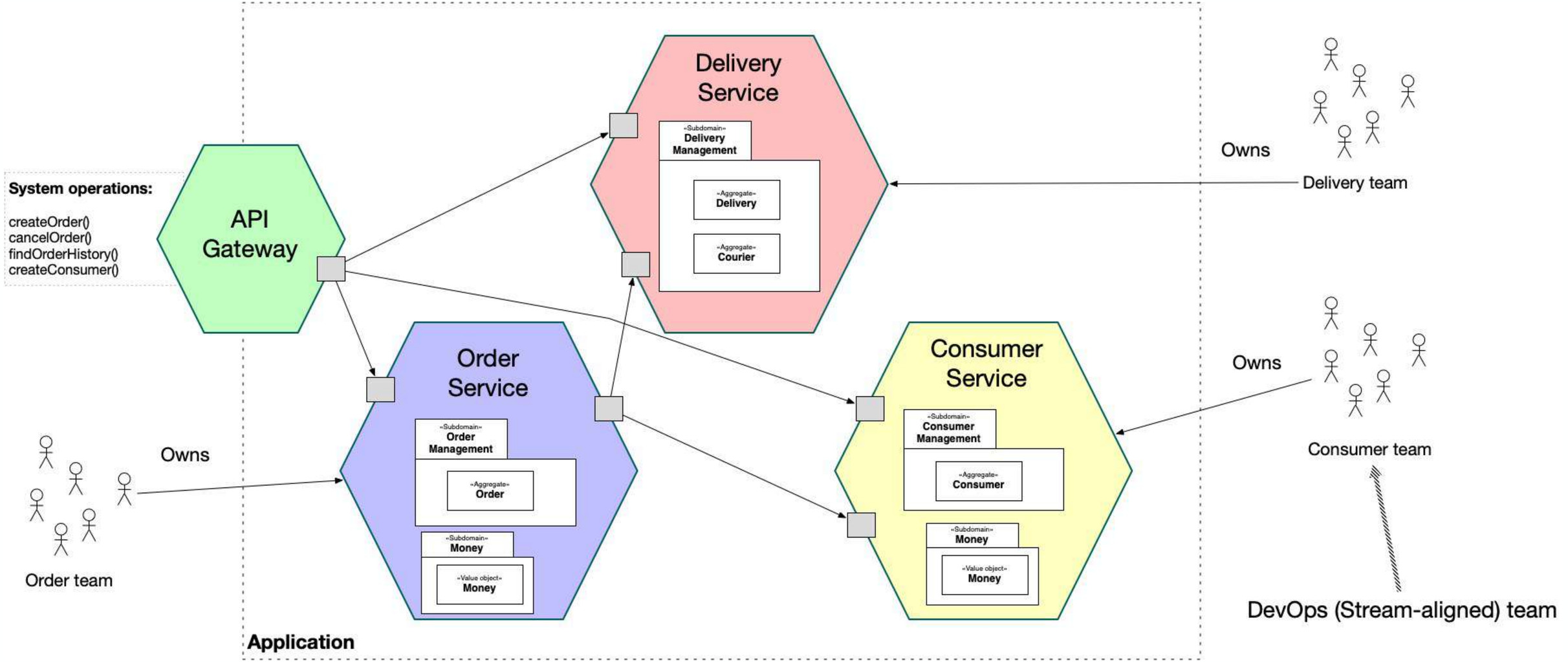


Cross-functional teams…

…organized around capabilities

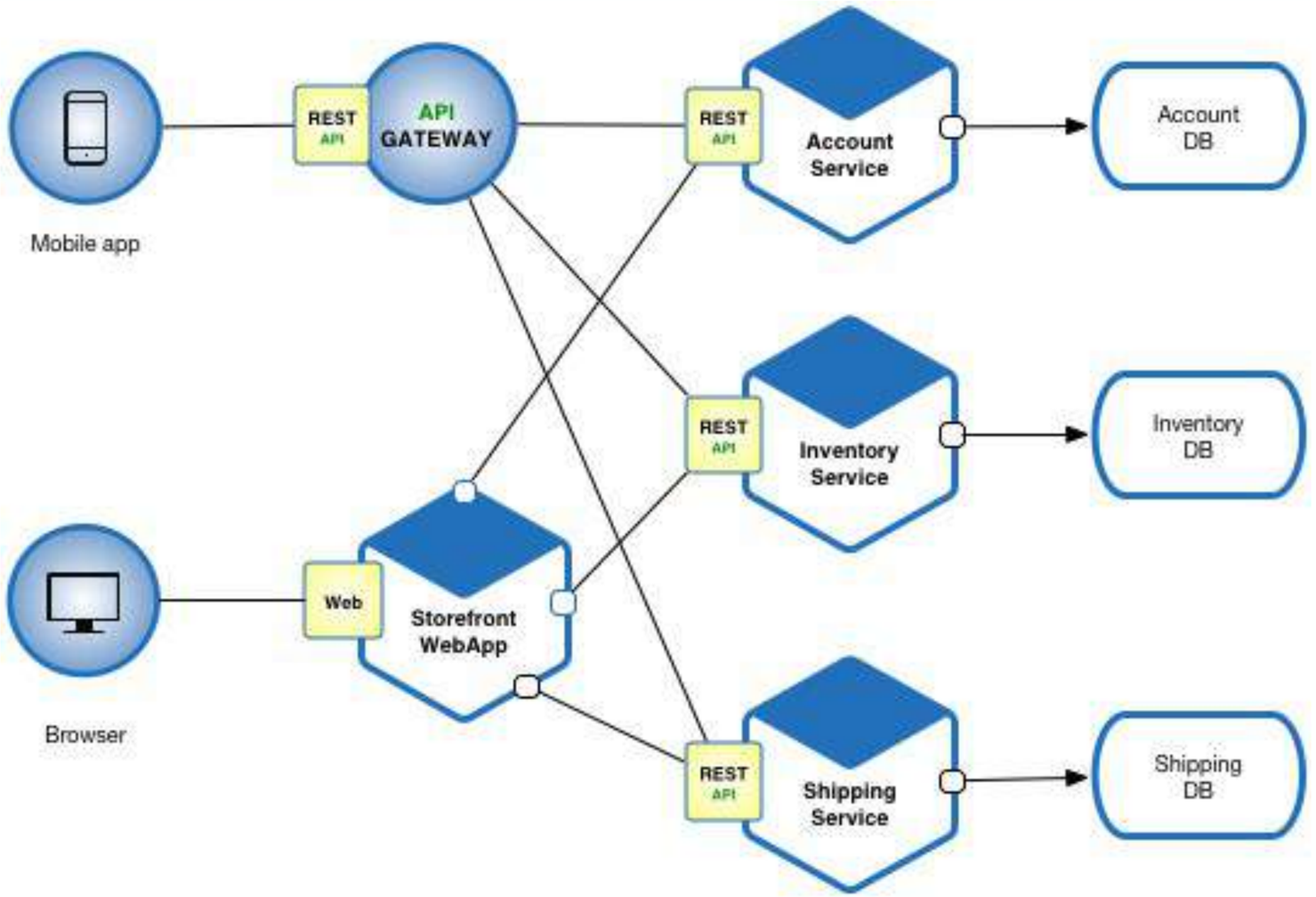
Source: https://martinfowler.com/articles/microservices.html

Example



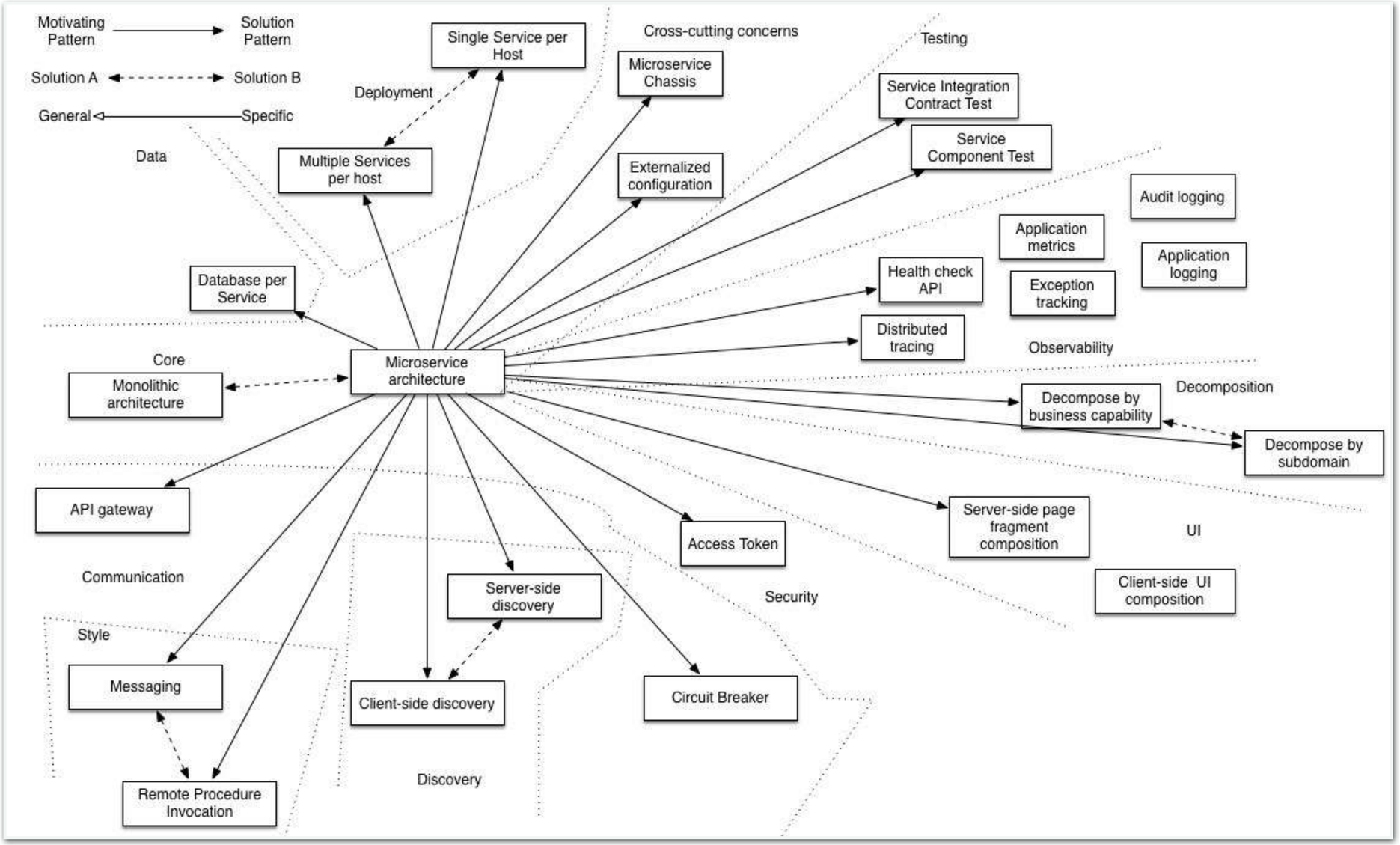
Source: https://microservices.io/patterns/microservices.html

