MODULE 3: DOMAIN-DRIVEN DECOMPOSITION — The Heart of the Course

"Get boundaries wrong, and everything fails."

How Do You Decide Where to Cut?

We know:

- The monolith must be split (Module 1 — the pain)
- Y-axis scaling = functional decomposition (Module 2 — the strategy)

The question NOW:

- Where exactly do you draw the lines?
- Which code goes into which service?
- How do you KNOW your boundaries are right?

> "The hardest part of microservices is not building them. It's finding the right boundaries."

Right Boundaries vs Wrong Boundaries

Right boundaries:

- Services are loosely coupled — change one without affecting others
- Services are highly cohesive — related things stay together
- Teams work independently
- Communication between services is minimal

Wrong boundaries:

- Services constantly call each other (chatty)
- Every feature requires changing 4 services
- Teams are always coordinating, always waiting
- "We have microservices, but we deploy them all together"
- This is a DISTRIBUTED MONOLITH

DDD — The Decomposition Backbone

Domain-Driven Design (Eric Evans, 2003):

- A methodology for modelling complex software systems
- Aligns software design with the business domain
- Provides tools for identifying service boundaries

Why DDD matters for microservices:

Bounded Contexts	Ubiquitous Language	Context Maps	Subdomain classification
become service boundaries	prevents model confusion across services	define how services relate to each other	guides investment decisions

> "Microservices without DDD is just distributed chaos."

What Is a Domain?

Domain = the problem space your software solves

FTGO's domain:

- Online food delivery
- Connecting consumers with restaurants through couriers
- Handling the full lifecycle: browse → order → pay → cook → deliver

The domain contains MANY sub-problems:

- How do consumers find restaurants?
- How are orders managed?
- How is food preparation tracked?
- How are couriers assigned and tracked?
- How are payments processed?
- How are notifications sent?

Each sub-problem = a SUBDOMAIN

Types of Subdomains

Not all subdomains are equal. DDD classifies them:

CORE Subdomain

- The competitive advantage of the business
- What makes FTGO different from competitors
- Invest heavily, build in-house, optimise
- FTGO core: Order Management, Delivery Optimisation

SUPPORTING Subdomain

- Necessary for the business but not differentiating
- Build in-house, but don't over-engineer
- FTGO supporting: Kitchen Management, Restaurant Management

GENERIC Subdomain

- Solved problems — every business needs them
- Buy off-the-shelf or outsource
- FTGO generic: Payments (Stripe), Notifications (Twilio), User Auth (Auth0)

FTGO Subdomains Classified

Order Management	Core	Build, invest heavily
Delivery Optimisation	Core	Build, invest heavily
Kitchen Management	Supporting	Build, keep simple
Restaurant Management	Supporting	Build, keep simple
Payments / Accounting	Generic	Use Stripe/Razorpay
Notifications	Generic	Use Twilio/Firebase
User Authentication	Generic	Use Auth0/Keycloak

Why this matters:

- Core services get the most engineering investment
- Generic services might not need to be microservices at all
- Helps prioritise extraction from the monolith

Bounded Contexts — The Key to Service Boundaries

Bounded Context:

A boundary within which a domain model is CONSISTENT

- Inside the boundary: one model, one language, one team
- Outside the boundary: different model, different language, different team

Why "bounded"?

- Models only make sense WITHIN their context
- The same real-world concept has DIFFERENT meanings in different contexts
- Forcing one model everywhere leads to a bloated, confusing mess

Rule: One bounded context ≈ one microservice

Same Word, Different Meanings

The word "Order" means different things in different contexts:

Order Context (Order Service)

```
{orderId, consumerId, restaurantId, items, totalPrice, status, placedAt}
```

→ Focus: lifecycle, consumer, pricing

Kitchen Context (Kitchen Service)

```
{ticketId, orderId, items, preparationTime, chef, priority}
```

→ Focus: preparation, timing, chef assignment

Delivery Context (Delivery Service)

```
{deliveryId, orderId, pickupAddress, deliveryAddress, courierId, ETA}
```

