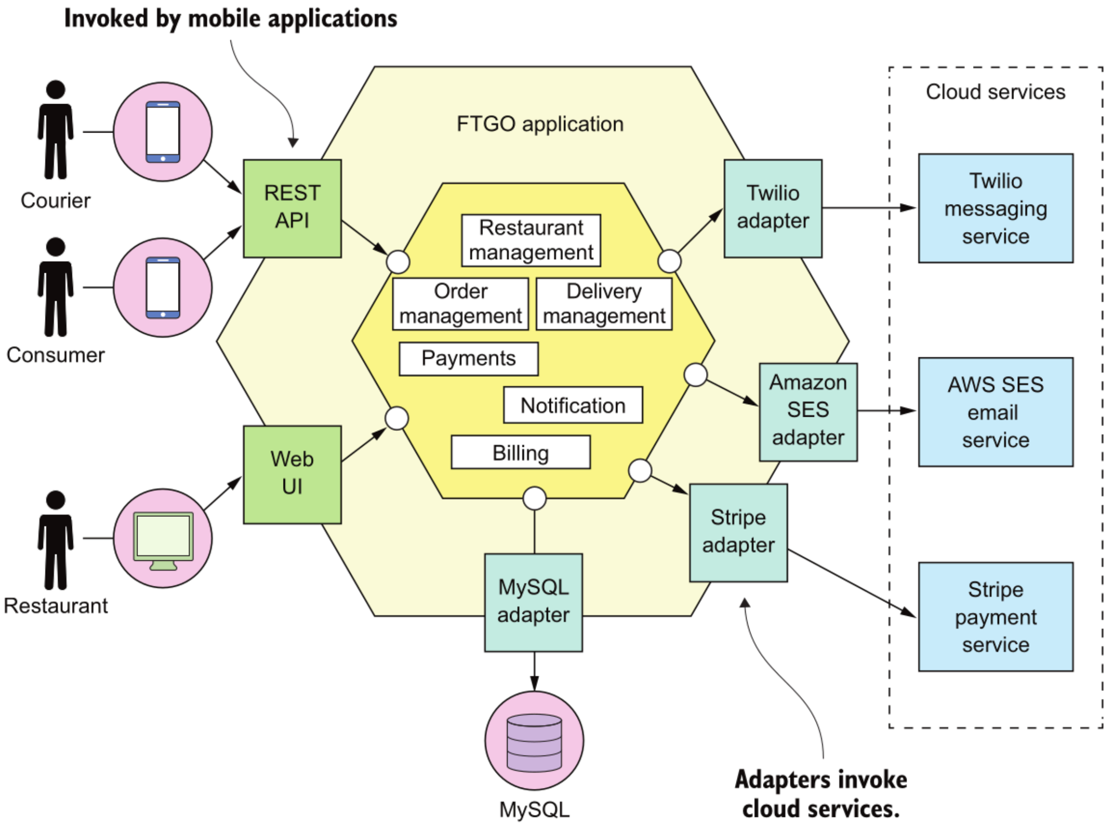
Escaping monolithic hell

The architecture of the Food to Go, Inc. (`FTGO`) application

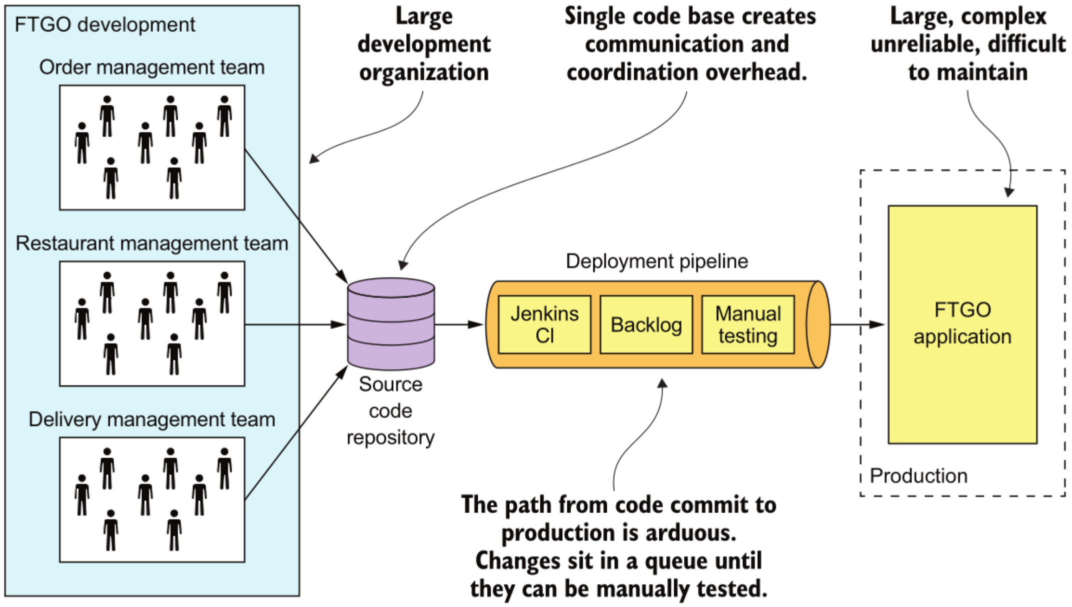


The benefits of the monolithic architecture

1. Simple to develop

2. Easy to make radical changes to the application
3. Straightforward to test
4. Straightforward to deploy
5. Easy to scale (but required big infra)

