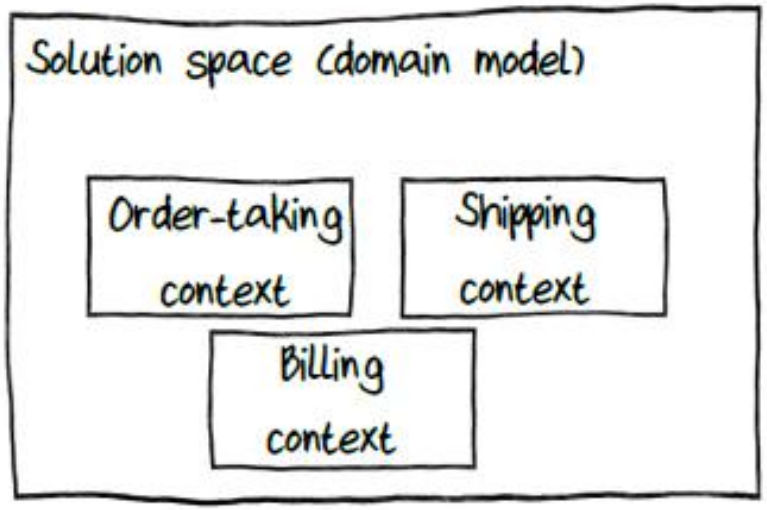
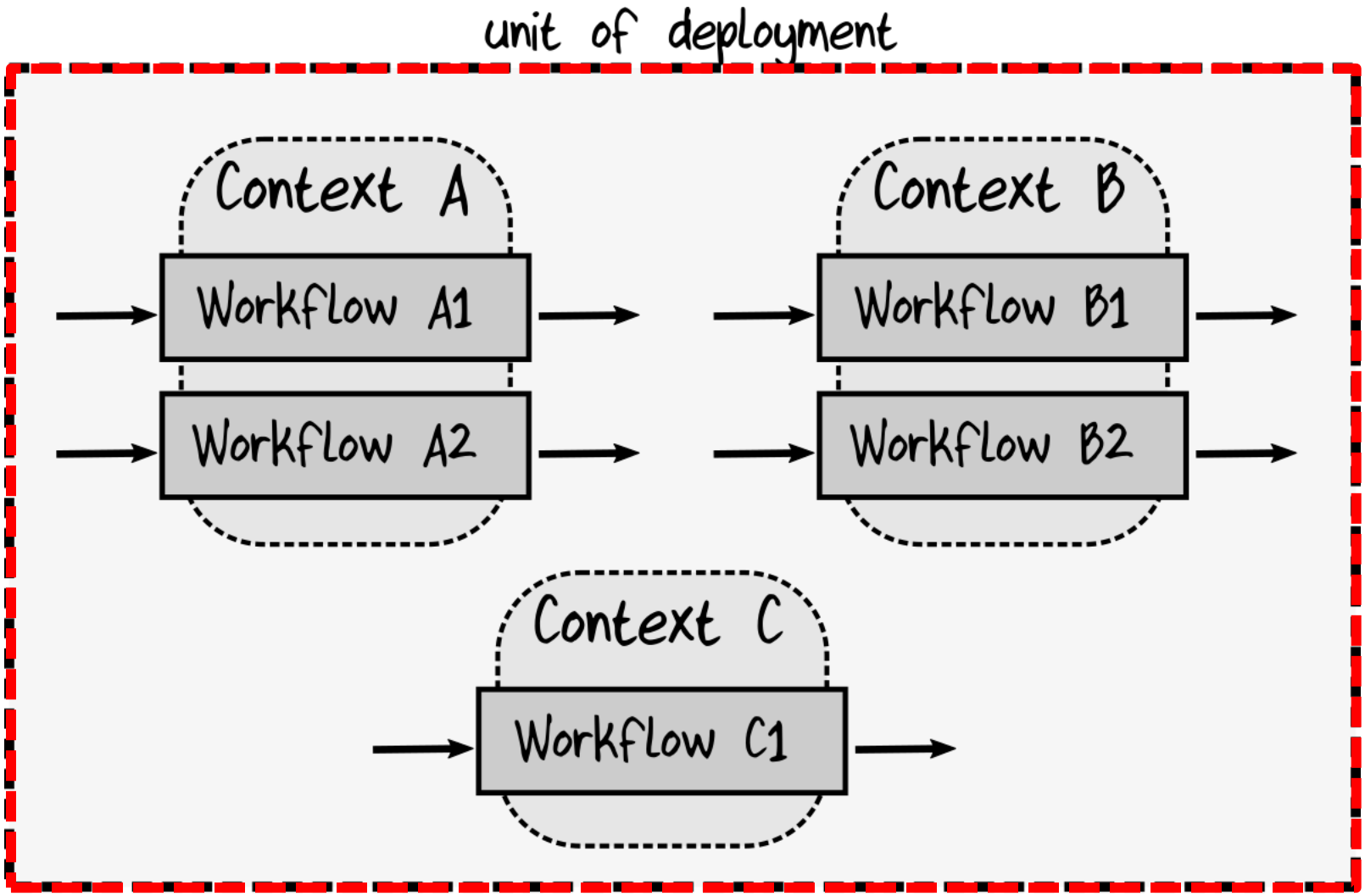
Functional Architecture

Bounded contexts are autonomous software components

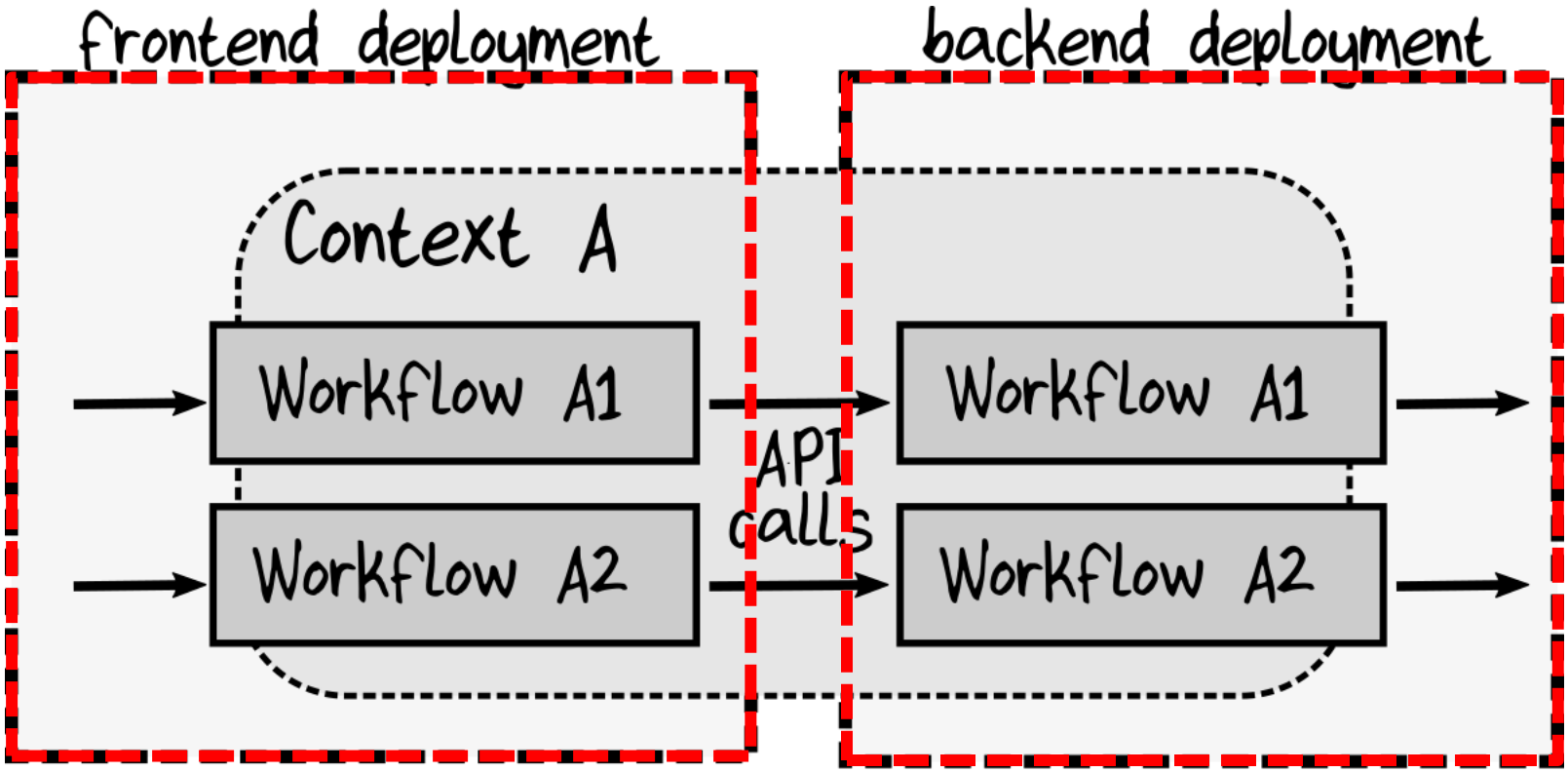


3 different architectures…

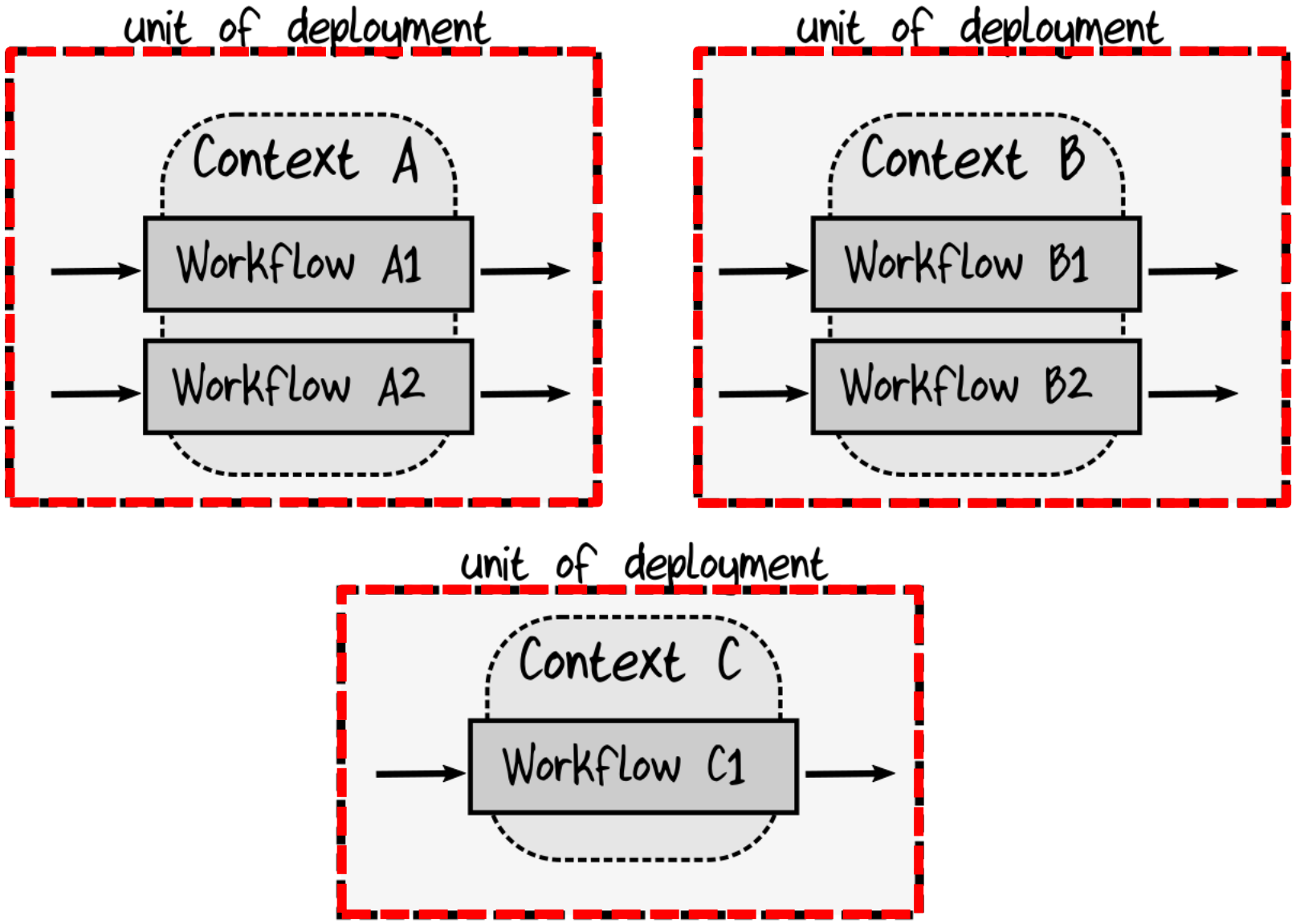
- For monoliths:
  - Each bounded context is a separate module with a well-defined interface
- For service-oriented architecture:
  - Each bounded context is a separate container
- For serverless:
  - Each individual workflow is deployed separately