Example



Source: https://microservices.io/patterns/microservices.html

Microservice patterns



Source: https://microservices.io/patterns/microservices.html

Example: database per service

- 🗃️ Each microservice owns its **own private database** — no direct access from other services
- 🧩 **Core principle**: enforces **strong service boundaries** on the domain level, services **communicate via APIs**, not shared schemas or DB joins
- ✅ **Advantages**
  - **Loose coupling**: changes in one DB don’t affect others
  - Enables **independent scaling**, deployments, and schema evolution
  - Improves **data ownership** and **security isolation**
  - Matches well with **bounded contexts** in domain driven design
- ⚠️ **Disadvantages**
  - **Cross service queries** are harder (no joins → must aggregate via APIs or events)
  - Potential for **data duplication**
  - More complex **data governance and backups**
  - Requires **eventual consistency** and **saga distributed transaction pattern** for transactions

Coordinates a series of local transactions across services using event-based or command-based compensation

Core principles of microservices

- **Single responsibility**: each service focuses on a single capability
- **Independent deployment**: services can be deployed without affecting others
- **Decentralized data management**: each service can manage its own database
- **Lightweight communication**: services communicate using APIs, often REST or gRPC

Benefits of microservices

- **Scalability**: scale individual components based on demand
- **Resilience**: failures in one service don’t bring down the entire system
- **Flexibility**: use different technologies for different services
- **Faster development**: independent teams can work on separate services

Challenges of microservices

- **Complexity**: managing many services increases operational overhead
- **Data consistency**: maintaining consistency across services is harder
- **Network latency**: increased communication may lead to performance bottlenecks
- **Deployment**: requires robust CI/CD pipelines and monitoring
- **Testing:** more difficult to debug and test the entire system

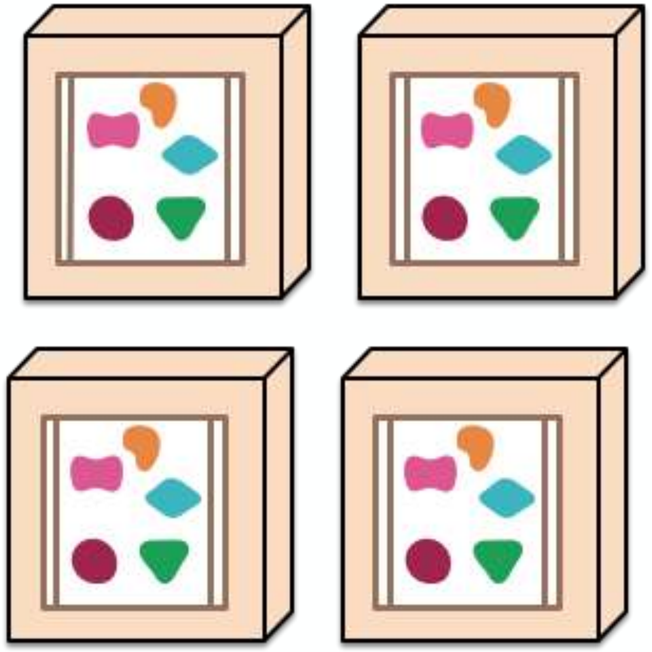
Key components of a microservices architecture

- **Service registry**: keeps track of available services (e.g., Eureka)
- **API gateway**: acts as an entry point for clients
- **Inter service communication**: REST, gRPC, or messaging queues like Apache Kafka
- **Data management**: polyglot persistence (each service can have its own database)
- **Monitoring**: tools like Prometheus and Grafana for tracking performance

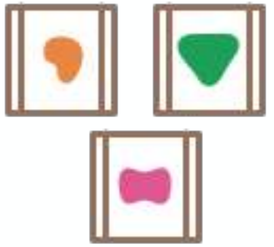
Scaling

A monolithic application puts all its functionality into a single process...

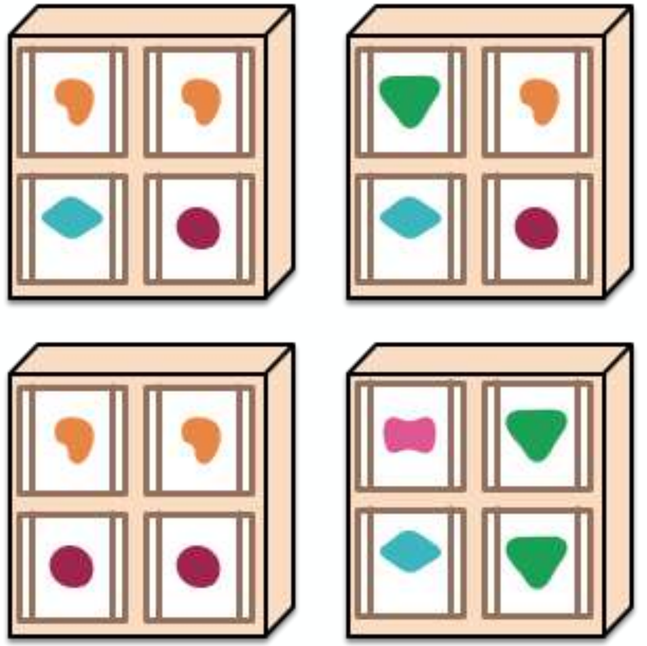
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.



Source: https://martinfowler.com/articles/microservices.html

Microservices best practices

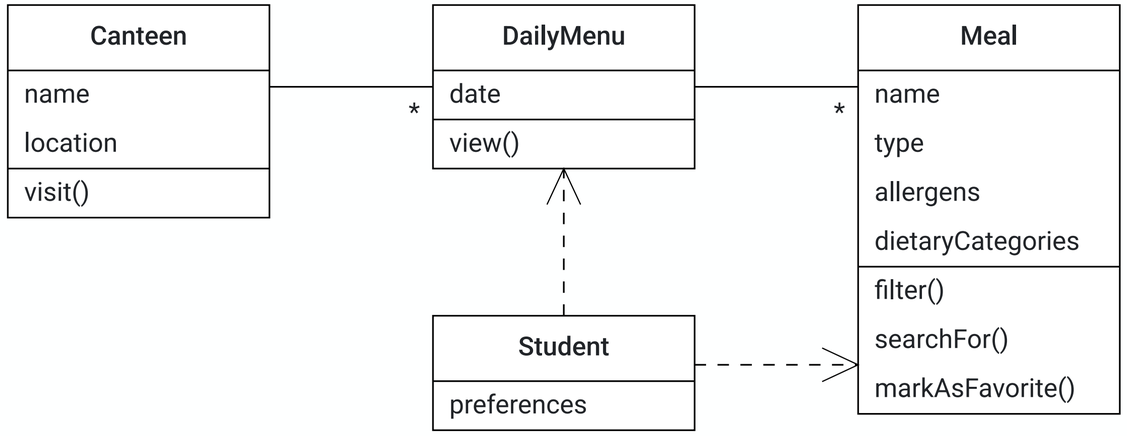
- Design services around business capabilities
- Automate deployment and monitoring
- Keep services loosely coupled
- Use asynchronous communication where possible
- Maintain clear APIs for interaction