→ Focus: logistics, routing, courier

Accounting Context (Accounting Service)

```
{invoiceId, orderId, amount, paymentMethod, chargeStatus, refundable}
```

→ Focus: money, billing, compliance

Ubiquitous Language

Ubiquitous Language:

- The shared vocabulary between developers and domain experts
- Consistent WITHIN a bounded context
- DIFFERENT across bounded contexts — and that's OK

FTGO examples:

In Order Context:

- "Order" = the thing a consumer places
- "Item" = a menu item with quantity and price
- "Status" = PENDING, APPROVED, DELIVERED, CANCELLED

In Kitchen Context:

- "Ticket" = the thing a chef works on (NOT "order")
- "Item" = an item to prepare (NOT priced — kitchen doesn't know prices)
- "Status" = ACCEPTED, PREPARING, READY_FOR_PICKUP

In Delivery Context:

- "Delivery" = the thing a courier fulfils (NOT "order")
- "Pickup" = where the courier gets the food
- "Dropoff" = where the food goes

From Bounded Contexts to Microservices

The mapping:

Order Context	Order Service
Kitchen Context	Kitchen Service
Delivery Context	Delivery Service
Restaurant Context	Restaurant Service
Accounting Context	Accounting Service
Notification Context	Notification Service

Each service:

- Encapsulates its bounded context
- Has its own model (different "Order" in each)
- Has its own database (stores its own model)
- Exposes APIs using ITS language (ticket, not order, for Kitchen)

Context Map — How Services Relate

A Context Map defines the relationships between bounded contexts

Why it matters:

- Services don't exist in isolation — they interact
- The NATURE of the relationship matters
- Who depends on whom? Who adapts to whom?
- This drives API design, team dynamics, and coupling

Types of Context Relationships

Partnership:

- Two teams cooperate closely, co-evolve their models
- Example: Order and Kitchen teams align on ticket creation

Customer-Supplier:

- Upstream (supplier) provides, downstream (customer) consumes
- Supplier accommodates customer needs
- Example: Order (customer) ← Restaurant (supplier) for menu data

Conformist:

- Downstream CONFORMS to upstream's model without negotiation
- "Take it or leave it"
- Example: FTGO conforms to Stripe's payment API

Anti-Corruption Layer (ACL):

- Downstream translates upstream's model to protect its own model
- Example: Accounting Service translates Stripe's model into its own

