MODULE 1: THE FTGO MONOLITH — Appreciating What Works

"Before we break it apart, let's understand what we're breaking."

Meet FTGO — Food To Go

- FTGO is a food delivery platform
- Consumers order food from local restaurants
- Restaurants prepare the food
- Couriers pick up and deliver
- Payments are processed automatically
- Everyone gets notified at every step

FTGO Today — A Growing Business

50K+

Orders per day

3K+

Restaurant partners

5K+

Delivery couriers

12

Developers across 3 teams

15

Cities

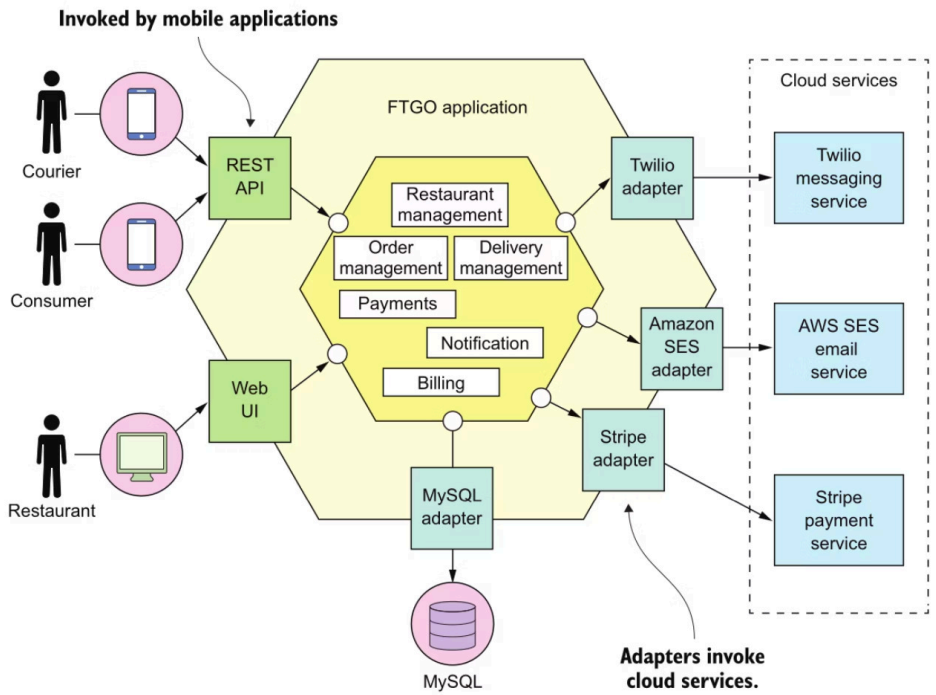
40%

Revenue growth year-over-year

Founded 4 years ago

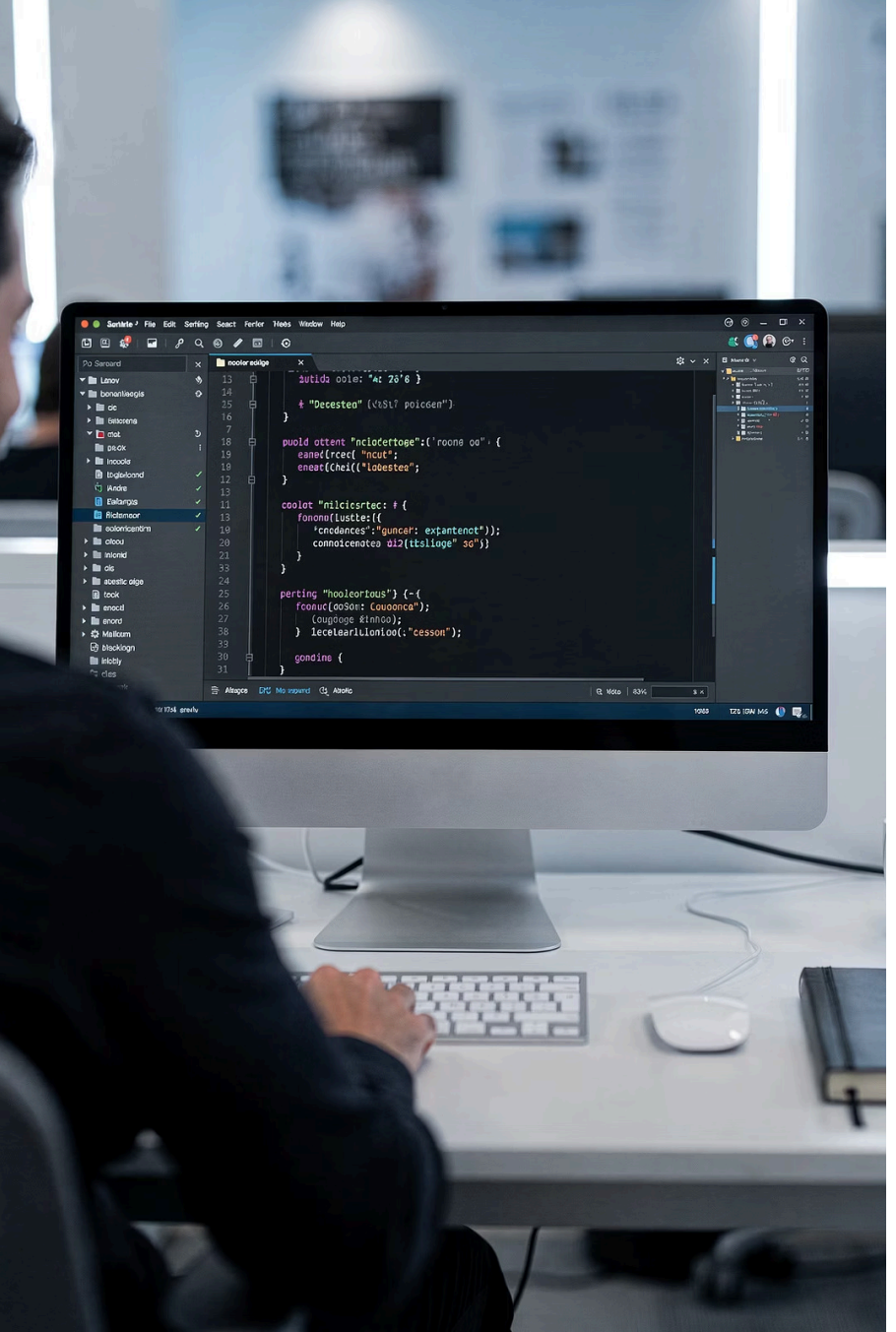
FTGO Architecture — One Application, One Database

- Single deployable unit (WAR/JAR)
- All modules in one codebase
- One shared PostgreSQL database (120+ tables)
- REST APIs for mobile and web clients
- Single CI/CD pipeline
- Deployed as one unit on multiple servers behind a load balancer


Invoked by mobile applications
Courier
Consumer
Restaurant
REST API
Web UI
FTGO application
Restaurant management
Order management
Delivery management
Payments
Notification
Billing
MySQL adapter
MySQL
Twilio adapter
Amazon SES adapter
Stripe adapter
Cloud services
Twilio messaging service
AWS SES email service
Stripe payment service
Adapters invoke cloud services.

The FTGO Modules

1. Order Module
   Manages the entire order lifecycle
2. Kitchen Module
   Handles food preparation tracking
3. Delivery Module
   Assigns couriers, tracks delivery
4. Restaurant Module
   Manages restaurant profiles, menus, hours
5. Accounting Module
   Handles payments, invoices, billing
6. Notification Module
   Sends emails, SMS, push notifications
7. User Module
   Authentication, profiles, preferences



Benefit 1 — Simple to Develop

- One codebase — one IDE, one project
- Search across the entire application instantly
- Refactoring is a right-click away
- "Find all references" works everywhere
- One language, one framework, one build tool
- New developer setup: clone repo → build → run

Benefit 2 — Simple to Test

- Start one process — the entire application is running
- Integration tests are straightforward
- No network mocking needed
- Test the full flow in one process: order → payment → kitchen → delivery
- Debugging: set breakpoint in Order → step into Kitchen → step into Payment

Benefit 3 — Simple to Deploy

- One artifact: build it → deploy it → done
- No service orchestration needed
- No dependency management between services
- Rollback = deploy the previous version
- Everyone ships together — one pipeline, one release