Case study

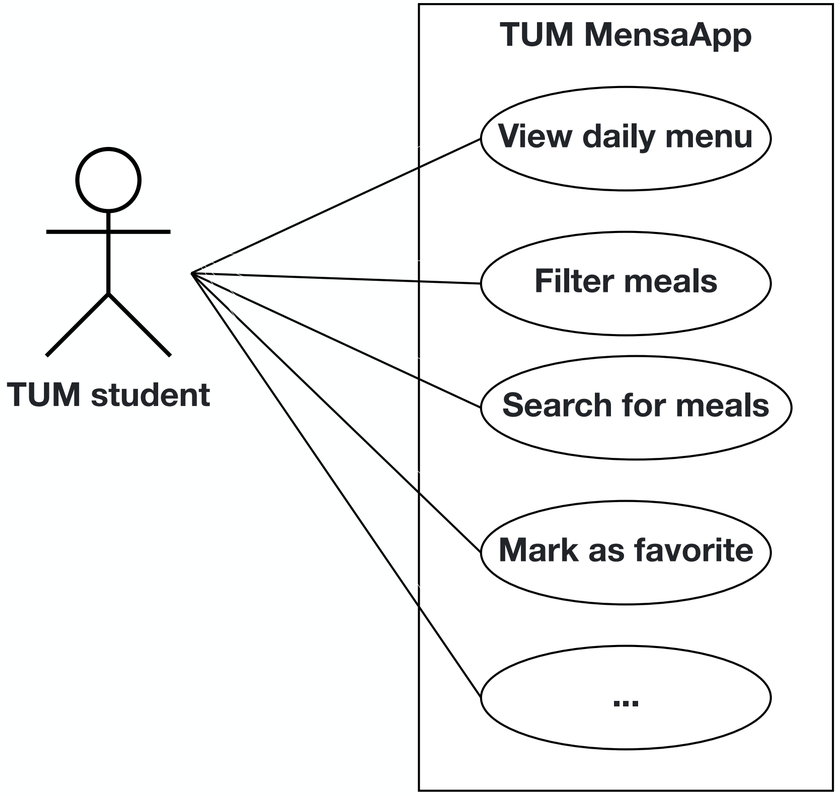
Extend TUM Mensa App to microservices

Existing requirements models

Analysis object model

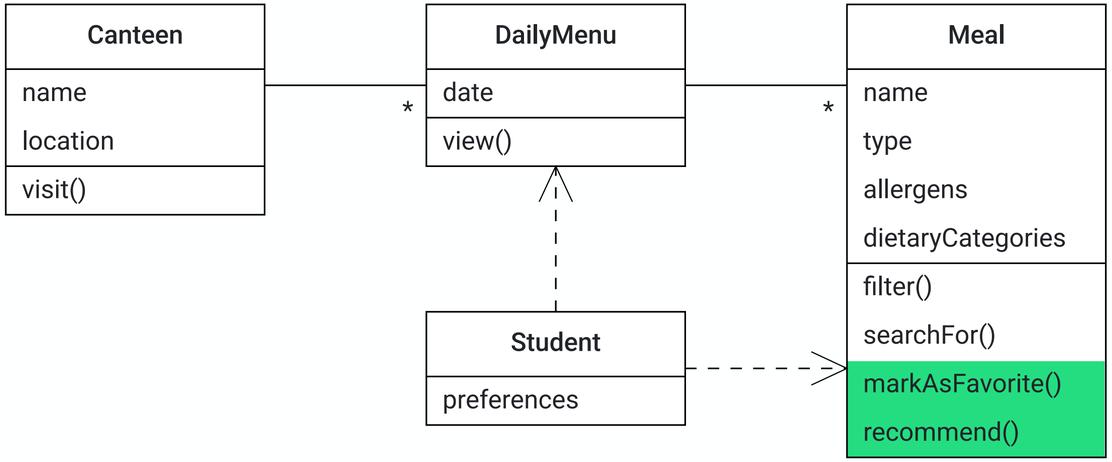


Use case model

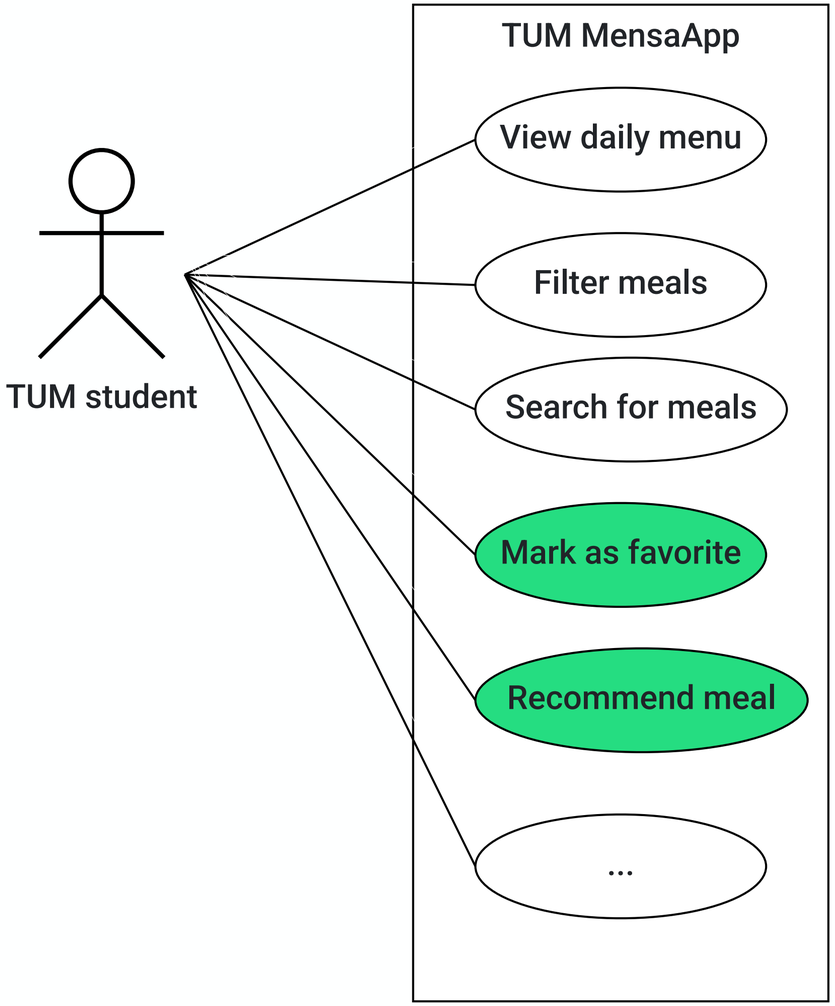


Extended requirements models

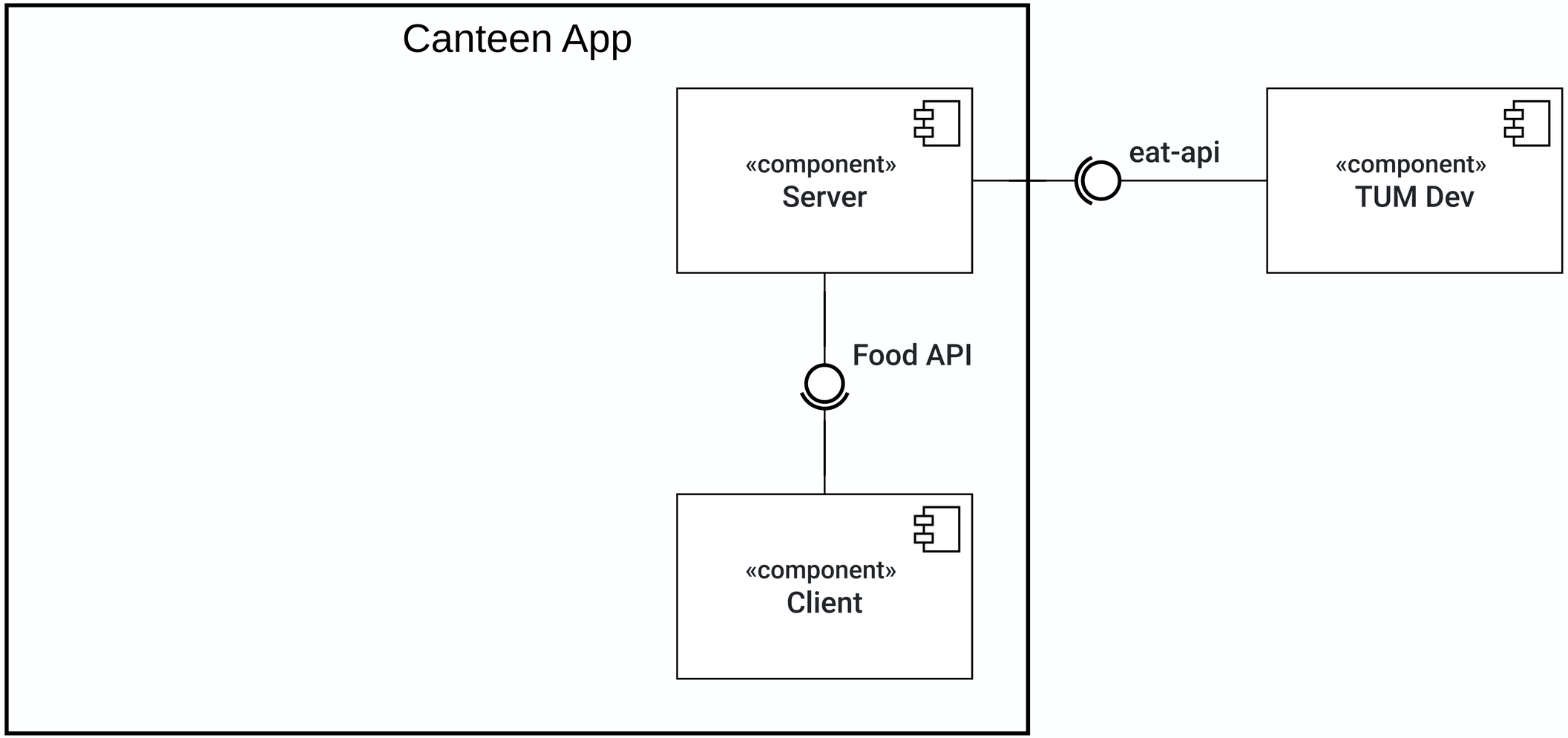
Analysis object model



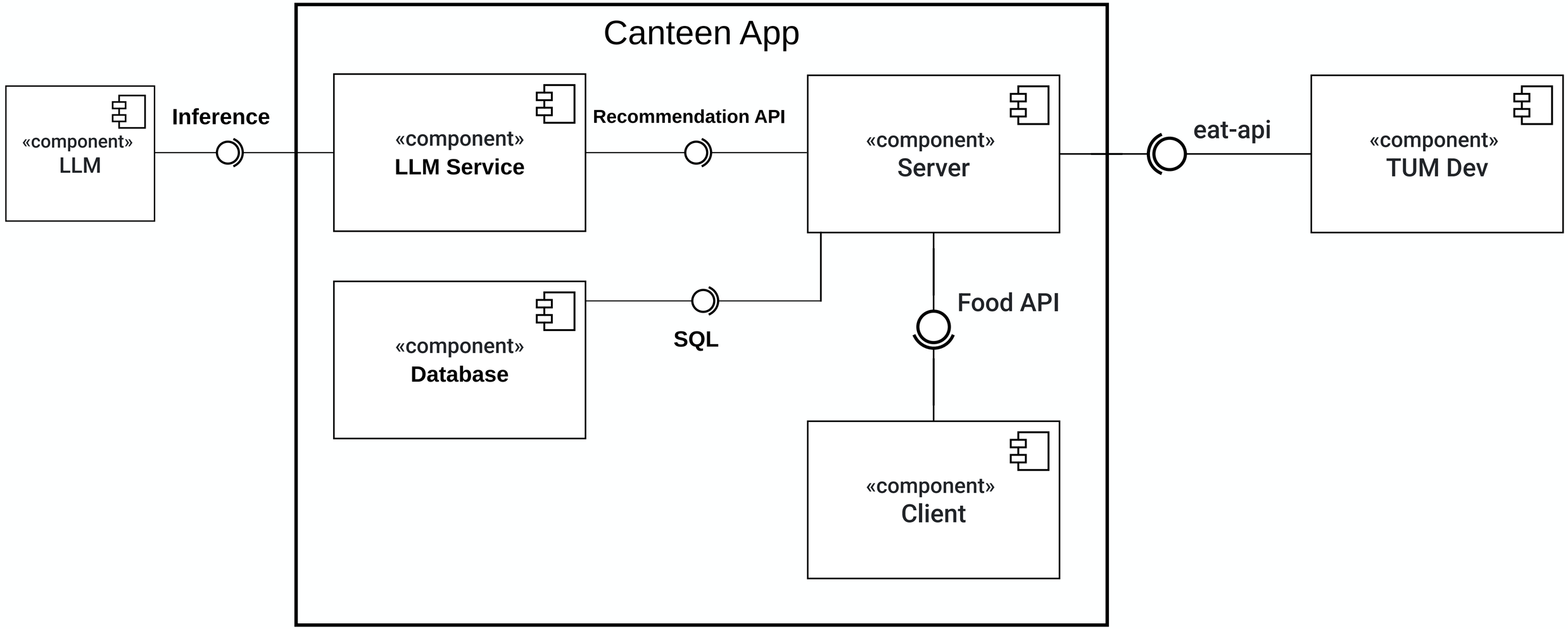
Use case model



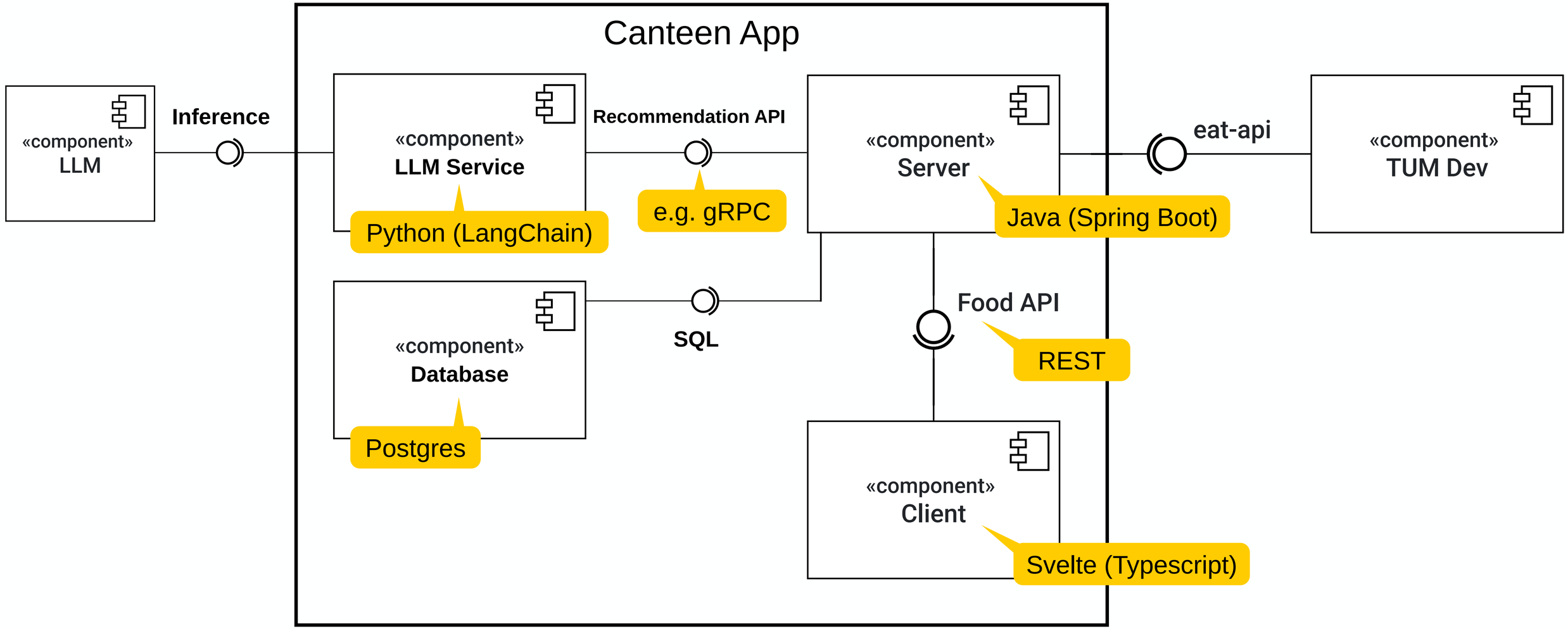
Existing architecture



Extended microservices architecture



Extended microservices architecture



API driven development

- **Define APIs first**: contract first development using OpenAPI/Swagger or gRPC
- **Mock early**: use mock servers before implementation
- **Design by interface**: clients and services interact via stable, versioned APIs
- **Iterate independently**: server and client can evolve in parallel
- Benefits
  - ✅ Clear separation of concerns (team autonomy)
  - ✅ Enables parallel development (client/server decoupling)
  - ✅ Reusable and composable services
  - ✅ Simplifies integration across teams and systems

OpenAPI

- A language agnostic specification for defining RESTful APIs
- Formerly known as Swagger specification
- Defines endpoints, request/response formats, parameters, authentication, and more
- Key features
  - Machine readable contract (`openapi.yaml` / `openapi.json`)
  - Enables API first development
  - Supports automatic code generation (clients, servers, SDKs)
  - Compatible with tools like Swagger UI, Postman, and Pact
- Sources
  - https://www.openapis.org
  - https://swagger.io/solutions/api-design

Example

chat.yaml (part 1)

```yaml
openapi: 3.0.3
info: { title: LLM Chat API, version: 1.0.0 }

paths:
  /chat:
    post:
      summary: Stream LLM output
      requestBody:
        required: true
        content:
          application/json:
            schema: { $ref: '#/components/schemas/ChatRequest' }
      responses:
        '200':
          description: Streamed response
          content:
            text/event-stream:
              schema: { $ref: '#/components/schemas/ChatResponse' }
```

chat.yaml (part 2)

```yaml
components:
  schemas:
    ChatRequest:
      type: object
      required: [user_id, prompt]
      properties:
        user_id: { type: string }
        prompt: { type: string }
        context:
          type: array
          items: { type: string }

    ChatResponse:
      type: object
      properties:
        message_chunk: { type: string }
        is_final: { type: boolean }
```

gRPC: High-Performance RPC with protocol buffers

- A language-neutral, platform-independent RPC framework by Google
- Uses Protocol Buffers (.proto files) for API contracts
- Supports HTTP/2, bidirectional streaming, and multiplexing
- Key components
  - 📄 .proto files define services, methods, request/response messages
  - 🧬 Code generation for clients/servers in multiple languages
  - 🔐 Built-in authentication, deadlines, and retries
- Sources
  - https://grpc.io/docs/what-is-grpc
  - https://grpc.io/docs/what-is-grpc/core-concepts

Why use gRPC?

- ⚡ Efficient: binary format (smaller/faster than JSON)
- 📡 Streaming support: client, server, and bidirectional
- 🤝 Strong typing and schema enforcement
