Taming Complexity in Distributed Systems

What's behind Revolut success

Revolut

Hi!
I am Donato

Partner and Head of Technology at Revolut

in donatolucia

"

Revolut hypergrowth?

2017

1M
Customers

35K
Sign-ups per month

30
Live countries

5M
Monthly user transactions

2025

55M
Customers

1.5M
Sign-ups per month

48
Live countries

1.2B
Monthly user transactions

"

Hypergrowth - what’s behind?

2017

1
Repository

5
Production services

1
Database cluster

15
Engineers

2025

1.9K
Repositories

4K
Production services

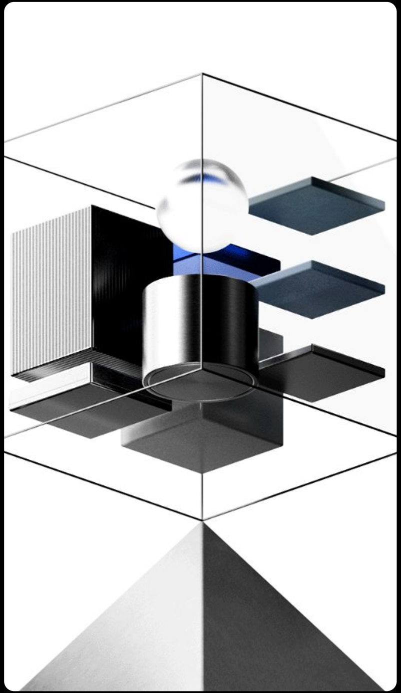
700
Database cluster

1.4K
Engineers

From Simplicity

To Chaos

Accidental Complexity



Not inherent to business logic

Caused by poor structure, over-flexibility

Manifests as:

- Runtime risks (scalability, resilience)

- Dev time risks (Unpredictable code placement, complexity of design decision making, Operational overhead)

Accidental Complexity = Operational Risk

"

The Solution?

