Modular Monolith

Microservices

Microservices

An **architectural style**

that structures **an application**

as a collection of **small**,

loosely coupled, **autonomous** services,

organized around **busine$$ capabilities**,

owned by a **small team**

History

- **Early 2000s**: Amazon and Netflix started breaking down their monolithic apps into smaller more manageable pieces
- **2011-2012**: Martin Fowler and James Lewis started talking about "Microservices"
- **2014**: M.Fowler and J.Lewis popularized it via talks and terms
- **2015-2020**: Organizations started adopting them + Docker/k8s
- **2018-...**: Microservices is a hype-driven decision ⚠
- **Today**: mainstream; serverless and service mesh appeared

Benefits of Microservices

- ✓ **Faster Time-to-Market**** => 😁 Business
  if independently deployable by autonomous👑 teams
- ✓ **Lower Cognitive Load**** => 😁 Developers
  if small, business-aligned microservices with clear boundaries
- ✓ **Scalability** for the hot🔥 parts that require it
- ✓ **Availability**: fault-tolerance to partial failures
- ✓ **Technology Stack Freedom**
  Safer to upgrade/change the language/library
- ✓ **Security / Privacy** (GDPR) / Compliance

** **a Modular Monolith can** also give you these

Drawbacks of Microservices

- Development Complexity
- Eventual Consistency
- Network Latency & Reliability
- Deployment Rampage
- Distributed Tracing & Debugging

What's faster to build from scratch?

One Monolith or **3 Microservices**

ld from scratch?

Growing Coupling
eventually starts
slowing us down

Microservices

Big Ball of Mud

(BBoM pattern)

= a haphazardly structured,
sprawling, sloppy, duct-tape, spaghetti-code jungle,

showing signs of unregulated growth,
and repeated, expedient repair.

- Brian Foote and Joseph Yoder

Developed by a too large team (eg 20 devs),
with code in need of refactoring, but lacking tests.

https://wiki.c2.com/?BigBallOfMud

From a BBoM to Microservices - Strategies

1) **Big-Bang Rewrite** of a 2M lines, 12 years-old codebase

- Start from scratch, re-gather all requirements
- Changes to old system: (a) reject, (b) delay 1 year, or (b) cost*2 (old+new)
- **20-30% success rate. Worse: likely to turn into a Distributed Monolith** 😱

