MODULE 2: THE SCALE CUBE & MICROSERVICES

"Three ways to scale. Only one leads to microservices."

Where We Left Off

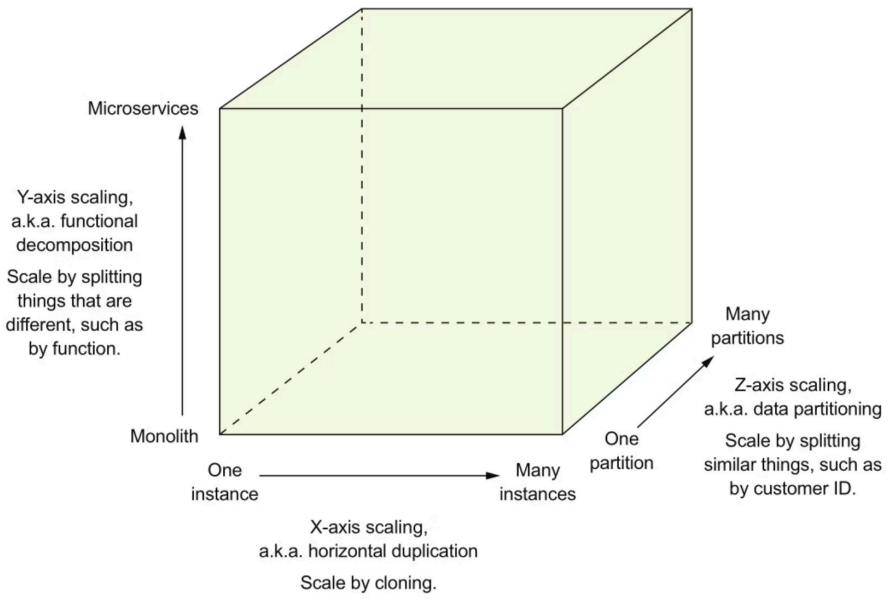
FTGO is in monolithic hell:

- 600K lines of code, nobody understands all of it
- 45-minute builds, monthly releases
- Saturday night: Kitchen is the bottleneck, but we scale EVERYTHING
- One bug in Notification → entire platform crashes

The question: "How do we scale FTGO without these problems?"

The AKF Scale Cube

- Created by AKF Partners (Martin Abbott & Michael Fisher)
- A 3-dimensional model for scaling applications
- Each axis represents a DIFFERENT scaling strategy
- You can combine axes — they're not mutually exclusive

