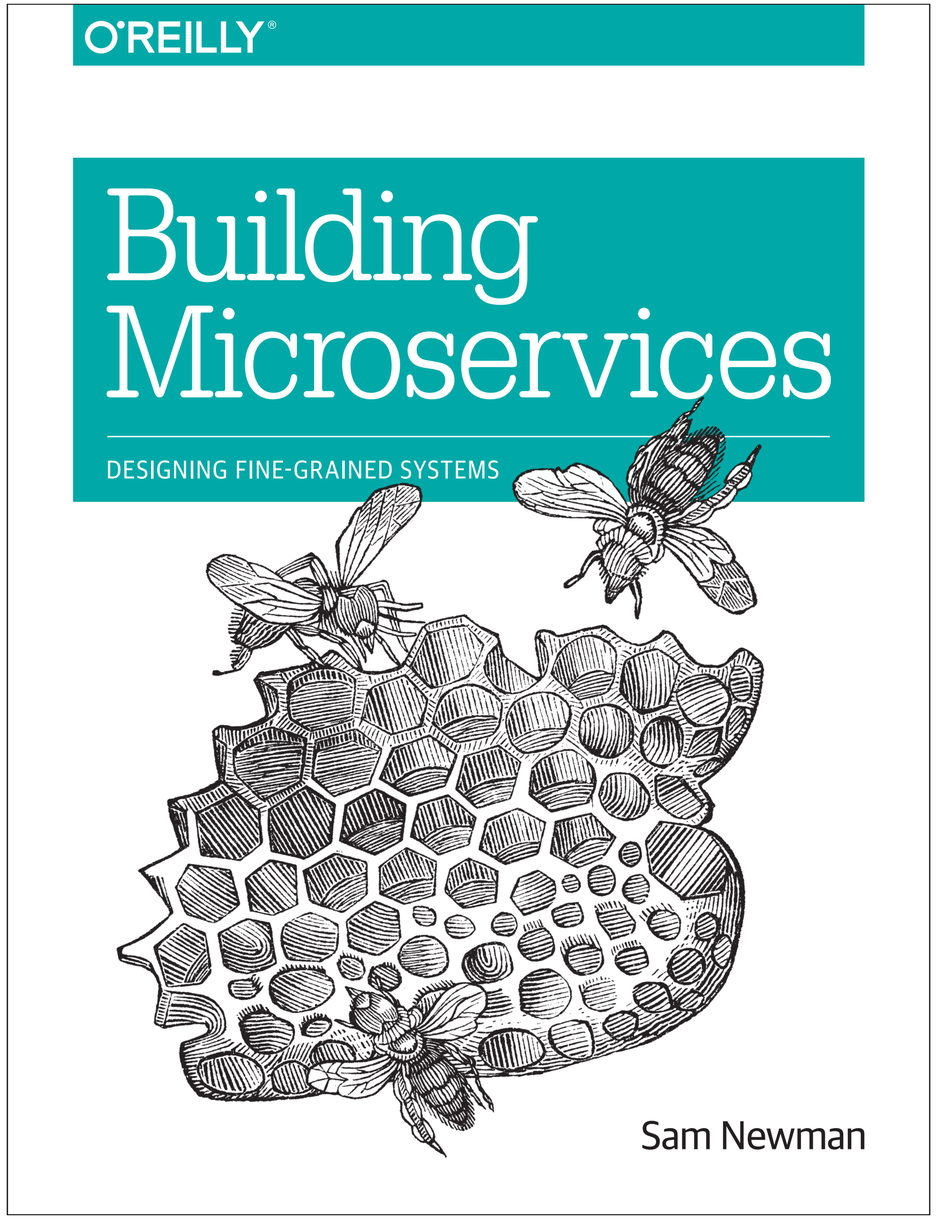
MICROSERVICE DECOMPOSITION PATTERNS

Sam Newman

https://www.flickr.com/photos/spongebabyalwaysfull/8793058944/



Sam Newman
& Associates

NEW BOOK!

Monolith To Microservices.

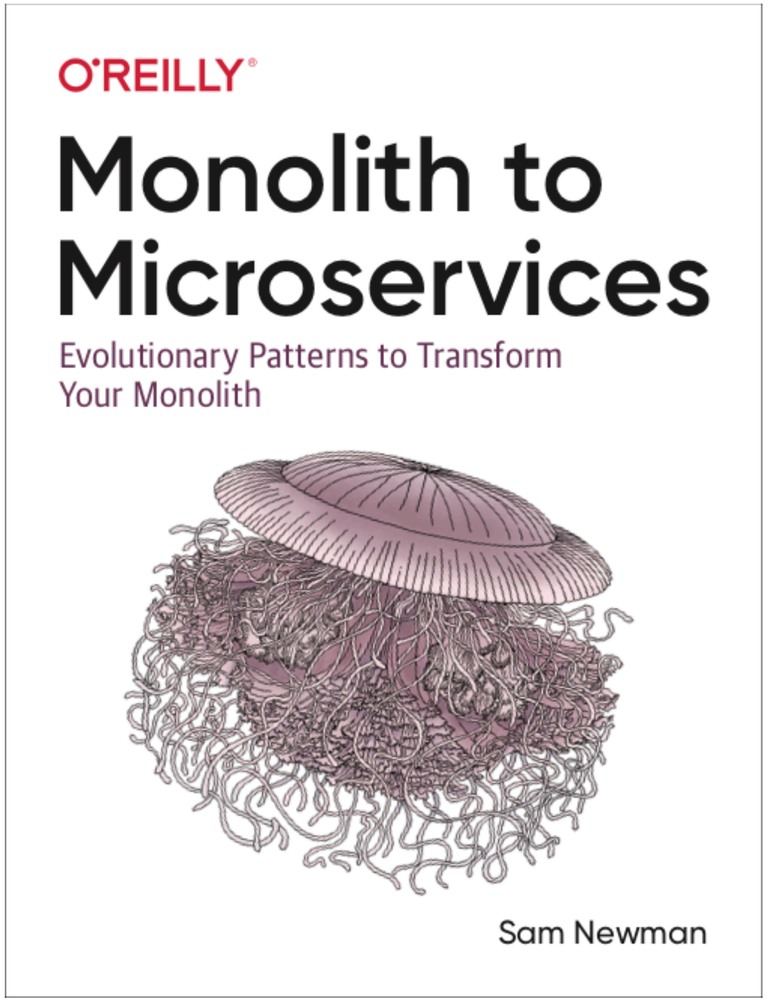
Monolith To Microservices is a forthcoming book on system decomposition from O'Reilly

How do you detangle a monolithic system and migrate it to a microservices architecture? How do you do it while maintaining business-as-usual? As a companion to Building Microservices, this new book details a multiple approaches for helping you transition from existing monolthic systems to microservice architectures. This book is ideal if you're looking to evolve your existing systems, rather than just rewriting everything from scratch.

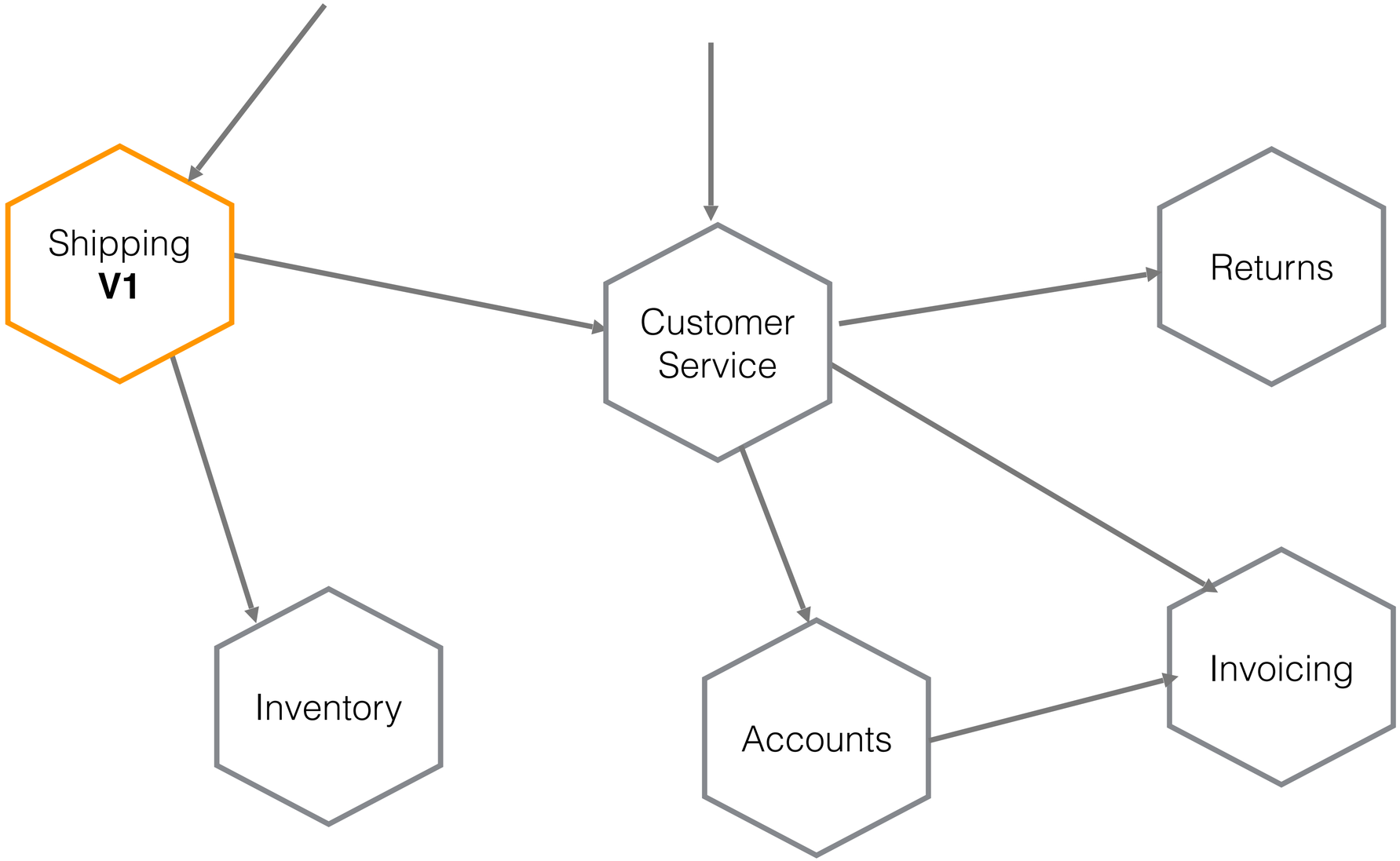
Topics include:

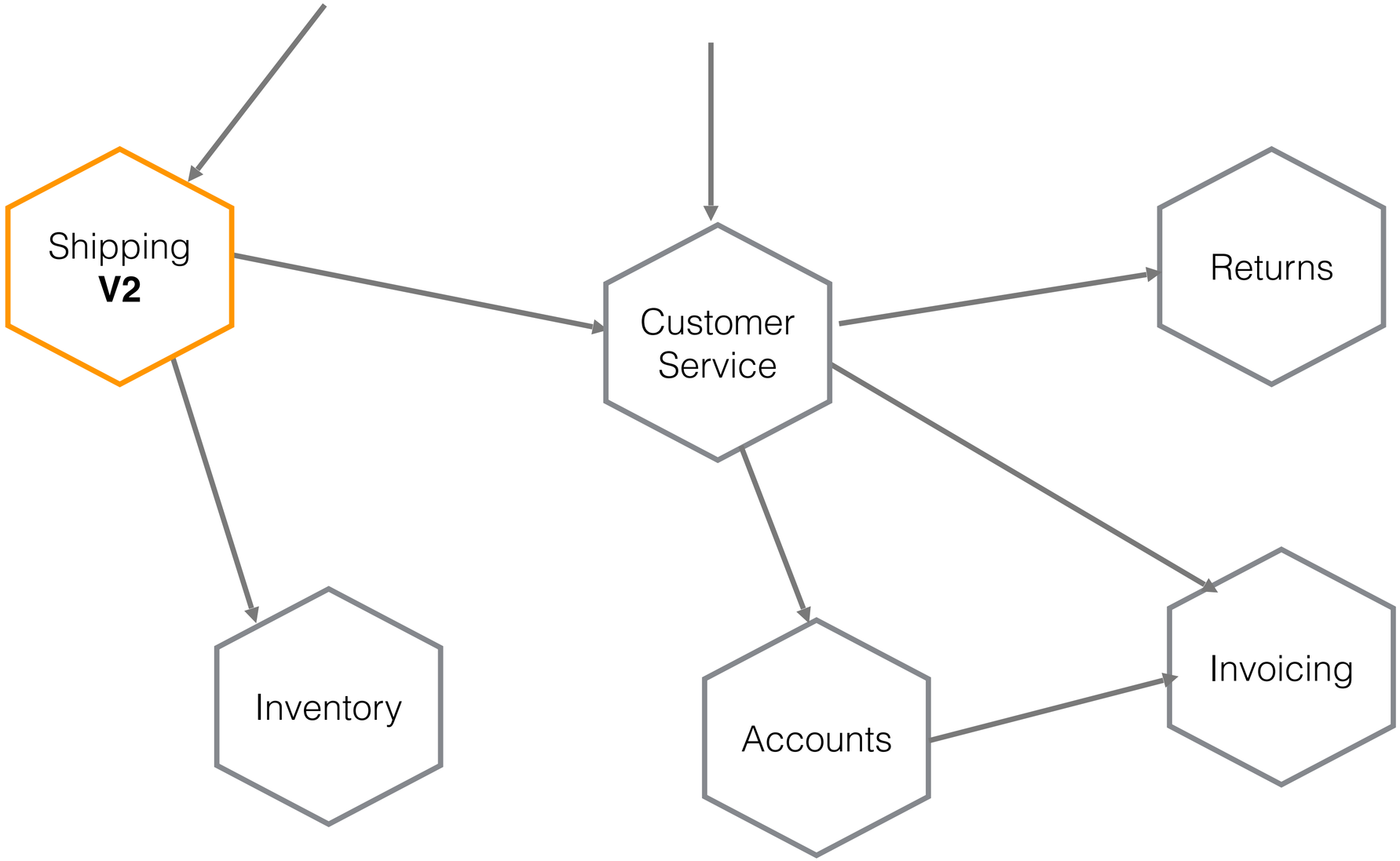
- Should you migrate to microservices, and if you should, how do you prioritise where to start
- How do you incrementally decompose an application
- Discusses multiple migration patterns and where they apply
- Delves into details of database decomposition, including the impact of breaking referential and transactional integrity, new failure modes, and more
- The growing pains you'll experience as your microservice architecture grows

Read The Early Access Version!

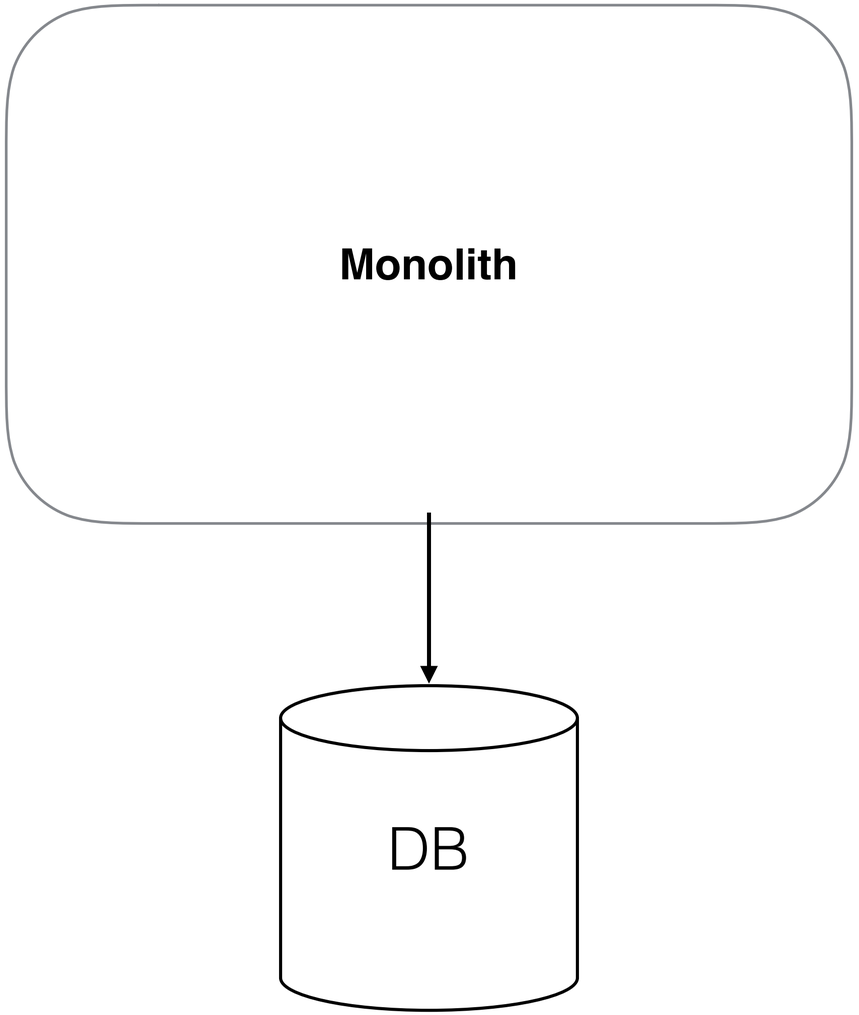


https://samnewman.io/books/monolith-to-microservices/


Shipping
V1
Customer
Service
Returns
Inventory
Accounts
Invoicing

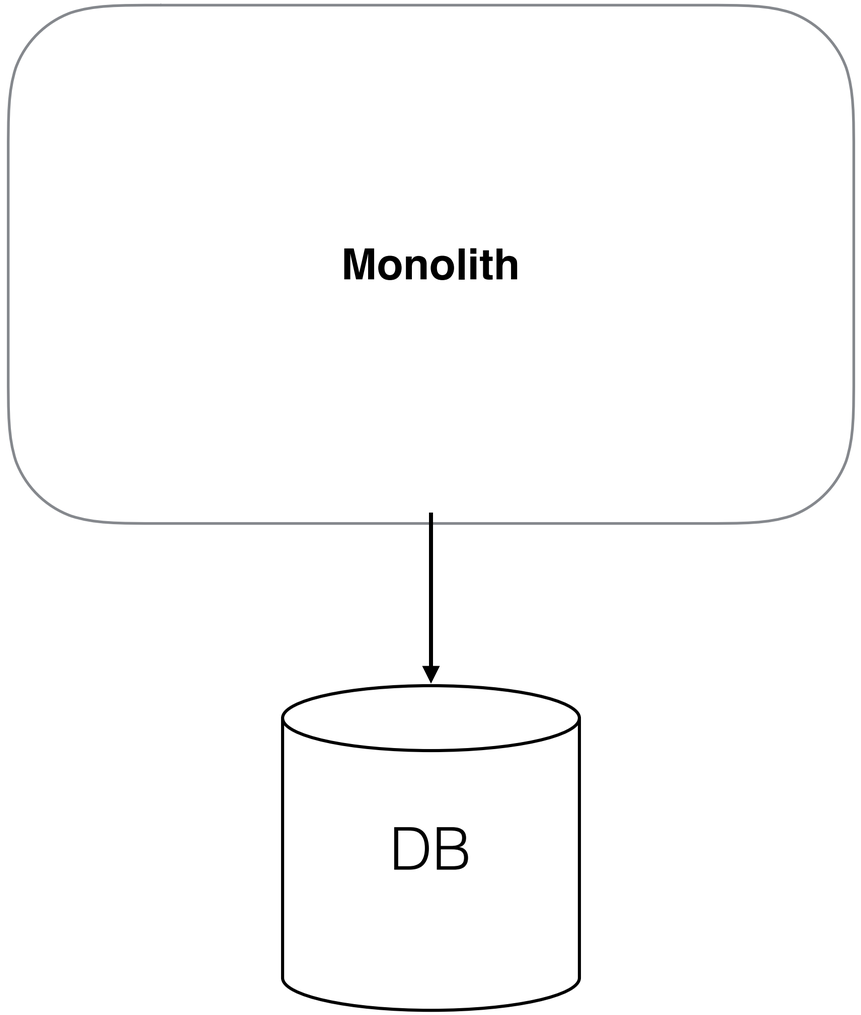

Shipping
V2
Customer
Service
Returns
Inventory
Accounts
Invoicing

SINGLE PROCESS MONOLITH



SINGLE PROCESS MONOLITH

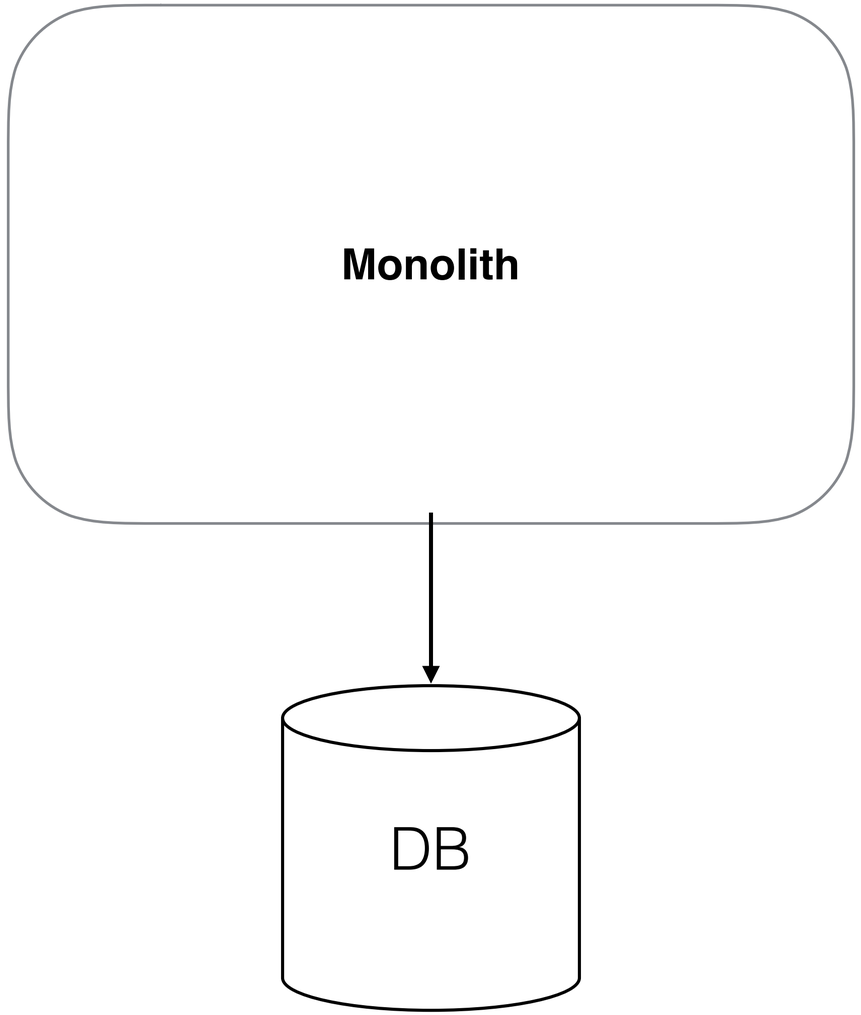
All code packaged into a single process



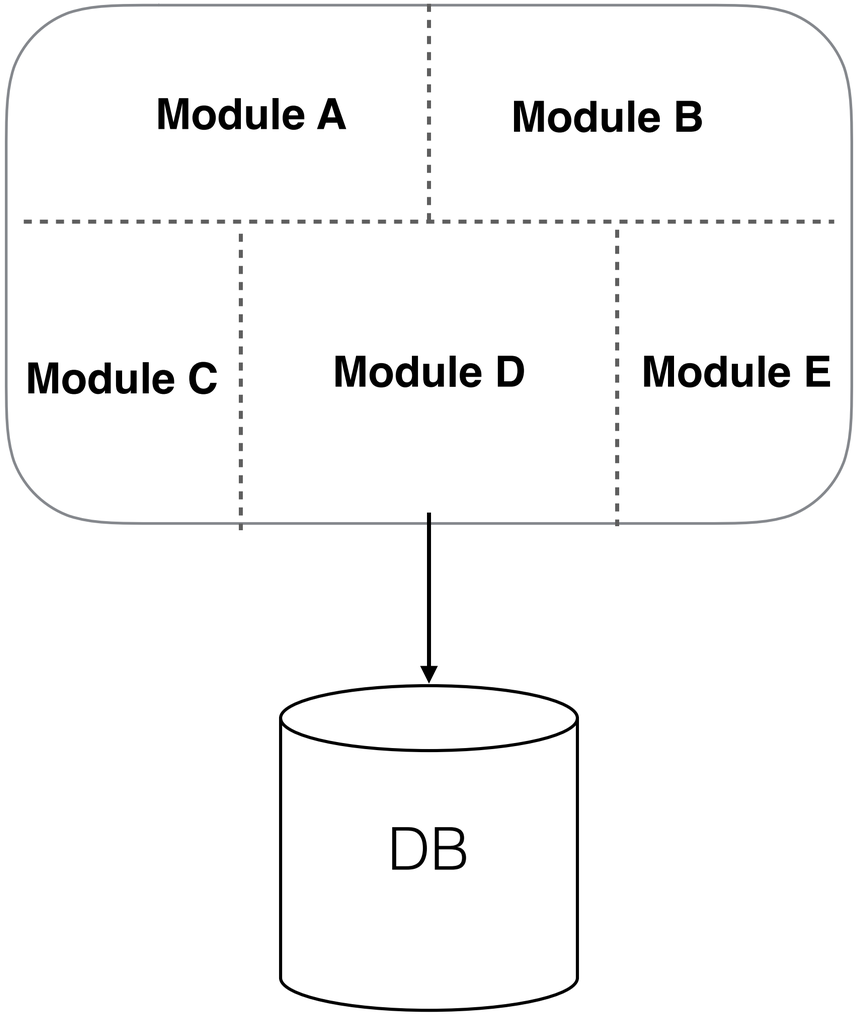
SINGLE PROCESS MONOLITH

All code packaged into a single process

All data stored in a single database

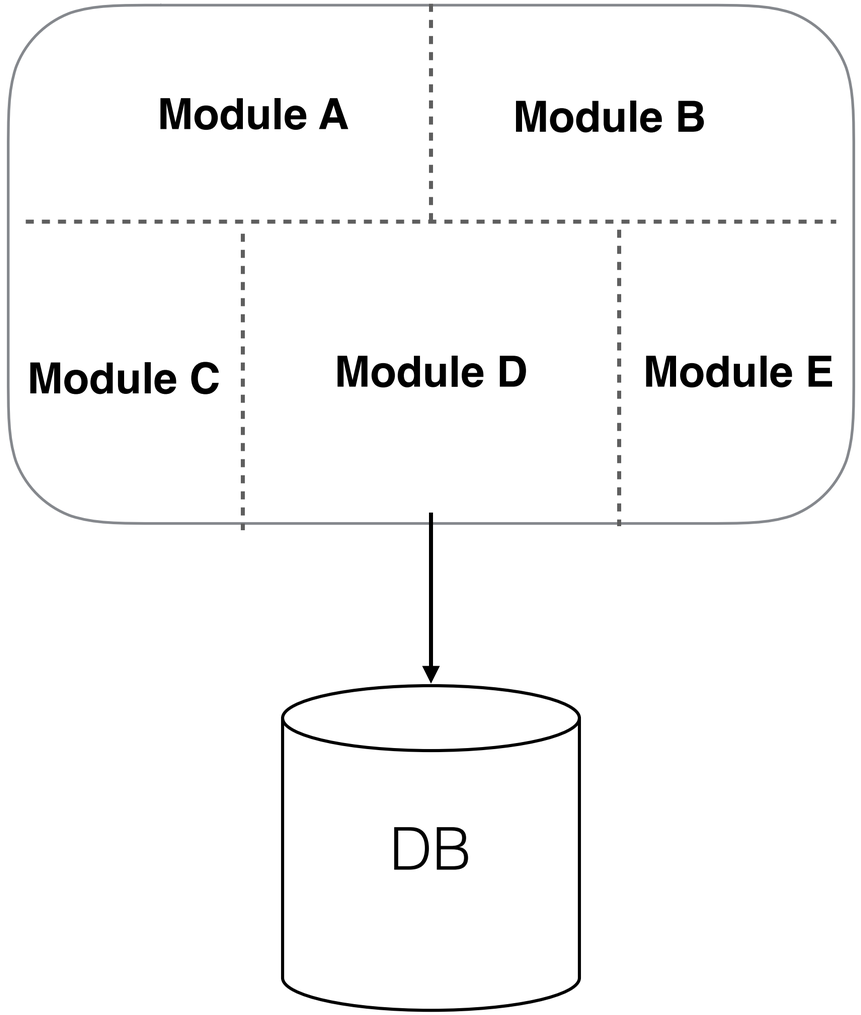


“MODULAR” MONOLITH



“MODULAR” MONOLITH

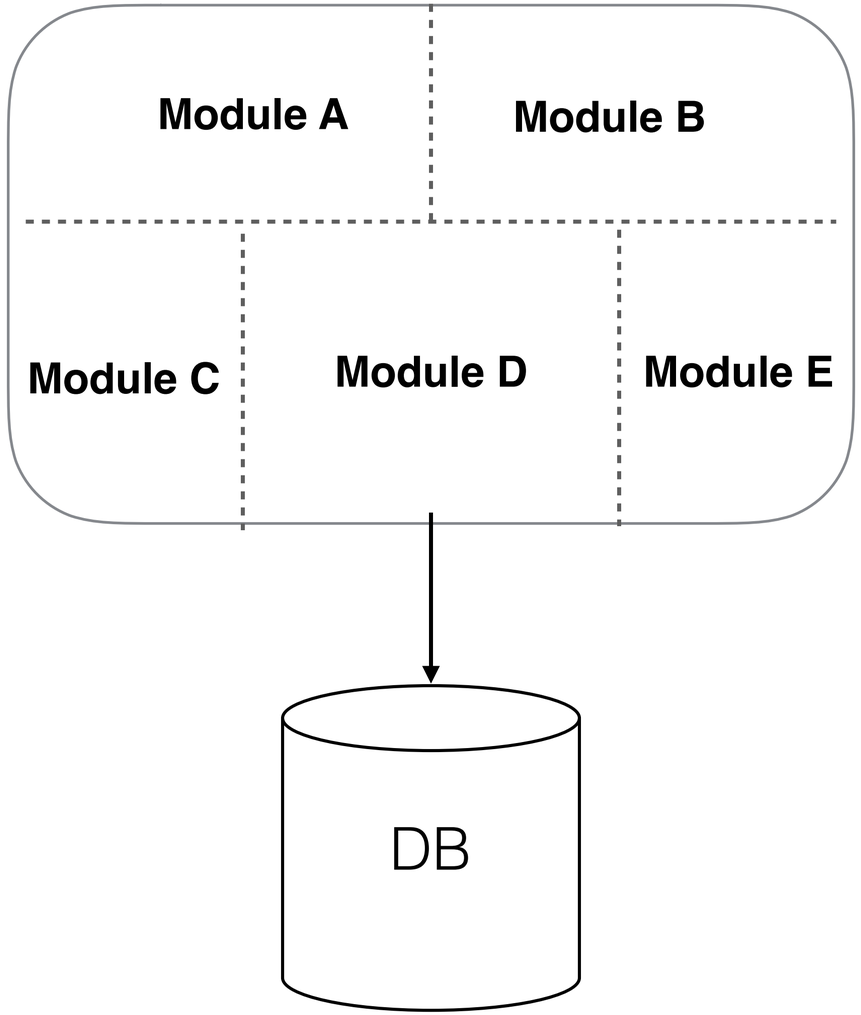
The code is broken into modules



“MODULAR” MONOLITH

The code is broken into modules

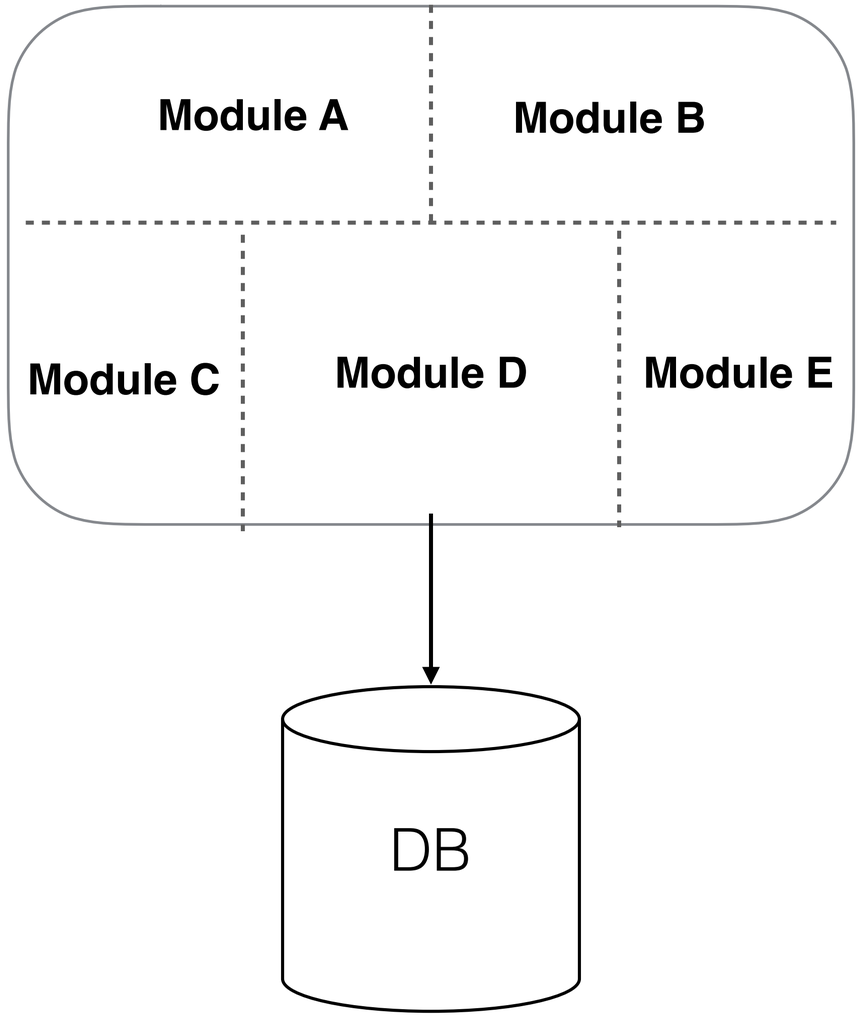
Each module packaged together into a single process