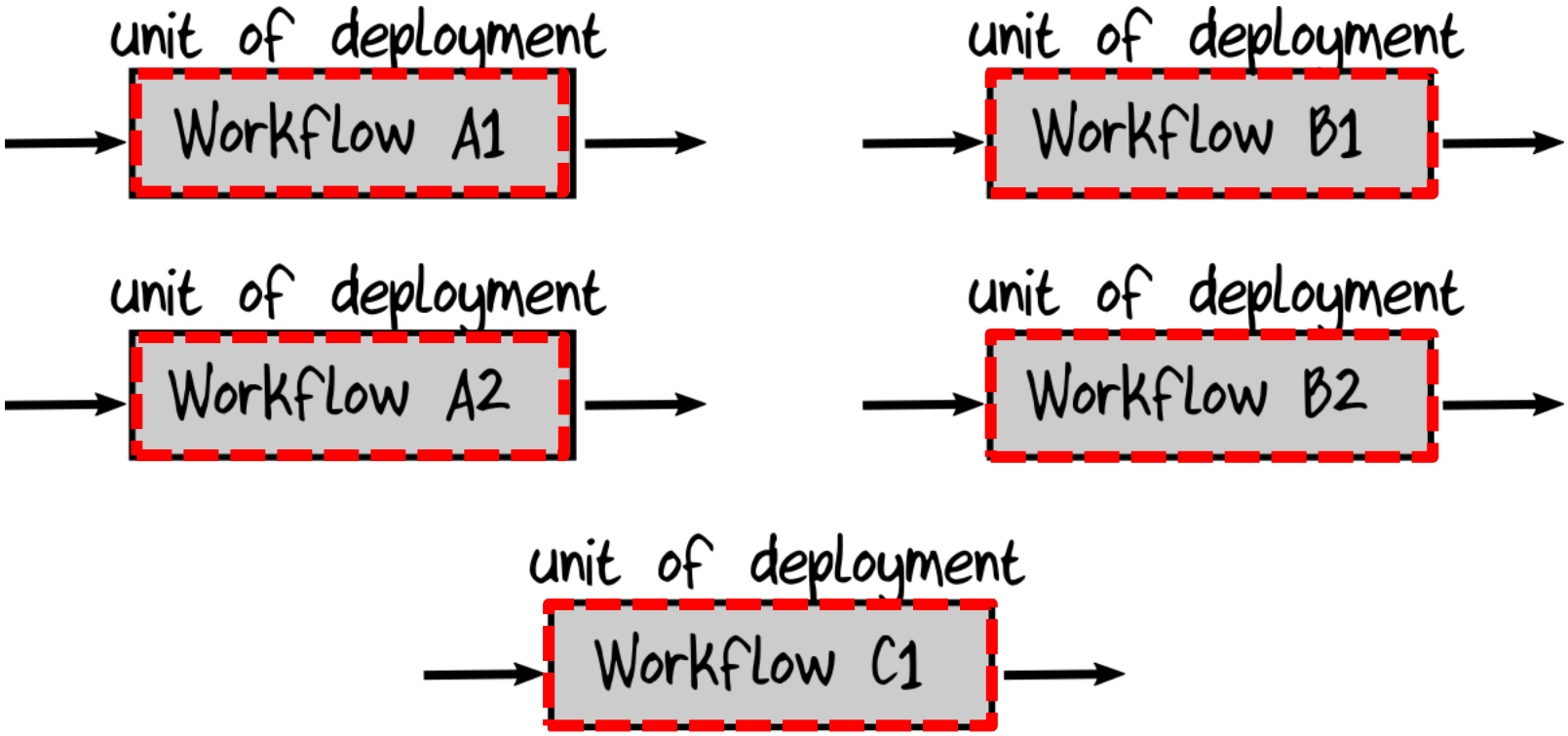
Modular Monolith



Split deployment



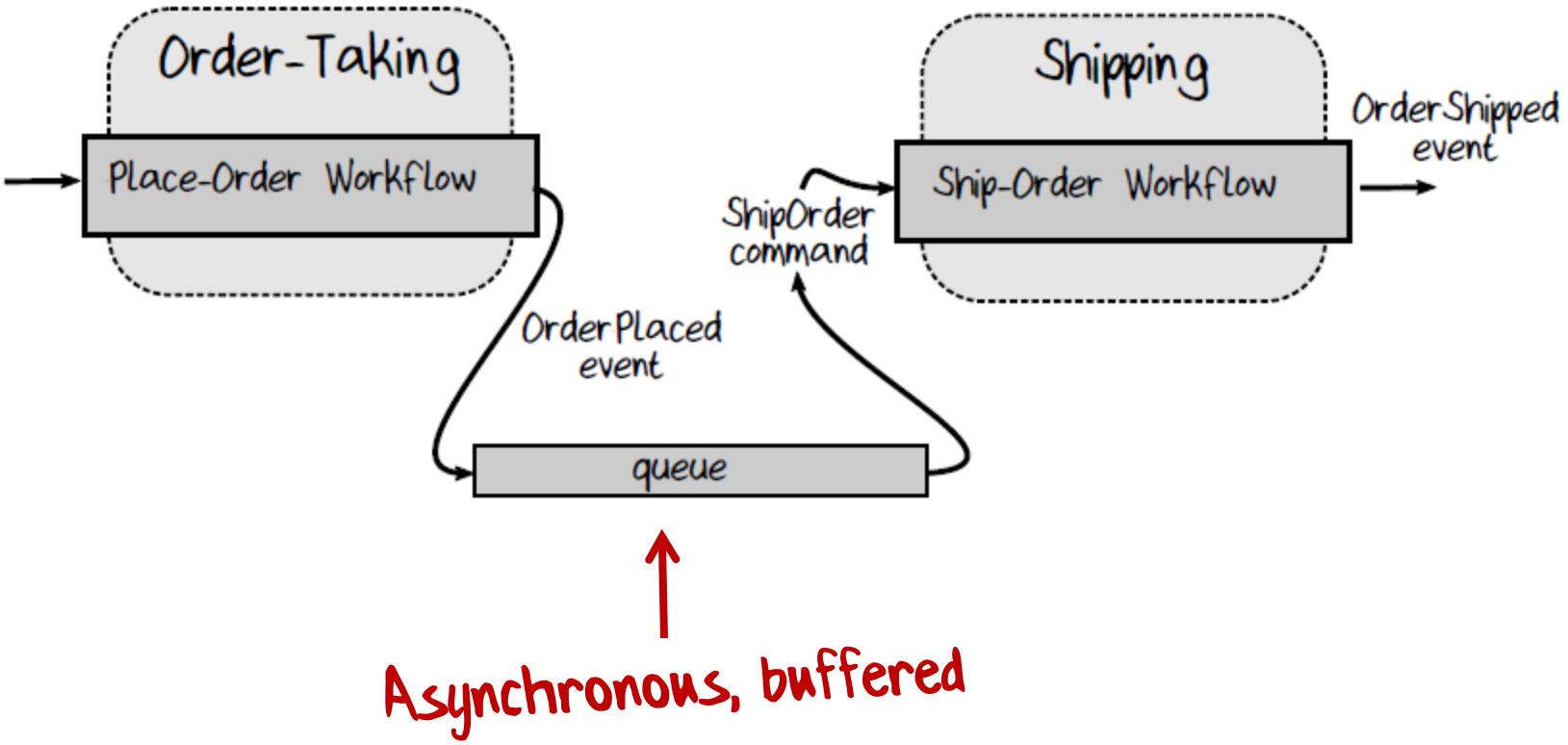
Microservices


unit of deployment
Workflow A1
unit of deployment
Workflow B1
unit of deployment
Workflow A2
unit of deployment
Workflow B2
unit of deployment
Workflow C1

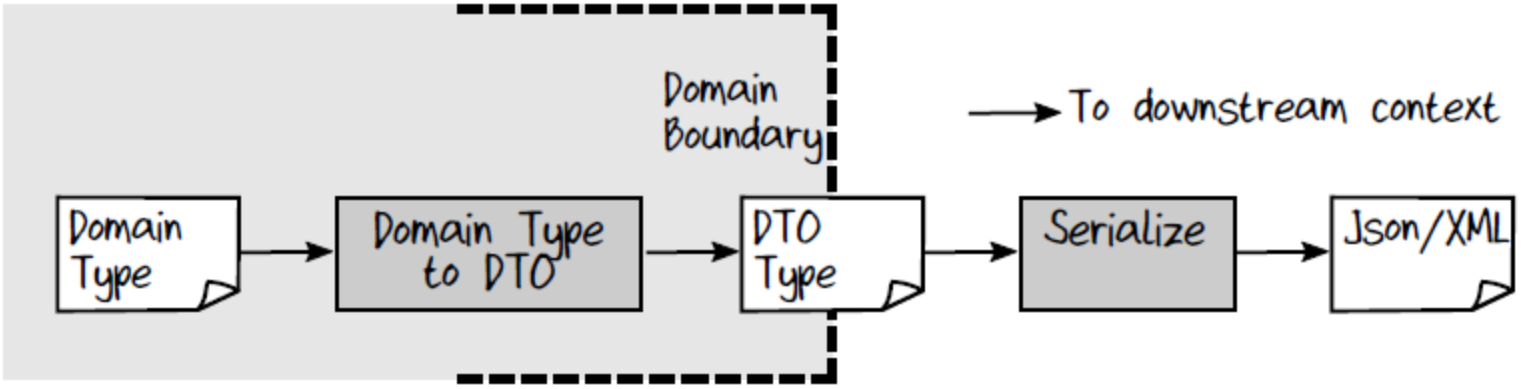
Serverless

How to communicate between bounded contexts?

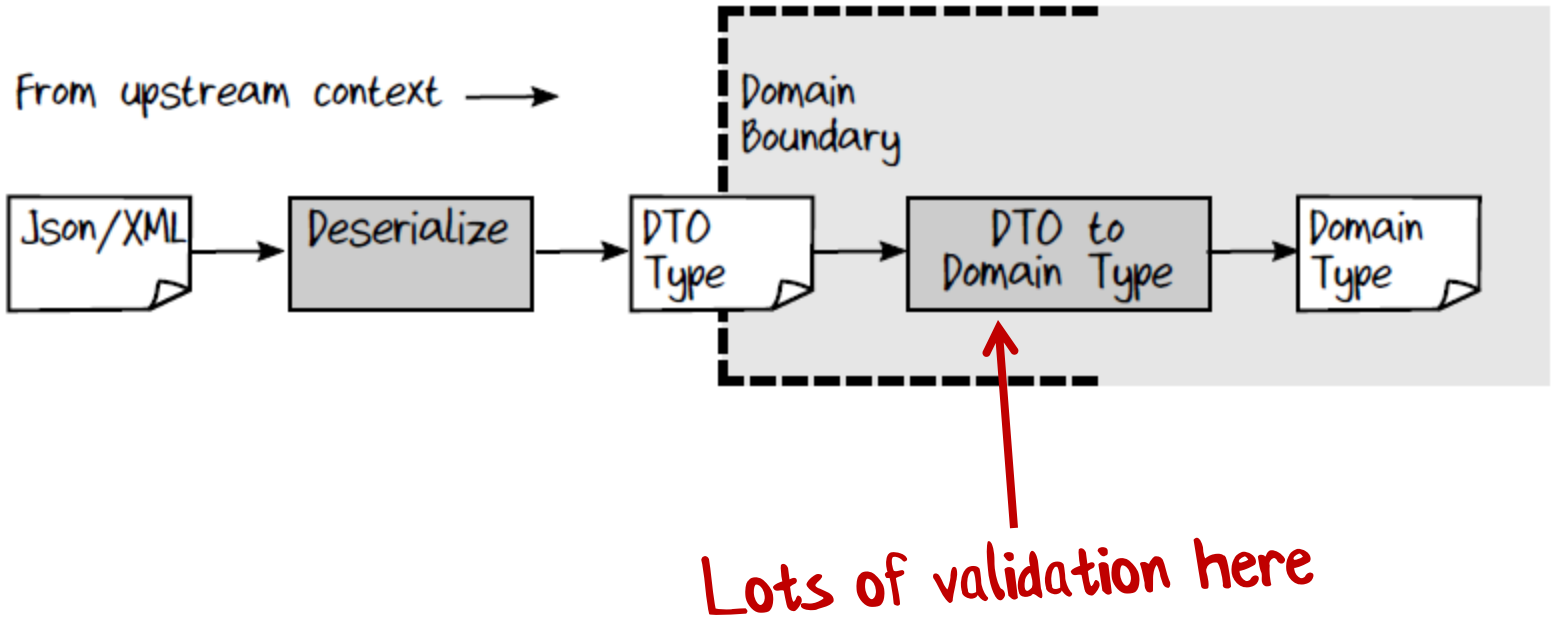
Answer: queues



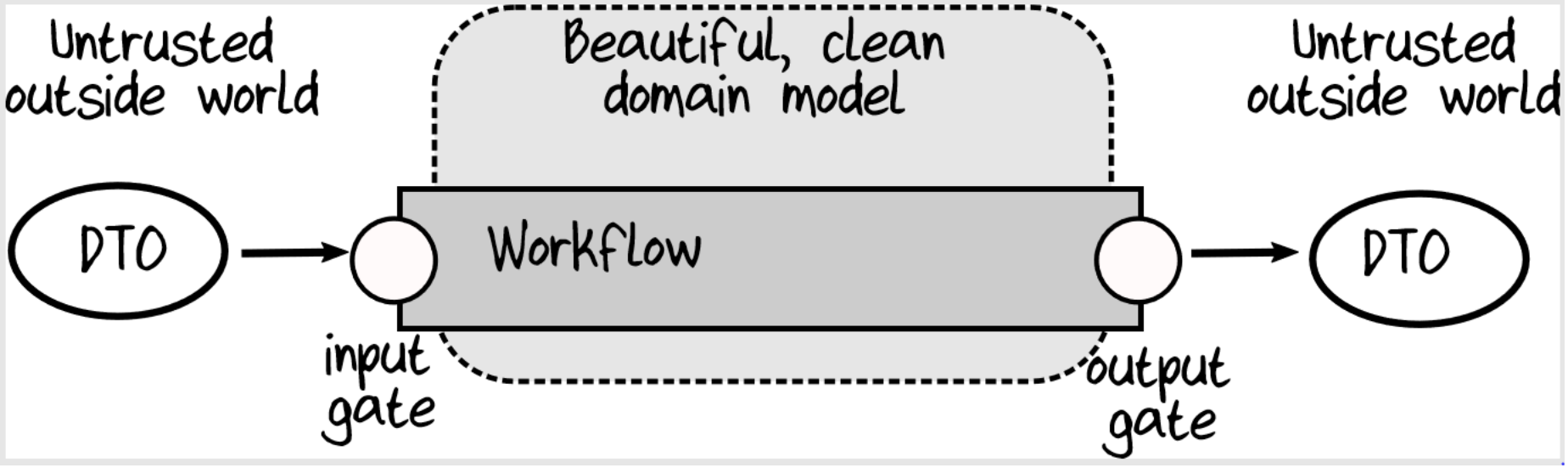
On the way out, domain objects become DTOs