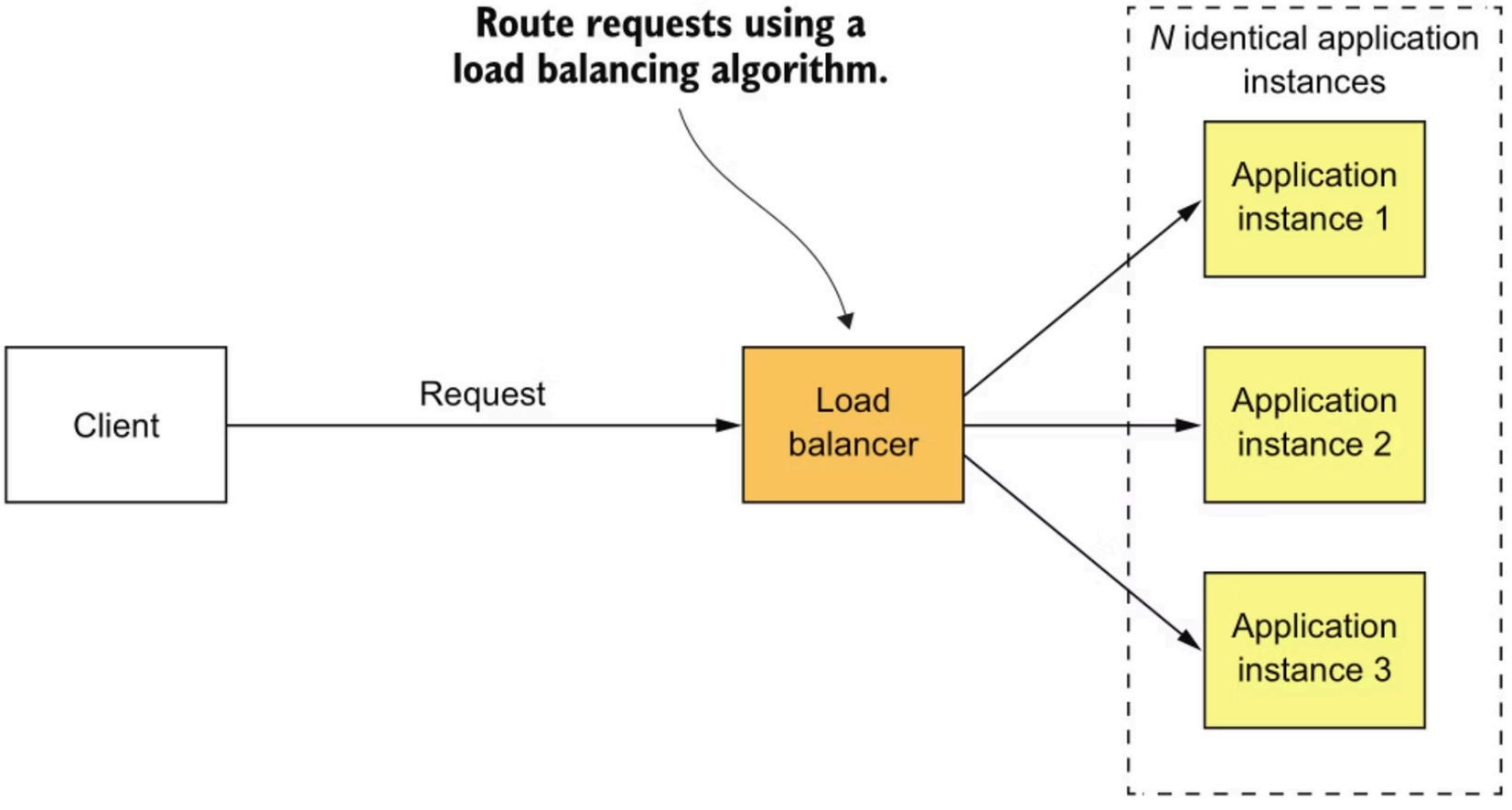


X-Axis — Run More Copies

How it works:

- Clone the ENTIRE application N times
- Place a load balancer in front
- Each instance is IDENTICAL — handles any request
- Load balancer distributes requests (round-robin, least connections)

X-Axis


Route requests using a load balancing algorithm.
N identical application instances
Application instance 1
Application instance 2
Application instance 3
Client
Request
Load balancer

X-Axis Applied to FTGO

Saturday night scenario:

- Order volume spikes 10x
- Kitchen module is the bottleneck
- Solution: Run 10 identical FTGO instances

What happens:

- All 10 instances have Order + Kitchen + Delivery + Accounting + Notification
- Each instance can handle any request
- Load balancer spreads the traffic
- You're paying for 10 copies of Notification you don't need

X-Axis Scorecard

What it SOLVES:

- Throughput — more instances = more requests handled
- Availability — if one instance dies, others continue
- Simple to implement — just add instances

What it DOESN'T solve:

- Codebase complexity — every instance has 600K lines
- Build/deploy speed — still one 45-minute build
- Team friction — still one codebase, one pipeline
- Technology lock-in — still one tech stack
- Targeted scaling — can't scale Kitchen without scaling Notification
- Data volume — all instances hit the same database