A convention-based Architecture framework

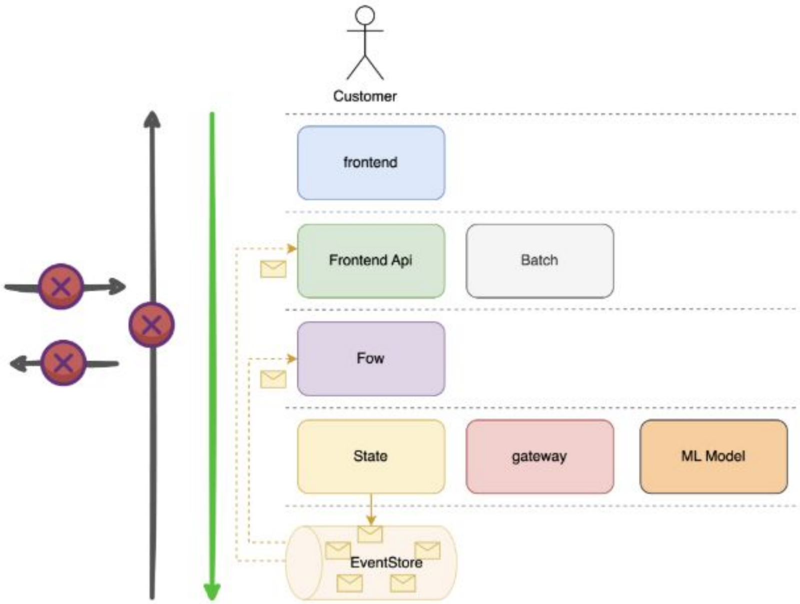
Less choices

Less entropy

Less complexity

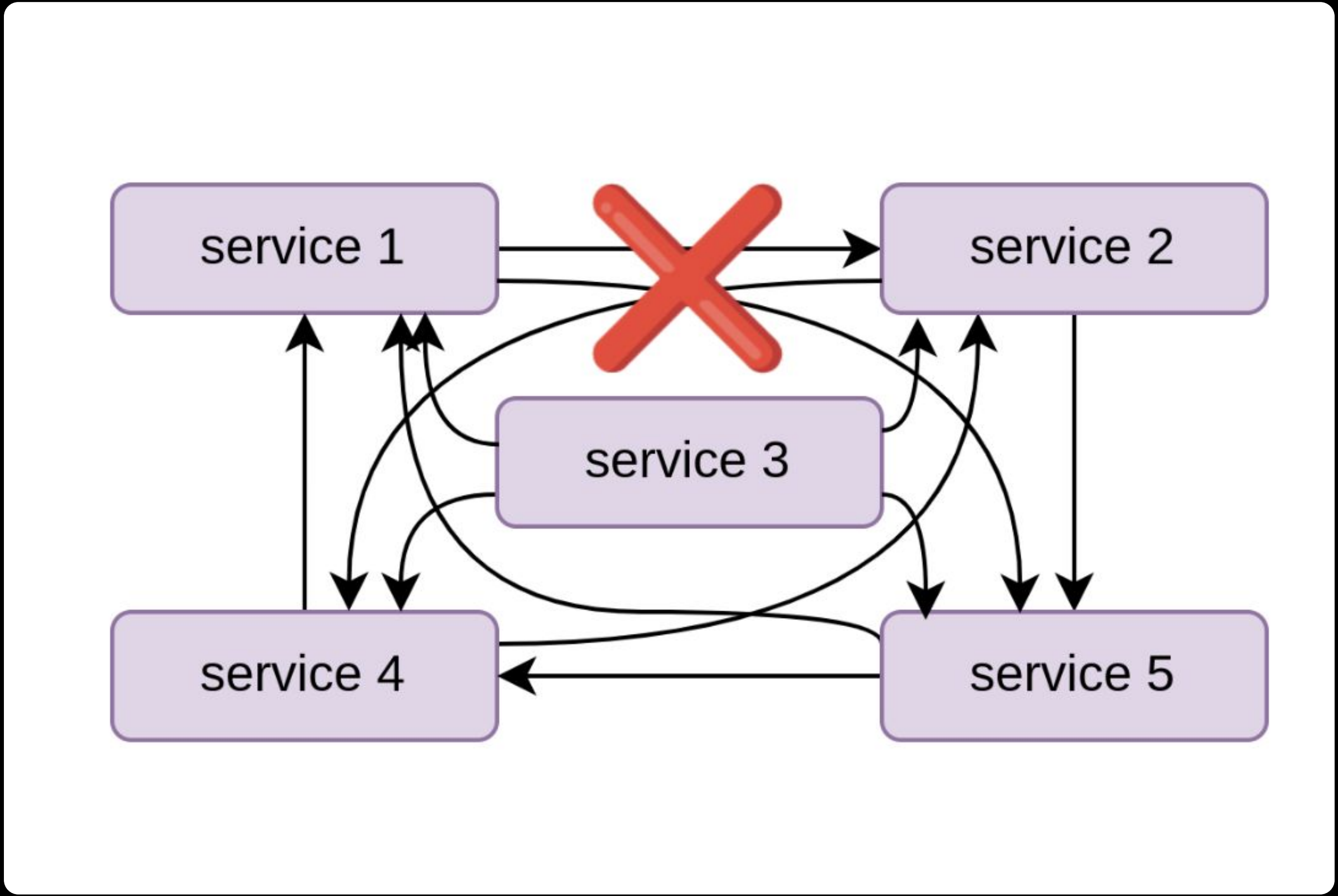
Less risks

More speed



Revolut service topology

Frontend API	To be used by frontend only (ie mobile or Web)
Flow	Orchestrates a business flow/process Typically depends on other services - state, gateway (deps must be below in the stack)
State	Provides API to manage domain state Supports many use cases Always has a database
Gateway	A portal to a 3rd party system
ML Model	Stateless service that hosts machine learning models. It transforms inputs into outputs for predictive tasks based on patterns learned during training process
Batch	Batch data processing on a defined schedule
Eventstore (supporting event-driven architecture)	All state components publish change events, any other components can subscribe to any events