Living in monolithic hell

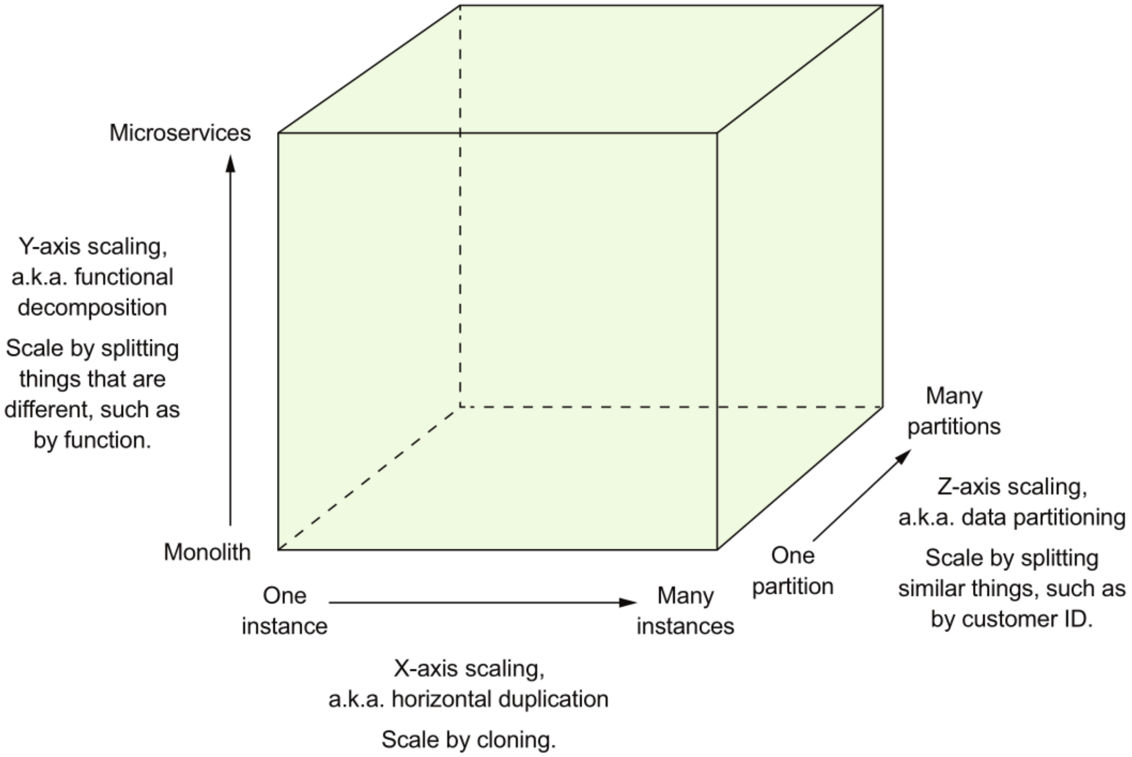

FTGO development
Order management team
Restaurant management team
Delivery management team
Large development organization
Single code base creates communication and coordination overhead.
Large, complex unreliable, difficult to maintain
Deployment pipeline
Jenkins CI
Backlog
Manual testing
Source code repository
FTGO application
Production
The path from code commit to production is arduous. Changes sit in a queue until they can be manually tested.