On the way in, DTOs become domain objects

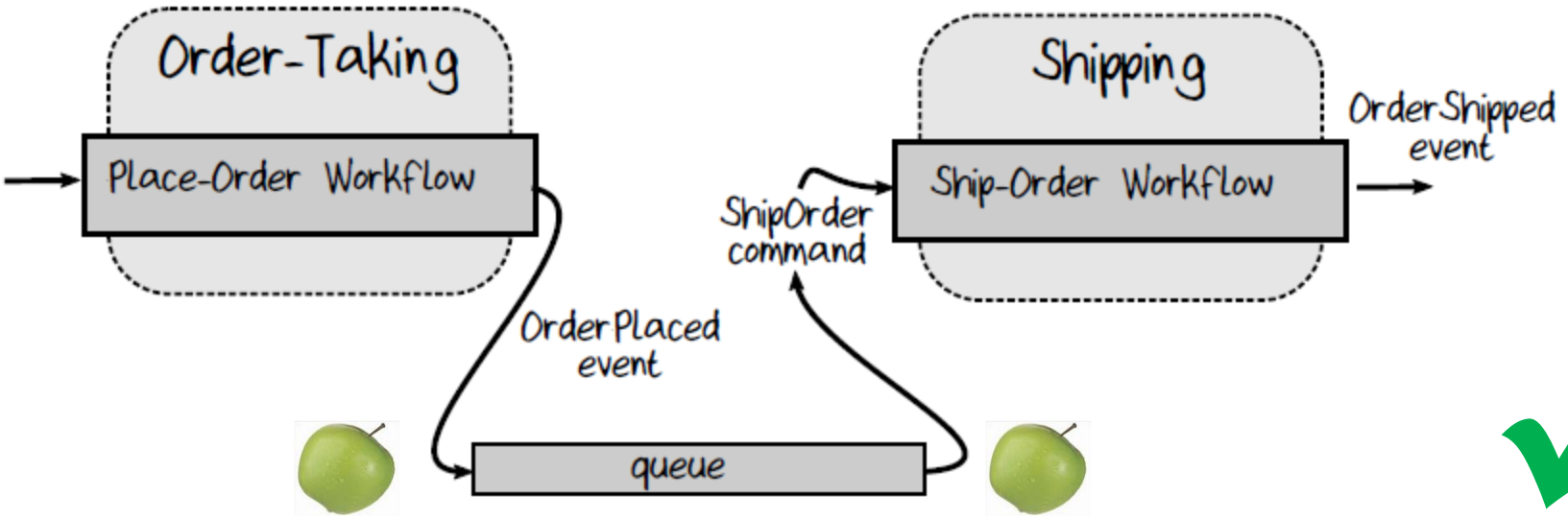


Do not trust the outside world

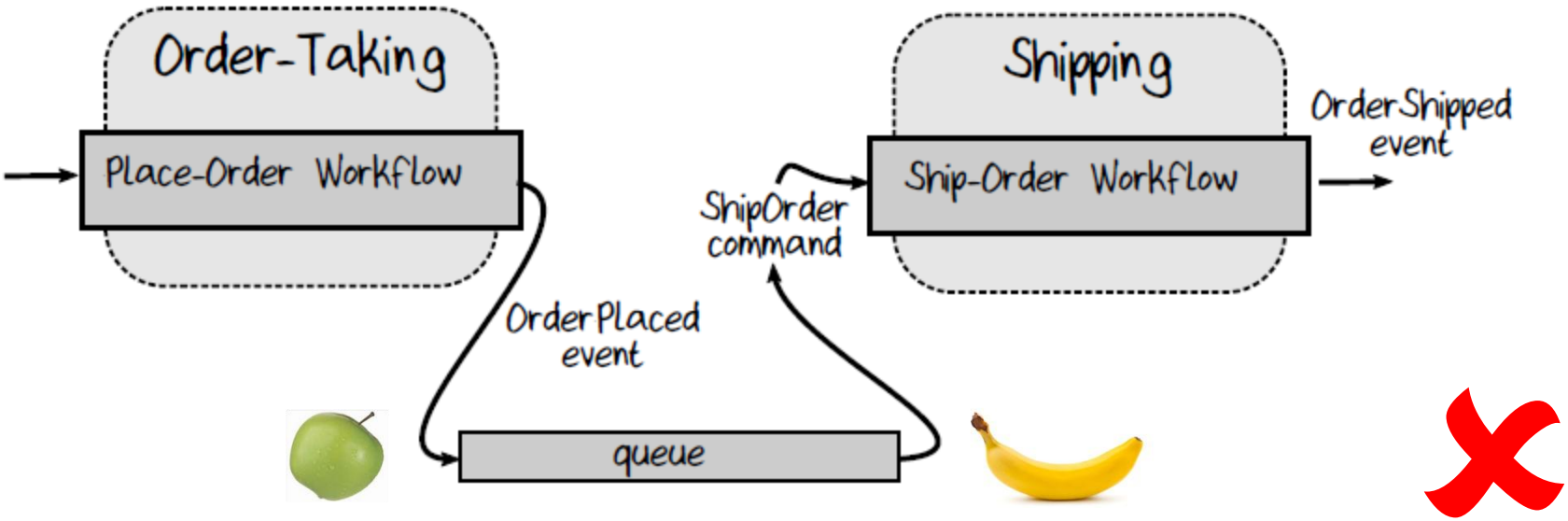


DTOs are contracts
between bounded contexts

DTOs are contracts



DTOs are contracts



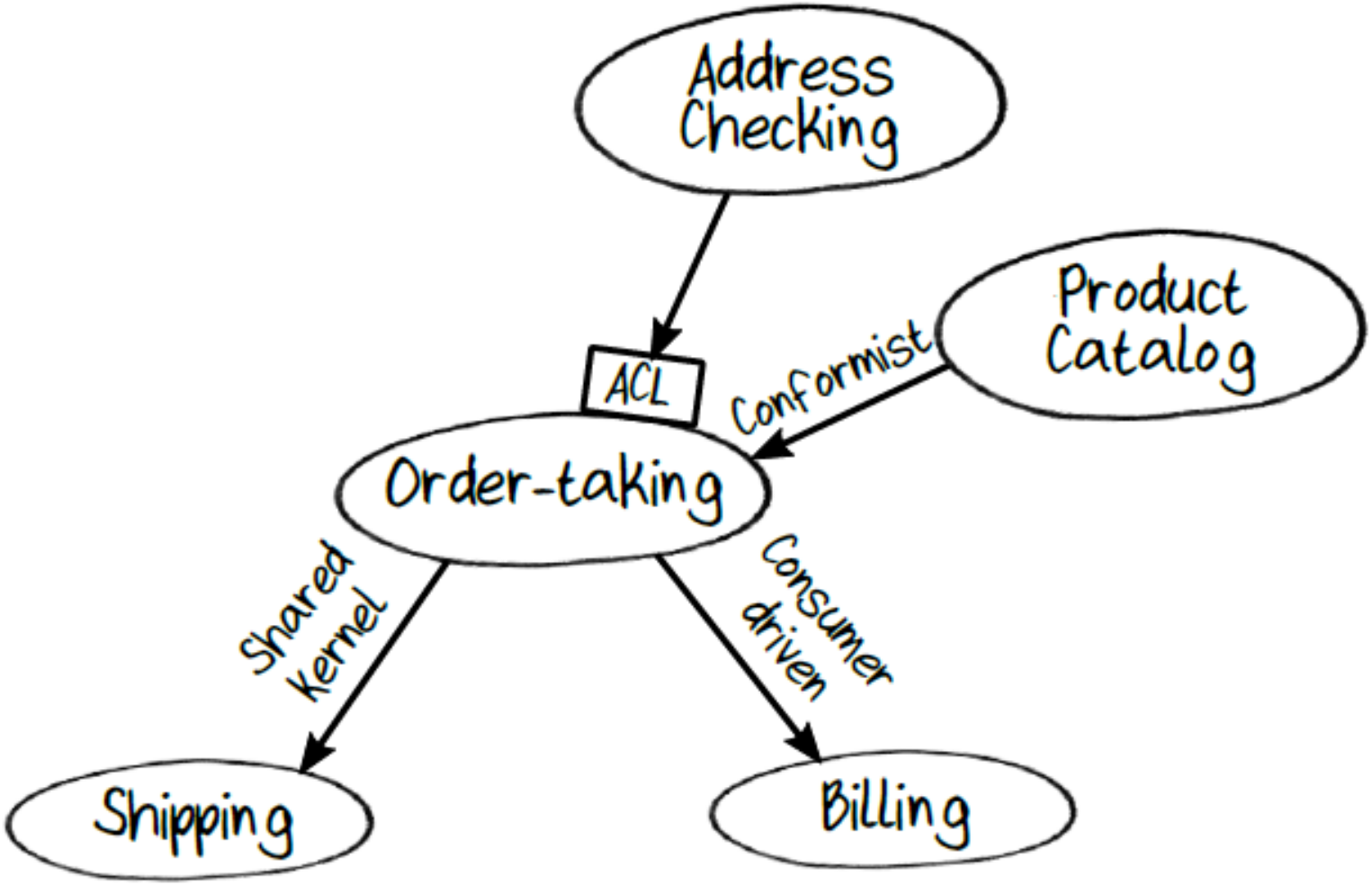
Relationships between contexts

- *Shared Kernel*
  - Two contexts share some common domain design, so the teams involved must collaborate.
- *Consumer Driven*
  - The downstream context defines the contract
- *Conformist*
  - The downstream context accepts the contract provided by the upstream context

Anti-Corruption Layer

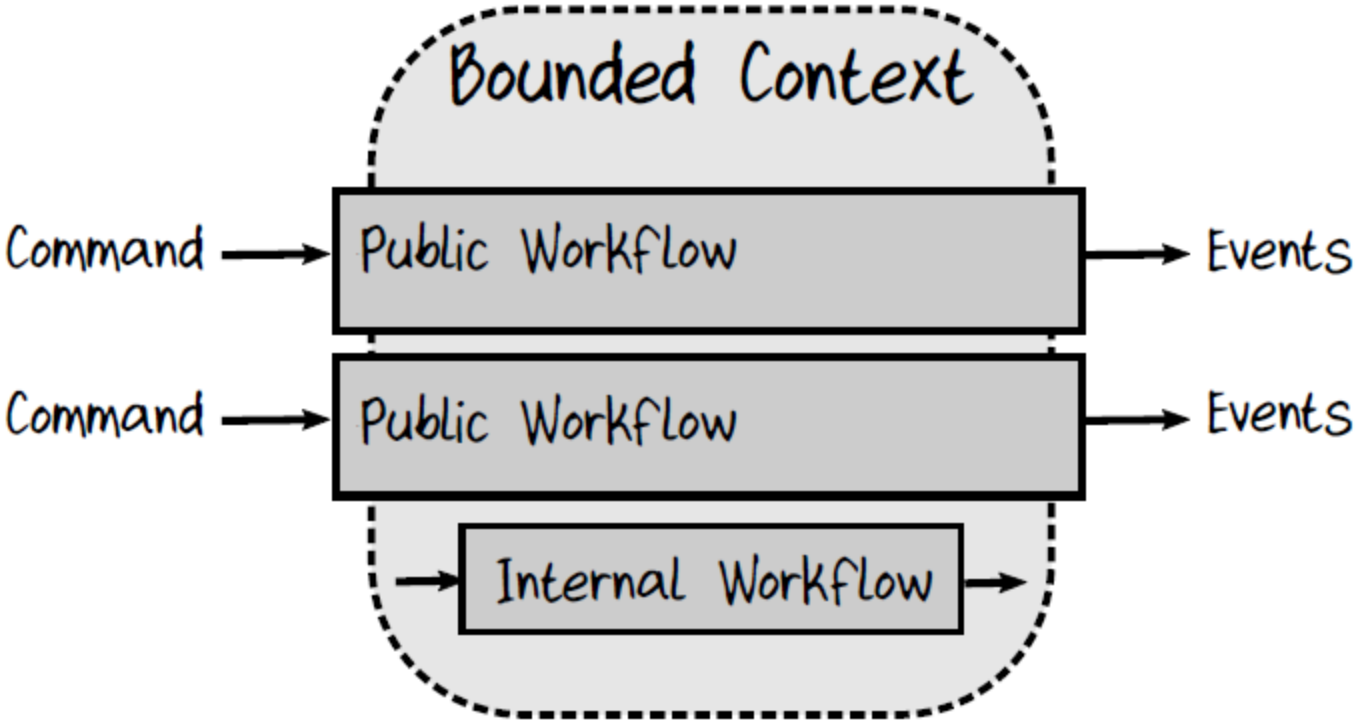
- Acts as a translator between two different languages
  - the language used in the upstream context
  - The language used in the downstream context