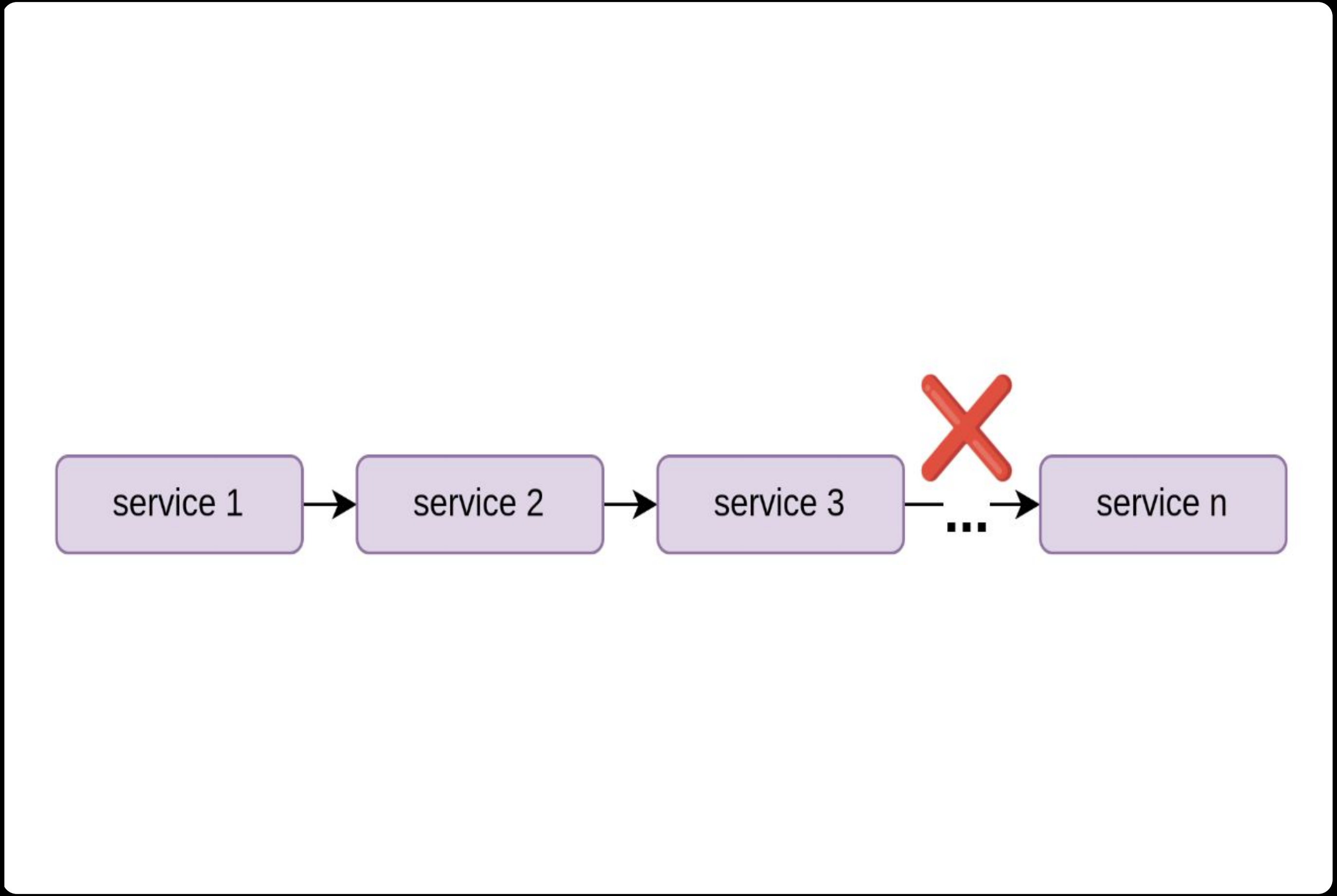


Framework Invariants

Restricted component integrations

Direction of integrations is predefined

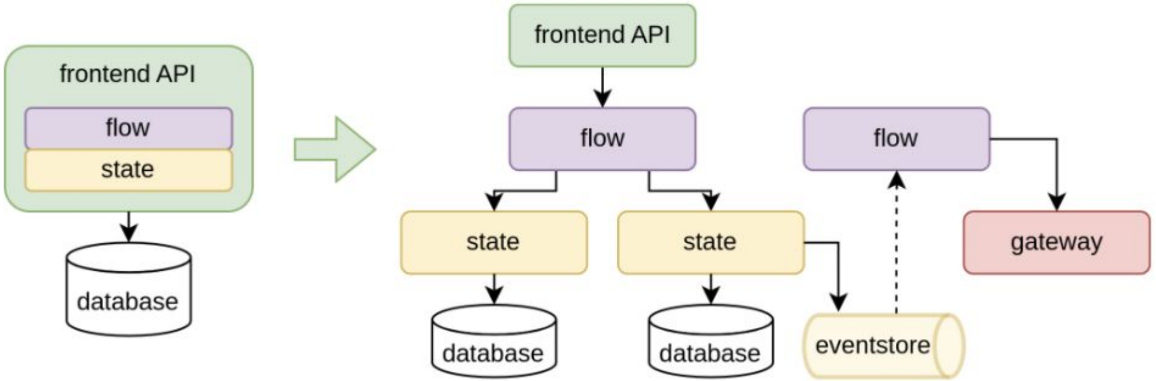
No cyclic dependencies



Framework Invariants

Minimal delegation

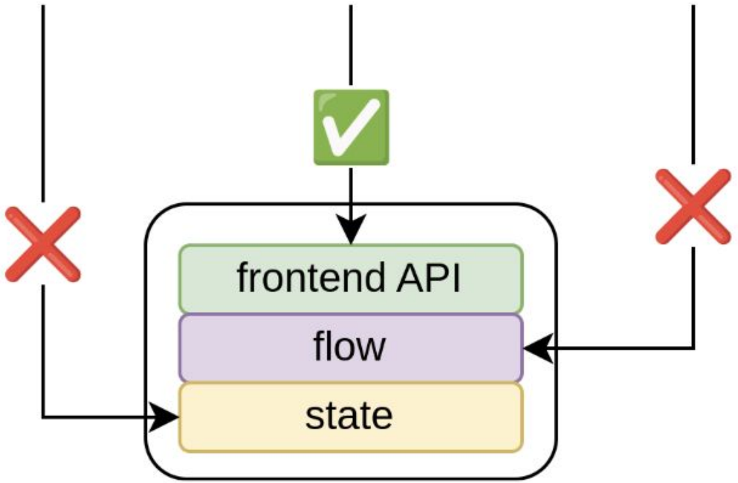
Maximum 3 layers of delegation down the service stack