Simple pipeline — Code → Build → Test → Deploy (one straight line)

Benefit 4 — Simple Transactions

- One database — one transaction — ACID guaranteed
- "Create order + charge payment + create kitchen ticket" = one DB commit
- If payment fails → everything rolls back automatically
- No partial state. No inconsistency. No data corruption.

```
@Transactional
public Order createOrder(OrderRequest req) {
  Order order = orderRepo.save(new Order(req));
  payment.authorize(req.paymentInfo);
  kitchen.createTicket(order);
  return order; // ALL or NOTHING
}
```

Single transaction wrapping Order + Payment + Kitchen in one commit

Benefit 5 — Simple to Scale (Sort Of)

- Run multiple identical instances behind a load balancer
- Any instance handles any request
- Horizontal scaling is straightforward
- Session management: sticky sessions or external session store

The Monolith Is NOT the Enemy

- The monolith is a valid architectural pattern
- It works well for small-to-medium teams
- It's the right choice for many applications
- Most successful products STARTED as monoliths — Amazon, Netflix, Uber, Twitter — all started monolithic

The real enemy? → A POORLY MANAGED monolith that has outgrown its architecture.

So When Does It Break?

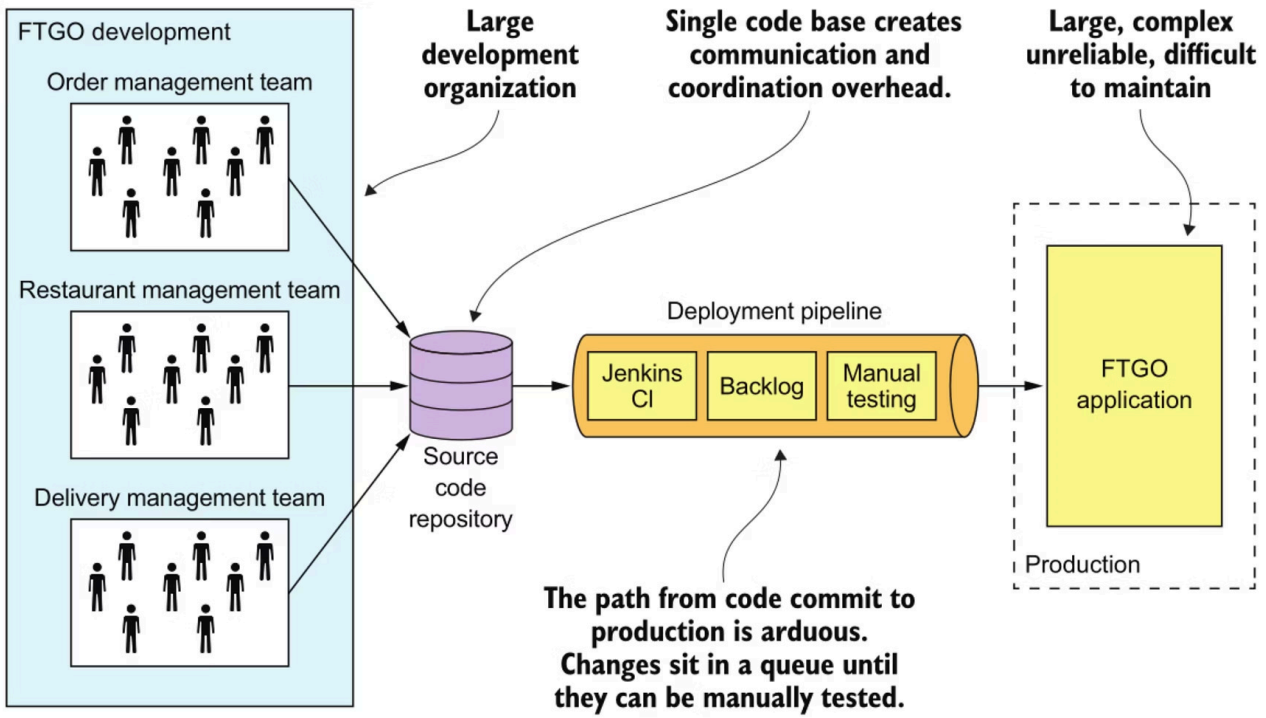
"FTGO grew.

