Image by Ylanite Koppens from Pixabay

Legacy Architecture Migration Patterns with DDD & AI

Nick Tune

nick-tune.me

/in/nick-tune

@nick-tune.me

Using Common patterns is helpful, but understanding the building blocks is equally important.

Most legacy migration initiatives involve hundreds of micro-decisions. Patterns don't cover everyting

Legacy Migration Building Blocks (might not be all of them)

Migrate write operation

Run new and old in parallel

Migrate read operation

Route requests (new vs old etc.)

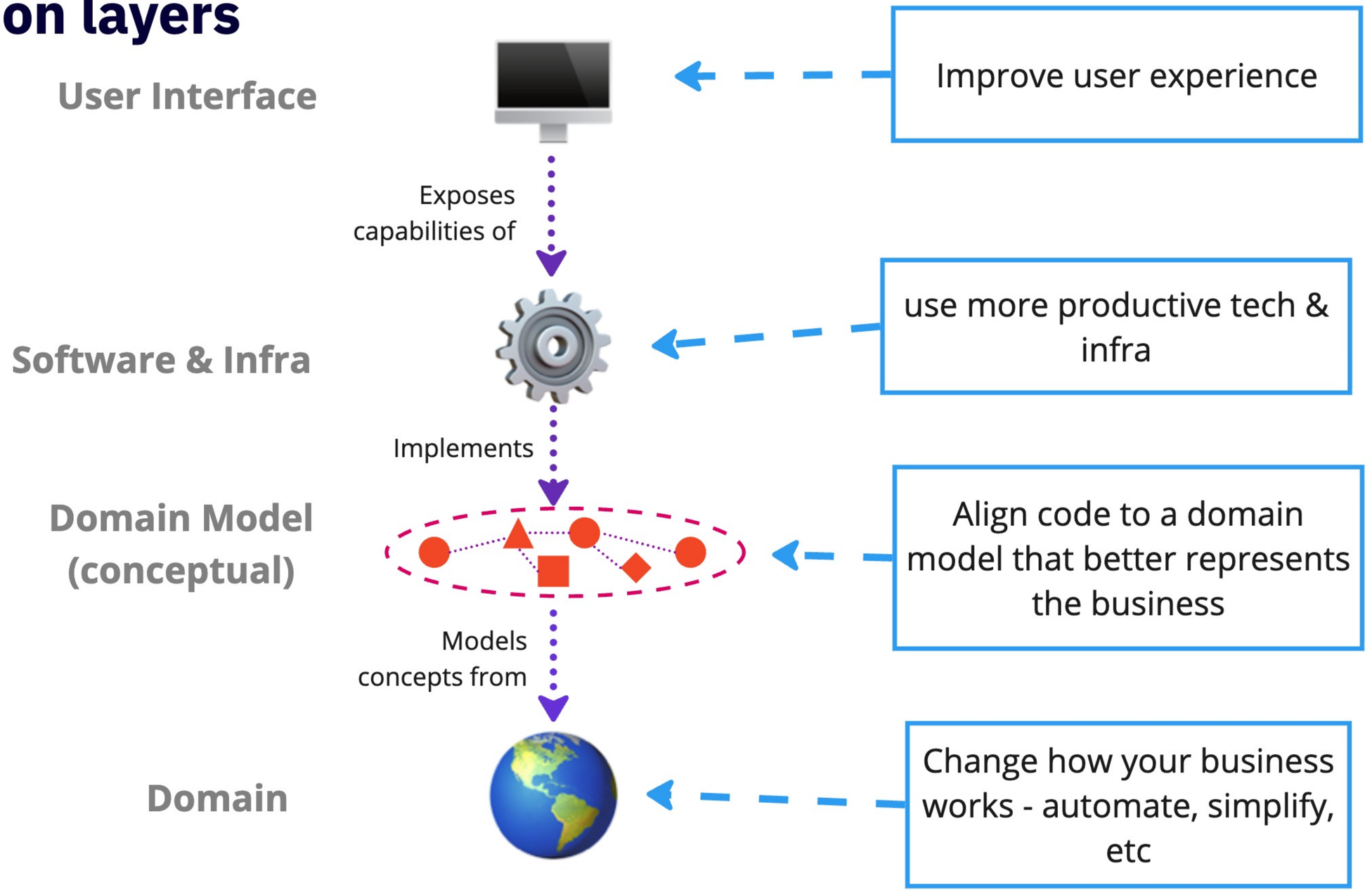
Migrate data

Remove & change functionality

Map between (domain) models

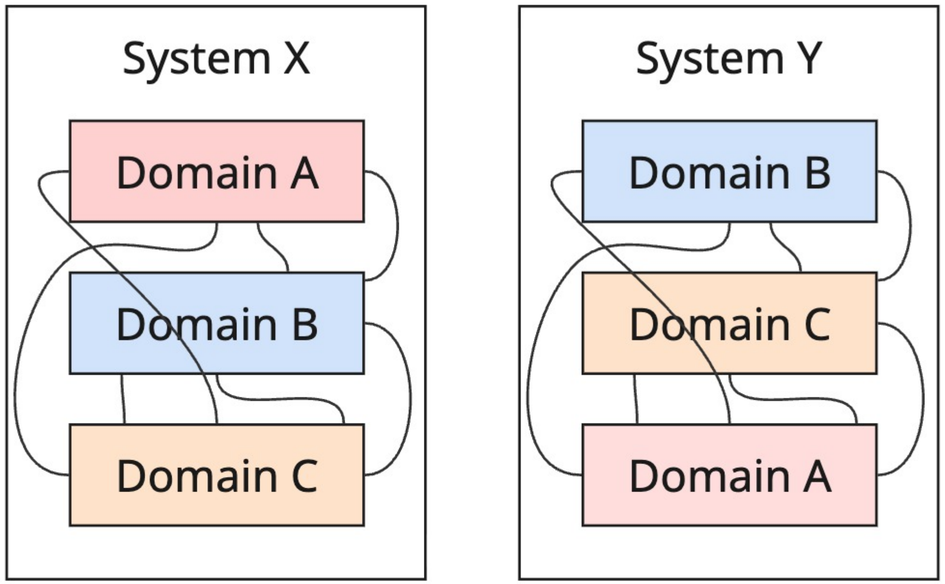
Analyze existing system

Modernization layers

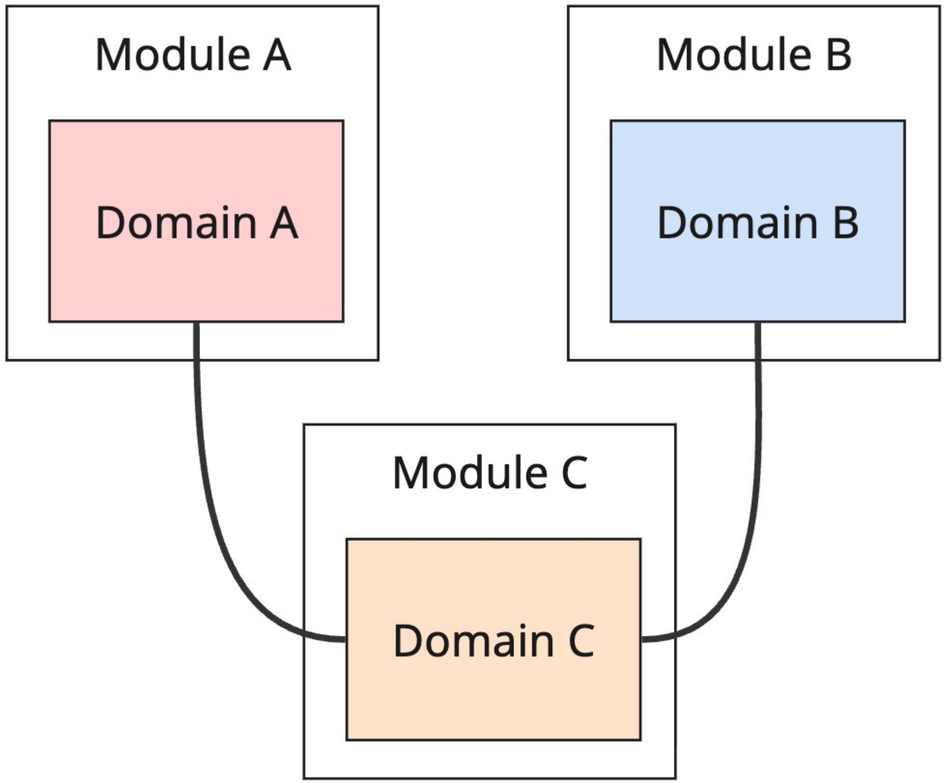


Domain boundaries need to evolve to reduce coupling + often team boundaries

Legacy Boundaries
(low modularity)



Target Boundaries
(good modularity)



Can AI help with legacy modernization?

Where AI genuinely helped

1. Build our optimal workflow & tooling
2. Project assistant who knows everything about the project and can assist with any topic
3. Map out current state 10 - 20x faster in more detail
4. Execute 2x - 3x faster
5. Reduce cognitive load of context switching -> allowed us to mage progress on multiple streams in parallel

My Personal Recomendation

Anthropic marketing department want you to believe Claude is a "competent senior engineer" who you can delegate to. I disagree.

I think that AI can provide huge benefits to your modernisation project, but you should rely on the model as little as possible.