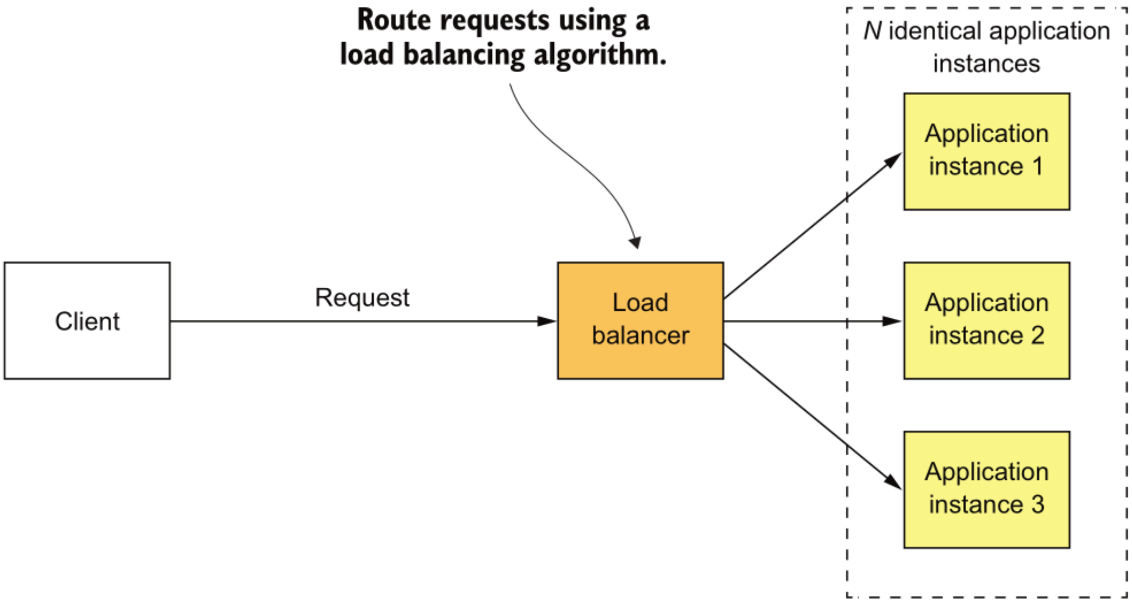
1. COMPLEXITY INTIMIDATES DEVELOPERS
2. DEVELOPMENT IS SLOW
3. PATH FROM COMMIT TO DEPLOYMENT IS LONG AND ARDUOUS

4. SCALING IS DIFFICULT
5. DELIVERING A RELIABLE MONOLITH IS CHALLENGING
6. **LOCKED INTO INCREASINGLY OBSOLETE TECHNOLOGY STACK**

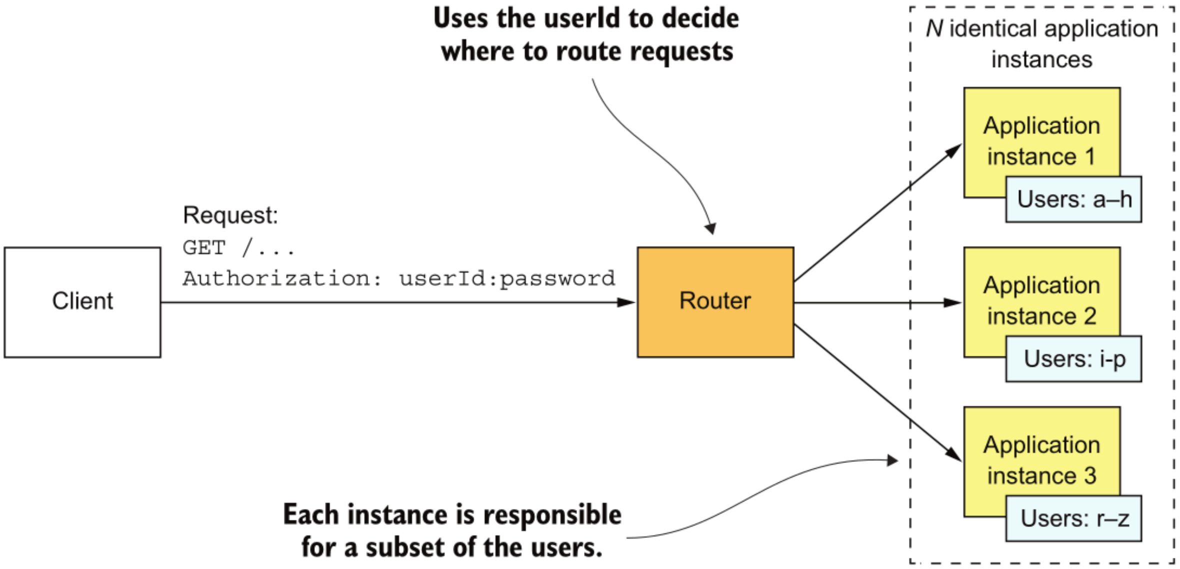
`Microservice` architecture to the rescue