“MODULAR” MONOLITH

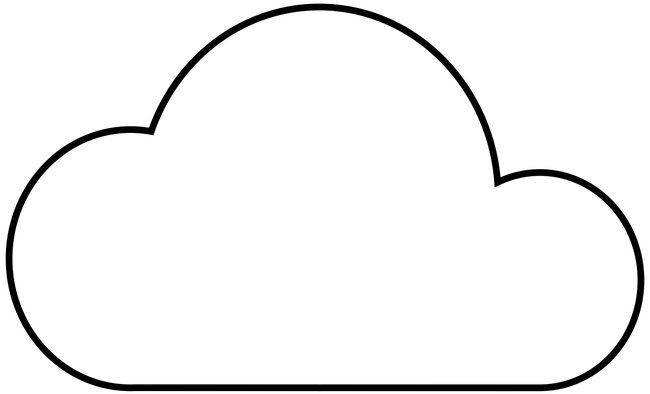
The code is broken into modules

Each module packaged together into a single process



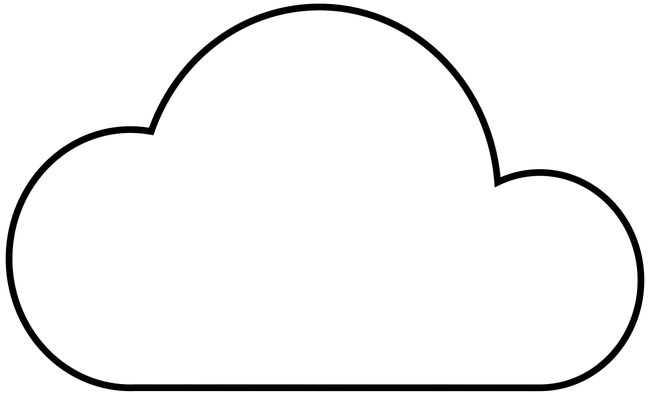
Highly underrated option

3RD PARTY MONOLITH



3RD PARTY MONOLITH

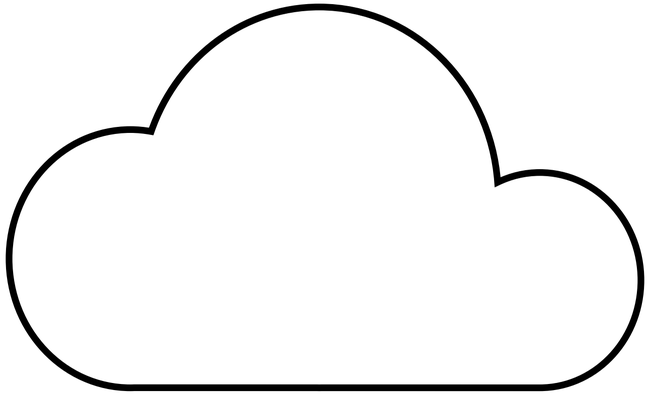
Could be on-prem software, or a SAAS product



3RD PARTY MONOLITH

Could be on-prem software, or a SAAS product

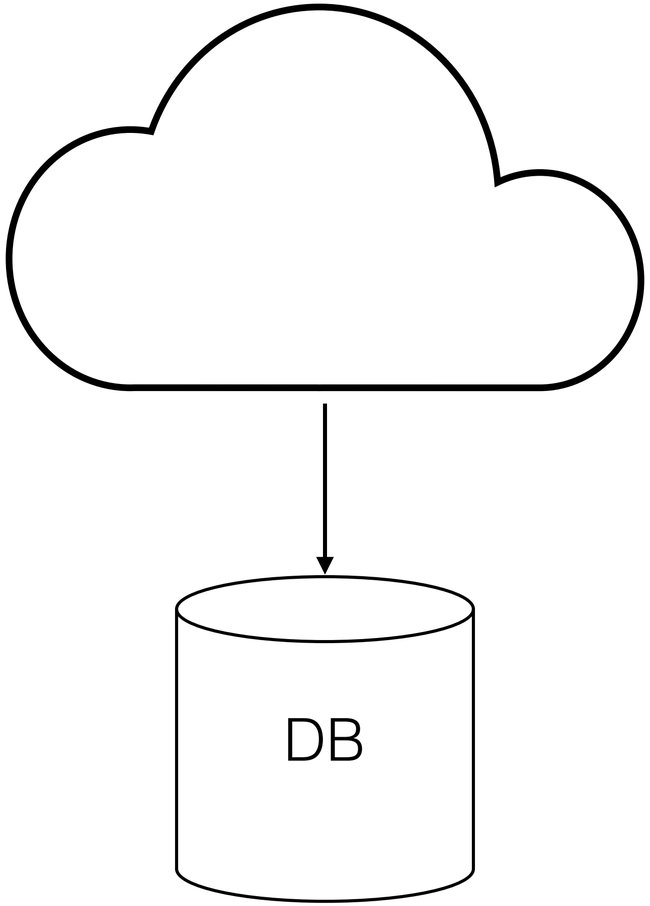
You have limited to no ability to change the core system



3RD PARTY MONOLITH

Could be on-prem software, or a SAAS product

You have limited to no ability to change the core system

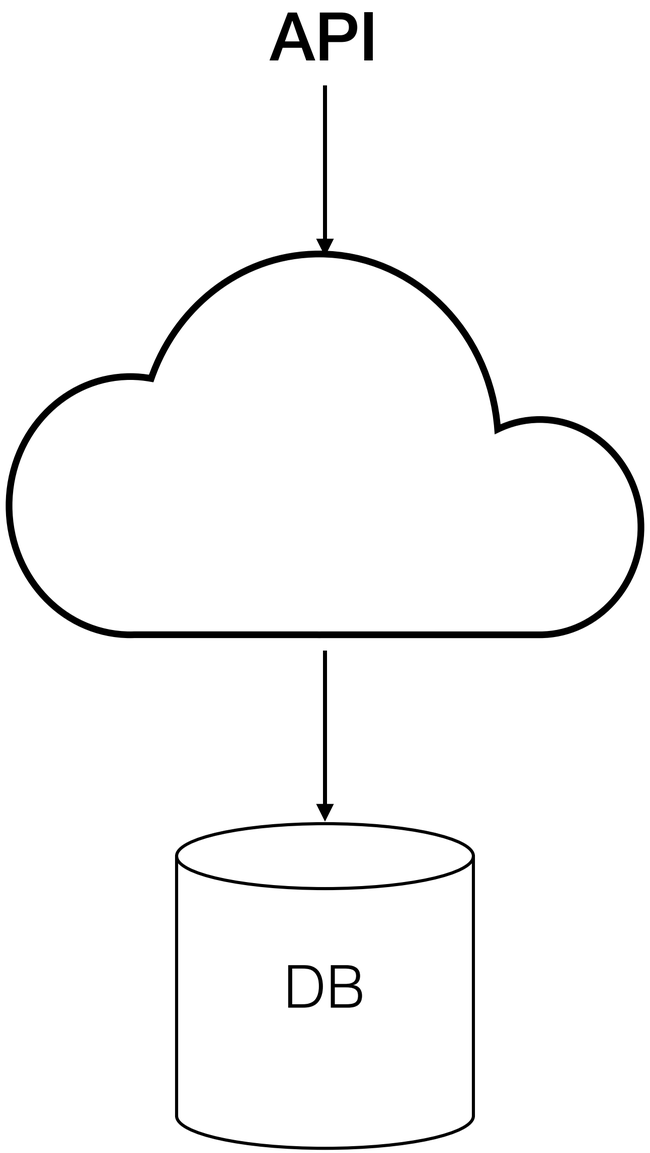


You *might* have access to underlying storage…

3RD PARTY MONOLITH

Could be on-prem software, or a SAAS product

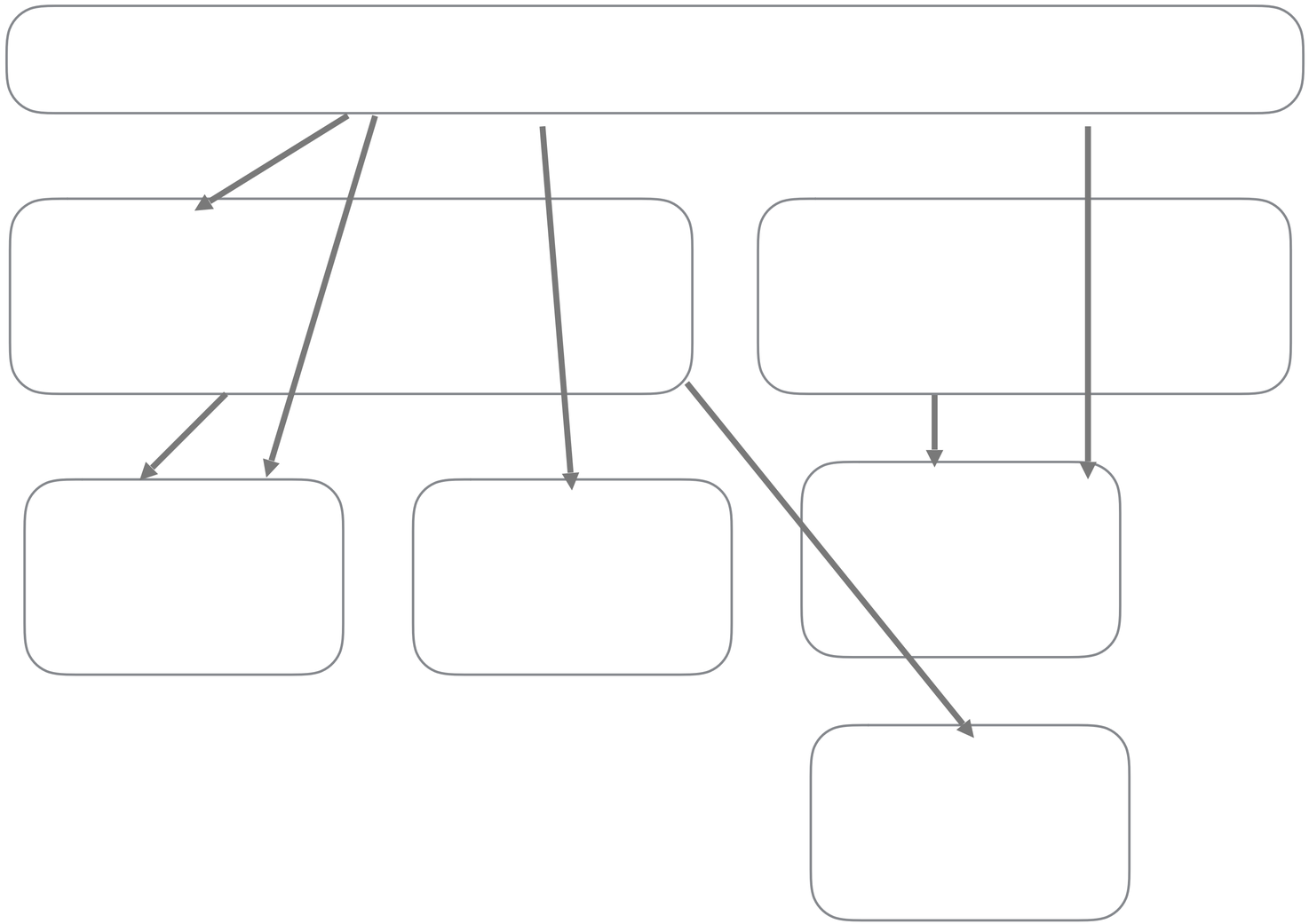
You have limited to no ability to change the core system



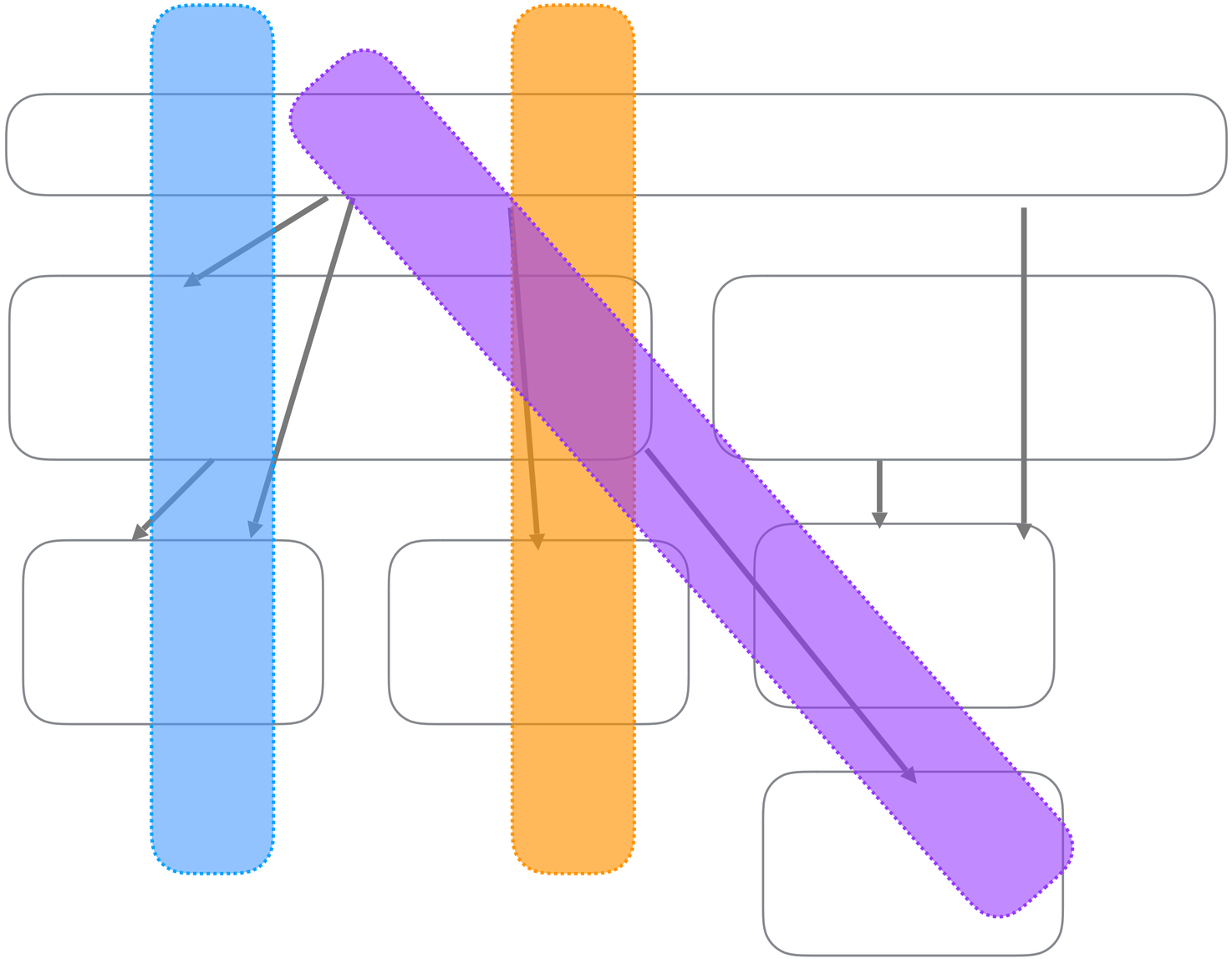
You *might* have access to underlying storage…