Example of different contracts

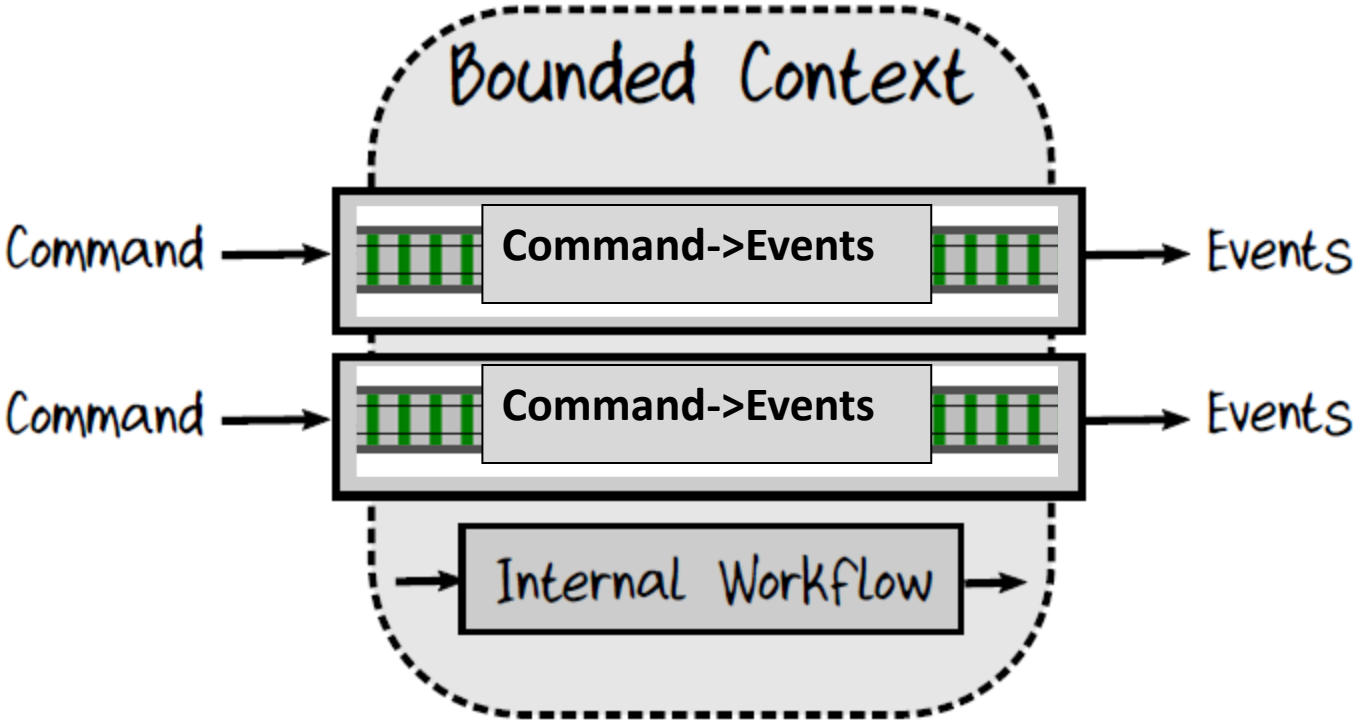


Workflows within bounded contexts

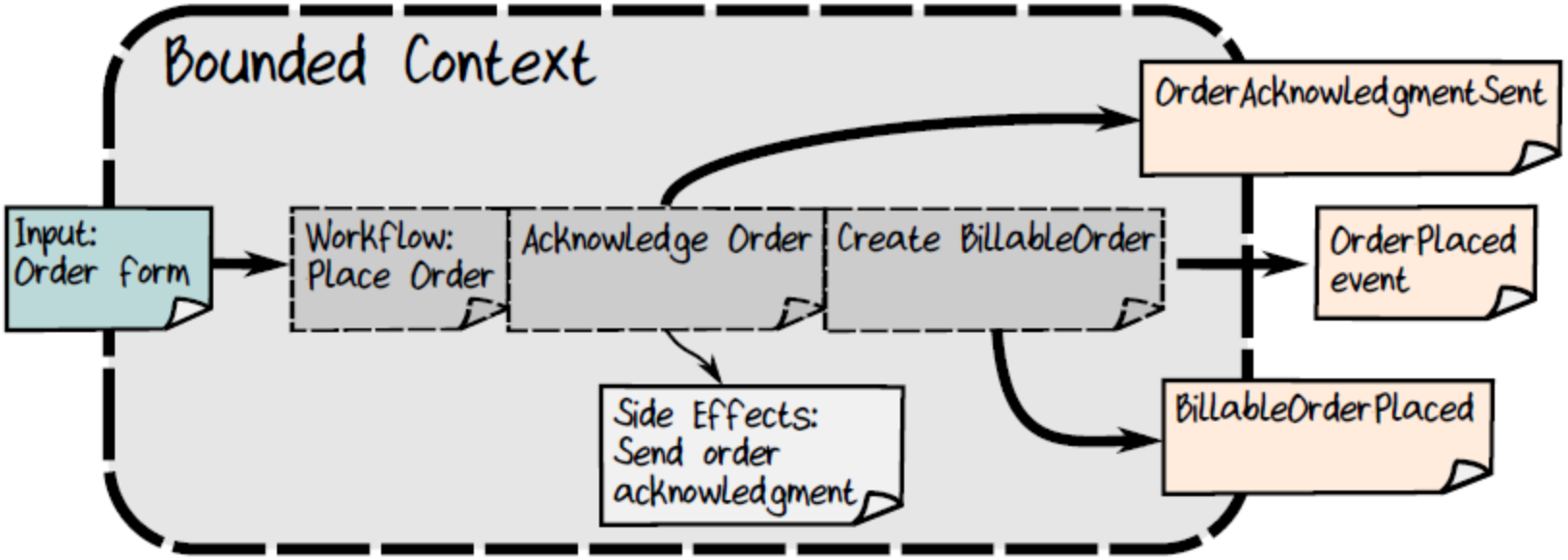
Workflow inputs and outputs



Workflows are functions!

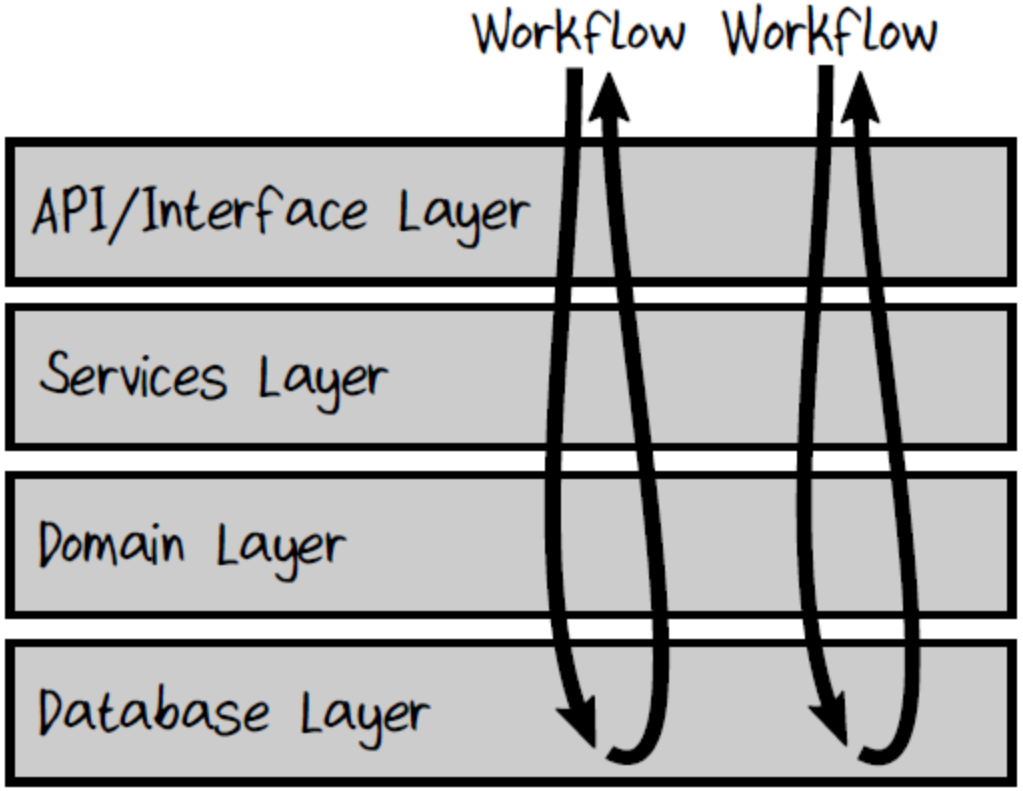


Workflow example



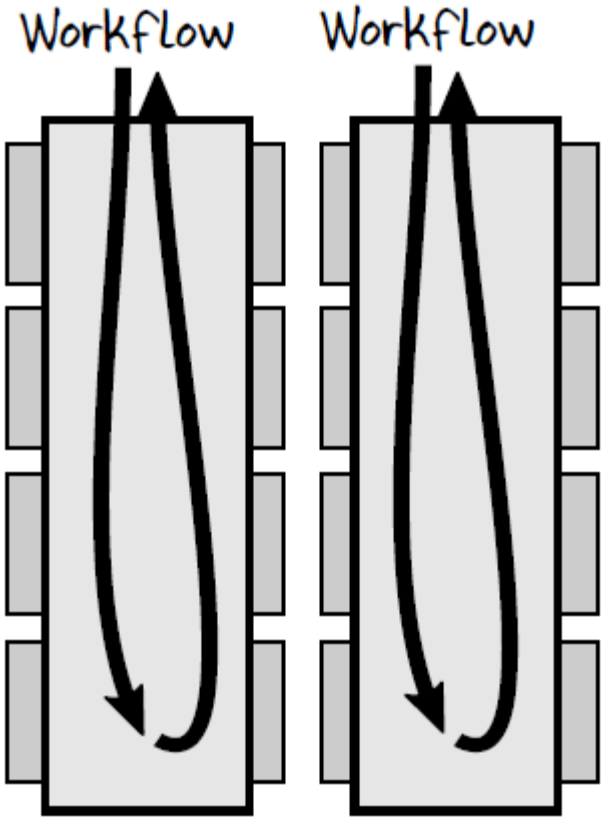
Implementing a workflow functionally

Traditional layered model



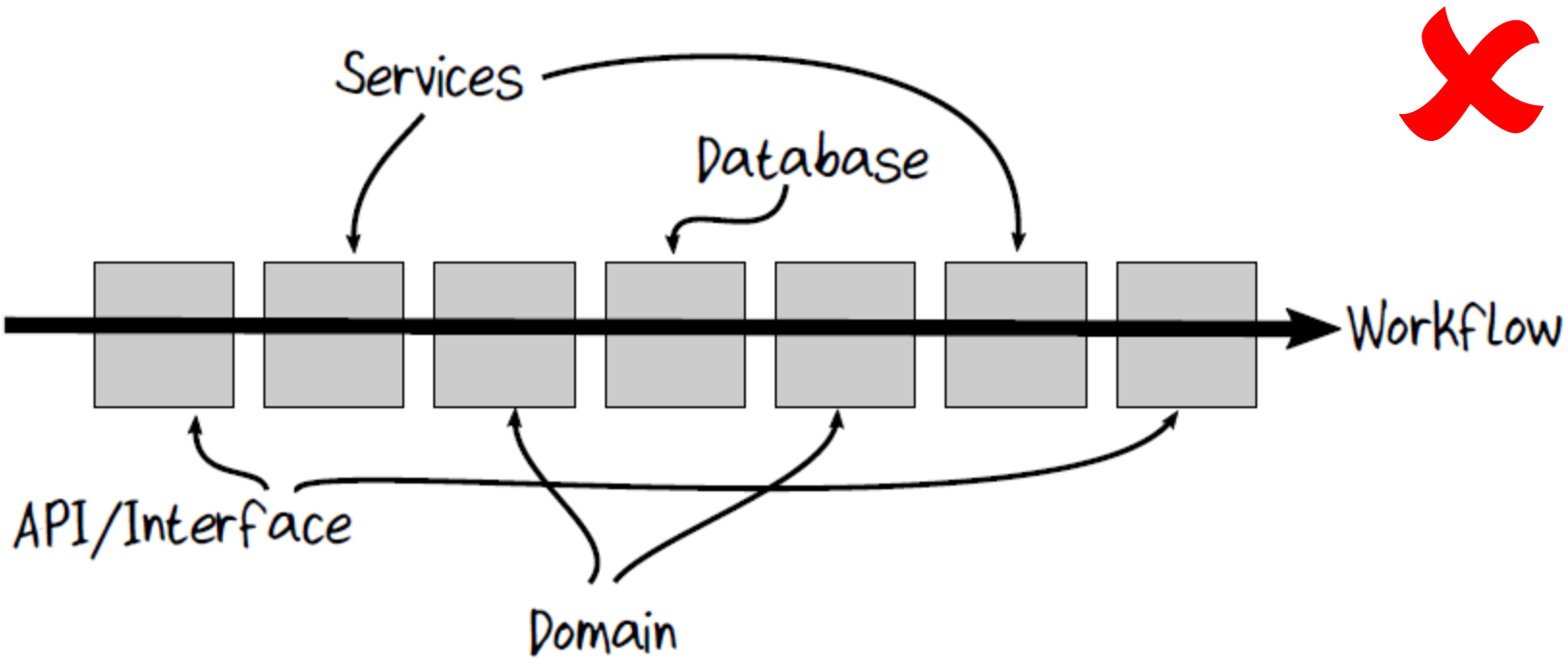
A change to the way that a workflow works means that you need to touch every layer.