`Scale-cube and microservices`

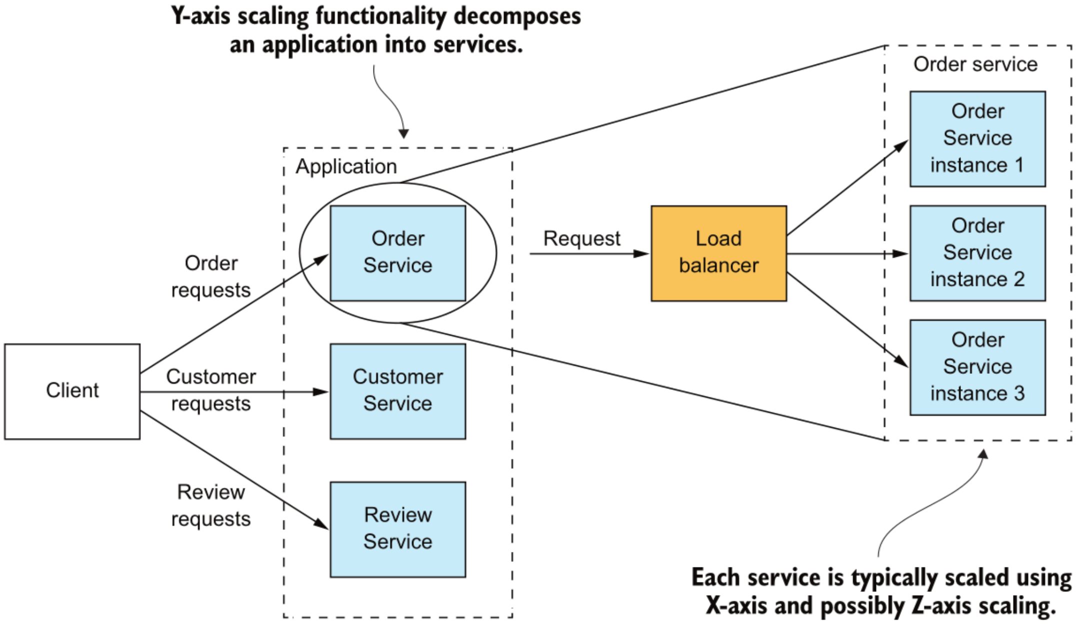

Microservices
Y-axis scaling, a.k.a. functional decomposition
Scale by splitting things that are different, such as by function.
Monolith
One instance
Many instances
X-axis scaling, a.k.a. horizontal duplication
Scale by cloning.
One partition
Many partitions
Z-axis scaling, a.k.a. data partitioning
Scale by splitting similar things, such as by customer ID.



X-axis scaling runs multiple, identical instances of the **monolithic** application behind a `load balancer.`

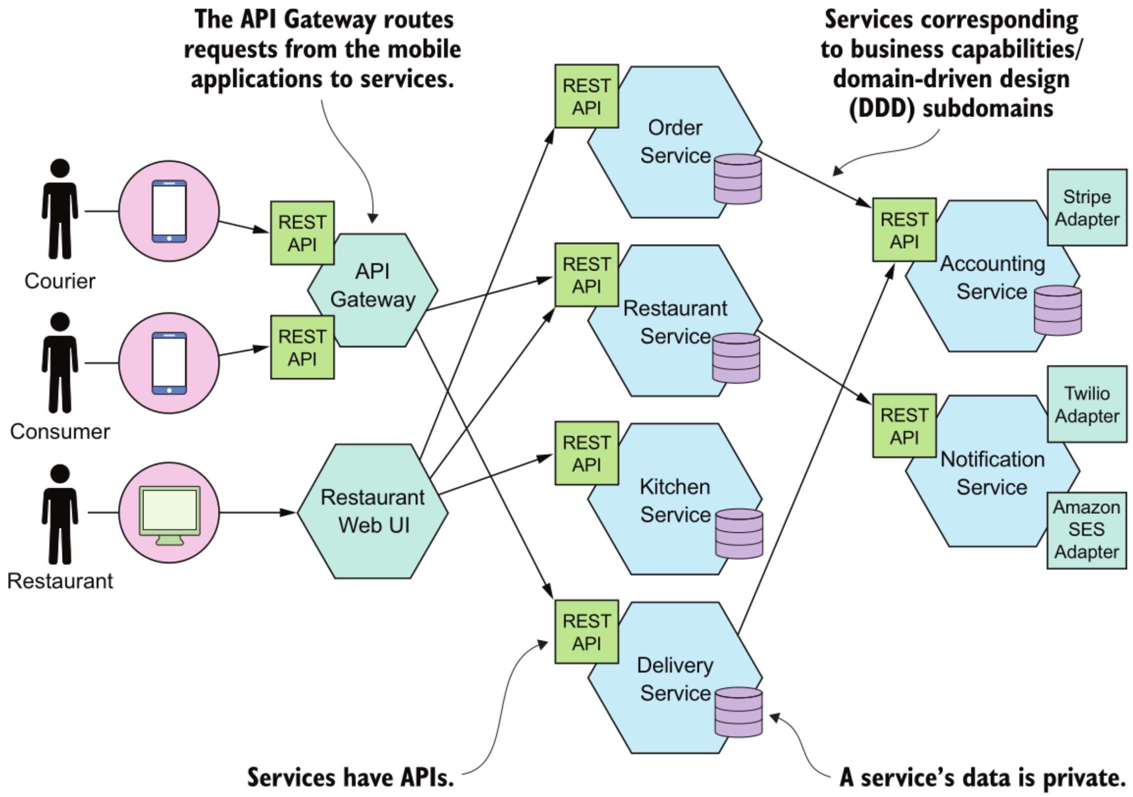

Uses the userId to decide where to route requests
N identical application instances
Application instance 1
Users: a–h
Request:
GET /...
Authorization: userId:password
Client
Router
Application instance 2
Users: i-p
Application instance 3
Users: r–z
Each instance is responsible for a subset of the users.