…or perhaps APIs

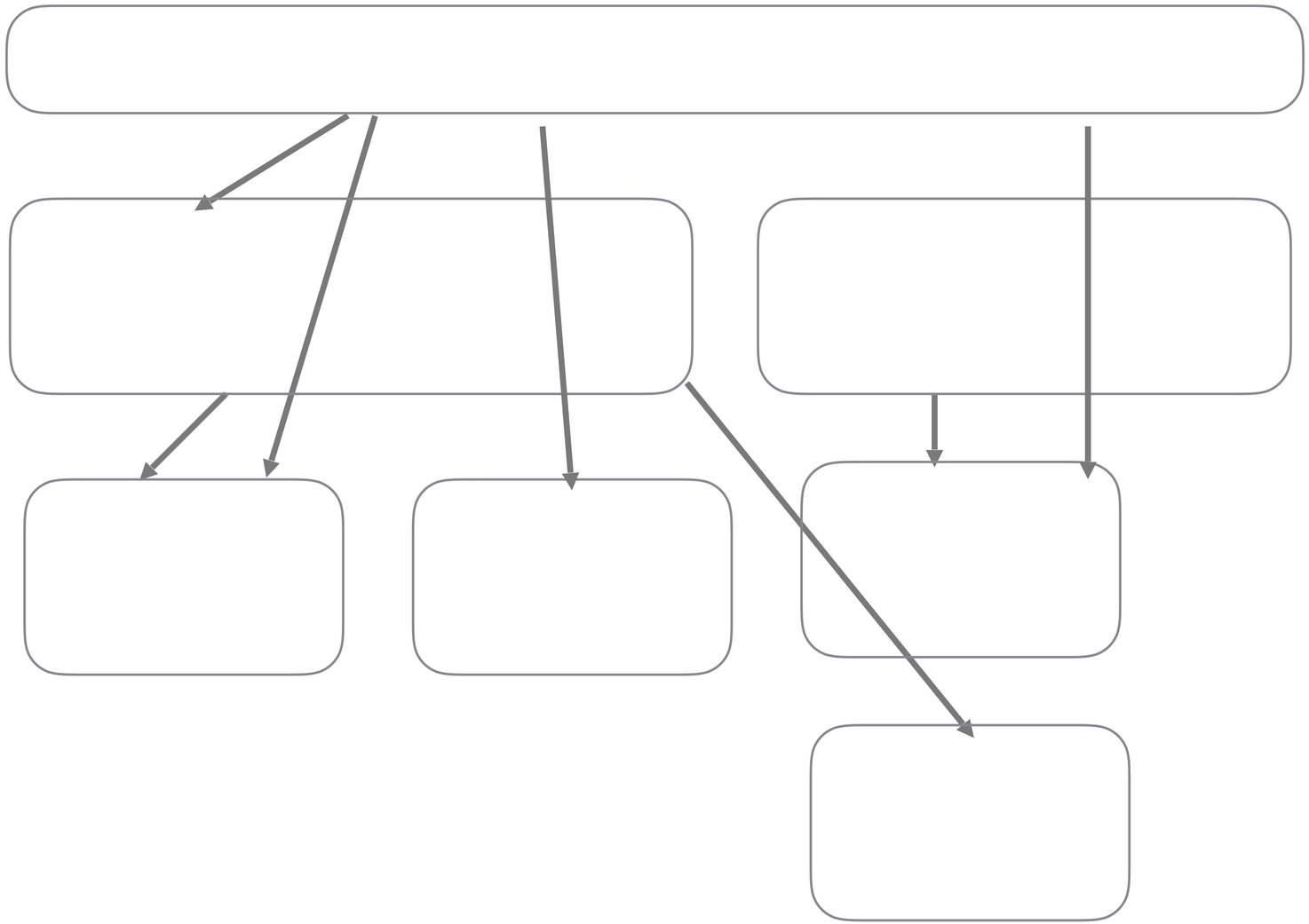
DISTRIBUTED MONOLITH



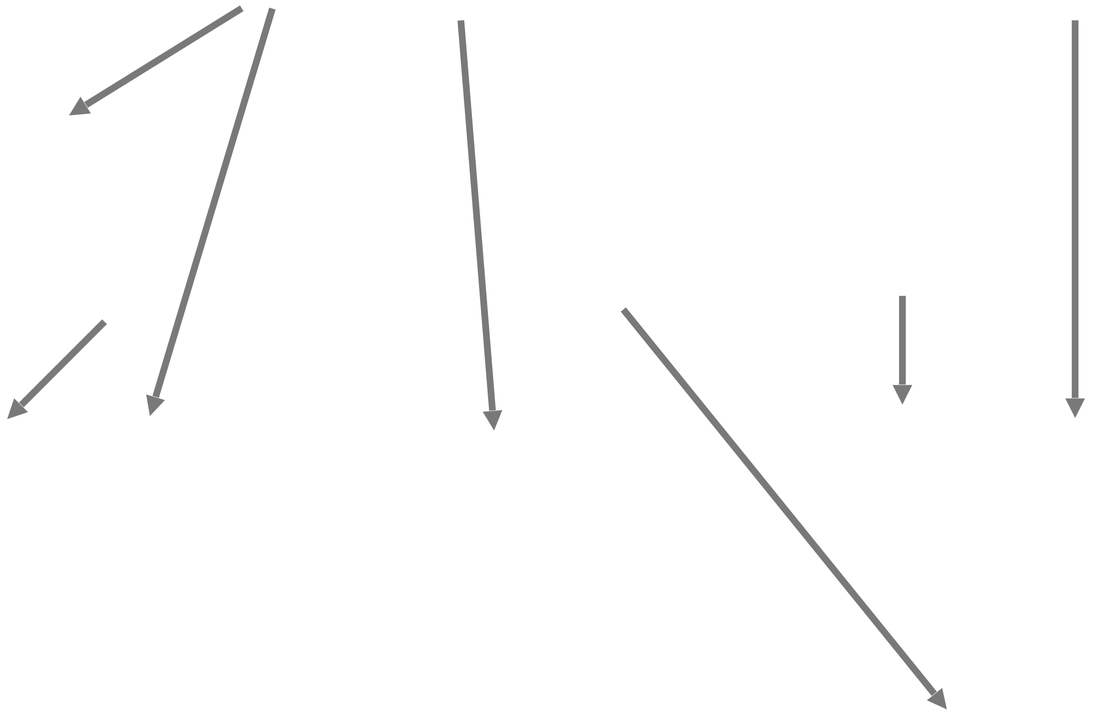
DISTRIBUTED MONOLITH



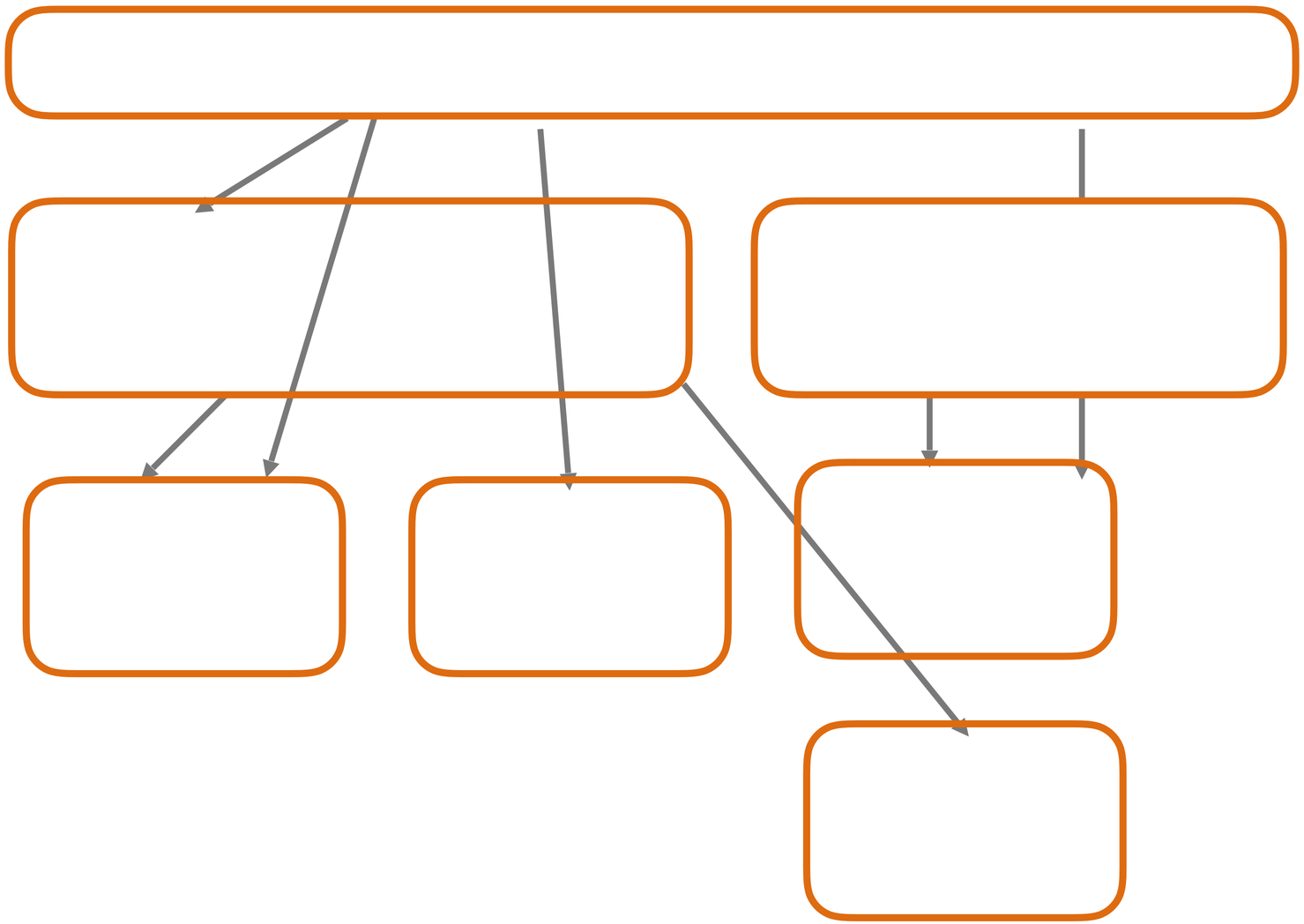
DISTRIBUTED MONOLITH



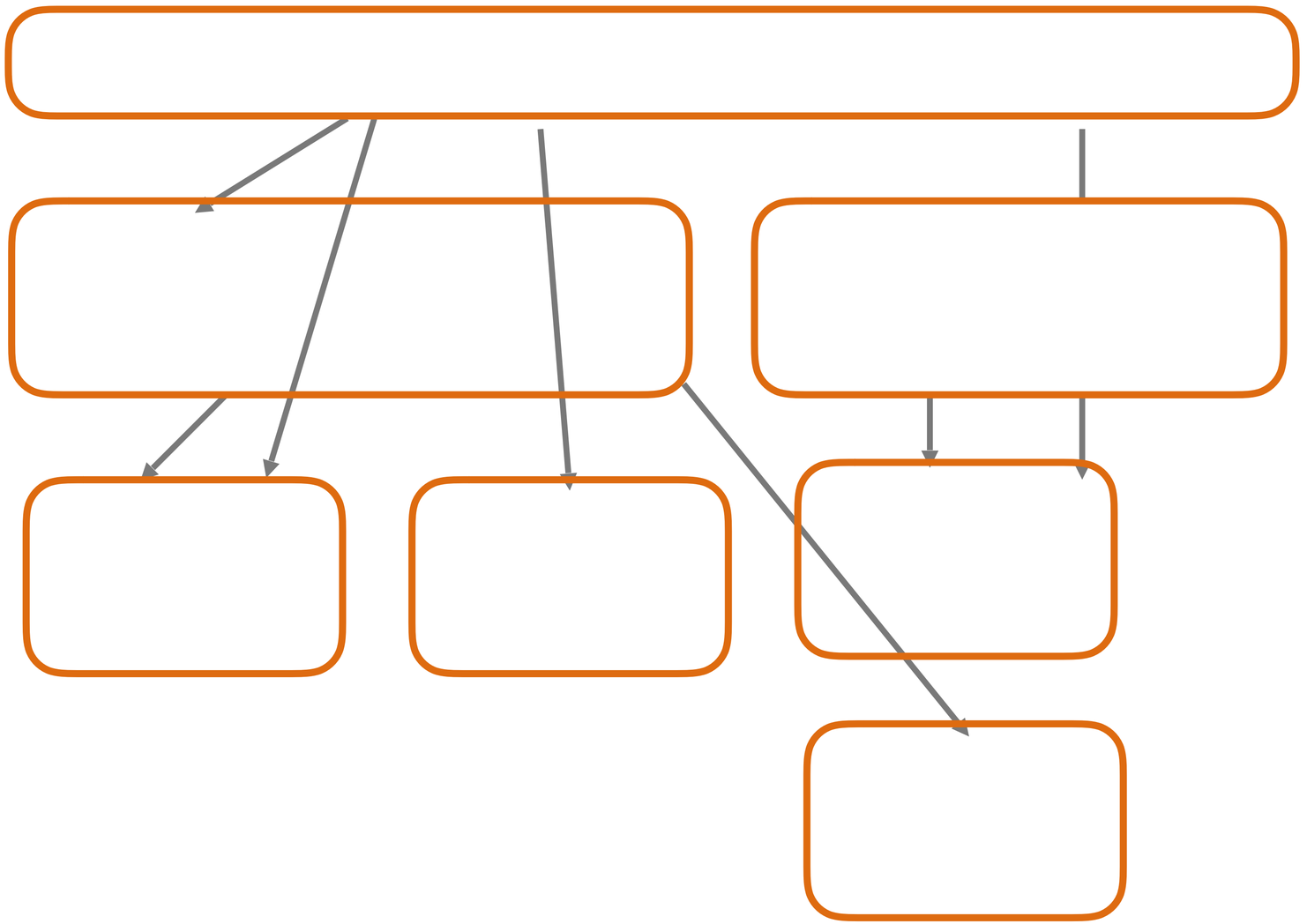
DISTRIBUTED MONOLITH



DISTRIBUTED MONOLITH

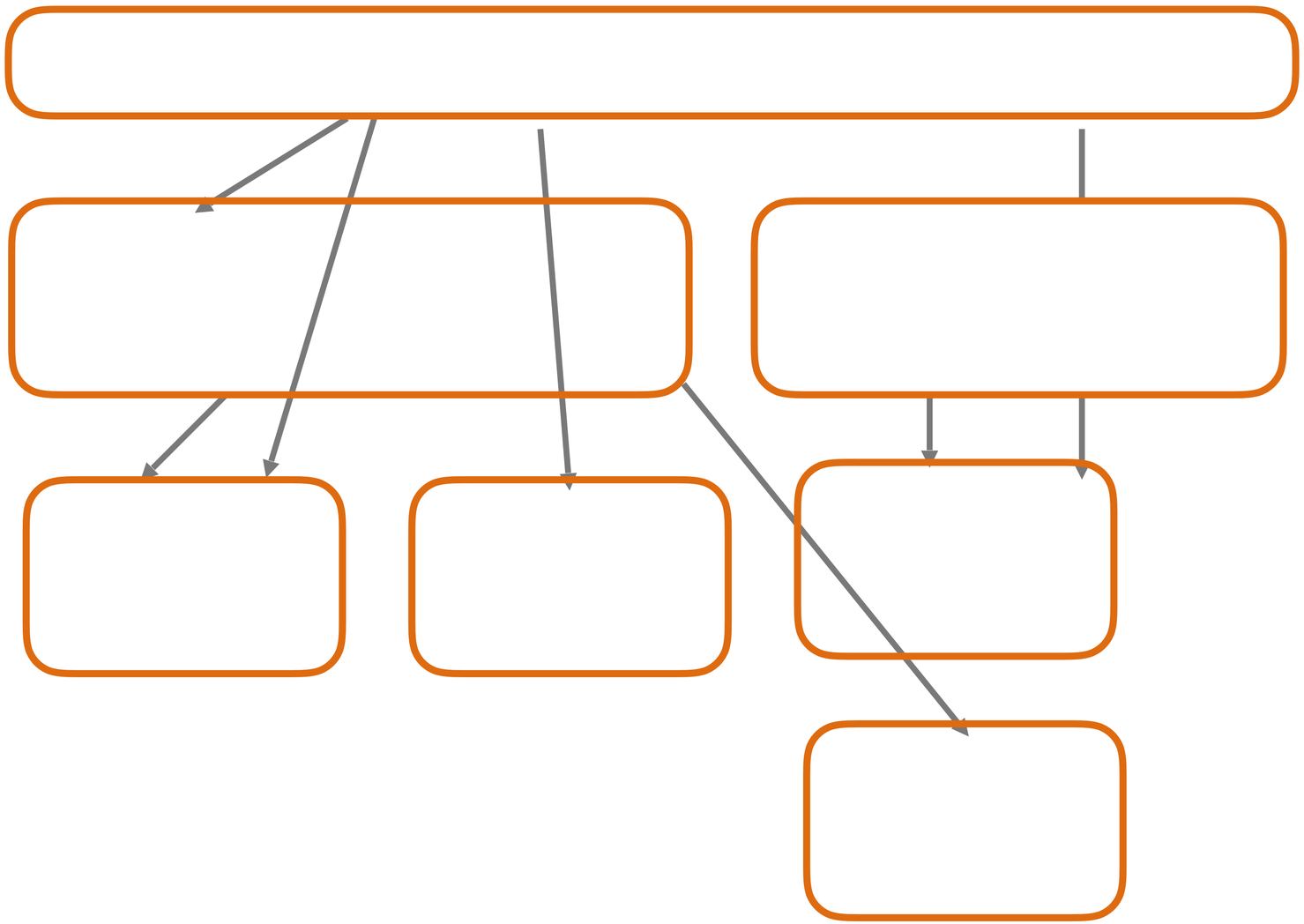


DISTRIBUTED MONOLITH



High cost of change

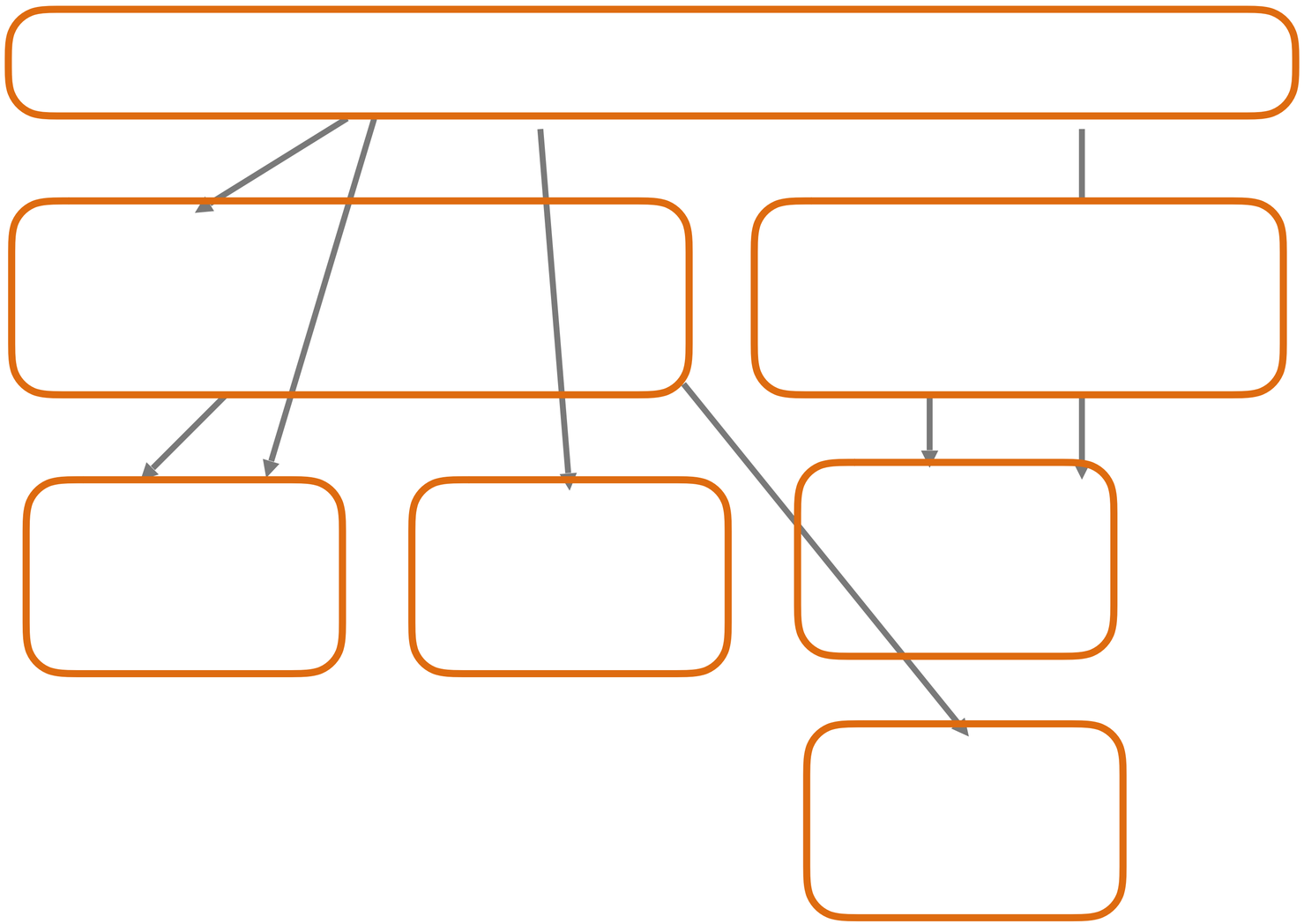
DISTRIBUTED MONOLITH



High cost of change

Larger-scoped deployments

DISTRIBUTED MONOLITH

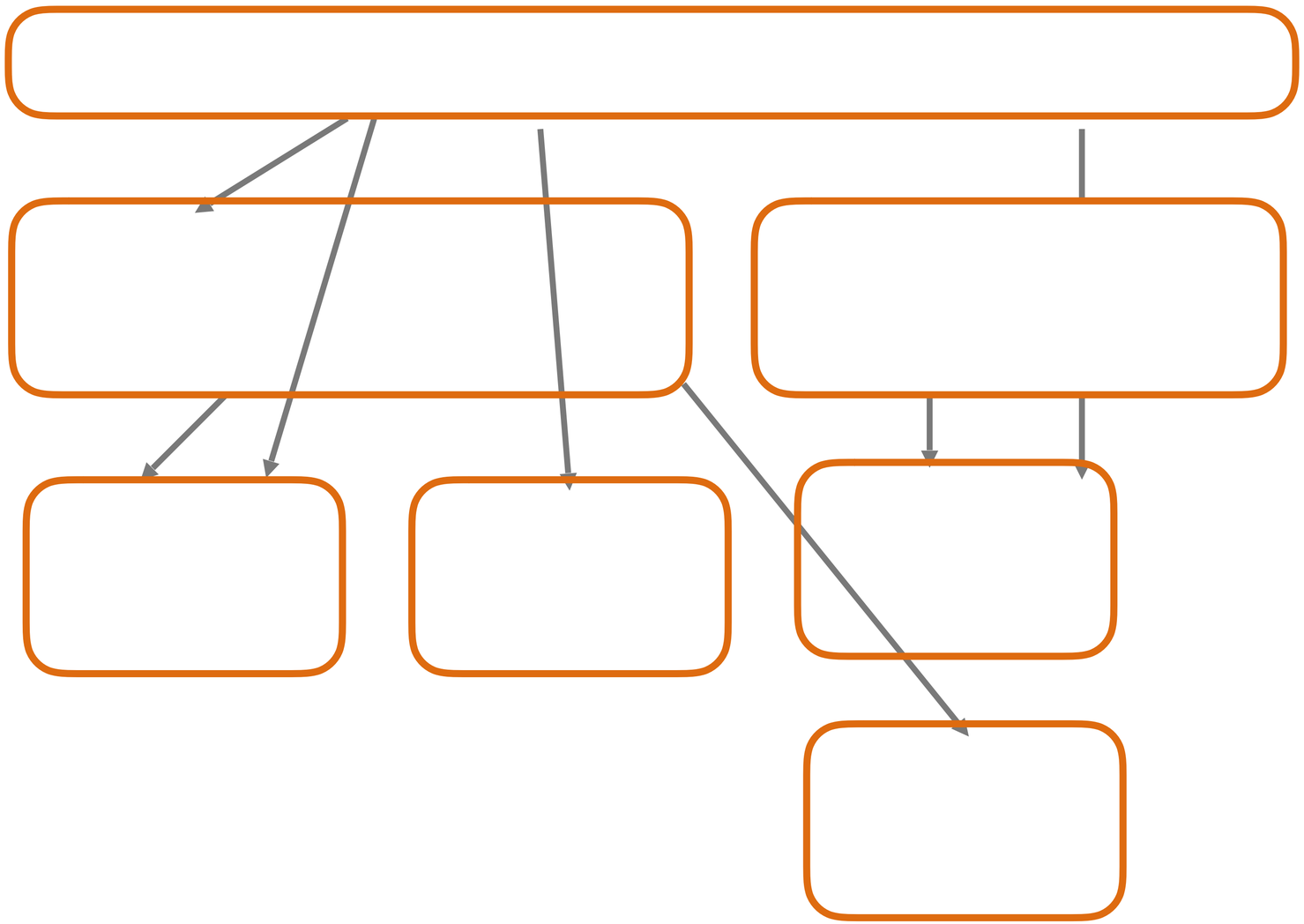


High cost of change

Larger-scoped deployments

More to go wrong

DISTRIBUTED MONOLITH



High cost of change

Larger-scoped deployments

More to go wrong

Release co-ordination

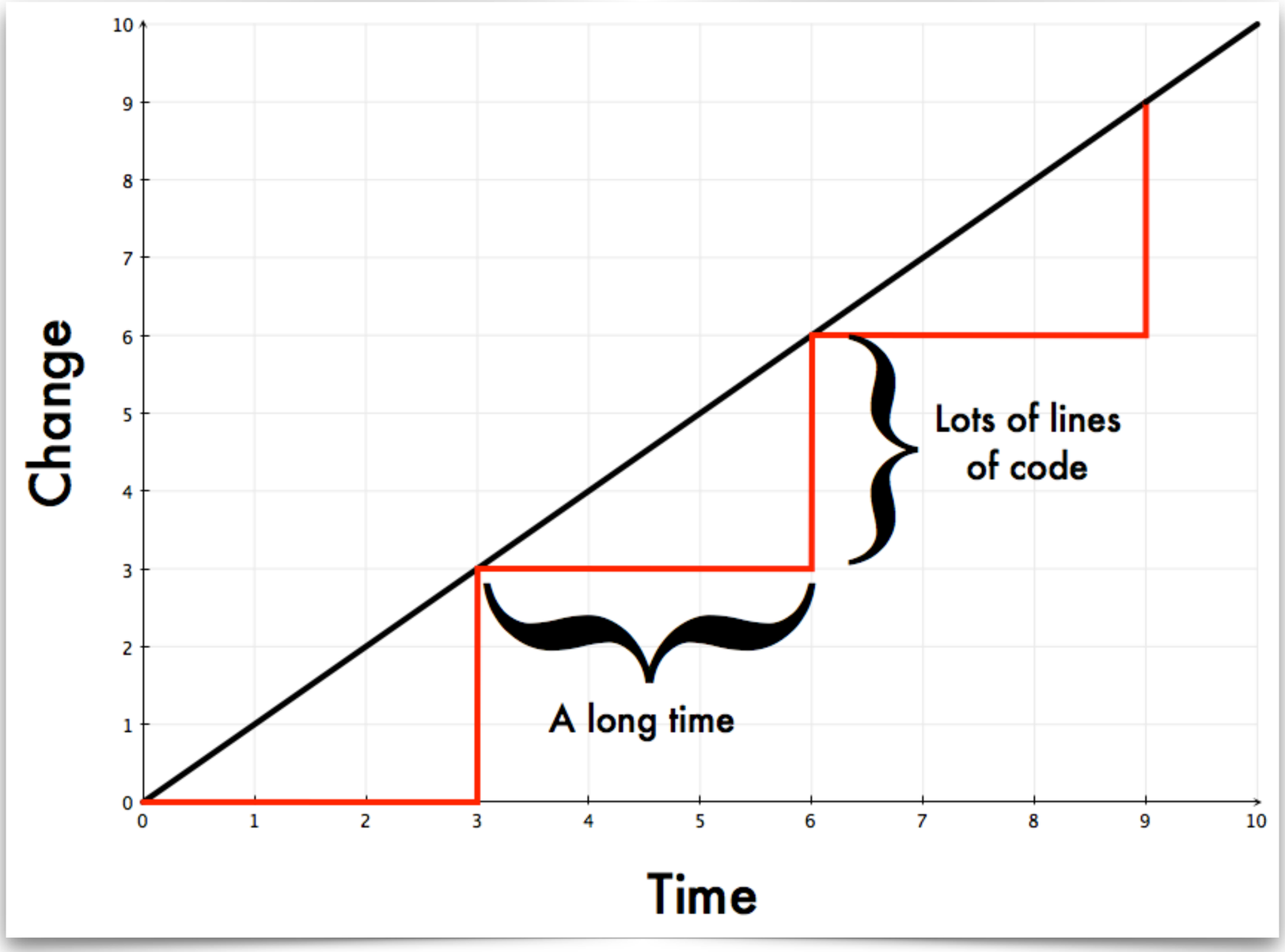
https://www.flickr.com/photos/kansasscanner/32189230183/

SAFE

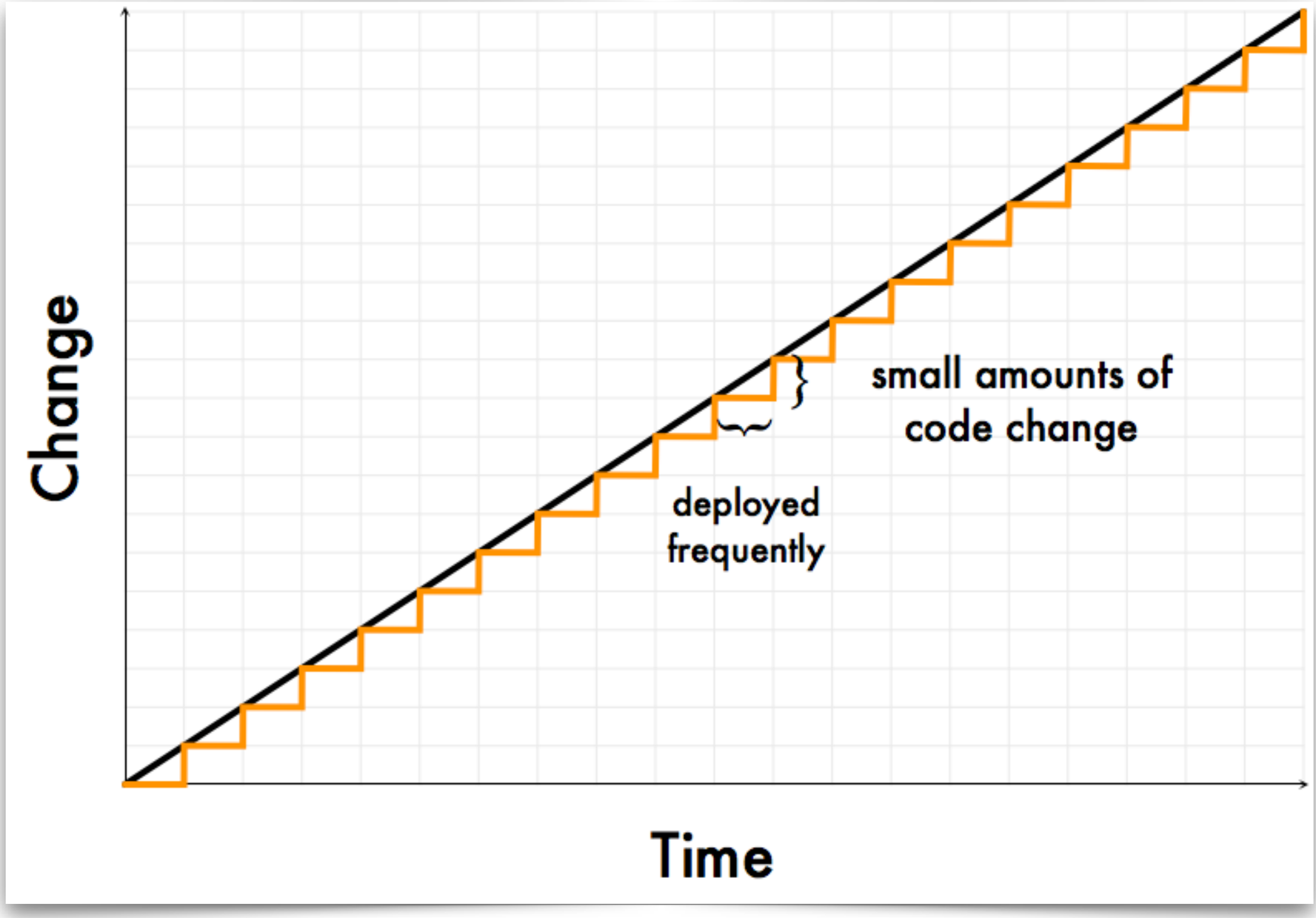
SAFE

Watch out, there’s a release train coming!

http://www.slideshare.net/jallspaw/ops-metametrics-the-currency-you-pay-for-change-4608108



http://www.slideshare.net/jallspaw/ops-metametrics-the-currency-you-pay-for-change-4608108



http://www.slideshare.net/jallspaw/ops-metametrics-the-currency-you-pay-for-change-4608108

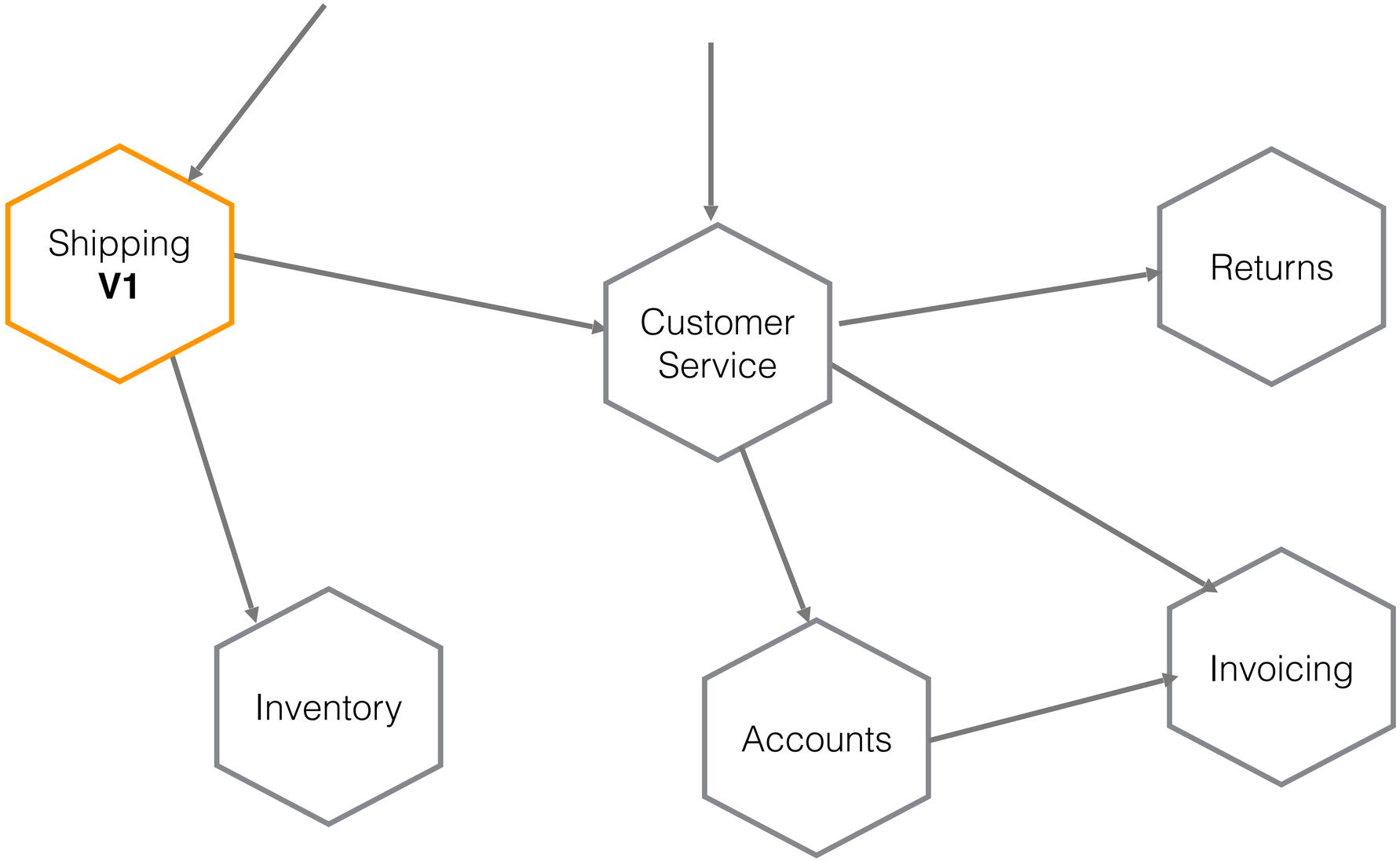
If you stick with a release train for too long, you'll end up with a distributed monolith