Vertical slices



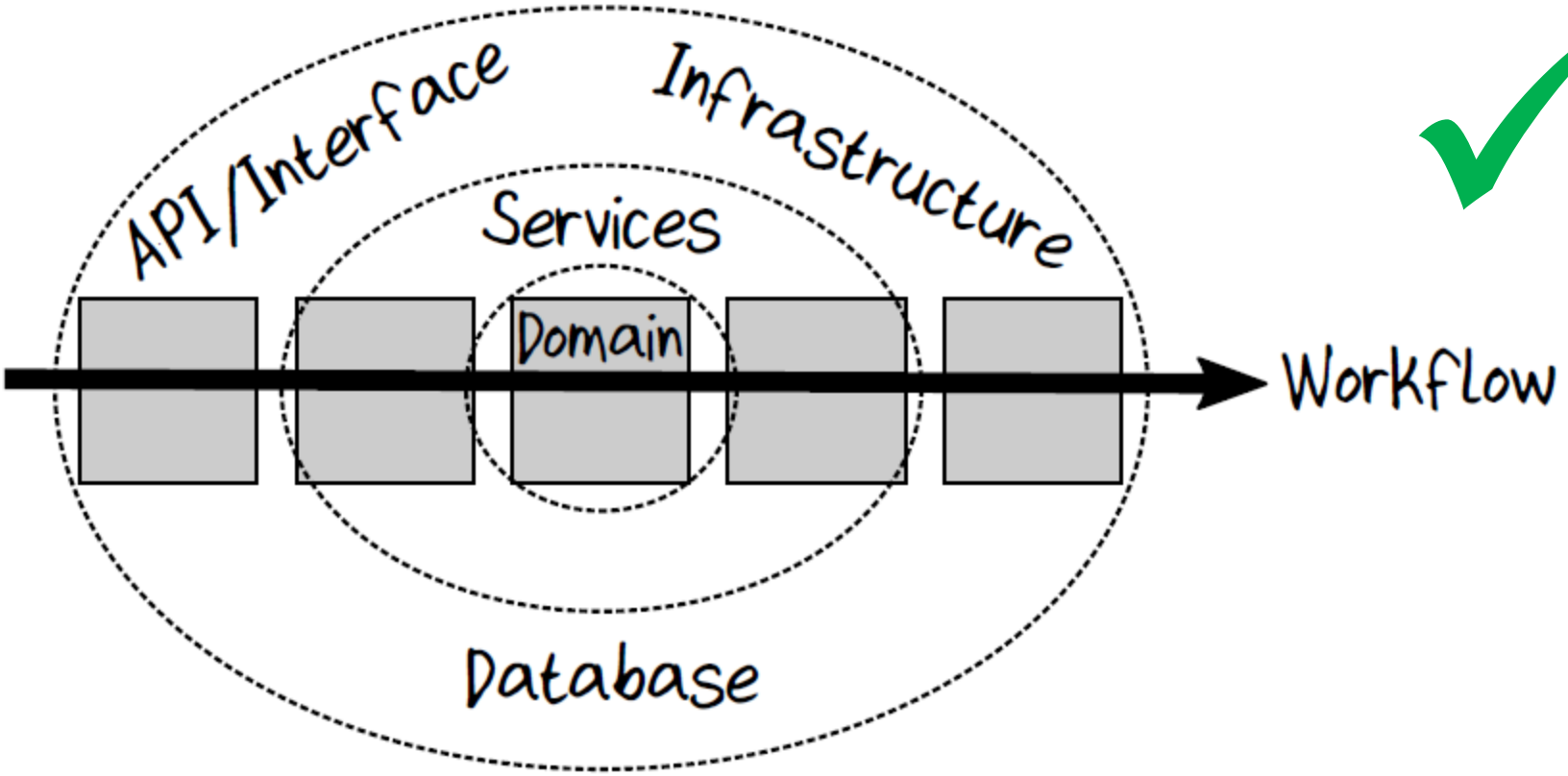
Each workflow contains all the code it needs to get its job done. When the requirements change for a workflow, only the code in that particular vertical slice needs to change.

Vertical slices stretched out



Confusing!

The "onion" architecture

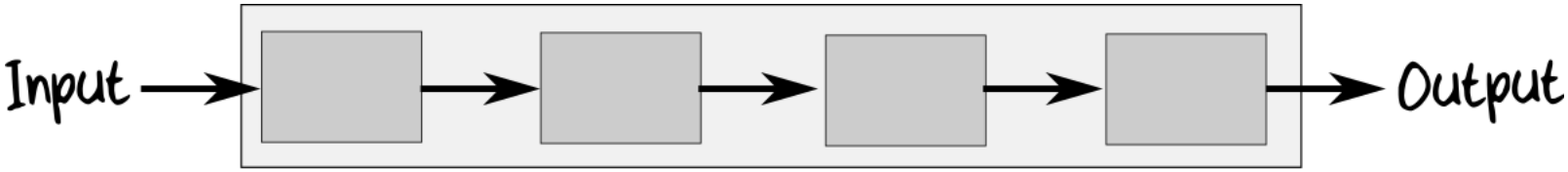


Core domain is pure, and all I/O is at the edges

See "Functional Core/Imperative Shell"

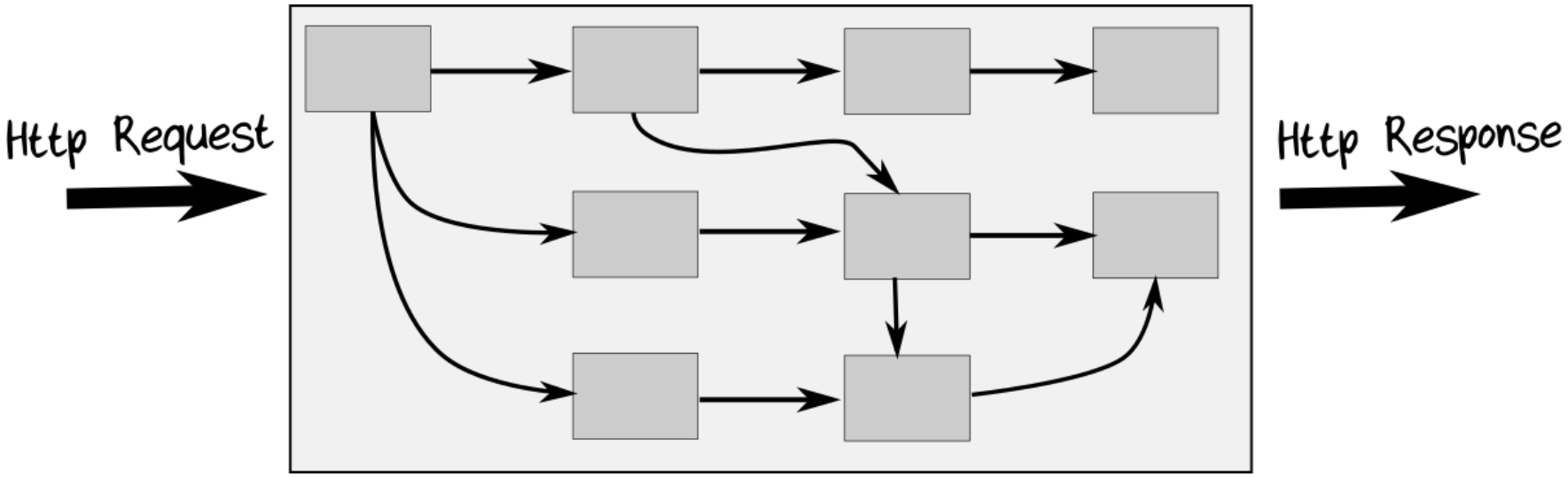
FP-style workflows are less complex than OO domain models.

FP workflow



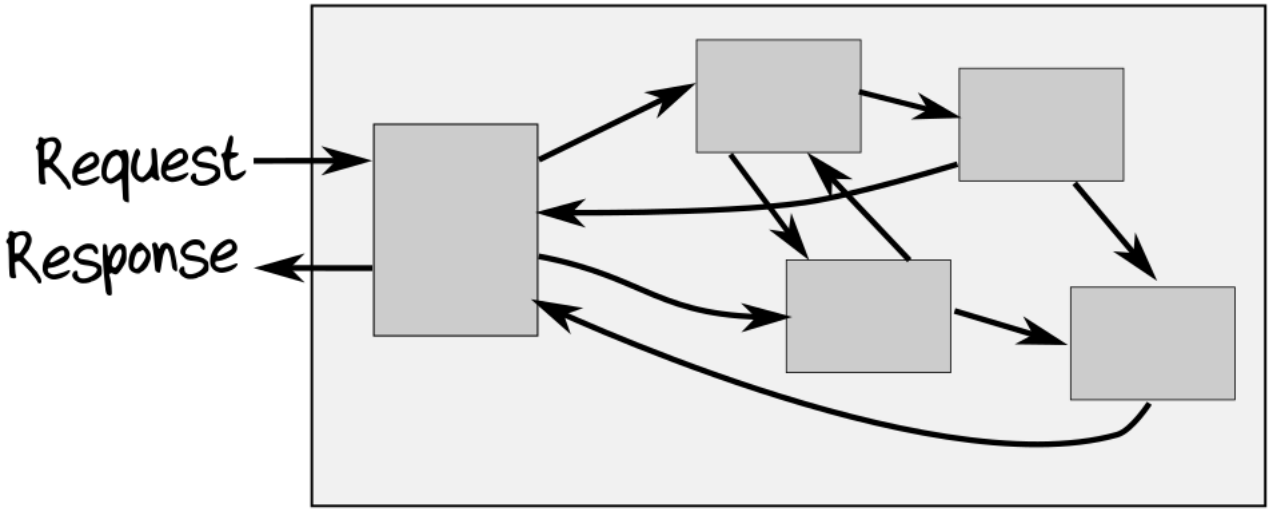
All arrows go left to right

Example: a web backend



One directional flow,
even with branching

Object Oriented workflow



Arrows go in all directions

We don't design microservices this way!