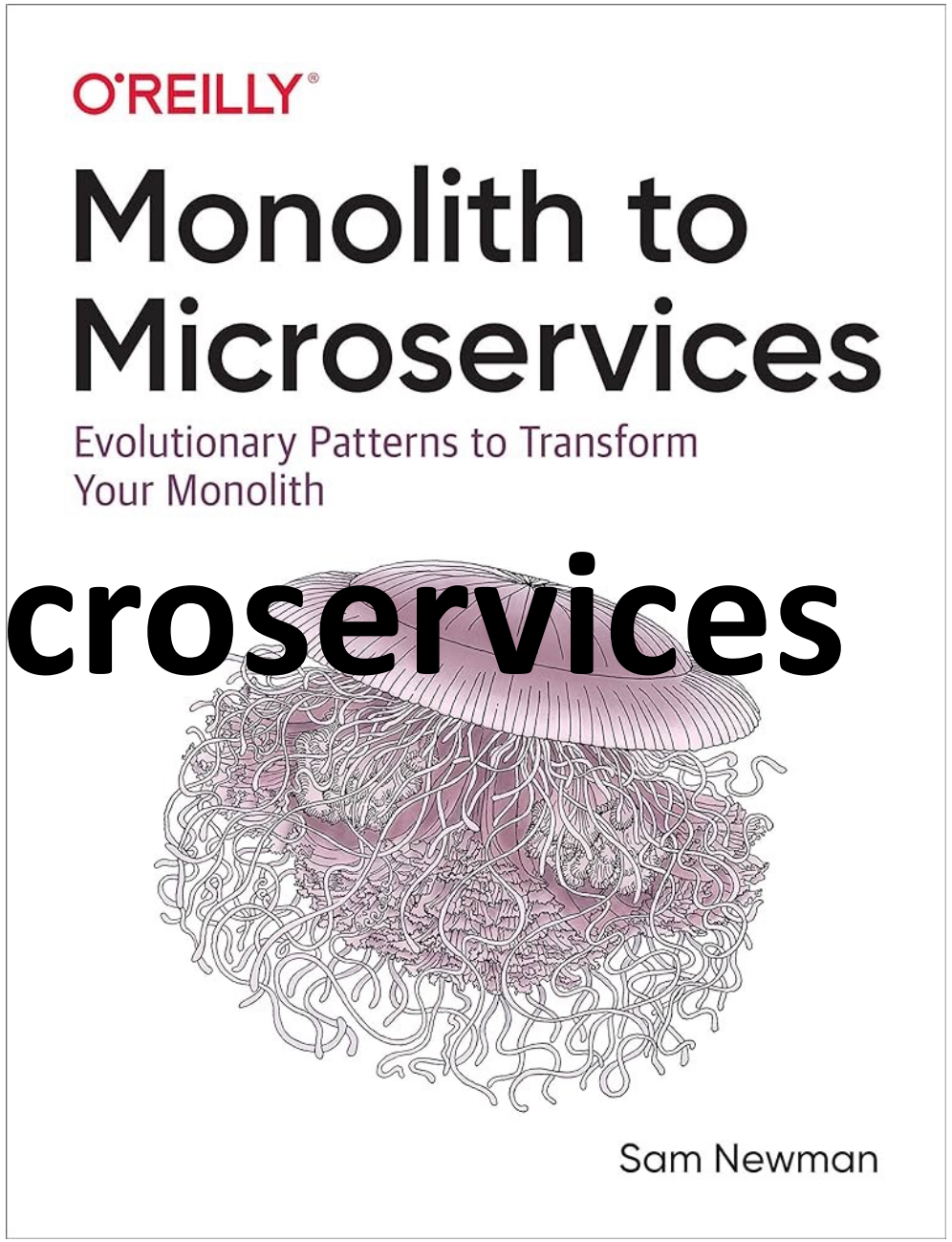
2) Strangler-Fig Pattern (outside-in) ✅

- Move behind a proxy **simple features** to separate microservices

3) Split in Modules > Extract (inside-out) ✅

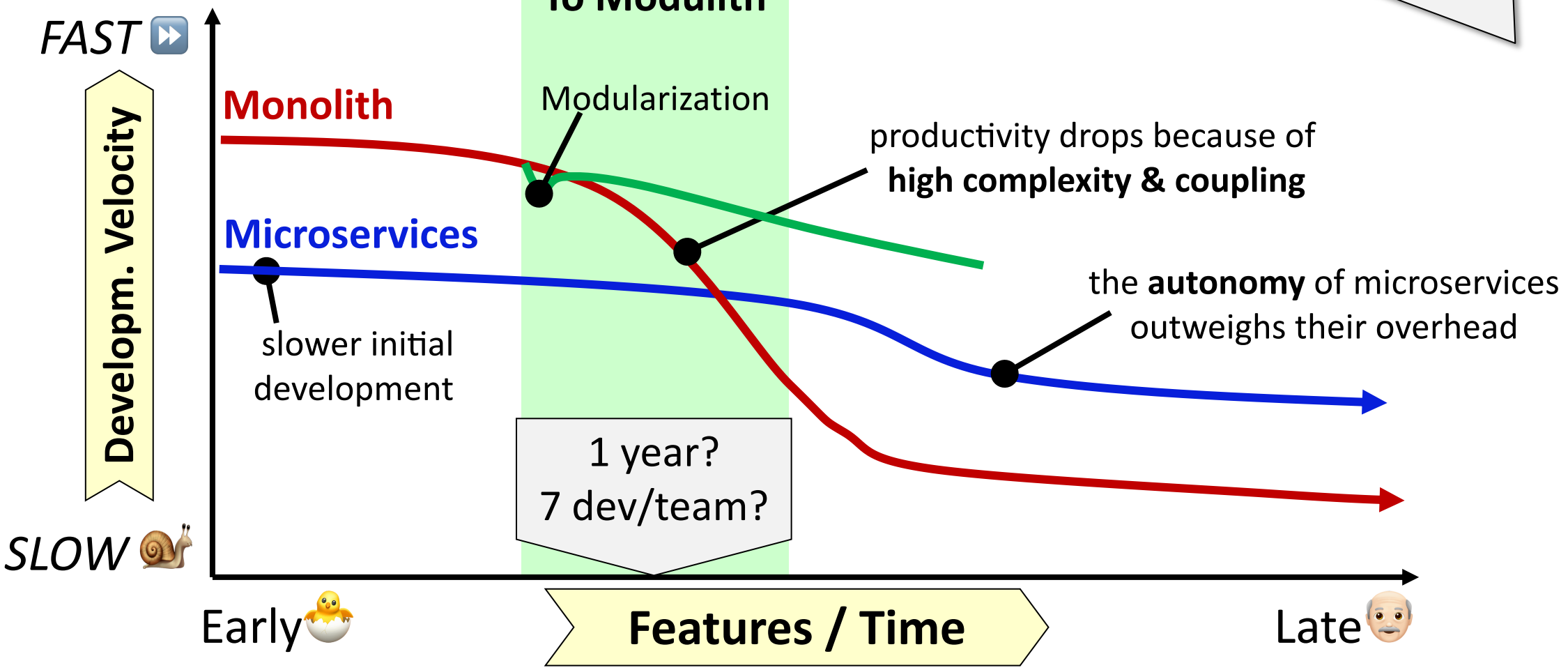
- **Progressively decouple** logic and data in the monolithic codebase

Monolith to Microservices



Productivity

My system **will be complex**, so let's start with microservices!



You should NOT start a new project with microservices, even if you're sure your application will be big enough to make it worthwhile. **- Martin Fowler**

https://martinfowler.com/bliki/MonolithFirst.html (2015)

principal microservice evangelist

Instead: start simple and address the system's natural bottlenecks as they occur
(functional, social, technical)

😔😔😔

Meanwhile, in Reality: If we don't start with microservices, THEY (the bosses) **won't give us the time we need to break it** into microservices later = **WASTE**

Monolith VS Microservices

Can we have the best of both worlds?