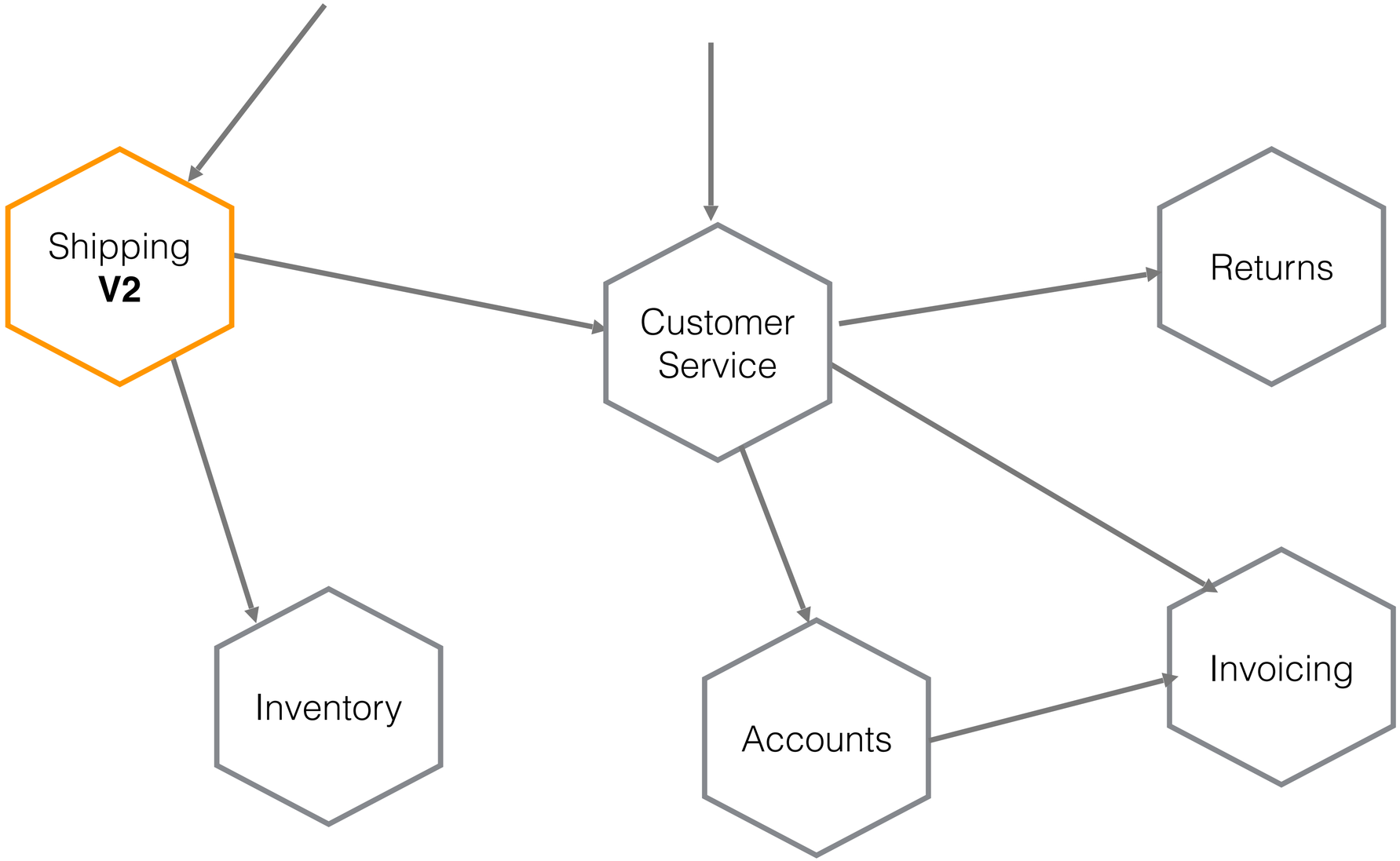
The Distributed Monolith

For when life isn't already complicated enough

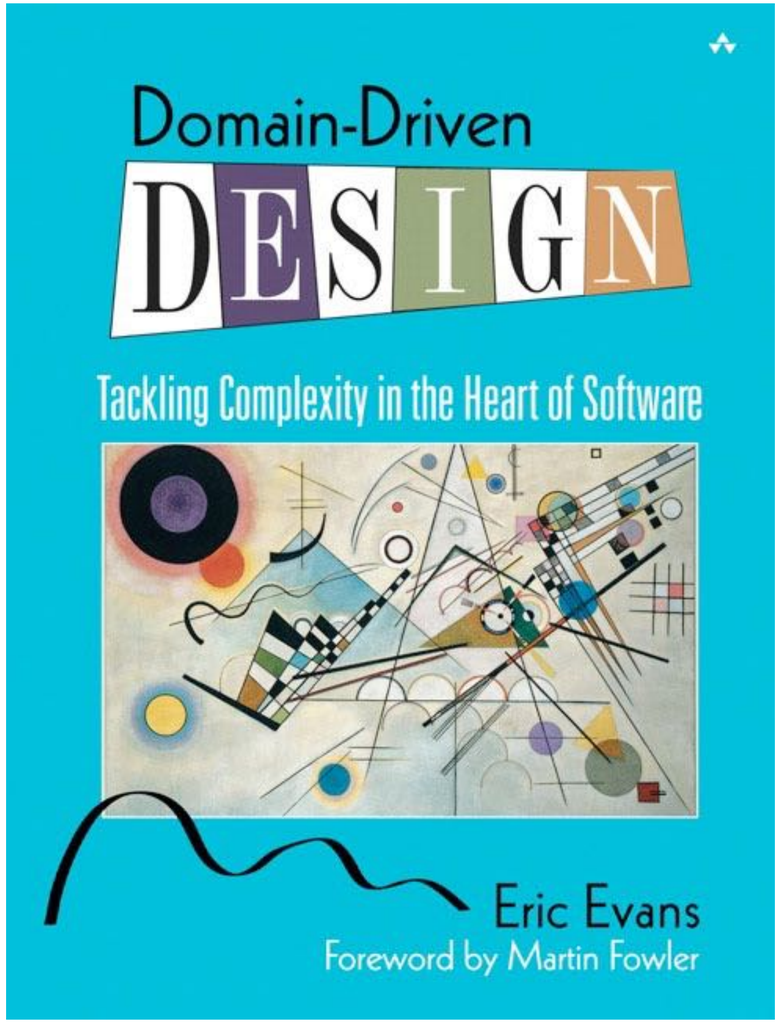
Continuous Delivery

Release on demand

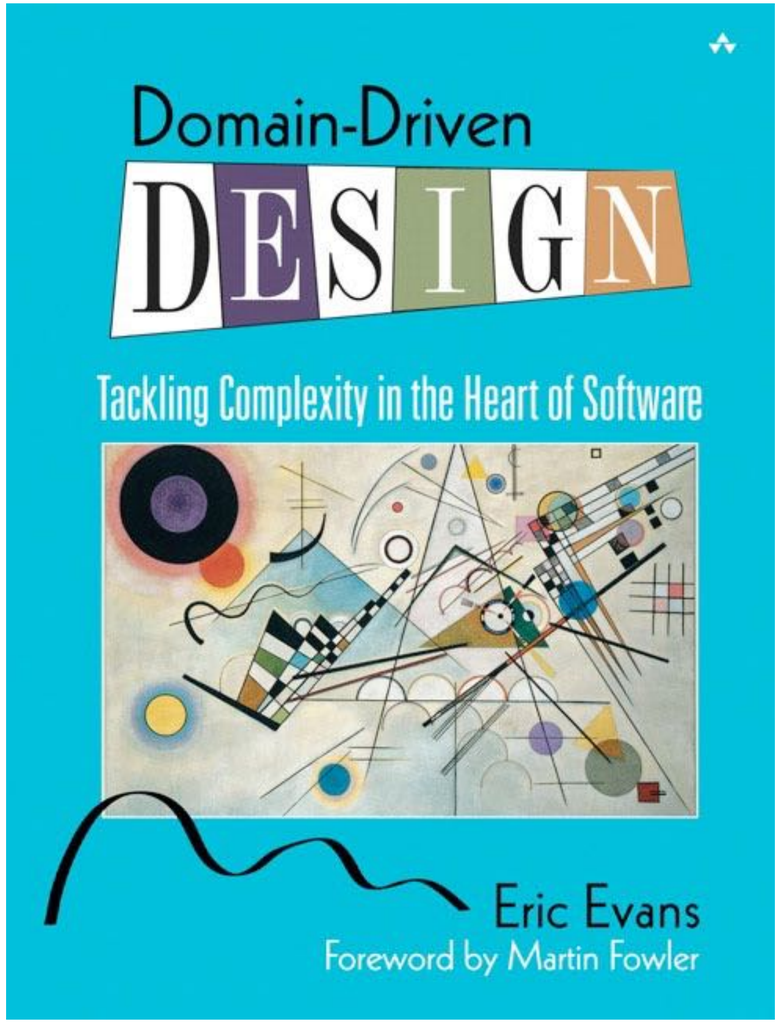

Shipping
V1
Customer
Service
Returns
Inventory
Accounts
Invoicing

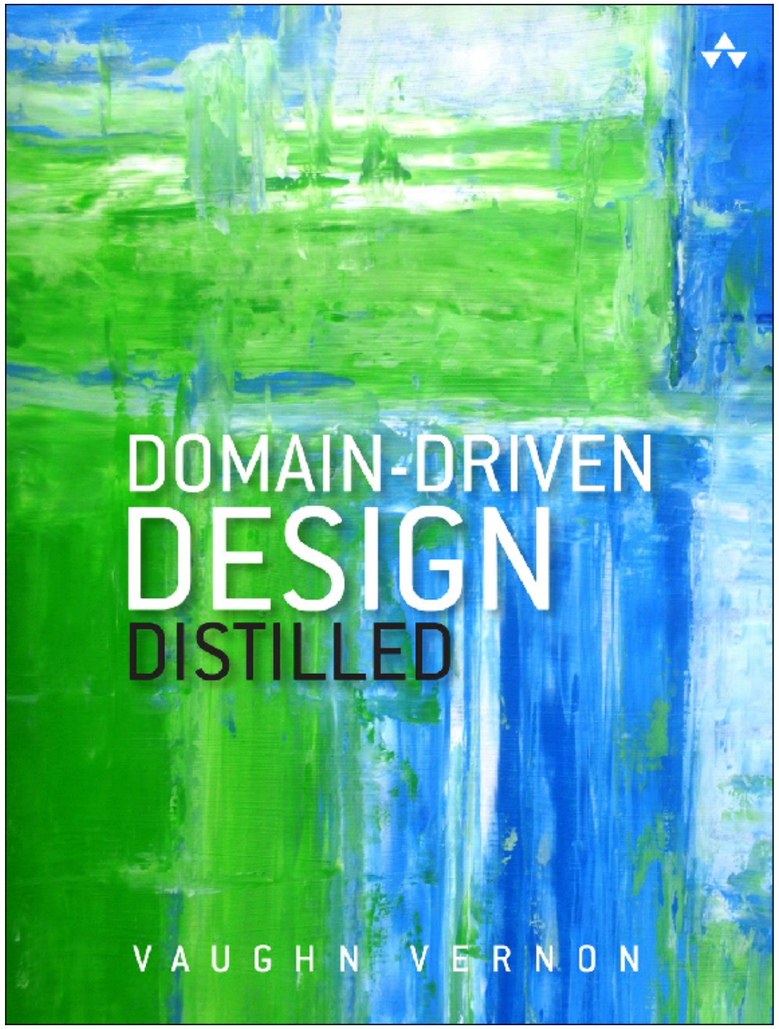

Shipping
V2
Customer
Service
Returns
Inventory
Accounts
Invoicing

DOMAIN DRIVEN DESIGN

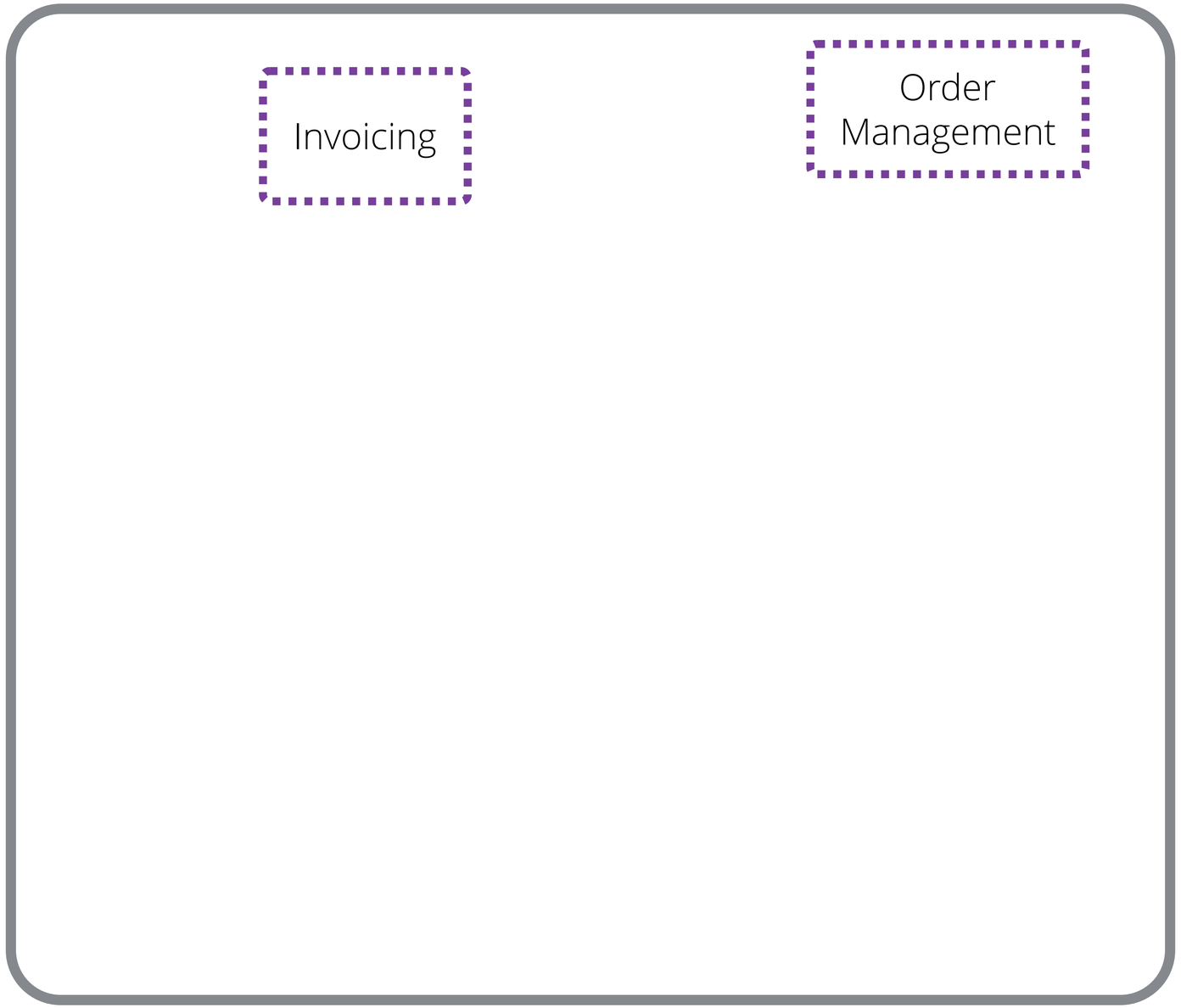


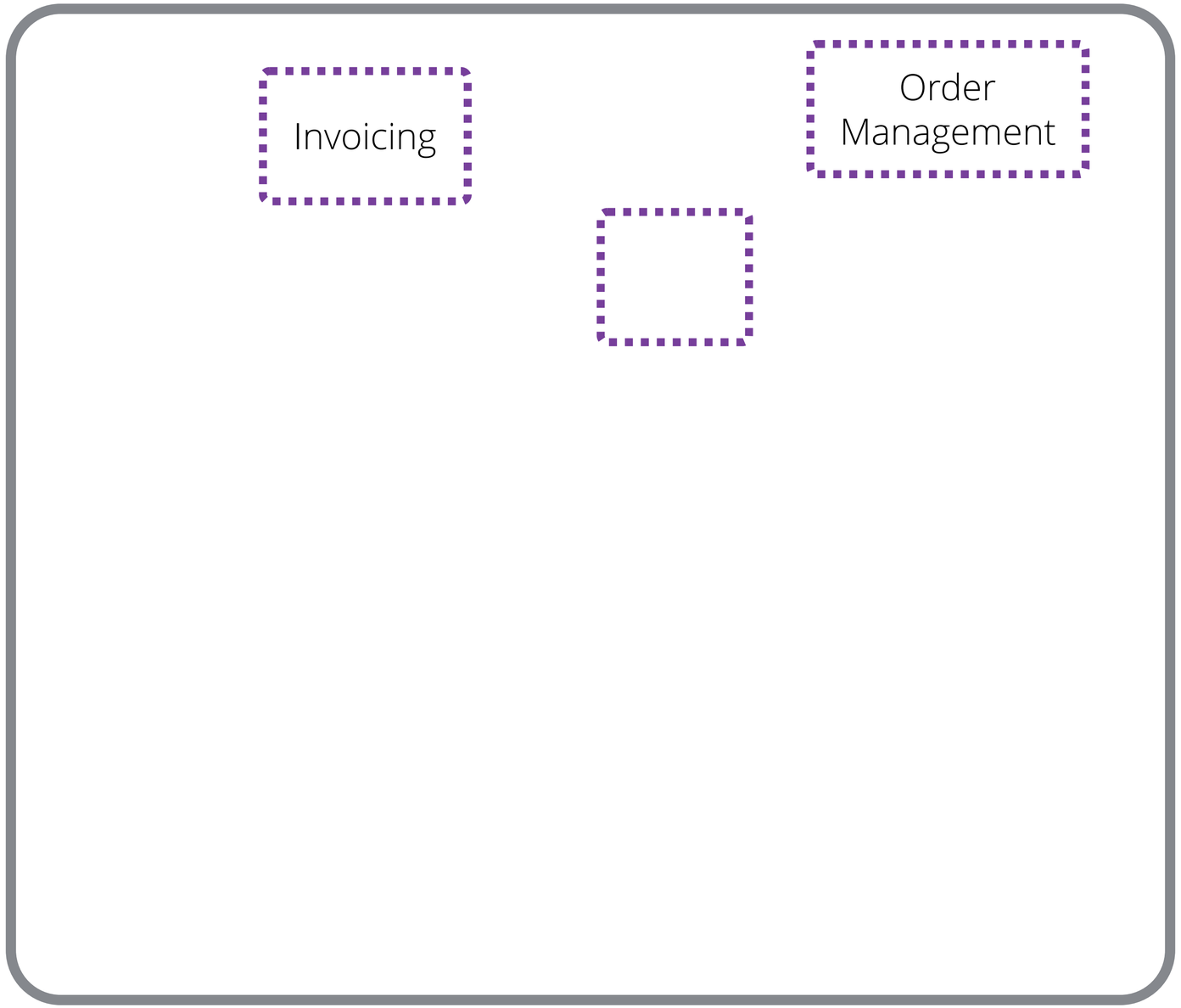
DOMAIN DRIVEN DESIGN

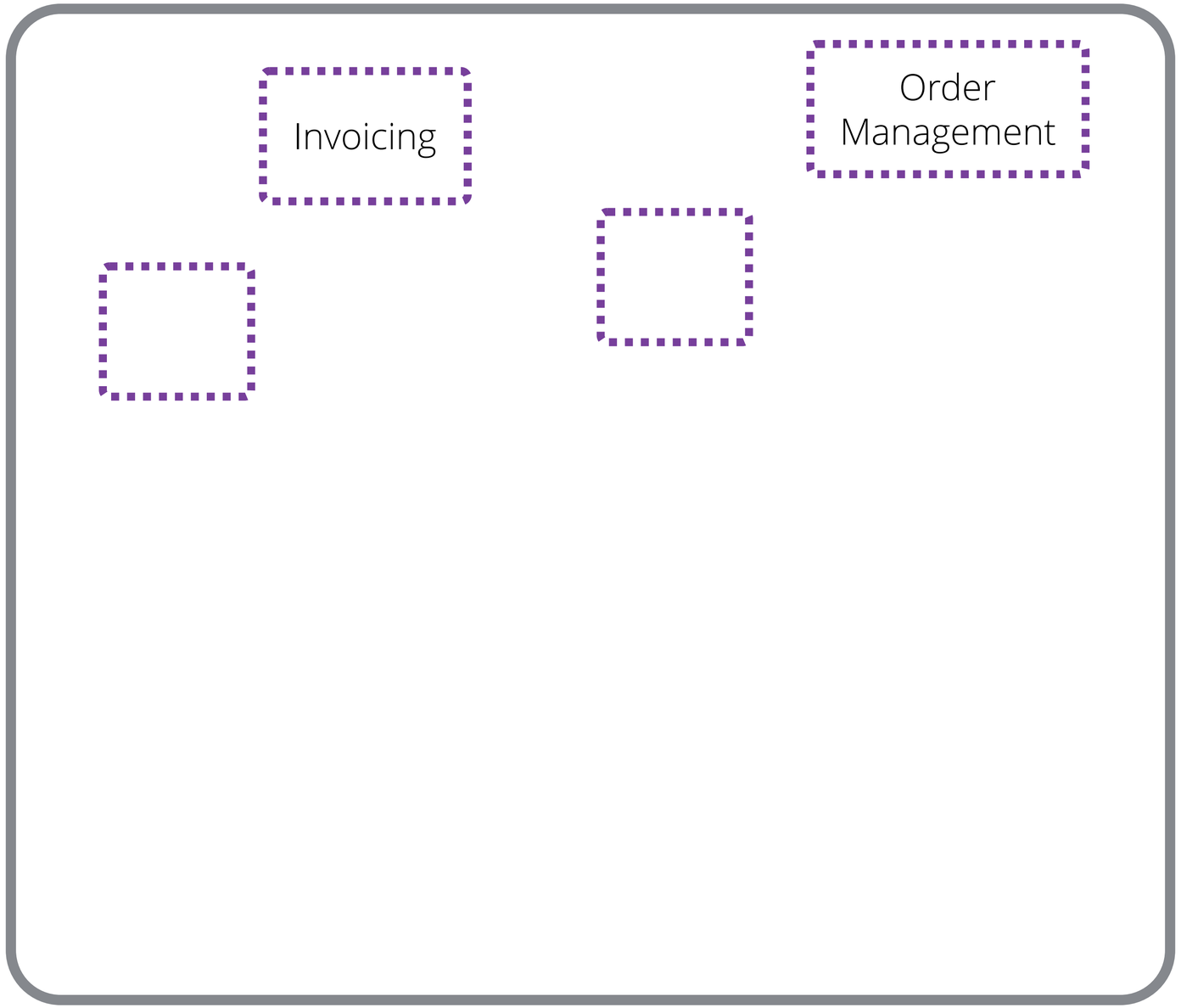


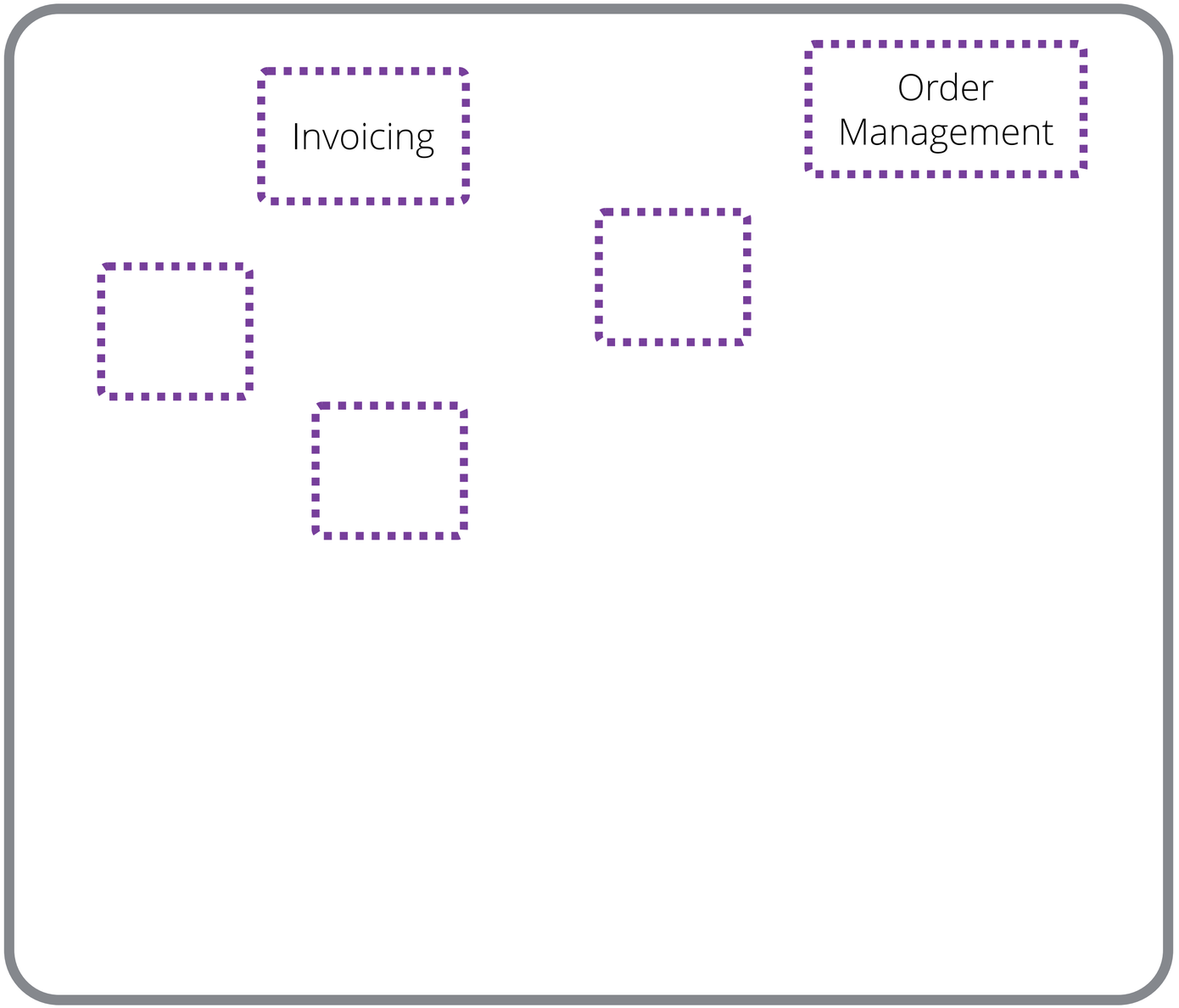


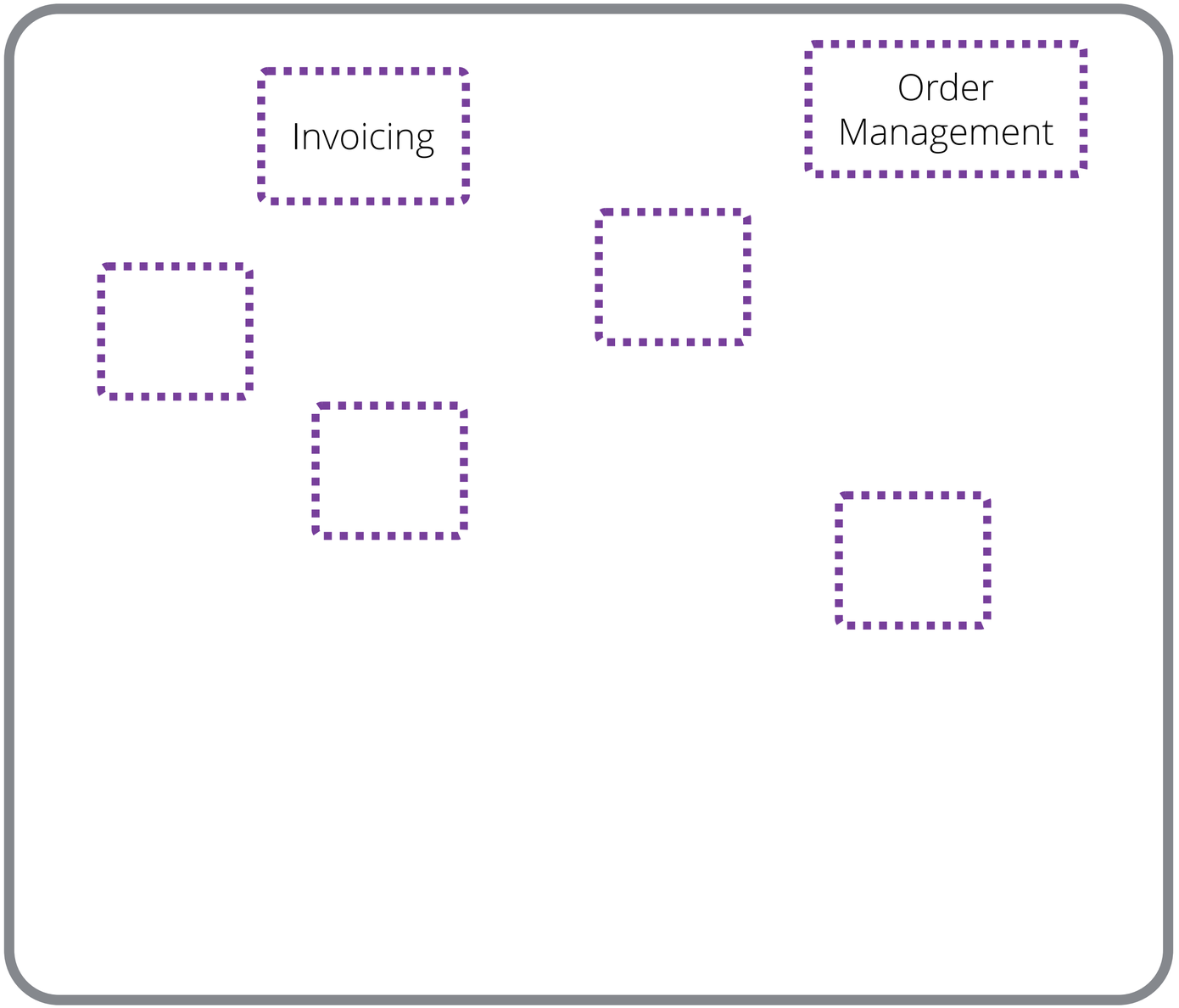

Order
Management

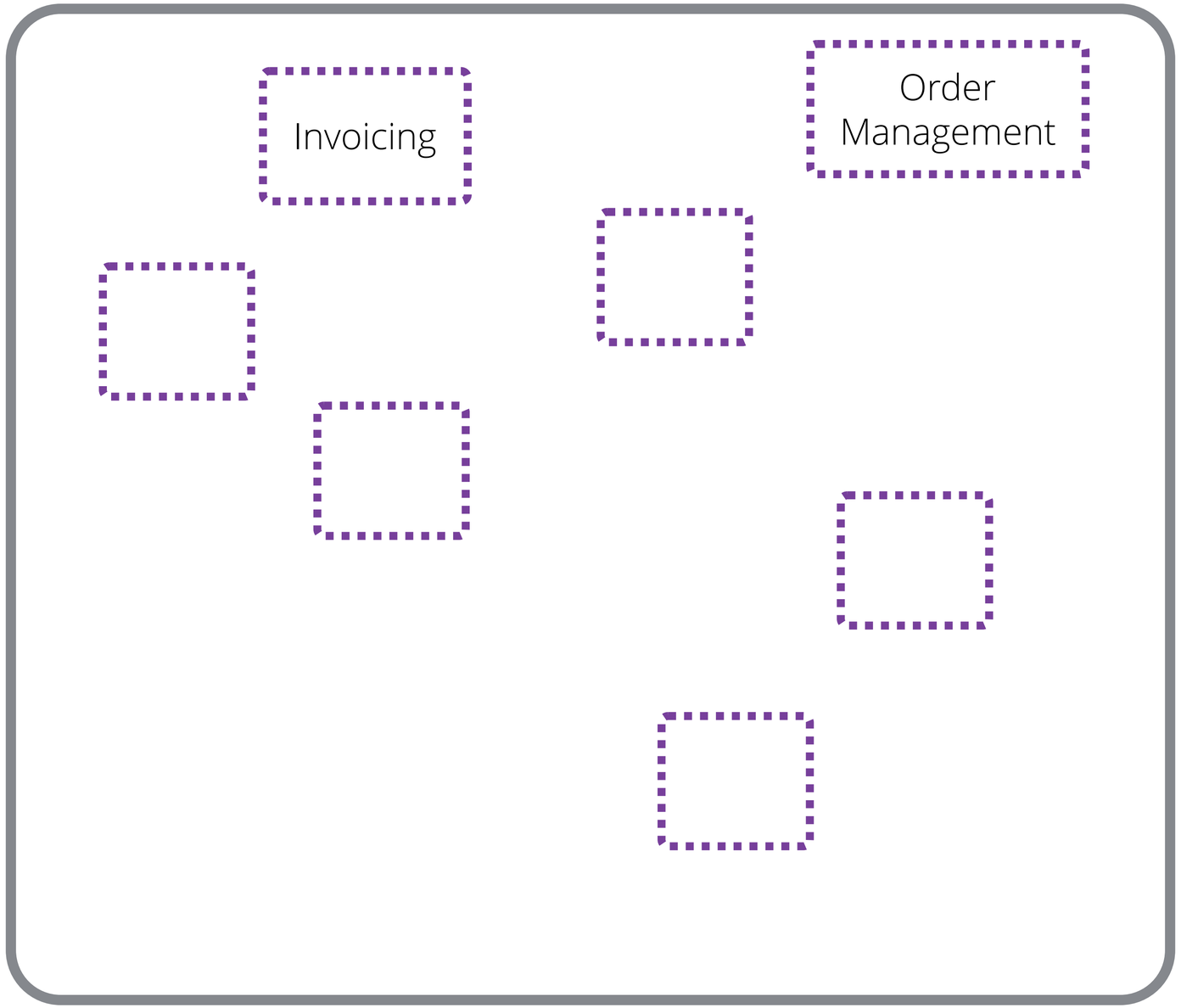

Invoicing
Order Management

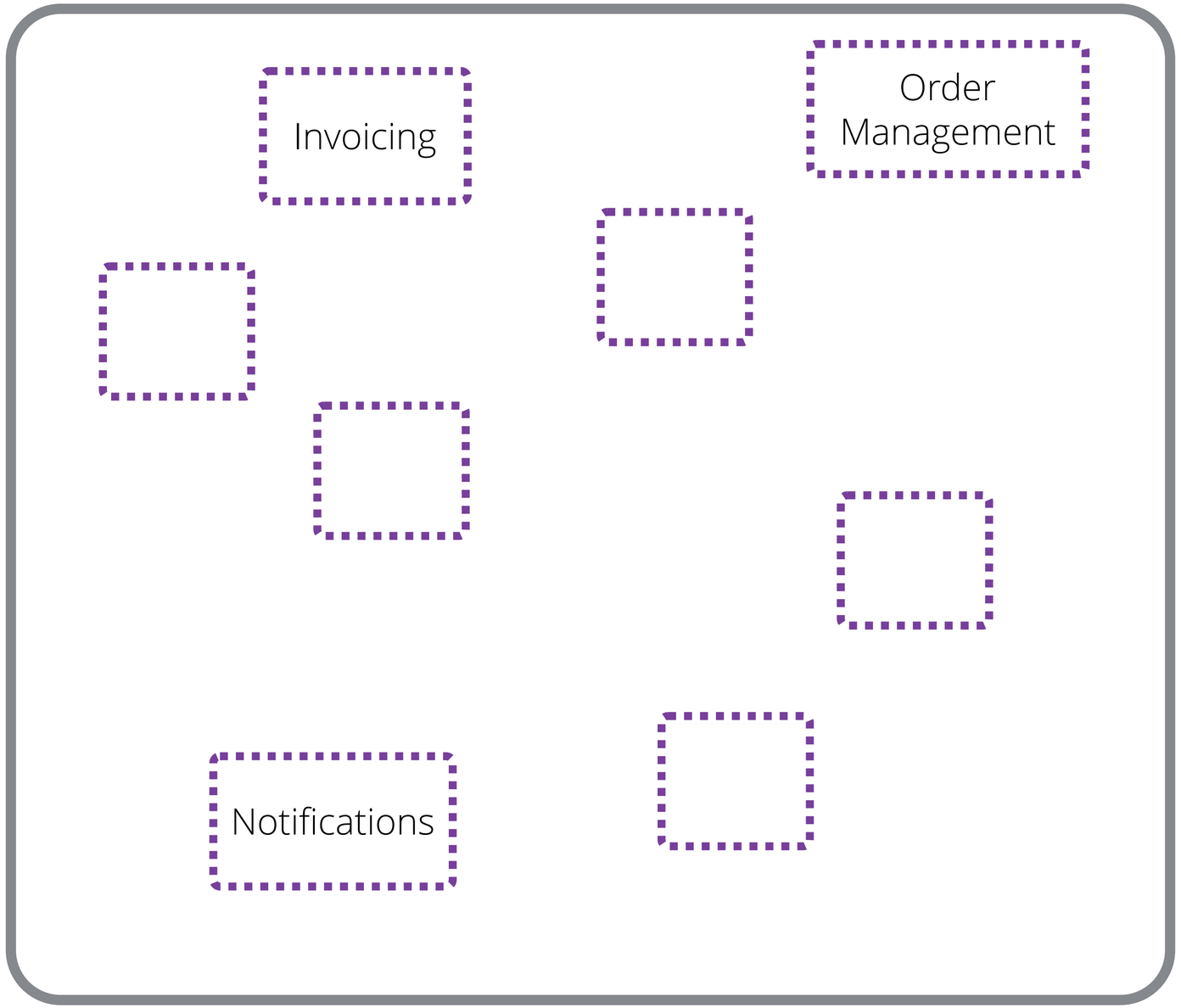

Invoicing
Order Management

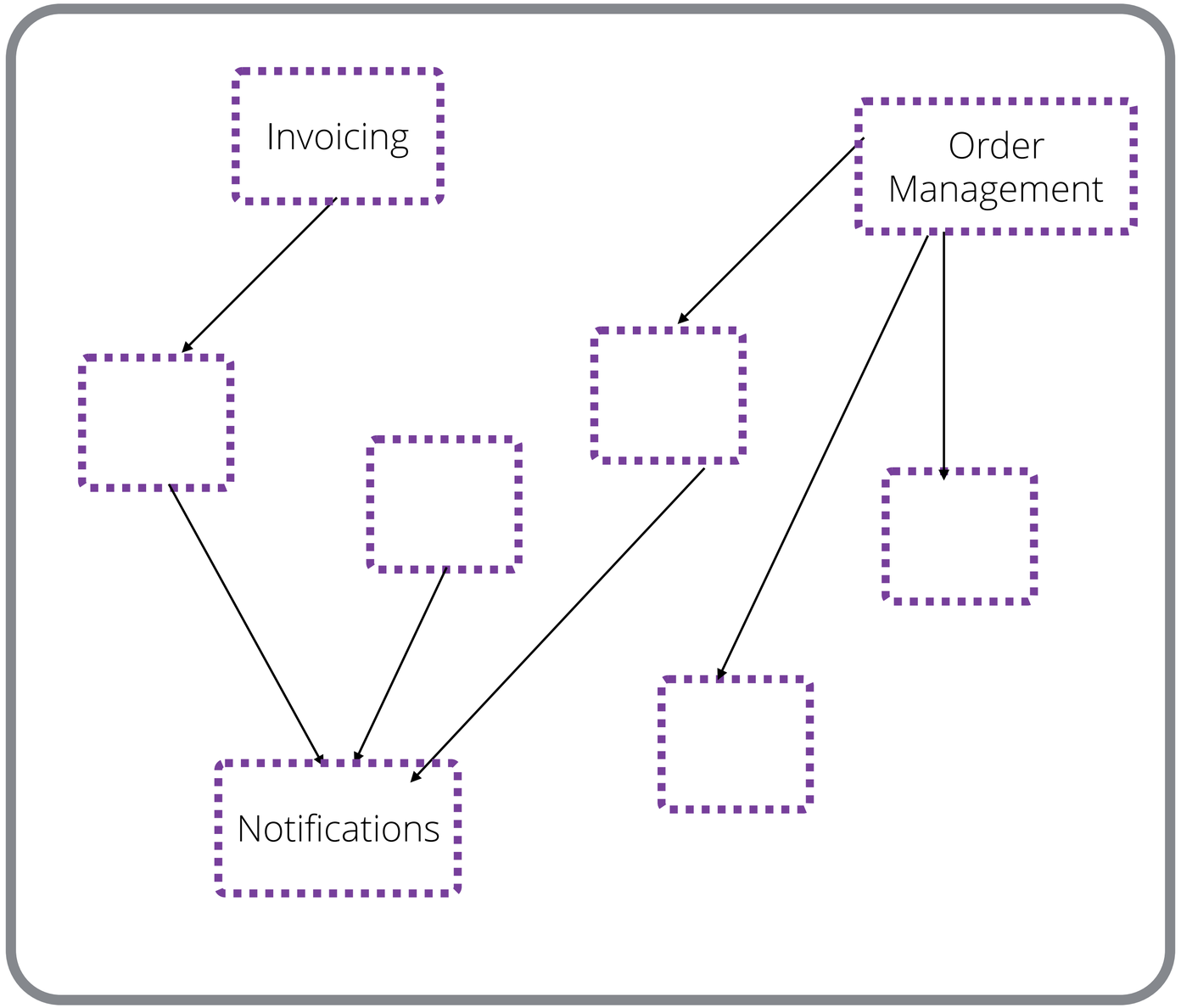

Invoicing
Order Management

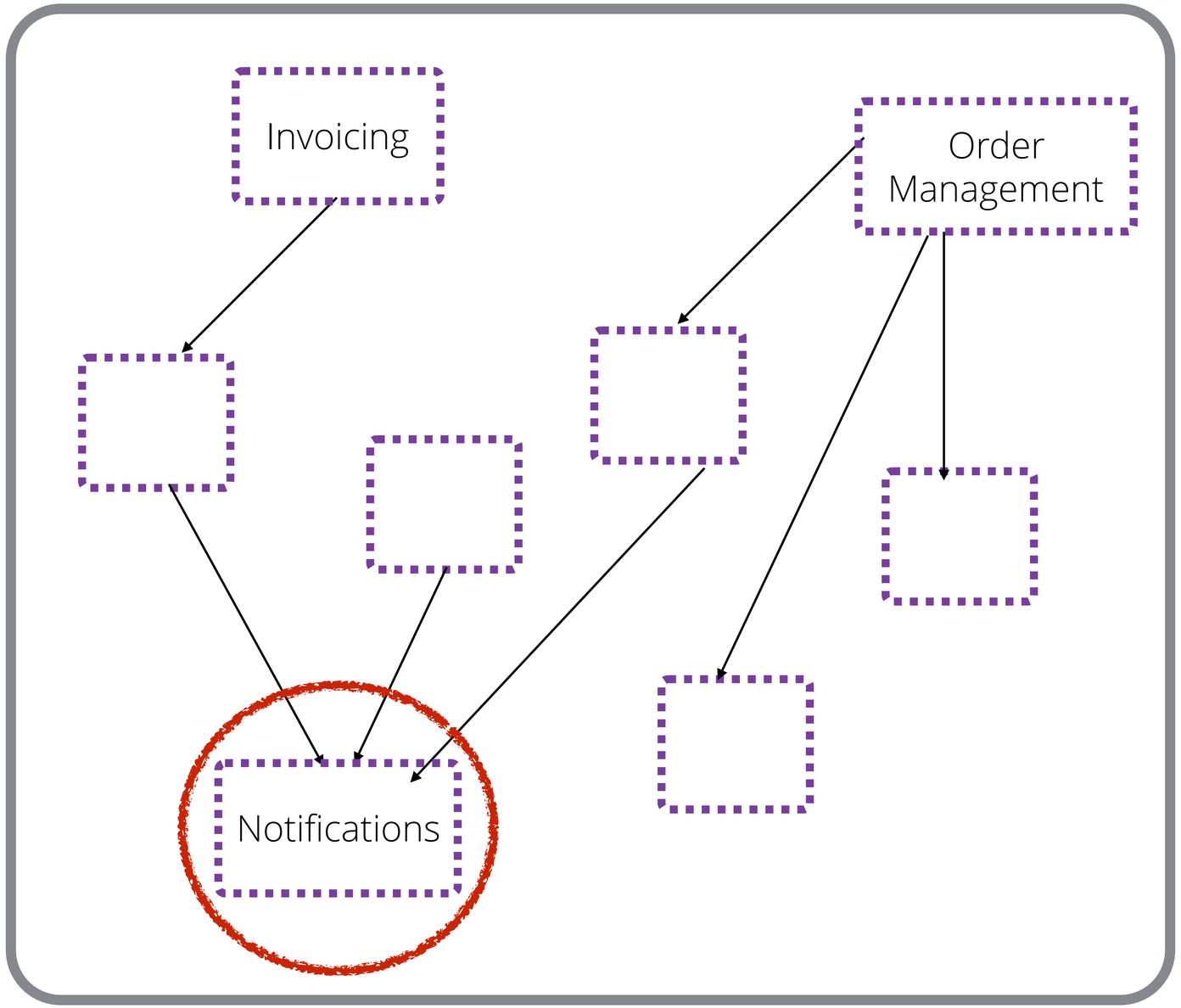

Invoicing
Order
Management

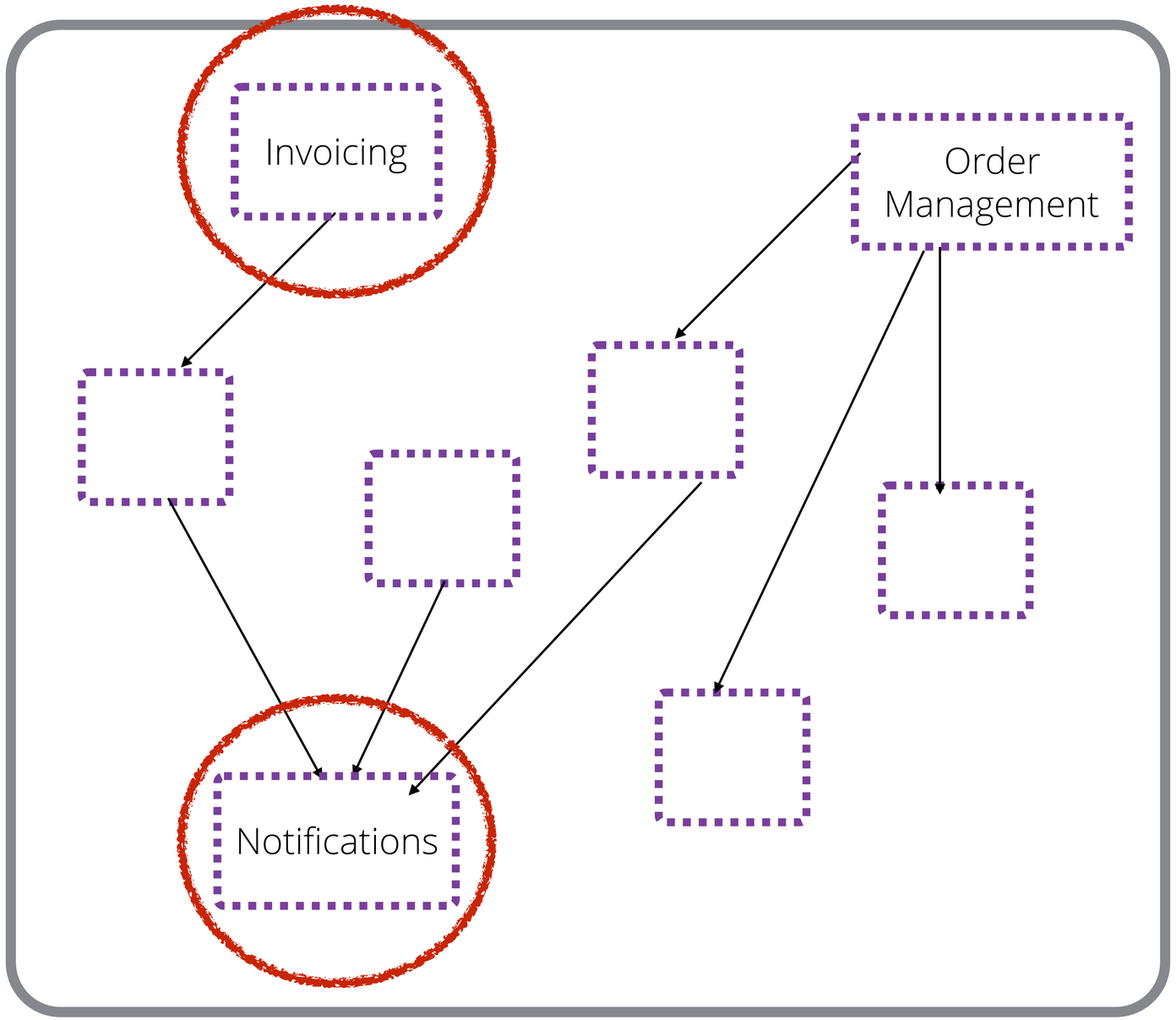

Invoicing
Order
Management


Invoicing
Order
Management


Invoicing
Order Management
Notifications


Invoicing
Order Management
Notifications


Invoicing
Order Management
Notifications


Invoicing
Order Management
Notifications

The problems you are trying to solve with microservices will drive *how* you decompose them

VS

Ease Of Extraction

vs

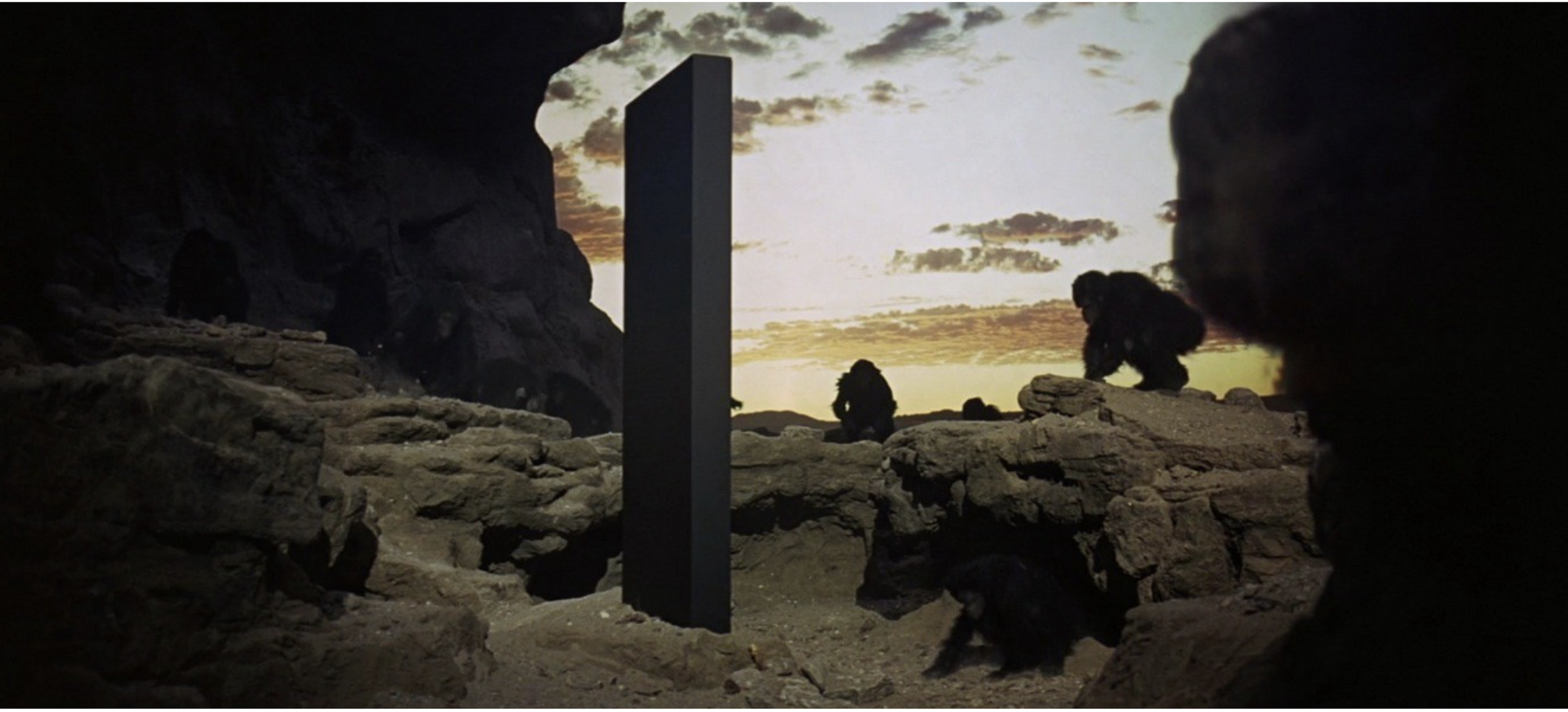
Ease Of Extraction

vs

Benefits of Decomposition

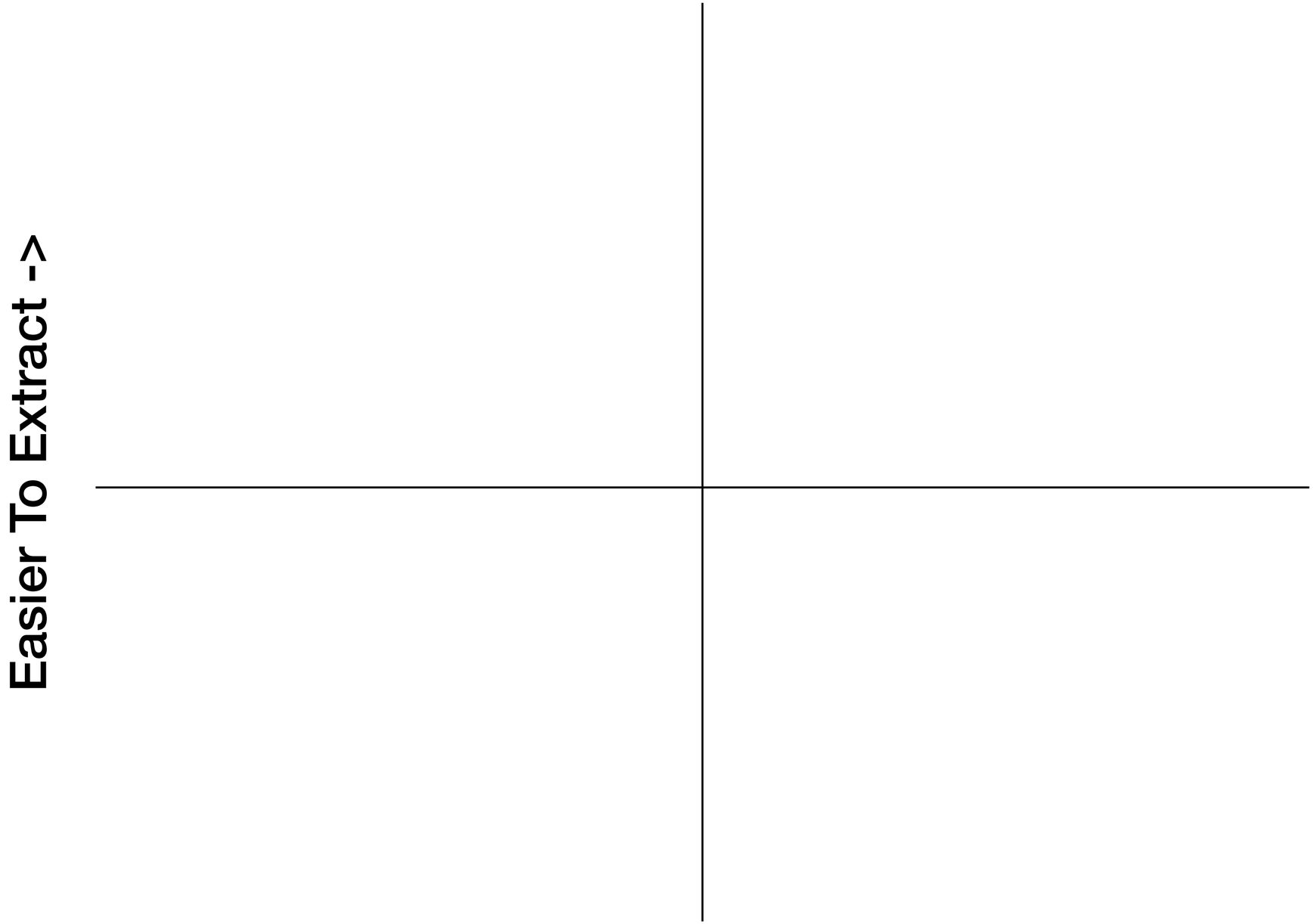
Monolith has become another word for “legacy”

Sometimes the monolith isn’t the enemy!