The team grew.

The codebase grew.

And slowly, the monolith became... a problem."

FTGO

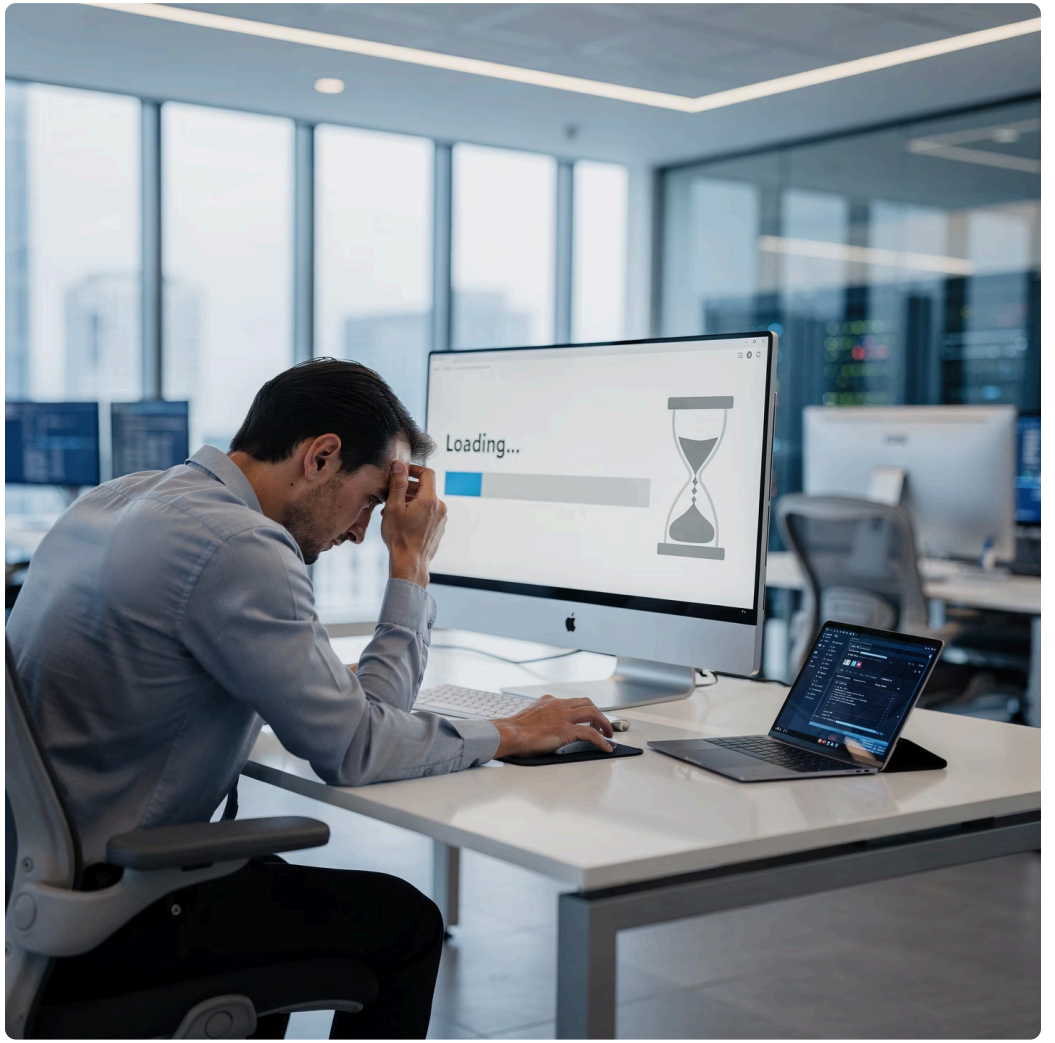

FTGO development
Order management team
Restaurant management team
Delivery management team
Large development organization
Single code base creates communication and coordination overhead.
Large, complex unreliable, difficult to maintain
Source code repository
Deployment pipeline
Jenkins CI
Backlog
Manual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