1. Turn your expertise into commands and workflows

2. Continuously bake-in learnings so that your workflow produces higher quality & fewer mistakes each time

3. Stay close to the code and understand what AI is doing

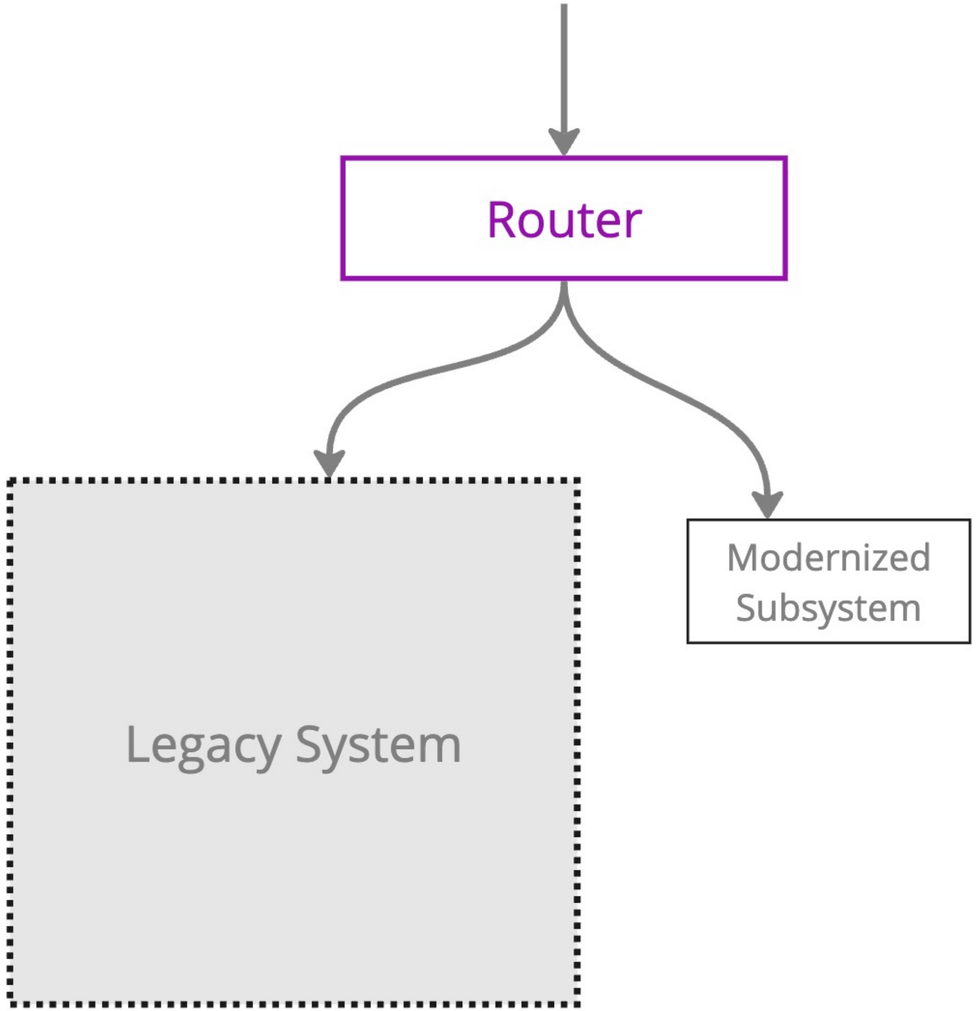
Part 1:
Legacy Migration Patterns

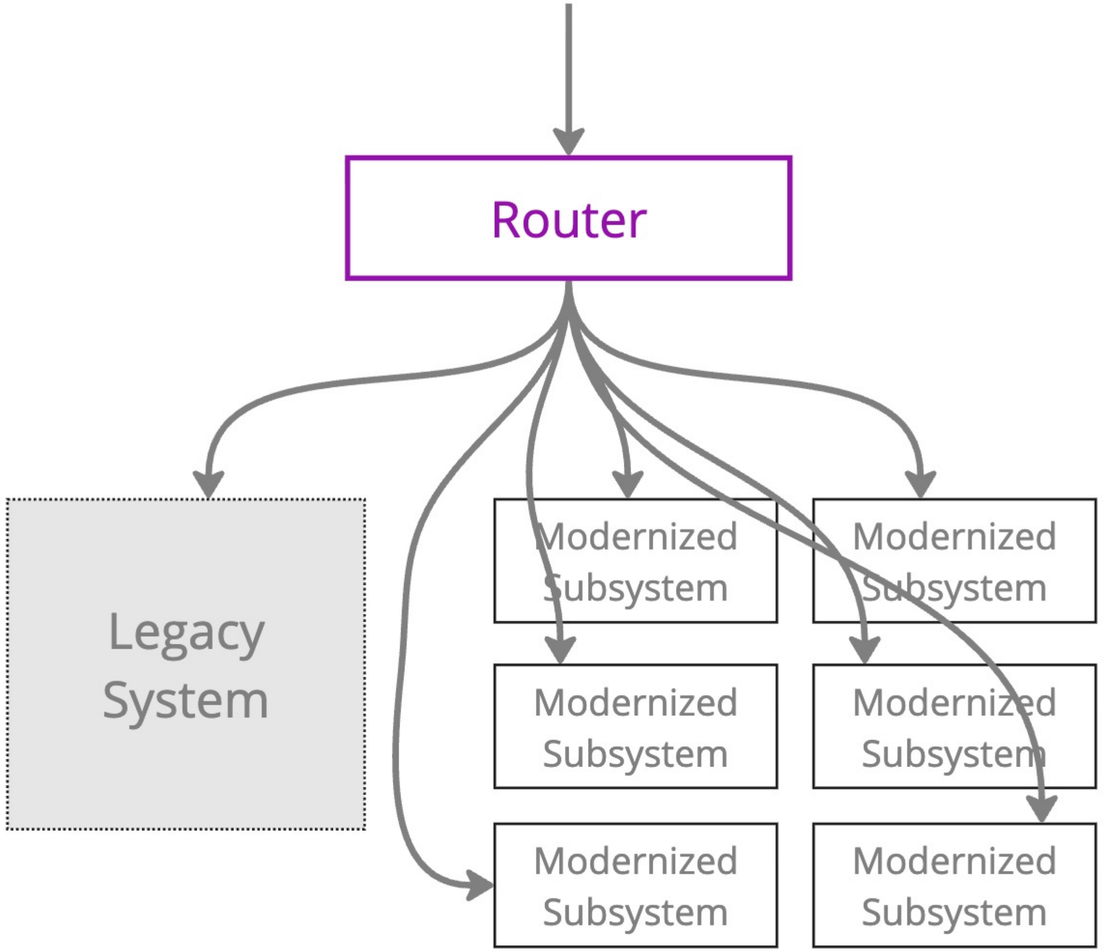
Part 2:
Legacy Migration Patterns at PayFit

Part 3:
Legacy Modernization with AI at PayFit

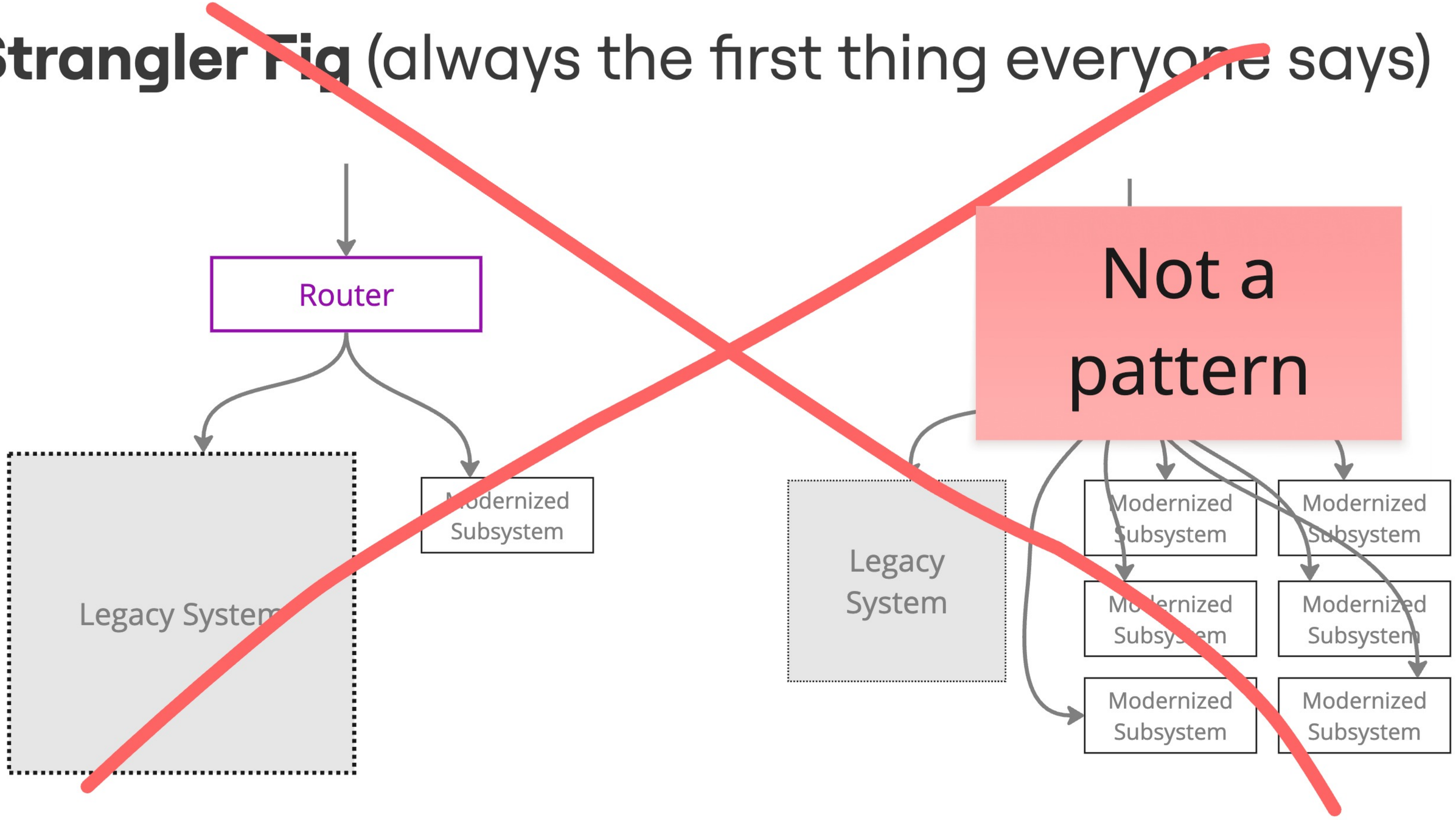
1.
Legacy Migration Patterns

Strangler Fig (always the first thing everyone says)

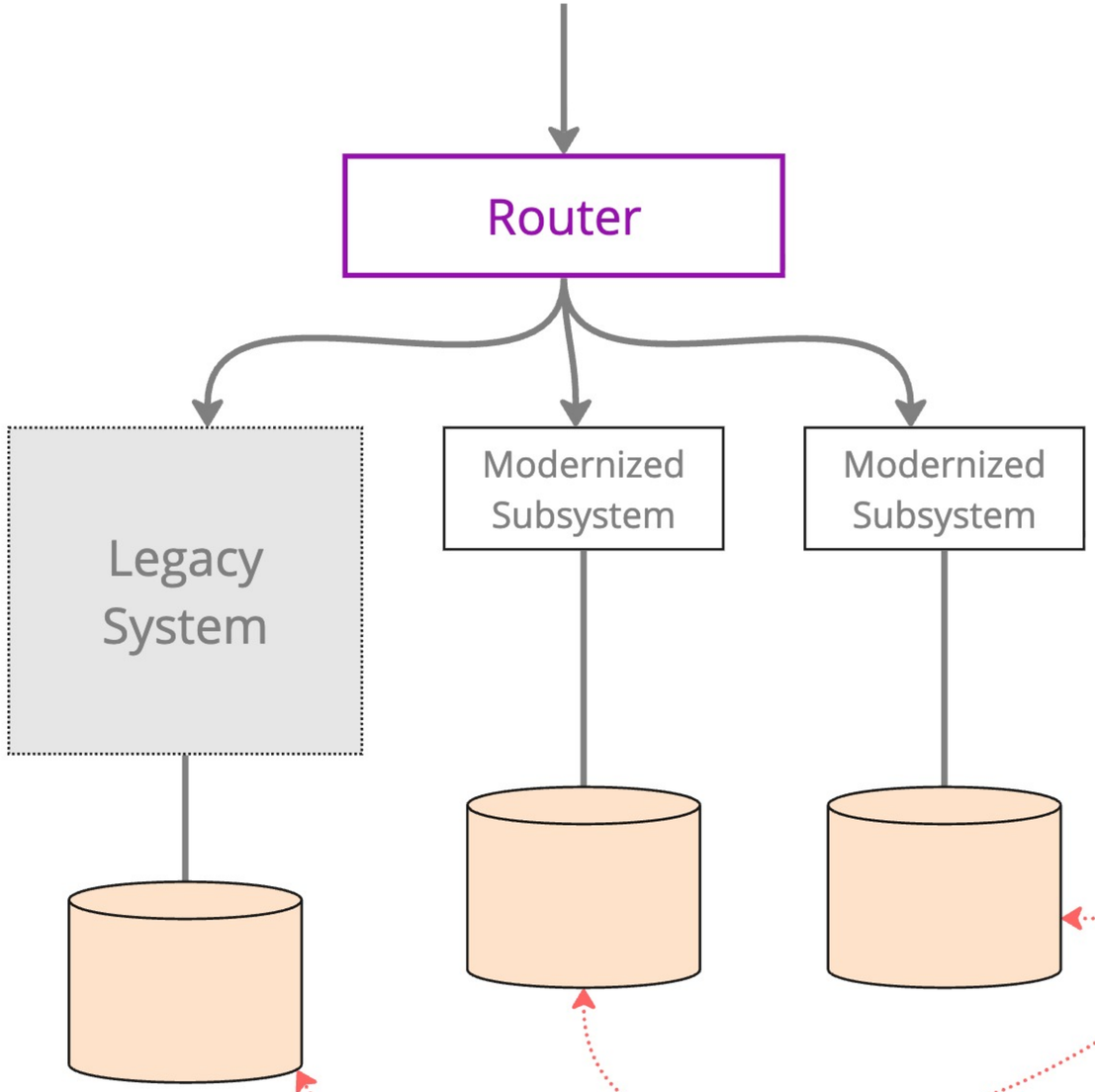




Strangler Fig (always the first thing everyone says)



Data integration



domain events

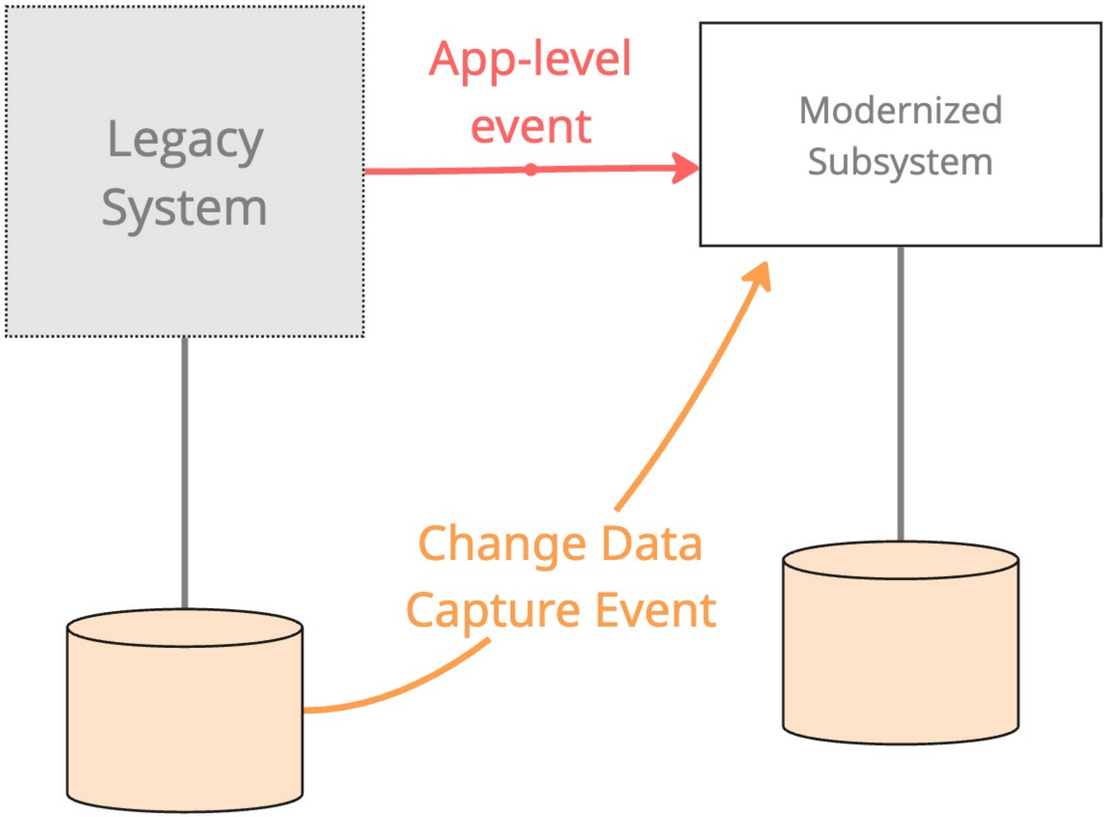
CDC

batch / import jobs

shared DB

Could be different schemas inside same physical DB, or completely different DBs (e.g. a mix of SQL and NoSQL)