- 🔄 Inter-service communication in microservice architectures
- **Challenges and adoption barriers**
  - 🌐 Limited browser support
  - 🛠️ Steeper learning curve
  - 🔧 More difficult debugging and inspection
  - 🔄 No standardized versioning
  - ⚙️ Fewer tools for mocking and testing

Requires gRPC-Web or fallback REST APIs

Example

chat.proto

```
syntax = "proto3";

package llmchat;

service ChatService {
  // Sends a user message and streams back the LLM-generated response
  rpc StreamChat (ChatRequest) returns (stream ChatResponse);
}

// Request message containing the user prompt
message ChatRequest {
  string user_id = 1;
  string prompt = 2;
  repeated string context = 3; // optional: past messages or system instructions
}

// Streamed response containing a chunk of the answer
message ChatResponse {
  string message_chunk = 1;
  bool is_final = 2; // optional: signals end of generation
}
```

🧭 Two typical workflows in API driven development

1. Server first (code first) workflow

- ✅ Start by implementing the service manually
- ➡️ Use tools to generate the OpenAPI/gRPC spec
- ➡️ Then generate client code from that spec
- 🔧 Common in fast moving teams with tight server control or when migration towards API driven development

2. Spec first (contract first) workflow

- ✅ Start by writing the API spec manually (OpenAPI or .proto)
- ➡️ Generate server stubs and client code from the contract
- 📐 Enforces alignment and supports parallel team development
- 🔧 Common for new projects

Exercise **W07E01**: API design

5 min

- Let’s model the **Recommendation API** using gRPC
- Create a proto file in your favorite text editor based on the previous example
- Collaborate with your neighbor(s) and justify the design (rationale)

Example solution

meal_recommendation.proto

```
syntax = "proto3";

package tum.mensa;

service MealRecommendationService {
  // Recommend a meal for the user based on today's menu and previous favorites
  rpc RecommendMeal (MealRecommendationRequest) returns (MealRecommendationResponse);
}

message MealRecommendationRequest {
  string user_id = 1;
  repeated string favorite_meals = 2;
  repeated MenuMeal today_menu = 3;
}

message MenuMeal {
  string name = 1;
  string description = 2;
  repeated string tags = 3; // e.g. ["vegetarian", "low-carb", "asian"]
}

message MealRecommendationResponse {
  string recommended_meal_name = 1;
  string reasoning = 2; // Optional: explain why this meal was selected
}
```

Design rationale

- **One shot RPC**: no need for streaming here; the Spring Boot service sends one request, receives one response
- **Extensible**: `MenuMeal` includes tags for dietary or semantic enrichment; easy for LLMs to reason over
- **Explainability**: reasoning field supports transparency (optional for user interface or logging)
- **Portable**: the Python LLM service (e.g., using Langchain with gRPC Python) can directly consume this and map to prompts

Break

10 min

The workshop will continue at **13:10**

Feedback survey

10 min

- We want to evaluate the course early and improve it for the 2nd half of the semester based on your valuable feedback
- Please fill out the anonymous survey now
- You should have received an invitation via email