Deployment architecture of a **Monolith**

Logical decoupling of **Microservices**

+ Ability to **rapidly extract** a Microservice out when necessary

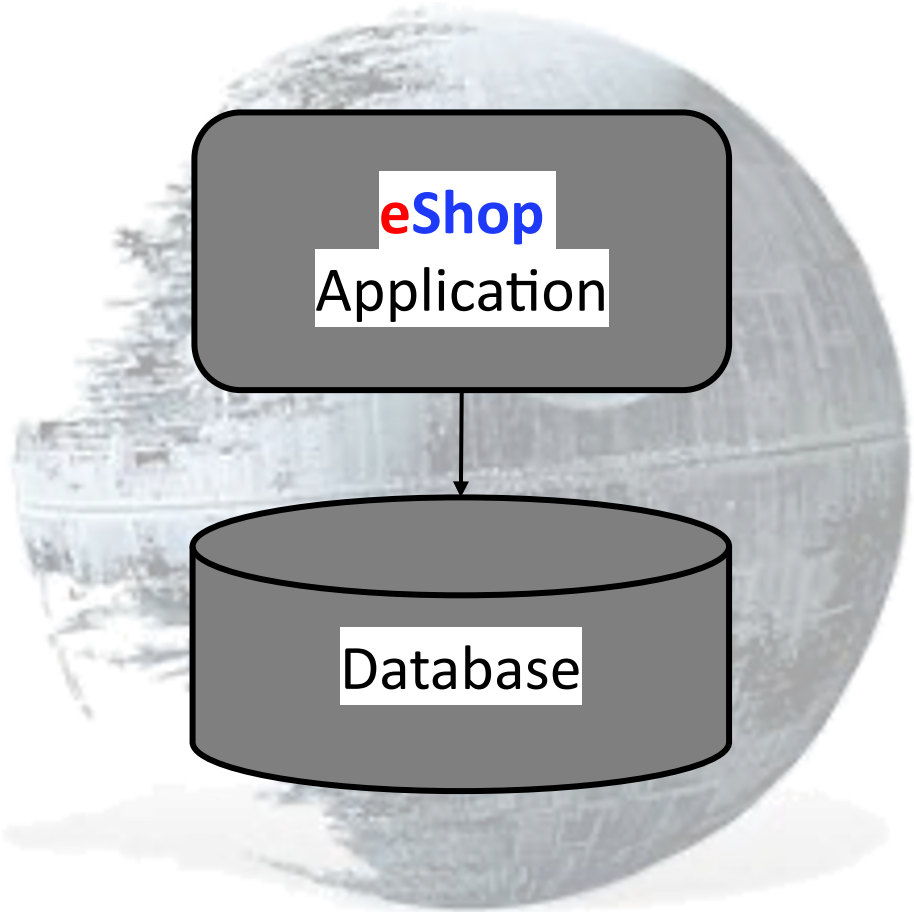
Modular Monolith

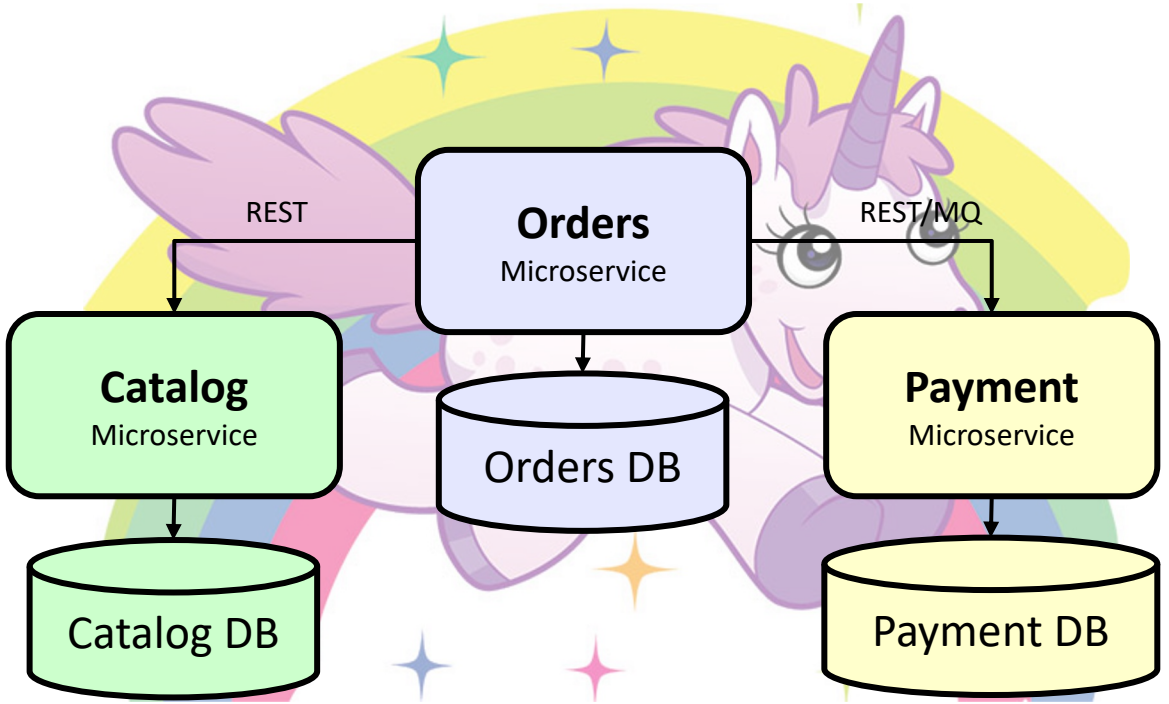
Modulith

BBoM Monolith

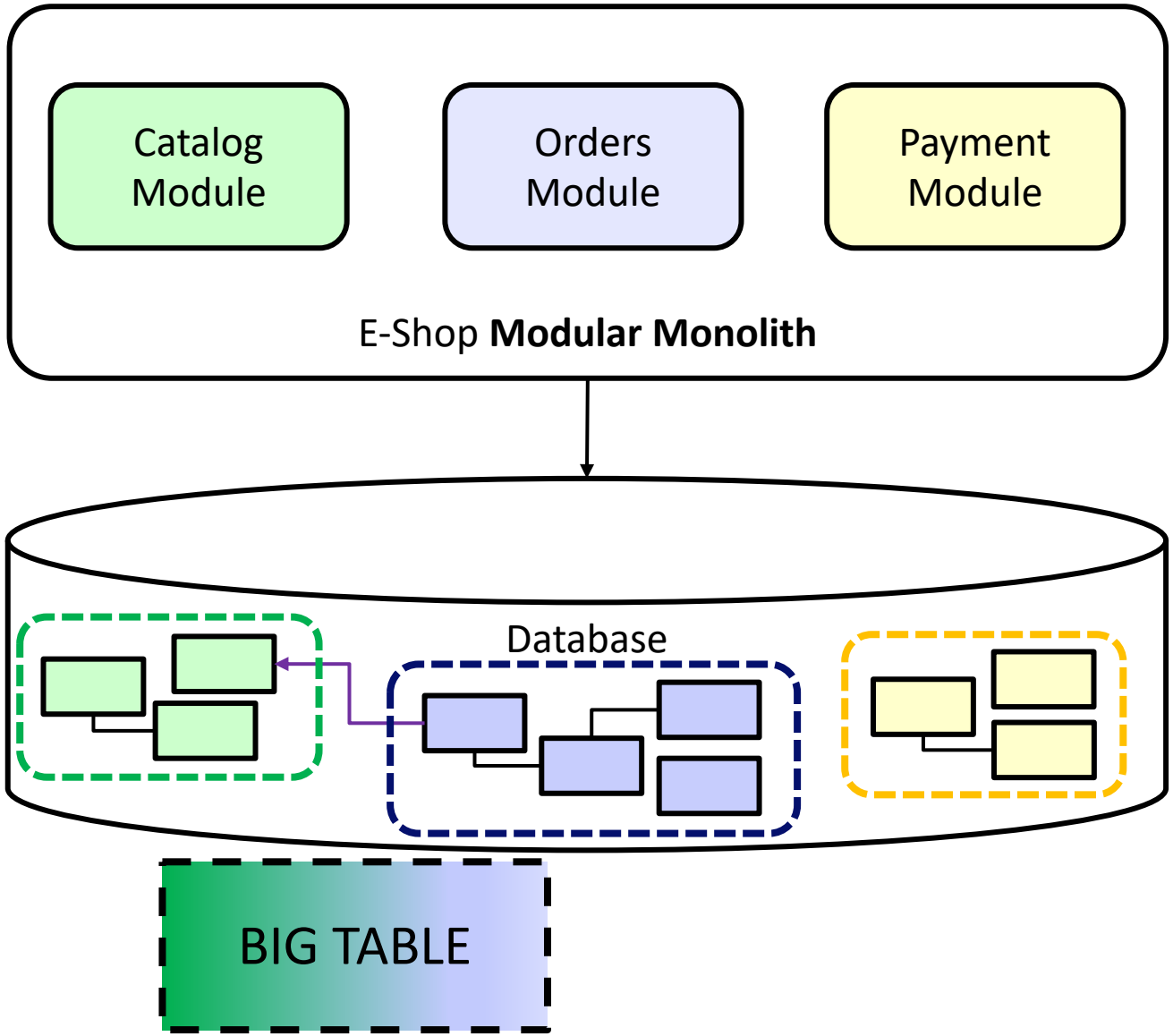
vs

Microservices





Modular Monolith



What is a Module?

= a stand-alone logical application, having its own:

- **business features** to offer to users
- **private implementation**: domain model&logic, Rest Apis, infra
- **public API**:
  - **internal**: **for other modules**, via in-memory **calls** or **events**
  - **external**: **for other systems**, via REST, Rabbit, Kafka...
- **private persistence** in its own DB schema
- micro-**frontend**: screens & shared components (monorepo)

Life in a Modulith

independent teams working on **decoupled modules**

Example #1 (ehealth)

8 teams x 6-7pax {4-5 DEV+FE+PO+QA},

owning ~4 modules {with private DB schema}

+ 2 x Product Architects (Functional) + 3 x Core Team ("Platform")

CI build takes ≤15m (hard-core tuning involved)

Deploying in production 1/week

Example #2 (pharmacy)