Problem 1 — Complexity Intimidates Developers

- 600K+ lines of code
- 120+ database tables
- No one understands the full system anymore
- New developer asks: "Where does order creation start?" → Answer: "It depends... there are 7 entry points"
- Onboarding takes 3-4 weeks before first meaningful contribution
- Developers are AFRAID to change code they don't understand
- Result: declining velocity, increasing bugs

Problem 2 — Development Is Slow

- Build time: 45 minutes (full compilation)
- Test suite: 30 minutes
- "I changed one line in Notification → full rebuild"
- IDE struggles with the codebase size (indexing, autocomplete lag)
- Developer productivity: declining quarter over quarter
- Developers spend more time WAITING than CODING



Problem 3 — Commit to Deployment Is Painful

- Feature branches live for weeks (long-lived branches)
- Merge conflicts are a daily occurrence
- "Code freeze" period before each release
- Monthly releases — features wait 4-8 weeks to reach production
- Hotfix = rebuild and redeploy EVERYTHING
- Deployment takes a full weekend (manual, risky)
- "Don't deploy on Friday" is an actual rule

Problem 4 — Scaling Is All-or-Nothing

- Saturday night: order volume goes 10x
- The bottleneck is ONLY the Kitchen module
- But you must scale the ENTIRE monolith
- 10x infrastructure cost to handle 1 hotspot
- Memory-hungry Accounting module runs on every instance (wasted resources)
- Can't scale Order processing independently from Notification

Problem 5 — Reliability Is Fragile

- Memory leak in Notification → OutOfMemoryError → ALL modules crash
- Bad SQL query in Reporting → locks the Orders table → Ordering stops
- One bug's blast radius = THE ENTIRE PLATFORM
- No fault isolation between modules

Scenario:

Notification has a memory leak → consumes all heap → JVM OOM → Order, Kitchen, Payment — ALL go down.

50,000 orders/day → ZERO.

Problem 6 — Locked Into Obsolete Technology

- FTGO started on Java 8 + Spring 4
- Want Go for Delivery? (low latency, small footprint) → Can't
- Want MongoDB for Restaurant menus? (flexible schema) → Can't
- Want Node.js for real-time Notifications? (WebSocket native) → Can't
- Want to upgrade to Java 17? → Must upgrade EVERYTHING at once
- The ENTIRE team must agree on ONE technology stack

Problem 7 — Organisational Friction

- 3 teams, 1 codebase, 1 deployment pipeline
- Team A wants to release → Team B isn't ready → Team A waits
- "Please don't touch the Order module, we're refactoring it"
- Merge conflicts between teams on adjacent modules
- Coordination meetings consume development time
- Teams can't move at their own pace

Monolithic Hell — Summary

Problem	Impact
Complexity intimidates devs	Slow onboarding, fear of change
Development is slow	45-min builds, lost productivity
Long commit-to-deploy path	Monthly releases, slow time-to-market
Scaling is all-or-nothing	10x infra cost for 1 hotspot
Reliability is fragile	One bug → entire platform down
Technology lock-in	Can't adopt best-fit technologies
Organisational friction	Teams blocked by each other

"Does any of this sound familiar?"

Escaping Monolithic Hell

The answer is NOT:

- "Rewrite everything in microservices over the weekend"
- "Use Kubernetes and Docker and everything will be fine"
- "Just use Spring Cloud"

The answer IS:

- Understand the ARCHITECTURE PATTERNS that solve each specific problem
- Apply them INCREMENTALLY
- Let the system EVOLVE based on real needs

Coming up next:
→ The Scale Cube — three dimensions of scaling
→ Where microservices fit in the picture

Key Takeaways

1. The monolith is a VALID architecture — don't demonise it
2. Monoliths work great for small teams and simple domains
3. As the system grows, specific pain points emerge:
   - Complexity, slow builds, painful deployments
   - All-or-nothing scaling, fragile reliability
   - Technology lock-in, team friction
4. The monolith becomes "hell" when it outgrows its architecture
5. The solution is NOT "microservices because everyone does it"
6. The solution IS pattern-driven, incremental evolution

Pattern: Monolithic Architecture

Ref: microservices.io/patterns/monolithic.html