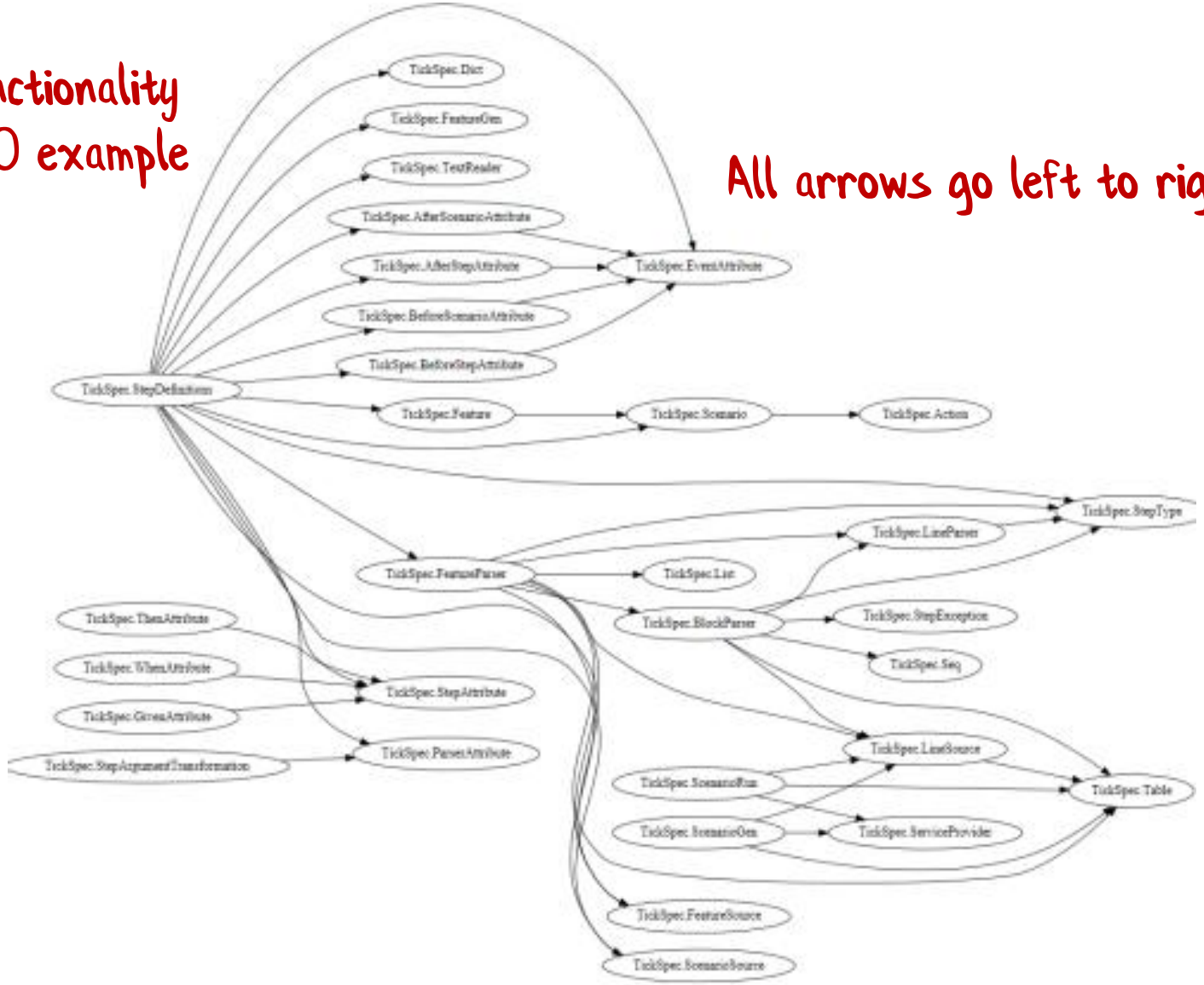
Real-world OO dependency graph

Multiple cyclic dependencies ☹

Real-world FP dependency graph

Same functionality as the OO example

All arrows go left to right ☺



FP-style workflows
are resistant to bloat

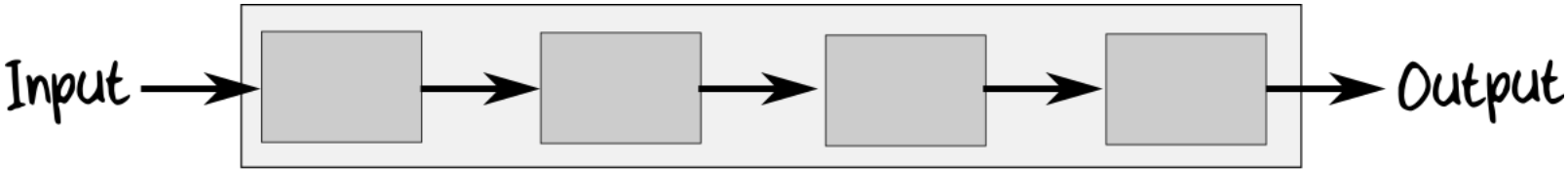
Some OO-style interfaces

```
interface IRepository {
        Insert
        InsertAsync
        Delete
        DeleteAsync
        Update
        UpdateAsync
        CommitChanges
        CommitChangesAsync
        Query
        QueryByKey
        QueryWithFilter
        QueryWithSpecification
        Contains
        Count
        QuerySummary
        QuerySummaryV2
        ChangePassword
        ChangePasswordV2
        DeleteAllRows
        LaunchMissiles
        LaunchMissilesV2
        }
```

One interface that supports *every* possible workflow!

Where's the Interface segregation principle?

FP workflow



Workflows only contain what they need.

Every part is relevant.
You get the ISP for free.

Libraries vs. Frameworks

- When you call into a **library**, you are in control.

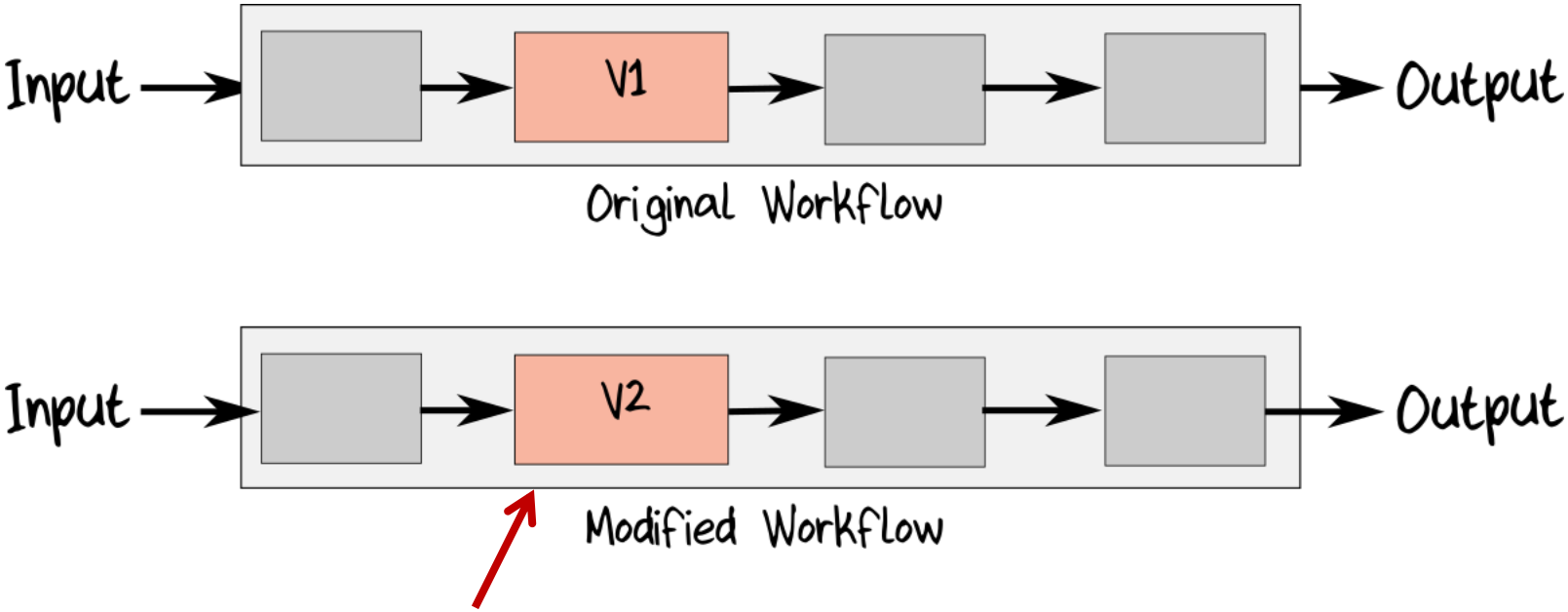
Composable. Only use what you need.

- With a **framework**, the control is inverted: the framework calls into you.

Not composable.
Often forced to implement
things you don't need.

FP workflows can be modified with confidence

Modifying a workflow

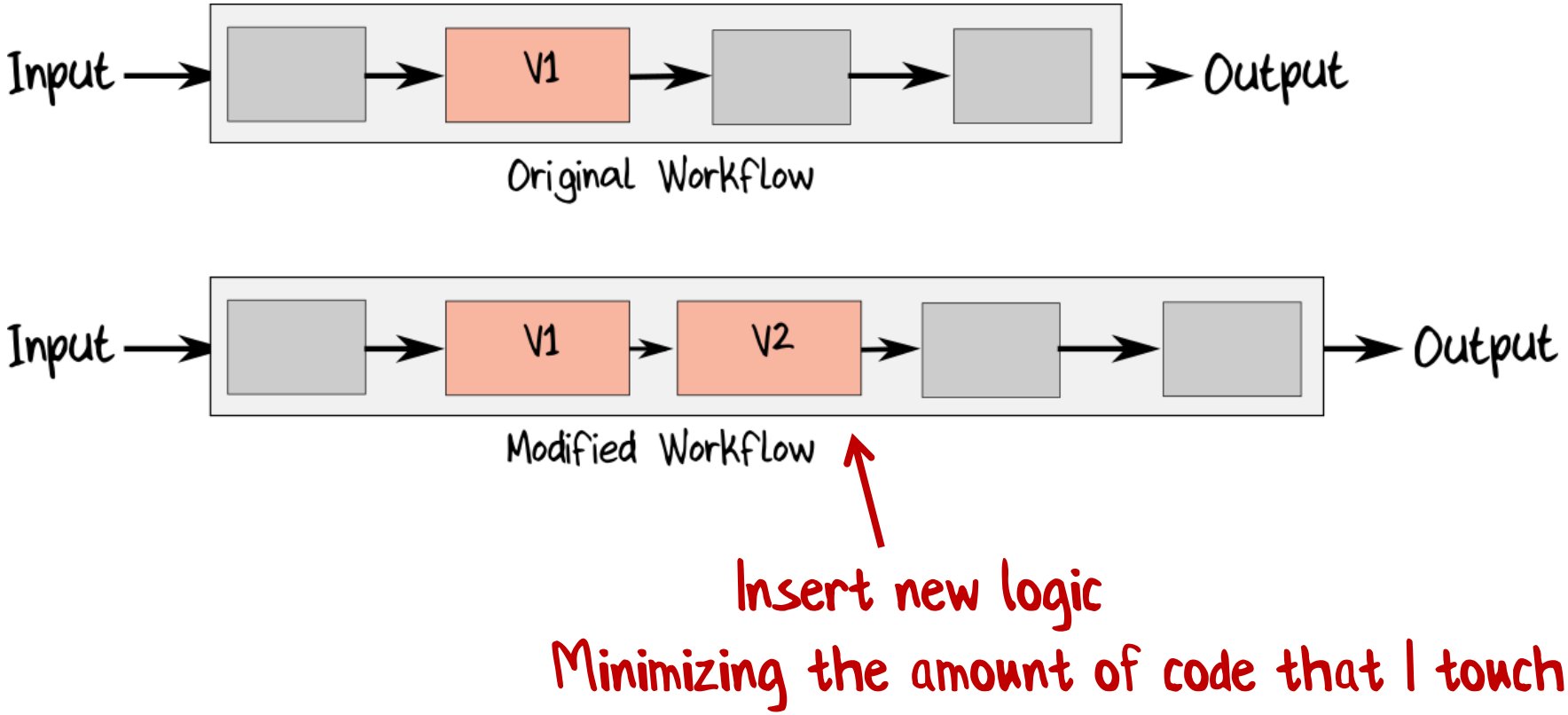


Replace a component

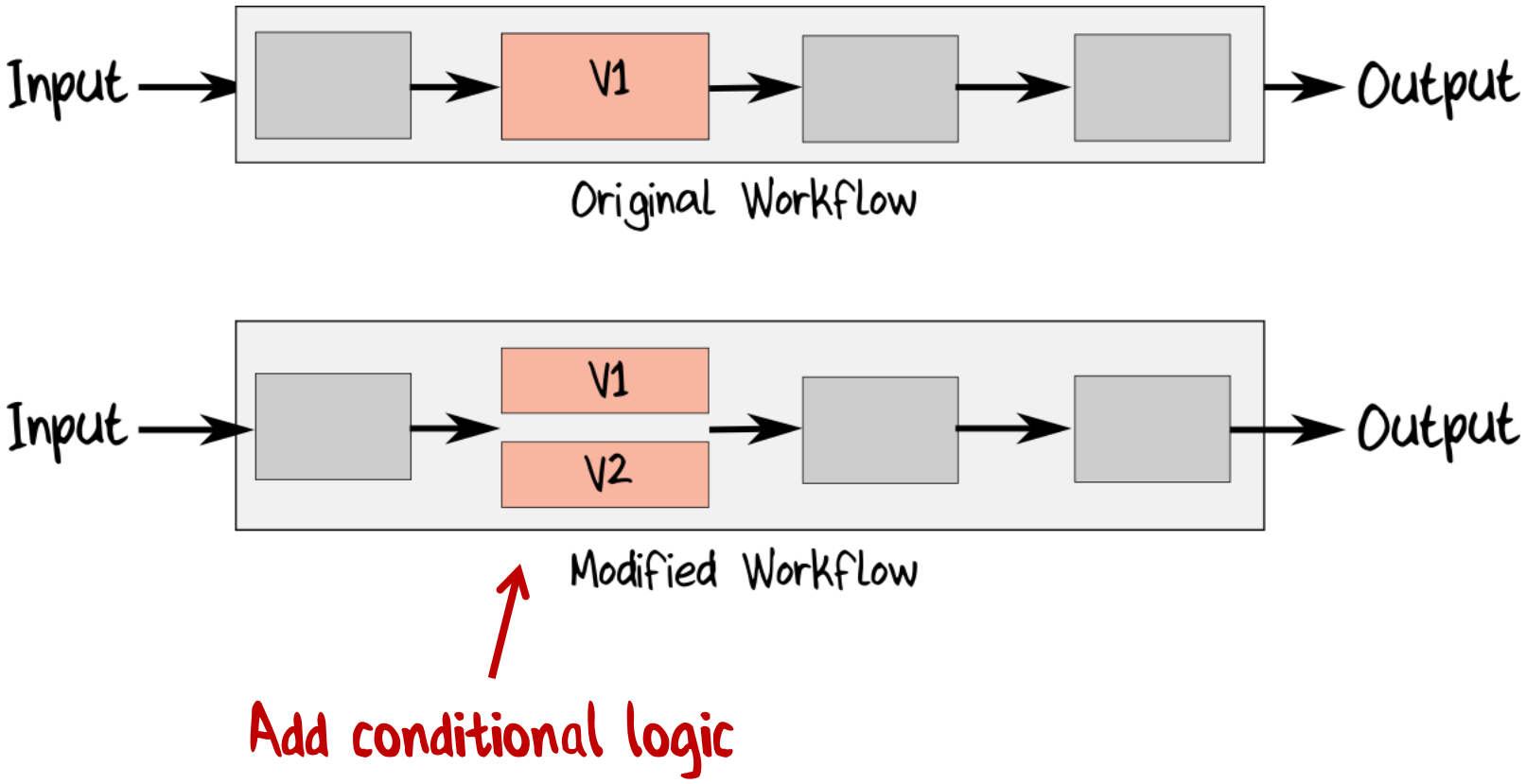
Minimizing the amount of code that I touch