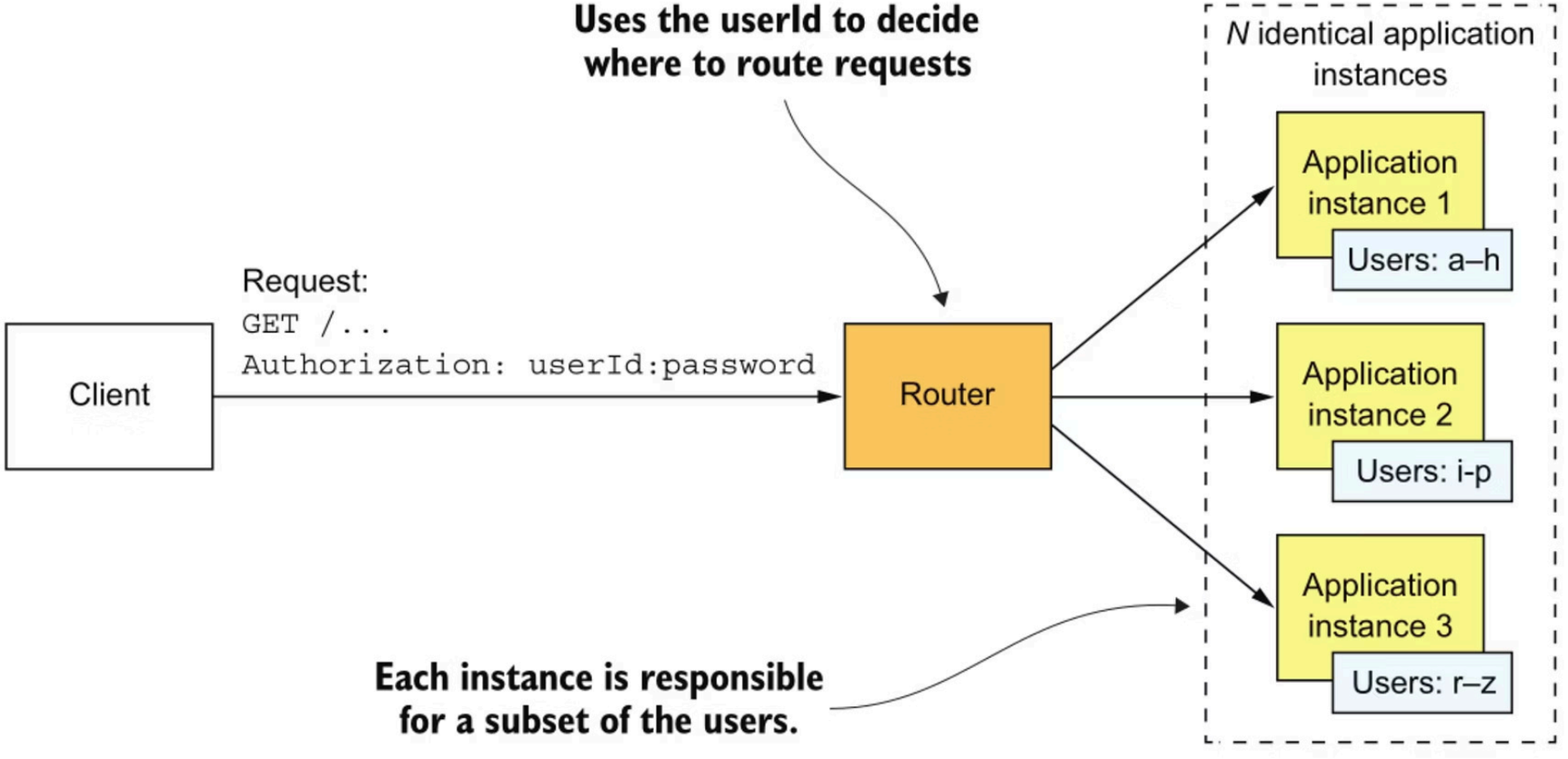
Verdict: Necessary, but NOT sufficient for FTGO's problems

Z-Axis — Split by Data

How it works:

- Run multiple instances (like X-axis)
- BUT each instance handles a SUBSET of the data
- A router directs requests based on an attribute (customer ID, region, restaurant city)
- Each instance is specialized by data, not by function