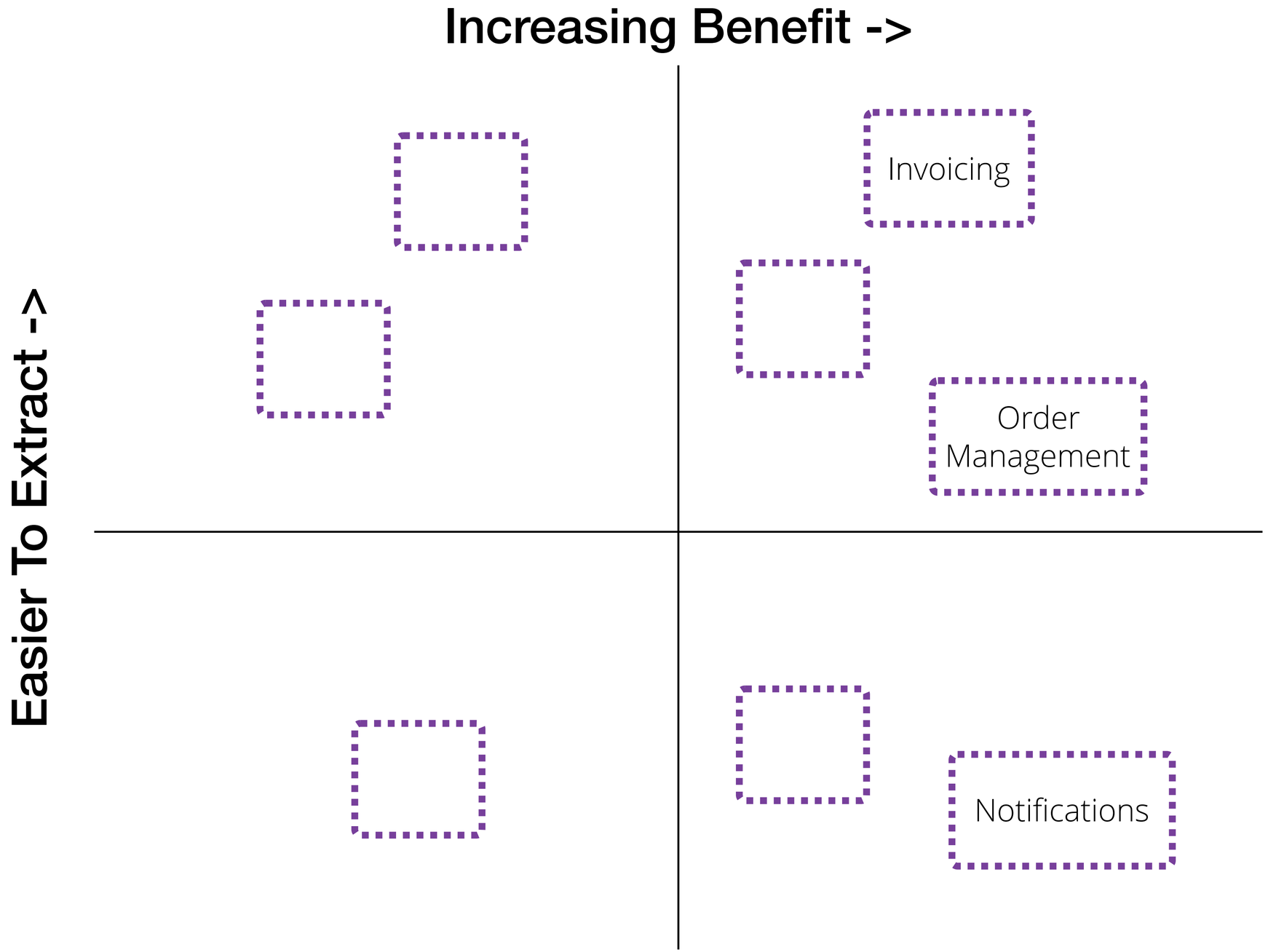


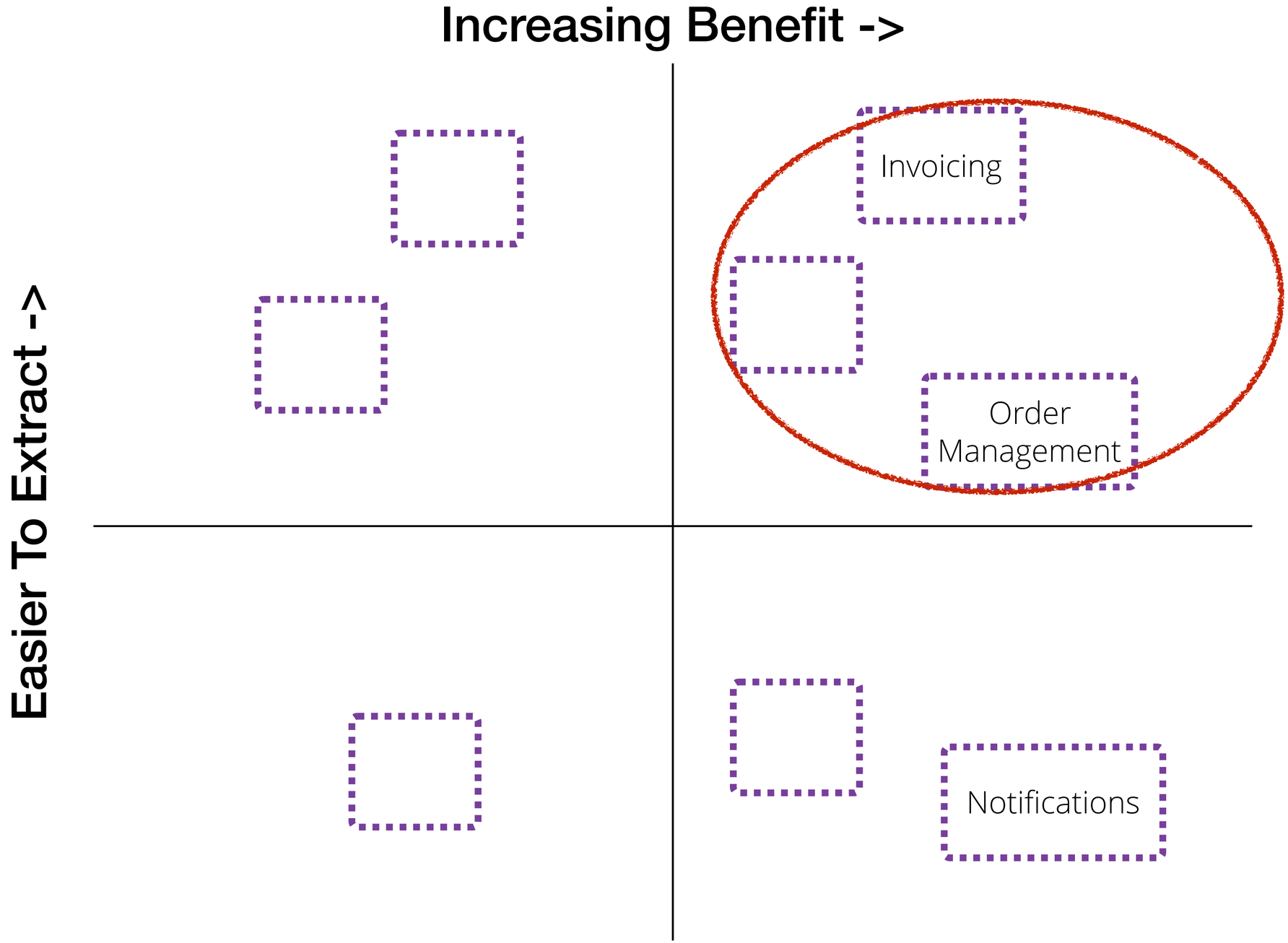
You do need a good reason to switch

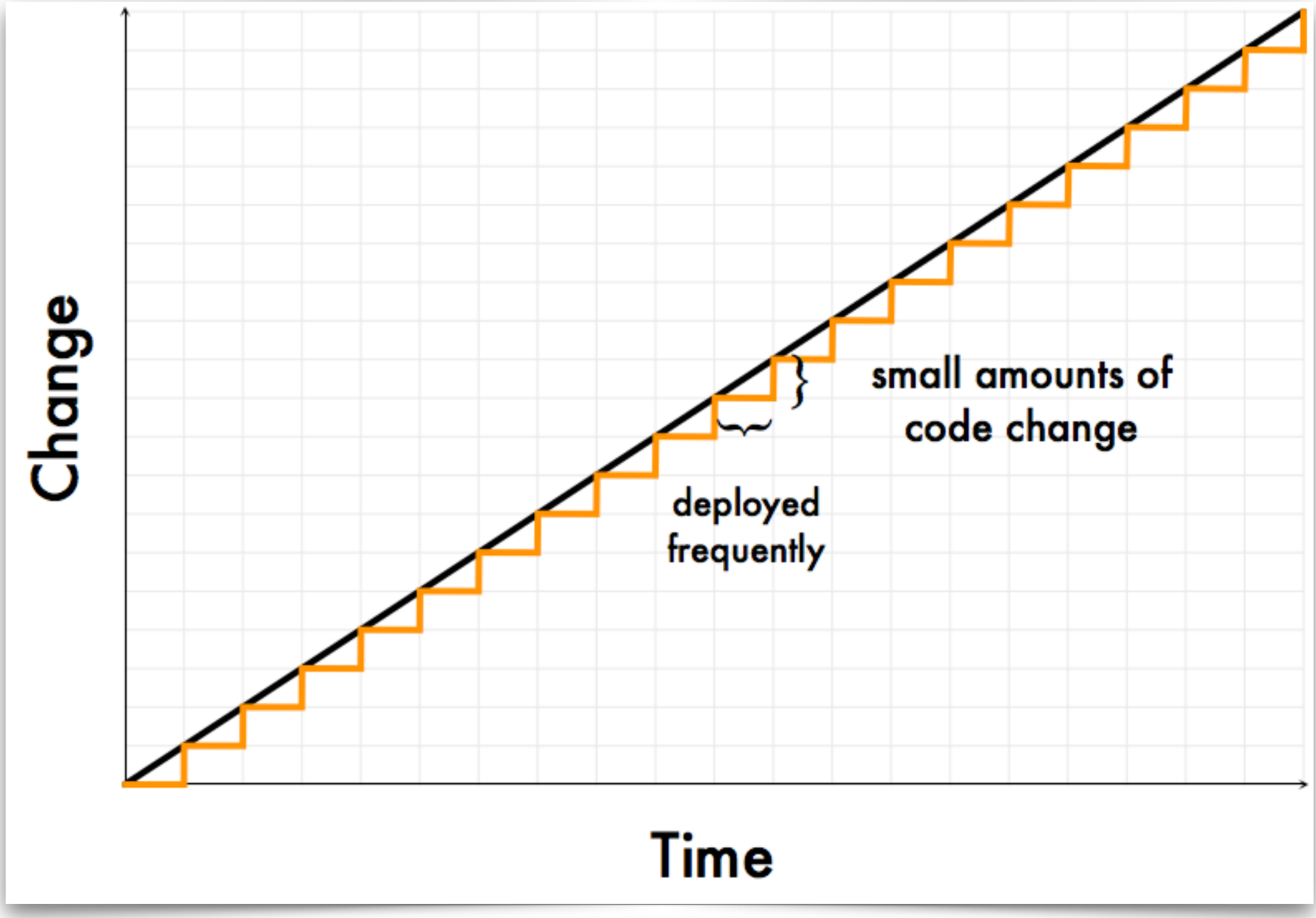
Increasing Benefit ->



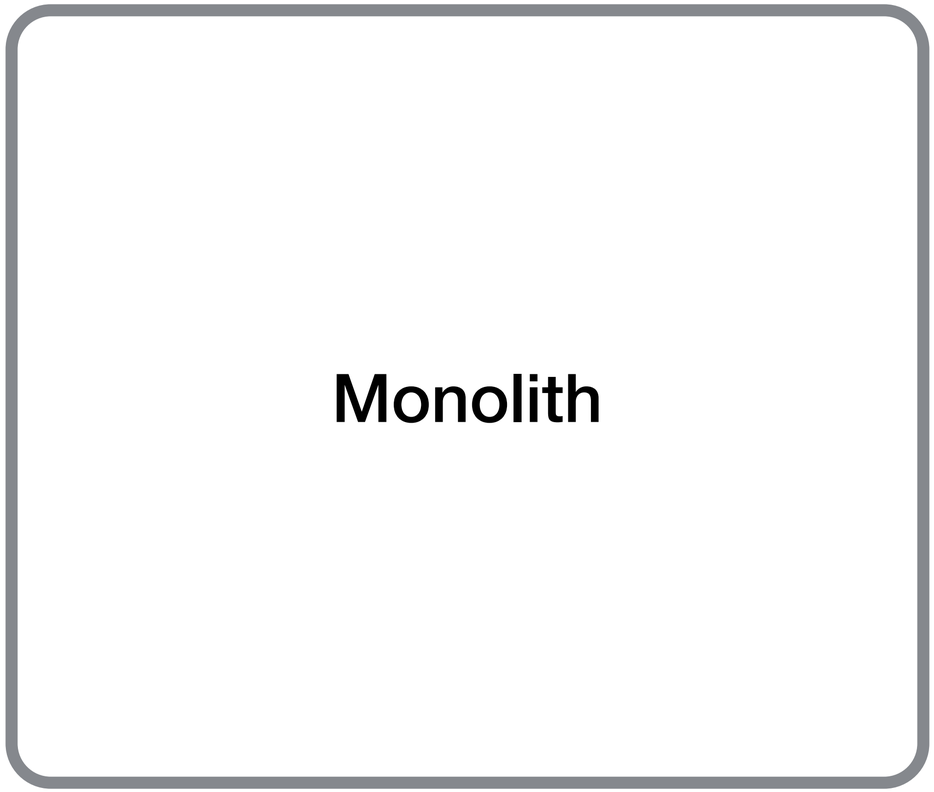
Easier To Extract ->


Increasing Benefit ->
Easier To Extract ->
Invoicing
Order Management
Notifications

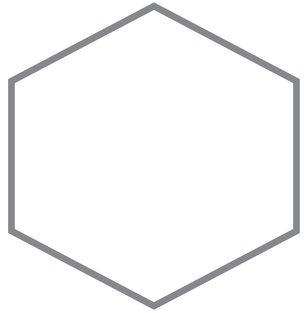

Increasing Benefit ->
Easier To Extract ->
Invoicing
Order Management
Notifications



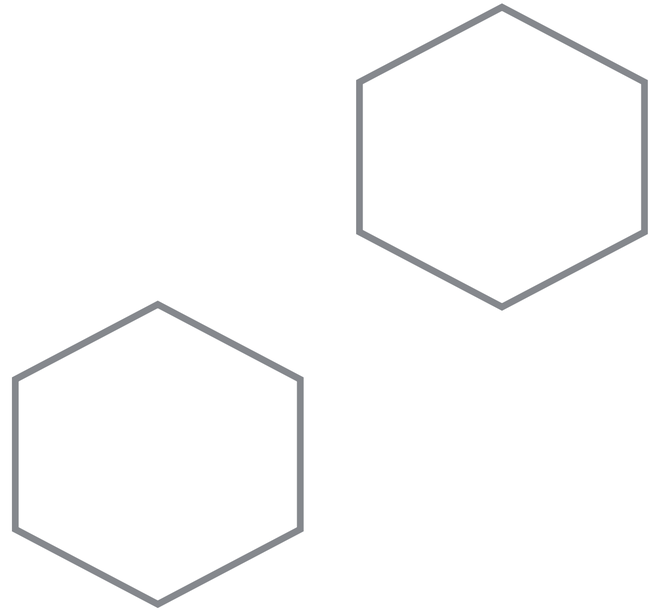
http://www.slideshare.net/jallspaw/ops-metametrics-the-currency-you-pay-for-change-4608108


Monolith

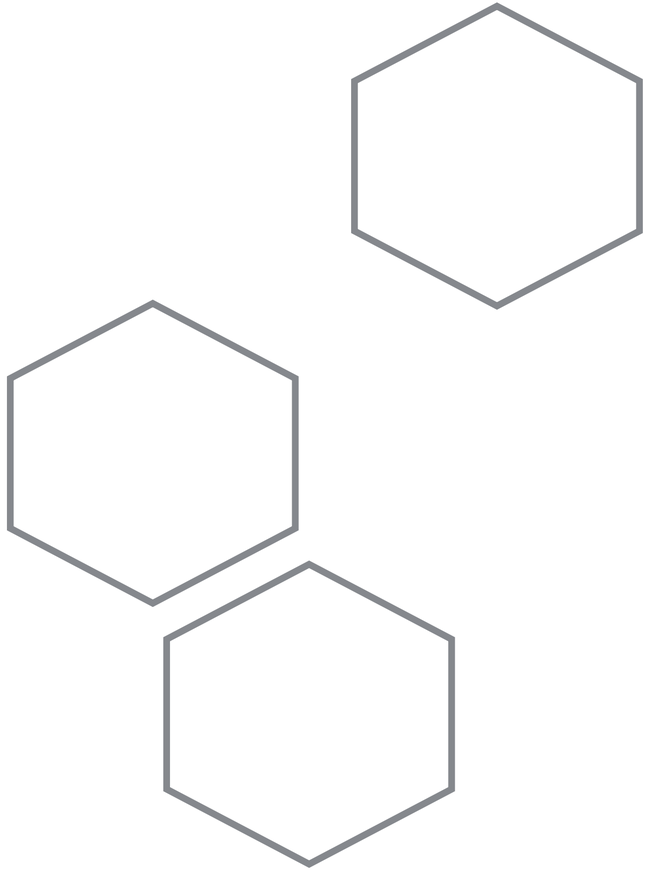
Monolith



Monolith

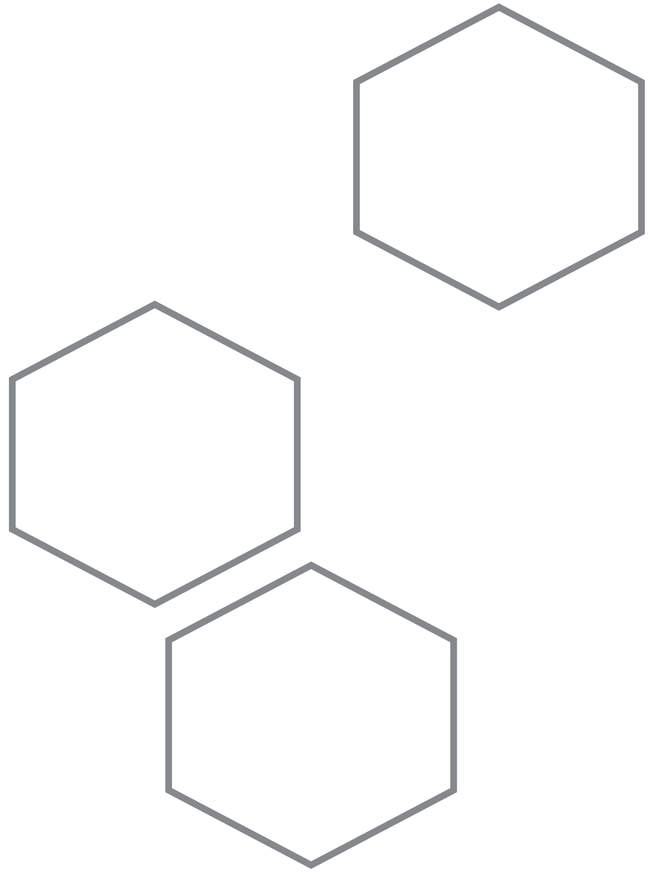


Monolith

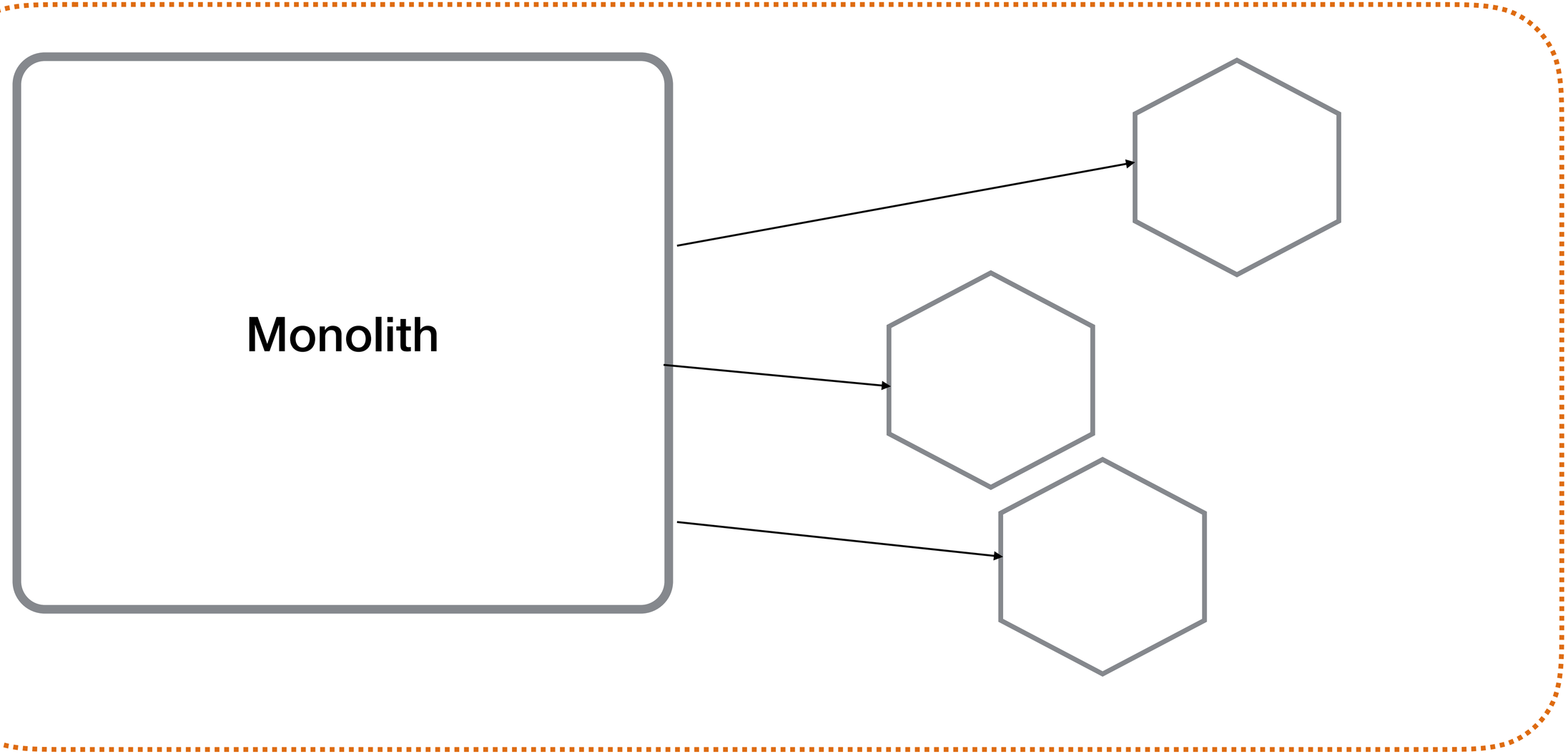


Production

Monolith



Production



“If you do a big bang rewrite, the only thing you’re certain of is a big bang”

- Martin Fowler (paraphrased)

https://www.flickr.com/photos/pahudson/3224209758/

So migration patterns we use must allow for incremental change

Patterns that allow for our architecture to evolve, whilst still delivering features

https://www.flickr.com/photos/cynren/16012074217/

IMPLEMENTING A STRANGLER FIG PATTERN

1. Asset capture:

Identify the functionality to move to a new microservice

IMPLEMENTING A STRANGLER FIG PATTERN

1. Asset capture:

Identify the functionality to move to a new microservice

2. Redirect calls

Intercept calls to old functionality, and redirect to the new service

“MOVING” FUNCTIONALITY?

“MOVING” FUNCTIONALITY?

It might be copy and paste

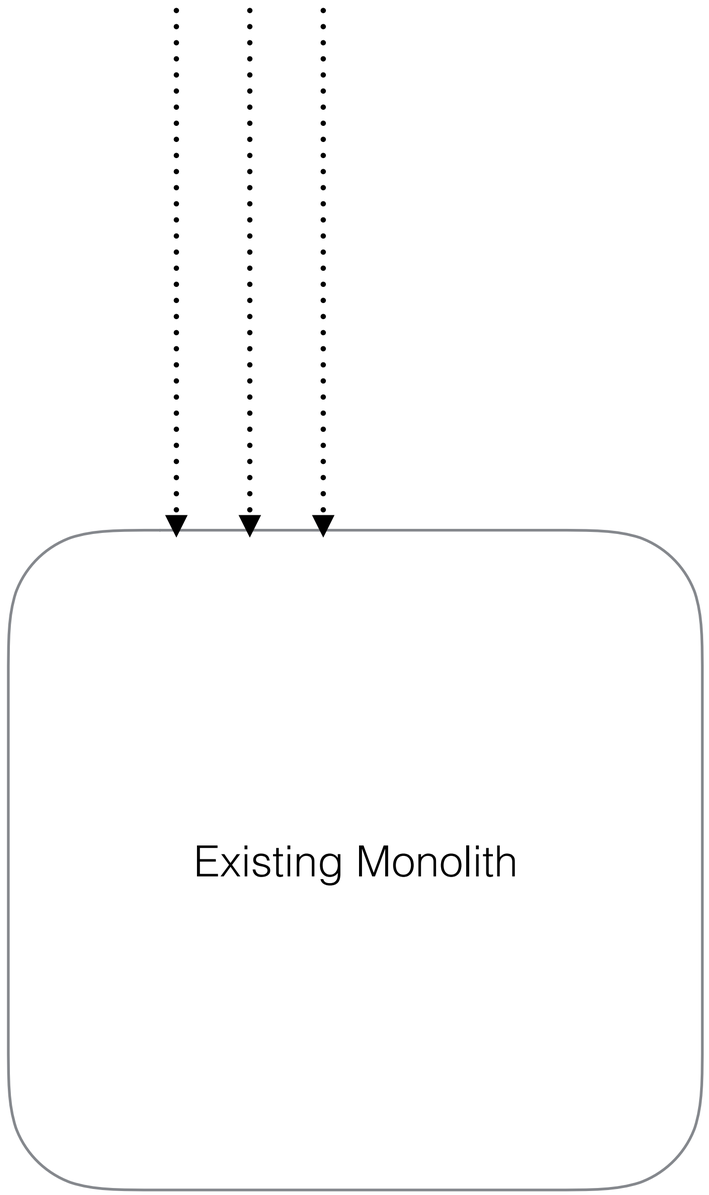
“MOVING” FUNCTIONALITY?