CDC vs Application-level events



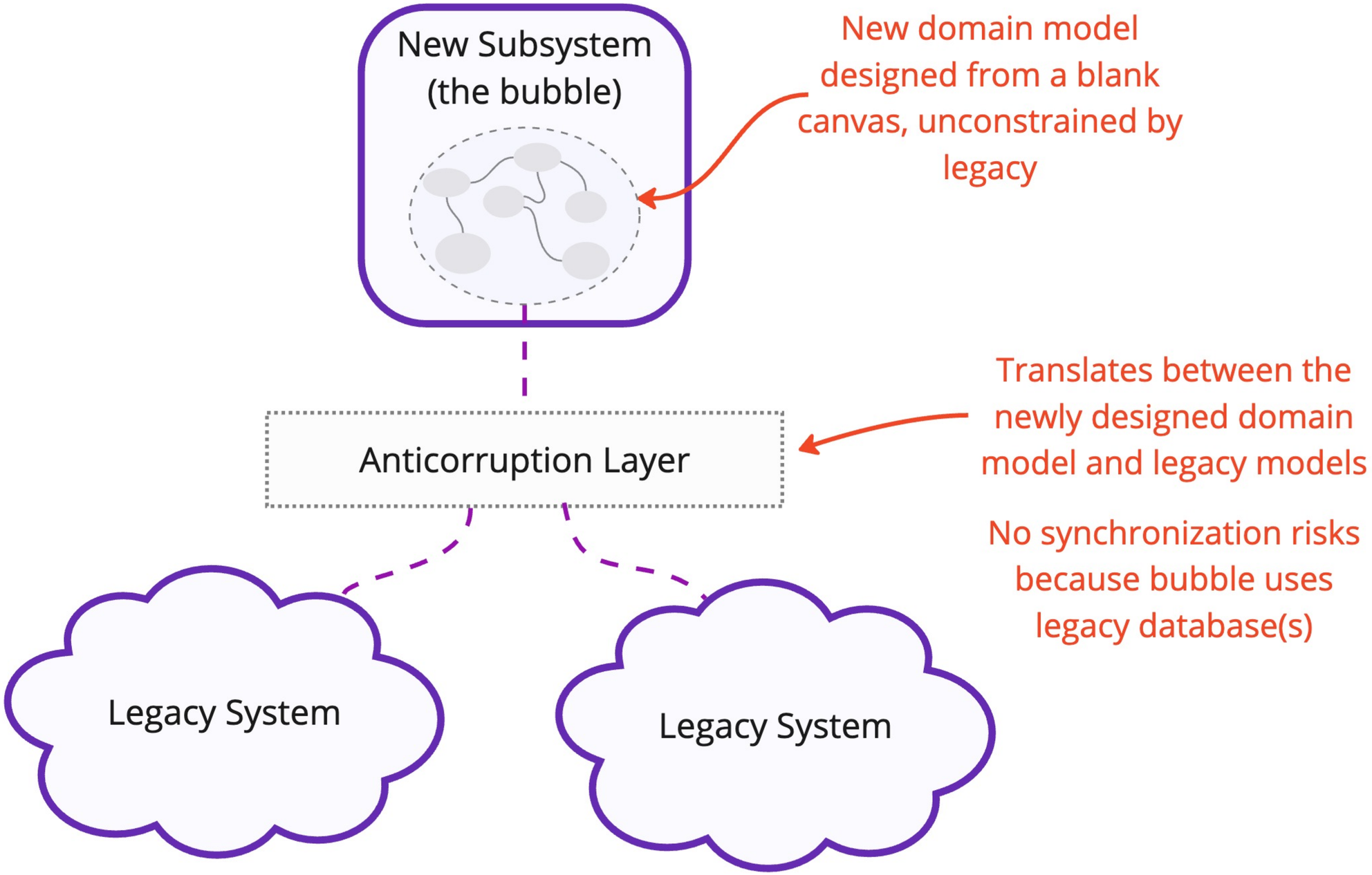
CDC

- ✅ Easier to setup in legacy
- ✅ Catches all data modification scenarios

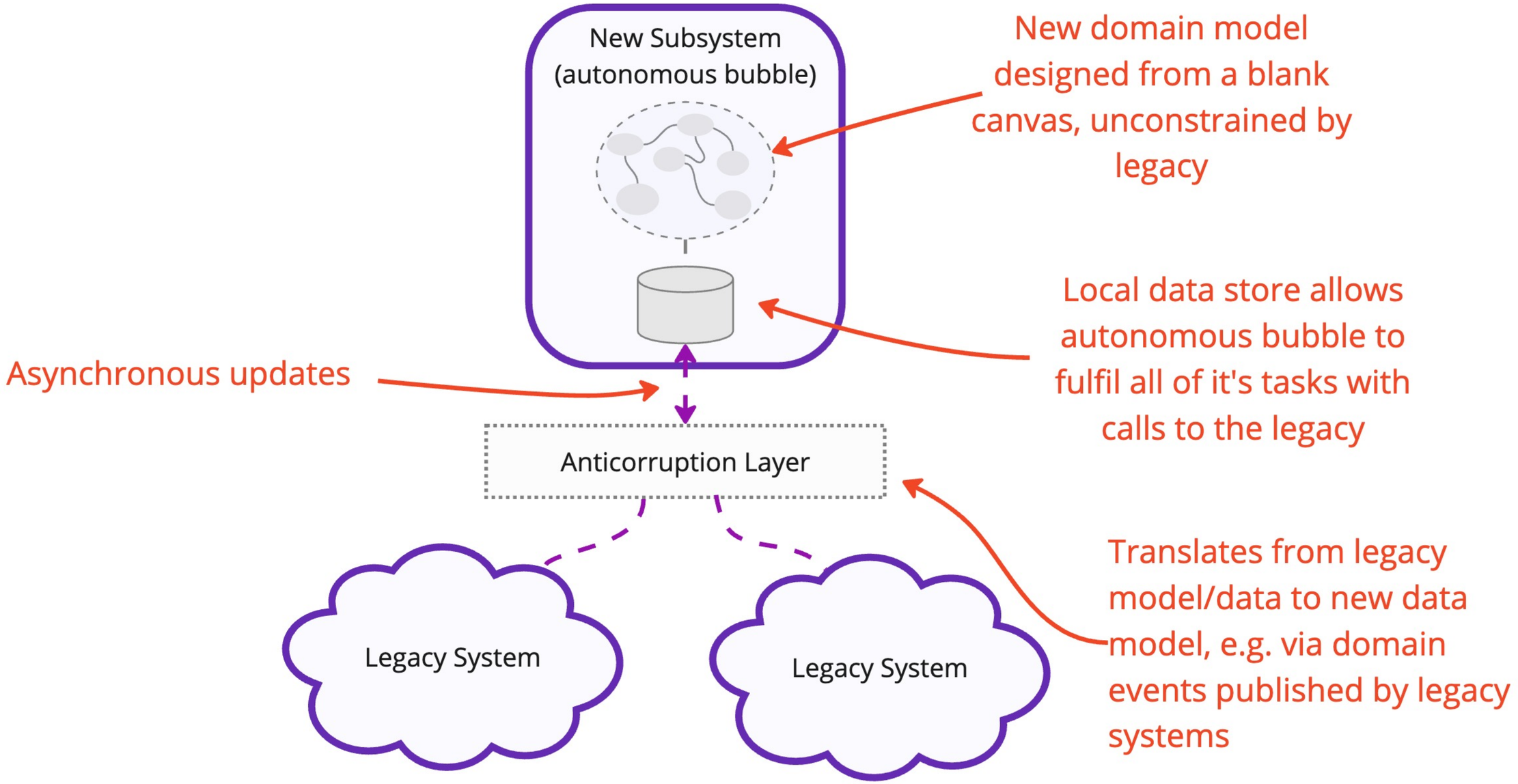
App-level

- ✅ Provides more context about the event
- ✅ Avoid leaking legacy model & coupling to legacy DB

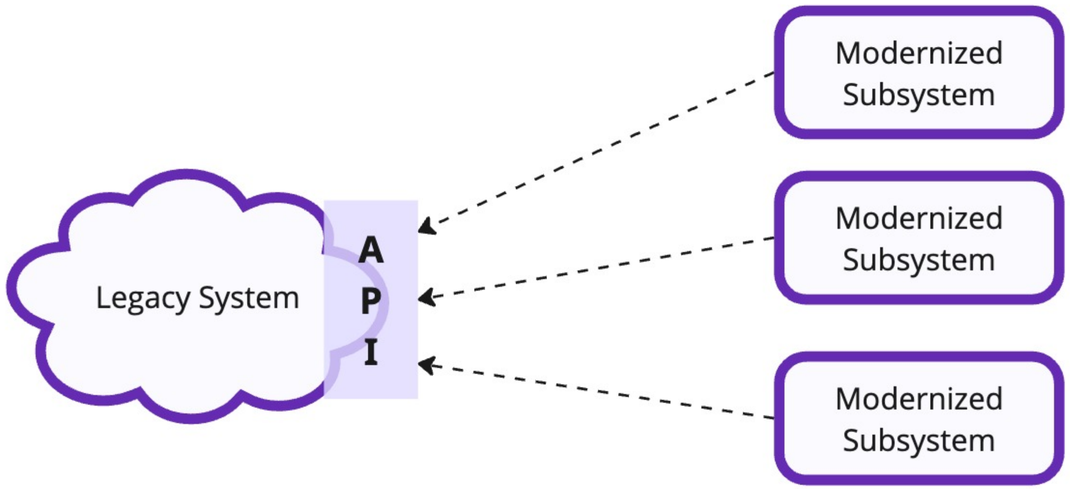
Bubble


New Subsystem
(the bubble)
New domain model designed from a blank canvas, unconstrained by legacy
Anticorruption Layer
Translates between the newly designed domain model and legacy models
No synchronization risks because bubble uses legacy database(s)
Legacy System
Legacy System