Exercise W07E02

10 min

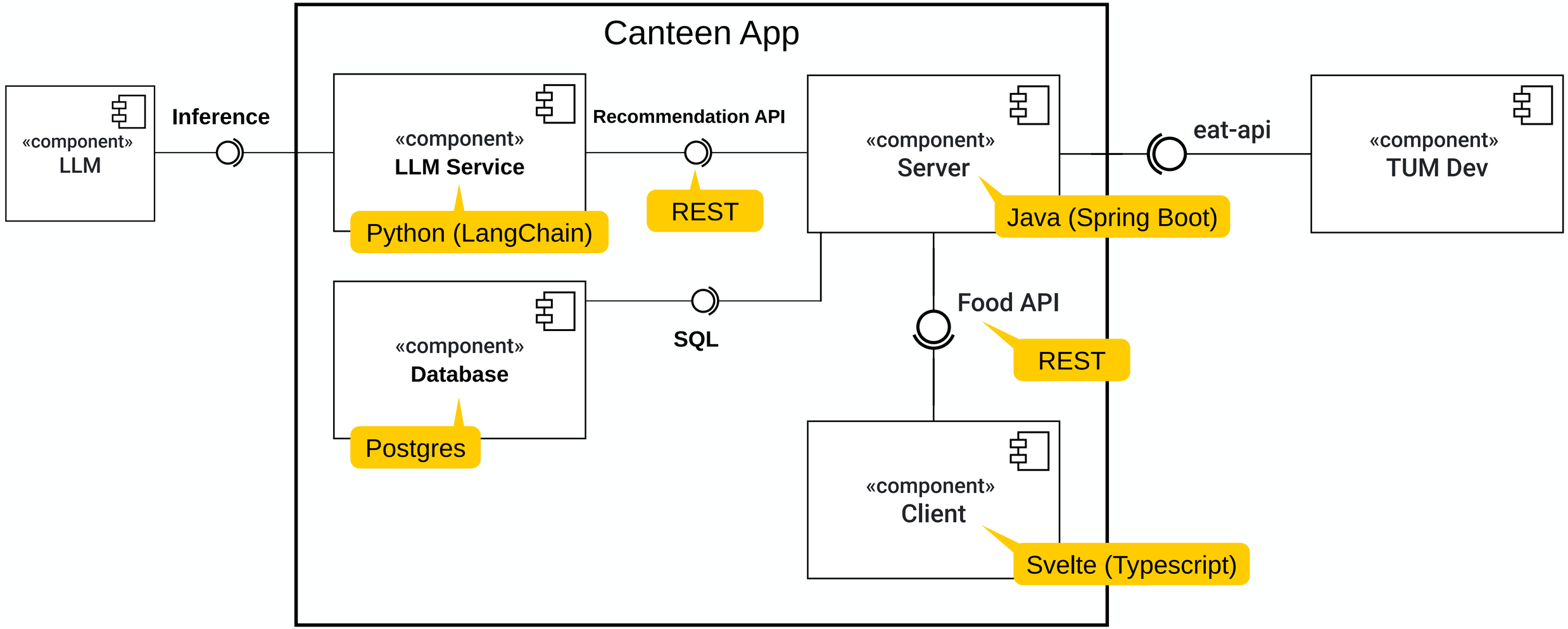
- Github repo: https://github.com/AET-DevOps25/w07-template
- New features
  - Mark as favorite (already implemented)
  - Recommend today's meal (implementation started)
- **Your task: finish the implementation**
  1. Clone the repository and open it in your favorite IDE
  2. **Python** microservice: use Langchain to generate a recommendation
  3. **Java** microservice: invoke the provided REST **Recommendation API** in **`LLMRestClient`**
  4. **Java** microservice: implement the business logic in **`LLMRecommendationService`**
  5. **Java** microservice: finish the implementation of the REST controller so that the client can invoke it

Provide **Recommendation API**

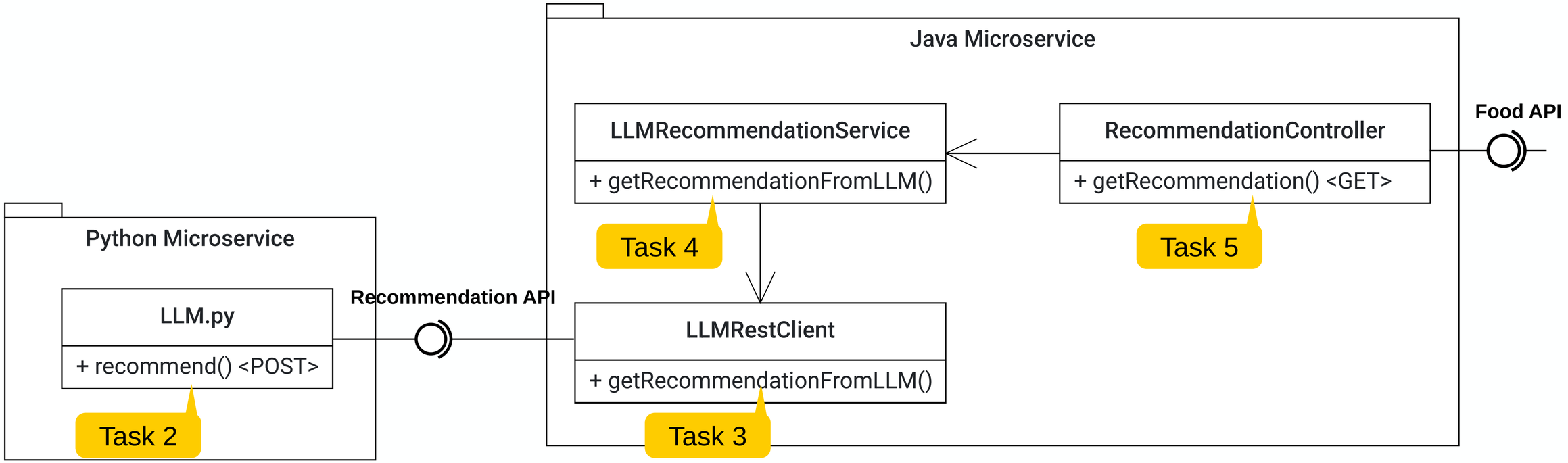
Consume **Recommendation API**

Extend the provided **Food API**

Reminder: top level architecture



Detailed design



Optional challenges

1. Use the specified gRPC API from W07E01 and implement the communication between the Java micro service and the python micro service in gRPC
2. Generate the server (python) stub and the client (Java) code for the gRPC API to avoid implementing it manually

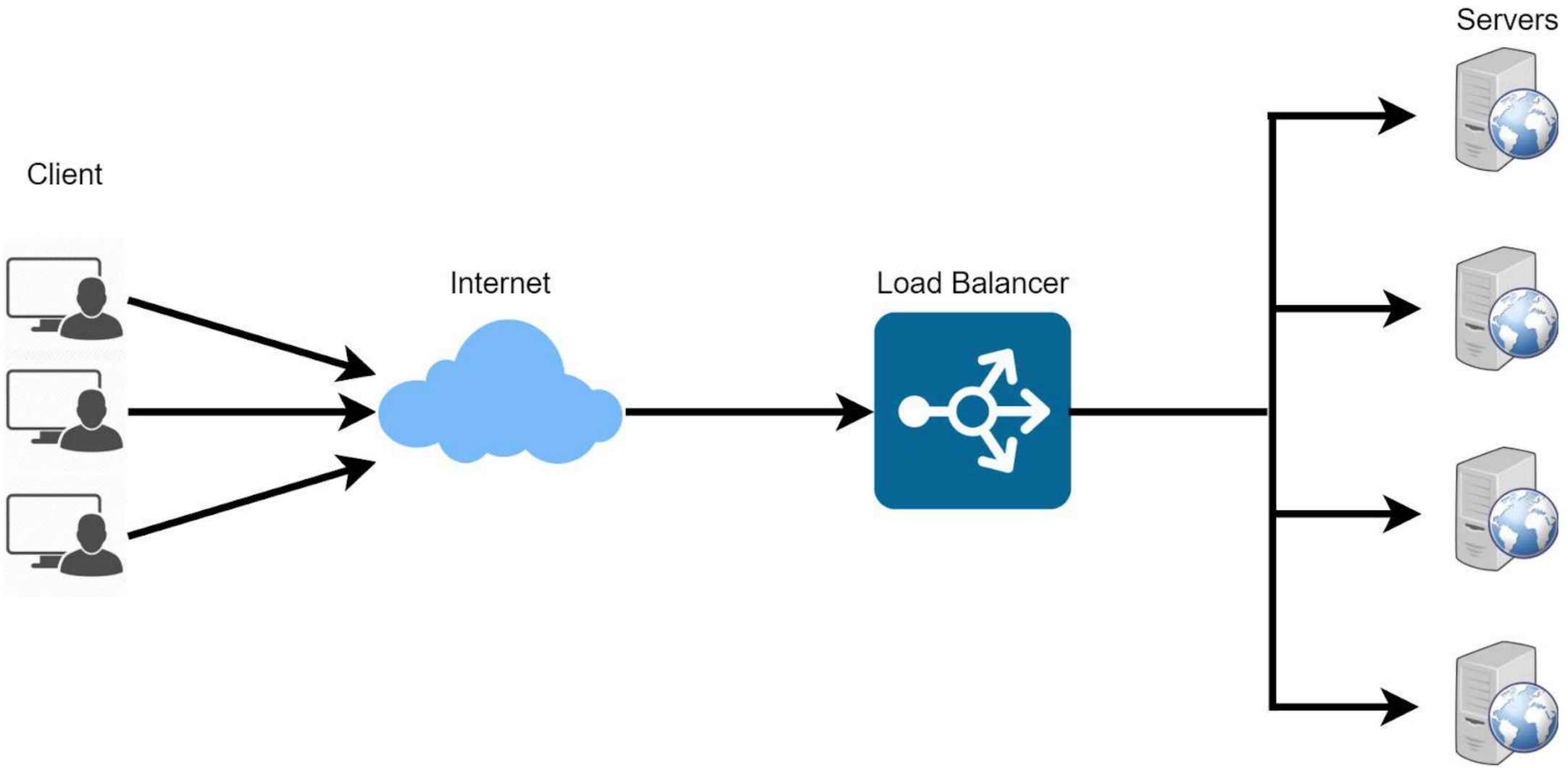
✓ If you participate: send your solution to your tutor for evaluation

Example solution

Outline

- Microservices design principles
- **Scaling and load balancing**
- Service discovery and API gateways
- Service meshes