Static type checking ensures that sub-components are used correctly

Add features to a workflow

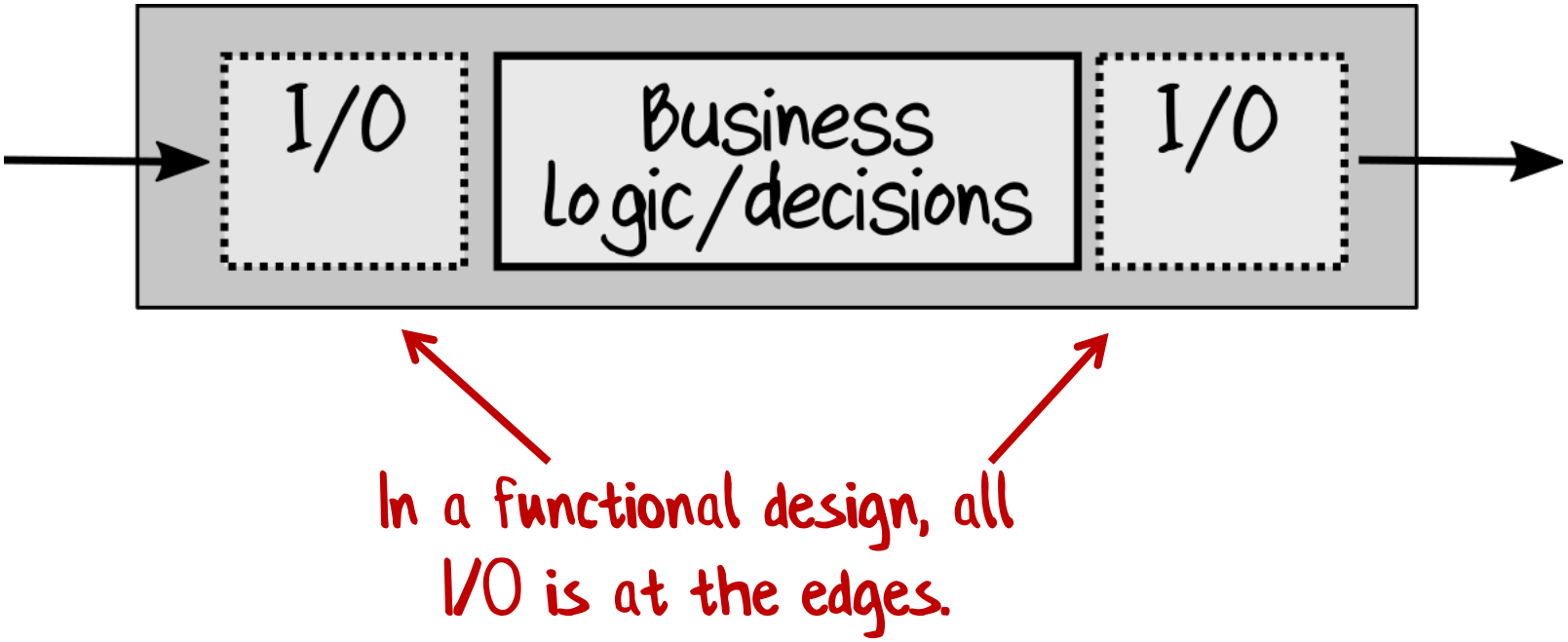


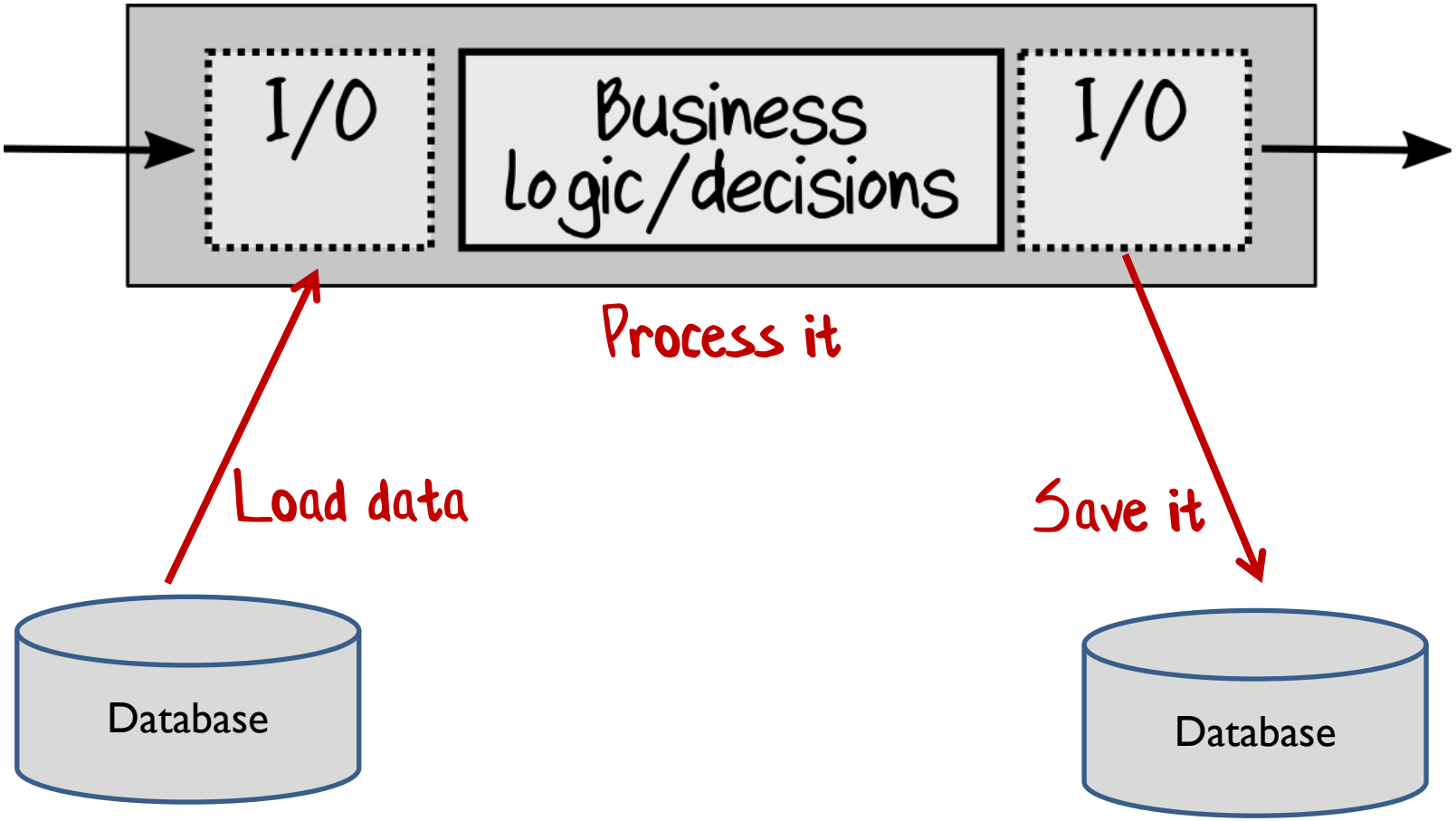
Add branching to a workflow



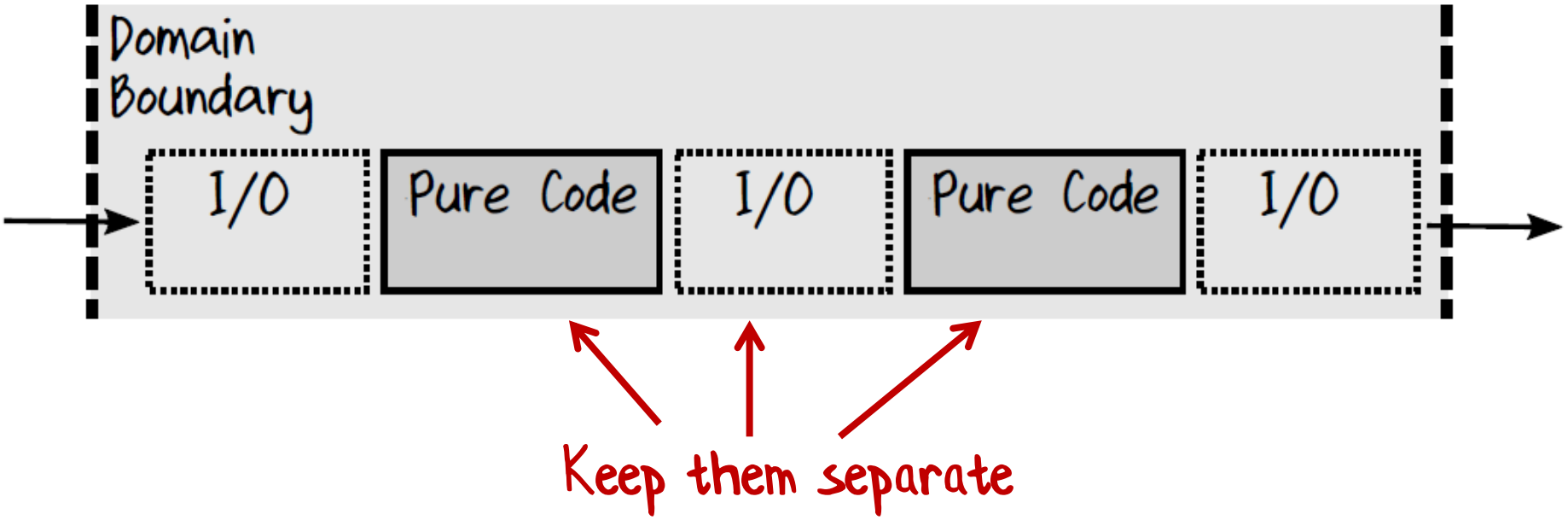
I/O at the edges

A FP-style workflow




I/O
Business logic/decisions
I/O
Process it
Load data
Save it
Database
Database

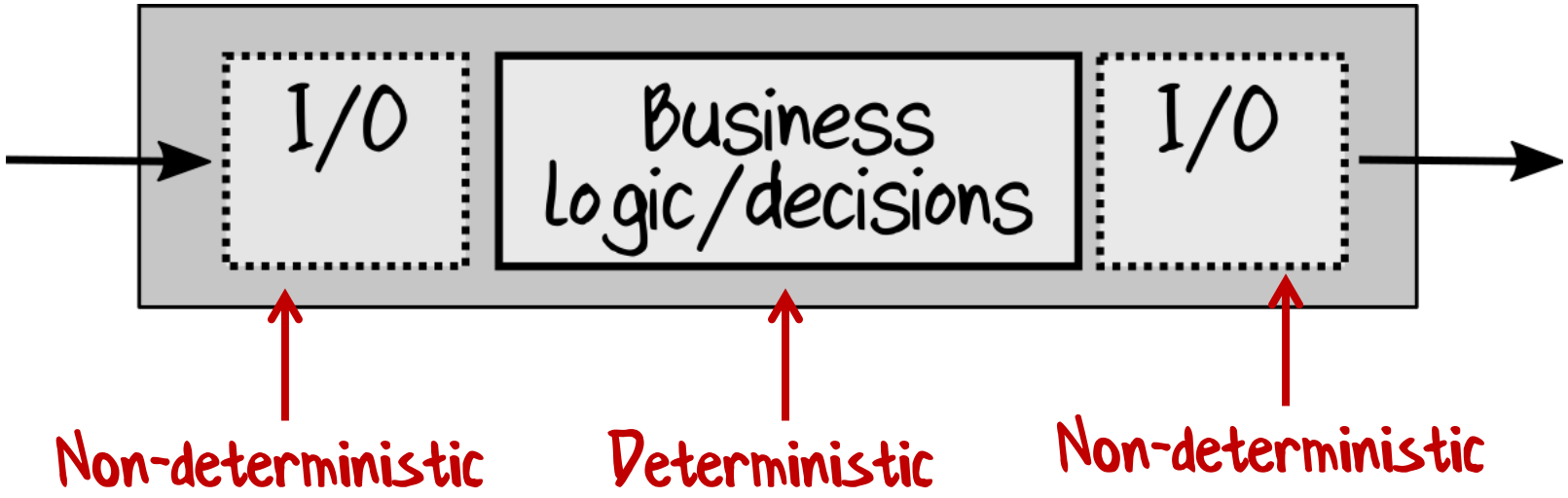
I/O in the middle of a workflow



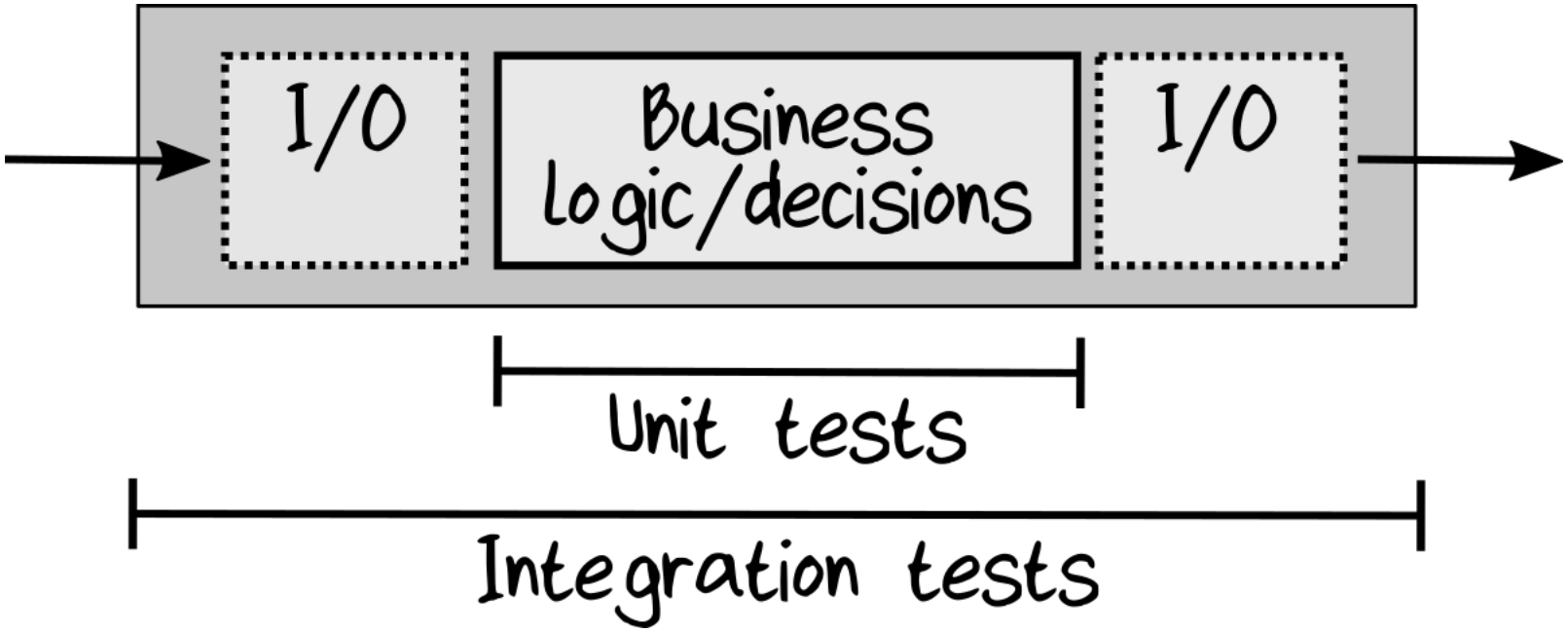
Testing workflows

Review of Key Testing Concepts

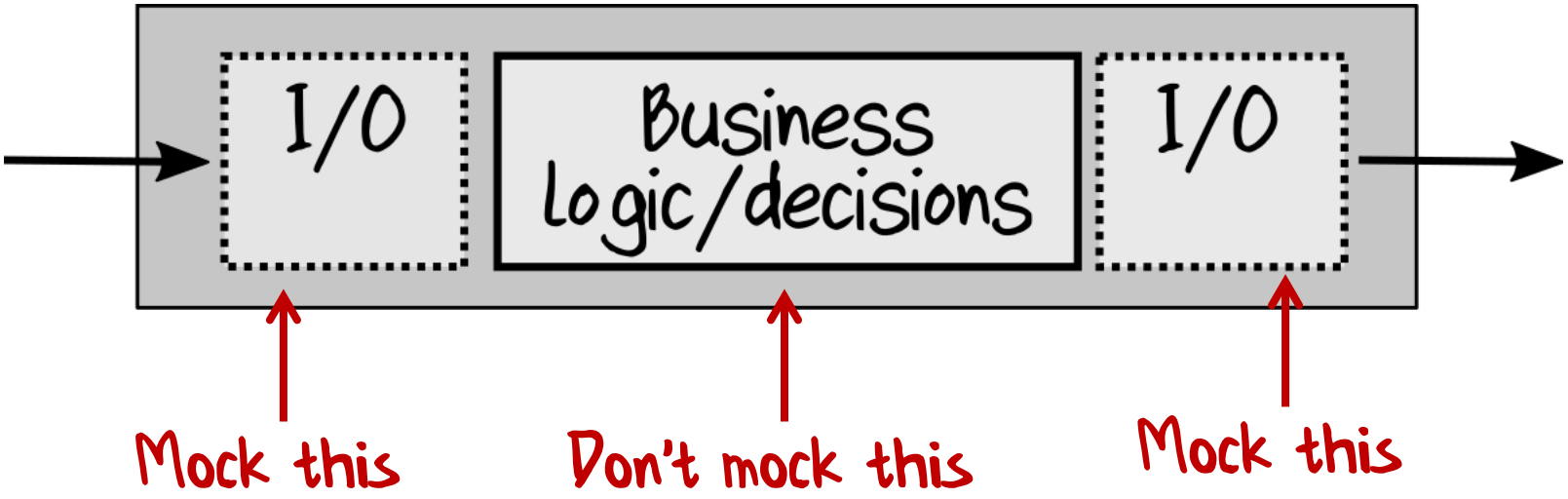
- The SUT (System Under Test) should be a unit of business value