Z-Axis


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 Applied to FTGO

Partition by city:

- FTGO Mumbai instance — handles all Mumbai restaurants, orders, deliveries
- FTGO Delhi instance — handles all Delhi restaurants, orders, deliveries
- FTGO Chennai instance — handles all Chennai restaurants, orders, deliveries

What it enables:

- Each instance handles less data
- Regional scaling: Mumbai has high volume? Add more Mumbai instances
- Data locality: Mumbai data stays close to Mumbai users
- But each instance STILL has ALL modules

Z-Axis Scorecard

What it SOLVES:

- Data volume — each instance handles a smaller dataset
- Regional performance — lower latency for local users
- Targeted data scaling — scale high-volume regions independently
- Fault isolation by region — Mumbai outage doesn't affect Delhi

What it DOESN'T solve:

- Codebase complexity — still 600K lines per instance
- Build/deploy speed — still one monolithic build
- Team friction — still one codebase
- Technology lock-in — still one tech stack
- Module-level scaling — can't scale Kitchen independently

Verdict: Useful for data-heavy apps, but doesn't solve organisational or complexity problems

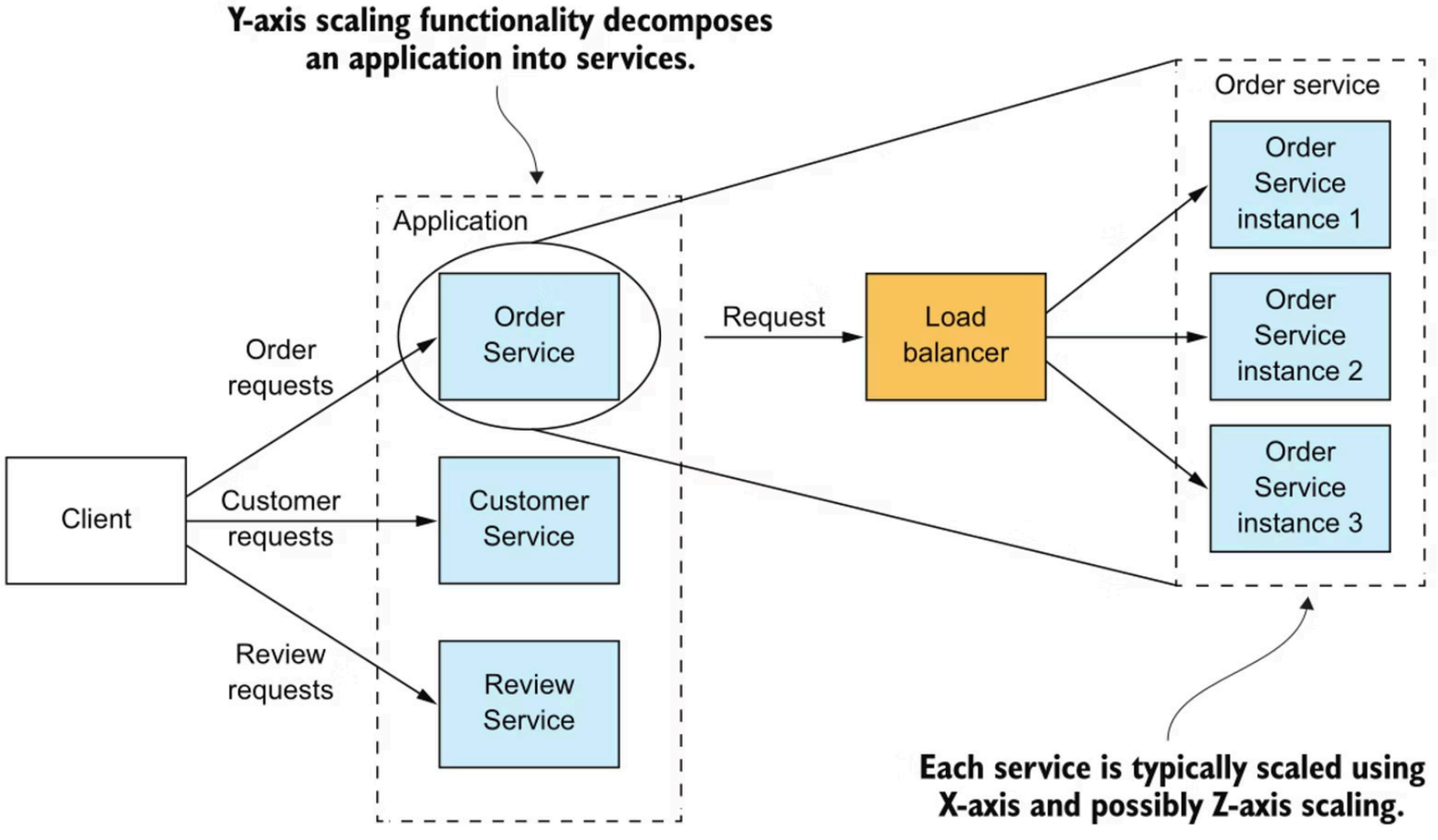
Y-Axis — Split by Function

How it works:

- Split the application into SEPARATE SERVICES by function
- Each service is responsible for ONE business capability
- Each service is independently developed, deployed, and scaled
- Services communicate via APIs or messages

This IS microservices.

Y-Axis


Y-axis scaling functionality decomposes an application into services.
Application
Order Service
Customer Service
Review Service
Client
Order requests
Customer requests
Review requests
Request
Load balancer
Order service
Order Service instance 1
Order Service instance 2
Order Service instance 3
Each service is typically scaled using X-axis and possibly Z-axis scaling.

Y-Axis Applied to FTGO

FTGO splits into independent services:

Order Service
manages order lifecycle