5 teams x 3-4 DEV owning ~4 modules

CI = 30 minutes

CD = 2 weeks

Modulith Challenges

Finding correct module boundaries

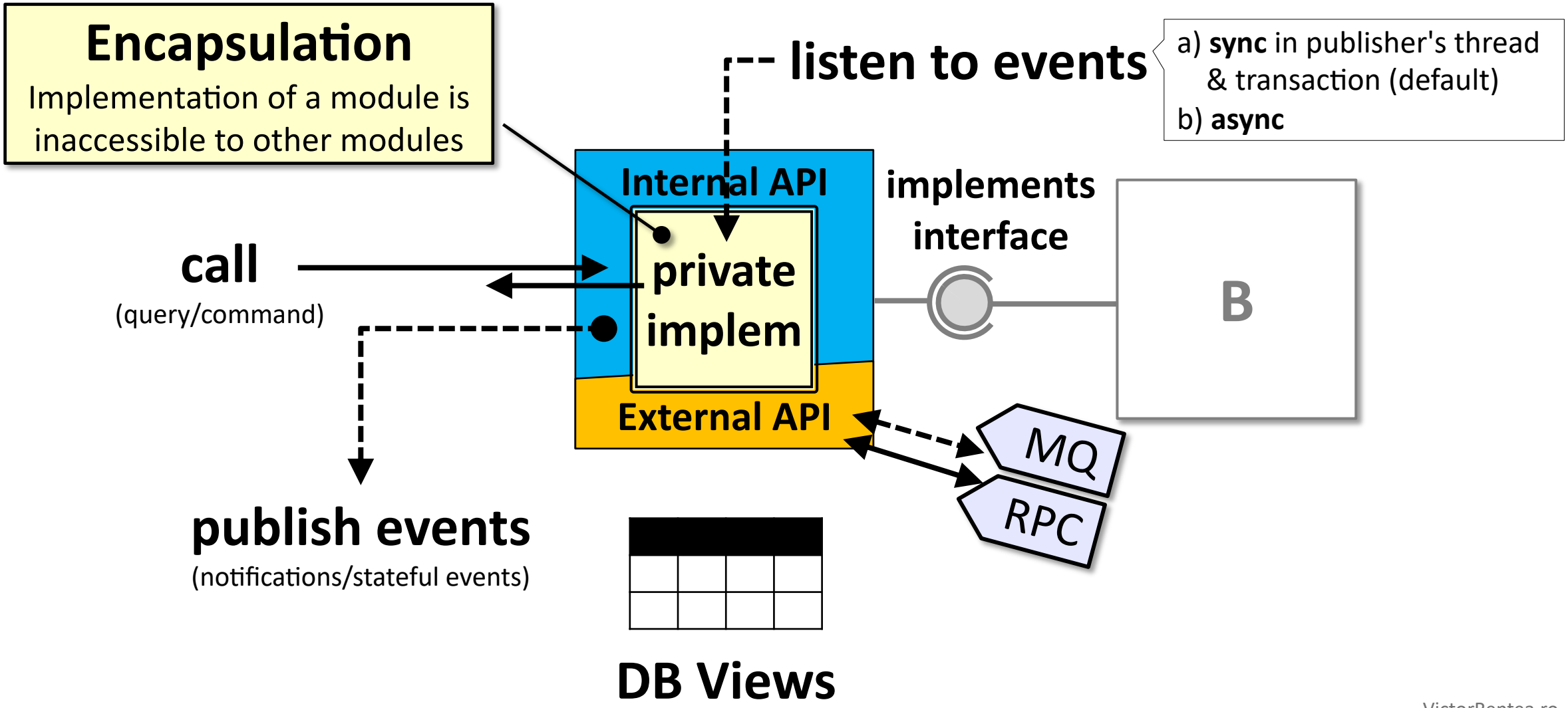
Encapsulation of Modules

Data Decoupling

Testing

Module Public API

internal (for other modules)
external (for other systems)



The Screaming Architecture

Top-level packages reflect your domain structure

com.myorg.myapp

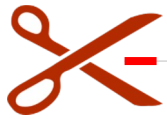
.entity

.service .order

.repo

...

separate modules



.product .service

...

https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html

Enforce Boundaries

- **Separate Build Units:** eg Maven/Gradle modules
  - *customer-impl* depends on *customer-api* + the *order-api* it calls/listenes
- **ArchUnit** www.archunit.org

```
@Test // unit test on CI
public void independentSubdomains() {
  slices().matching("..myapp.(*)..*")
      .should().notDependOnEachOther()
      .ignoreDependency(resideInAnyPackage("..shared.."))
      .check(new ClassFileImporter().importPackages("com.myapp"));
}
```

- **Spring Modulith** spring.io/projects/spring-modulith

```
@Test
void verifyModularity() { // uses ArchUnit rules under the hood
  ApplicationModules.of(ModulithApp.class).verify();
}
```

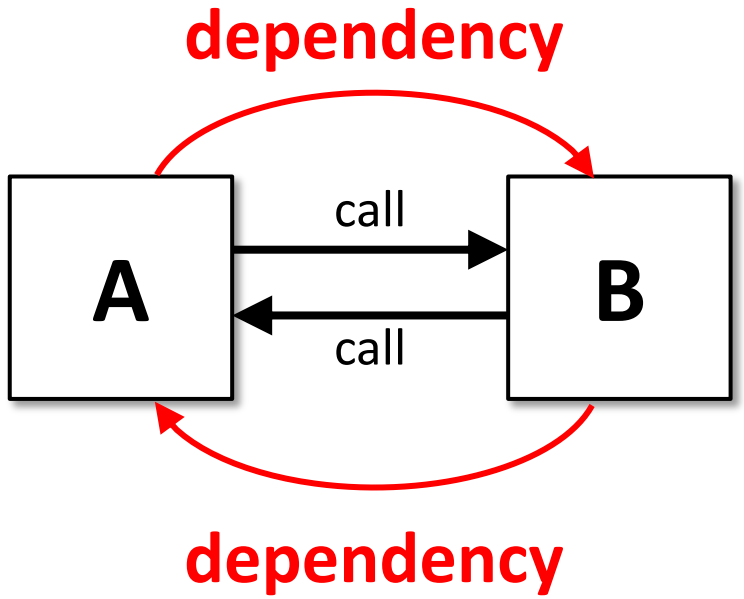
Architecture is the art of deferring decisions

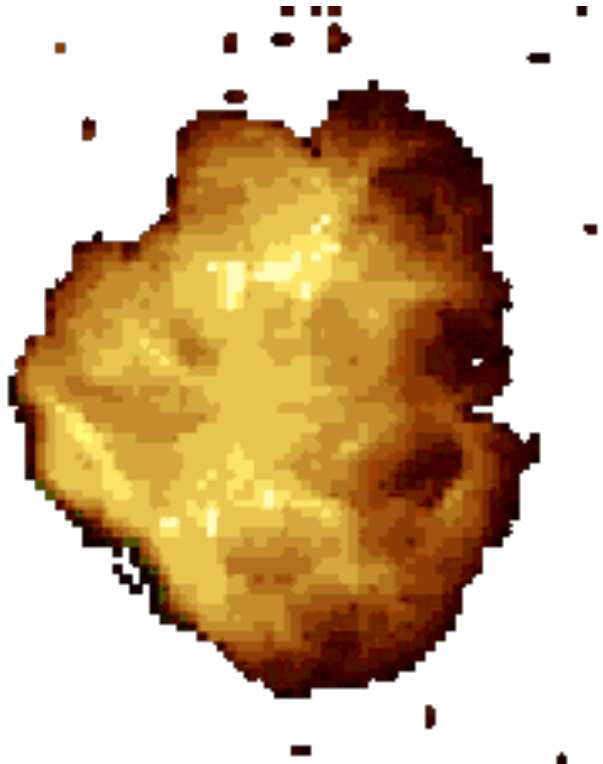
Why makes Modulith more Maintainable?