  - Test workflows, not classes!
- Tests should apply to the boundaries of a system not its internals
  - Tests should be done at the workflow level
- A "Unit" test means the test is isolated
  - It produces no side effects and can be run in isolation. No I/O!
  - A "unit" is not a class!


I/O
Business logic/decisions
I/O
Non-deterministic
Deterministic
Non-deterministic

Where are the test boundaries?

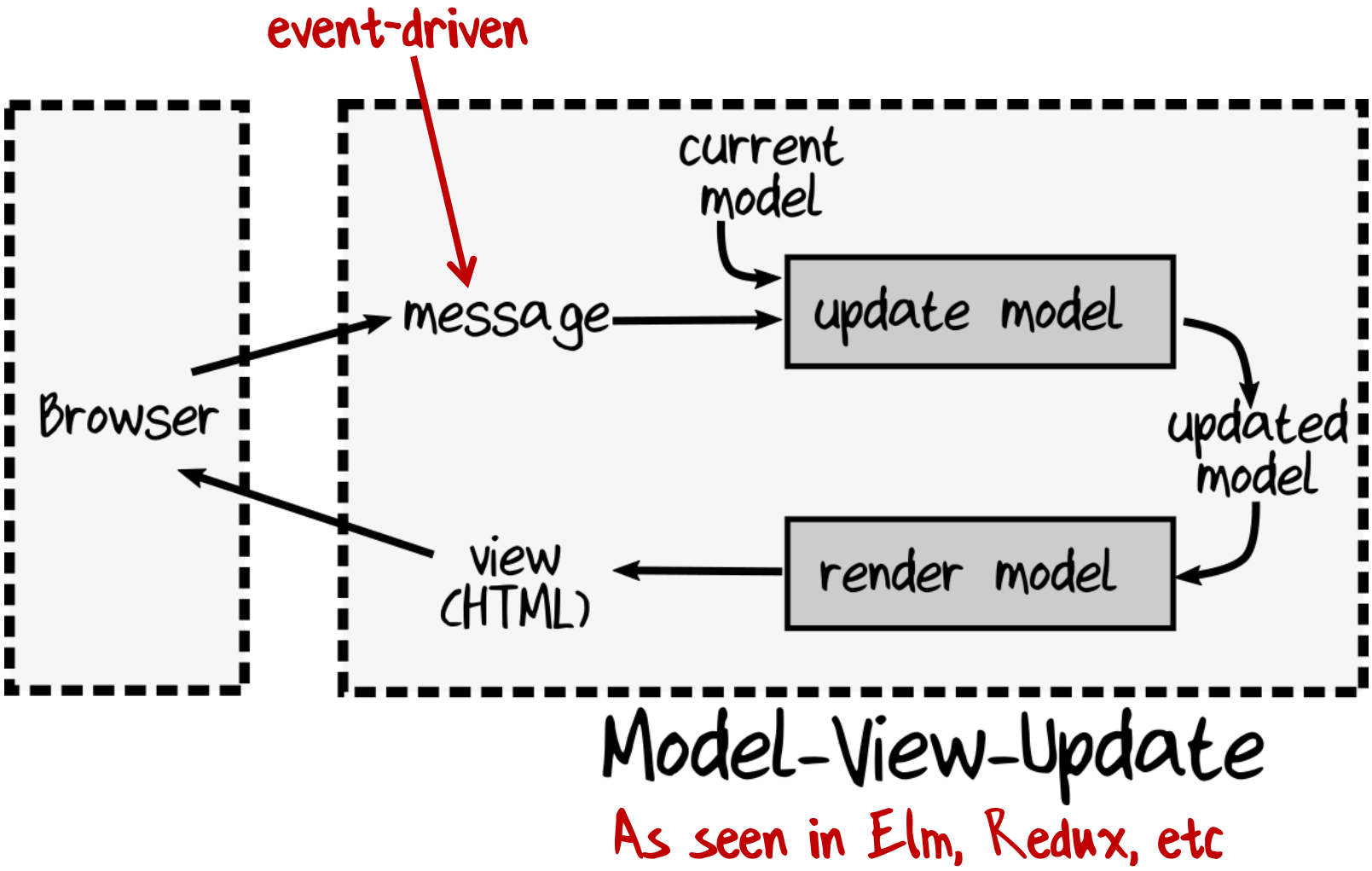


What to mock?

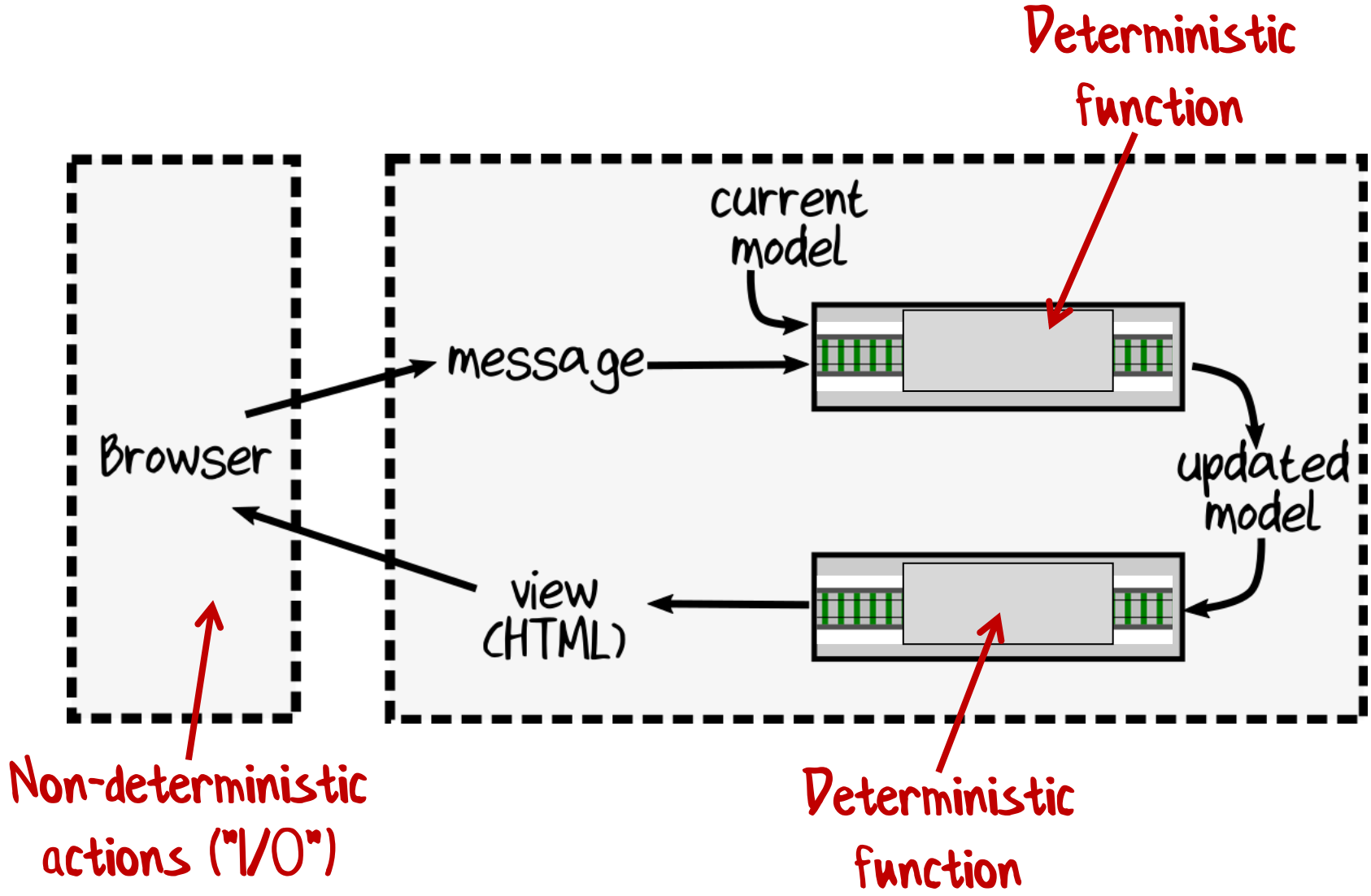

I/O
Business logic/decisions
I/O
Mock this
Don't mock this
Mock this

FP workflows work
on the front end too!

MVU is a FP style approach



MVU is a FP style approach



End