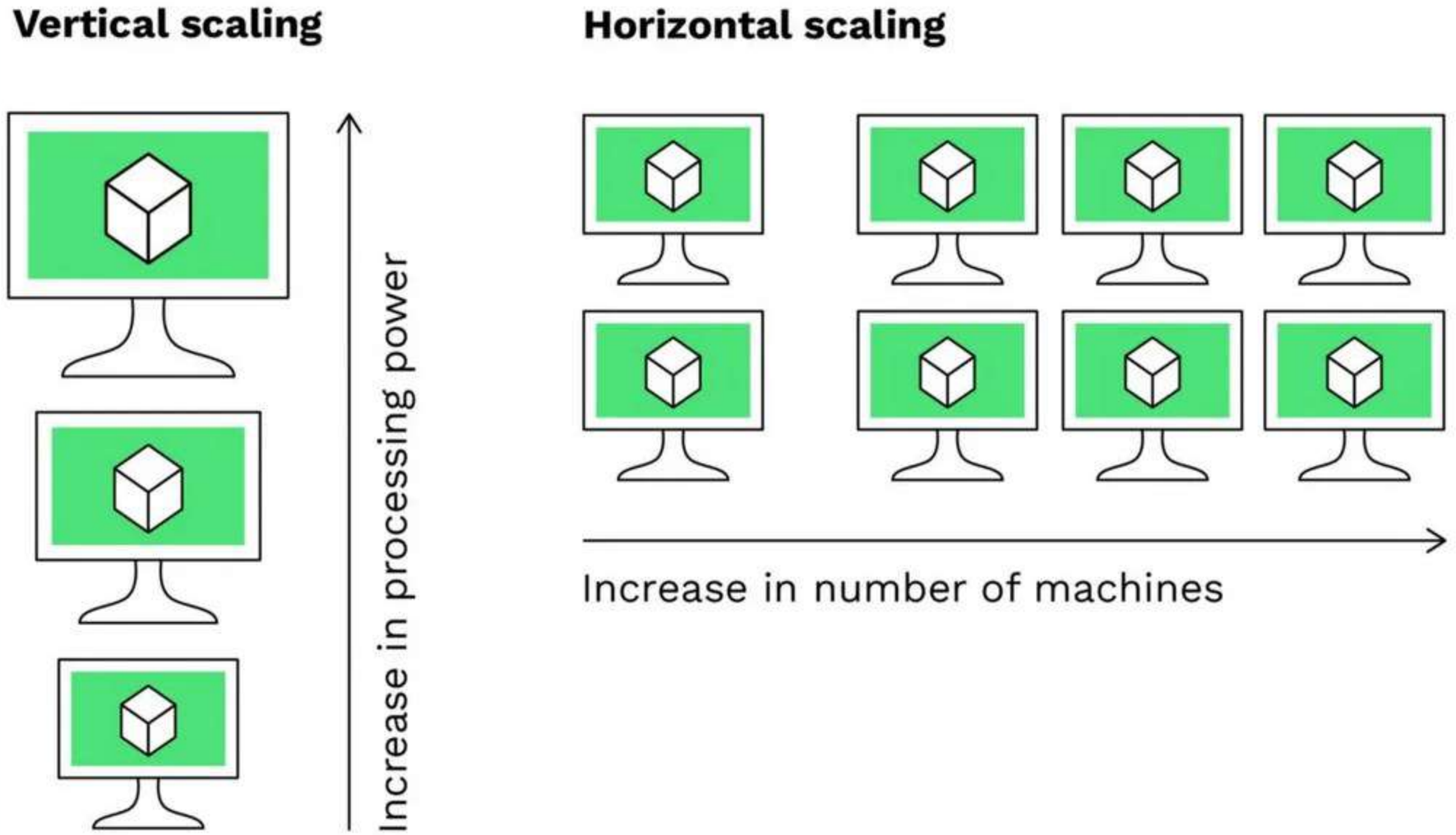
Load balancer



Load balancer

- **Distributes incoming network traffic** across multiple service instances to ensure reliability, scalability, and performance
- Decouples **clients** from **instances** — service discovery + abstraction
- Enables **horizontal scaling** for stateless services
- Improves **fault tolerance** by routing around failed instances
- **Automates routing** in CI/CD environments
  (e.g. blue/green or canary deployments)
- Integrates with **service discovery** (e.g. via Kubernetes, Docker, Consul)
- Works with **observability tools** to monitor service health and latency

Horizontal vs. vertical scaling



Source: https://medium.com/javarevisited/difference-between-horizontal-scalability-vs-vertical-scalability-67455efc91c

Example: nginx

```
http {
    upstream myapp1 {
        server srv1.example.com;
        server srv2.example.com;
        server srv3.example.com;
    }

    server {
        listen 80;

        location / {
            proxy_pass http://myapp1;
        }
    }
}
```

Example: Traefik

```yaml
http:
  routers:
    myapp-router:
      rule: "PathPrefix(`/`)"
      entryPoints:
        - web
      service: myapp-service

  services:
    myapp-service:
      loadBalancer:
        servers:
          - url: "http://srv1.example.com"
          - url: "http://srv2.example.com"
          - url: "http://srv3.example.com"
```

Example: Traefik (integrated in docker compose)

```yaml
services:
  traefik:
    image: traefik:v3.0
    command:
      - --entrypoints.web.address=:80
      - --providers.docker=true
    ports:
      - "80:80"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock

  myapp:
    image: your-app-image
    deploy:
      replicas: 3
    labels:
      - "traefik.http.routers.myapp.rule=PathPrefix(`/`)"
      - "traefik.http.routers.myapp.entrypoints=web"
      - "traefik.http.services.myapp.loadbalancer.server.port=80"
```

Advantage: you can configure everything here

```
docker-compose up --scale myapp=3
```

Load balancer comparison

Feature	Nginx	Traefik
Needs config file	✅ Always	❌ Optional Alternative: use labels
Docker service discovery	❌ No	✅ Yes
Auto reconfig on scale changes	❌ No	✅ Yes
Suitable for static setups	✅ Yes	✅ Yes
Best for dynamic/ microservice setups	⚠️ Workaround needed	✅ Native support

Load balancing strategies

- **Round robin**: simple rotation
- **Least connections**: choose server with fewest active requests
- **IP hash / sticky sessions**: keep session affinity
  - Can cause uneven distribution
  - Required for canary deployment to ensure a consistent user experience
- **Weighted**: favor stronger instances (used in canary or blue/green rollouts)

Health checks and fault tolerance

- Regular **health checks** ensure traffic goes only to healthy instances
- Auto remove **crashed or unresponsive services** from the pool
- Combine with **circuit breakers** and **retry policies** for robustness
  - **Circuit breaker**: temporarily stops requests to a failing service to prevent cascading failures and allow recovery
  - **Retry policy**: automatically re-attempts failed requests a limited number of times before giving up

Observability and metrics

- Track request distribution, latency, error rates, and instance health
- Essential for debugging, alerting, and **capacity planning**
- Integrates with Prometheus, Grafana, ELK stack, Datadog, etc

Security and access control

- Load balancers can act as a **security layer**
- ✅ TLS termination: offload TLS at the load balancer to simplify service configuration
  - Use **strong ciphers**, TLS 1.2+ (disable TLS 1.0/1.1), and perfect forward secrecy (PFS)
  - Prefer ECDHE over RSA key exchange
- 🔒 Certificates
  - Use trusted CA-issued certificates (or ACME / let’s encrypt automation)
  - Prefer ECDSA certificates (smaller, faster) where supported
  - Enforce OCSP stapling, HSTS, and SNI for security and performance

Security and access control

- 🌐 Protocol and compression
  - Enable http/3 (quic) for faster and secure UDP-based connections
  - Use gzip or brotli with proper MIME type filtering for faster transfer and to prevent attacks
  - Disable legacy protocols (e.g. SSLv3, early TLS, weak DH groups)
- Allows **IP whitelisting**, **GeoIP blocking**, or **rate limiting** at the edge
- Can be combined with JWT/OAuth validation via API gateway layer
- Helps enforce **zero trust** architecture principles

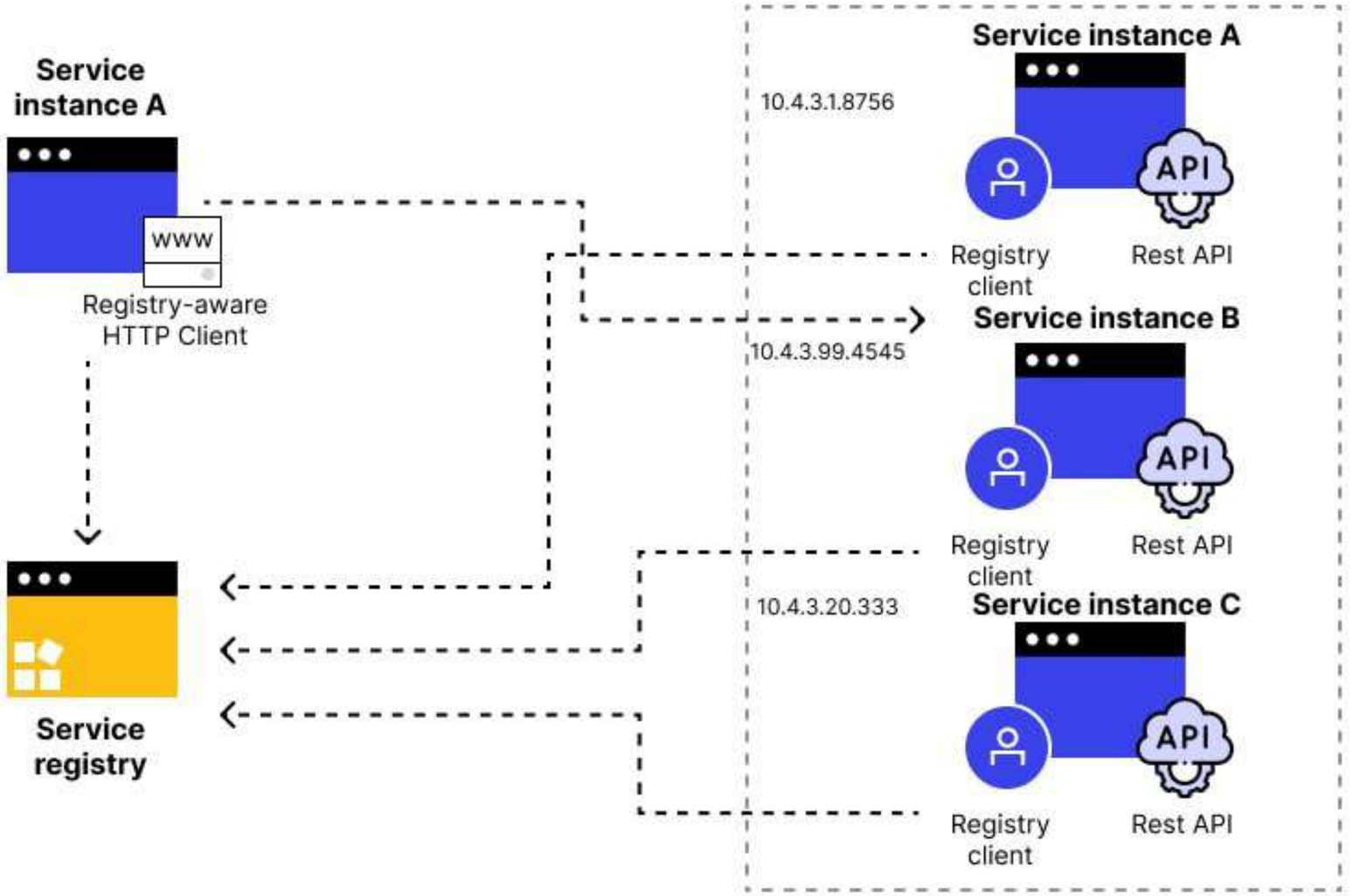
Outline

- Microservices design principles
- Scaling and load balancing
- **Service discovery and API gateways**
- Service meshes