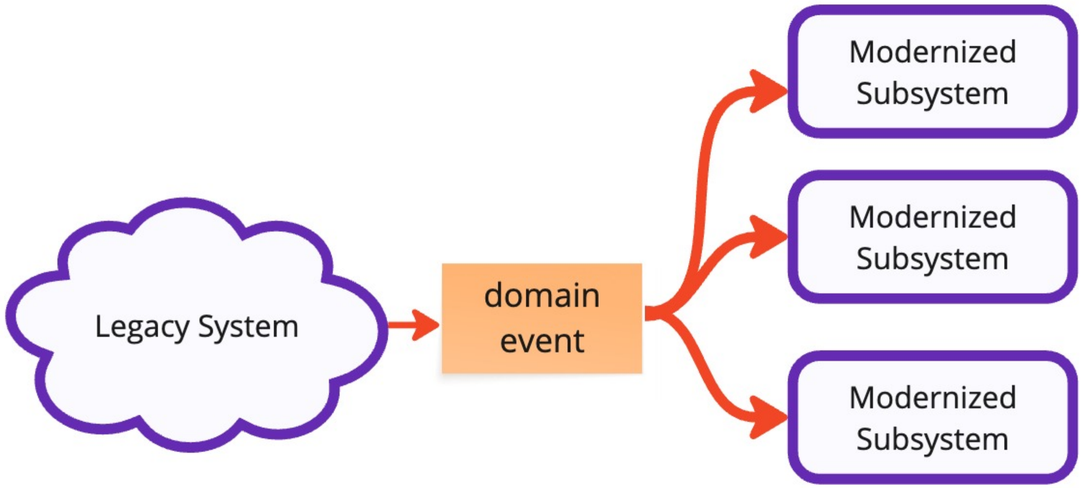
Autonomous Bubble



Expose Legacy Asset

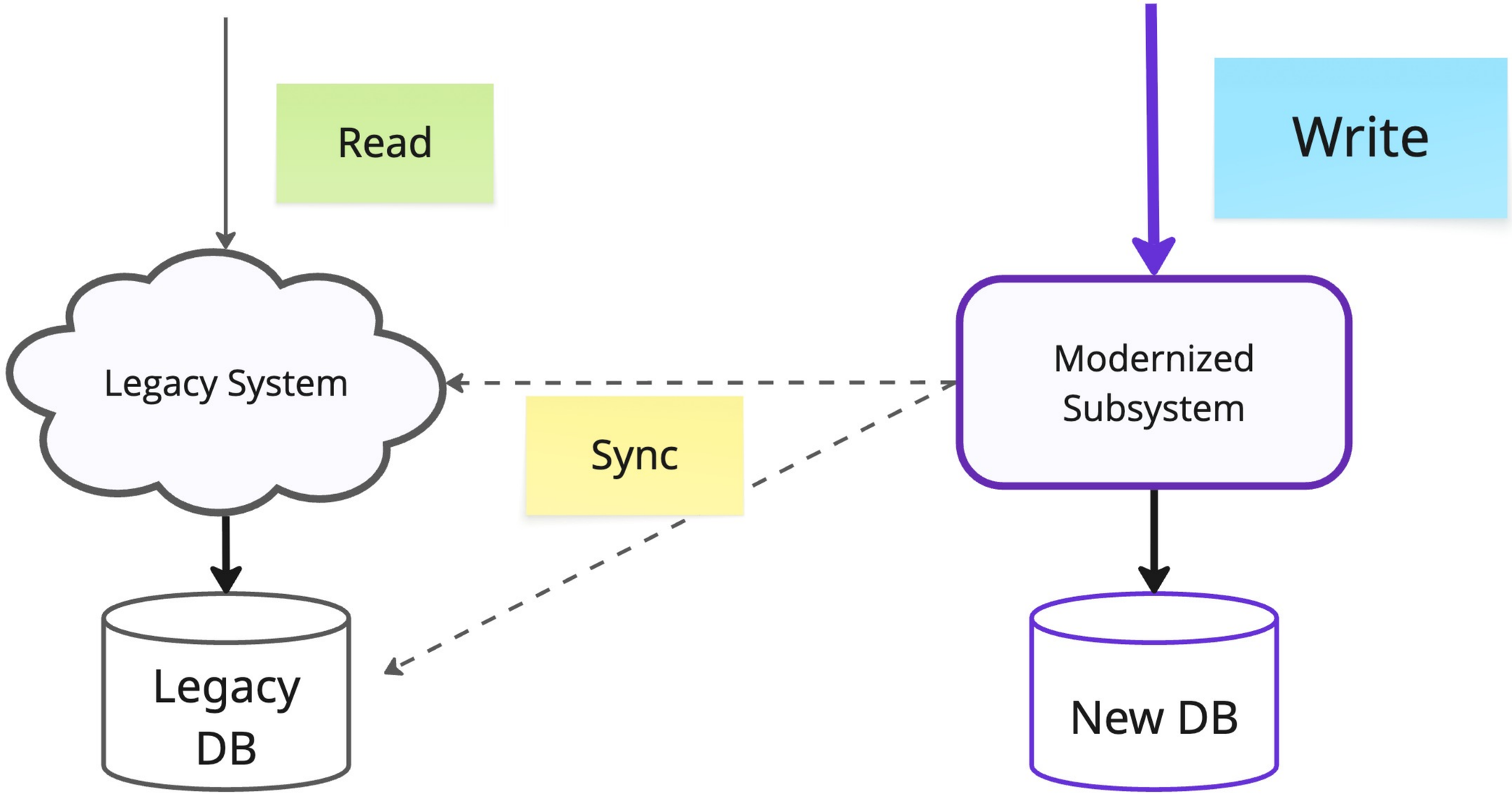


Expose legacy via API

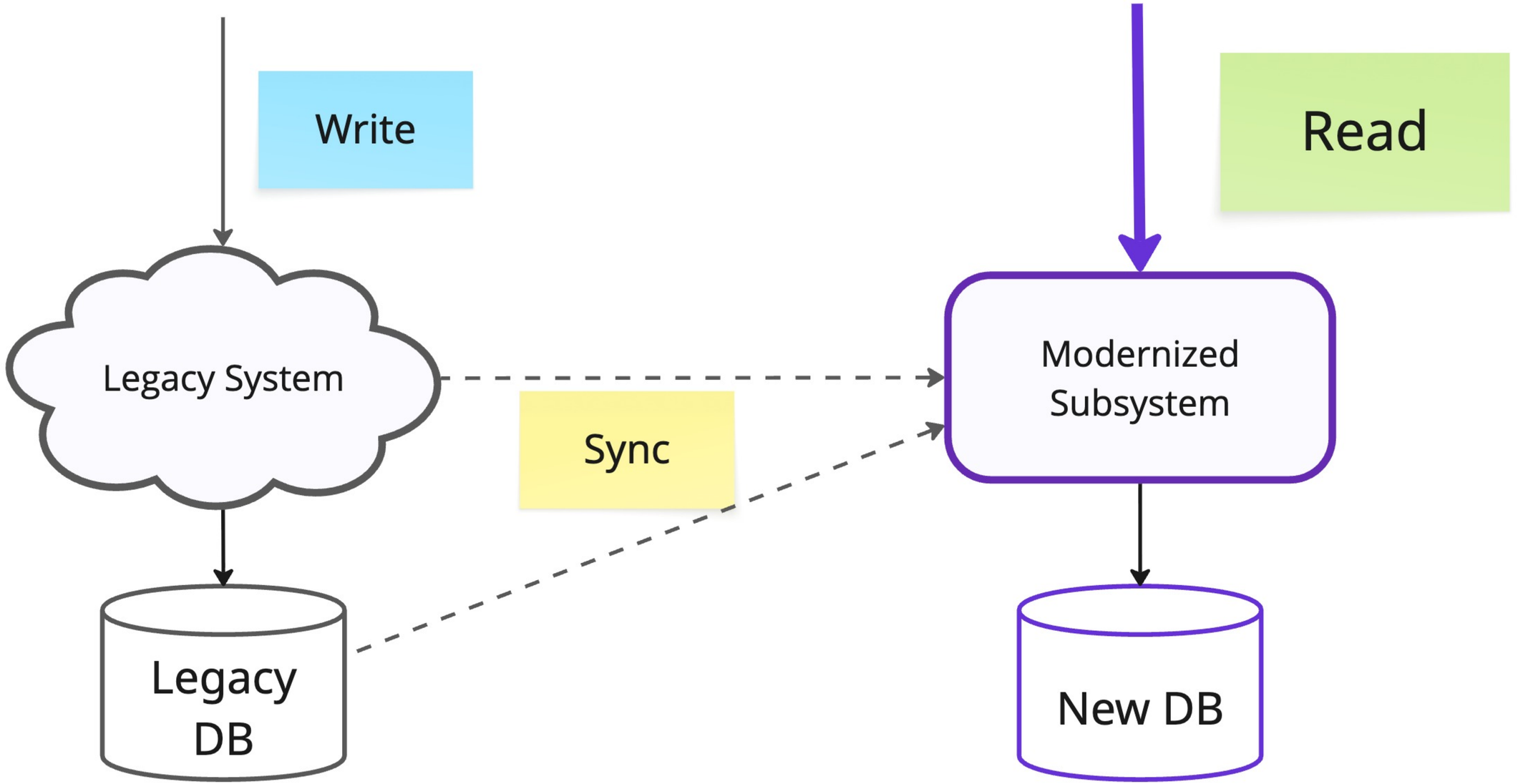


Expose legacy via events

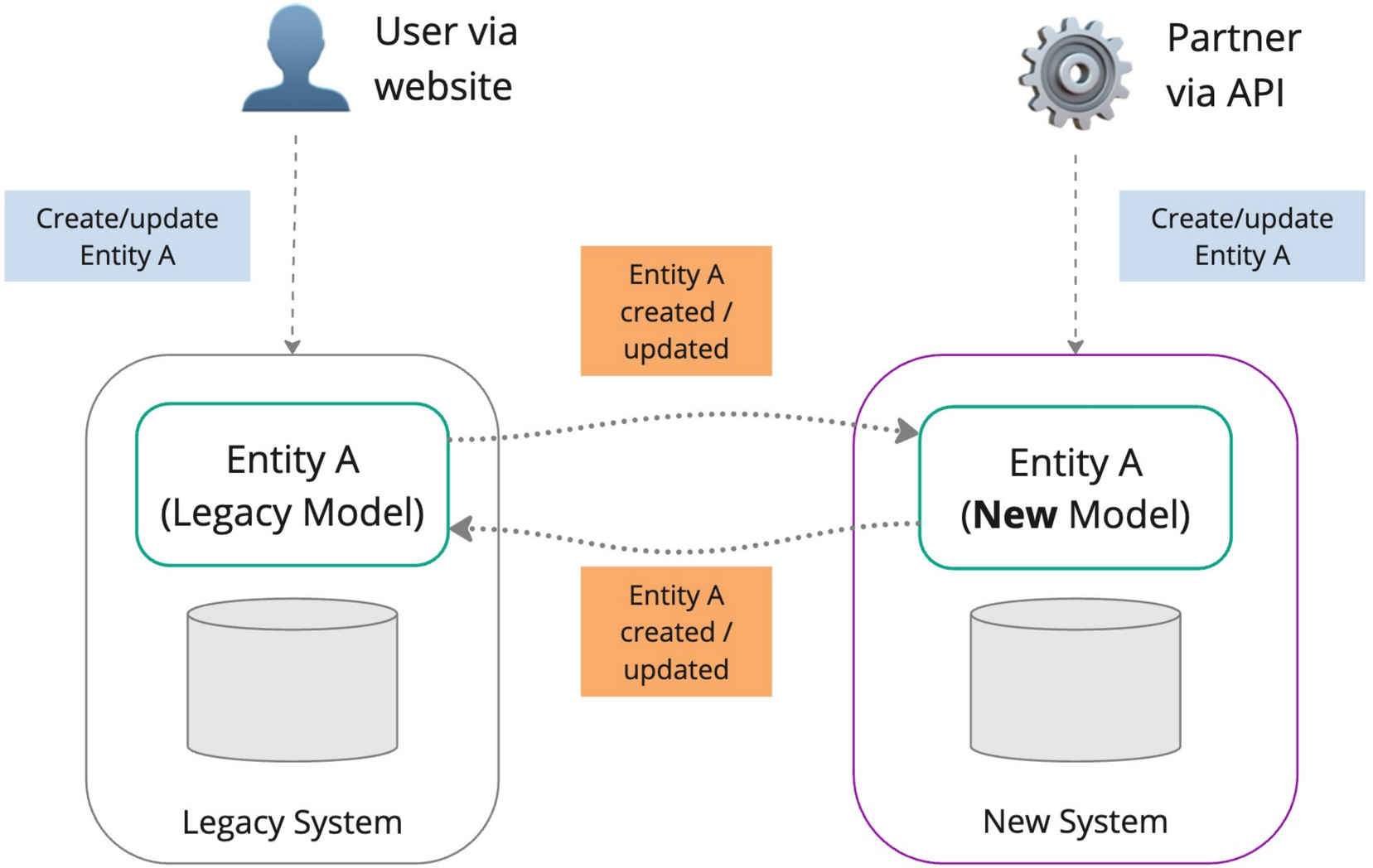
Migrate Writes First (within a specific scope)



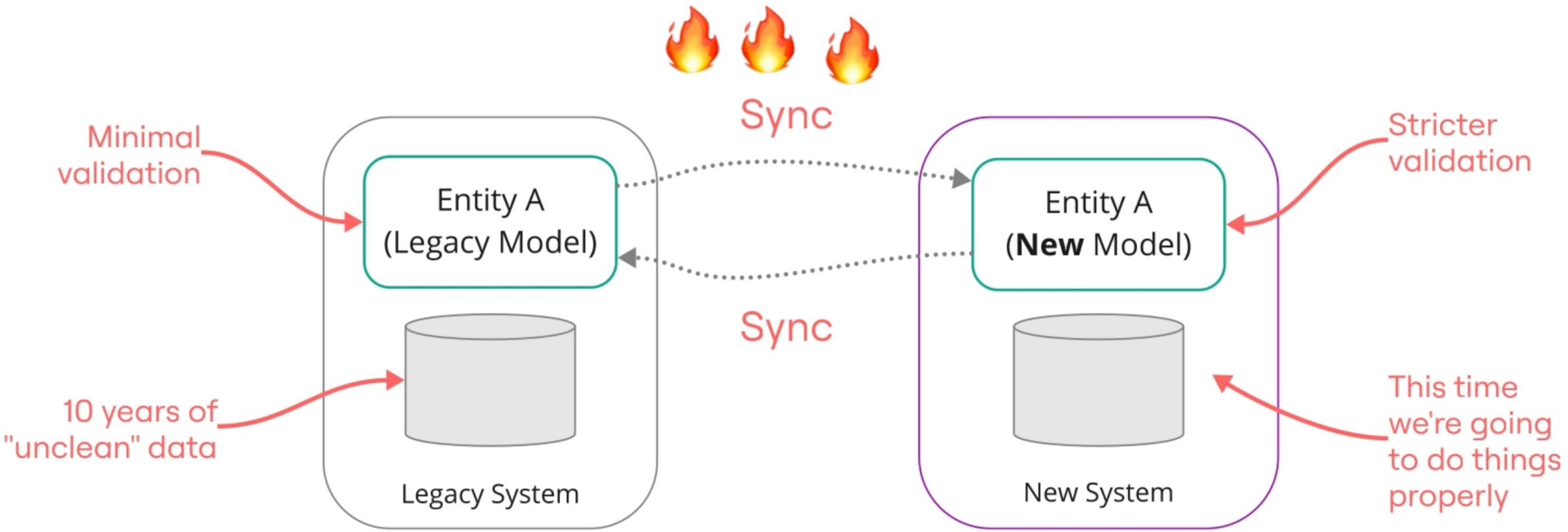
Migrate Reads First (within a specific scope)