It might be copy and paste

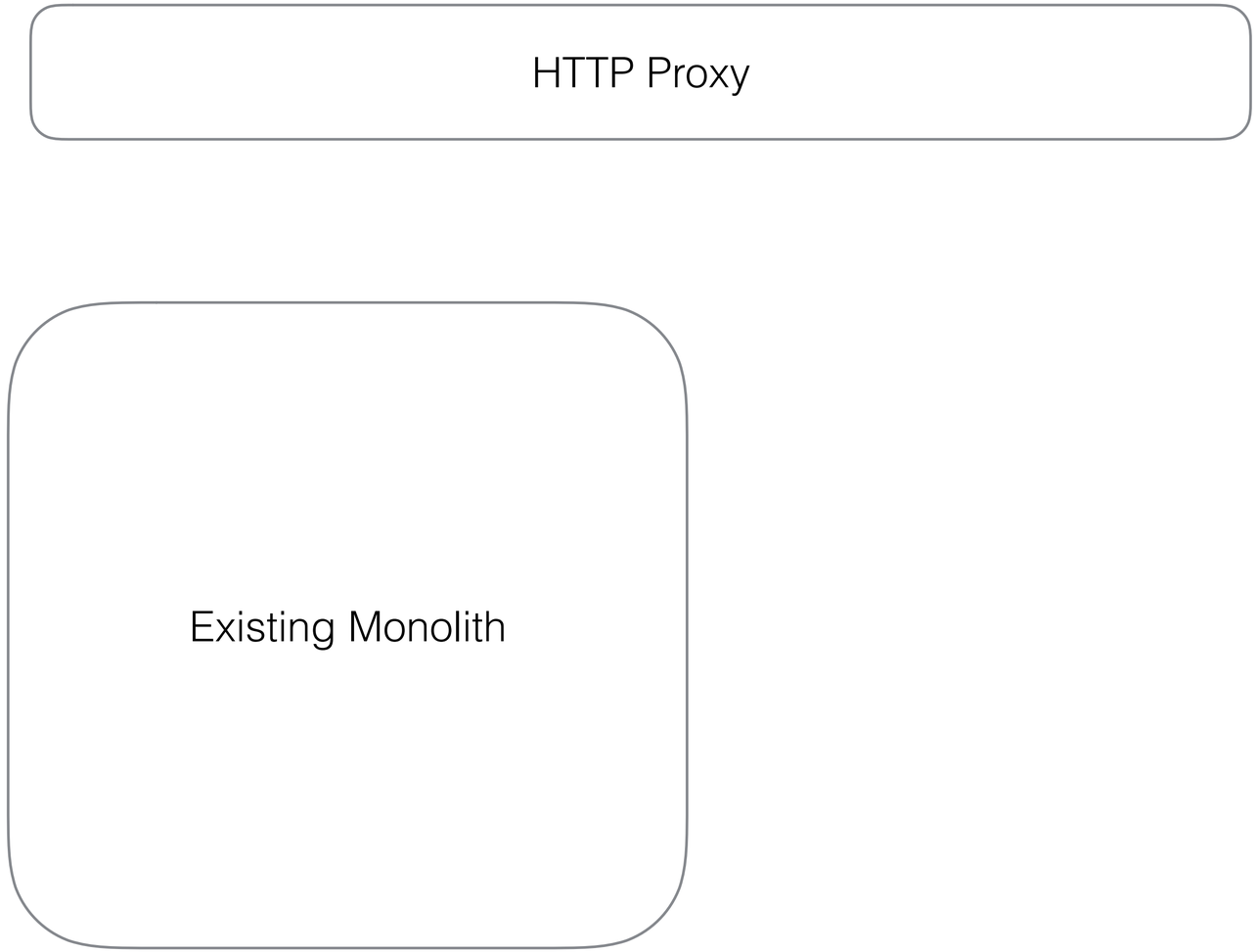
More likely is a total or partial rewrite

HTTP PROXY

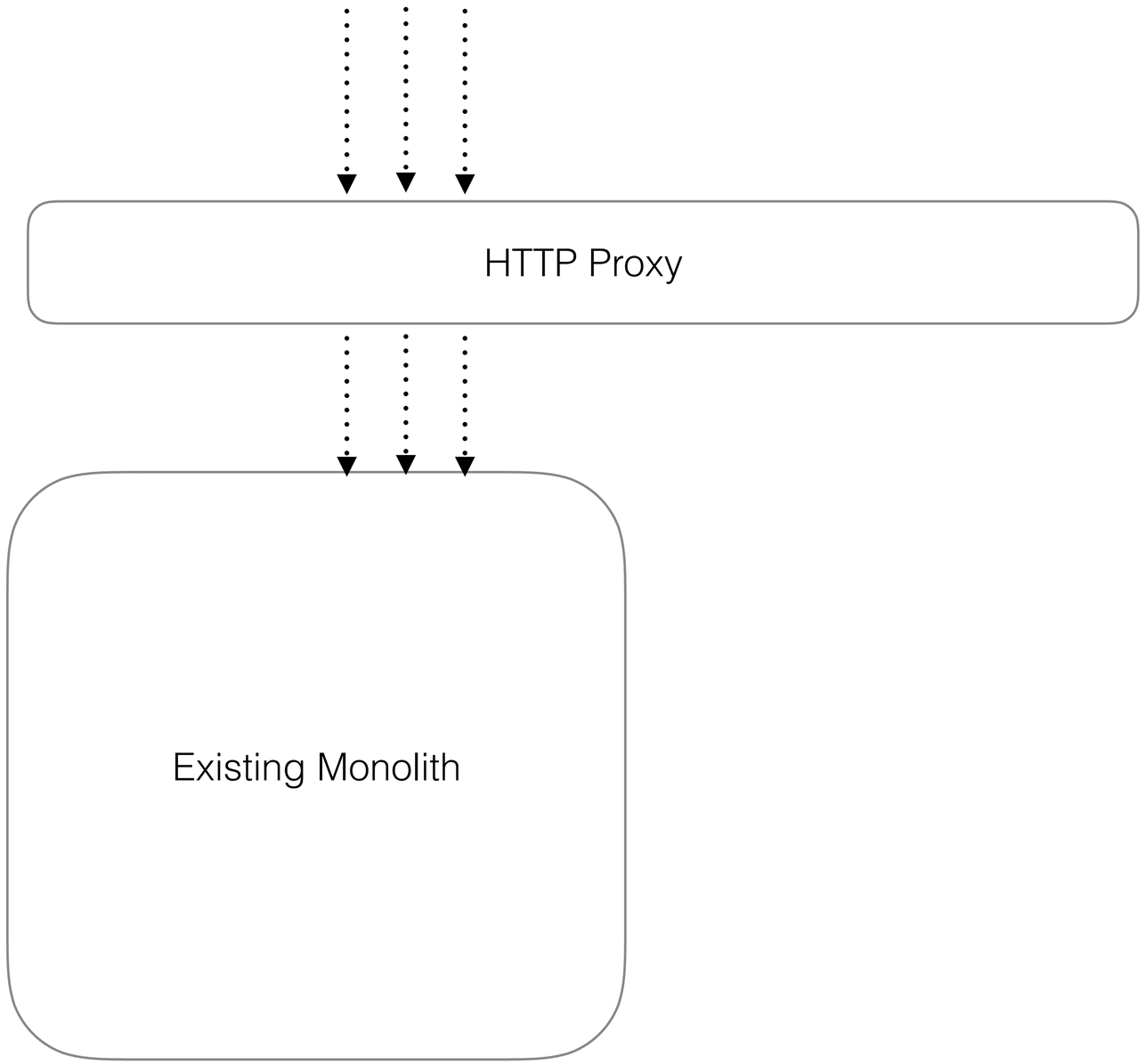
HTTP PROXY



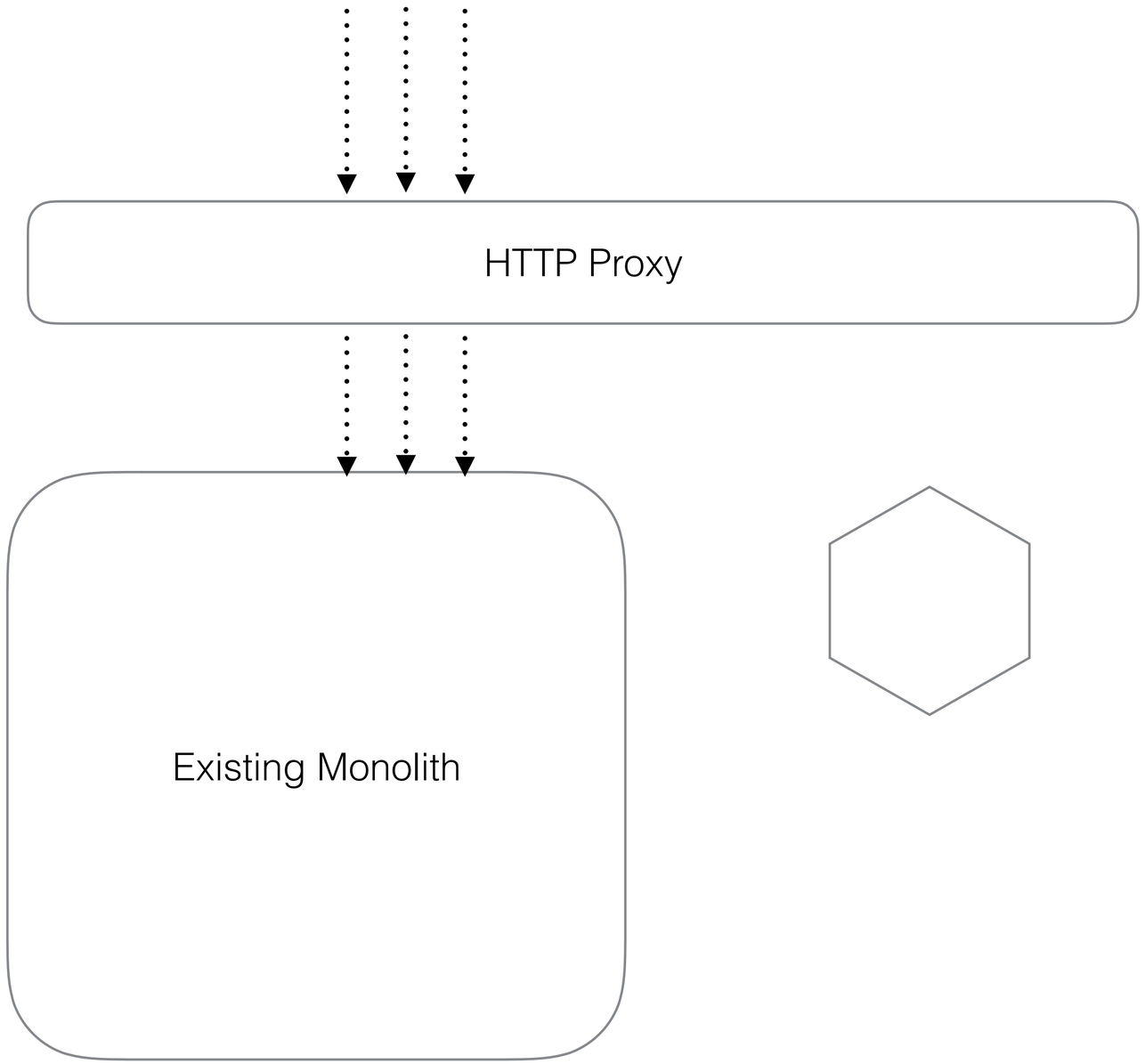
HTTP PROXY



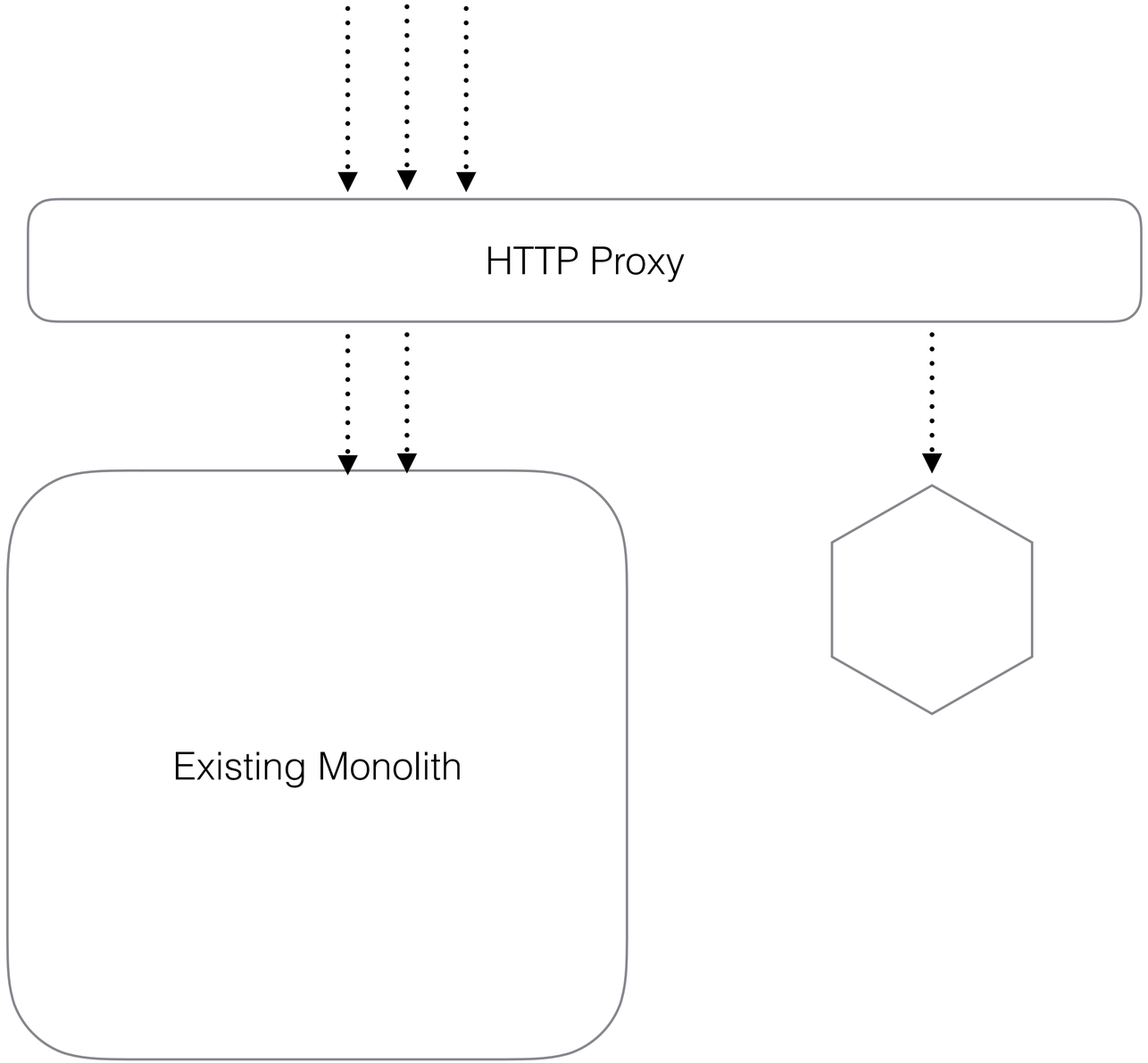
HTTP PROXY



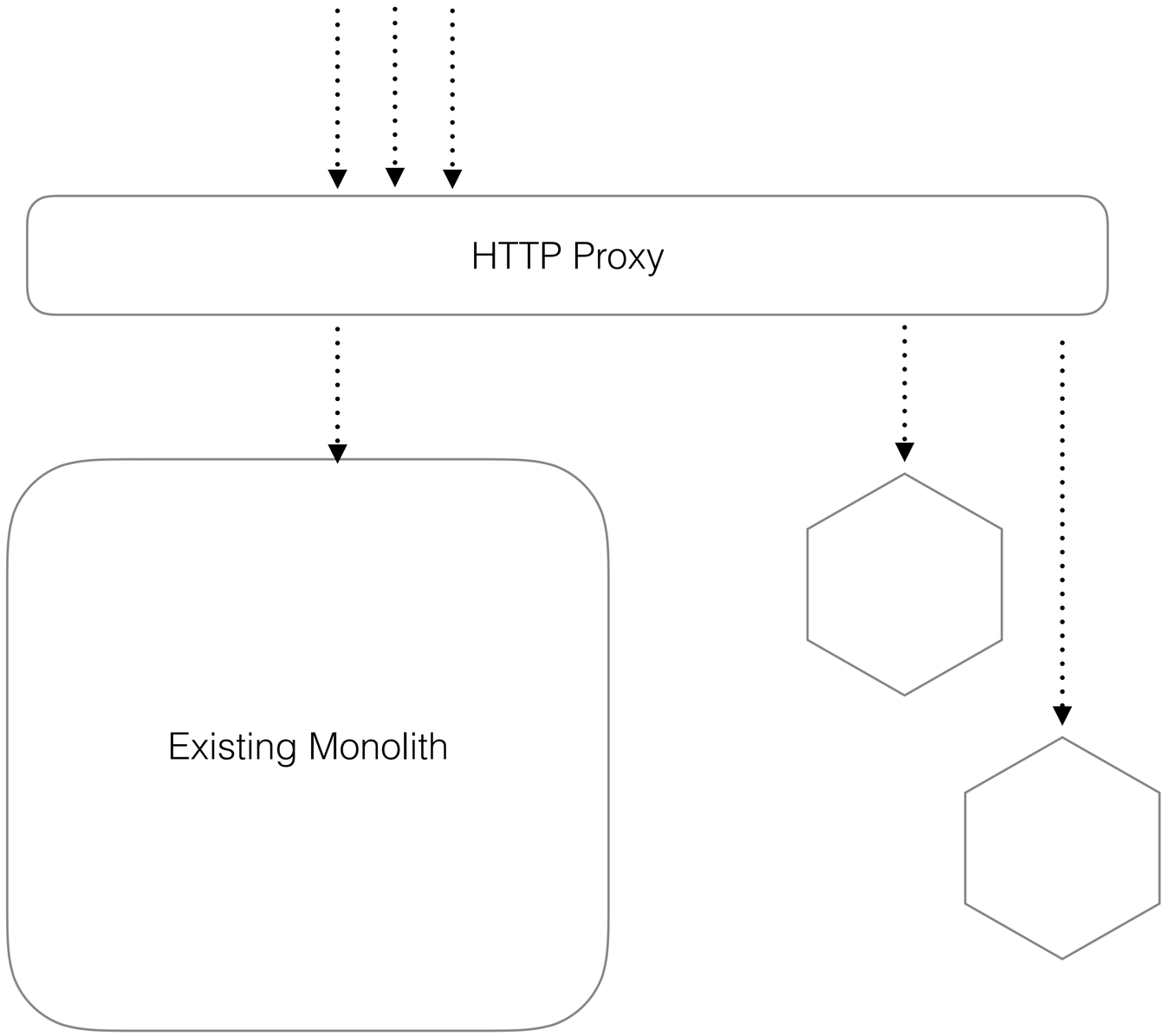
HTTP PROXY



HTTP PROXY



HTTP PROXY

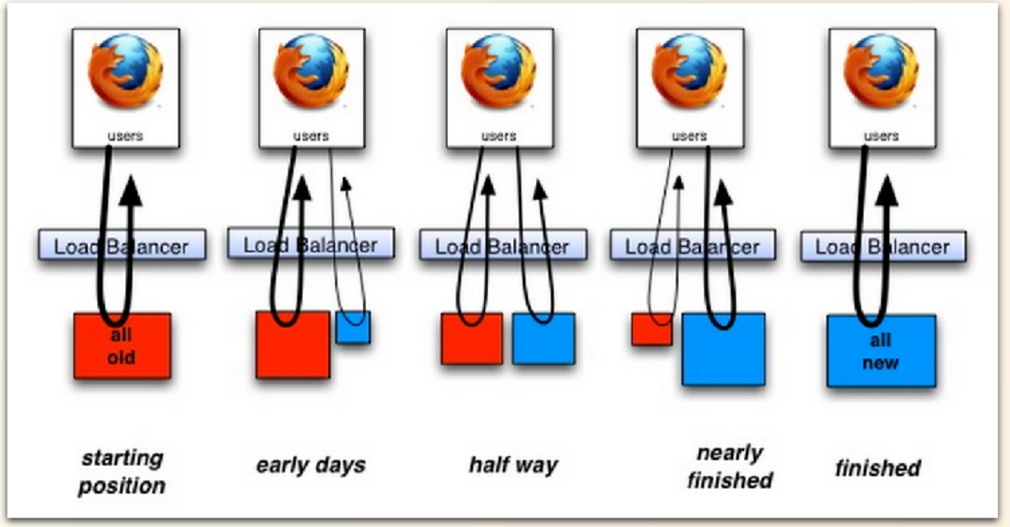


Legacy Application Strangulation : Case Studies

Strangler Applications

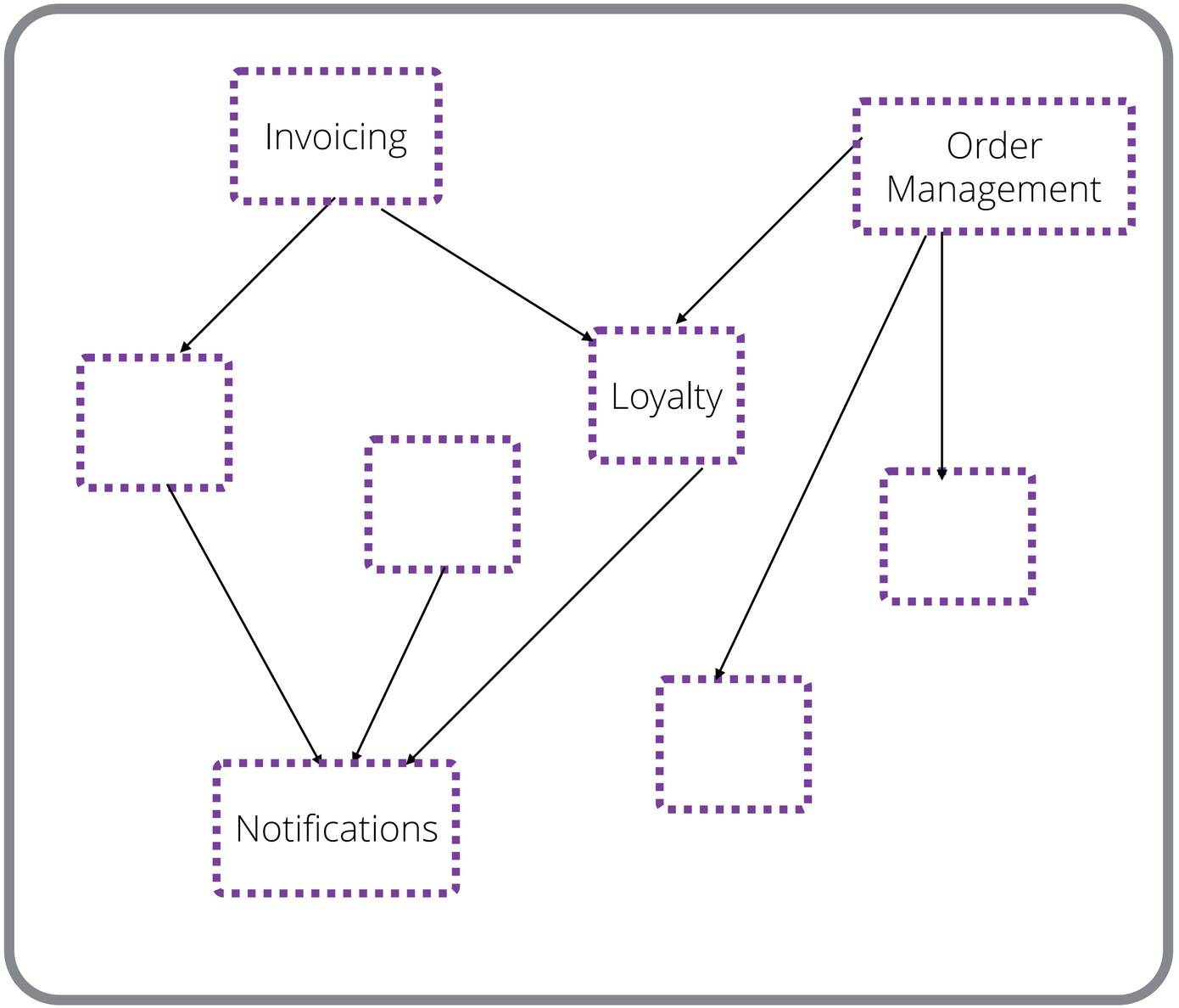
Martin Fowler wrote an article titled "Strangler Application" in mid 2004.

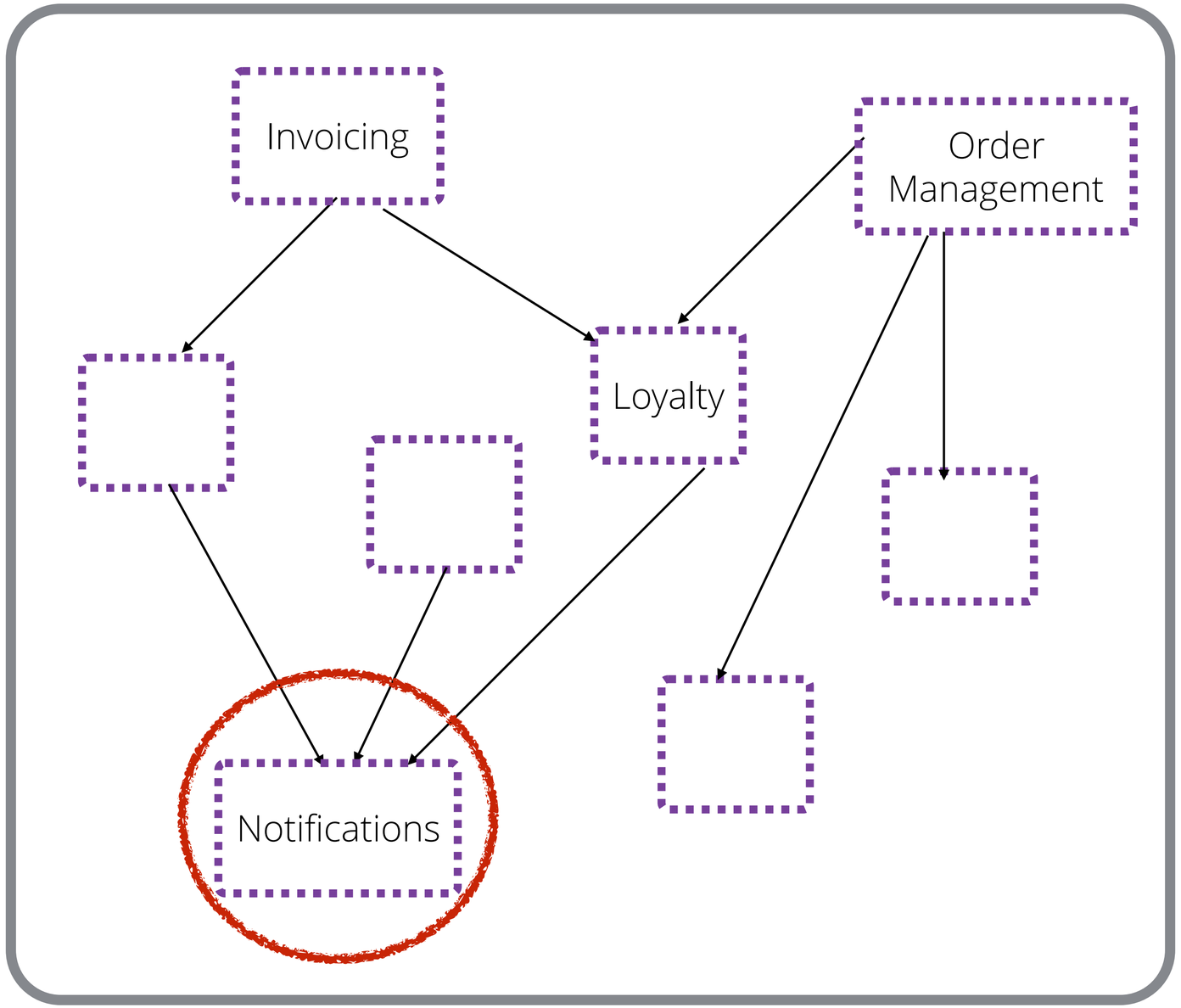
Strangulation of a legacy or undesirable solution is a safe way to phase one thing our for something better, cheaper, or more expandable. You make something new that obsoletes a small percentage of something old, and put them live together. You do some more work in the same style, and go live again (rinse, repeat). Here's a view of that (for web-apps):

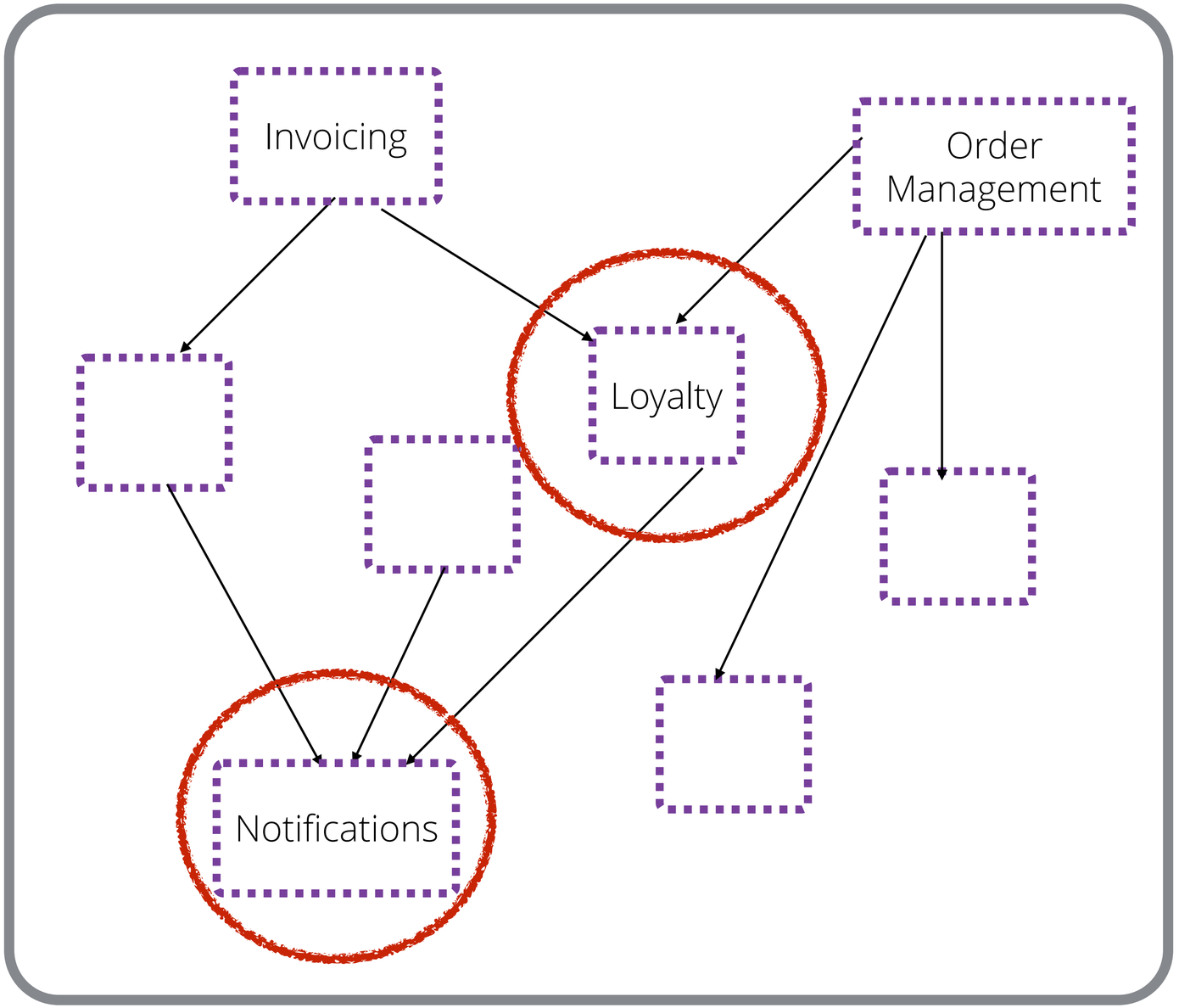


You could migrate all functionality from an old technology solution to new one in a series of releases that focussed on nothing else. Some companies will do that as there is a lot of sense to getting your house in order before doing anything else. However people outside the developer team may see that as a non-productive period, that could lengthen at any time, if it were asked for at all. People paying for that will notice, and may object. I mean execs, the board, or shareholders looking at the balance sheet.