Z-axis scaling runs multiple identical instances of the **monolithic** application behind
a router, which routes based on a request attribute . Each instance is responsible for a subset
of the data.



Y-axis scaling **splits** the application into a set of services. Each service is responsible for
a particular function (Single-Focus/Micro). A service is scaled using X-axis scaling and, possibly, Z-axis scaling.

The `FTGO` `microservice` architecture



The API Gateway routes requests from the mobile applications to services.

Services corresponding to business capabilities/ domain-driven design (DDD) subdomains

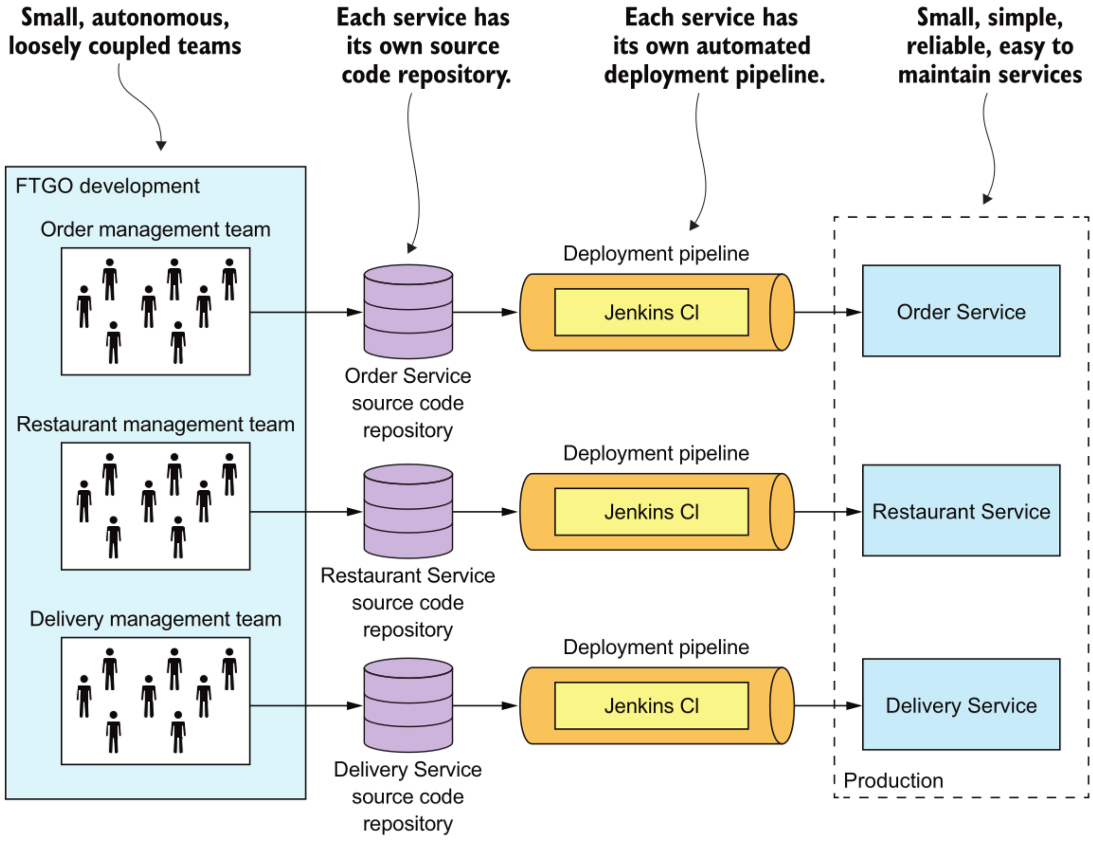
Services have APIs.

A service's data is private.

- **Order Service** —Manages orders
- **Delivery Service** —Manages delivery of orders from restaurants to consumers
- **Restaurant Service** —Maintains information about restaurants
- **Kitchen Service** —Manages the preparation of orders
- **Accounting Service** —Handles billing and payments