Kitchen Service
handles food preparation

Delivery Service
manages couriers and delivery

Restaurant Service
maintains restaurant info, menus

Accounting Service
handles payments and billing

Notification Service
sends emails, SMS, push

Each service:

- Has its own codebase
- Has its own database
- Has its own deployment pipeline
- Is owned by a dedicated team
- Can be scaled independently

Y-Axis — What It Solves

Complexity intimidates developers	Each service is small (10K-50K lines)
Development is slow	Each service builds in seconds
Long commit-to-deploy path	Independent deployment per service
Scaling is all-or-nothing	Scale each service independently
Reliability is fragile	Fault isolation — one fails, others live
Technology lock-in	Each service picks its own tech stack
Organisational friction	Each team owns their service end-to-end

Y-Axis — What It Introduces

New challenges (we'll solve these in the coming modules):

- Distributed system complexity — network is unreliable
- Cross-service transactions — no more ACID across services
- Cross-service queries — no more SQL JOINs across services
- Service communication — how do services talk?
- Service discovery — how do services find each other?
- Operational overhead — 6 services to monitor, deploy, debug
- Data consistency — eventual consistency replaces immediate consistency
- Testing complexity — can't test everything in one process

Scale Cube — Side by Side

X-Axis	Clone everything	Throughput, availability	Complexity, team autonomy
Z-Axis	Partition by data	Data volume, regional latency	Complexity, team autonomy
Y-Axis	Split by function	Complexity, autonomy, independent scaling & deployment	Introduces distributed system challenges

They are NOT mutually exclusive — FTGO will use ALL THREE:

- Y-axis → split into services
- X-axis → scale each service horizontally
- Z-axis → partition data where needed (Delivery by region)

FTGO Uses All Three Axes

Y-Axis (functional split):

- Order Service, Kitchen Service, Delivery Service, etc.