Framework Invariants

Start simple and evolve as necessary

Start with Frontend API as a monolith, than extract flow/state/gateway



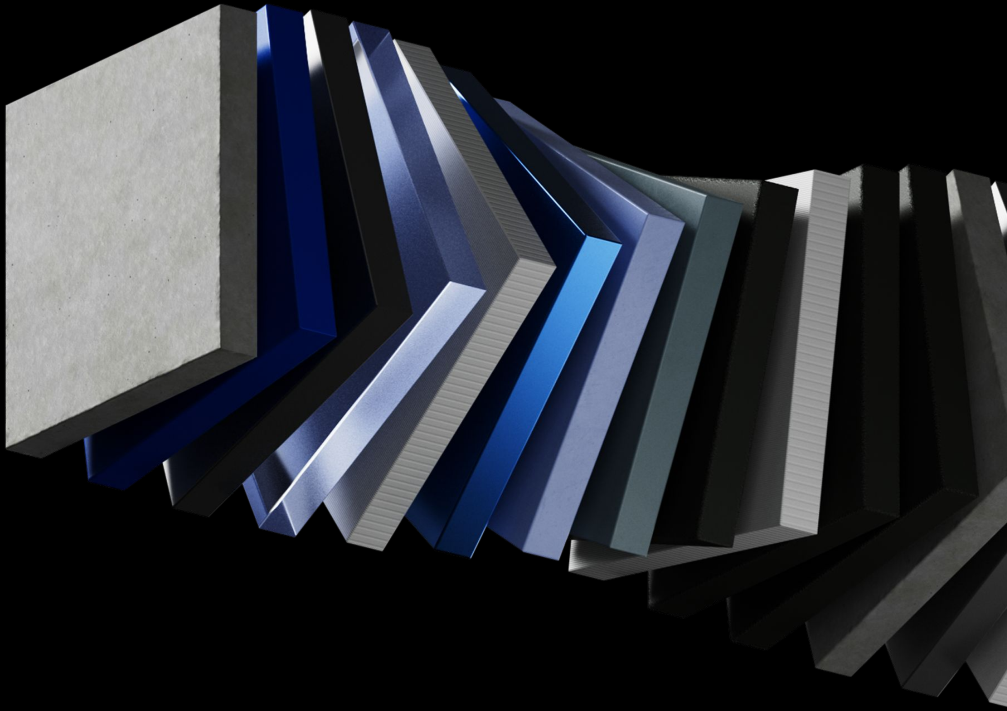
Framework Invariants

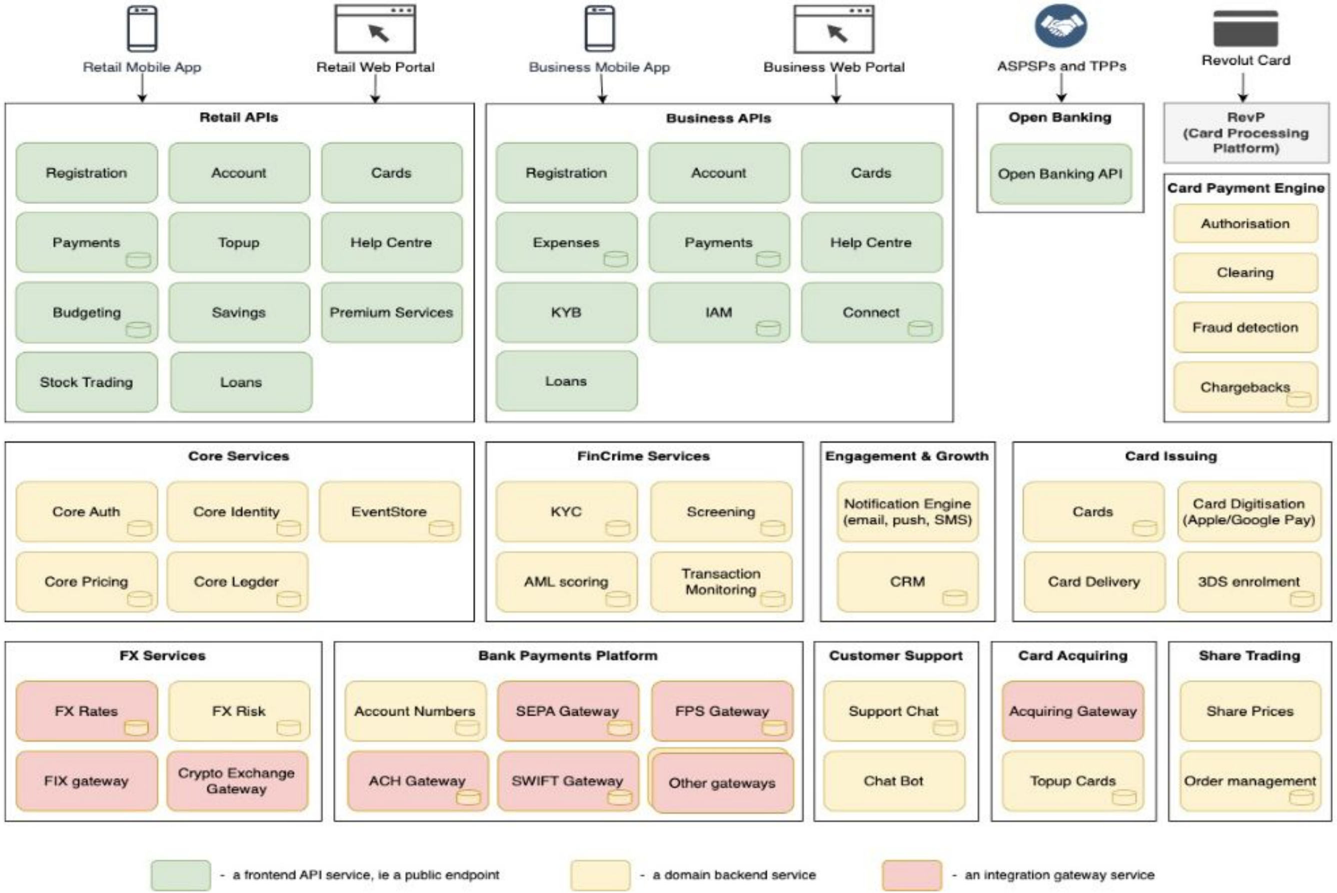
Encapsulation

Expose only relevant contract

Extract smaller "matryoshkas" as required for reuse or for scaling

Revolut service topology
In practice





Retail Mobile App → Retail APIs

Retail Web Portal → Retail APIs

Business Mobile App → Business APIs

Business Web Portal → Business APIs

ASPSPs and TPPs → Open Banking

Revolut Card → RevP (Card Processing Platform)

Retail APIs
- Registration
- Account
- Cards
- Payments
- Topup
- Help Centre
- Budgeting
- Savings
- Premium Services
- Stock Trading
- Loans

Business APIs
- Registration
- Account
- Cards
- Expenses
- Payments
- Help Centre
- KYB
- IAM
- Connect
- Loans

Open Banking
- Open Banking API

RevP (Card Processing Platform)

Card Payment Engine
- Authorisation
- Clearing
- Fraud detection
- Chargebacks

Core Services
- Core Auth
- Core Identity
- EventStore
- Core Pricing
- Core Legder

FinCrime Services
- KYC
- Screening
- AML scoring
- Transaction Monitoring

Engagement & Growth
- Notification Engine (email, push, SMS)
- CRM

Card Issuing
- Cards
- Card Digitisation (Apple/Google Pay)
- Card Delivery
- 3DS enrolment

FX Services
- FX Rates
- FX Risk
- FIX gateway
- Crypto Exchange Gateway

Bank Payments Platform
- Account Numbers
- SEPA Gateway
- FPS Gateway
- ACH Gateway
- SWIFT Gateway
- Other gateways