1. It enables the continuous delivery and deployment of large, complex applications.

Benefits of the `microservice` architecture

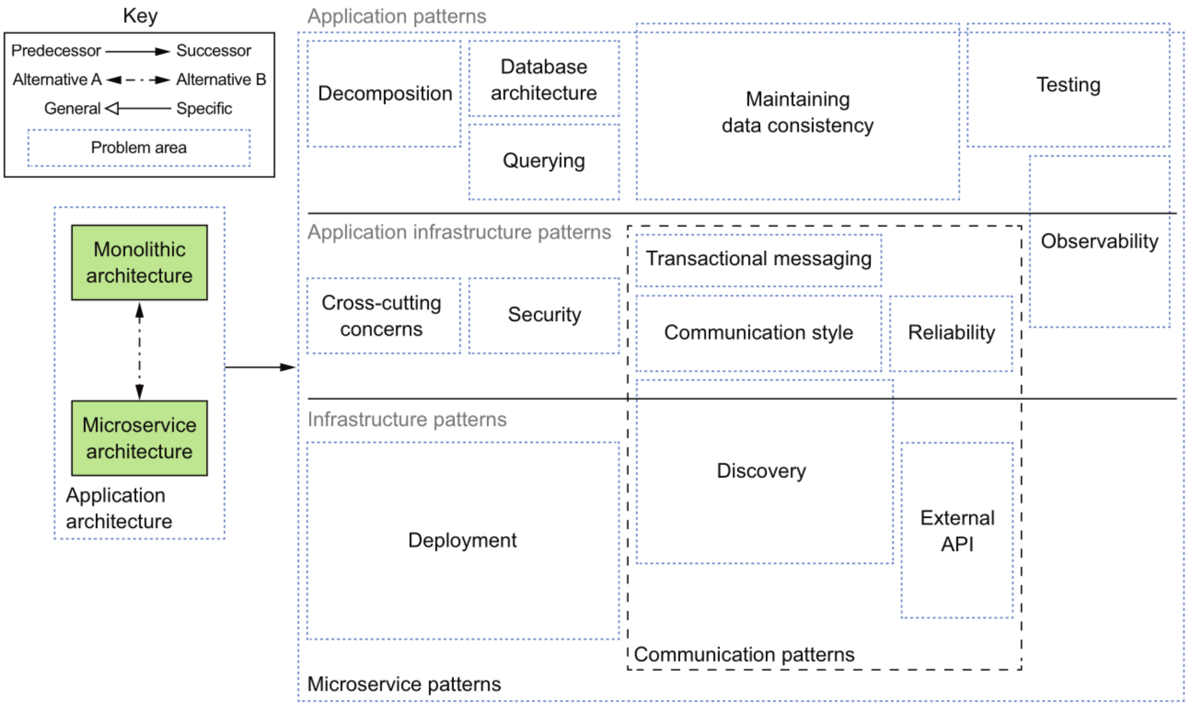

Small, autonomous, loosely coupled teams
Each service has its own source code repository.
Each service has its own automated deployment pipeline.
Small, simple, reliable, easy to maintain services
FTGO development
Order management team
Restaurant management team
Delivery management team
Order Service source code repository
Restaurant Service source code repository
Delivery Service source code repository
Deployment pipeline
Jenkins CI
Deployment pipeline
Jenkins CI
Deployment pipeline
Jenkins CI
Order Service
Restaurant Service
Delivery Service
Production

2. Services are small and easily maintained.
3. Services are independently deployable.
4. Services are independently **scalable**.
5. The microservice architecture enables teams to be autonomous.
6. It allows easy experimenting and adoption of new technologies.
7. It has better fault isolation.

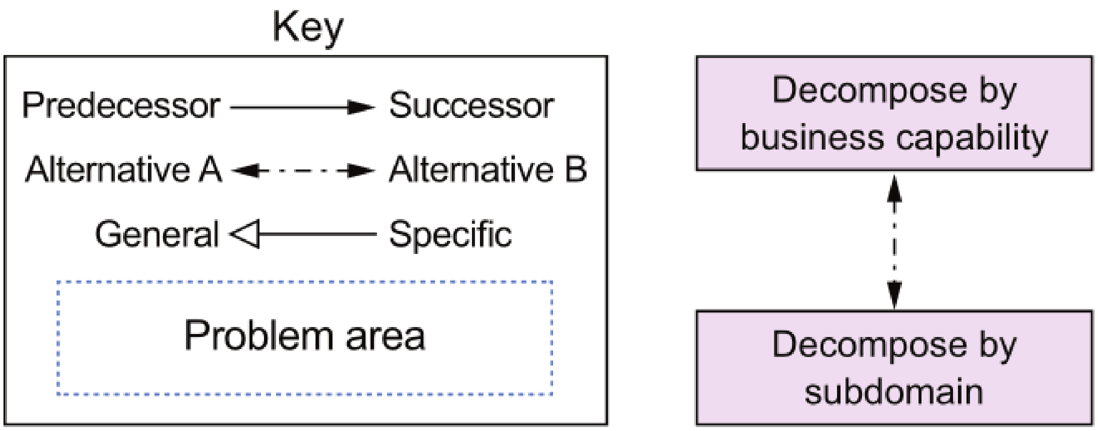
Challenges of the microservice architecture

- Finding the right set of services is challenging.
- Distributed systems are complex, which makes development, testing, and deployment difficult.
- Deploying features that span multiple services requires careful coordination.
- Deciding when to adopt the microservice architecture is difficult.