X-Axis (horizontal scaling per service):

- Kitchen Service: 10 instances on Saturday night
- Notification Service: 2 instances (low traffic)

Z-Axis (data partitioning where needed):

- Delivery Service: sharded by city
  - Mumbai couriers → Mumbai partition
  - Delhi couriers → Delhi partition

Defining Microservices

A microservice architecture structures an application as a collection of services that are:

- **Loosely coupled**
  minimal dependencies between services
- **Independently deployable**
  ship one service without touching others
- **Organised around business capabilities**
  not technical layers
- **Owned by a small team**
  "two-pizza team" (5-9 people)
- **Communicating via well-defined APIs**
  REST, gRPC, or messages
- **Having its own data store**
  no shared database

Common Misconceptions

Microservices are NOT:

"Small services"

- Size is irrelevant. RESPONSIBILITY matters.
- A 50K-line service with clear boundaries is fine.

"REST APIs"

- Communication style is a CHOICE, not a defining characteristic

"Docker containers"

- Docker is a DEPLOYMENT technology, not an architecture

"The opposite of monolith"

- It's a different STRUCTURAL PATTERN
- A well-designed monolith can outperform badly designed microservices

"The solution to all problems"

- They solve specific scaling, complexity, and autonomy problems
- They introduce their own complexity

The Five Characteristics

SHARED-NOTHING ARCHITECTURE

- No shared memory, no shared database, no shared file system

WELL-DEFINED INTERFACES

- API calls (REST, gRPC) or Messages (Kafka, RabbitMQ)
- Contract between services — versioned, documented

SEPARATE RUNTIME PROCESS

Each service runs as its own process (container, VM, serverless)