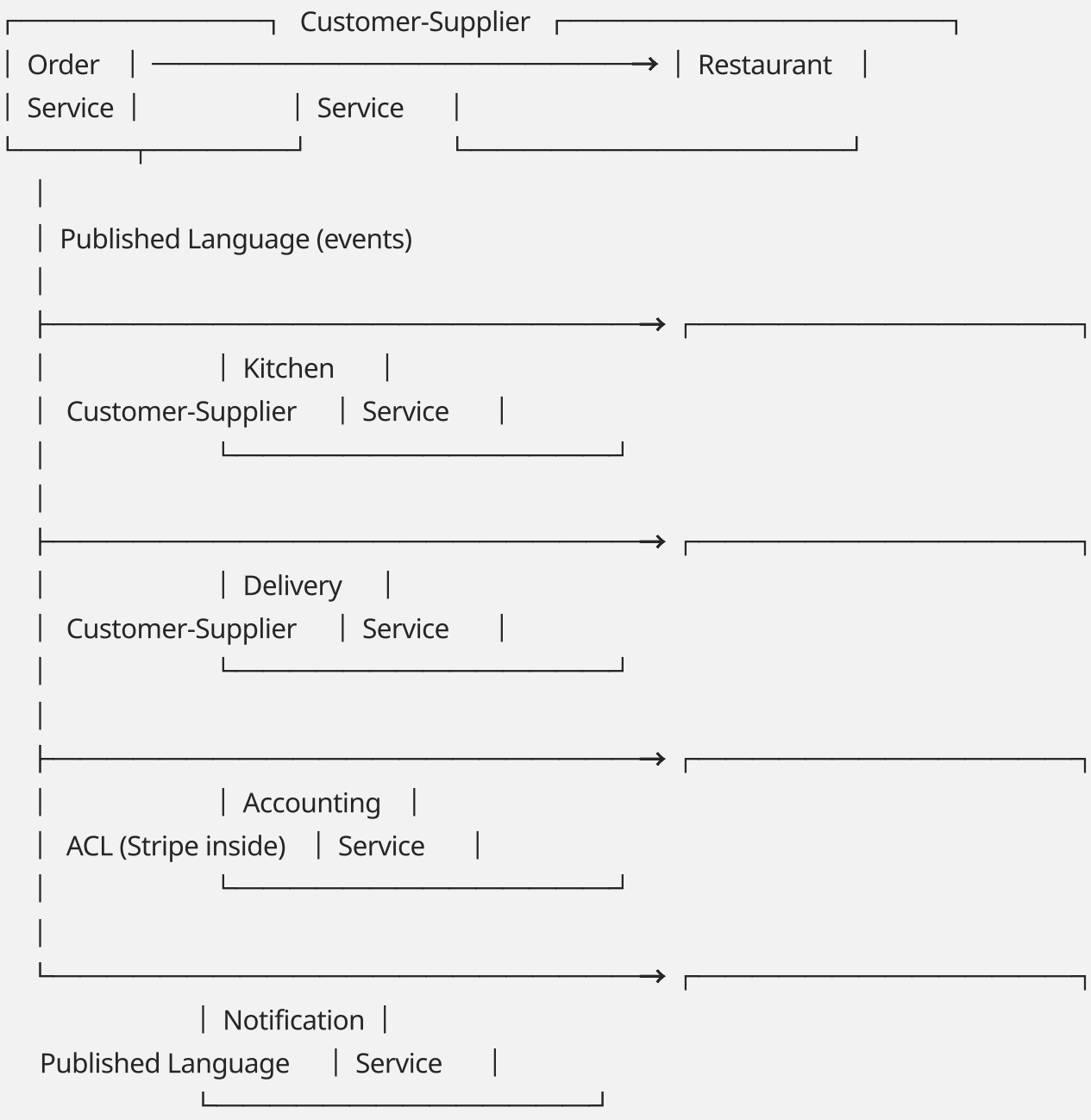
Shared Kernel:

- Two contexts share a SMALL, explicitly defined model
- Use sparingly — creates coupling
- Example: Shared "Money" value object (amount + currency)

Published Language:

- Upstream publishes a well-defined, documented API/schema
- Example: Order Service publishes OrderCreated event (Avro schema)

FTGO Context Map



Key observations:

- Order Service is the HUB — most relationships radiate from it
- Accounting has an ACL to translate Stripe's external model
- Notification consumes events — loosely coupled via Published Language

Pattern — Decompose by Business Capability

What is a business capability?

- Something the business DOES to generate value
- Stable over time (technology changes, capabilities don't)
- Identified by analysing the business purpose, not the code

FTGO business capabilities:

1. Order Management
   → Order Service
2. Food Preparation Management
   → Kitchen Service
3. Delivery Management
   → Delivery Service
4. Restaurant Management
   → Restaurant Service
5. Payment Processing
   → Accounting Service
6. Customer Notification
   → Notification Service

Rule: One capability = one service = one team

Pattern: Decompose by Business Capability
Ref: microservices.io/patterns/decomposition/decompose-by-business-capability.html

Pattern — Decompose by Subdomain

How it works:

01	02	03
Use DDD to identify subdomains	Map each subdomain to a service	Align with bounded contexts

FTGO subdomains → services:

- Core: Order Management → Order Service
- Core: Delivery Optimisation → Delivery Service
- Supporting: Kitchen Management → Kitchen Service
- Supporting: Restaurant Management → Restaurant Service
- Generic: Payments → Accounting Service (wraps Stripe)
- Generic: Notifications → Notification Service (wraps Twilio)

Difference from business capability?