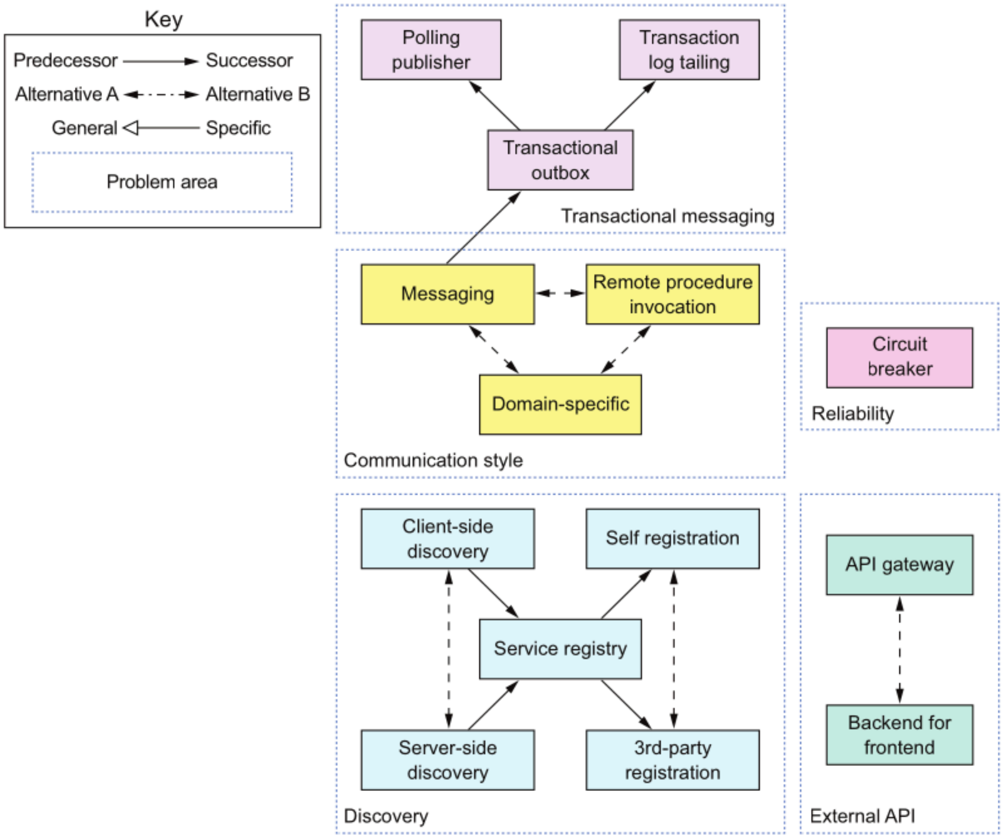
Microservice architecture pattern language