http://paulhammant.com/2013/07/14/legacy-application-strangulation-case-studies/

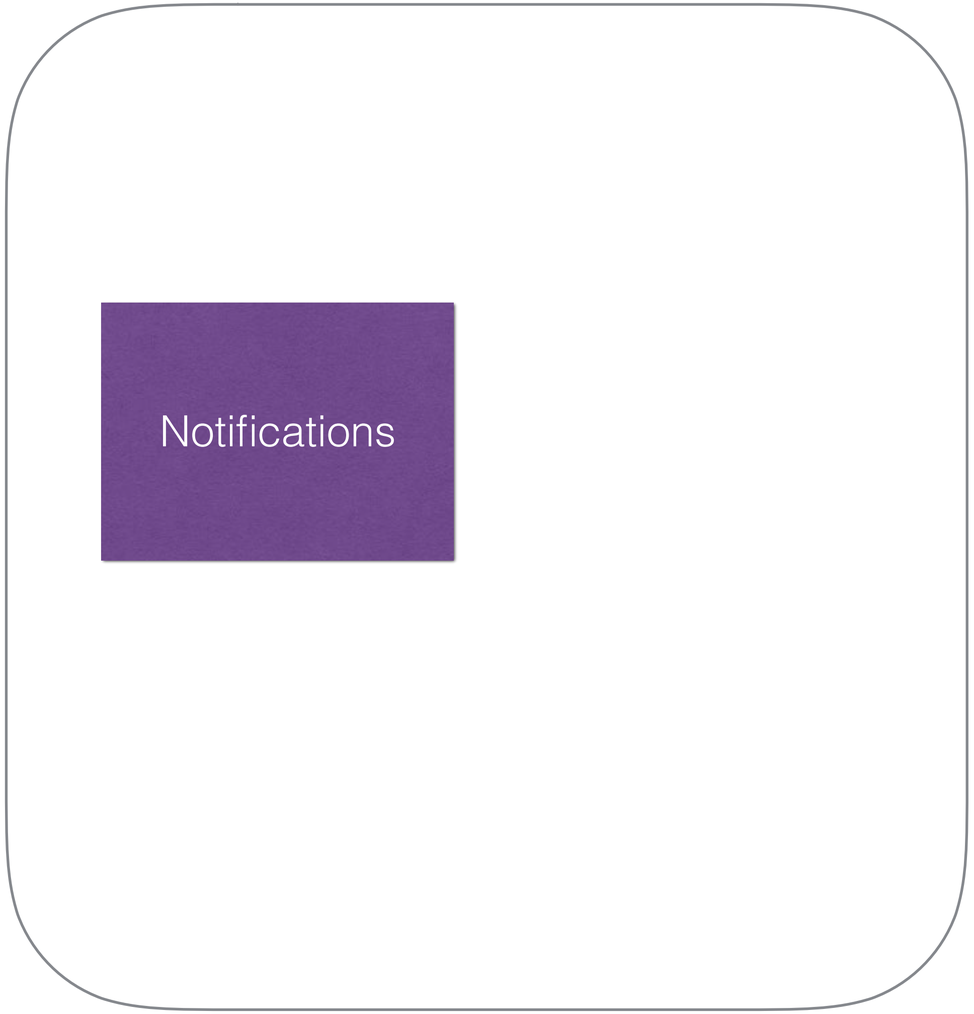

Invoicing
Order Management
Loyalty
Notifications


Invoicing
Order Management
Loyalty
Notifications

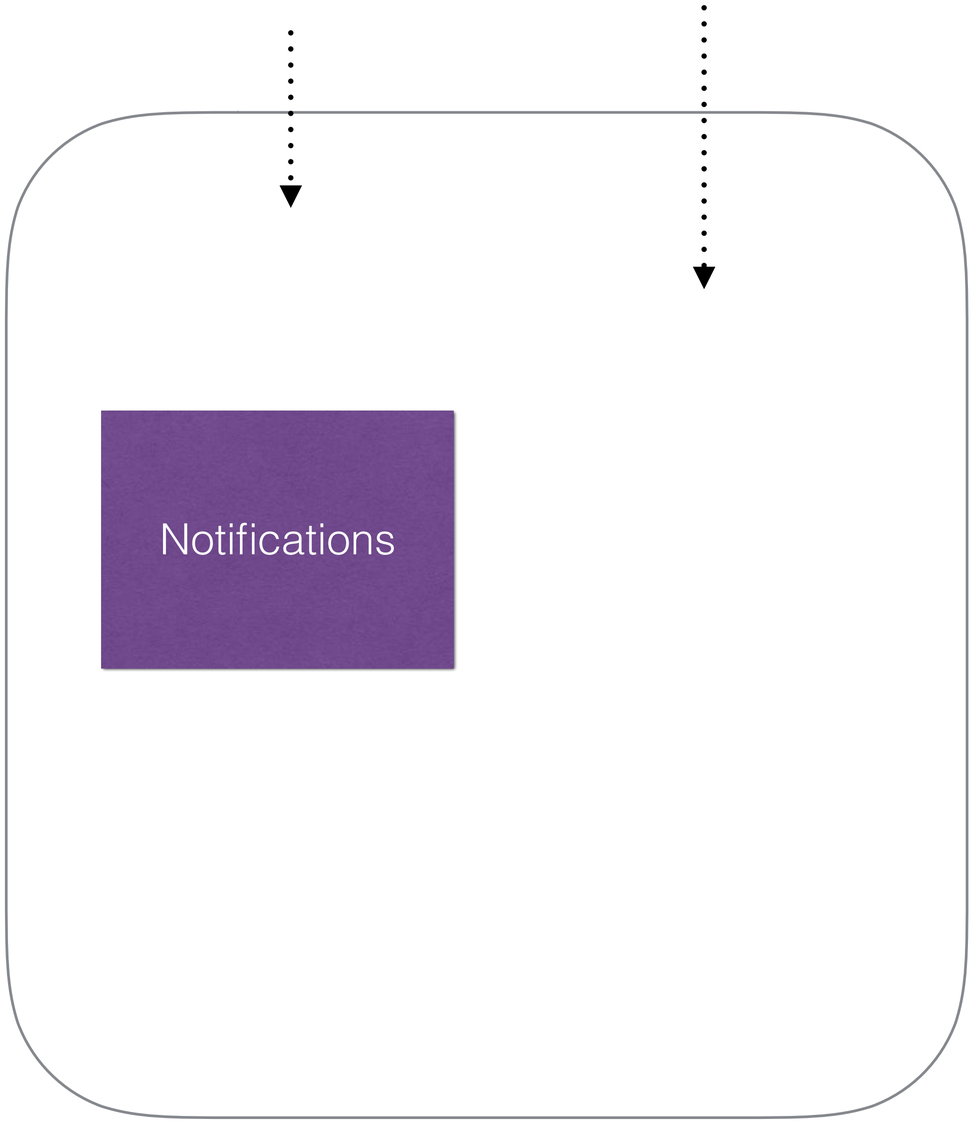

Invoicing
Order Management
Loyalty
Notifications

Branch By Abstraction

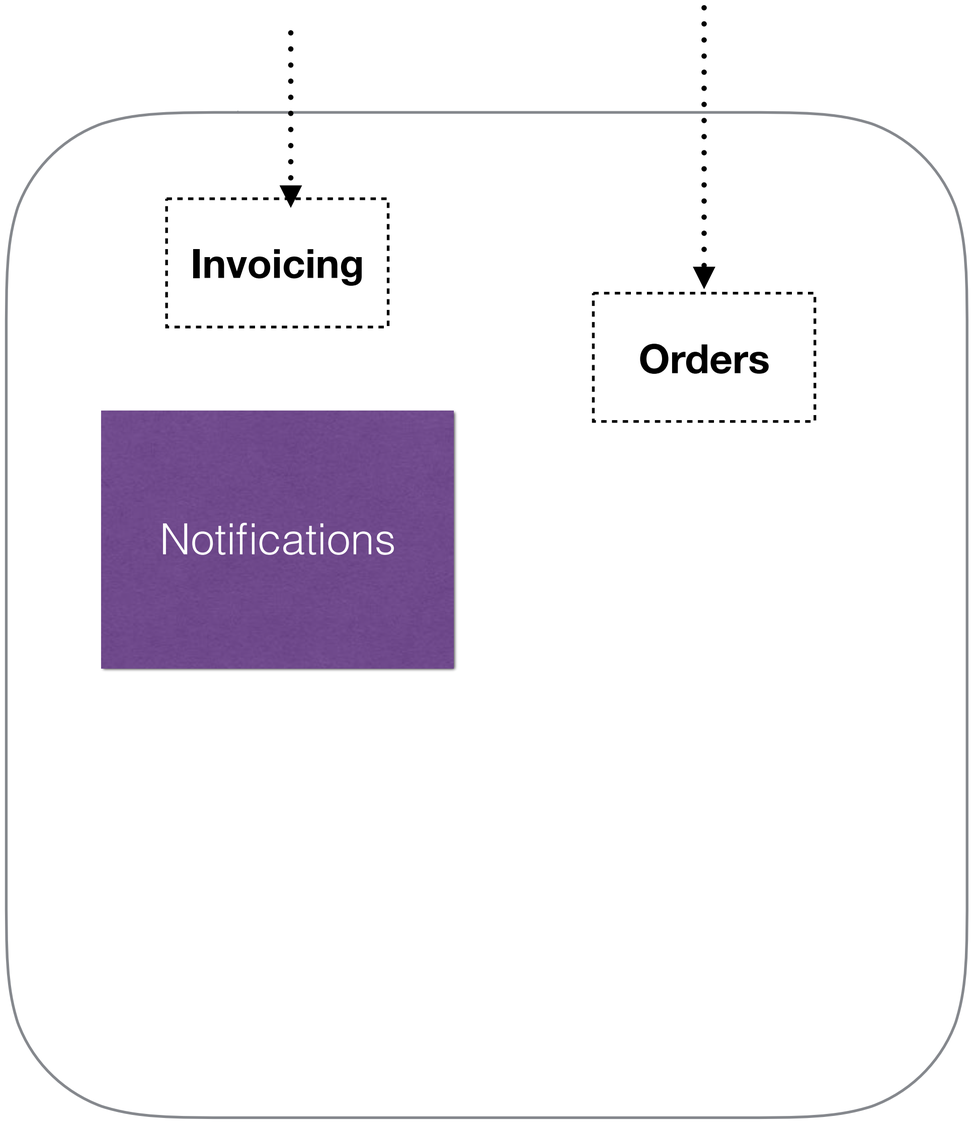
BRANCH BY ABSTRACTION



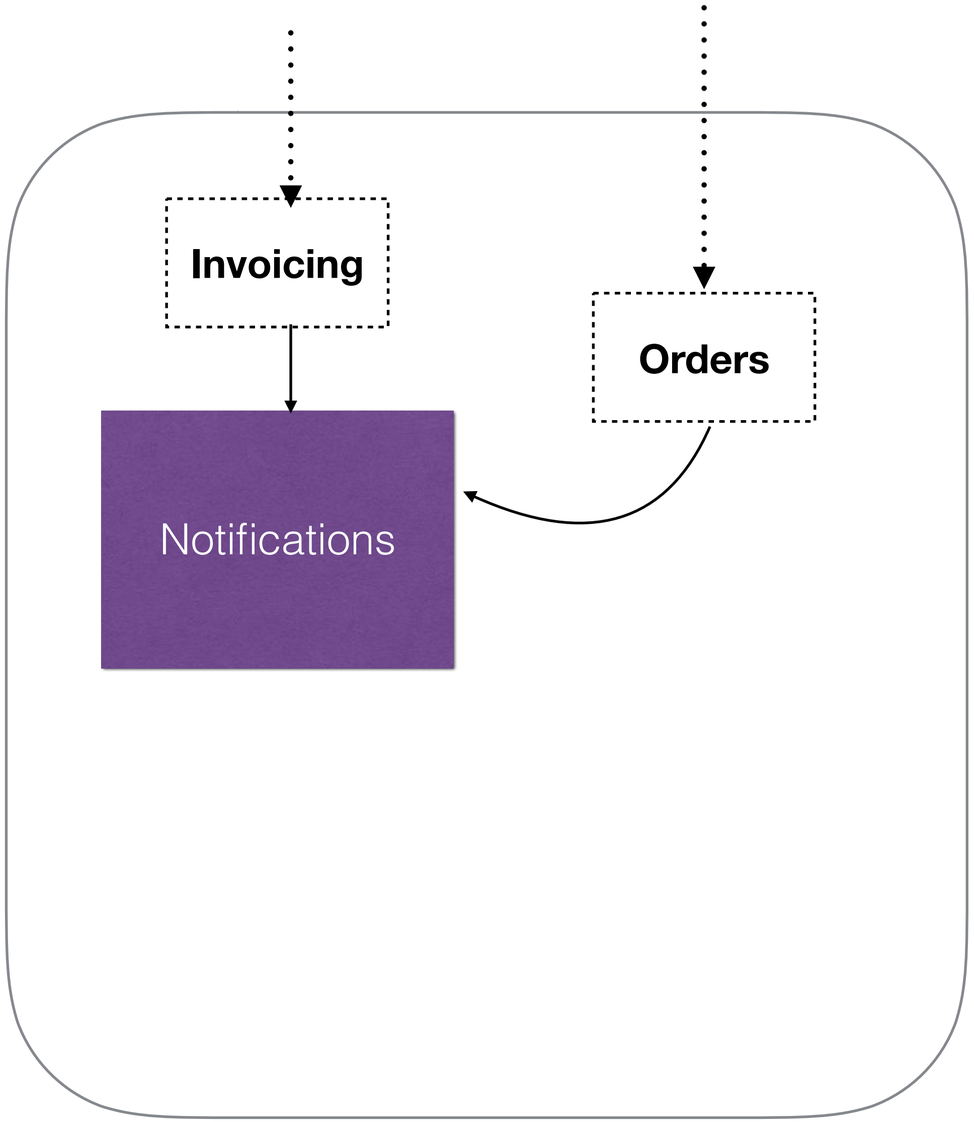
BRANCH BY ABSTRACTION



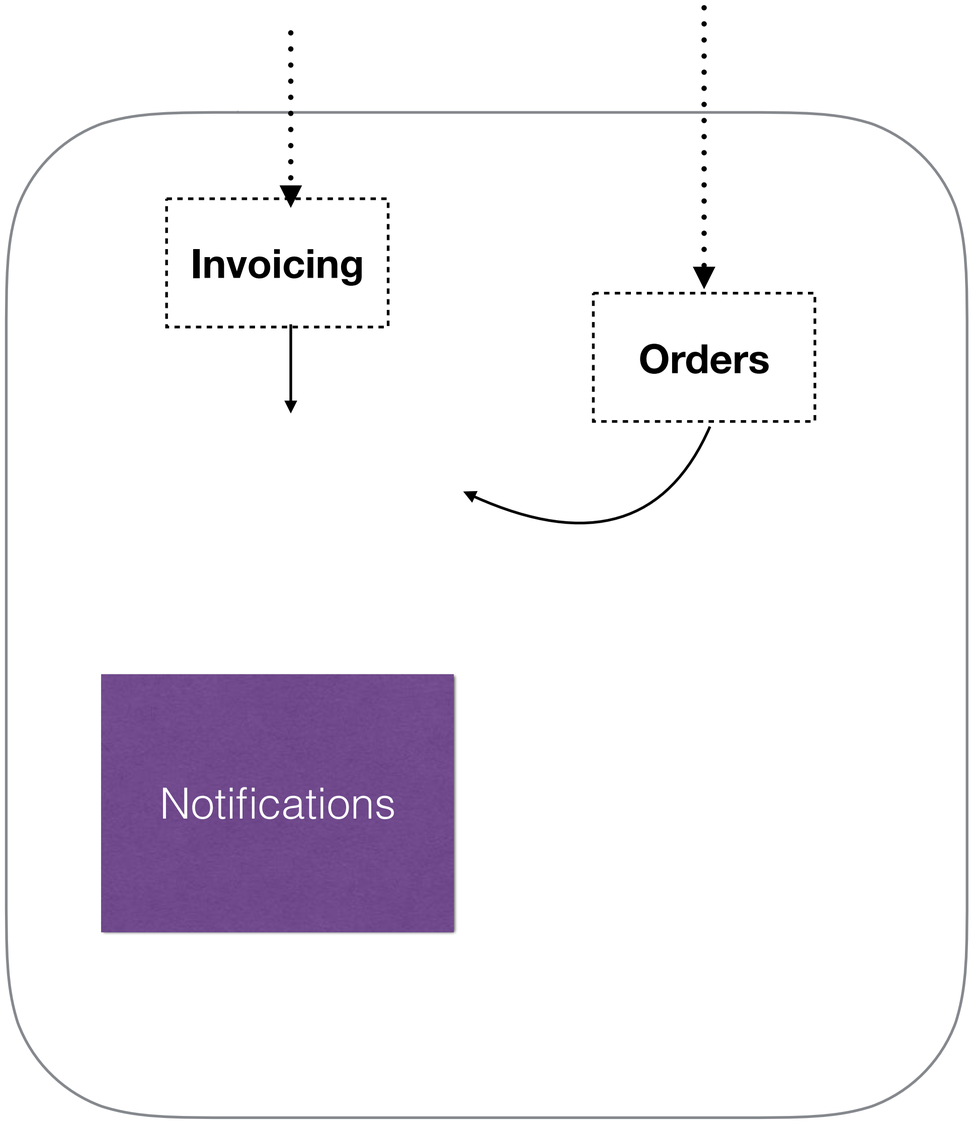
BRANCH BY ABSTRACTION



BRANCH BY ABSTRACTION

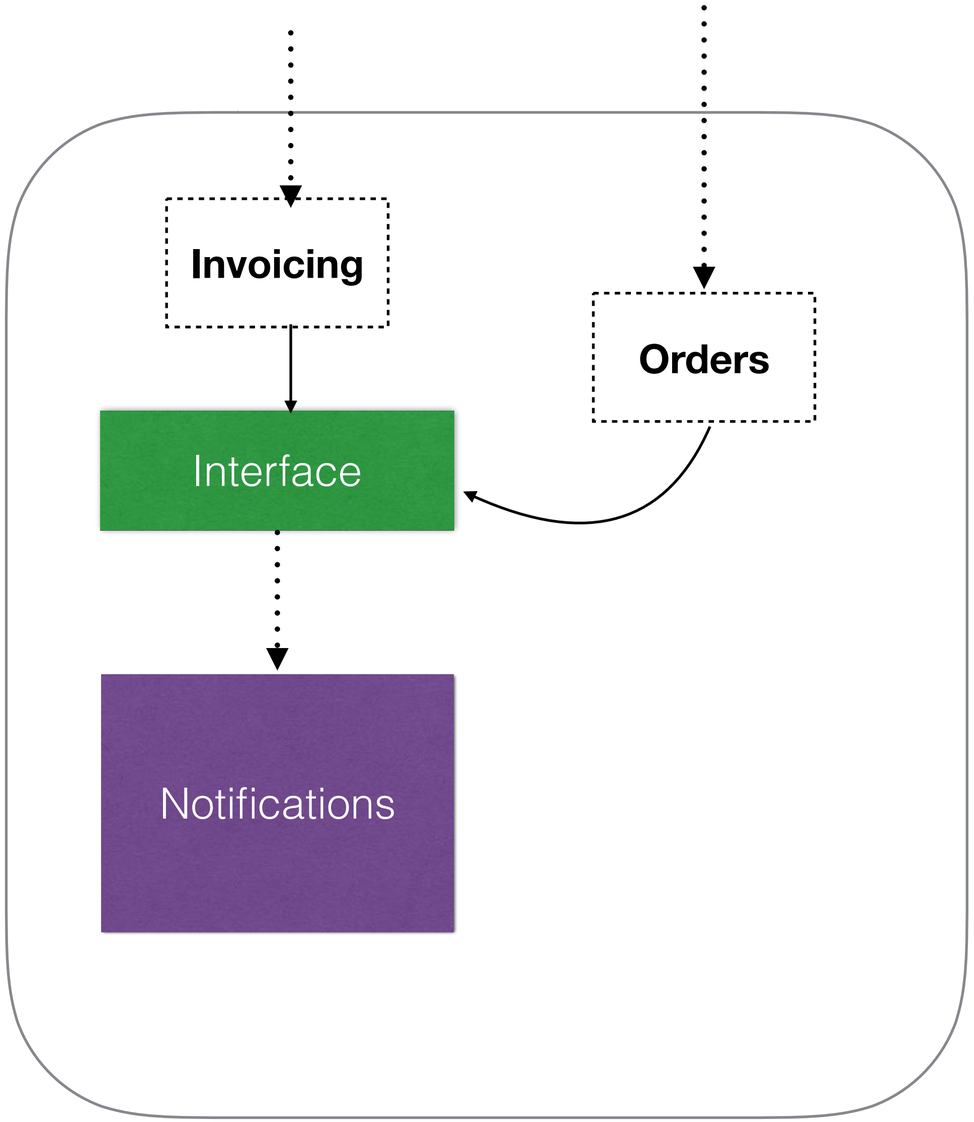


BRANCH BY ABSTRACTION

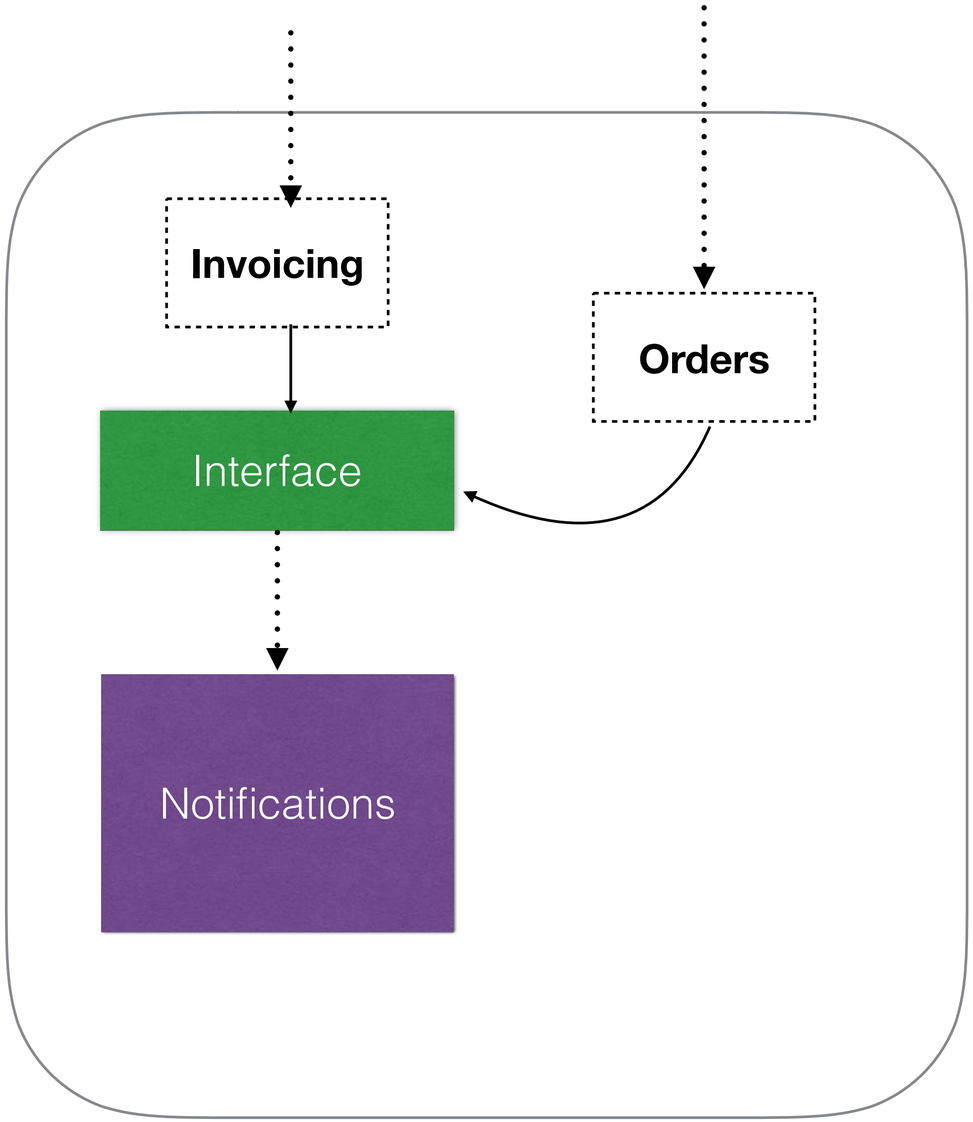


BRANCH BY ABSTRACTION

1. Create abstraction point



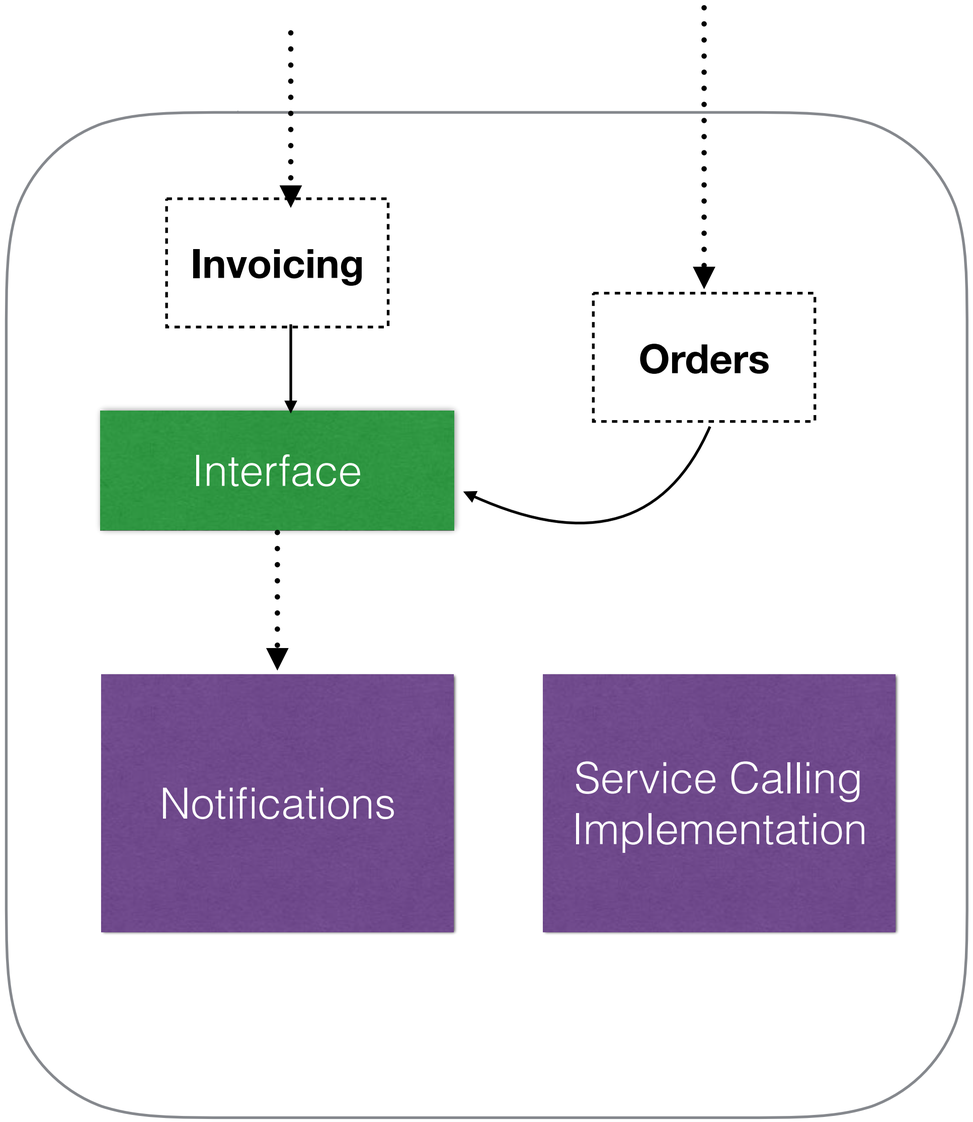
BRANCH BY ABSTRACTION



1. Create abstraction point

2. Start work on new service implementation

BRANCH BY ABSTRACTION



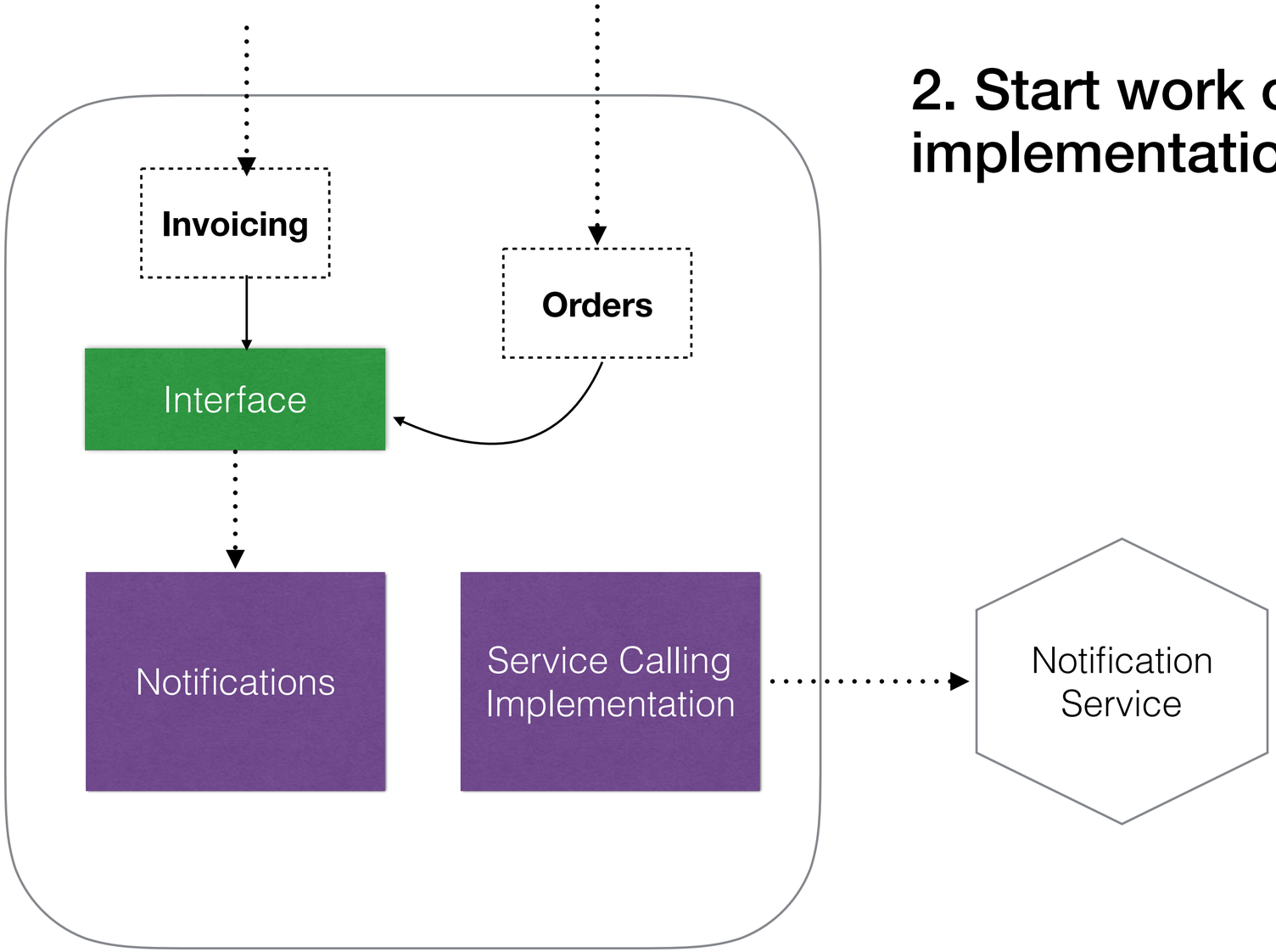
1. Create abstraction point

2. Start work on new service implementation

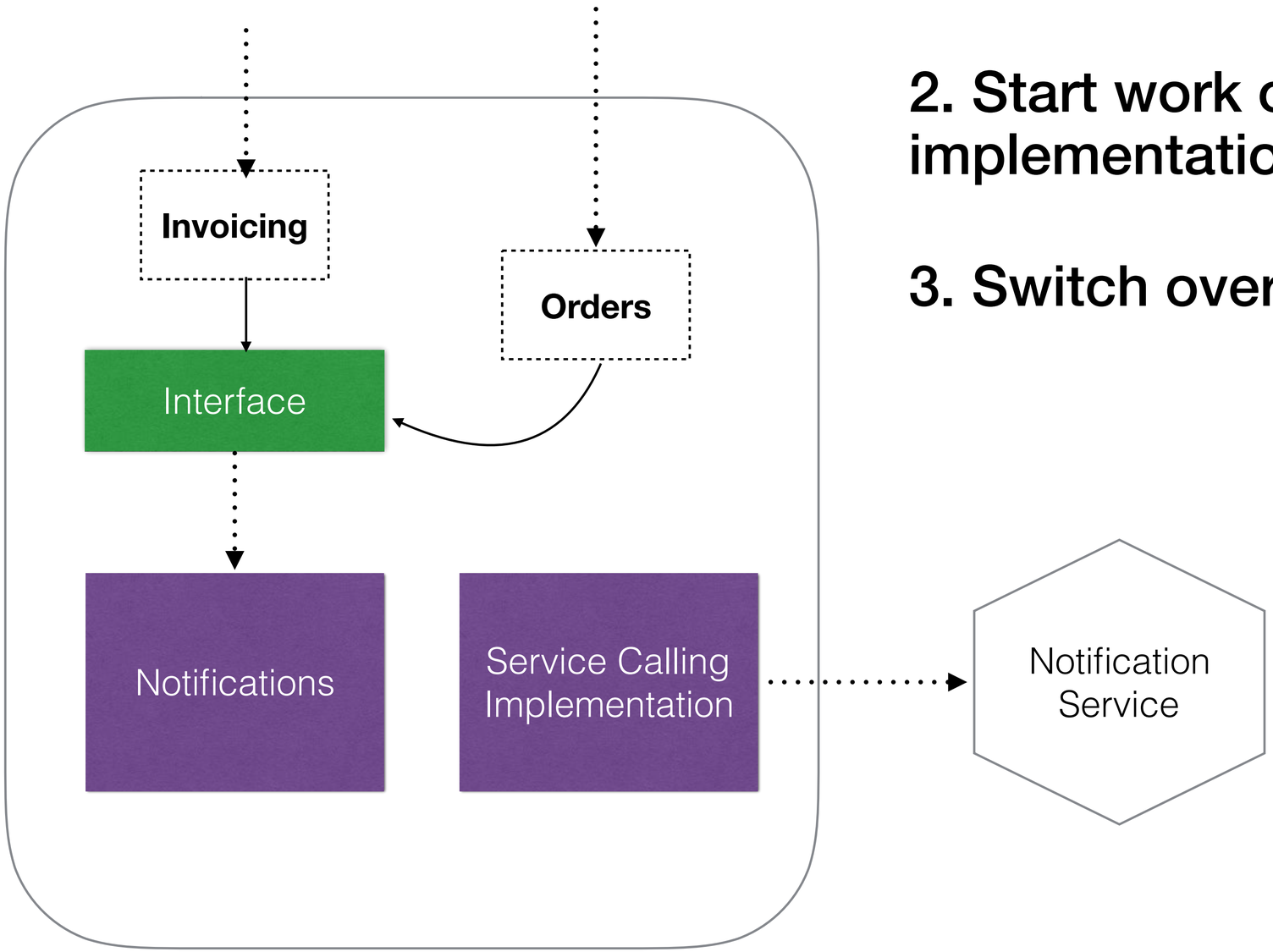
BRANCH BY ABSTRACTION

1. Create abstraction point