Bi-directional model sync (risky)

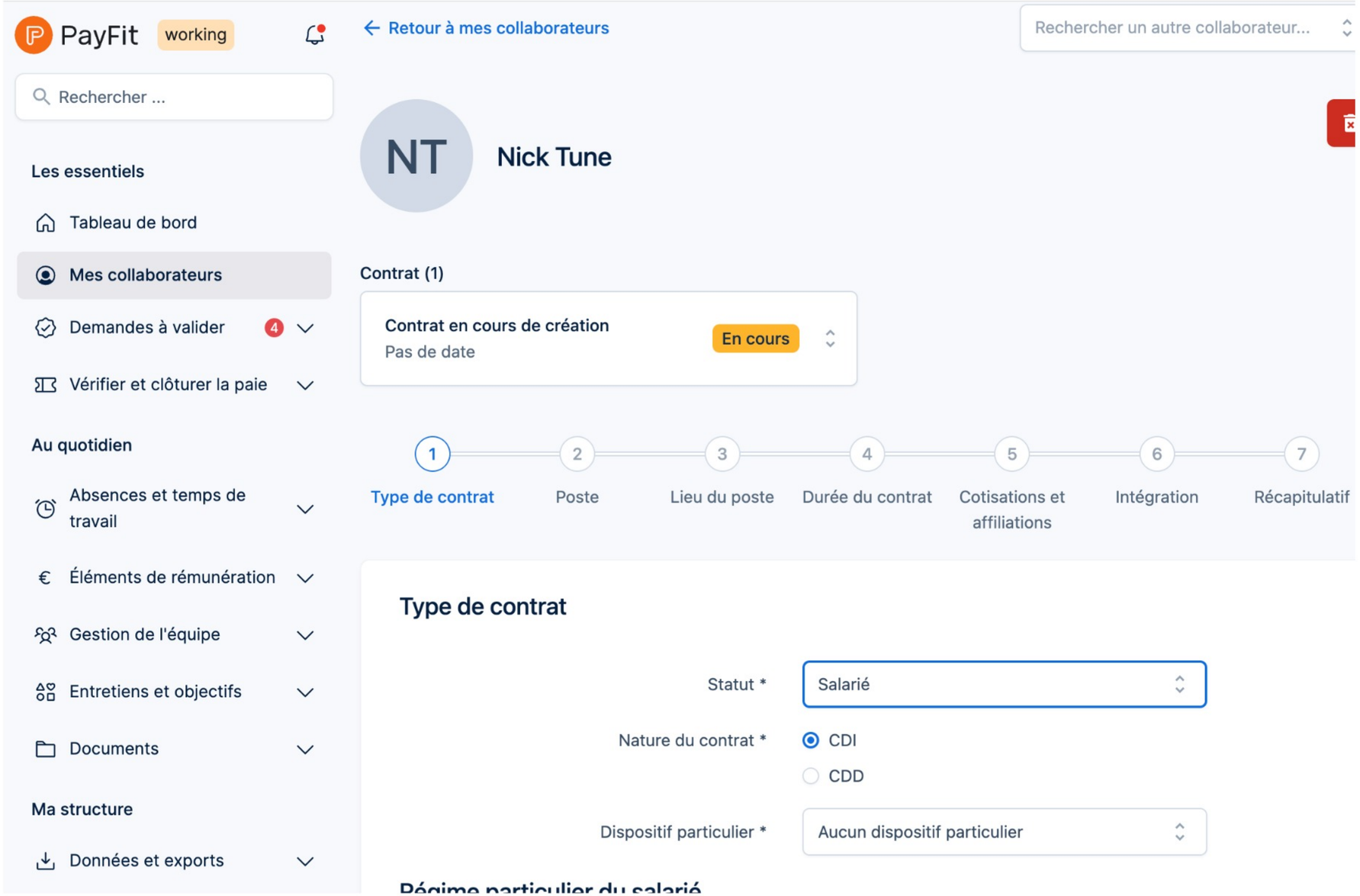


Asymmetrical Validation



2.
Legacy Migration Patterns at PayFit

PayFit: HR & Payroll

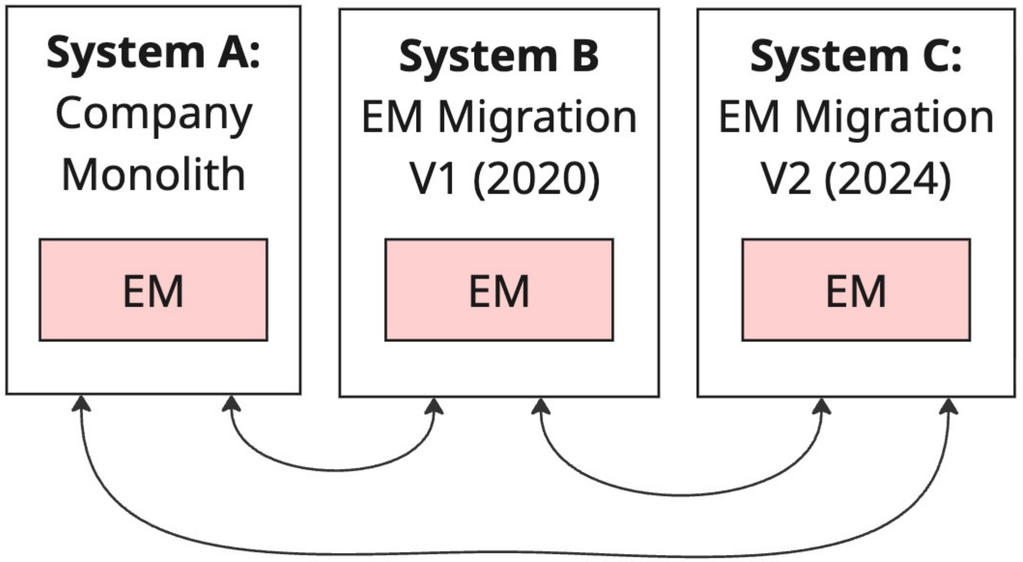


Employee Management Domain: LTAV

🇪🇸
🇬🇧
🇫🇷

Current State

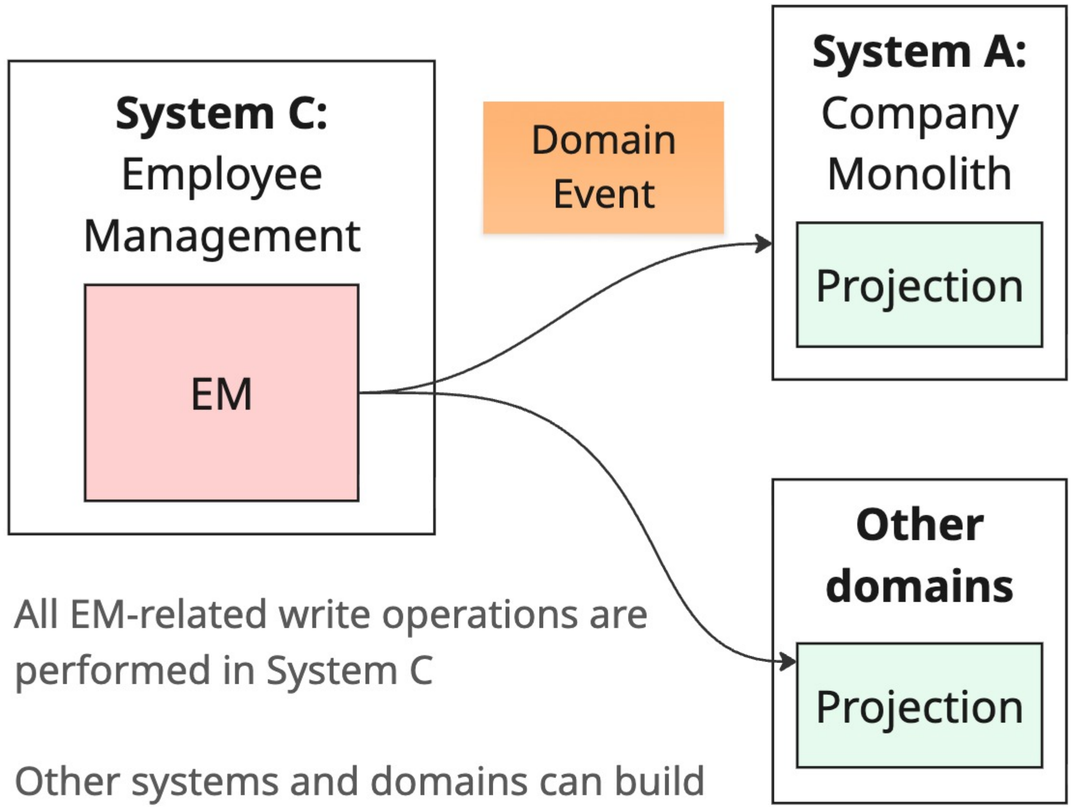
3 systems, 3 models



- Read and write operations happen in all 3 systems.
- All 3 need to be synchronised in various ways
- Complexity spreads to other domains as well

Target State

1 model, various projections



- All EM-related write operations are performed in System C
- Other systems and domains can build local projections based on events
- System B & C merged

Competing* migration priorities

1. Other teams are waiting for our EDA events so they can fully migrate
   (events first ?)

2. New search APIs needed
   (read model first ?)

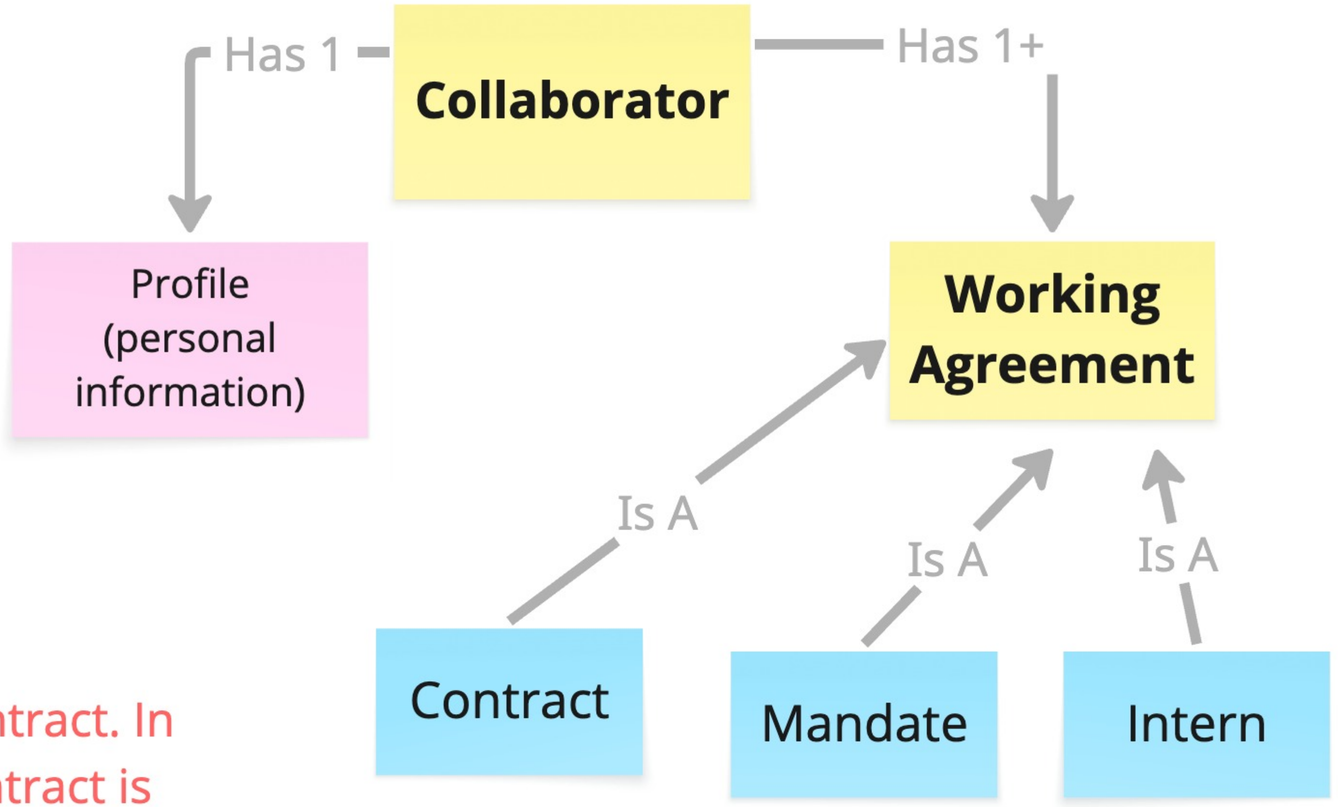
3. Build new features, reduce bugs, innovate faster (write model first ?)