Customer Support
- Support Chat
- Chat Bot

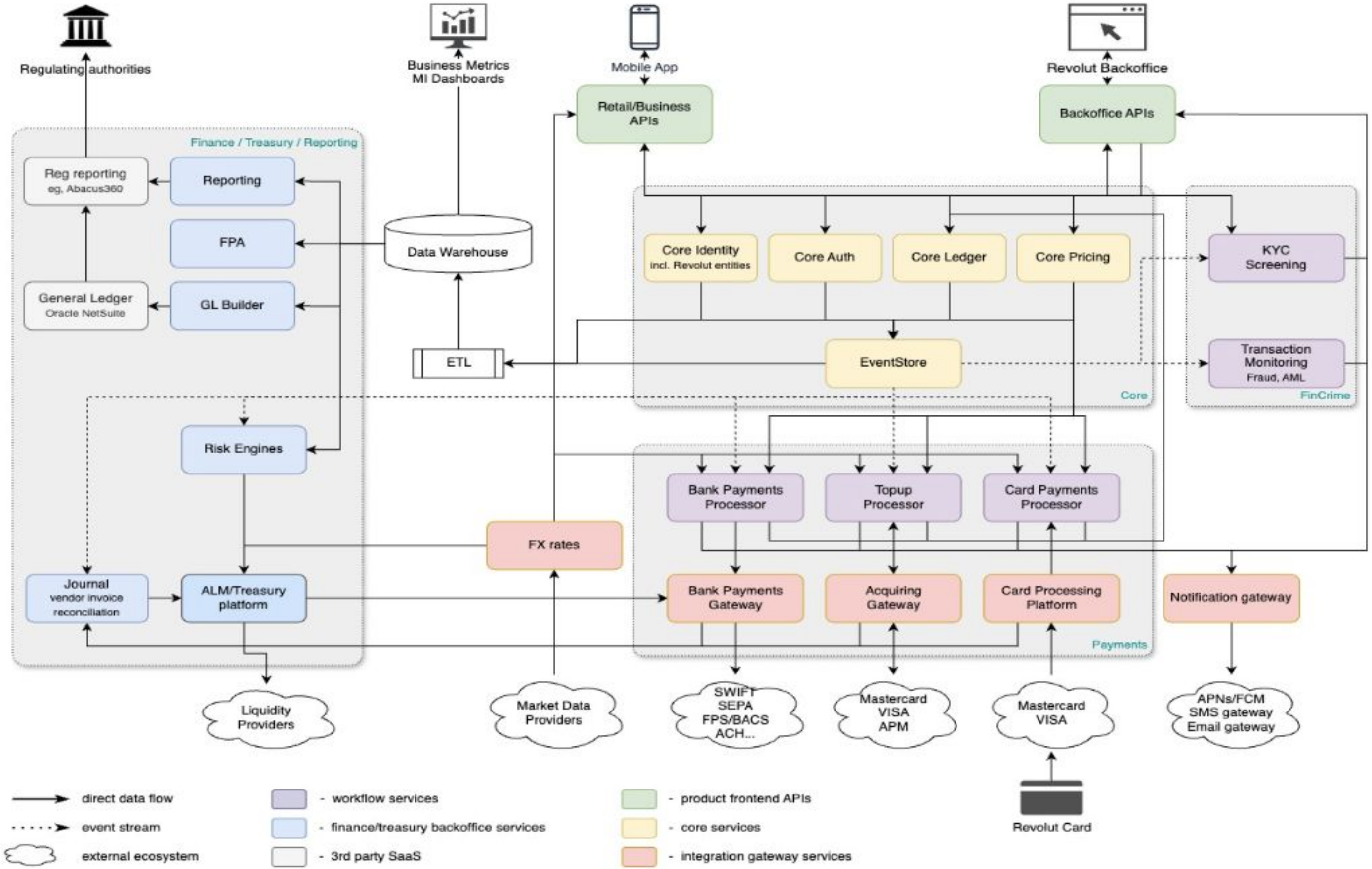
Card Acquiring
- Acquiring Gateway
- Topup Cards

Share Trading
- Share Prices
- Order management

🟩 - a frontend API service, ie a public endpoint

🟨 - a domain backend service

🟥 - an integration gateway service



Regulating authorities

Business Metrics
MI Dashboards

Mobile App

Revolut Backoffice

Retail/Business
APIs

Backoffice APIs

Finance / Treasury / Reporting

Reg reporting
eg. Abacus360

Reporting

FPA

General Ledger
Oracle NetSuite

GL Builder

Data Warehouse

ETL

Risk Engines

Journal
vendor invoice
reconciliation

ALM/Treasury
platform

FX rates

Core Identity
incl. Revolut entities

Core Auth

Core Ledger

Core Pricing

EventStore

Core

KYC
Screening

Transaction
Monitoring
Fraud, AML

FinCrime

Bank Payments
Processor

Topup
Processor

Card Payments
Processor

Bank Payments
Gateway

Acquiring
Gateway

Card Processing
Platform

Notification gateway

Payments

Liquidity
Providers

Market Data
Providers

SWIFT
SEPA
FPS/BACS
ACH...