.... than a Big Ball of Mud?

1. **Decoupled Logic** via clear Module APIs
   - Less accidental coupling
   - Lower cognitive load per module
2. **Smaller Domain Model** can be more specialized
   - catalog.Product vs order.Product
3. **Separate Persistence** in private schema/module
   - More constraints in DB
   - Control over tables: can keep them in sync with Domain remodeling

Circular dependencies between 2 modules

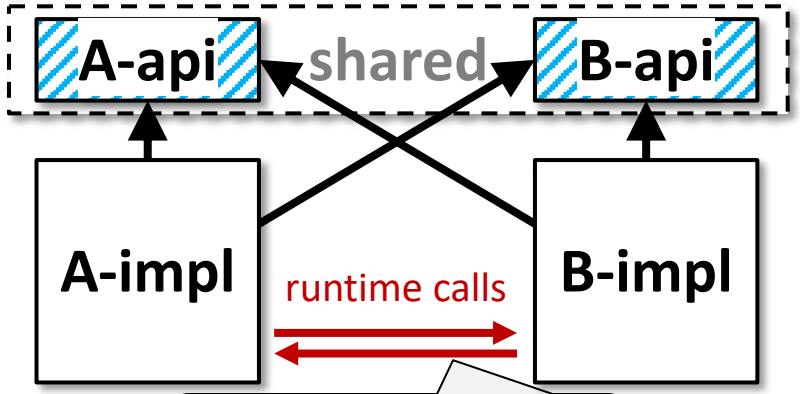




= Tight coupling

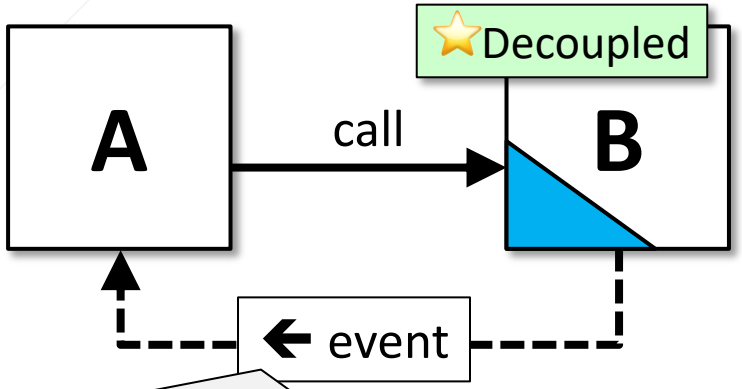
- **Build fails** for separate compilation units (eg maven/gradle)

Extract API as separate module



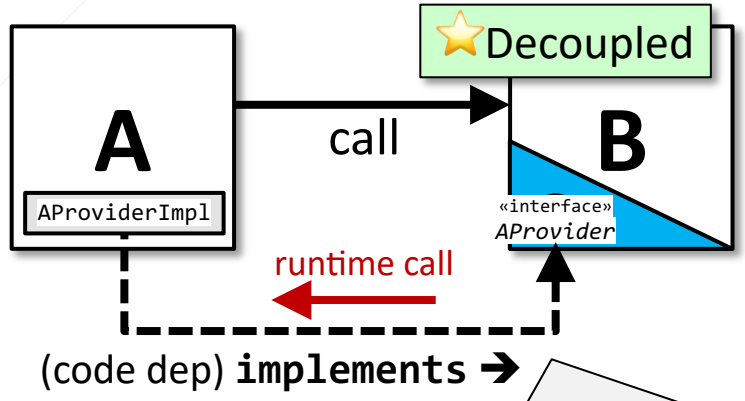
Allow the call cycle, just fix the code cycle

Publish Events (in-mem)



Module A must **react to changes** in B

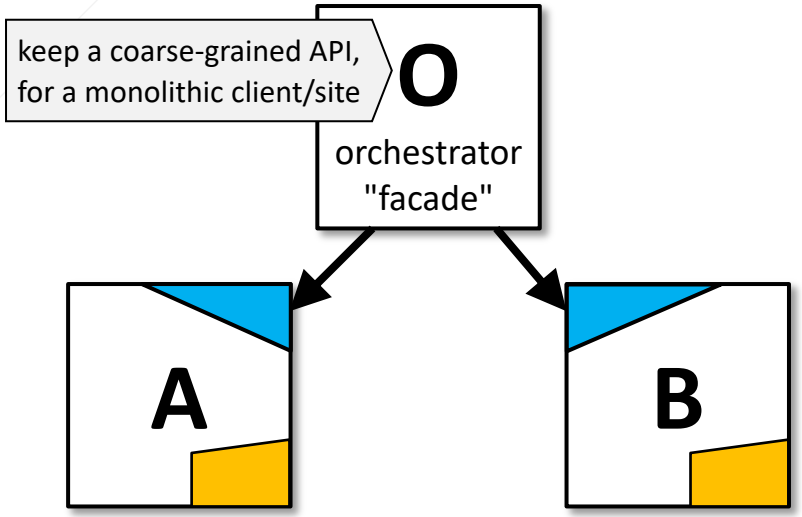
Dependency Inversion



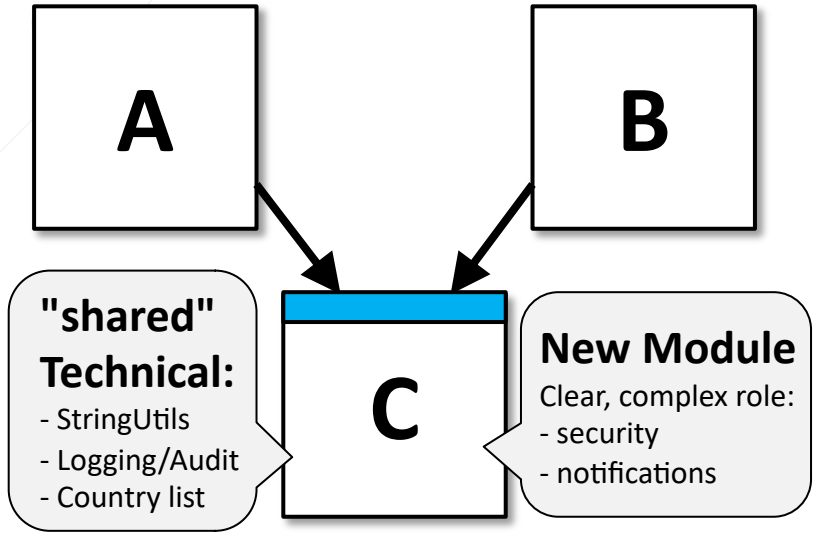
Using **Dependency Inversion**
B can let A implement logic, without depending on A