Drifting domain model - example

Model implemented in system

~~Employee~~ **Contract**
- personal information
- contract details
- contract type

Target Domain Model



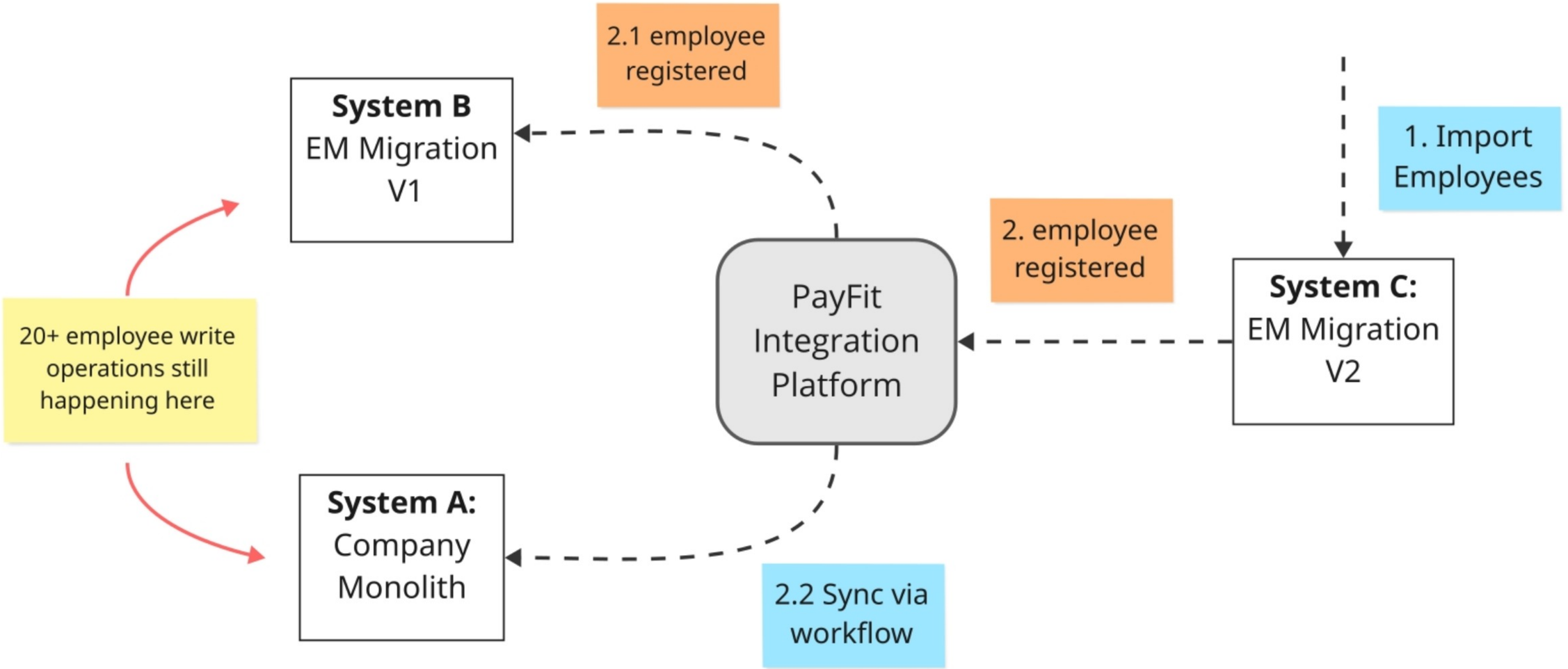
In system A, "personal information" is in every contract. In system C, it's at collaborator level. In system C contract is a subset of working agreements (a new concept)

Migration Pattern Decision

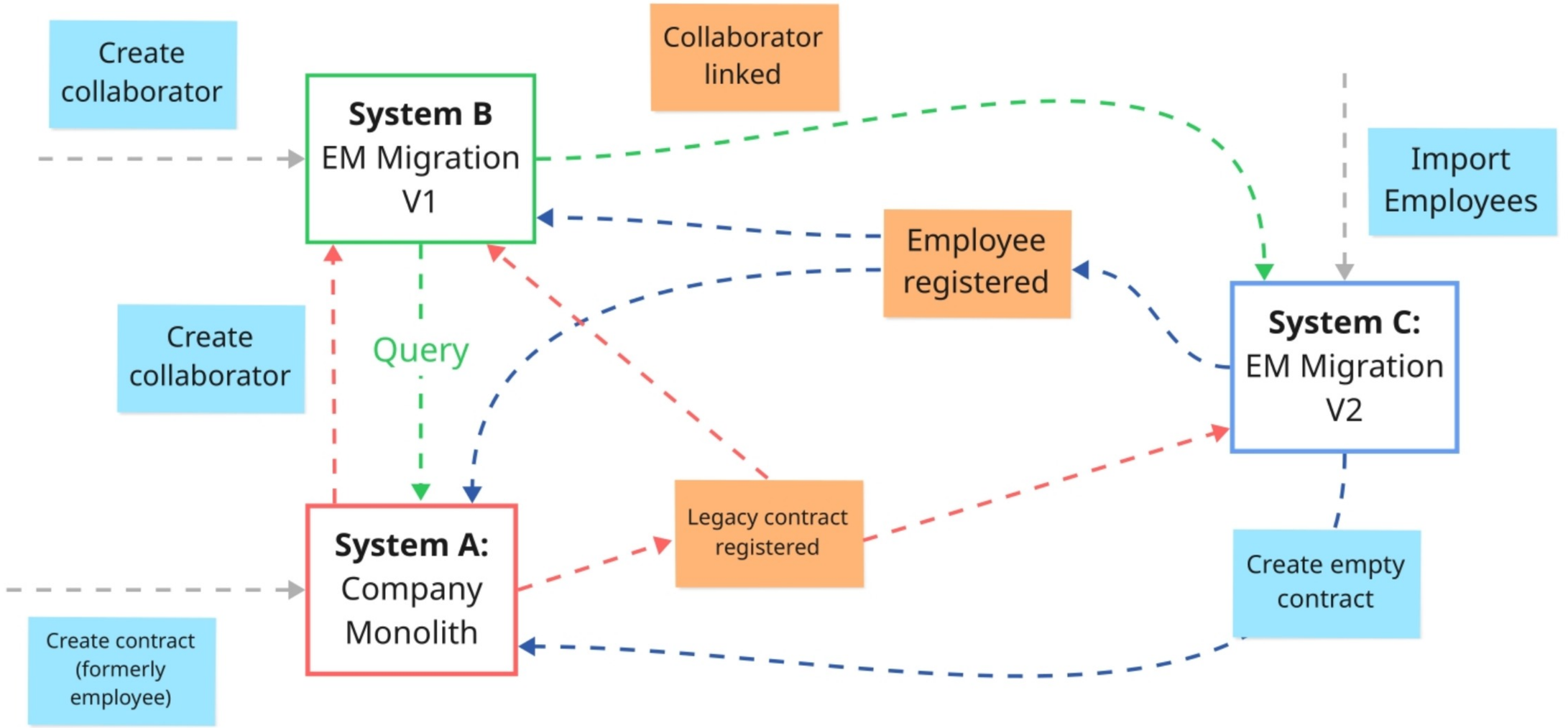
Employee / collaborator

For this entity we began by migrating write operations first so that we could implement new use cases like creation partner integrations

Migration decision: incrementally migrate writes



Incremental = tri-directional sync



Summary - overall happy with the decision 👍

✅ Feels great to take ownership of the domain model and continuously evolve it

✅ Nice to build new features with the new model

⚠️ Synchronisations need to built and maintained

⚠️ Cost of investigating and resolving sync issues can be high sometimes

Migration Pattern Decision

Contract / Working Agreement

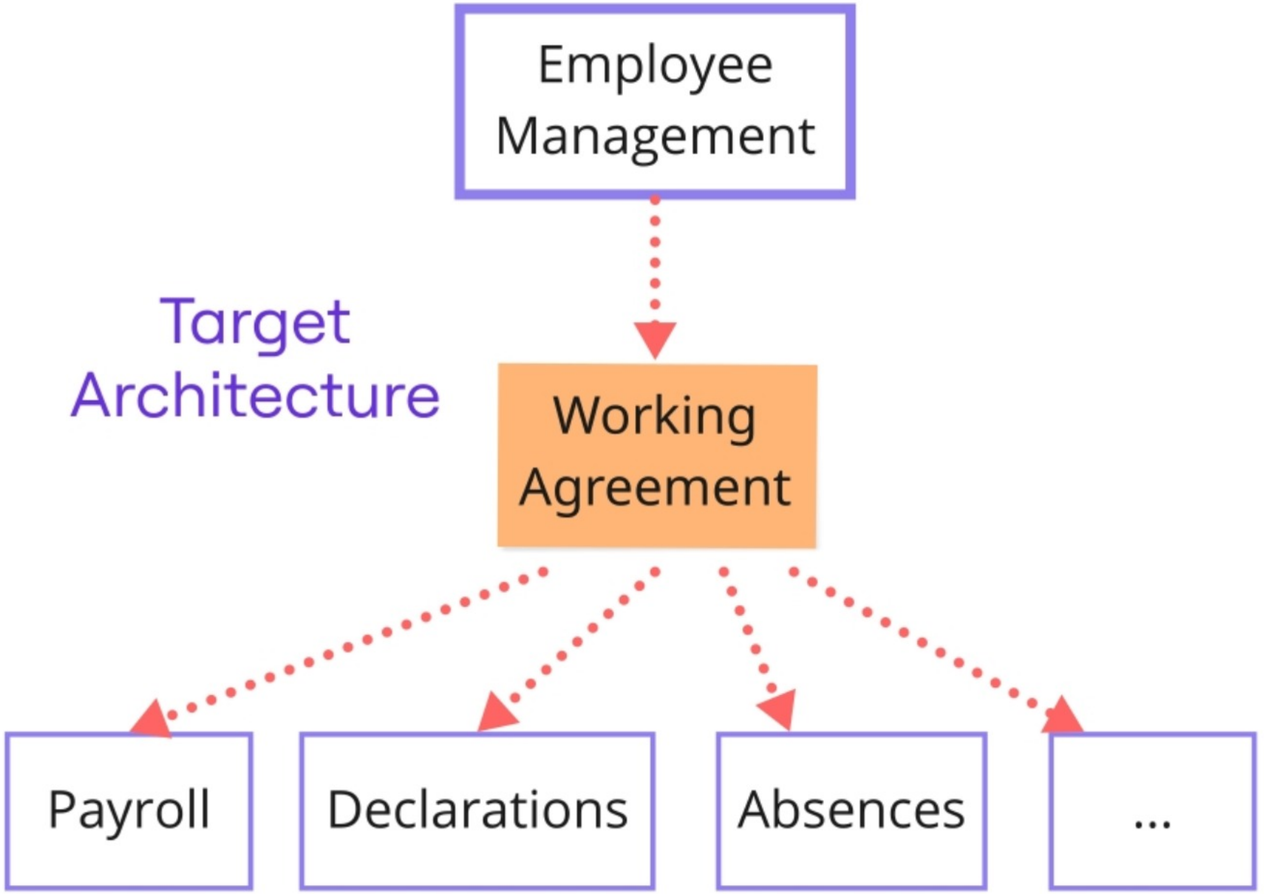
For this entity we began by migrating read (via events) to unlock other domains before taking ownership of write operations which would have taken a long time

Challenge: EM Team is the bottleneck for other teams that want to transform

- Employee Management is upstream of many other domains who want to migrate away from legacy.