Mastercard
VISA
APM

Mastercard
VISA

APNs/FCM
SMS gateway
Email gateway

Revolut Card

direct data flow

event stream

external ecosystem

- workflow services

- finance/treasury backoffice services

- 3rd party SaaS

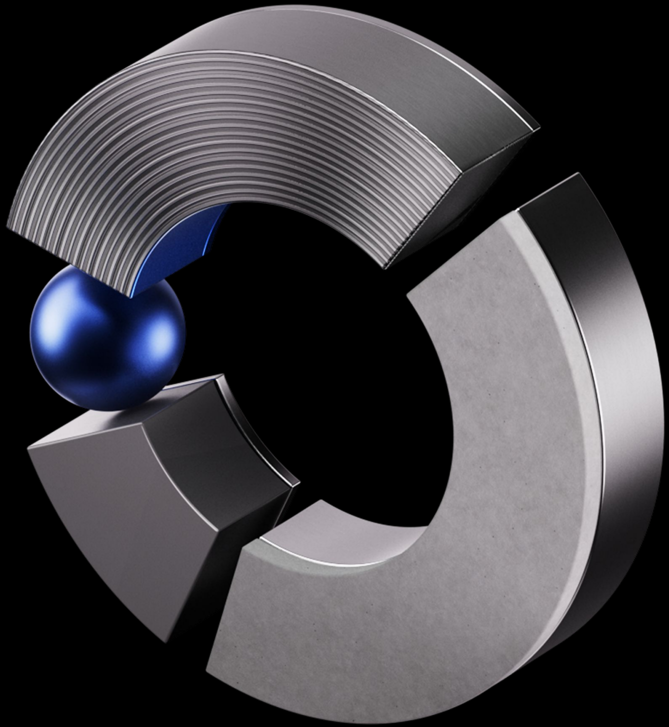
- product frontend APIs

- core services

- integration gateway services

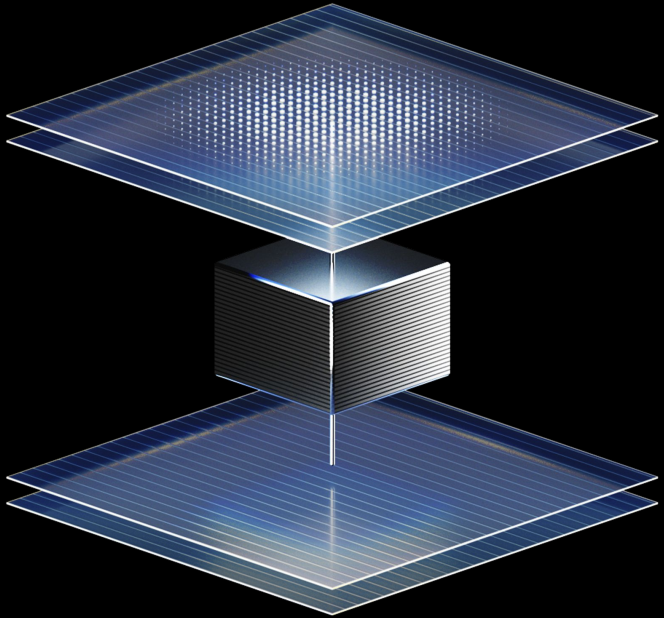
"

Takeaway?



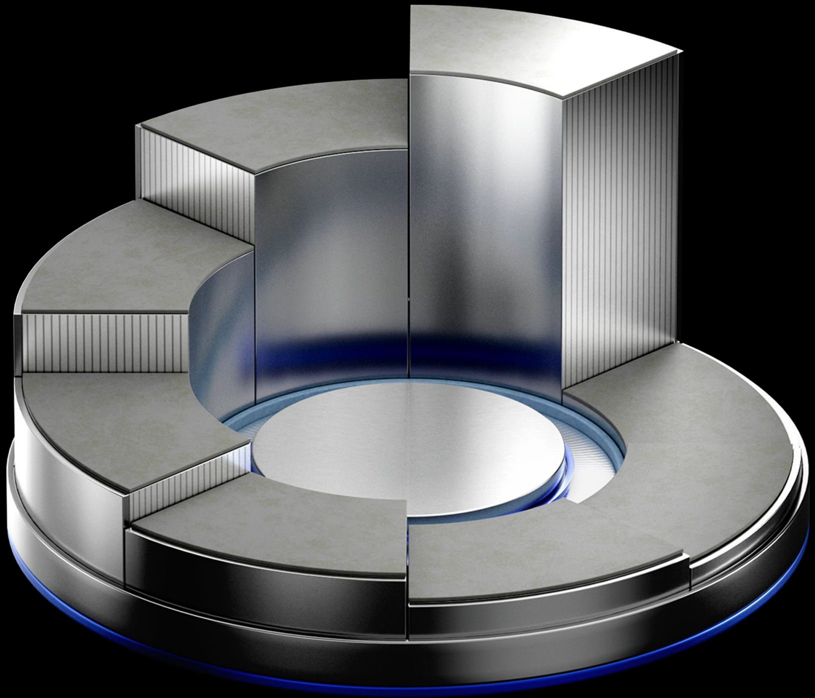
Replicable solution

- Ubiquitous architecture language: verbal and graphical
- Constraints push for faster design decisions (less choices)
- Cross-platform: same for Java, Python, <your favourite language>