Strategical Dependencies between Modules

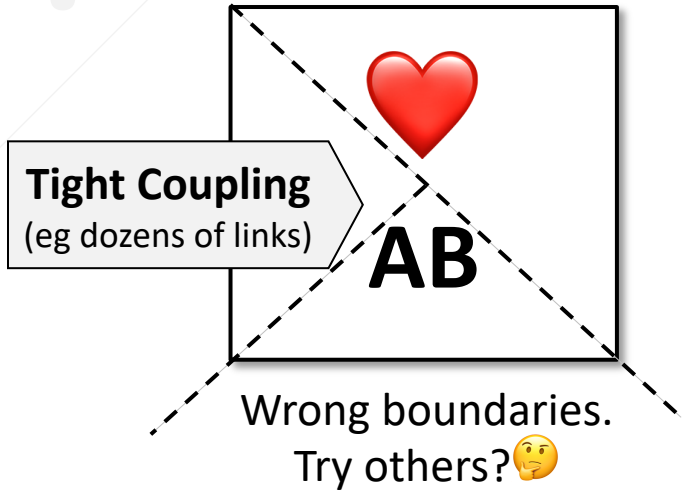
Pull Orchestration Up



Push Shared Down



Merge Modules



Wrong boundaries.
Try others? 🤔

Module Events

⚠️ Events are part of the Public Internal API of the **publisher** module

⚠️ Publisher should expect nothing back (no result, no effect)

☺ Decoupled: publisher is unaware of listeners (potentially multiple)

⚠️ Order in which listeners run should NOT matter (chain events otherwise)

⚠️ Listeners run sequentially in the publisher's thread using current transaction (if any)

☺ Potentially @Async

☺ Can persist your events (see @ApplicationModuleListener) or send yourself a rabbit

☹ Events are harder to navigate than a direct method call ➔ use events cross-module

🤔 Events can carry state to avoid a call back from listener (Event-Carried State Transfer)

Microservices/Modules should have private tables

Allows you to keep in sync:

Problem ⇔ Domain Model ⇔ DB

Data Isolation Between Modules

1. No Isolation (starting point): everyone freely **reads**/**writes** any table

⚠️ Data Inconsistencies: a module could write valid data, but read garbage data.

2. Write Isolation: **one** module **writes** into a table, **any** other can **read**

🧠 Might require splitting a table (in separate schemas)

INVENTORY.PRODUCTS **vs** CATALOG.PRODUCTS, CATALOG.PRODUCT_ATTRIBUTES

⚠️ Frozen Tables: an ALTER TABLE can break the readers

3. Exclusive Access🤩💖: a schema accessed by ONE single module

😏 Trick: expose read-only VIEWS for others to JOIN : INVENTORY.STOCK_VIEW(id, items)🤨

Strong Consistency

---

4. Consistency per Module - tables of two modules:

(a) do NOT share any **Foreign Key**, and

(b) are NOT updated in the same **Transaction**

Eventual Consistency

Defer until extraction as a **microservice is justified**

Agile DB Evolution

DB incremental migration scripts, eg: liquibase/flyway/dbmaintain

- **Never change a committed script**, unless to fix a syntax error
- **Unit tested** on CI after every commit

`V1_initial.sql:`

```
create table product (id int8 not null, ...);
```

`V2_add_supplier.sql`

```
create table supplier (id int8 not null, ...);
alter table product add constraint FK_PRODUCT_SUPPLIER ..
alter table product add column supplier_id int8;
```

`V2_UNDO.sql`

- **DB refactoring** in baby-steps: (eg split an OLD table in 2 NEW)
  - create 2 NEW tables; copy data from OLD to NEW ones
  - have logic write in both OLD and NEW but read from OLD => prod
  - 1 month later, if all looks good in NEW table in prod, make logic read from NEW => prod
  - Trick🧐: Create a VIEW identical to OLD table to allow progressive code migration to 2 NEW
  - 3 production month later, delete OLD table
- **UNDO scripts**😎 can quickly undo DB after a faulty deploy

Migrating to Microservice Databases, talk by Edson Yanaga: https://youtu.be/eRz3xjM08l0

Packages > Build Modules

(Maven/Gradle)

⚠ Only after module boundaries are clear AND the large team is about to split

- **Stronger separation**
  - Impossible to @Disable/add exclusions to the ArchUnit @Tests
  - Impossible to have cycles
  - Encapsulate implementation: (a) extract *x-api* module or (b) keep all apis in *shared* module
  - Better Ctrl-Space suggestions (imagine: 2 classes named 'Product' in different modules)
- **Selective dependencies**
  - A module can decide: *+ reactor, + poi, + jasperreports, - lombok*
  - ⚠ Libraries of all modules share the SAME VERSIONS
- **Partial packaging**
  - Client customizations: ***invoicing-nhs**.jar* is only included in the release for the NHS client
  - Purchased features: ***payment-exports**.jar* (included only if paid)

Difficulties of "Living together" in 1 Process

- **Fragile** vs crash / starvation
  - Modules share the threads/DB connection pools ➔ Monitoring +++
- **Shared Transactions** (±)
  - Exception/bug in listener module rolls back your transaction
- **Classes with same name** in different packages
  - Cumbersome code navigation + DI might fail (Spring fix)
- **Long CI build**
  - Tendency for fine-grained fragile unit-tests ➔ Module-tests!
- **Single OpenAPI**
  - Unique versioning? Type name collisions?

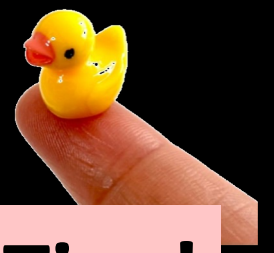
Steps to Extract a Module as a Microservice

- Separate persistence
  - No cross-schema SELECT. 🤔 Think...
  - Drop all **Foreign Keys** to/from other modules' schemas 🤔
  - Stop **sharing transactions** with other modules 🤔
- Separate threads
  - Turn method calls into **REST** localhost:8080 calls 🤔 (propagate metadata!)
  - Handle **events asynchronously:** @Async/@ApplicationModuleListene
  - Send events via **external messaging**: Rabbit, Kafka.. 🤔
- Move module to a separate Git/CI-CD/.pod... 🎉
  - Extract internal API into a separate -client library

DON'T DO THIS UNLESS EXTRACTING THE MODULE AS MICROSERVICE IS PLANNED IN NEAR-FUTURE

How small should microservices/modules be?

Smaller!



Tiny!

PTSD after BBoM?

* the same applies for "Microservice"

Smaller pieces,
more coupling

The Fallacy of Nano-Services

The **microservices honeymoon is over**.

Uber is refactoring **thousands of microservices into a more manageable solution**;

Kelsey Hightower is predicting **monoliths are the future**;

Sam Newman is declaring that **microservices should never be the default choice**, but rather a **last resort**.

https://vladikk.com/2020/04/09/untangling-microservices/

Take-Away
Modular Monolith is

- The **safest way to transition** to Microservices
  - Cheap to experiment with boundaries
  - Allows to progressively enforce them

- A **valid alternative to microservices**
  - For highly complex domains with decent NFRs

Target a Modulith instead of Microservices?