- Business capability: top-down (what does the business do?)
- Subdomain: domain-model driven (what are the bounded contexts?)
- In practice: they often produce the SAME result

Pattern: Decompose by Subdomain
Ref: microservices.io/patterns/decomposition/decompose-by-subdomain.html

Pattern — Self-Contained Service

The principle:

A service should handle a synchronous request WITHOUT waiting for another service

If you call another service synchronously, your availability depends on THEIR availability

FTGO example — BAD:

`createOrder()` calls Payment Service SYNCHRONOUSLY

→ If Payment is down, Order creation fails

FTGO example — GOOD:

`createOrder()` saves order as PENDING

→ Publishes event asynchronously

→ Payment Service processes later

→ Order updated to APPROVED when payment succeeds

Result: Order Service is available even when Payment Service is down

Pattern: Self-Contained Service
Ref: microservices.io/patterns/decomposition/self-contained-service.html

Why Synchronous Chains Kill Availability

If each service has 99.5% availability:

Synchronous chain (A → B → C → D):

Availability = 99.5% × 99.5% × 99.5% × 99.5% = 98.0% = 14.6 hours downtime / month

Each additional synchronous dependency MULTIPLIES your downtime.

18.2

hours / month

5 services in chain: 97.5%

25.0

hours / month

7 services in chain: 96.6%

Solution: Break the chain with asynchronous communication

- Save locally, publish event, process later
- Each service's availability is INDEPENDENT

Pattern — Service per Team

The principle:

- Each service is owned by ONE team
- Team size: 5-9 people (Amazon's two-pizza rule)
- Full ownership: code, data, deployment, monitoring, on-call
- Team is cross-functional: backend, frontend, QA, DevOps

FTGO teams:

Order Team (6 people)	Order Service
Kitchen Team (5 people)	Kitchen Service + Restaurant Service
Delivery Team (7 people)	Delivery Service + Courier App
Payments Team (5 people)	Accounting Service
Platform Team (8 people)	API Gateway, infra, observability
Notification Team (3 people)	Notification Service

> Key principle: "You build it, you run it"

Pattern: Service per Team
Ref: microservices.io/patterns/decomposition/service-per-team.html

Great, We Know the Target. How Do We Get There?

We know:

- FTGO should have 6 services
- We know the boundaries (bounded contexts)
- We know the relationships (context map)

We need: An INCREMENTAL migration strategy

But we CAN'T:

- Stop the business for 6 months to rewrite everything
- Do a "big bang" migration (too risky)
- Extract all services simultaneously (too complex)

Pattern — Strangler Fig Application

Named after the strangler fig tree:

Grows around a host tree

Eventually replaces it entirely

The host tree doesn't die suddenly — it's gradually replaced

Applied to software:

- New features → build as new microservices
- Existing features → gradually extract from monolith
- Route requests to new services when ready
- Monolith shrinks over time until nothing remains

Pattern: Strangler Application
Ref: microservices.io/patterns/refactoring/strangler-application.html

Strangler Fig — Step by Step

01

Identify the service to extract

02

Build the new service alongside the monolith

03

Route a SUBSET of traffic to the new service (shadow/canary)

04

Validate the new service works correctly

05

Route ALL traffic for that function to the new service

06

Remove the old code from the monolith

07

Repeat for the next service

The monolith shrinks with each extraction:

```
[ Monolith ████████████████████ ]
→ [ Monolith ████████████████ ] + [Notif Svc]
→ [ Monolith ██████████ ] + [Notif Svc] + [Restaurant Svc]
→ [ Monolith ██████ ] + [Notif] + [Rest] + [Kitchen]
→ [Order] + [Notif] + [Rest] + [Kitchen] + [Delivery] + [Accounting]
```

FTGO — Which Service to Extract First?