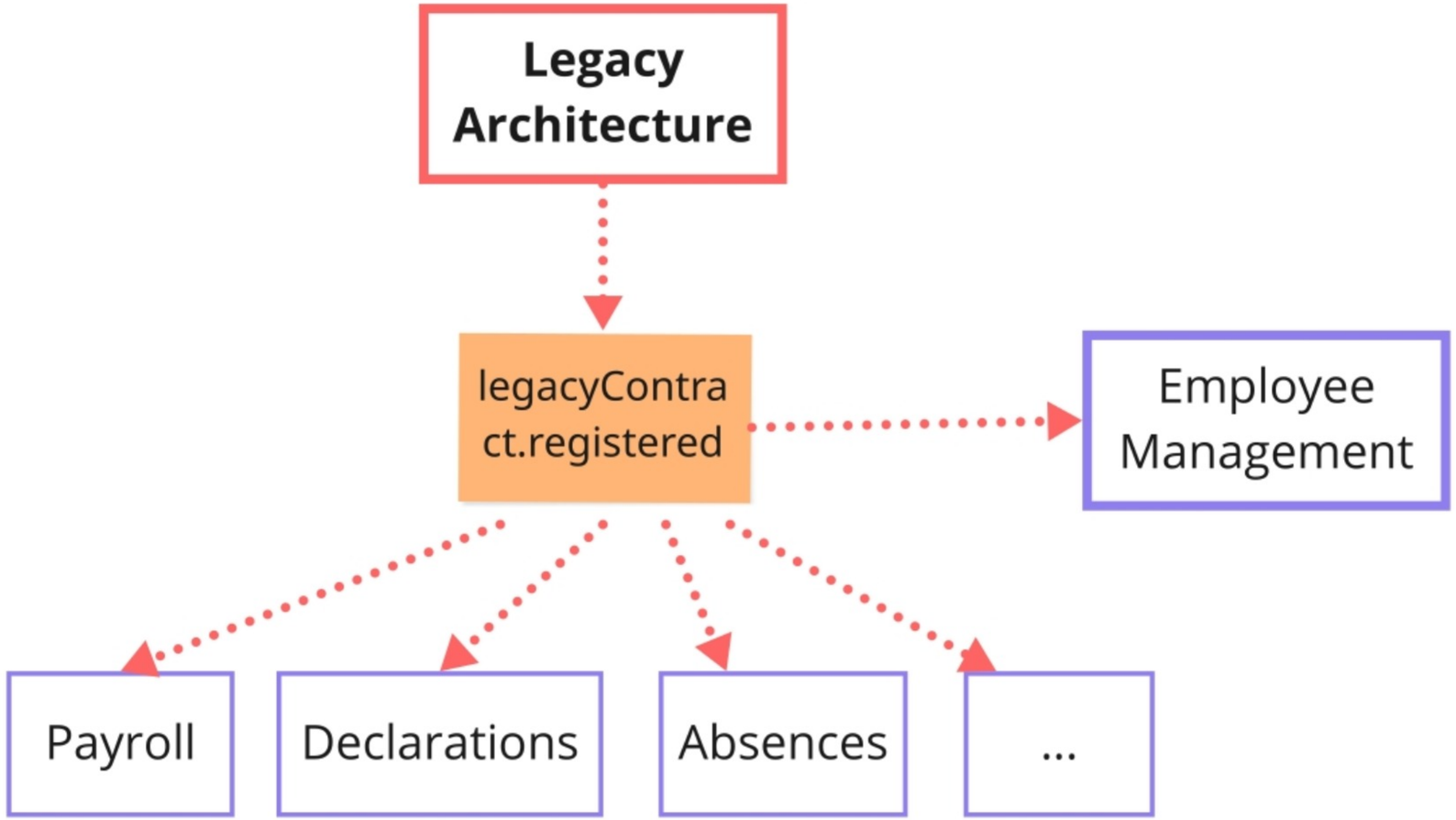
- They need to know about the lifecycle of contracts from the EM domain.

- But the EM team is focused on transforming the Employee/Collaborator first

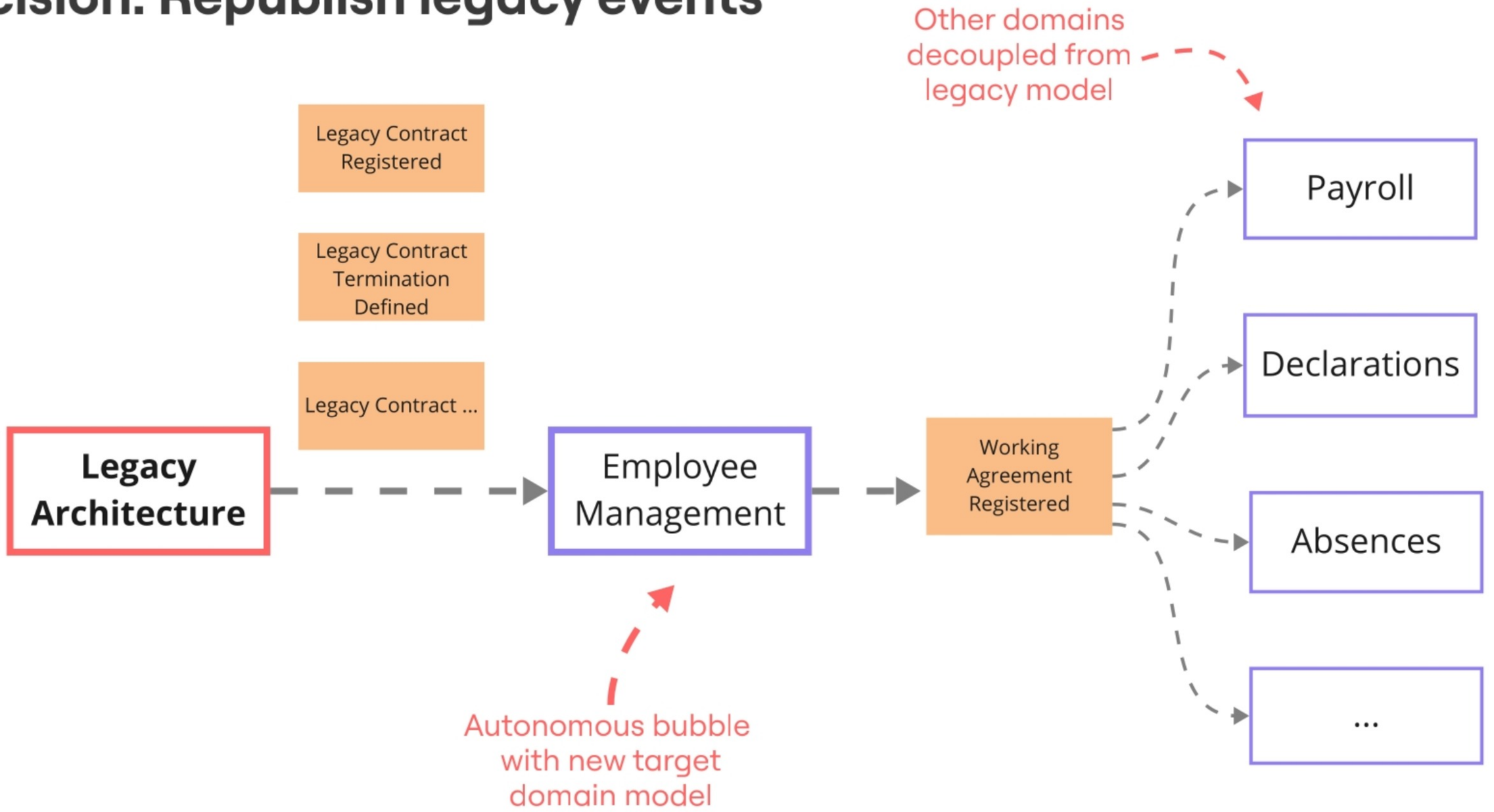


Option: Take events from legacy model

Proposed migration step architecture



Decision: Republish legacy events



Building your new model on top of an underlying system/model that is not rock solid requires very careful consideration.

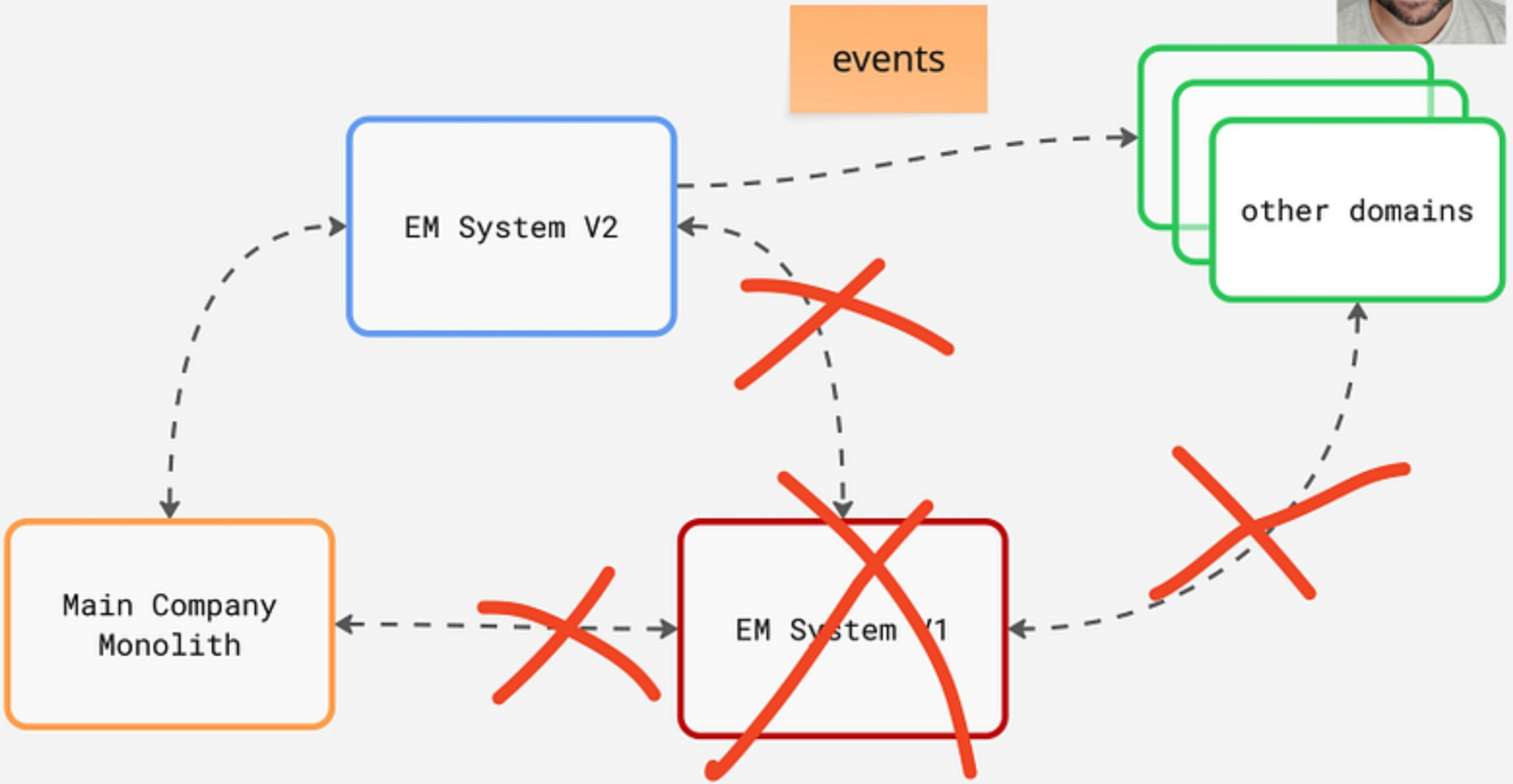
It's not just a few lines of mapping code. It can be models with tonnes of intricate nuances (and workarounds) built up over years.

"write model first" and "read model first" were the starting points on the two projects.

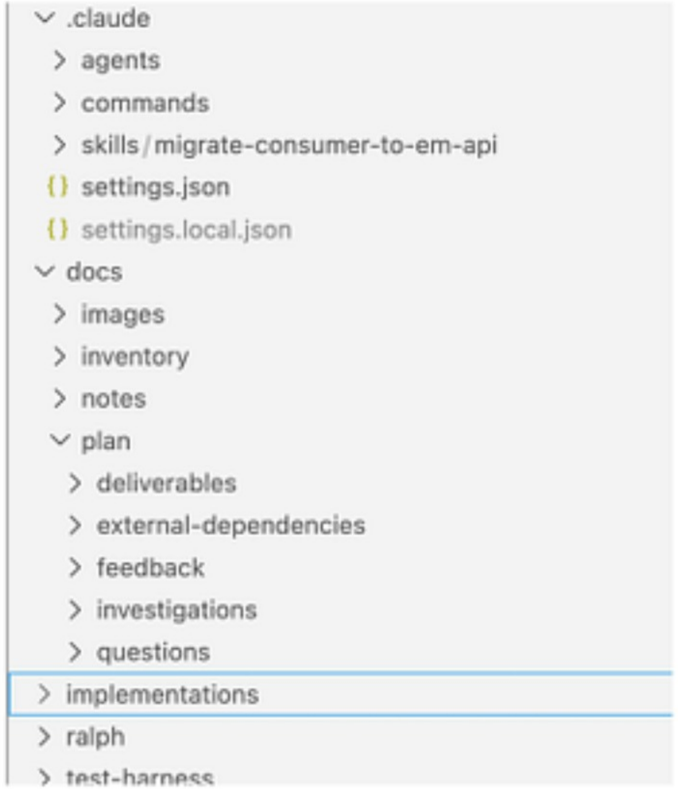
But on each project we made many micro decisions about ordering. It wasn't strictly all reads then all writes