Service discovery in microservices

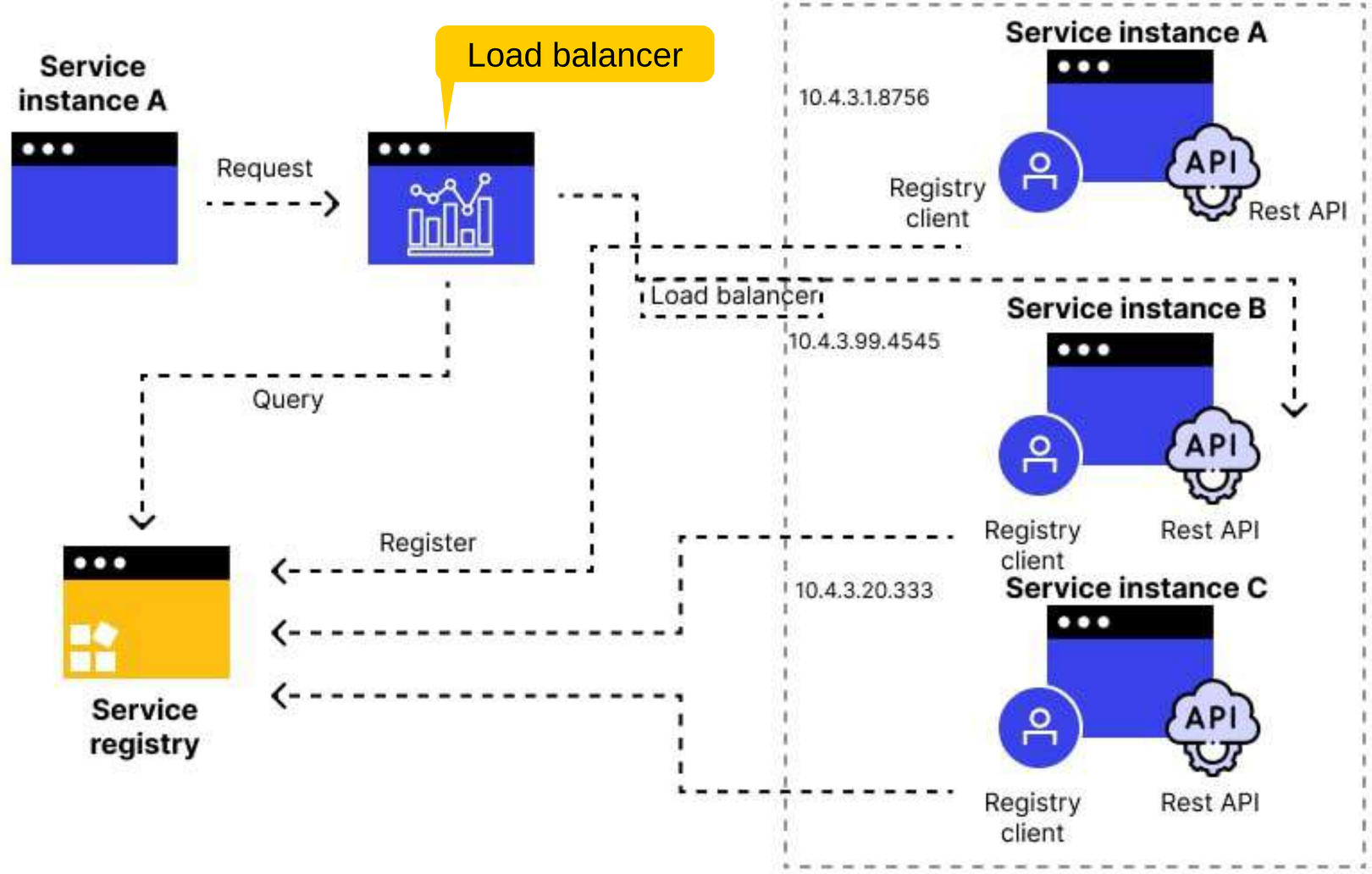
- 🔍 Dynamic mechanism to find service instances at runtime
- 🧩 Why it matters
  - Microservices scale up/down independently
  - IPs and ports are ephemeral (especially in containers)
  - Manual configuration is brittle and unscalable
- ⚙️ How it works
  1. **Client side discovery**: service fetches instance list (e.g. Netflix Ribbon)
  2. **Server side discovery**: load balancer queries registry (e.g. Traefik, Kubernetes)
- 📚 Examples
  - **Service registry**: Consul, etcd, Eureka
  - **Orchestrator native**: Kubernetes DNS, AWS Cloud Map

Client side service discovery



Source: https://www.wallarm.com/what/what-is-a-service-discovery

Server side service discovery



Source: https://www.wallarm.com/what/what-is-a-service-discovery

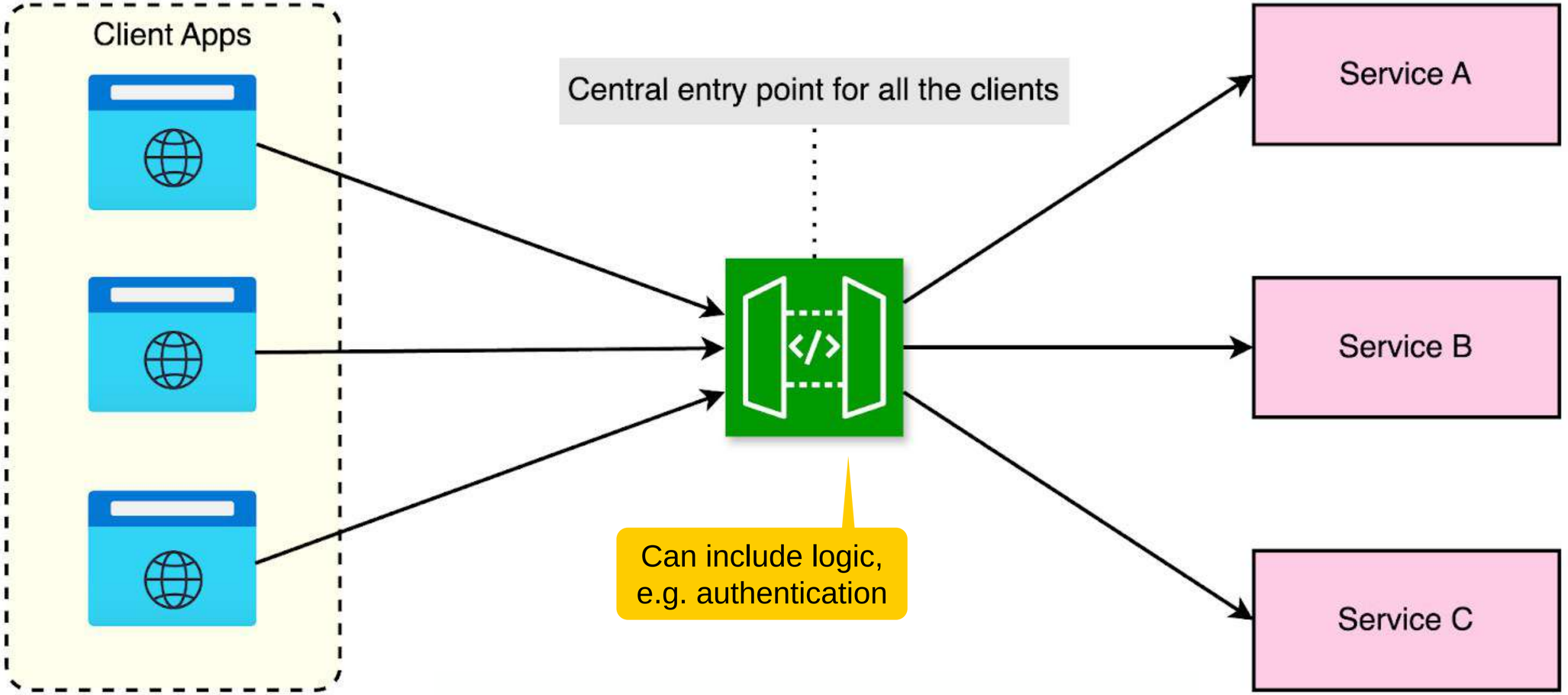
Service registration

- 📝 The act of making a service discoverable by adding it to a registry
- 🔄 Two modes
  1. **Self registration**: service registers itself (e.g. Spring Boot + Eureka)
  2. **Third party registration**: orchestrator/sidecar handles it (e.g. Kubernetes, Istio)
- ✅ Why it is important
  - Enables automation and dynamic scaling
  - Keeps registry updated with health status
  - Supports load balancing and routing
- 🛠️ Tools
  - Registry: Consul, etcd, Eureka
  - Auto registration: Kubernetes kubelet, Envoy sidecar

API gateway

- 🚪 Central entry point for client requests to a microservice architecture
- 🔐 Responsibilities
  - Request routing and load balancing
  - Authentication and rate limiting
  - Aggregating responses from multiple services
- 🌍 Use cases
  - Simplifies client to service interaction
  - Centralizes cross-cutting concerns (security, observability)
  - Supports blue/green, canary deployments, and versioning
- 💡 Popular gateways
  - **Open source**: Kong, Ambassador, Traefik
  - **Cloud**: AWS API Gateway, Azure API Management

API gateway



Source: https://blog.bytebytego.com/p/api-gateway

Outline

- Microservices design principles
- Scaling and load balancing
- Service discovery and API gateways
- **Service meshes**

Service mesh

- 🔍 Infrastructure layer that manages **service to service** communication in a microservices architecture
- 🧠 **Core idea**: move logic like routing, resilience, and authentication out of the application — into the infrastructure
- **Sidecar** (proxy): separate process or container deployed alongside a service to provide supporting functionality like proxying, logging, or security
- 🧩 **Key features**
  - Transparent traffic management
  - Observability (metrics, logs, traces)
  - Built-in security (mTLS*, policies)
  - Retries, circuit breakers, rate limiting