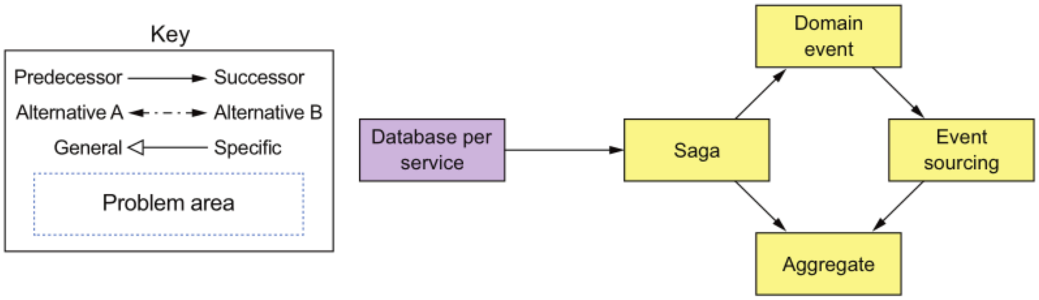
PATTERNS FOR DECOMPOSING AN APPLICATION INTO SERVICES



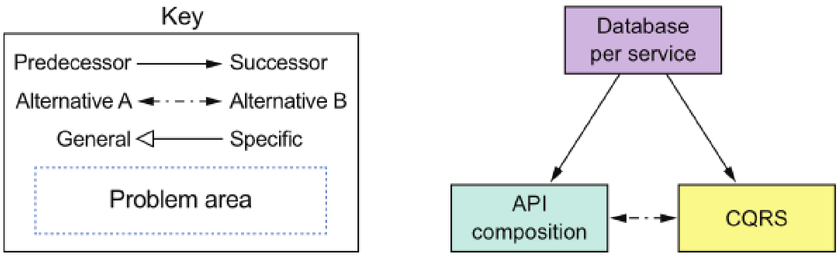
COMMUNICATION PATTERNS



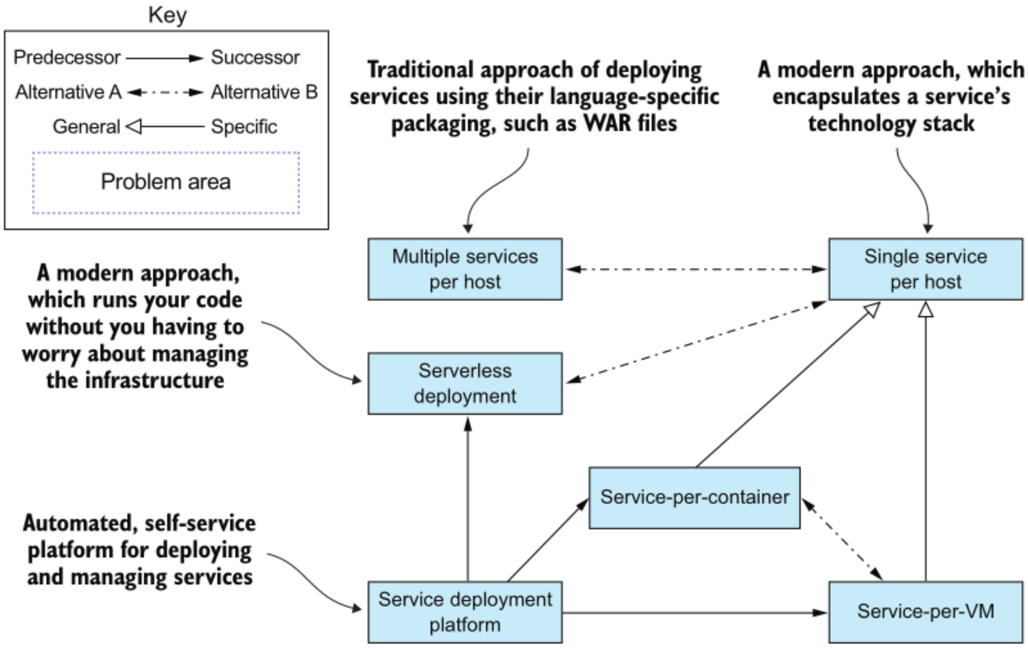
DATA CONSISTENCY PATTERNS FOR IMPLEMENTING TRANSACTION MANAGEMENT



PATTERNS FOR QUERYING DATA IN A MICROSERVICE ARCHITECTURE



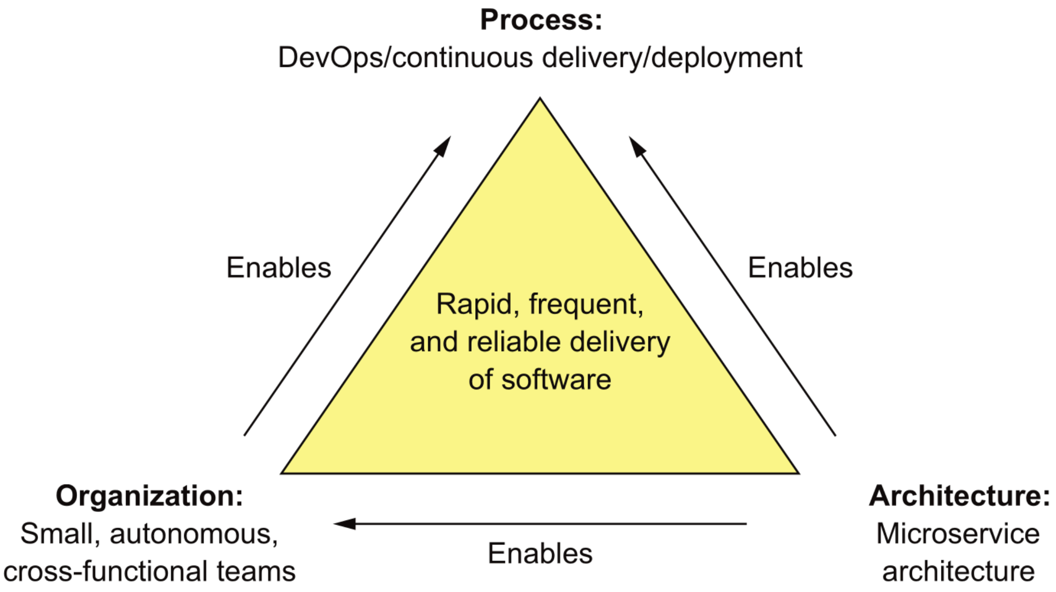
SERVICE DEPLOYMENT PATTERNS


Key
Predecessor
Successor
Alternative A
Alternative B
General
Specific
Problem area
Traditional approach of deploying services using their language-specific packaging, such as WAR files
A modern approach, which encapsulates a service's technology stack
A modern approach, which runs your code without you having to worry about managing the infrastructure
Automated, self-service platform for deploying and managing services
Multiple services per host
Single service per host
Serverless deployment
Service-per-container
Service deployment platform
Service-per-VM

OBSERVABILITY PATTERNS PROVIDE INSIGHT INTO APPLICATION BEHAVIOR

PATTERNS FOR HANDLING CROSS - CUTTING CONCERNS

SECURITY PATTERNS


Process:
DevOps/continuous delivery/deployment
Enables
Enables
Rapid, frequent, and reliable delivery of software
Organization:
Small, autonomous, cross-functional teams
Enables
Architecture:
Microservice architecture