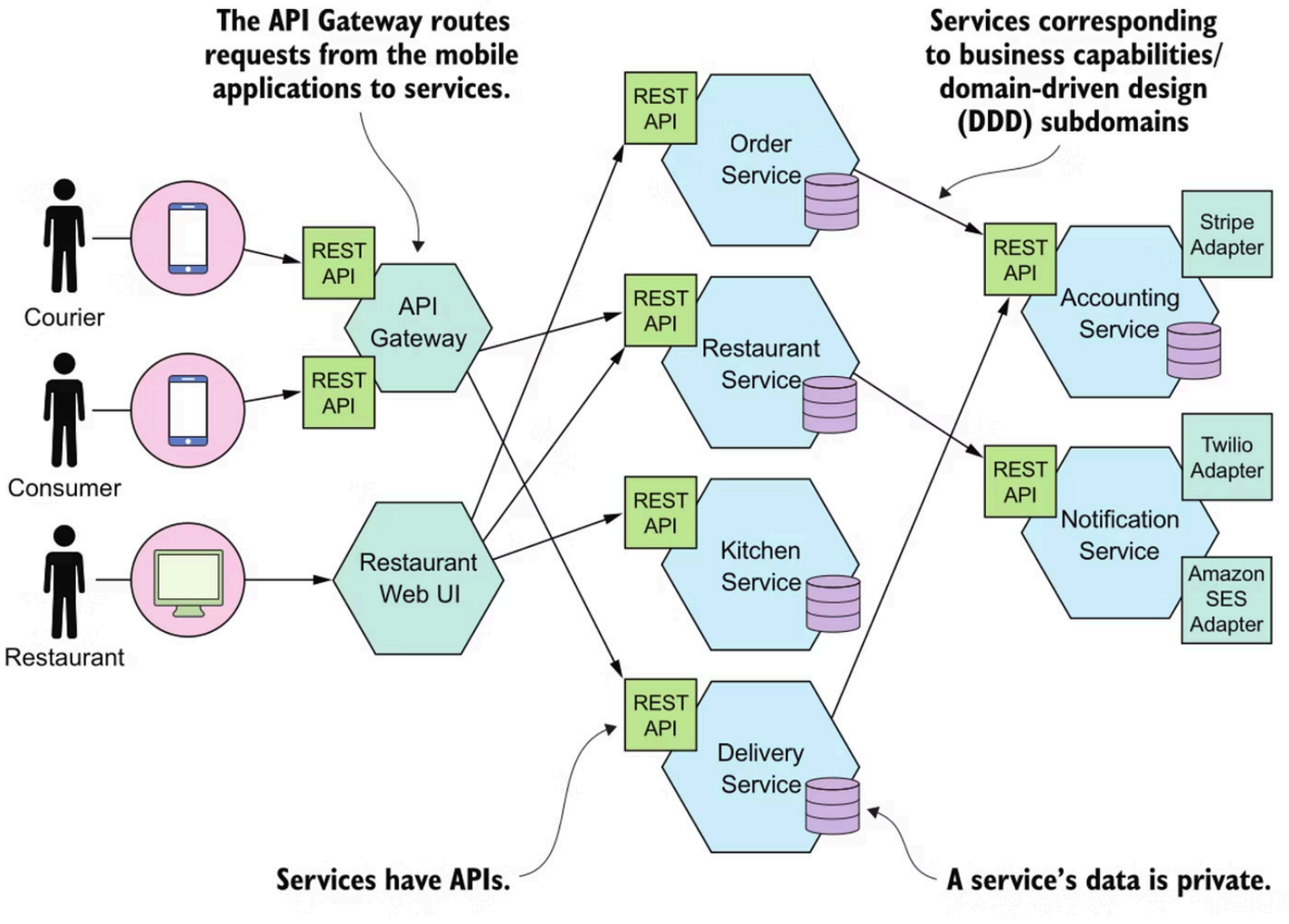
STATELESS INSTANCES

- Service instances don't hold session state
- Any instance can handle any request

INDEPENDENT DATA STORE

- Each service owns its data
- No direct database access by other services

FTGO — The Target Architecture



FTGO Service Responsibilities

Order Service	Manages order lifecycle (create, approve, cancel, track)	orders, items, order history
Kitchen Service	Manages food preparation (accept ticket, mark ready)	tickets, prep schedules
Delivery Service	Manages courier assignment & delivery (assign courier, track location, deliver)	deliveries, courier GPS
Restaurant Service	Manages restaurant info (profiles, menus, operating hours)	restaurants, menus
Accounting Service	Handles payments & billing (authorise, charge, refund, invoices)	payments, invoices
Notification Service	Sends notifications (email, SMS, push)	templates, delivery logs

Benefit 1 — Small and Maintainable

- Each service: 10K-50K lines (vs 600K monolith)
- A developer can understand ONE service completely
- Onboarding: hours, not weeks
- Codebase fits in your head
- Refactoring is safe — blast radius is limited to one service
- Code reviews are manageable

Monolith:

"Who understands the whole system?"

→ Nobody

Microservices:

"Who understands Order Service?"

→ The Order team

Benefit 2 — Independently Deployable

- Order team ships a new feature → deploys Order Service only
- Kitchen team is not affected. Delivery team doesn't know. Nobody waits.
- Each service has its own CI/CD pipeline
- Deploy 10 times a day or once a week — your choice
- Rollback one service without touching others
- No more "code freeze" or "release trains"

Monolith:

"Can we deploy?"

→ "Wait for Team B to finish testing"

Microservices:

"Can we deploy?"

→ "Already done. 3 minutes ago."

Benefit 3 — Independently Scalable

Saturday night at FTGO:

10%

Order Service

5 instances (moderate traffic)

31%

Kitchen Service

15 instances (bottleneck — peak food prep)

42%

Delivery Service

20 instances (highest traffic — GPS tracking)

6%

Restaurant Service

3 instances (low traffic)

6%

Accounting Service

3 instances (moderate)

4%

Notification Service

2 instances (fire-and-forget)

Total: 48 instances, OPTIMISED per service

Monolith equivalent: 20 full instances × all modules = massive waste

Benefit 4 — Team Autonomy

- Each team owns their service(s) end-to-end
- Development → Testing → Deployment → Monitoring → On-Call
- No permission needed from other teams to ship
- No merge conflicts with other teams
- Each team chooses their own tools, libraries, practices

> "You build it, you run it"
>
> — Werner Vogels (Amazon CTO)

Benefit 5 — Technology Diversity

Each service picks the BEST technology for its job:

Order Service
Java + Spring Boot (mature, enterprise-grade)