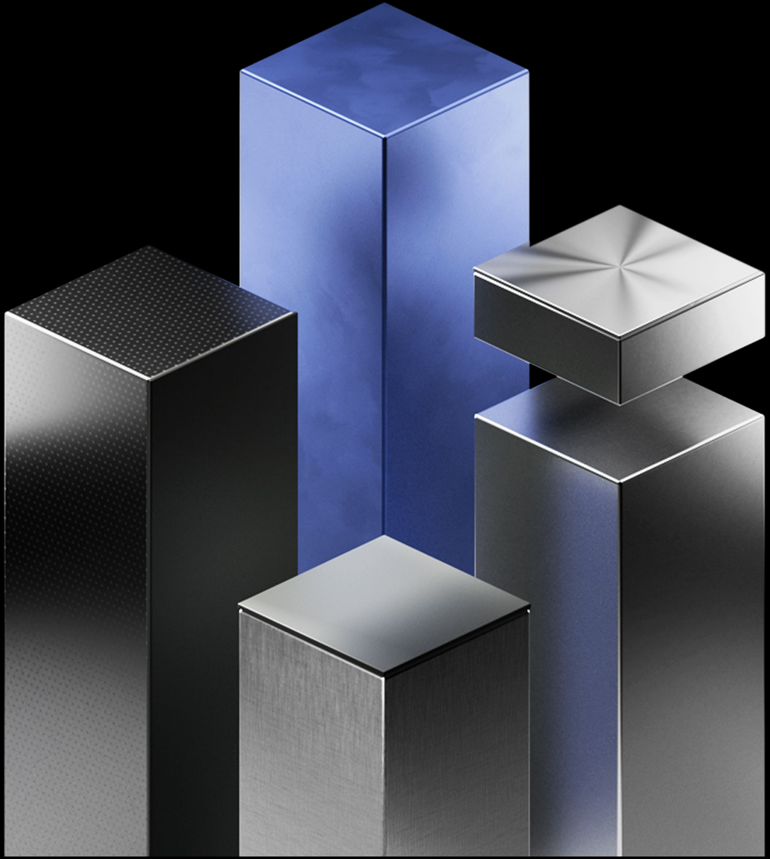
Placement of code abstractions

- Code modules organised according to (sub)domains and service layers
- Domain entities in state modules: read and write APIs
- Multistep business flows/orchestration in flow modules
- User authentication/authorisation Frontend API



Scalability levers

- State in database ⇒ read/write load separation, replicas, caching
- Stateless flows ⇒ Horizontal auto-scaling behind LB, or parallelisation of event processing or batches
- Gateway ⇒ defined by the 3rd party API behind it, has to compensate drawbacks
- Frontend API ⇒ horizontal auto-scaling behind LB



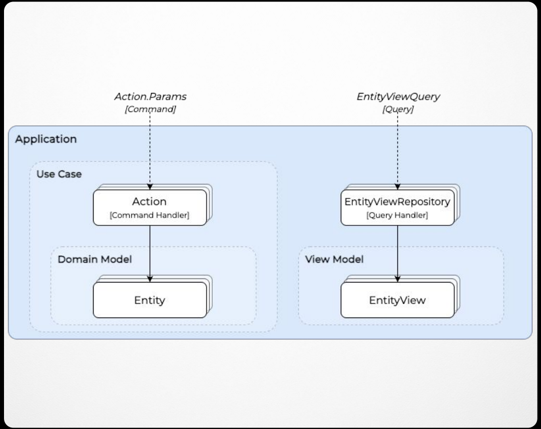
Resilience levers

- Timeouts, retries, standard response codes...
- Security
  - only frontend API can be exposed via a public endpoint
  - only gateways can connect to external systems

"

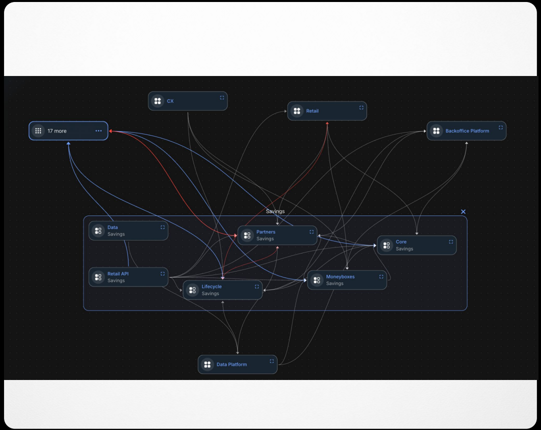
Is this it?

Tooling to Support Framework



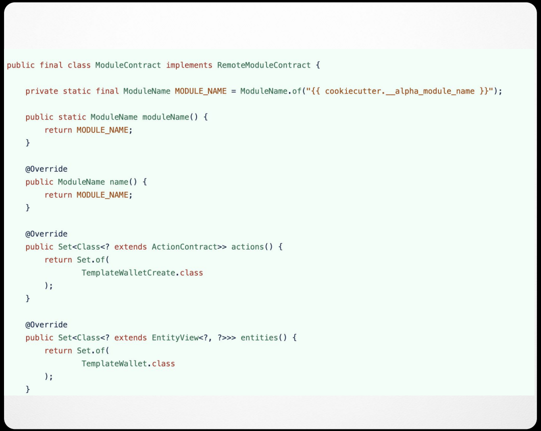
Alpha

DDD Framework



Tower

Infra UI, Governance



```java
public final class ModuleContract implements RemoteModuleContract {

    private static final ModuleName MODULE_NAME = ModuleName.of("{{ cookiecutter.__alpha_module_name }}");

    public static ModuleName moduleName() {
        return MODULE_NAME;
    }

    @Override
    public ModuleName name() {
        return MODULE_NAME;
    }

    @Override
    public Set<Class<? extends ActionContract>> actions() {
        return Set.of(
                TemplateWalletCreate.class
        );
    }

    @Override
    public Set<Class<? extends EntityView<?, ?>>> entities() {
        return Set.of(
                TemplateWallet.class
        );
    }
```

Predefined templates

for all service types

To summarize

- Complexity is inevitable, but accidental complexity isn't.
- A convention-based approach gave us speed, reliability, and safety.
- Structure enabled scale — not the other way around.

Thank you