3.
Legacy Modernization with AI at PayFit

Charles and Nick's: V1 Retirement Plan



We created a git repository as our control center

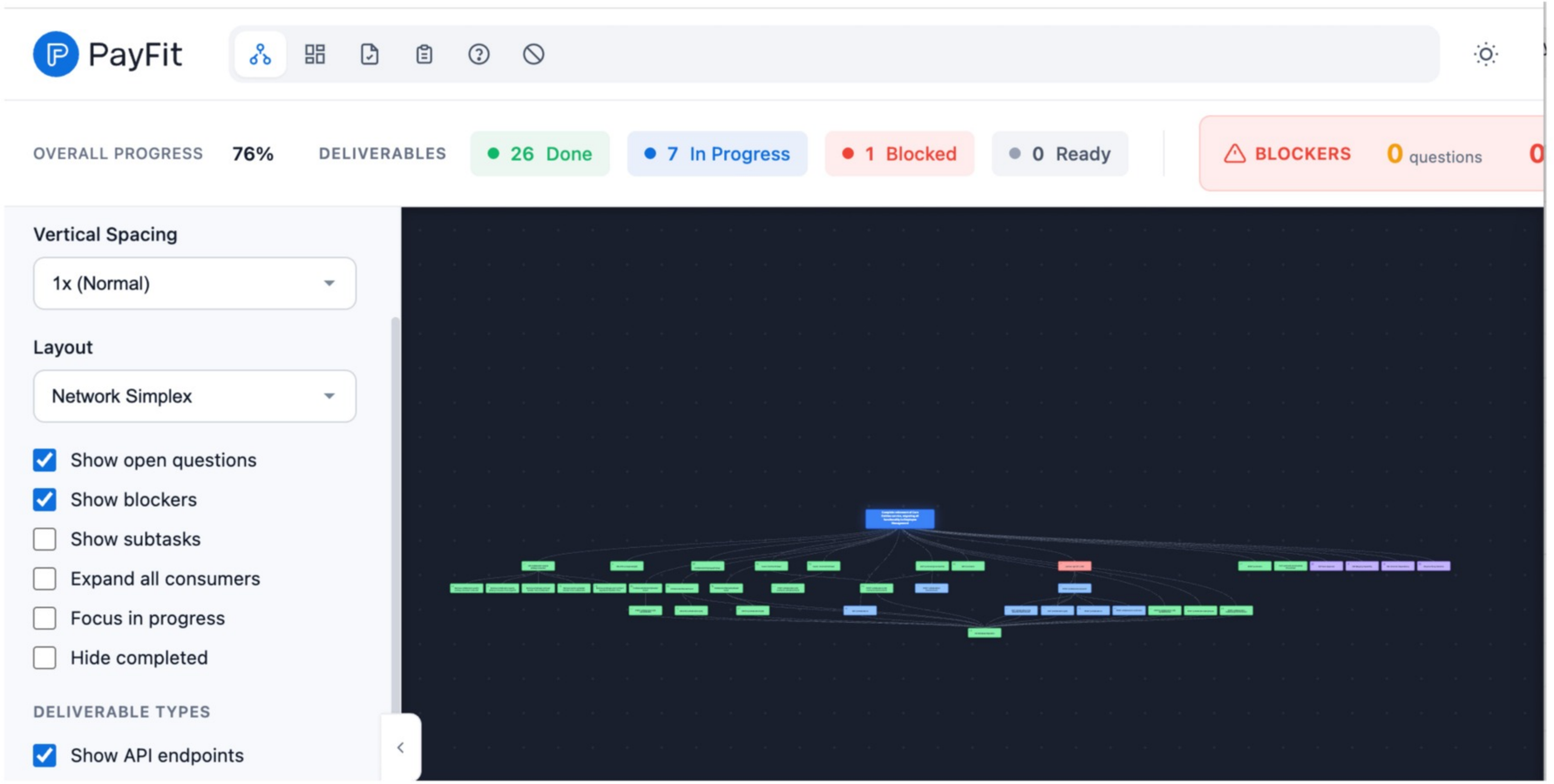


- We didn't need all the functionality of JIRA
- We wanted to store all of the knowledge about our project in a place where AI could easily access it (we optimised for AI consumption)
- Full git history of all changes in the project

Full control over our workflow: AI commands and knowledge storage co-designed

```
/update-plan <natural language describing what changed>

1. Parse ALL updates from user's input

 The input may contain multiple paragraphs with several updates. Parse each one:
   - Scan the entire text for updates (approvals, PRs, completions, questions,
feedback, etc.)
   - For each update found, identify the target item (search by name, title, or ID)
   - Extract details (PR numbers, step IDs, resolution text, feedback content)
   - Build a list of all changes to propose
```

Built our own project management tool around our conventions (cost us nothing)



This tool cost us nothing in terms of time. AI can build tools like this quickly and in parallel with other work

Hallucination proof: Frontmatter has a schema which is validated on every change

```
<schema-validation-errors>
    SCHEMA VALIDATION ERRORS -
    1 files need fixing:
      docs/plan/deliverables/migrate-partner-api-bff-to-em.md:-
    root: must NOT have additional properties"consumers"
</schema-validation-errors>
```

Use deterministic tools

OpenAPI specs as source of truth for API endpoints

EventCatalog as source of truth for events (including publisher and subscribers)

Nx as source of truth for codebase structure and package dependencies

Benefit from fresh context windows wherever possible

/discover-inventory

```
For each **source** repository:

1. List endpoint files: `docs/inventory/<source-repo>/endpoints/*.md`
2. Spawn subagent **api-analyzer** for each endpoint (can run in parallel) with:
 - inventory_path: path to the endpoint file
 - repo_path: from `reference-sources.md`

Wait for all analyzers to return `FINISHED`.

...
```

Force precision: define the algorithm not the end state

Agent: api-analyzer

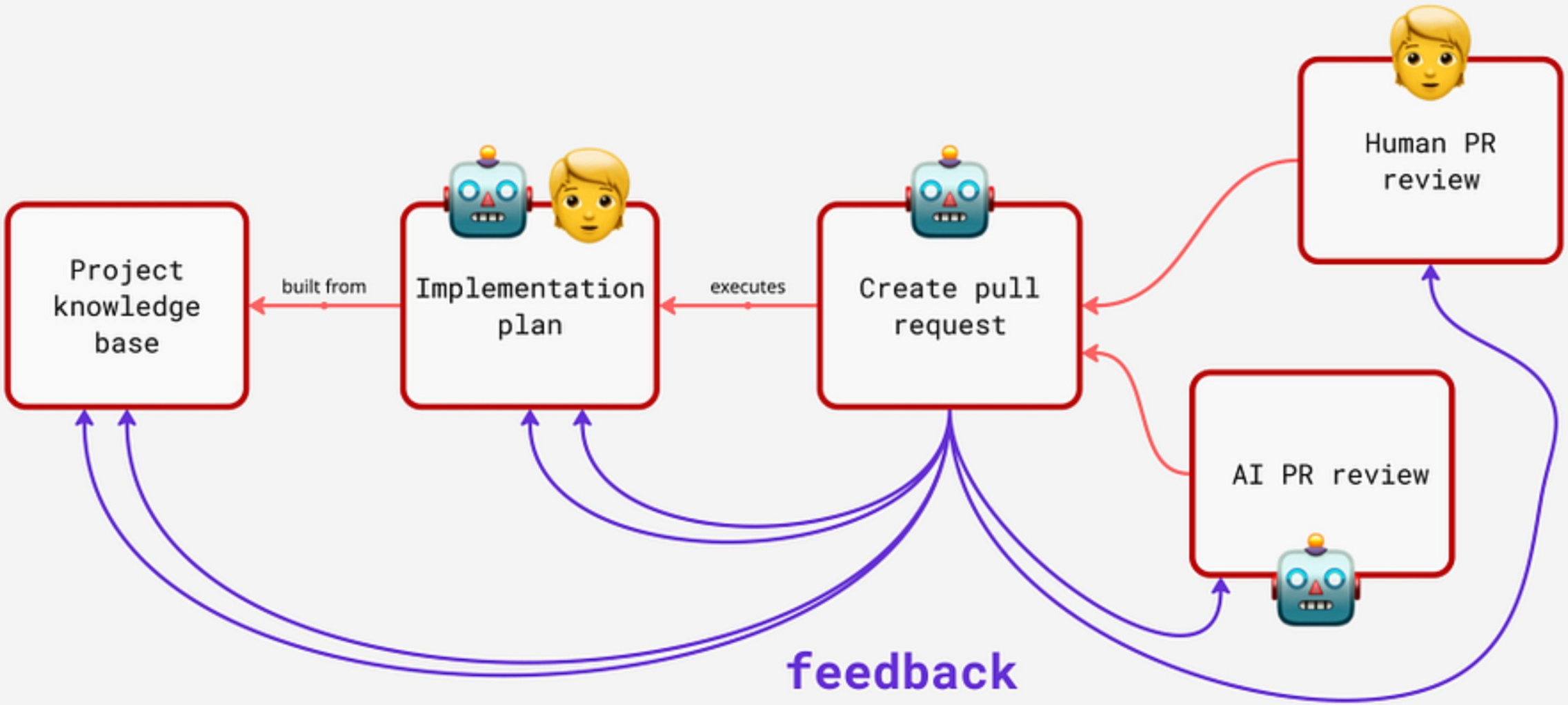
```
### Step 2: Deep Code Walkthrough

Follow the complete code path from route to response:

1. Route definition → Controller method
2. Controller → Service layer
3. Service → Repository/Data access
4. Service → External API clients

Document each step with **exact file path and line number**.

> **PayFit default:** NestJS is the standard framework. Look for these patterns
first. Otherwise, find equivalent patterns for the framework in use.
```

Domain fit: Where should code be migrated to?

Agent: migration-architect

```
### Step 2: Score Domain Fit Options

**Populates:** `## 4. Options Considered > ### Domain Fit` table

Assess based on:

- **Semantic alignment:** Does endpoint name match target domain language? Do related responsibilities
belong to same domain?
- **Data alignment:** Does endpoint access same entities as target? Does target already own this data?

For each option, provide confidence + specific rationale:

- target (assumed service)
- other:<service-name>
- split
- deprecate

...
```

Feedback loops: fix the system, don't patch it

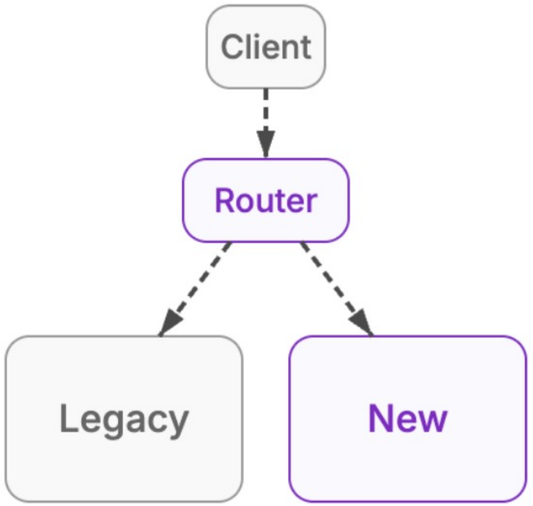


There is a more detailed slide deck of this case study at:

nick-tune.me/talks

A few final words...

More Patterns

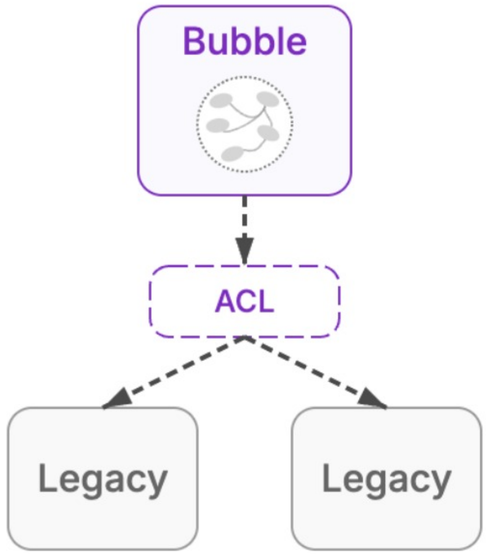


MIGRATION

Strangler Fig

The Strangler Fig pattern is a metaphor invented by Martin Fowler to describe a gradual migration away from a legacy system to a new system (aka the strangler fig application).

Read more →



MIGRATION

Bubble

The Bubble pattern is about creating a new domain model to encapsulate and gradually replace a specific subdomain/subsystem. Eventually, the bubble should pop.

Read more →

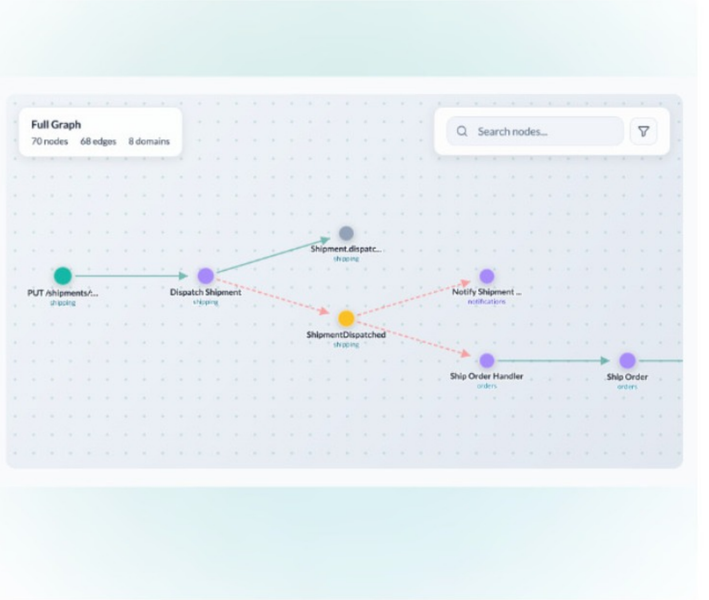
legacy-modernization.io

Rivière

Living Architecture

Extract flow-based architecture from code as living documentation

Get Started | View Demo | GitHub



- build graph of your architecture with deterministic tooling and AI
- AI harness in /tools/dev-workflow-v2 (planning & execution)

`github.com/ntcoding/living-architecture`

careers.payfit.com/

PayFit

Nick Tune

Software Engineer at PayFit

nick-tune.me

/in/nick-tune

@nick-tune.me