Kitchen Service
Java + MongoDB (flexible schema for recipes)

Delivery Service
Go (low latency, small footprint)

Restaurant Service
Java + PostgreSQL (standard CRUD)

Accounting Service
Java + PostgreSQL (ACID for financial data)

Notification Service
Node.js (async I/O, WebSocket native)

This is POLYGLOT architecture — right tool for the right job

Benefit 6 — Better Fault Isolation

Monolith:

Memory leak in Notification → OutOfMemory → ENTIRE PLATFORM DOWN

Blast radius: ONE service, not THE WHOLE SYSTEM

Microservices:

Memory leak in Notification Service → Notification restarts

- Order Service? Running fine.
- Kitchen Service? Running fine.
- Delivery Service? Running fine.

Impact: Customers don't get email confirmation. That's it.

Microservices Are NOT Free

Don't ignore these challenges:

- **Finding the right service boundaries is HARD**

  Get it wrong = distributed monolith

- **Distributed systems are inherently complex**

  Network is unreliable, latency is real

- **Cross-service features require coordination**

  "Add a new field" might touch 4 services

- **Data consistency is non-trivial**

  No more ACID across services

- **Operational overhead multiplies**

  6 services = 6 pipelines, 6 log streams, 6 things that can fail

- **Deciding WHEN to adopt is difficult**

  Too early = premature complexity. Too late = painful migration.

Should You Adopt Microservices?

YES, consider if:

- Large team (20+ developers) on one codebase
- Different modules have different scaling needs
- Team autonomy is blocked by shared codebase
- Technology diversity would provide real value
- You need frequent, independent releases
- Your DevOps maturity is high (CI/CD, monitoring, automation)

NO, stay with monolith if:

- Small team (< 10 developers)
- Simple domain
- No scaling bottleneck
- Team is not mature in DevOps
- Domain is not well understood (still exploring product-market fit)
- "Because Netflix does it" is your primary reason

Don't Start with Microservices

The Premature Decomposition Anti-Pattern:

- Starting a NEW project as microservices from day one
- You don't understand the domain yet
- You don't know where the boundaries should be
- Wrong boundaries = distributed monolith = worst of both worlds

The correct path:

1. Start with a well-structured monolith
2. Understand the domain deeply
3. Identify natural boundaries (bounded contexts)
4. Extract services incrementally (Strangler Fig)
5. Each extraction solves a REAL problem

> "Don't start with microservices. Start with a monolith you can decompose."
>
> — Sam Newman, Martin Fowler, Chris Richardson

A Pattern for Every Challenge

The microservices.io pattern language:

How to split?	Decomposition Patterns
How to manage data?	Data Patterns
How do services talk?	Communication Patterns
How do clients access?	External API Patterns
How to find services?	Discovery Patterns
How to handle transactions?	Transaction Patterns
How to handle failure?	Reliability Patterns
How to secure services?	Security Patterns
How to see what's happening?	Observability Patterns
How to deploy?	Deployment Patterns
How to test?	Testing Patterns

Source: microservices.io/patterns/index.html

Key Takeaways

1. The Scale Cube has THREE axes:
   - X-Axis: Clone everything (throughput)
   - Z-Axis: Partition by data (data volume)
   - Y-Axis: Split by function (complexity, autonomy) — this IS microservices

2. Y-axis solves FTGO's core problems:
   - Small codebases, independent deployment, targeted scaling
   - Team autonomy, technology freedom, fault isolation

3. Y-axis introduces NEW challenges:
   - Distributed complexity, data consistency, operational overhead
   - We have PATTERNS for every challenge (coming next)

4. Adopt microservices for the RIGHT reasons, not hype

5. Never start with microservices — start with a monolith, then decompose

Pattern References:

- Monolithic Architecture — microservices.io/patterns/monolithic.html
- Microservice Architecture — microservices.io/patterns/microservices.html

Coming Next: Module 3 — Decomposition Patterns

"HOW do we decide where to cut?"