Extraction order (lowest risk first → highest risk last):

Phase 1: Notification Service

1

- Lowest risk — no transactions, no critical data
- Event-driven — just consumes events and sends notifications
- Perfect for learning the extraction process

2

Phase 2: Restaurant Service

Read-heavy, simple CRUD, few dependencies, clear boundary

Phase 3: Delivery Service

3

Clear domain, different tech needs (real-time GPS)

4

Phase 4: Kitchen Service

Moderate complexity, involves saga participation

Phase 5: Accounting Service

5

Financial data — requires careful data migration

6

Phase 6: Order Service (LAST)

- Most complex, most dependencies, hub of all interactions
- Extract only when team has experience from previous extractions

Pattern — Anti-Corruption Layer (ACL)

The problem:

- During migration, new services must talk to the OLD monolith
- The monolith has a different model (legacy naming, old schema)
- If the new service uses the old model, legacy concepts LEAK in

The solution:

- Build a translation layer between old and new
- The ACL converts the legacy model to the new model
- New service stays clean — never sees legacy concepts

FTGO example:

- New Accounting Service has clean model: `{paymentId, amount, status}`
- Legacy monolith has messy model: `{trx_id, amt_in_cents, stat_code, legacy_flag}`
- ACL translates between them

Pattern: Anti-Corruption Layer
Ref: microservices.io/patterns/refactoring/anti-corruption-layer.html

FTGO Anti-Corruption Layers

During the migration, ACLs are needed at these boundaries:

- New Kitchen Service ←[ACL]→ Monolith's Order Module
  - Monolith sends order data in old format
  - ACL translates to Kitchen's ticket model
- New Accounting Service ←[ACL]→ Stripe API
  - Stripe has its own model (charge, refund, dispute)
  - ACL translates to FTGO's payment model
- New Delivery Service ←[ACL]→ Monolith's legacy database
  - During migration, Delivery reads some data from legacy DB
  - ACL translates legacy schema to new Delivery model

After migration is complete:

- ACLs to the monolith are REMOVED (monolith is gone)
- ACLs to external systems STAY (Stripe, Twilio, etc.)

Event Storming — How to FIND Bounded Contexts

Event Storming (Alberto Brandolini):

- A collaborative workshop technique
- Business + tech people in one room
- Use sticky notes to map domain events

Steps:

01

Identify DOMAIN EVENTS (orange stickies)
"OrderPlaced", "PaymentAuthorised", "FoodReady", "FoodDelivered"

02

Identify COMMANDS that trigger events (blue stickies)
"PlaceOrder", "AuthorisePayment", "AcceptTicket"

03

Identify AGGREGATES that handle commands (yellow stickies)
"Order", "Payment", "KitchenTicket", "Delivery"

04

Group related events/commands/aggregates → BOUNDED CONTEXTS emerge

05

Bounded contexts → services

FTGO Events → Contexts → Services

Domain Events discovered:

Order Context:

OrderPlaced, OrderApproved, OrderRejected, OrderCancelled, OrderDelivered

Kitchen Context:

TicketCreated, TicketAccepted, PreparationStarted, FoodReady

Delivery Context:

CourierAssigned, CourierAtRestaurant, FoodPickedUp, FoodDelivered

Restaurant Context:

RestaurantRegistered, MenuUpdated, RestaurantClosed

Accounting Context:

PaymentAuthorised, PaymentCharged, PaymentRefunded

Notification Context:

OrderConfirmationSent, DeliveryUpdateSent, PromotionSent

6 clusters → 6 bounded contexts → 6 services

Anti-Pattern — The Distributed Monolith

What it looks like:

- You have 10 "microservices"
- But they share a database
- You must deploy them all together
- Changing one service breaks three others
- Every feature requires coordinating 4 teams

How it happens:

- Split by technical layer instead of business capability
- Shared database "for convenience"
- Too many synchronous calls between services
- No clear bounded contexts

The result:

- ALL the complexity of microservices
- NONE of the benefits
- Worse than a monolith

Anti-Pattern — Anaemic Services

What it looks like:

- Services are just thin CRUD wrappers over database tables
- No business logic in the service
- All logic lives in an orchestrator or API Gateway

FTGO bad example:

```
OrderService:
  save(order)
  get(orderId)
  update(order)
  delete(orderId)
```

→ Where's the business logic?
→ In the API Gateway.