- **YES✅**: A **greenfield** implementation
  - eg a new peer-to-peer lending platform in the early stages of growing into a full suite of financial services.
- **YES✅**: A system with a **low to moderate scale**
  - eg a movie ticket booking platform with several thousand users and handles a few million transactions per week.
- **YES✅**: **Non-complex** business software platforms
  - eg a notes/documents syncing and management platform for consumer apps.
- **YES✅**: **Oversized BBoM legacy** applications before breaking into independent services
  - eg an existing large-scale banking platform monolith

- **NO❌**: When parts of the platform are managed by multiple independent teams
  - Especially if each team governs its choice of underlying **technology** and runtimes;
- **NO❌**: If a **service chassis** can significantly reduce entry cost of microservices
  - New, large-scale platforms built in mature environments
- **NO❌**: **High scale and volume** of business + good understanding of architecture

https://www.ufried.com/blog/microservices_fallacy_9_moduliths/
http://blogs.newardassociates.com/blog/2023/you-want-modules-not-microservices.html
https://www.techtarget.com/searchapparchitecture/tip/Understanding-the-modular-monolith-and-its-ideal-use-cases

Testing a Modulith

End-to-end Tests (all modules)

are easier in a monolith

Module-scoped Tests 💖

Integration test a Module in isolation

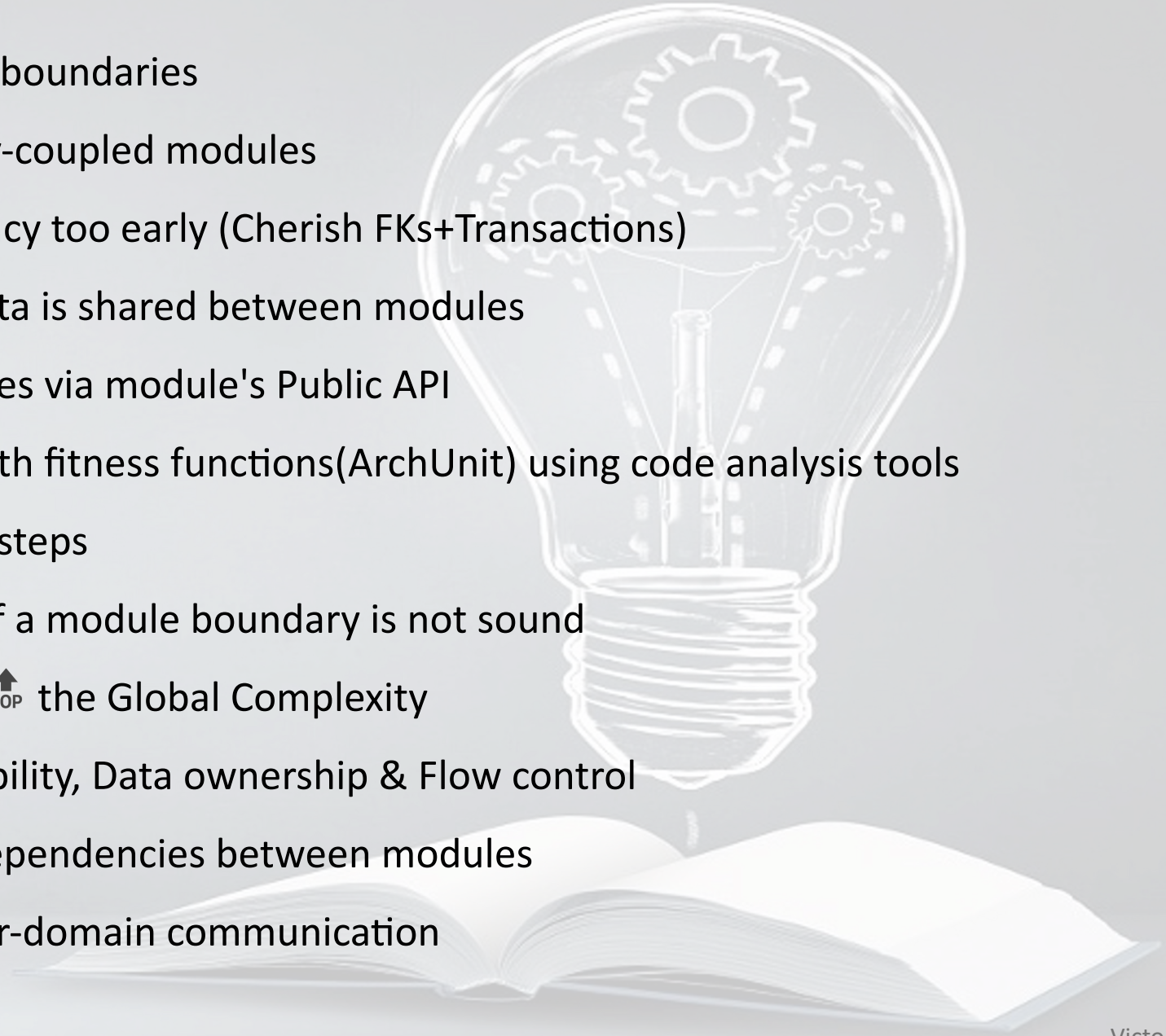
- mock public APIs of other modules
- mock the event bus

➔ See Spring Modulith for inspiration

Separate DB instances for module/micros?

- **(Yes) stricter control**: vs separate schema🤔
- **(Yes) fault tolerance**: DB crash should not impact other services
- **(Yes) scalability**: separate DB connections / service
- **(Yes) nosql is more suitable, eg**:
  - **Redis, Hazelcast** (in-memory, fast reads)
  - **Mongo** (document db)
  - **Cassandra** (heavy writes)
  - **Neo4j** (Graph DB), ☁️-specific
- **(No)** hurts performance / too complex (legacy)
  - **Triggers** or **PL/SQL** ➔ pull logic out of DB
  - **Views** ➔ move to eg. ElasticSearch
- **(No)** strict consistency is **IMPERATIVE** for business (with arguments)

Modulith – Lessons



- Carefully refine module boundaries
- Consider merging highly-coupled modules
- Don't sacrifice consistency too early (Cherish FKs+Transactions)
- Carefully design how data is shared between modules
- Don't leak internal classes via module's Public API
- Monitor architecture with fitness functions(ArchUnit) using code analysis tools
- Segregate code in baby-steps
- Be ready to stop/undo if a module boundary is not sound
- Too small modules will TOP the Global Complexity
- Focus on Business capability, Data ownership & Flow control
- Carefully refine cyclic dependencies between modules
- Consider Events for inter-domain communication

Towards Modules

in baby-steps

Decouple Domain Entities

catalog

```
class Product {
  id
  name
  ...
}
```

fulfillment

```
class OrderLine {
  ...
  Product product;
  Long productId;[1]
  String productName;[2]
}
```

Remove any object link to the Domain Model of another module:

1) Keep only the ID ✅

👌 Preserve the FK (in a Modulith)