The sidecar does **not** change the service itself

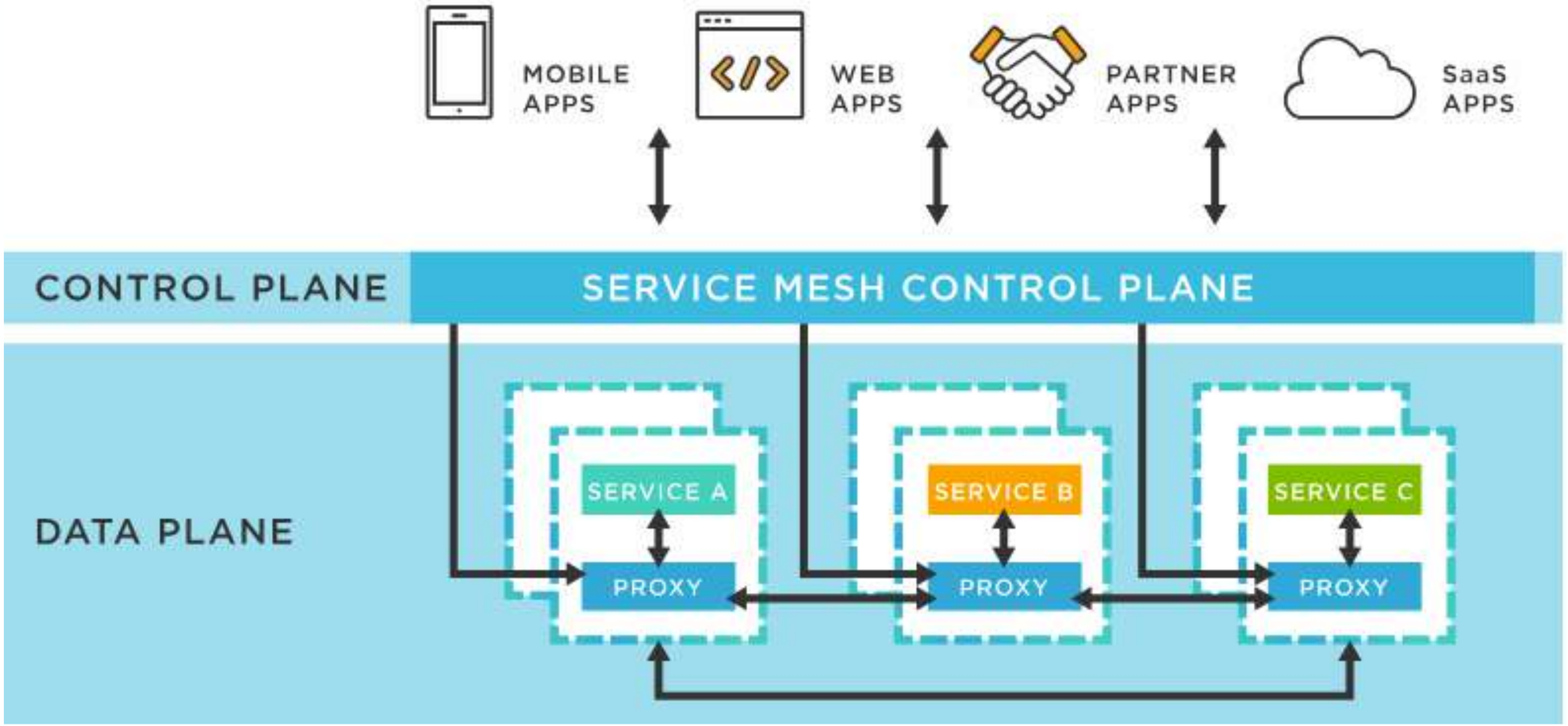
* **mTLS** (Mutual TLS) is a security protocol where both the client and the server authenticate each other using digital certificates, ensuring encrypted and trusted communication on both ends

How it works

- 🧱 Architecture
  - **Data plane**: lightweight sidecar proxies (e.g. Envoy) next to each service
  - **Control plane**: centralized config and orchestration (e.g. Istio, Linkerd)
- 🔁 Example flow
  1. Service A → Sidecar → Network → Sidecar → Service B
  2. All policies, retries, and security enforced at the proxy layer
- 🔧 Common meshes

Mesh	Data plane	Control plane
Istio	Envoy	Istiod
Linkerd	Linkerd2-proxy	Linkerd control plane
Consul	Envoy	Consul connect

Service mesh



Source: https://www.linkedin.com/pulse/understanding-microservice-meshes-architecture-luis-soares-m-sc-

Benefits and Challenges

- 🚀 **Benefits**
  - Consistent policies across services
  - Zero trust networking with mTLS
  - Real time telemetry for debugging and service level objectives
  - Enables progressive delivery (e.g. canaries, A/B)
- 🧪 **DevOps advantages**
  - Decouples developers from traffic handling logic
  - Reduces need for custom resilience code
  - Simplifies failure injection, observability, security audits
- ⚠️ **Challenges**
  - Added complexity and latency
  - Requires operational maturity (e.g. versioning, config drift)
  - Higher resource overhead in clusters

Example: 99.99% availability of the payment service per quarter

Service mesh vs. API gateway vs. load balancer

Feature	Load balancer	API gateway	Service mesh
Entry point	External traffic	External traffic	Internal traffic
Purpose	Distribute load	Expose APIs	Manage internal service to service
Authentication, rate limit	❌ (basic only)	✅	✅
mTLS*, retries	❌	❌	✅
Visibility	Limited logs	Request logs	Full metrics / traces

* **mTLS** (Mutual TLS) is a security protocol where both the client and the server authenticate each other using digital certificates, ensuring encrypted and trusted communication on both ends

Next steps

- Project work
  - Set up microservices
    (at least one using Spring Boot and at least one using Python for LLM inference)
  - Set up API gateways or load balancing and one or multiple databases
  - Apply API driven development from now on
- Read the following
  - **Microservices**: https://microservices.io/patterns/microservices.html and https://martinfowler.com/articles/microservices.html
  - **API driven development**: https://medium.com/@konstde00/guide-to-api-first-development-with-spring-boot-and-openapi-543b2a431a07

→ Deadline: **Friday, June 13, 11:00**

Summary

- 🧱 **Microservices** enable modular, scalable systems
- 🔍 **Service discovery** and **API gateways** enable dynamic routing
- 🚀 **Load balancing** and **scaling** ensure performance and resilience
- 🔗 **Service meshes** provide fine grained traffic control and security
- ⚖️ **Decentralized data** and **eventual consistency** are becoming the norm

References

- https://microservices.io/patterns/microservices.html
- https://martinfowler.com/articles/microservices.html
- https://blog.flexbase.in/digital-transformation-to-microservices-the-approach
- https://swagger.io/solutions/api-design
- https://www.openapis.org
- https://grpc.io/docs/what-is-grpc
- https://grpc.io/docs/what-is-grpc/core-concepts
- https://www.wallarm.com/what/what-is-a-service-discovery
- https://blog.bytebytego.com/p/api-gateway
- https://medium.com/javarevisited/difference-between-horizontal-scalability-vs-vertical-scalability-67455efc91c
- https://www.linkedin.com/pulse/understanding-microservice-meshes-architecture-luis-soares-m-sc-
- https://github.com/GoogleCloudPlatform/microservices-demo

Example solution

Generate recommendation (1)

- **main.py**

```
# Format arrays as comma-separated strings for better processing
favorite_meals_str = ", ".join(req.favorite_menu)
todays_meals_str = ", ".join(req.todays_menu)

# Use the chain to generate recommendation
recommendation = recommendation_chain.invoke({
    "favorite_menu": favorite_meals_str,
    "todays_menu": todays_meals_str
})

return RecommendResponse(recommendation=recommendation)
```

Use the following gist to copy and paste code: https://gist.github.com/robertjndw/92f7b1a5a8818e0244fa99f4f6069b39

Generate recommendation (2)

- **LLMRestClient.java**

```
// Create request body
RecommendRequest request = new RecommendRequest(favoriteMenu, todaysMenu);

// Make REST call
RecommendResponse response = restClient.post()
    .uri("/recommend")
    .contentType(MediaType.APPLICATION_JSON)
    .body(request)
    .retrieve()
    .body(RecommendResponse.class);

// Extract the recommendation
return response != null ? response.recommendation() : "";
```

Use the following gist to copy and paste code: https://gist.github.com/robertjndw/92f7b1a5a8818e0244fa99f4f6069b39

Generate recommendation (3)

- **LLMRecommendationService.java**

```
public String getRecommendationFromLLM(List<String> favoriteMeals,
List<Dish> todayMeals) {
  try {
    // Convert today's dishes to meal names
    List<String> todayMealNames = todayMeals.stream()
      .map(Dish::name)
      .collect(Collectors.toList());

    // TODO Call REST service
    return llmRestClient.generateRecommendations(favoriteMeals, todayMealNames);
  }
  ...
}
```

Use the following gist to copy and paste code: https://gist.github.com/robertjndw/92f7b1a5a8818e0244fa99f4f6069b39

Generate recommendation (4)

- **RecommendationController.java**

```
// Get the today's menu from canteenService
List<Dish> todaysMeals = canteenService.getTodayMeals("mensa-garching");

// Call llmRecommendationService to get recommendation based on the user's favorites
String responseFromLLMService = llmRecommendationService.getRecommendationFromLLM(
  userPreferences.getFavoriteMeals(),
  todaysMeals);

if (responseFromLLMService.isEmpty()) {
  return ResponseEntity.noContent().build();
}

return ResponseEntity.ok(Map.of("recommendation", responseFromLLMService));
```

Use the following gist to copy and paste code: https://gist.github.com/robertjndw/92f7b1a5a8818e0244fa99f4f6069b39

Test your implementation

1. Open a terminal session in **`w07-template`**
2. Export the Chair LLM Key: `export CHAIR_API_KEY=<your-key>`
3. Build images locally with docker compose

```
docker compose build
```

Don’t pull them put use your changes

4. Run the apps with docker compose

```
docker compose up -d
```

Spawn the containers in the detached mode

5. Inspect the running application on http://localhost:3000
6. Stop the containers and remove the images

```
docker compose down
```

Use the following gist to copy and paste code: https://gist.github.com/robertjndw/92f7b1a5a8818e0244fa99f4f6069b39