FTGO good example:

```
OrderService:
  placeOrder()
  approveOrder()
  cancelOrder()
  trackOrder()
```

- Business rules INSIDE the service
- Validates items, checks restaurant hours, calculates total
- The service owns the BEHAVIOUR, not just the data

Anti-Pattern — Chatty Services

What it looks like:

- Services constantly call each other
- One user request triggers 20+ inter-service calls
- Circular dependencies: A → B → C → A

FTGO bad example:

Customer places order:

```
API → Order → Restaurant → Order
→ Accounting → Order → Kitchen
→ Delivery
```

10 synchronous calls for one user action!

Why this happens:

- Wrong boundaries — related logic is split across services
- Missing: events, async messaging, data duplication

How to fix:

- Re-examine boundaries (merge chatty services?)
- Use async events instead of sync calls
- Allow services to cache data they need (CQRS)

Anti-Pattern — The God Service

What it looks like:

- One service that does EVERYTHING
- 50+ API endpoints
- Knows about orders, payments, kitchens, deliveries
- "We have microservices!" (but one service is 80% of the code)

FTGO bad example:

OrderService handles:

- Order creation, approval, cancellation
- Payment processing
- Kitchen ticket management
- Delivery scheduling
- Notification sending

→ It's a monolith with a different name

How to fix:

- Apply decomposition patterns again
- Extract responsibilities into dedicated services

Anti-Pattern — Entity Services (Nanoservices)

What it looks like:

- One service per database entity
- CustomerService, AddressService, PhoneService, MenuItemService
- Too fine-grained, too many services
- Every operation requires calling 5 services

FTGO bad example:

Get restaurant details:

```
RestaurantService
→ AddressService
→ MenuService
→ OperatingHoursService
```

Four calls for one screen!

Why it's wrong:

- Services should map to CAPABILITIES, not ENTITIES
- Restaurant info is ONE capability — one service
- Address, menu, hours are part of the Restaurant context

How to Validate Your Service Boundaries

Ask these questions for each proposed service:

- ☐ Does it map to a clear business capability?
- ☐ Does it have its own bounded context with distinct ubiquitous language?
- ☐ Can it be developed, deployed, and scaled INDEPENDENTLY?
- ☐ Does it own its data exclusively (no shared database)?
- ☐ Can a single team (5-9 people) own it?
- ☐ Is inter-service communication MINIMAL (not chatty)?
- ☐ Does it have business logic (not just CRUD)?
- ☐ Can it handle requests without synchronously depending on others?

If you answer NO to 3+ questions → re-examine your boundaries.

Key Takeaways

1 BOUNDARIES are the most critical decision in microservices

Right boundaries → success. Wrong boundaries → distributed monolith.

2 DDD provides the tools for finding boundaries:

- Subdomains → classify investment (Core, Supporting, Generic)
- Bounded Contexts → define service boundaries
- Ubiquitous Language → same word, different meaning in different contexts
- Context Maps → define service relationships

3 Four decomposition patterns:

- Decompose by Business Capability
- Decompose by Subdomain
- Self-Contained Service (avoid sync dependency chains)
- Service per Team

4 Two migration patterns:

- Strangler Fig → incremental extraction
- Anti-Corruption Layer → protect new services from legacy models

5 Five anti-patterns to AVOID:

Distributed Monolith, Anaemic Services, Chatty Services, God Service, Entity Services

---

Pattern References:

- microservices.io/patterns/decomposition/
- microservices.io/patterns/refactoring/

Coming Next: Module 4 — Day 1 Consolidation & Decision Framework