+ then later get from catalog on-demand

2) Snapshot *immutable* data

+ according to domain rules

3) Keep mutable data in sync 😨

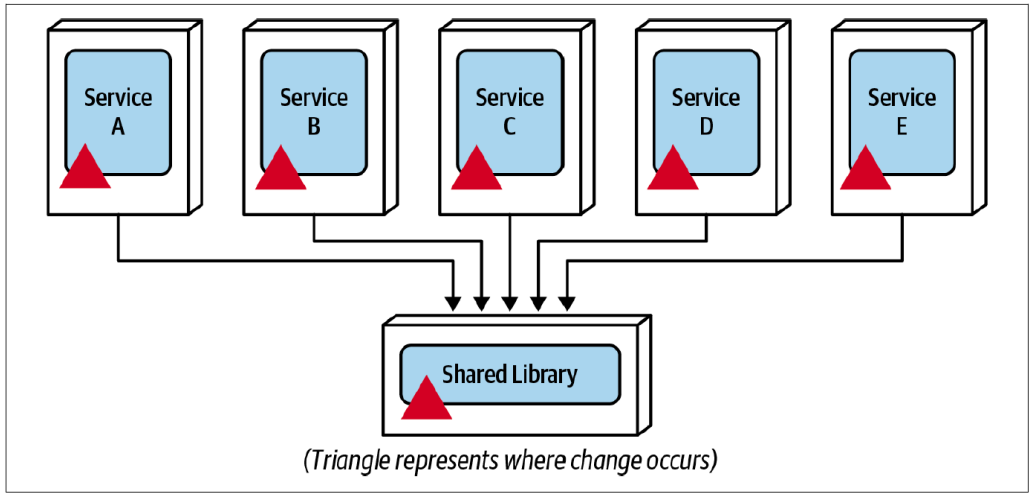
+ replicate changes (eg via events)

⚠️ Eventual Consistency (iff distributed)

Forces against extracting a Module out

- **Tight Coupling** by chained sync calls, especially bi-directional
  - Chained REST call would hurt ~~Availability~~ and ~~Performance~~

- **Shared Domain Model**
  - BAD: Services depend on a 'commons' library holding 40% of business logic&data
  - You must change, test, deploy together => ~~Independent Deployability~~



- **Data Consistency** is critical between 2 Modules
  - Many ACID flows and the complexity of sagas outweighs the benefits

Shared Library / Microservice Chassis

- **Problem:** implement common tasks in services without copy-paste
- **Solution:** create and manage a **shared library** helping with technical, non-domain specific tasks.
- **Debates** 🤔
  - Specialized libs (-commons-security-v5') vs a single large one ('commons-v32')
  - Should clients use version=LATEST? Dependabot auto-upgrade?
  - What's the deprecation policy?
  - Who is paid to maintain the lib? Who's its GateKeeper?
- Why turn a lib into a service?
  - a) better support high rate of change
  - b) reuse across languages
  - c) clearer boundary and ownership

Finding Boundaries

between Modules / Microservices

= one of the hardest architecture challenges

https://opencredo.com/blogs/identify-service-boundary-heuristics/
https://github.com/TeamTopologies/Independent-Service-Heuristics
https://www.dddheuristics.com/
https://www.michaelnygard.com/blog/2018/01/services-by-lifecycle/

Boundary Types

- **Business**: subparts of the problem
  - eg: shipping, invoicing , ...
  - We speak the same **Ubiquitous Language** within the **Bounded Context** of our sub-problem
- **Humans** (teams)
  - Align **Dev** Team with **Business** Team (= **Conway's Law**)
- **Technical** (codebase)
  - Dev Team should have ownership on a manageable code part
  - ie. break the legacy Big Ball of Mud in Modules / Microsevices?
- You will get them wrong first ➔ **PREPARE TO ITERATE** 🔁

Group by Data = central data concepts

(*Entity Service*)

order

Order & friends

invoice

Invoice & friends

product

Product & friends

Group by Bu$in€ss Capability = value to the users

(*Feature Service*)

fulfillment

place-
return-
workflow
~~history~~

invoicing

issue invoice
VAT discount
storno
taxes

catalog

search
filter
display
compare

Identify more business capabilities of an e-shop

Scales Better

https://www.michaelnygard.com/blog/2018/01/services-by-lifecycle/

Group by Business Capability

Example: product price

🤔

Think in terms of attributes, not classes

```
class Customer {
  firstName
  lastName
  status
  ...
}
```

```
class Product {
  name
  description
  price
  ...
}
```

«Entity-Service»

Customer

- customer first name
- customer last name

«Feature-Service»

Pricing

- customer status
- product price
- promotions🤯🤯🤯

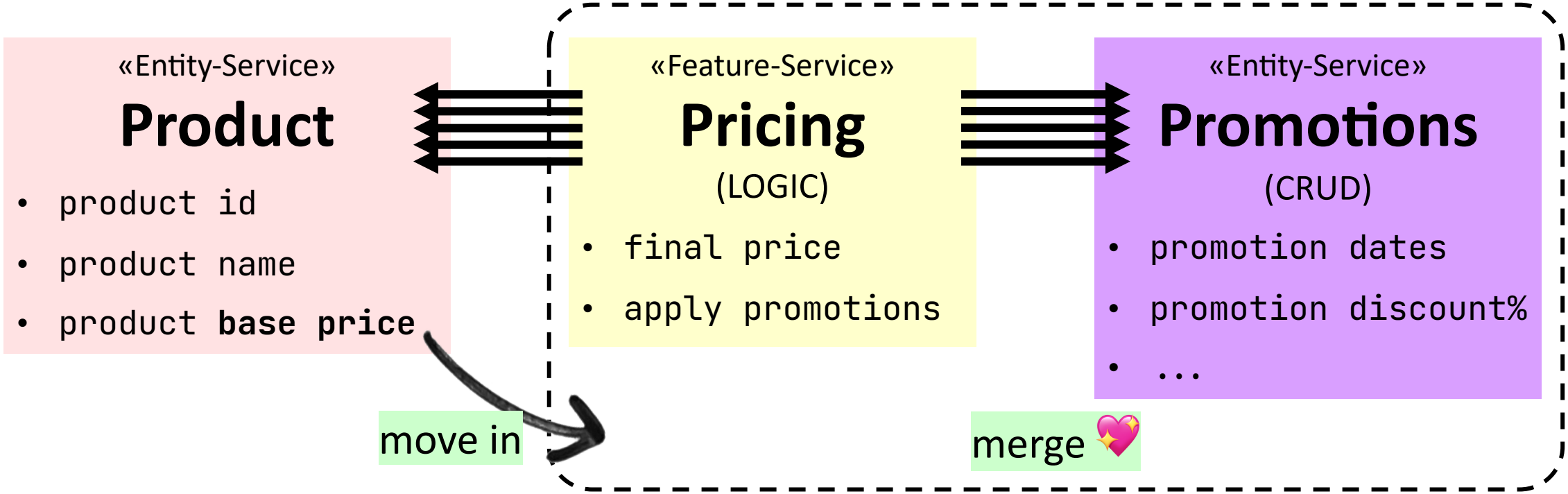
«Entity-Service»

Catalog

- product name
- product description

Group by Feature

Example: Promotions



Avoid Modules that are:

- Just **Behavior**

  Example: calculation, validation

  Except if they solve a very specialized domain problem.

- Just CRUD **Data**

  Example: Country, Tenant, Clinic

  ➔ Modulith: keep this data in a 'shared' module

  ➔ Micros: inject static data as config

A Module/Microservice is the

Technical Authority for a Business Capability

.product ⇨ .catalog

 * the same applies for "Microservice"