2. Start work on new service implementation



BRANCH BY ABSTRACTION

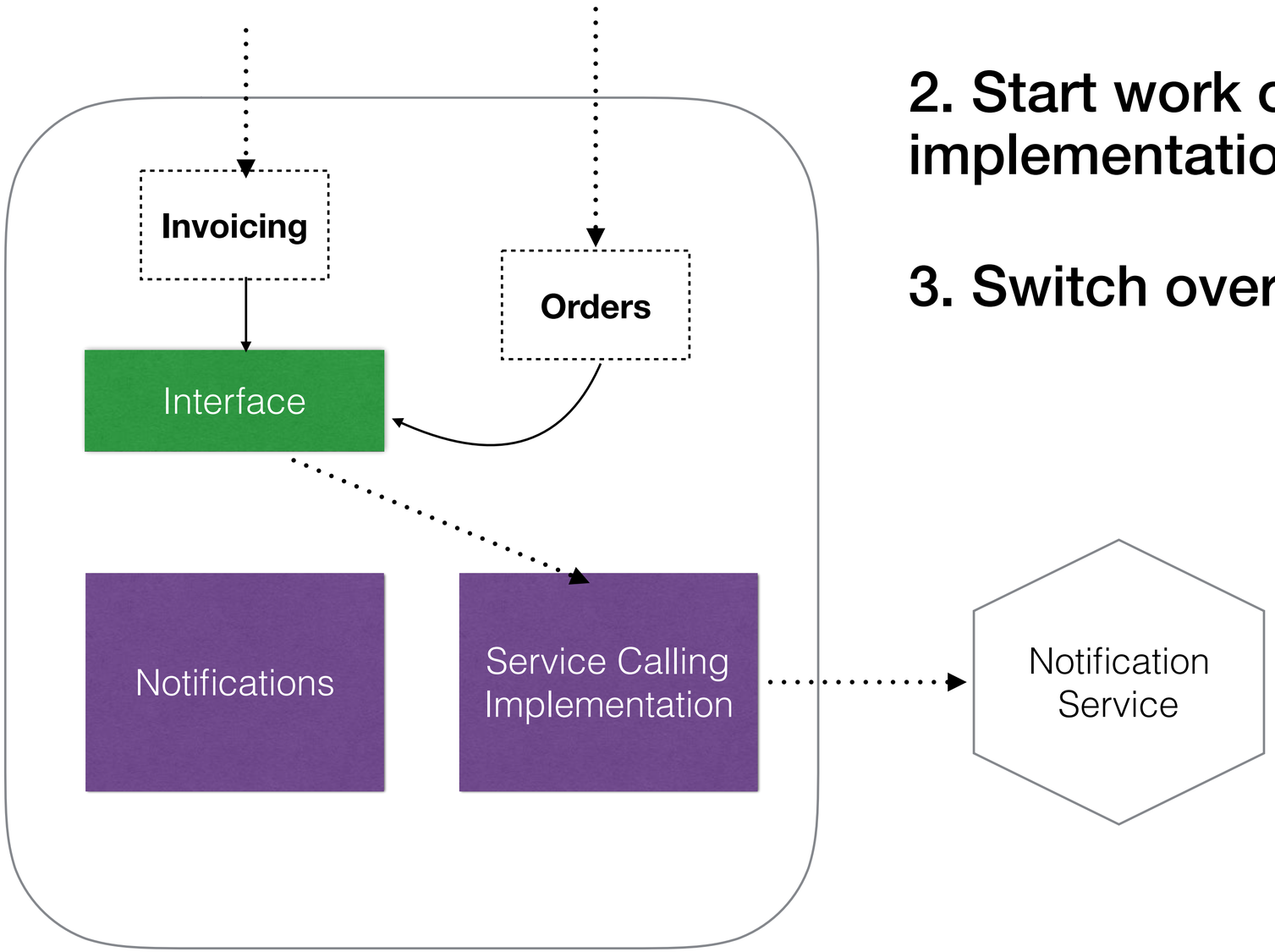


1. Create abstraction point

2. Start work on new service implementation

3. Switch over

BRANCH BY ABSTRACTION

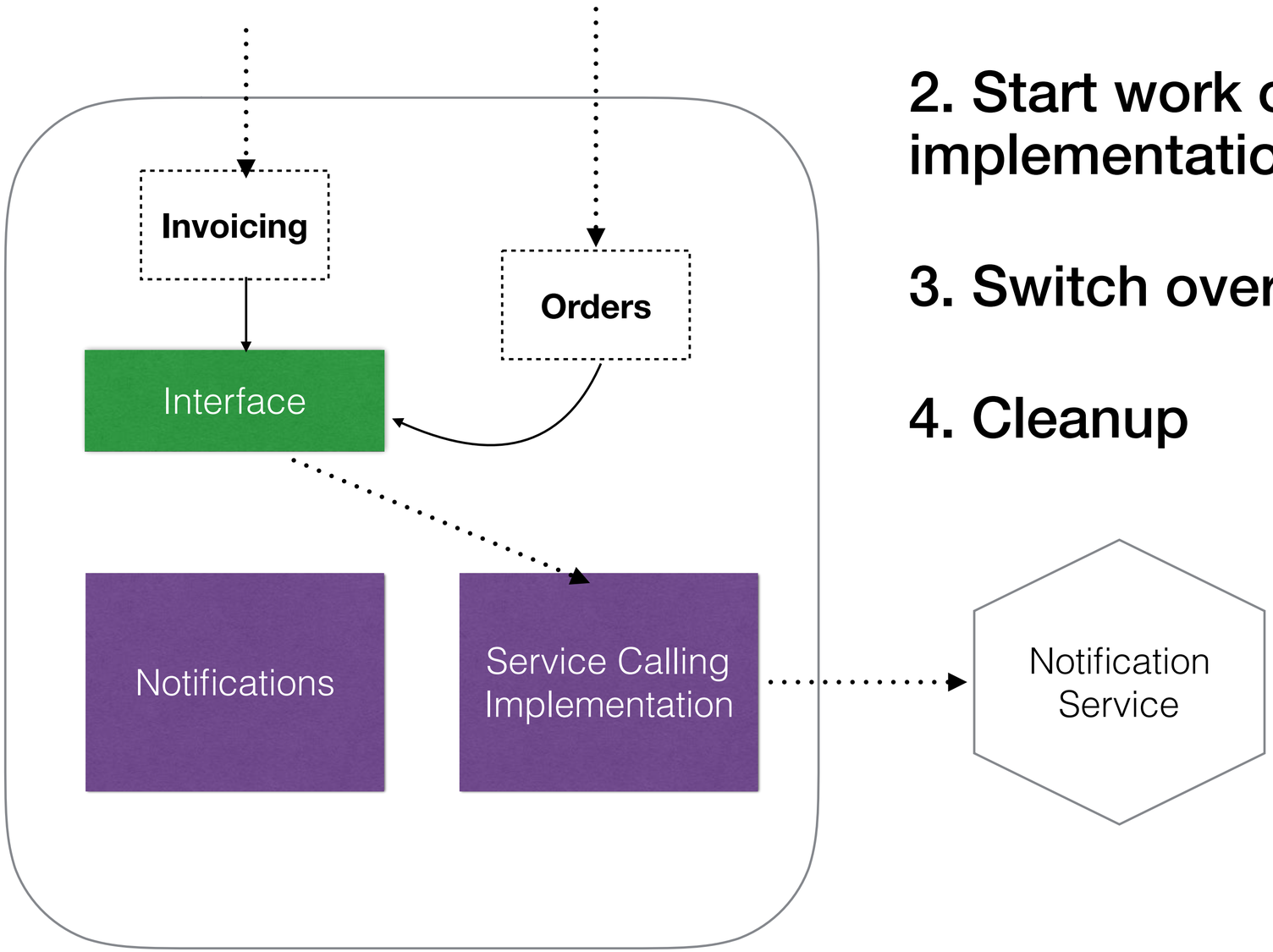


1. Create abstraction point

2. Start work on new service implementation

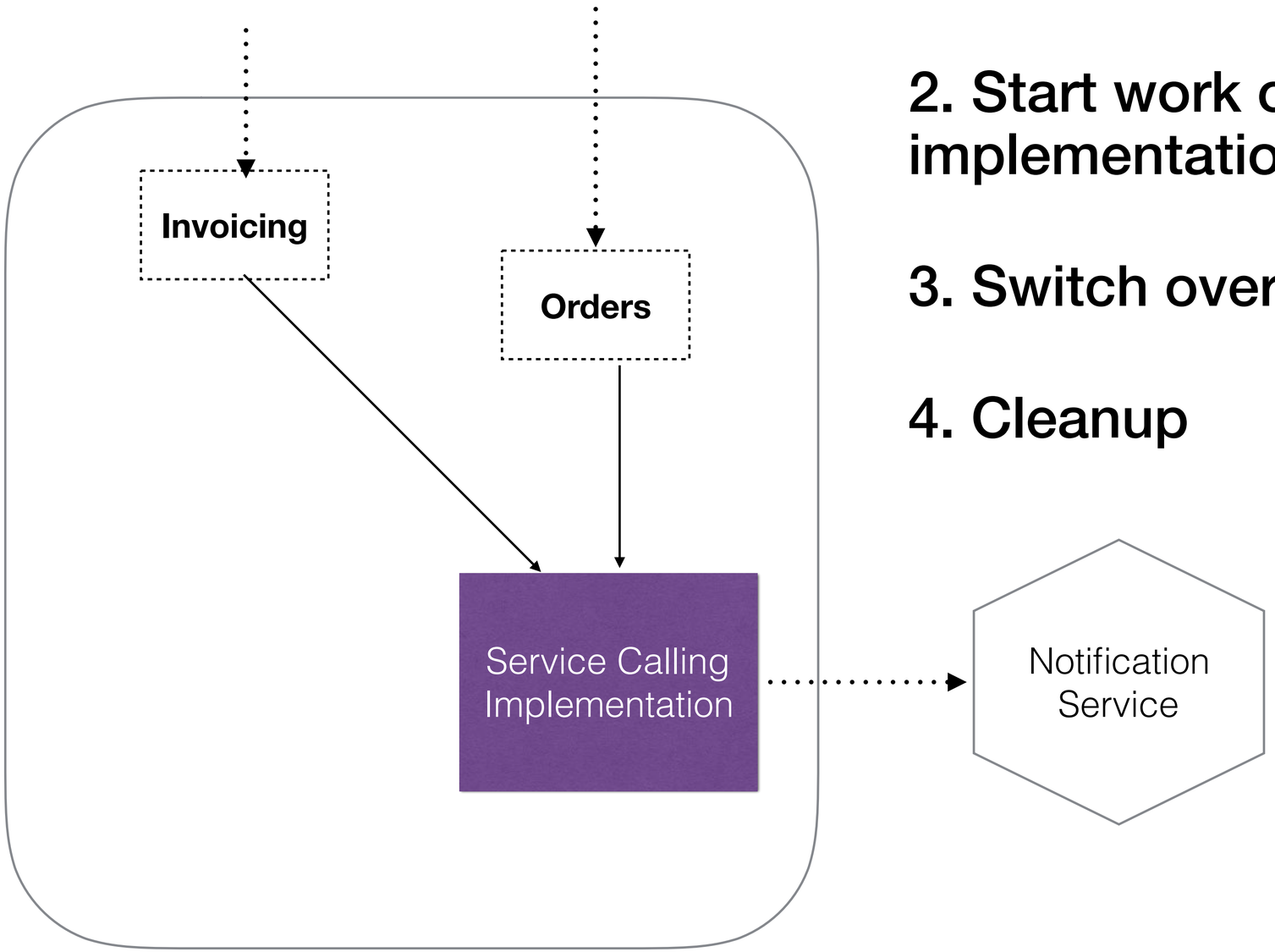
3. Switch over

BRANCH BY ABSTRACTION



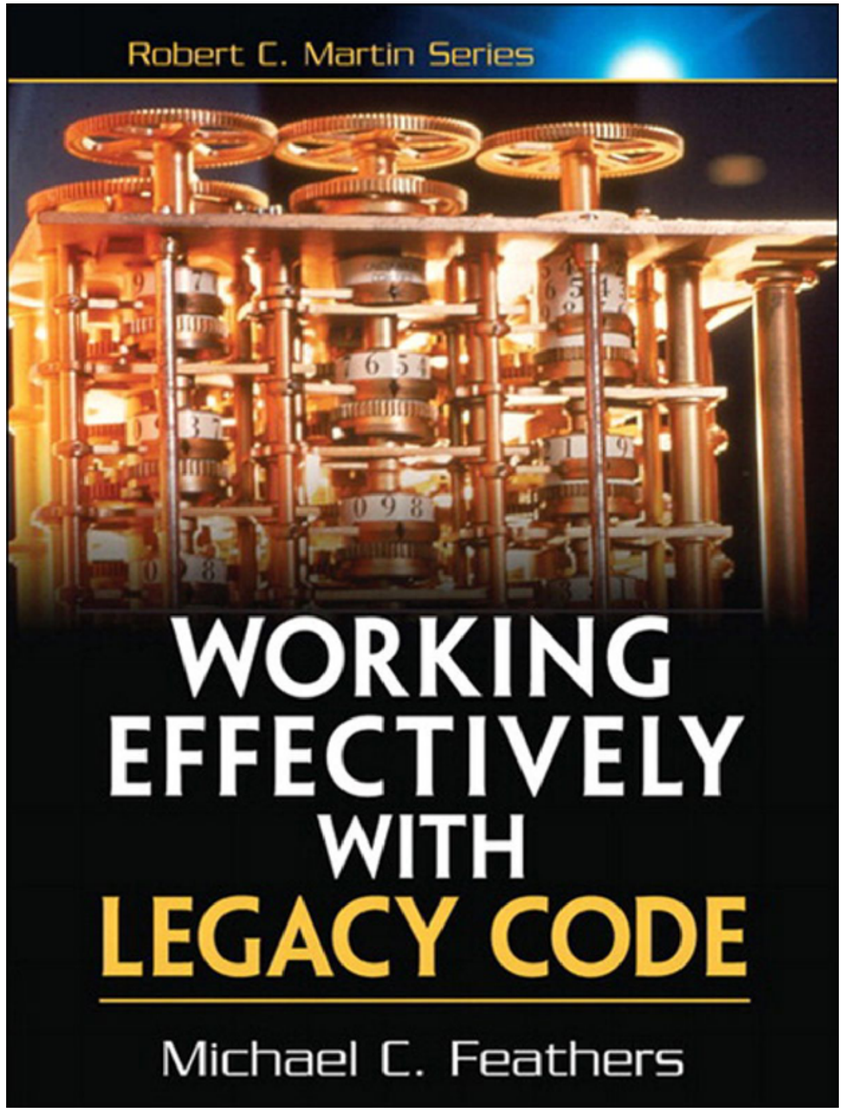
1. Create abstraction point
2. Start work on new service implementation
3. Switch over
4. Cleanup

BRANCH BY ABSTRACTION

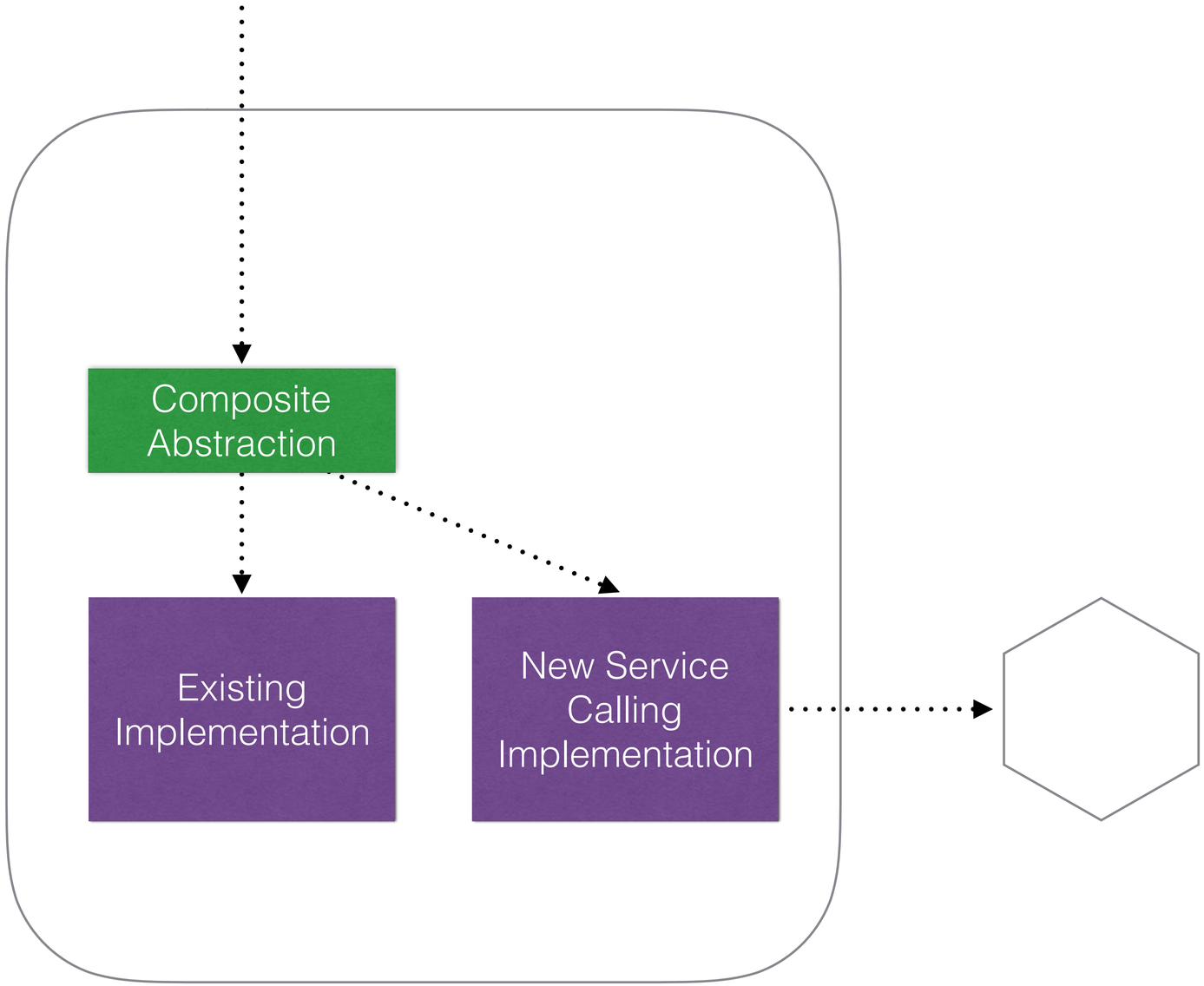


1. Create abstraction point
2. Start work on new service implementation
3. Switch over
4. Cleanup

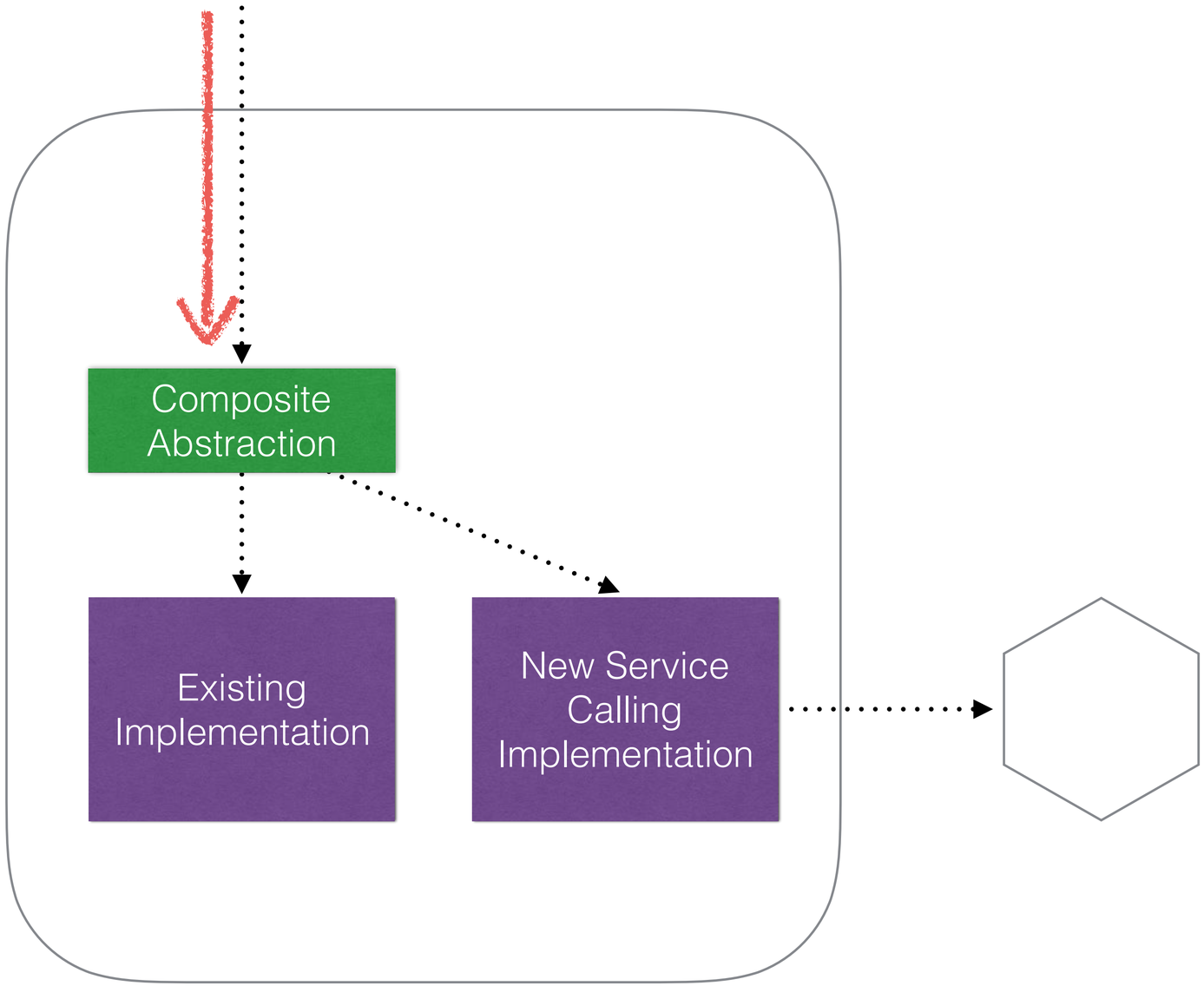
WORKING WITH LEGACY CODE



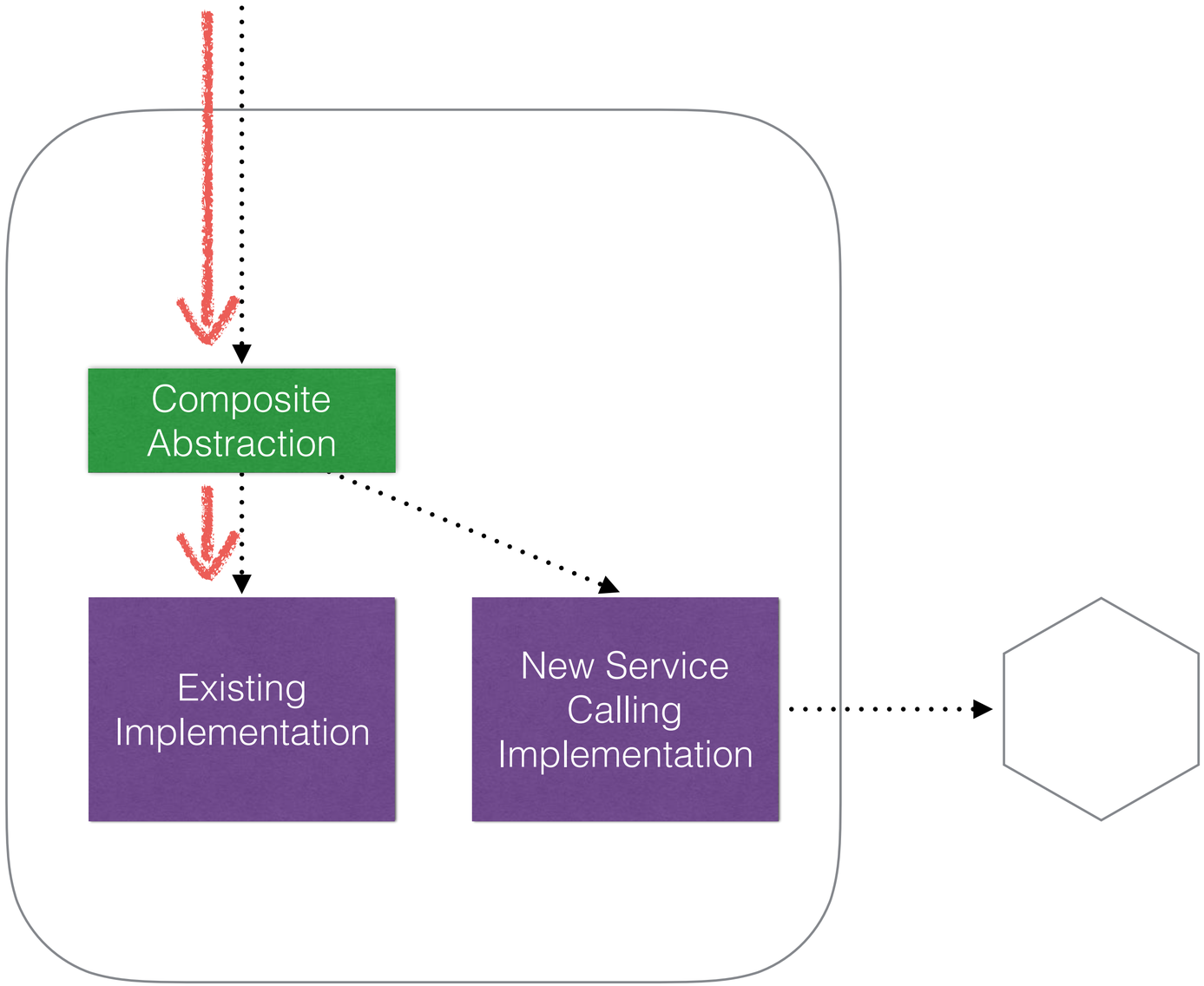
PARALLEL RUN



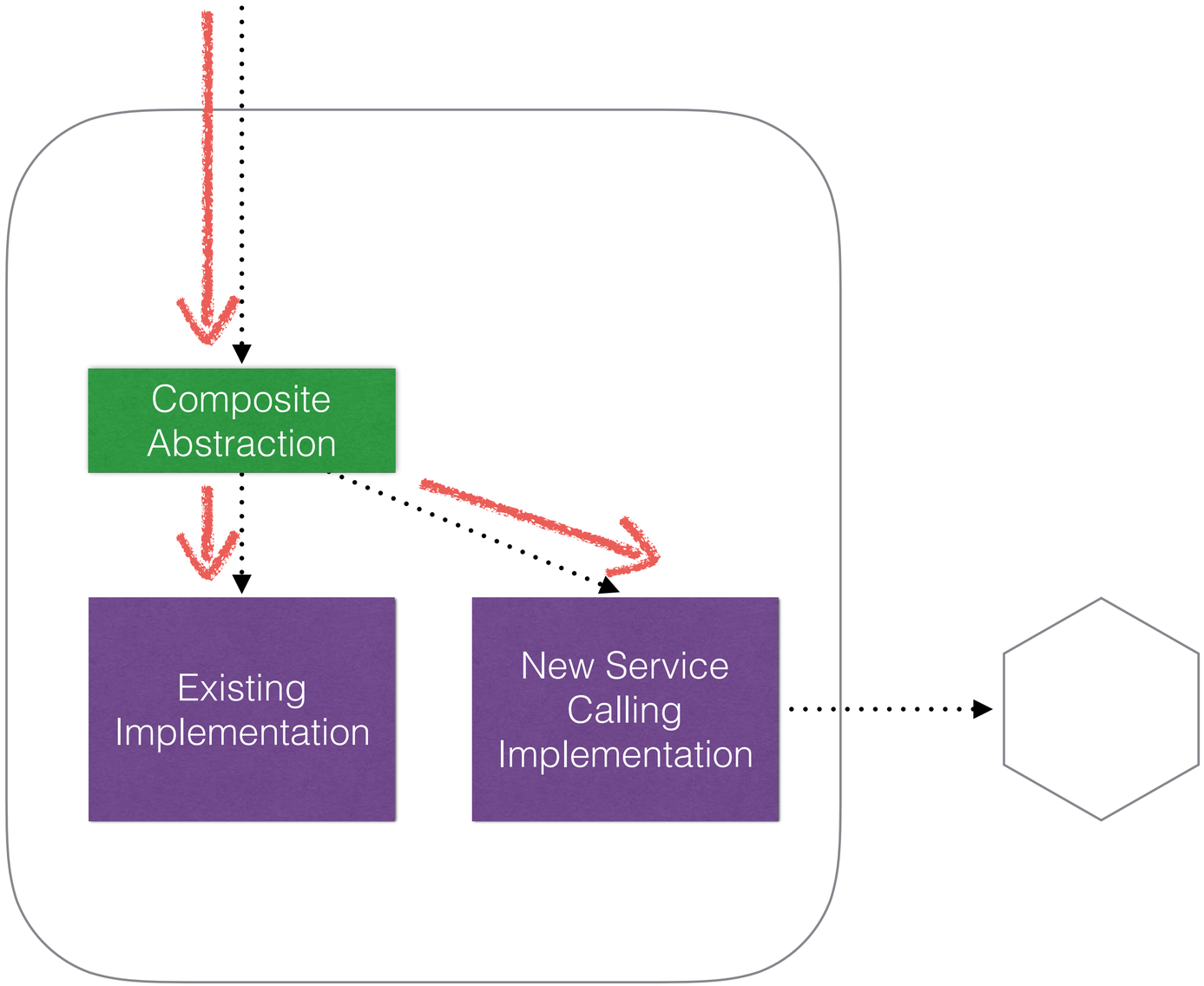
PARALLEL RUN



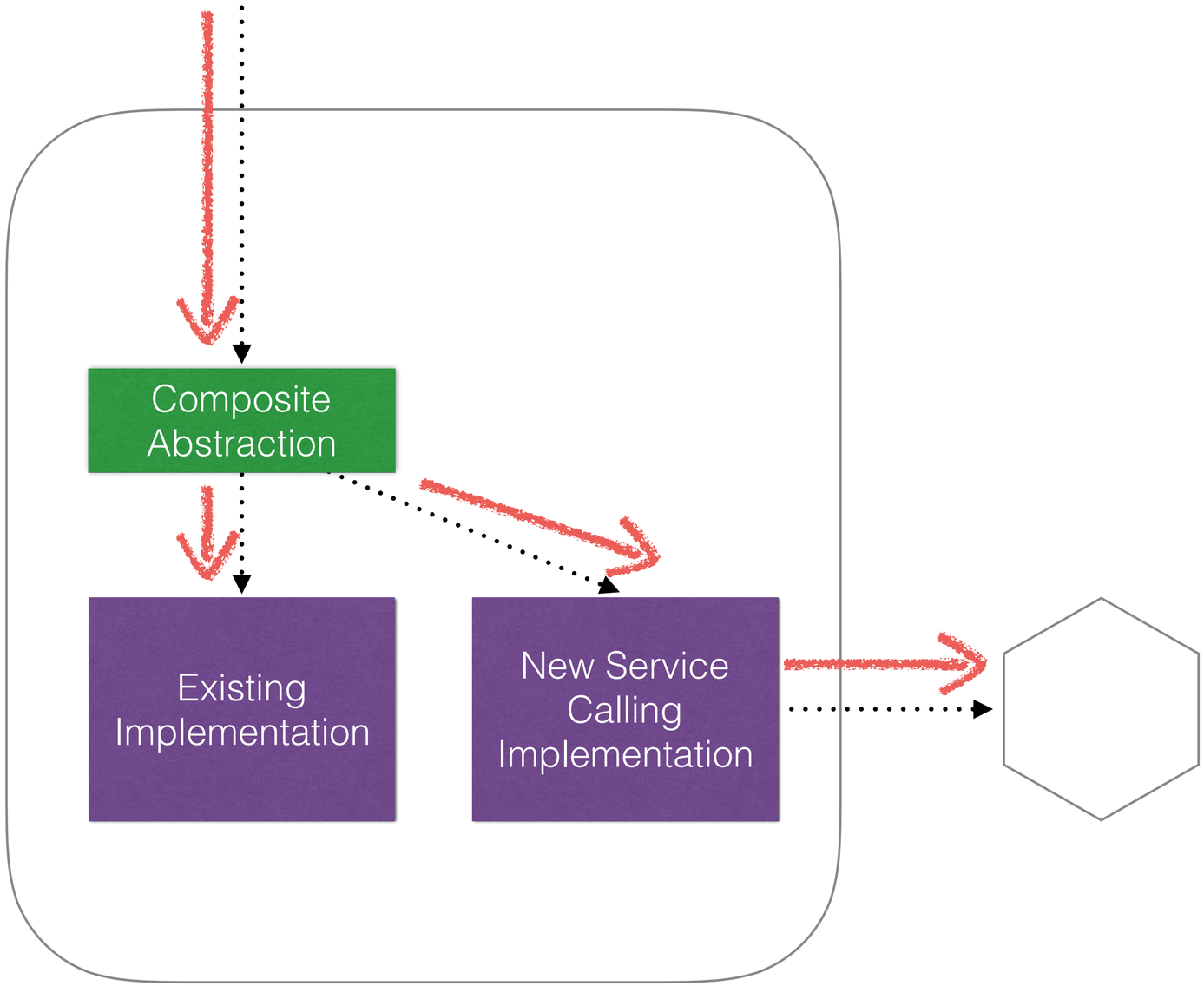
PARALLEL RUN



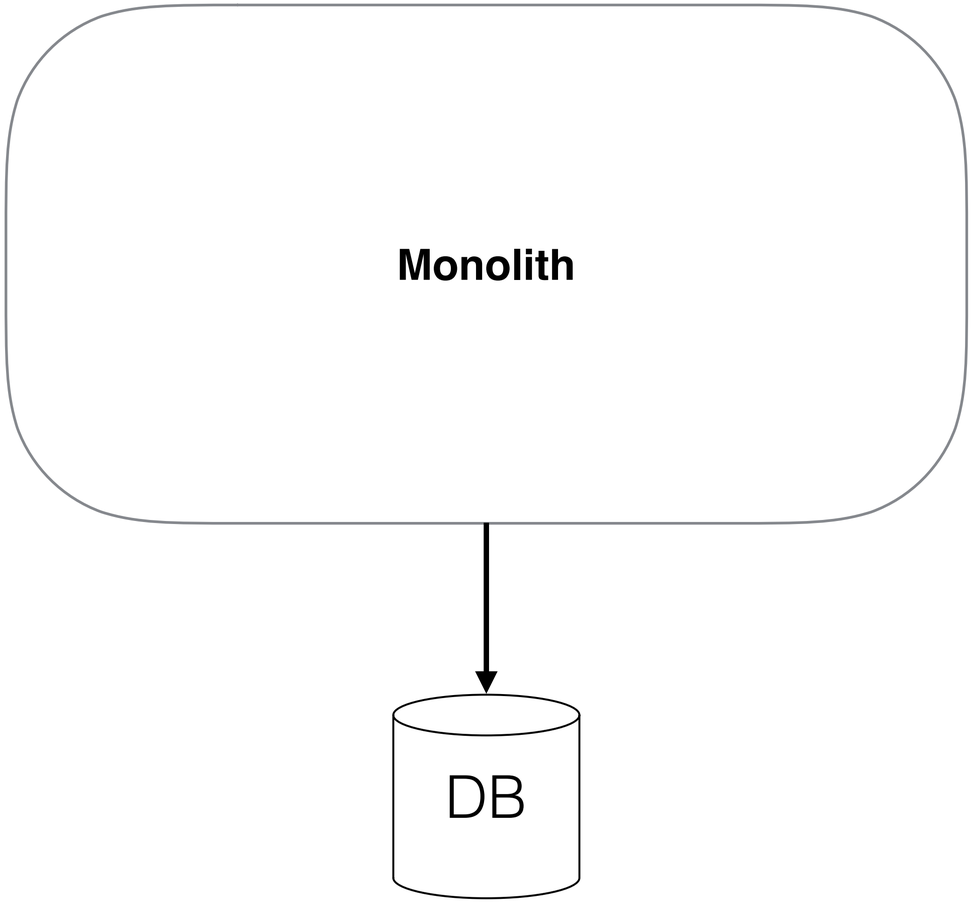
PARALLEL RUN



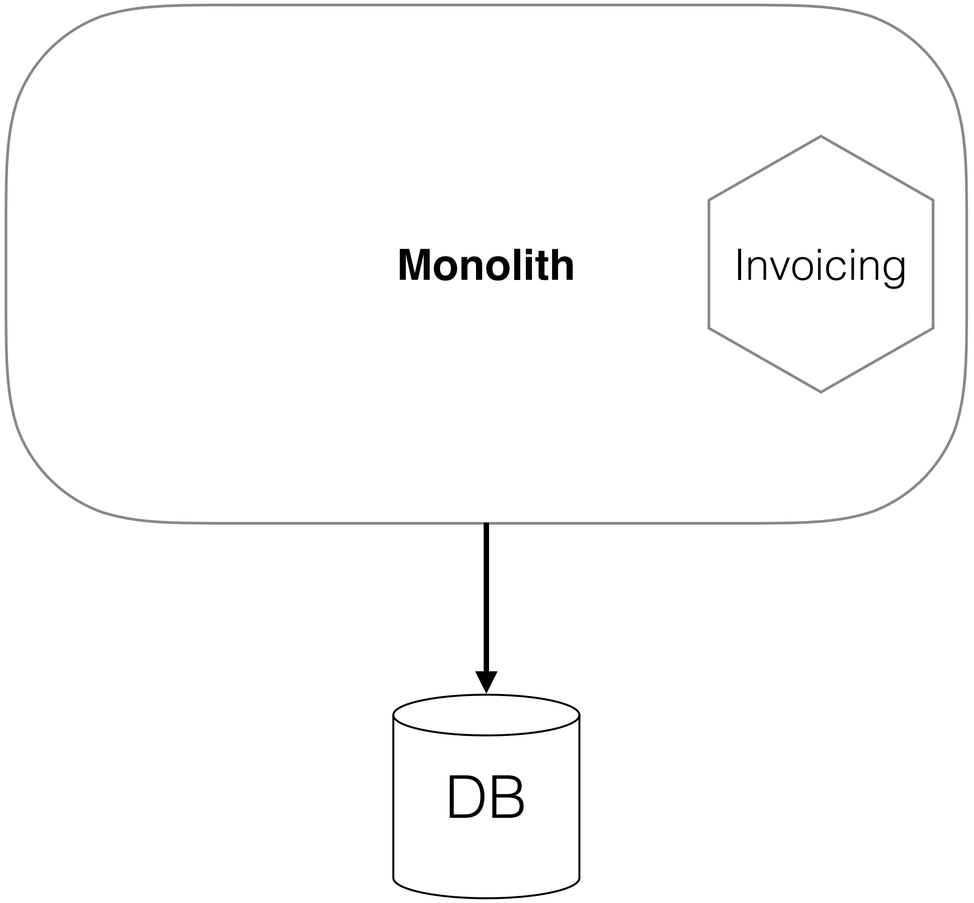
PARALLEL RUN



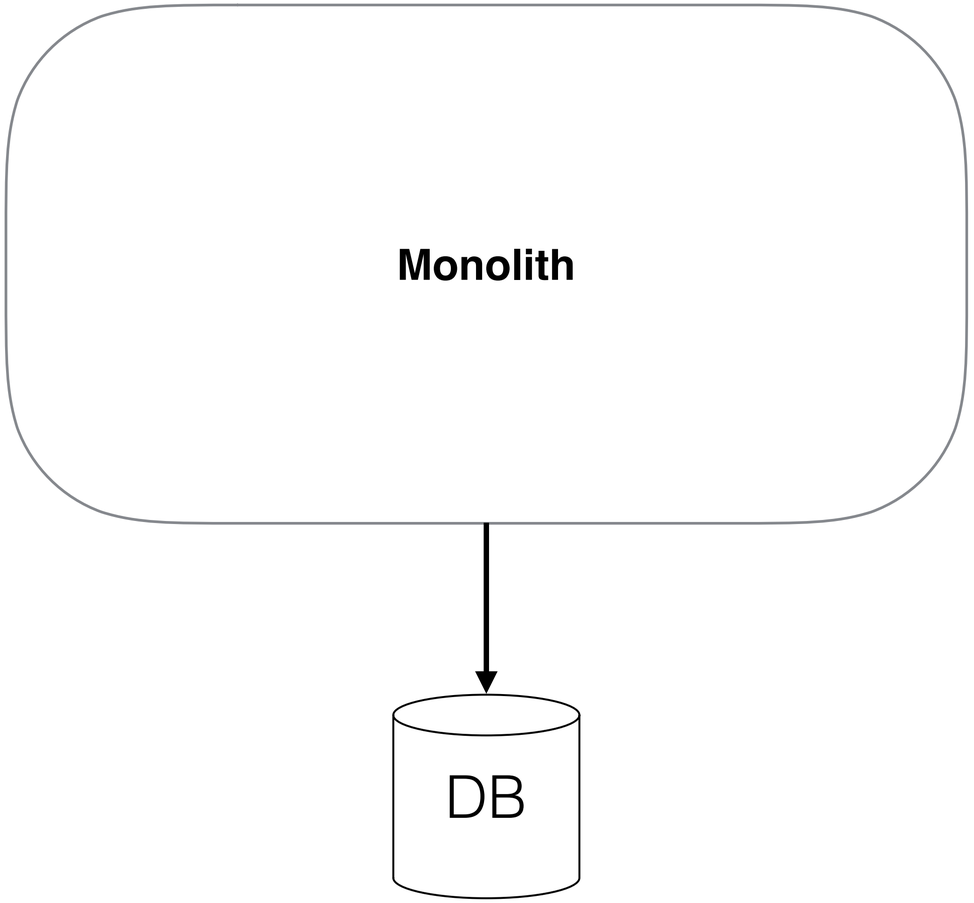
ACCESSING DATA



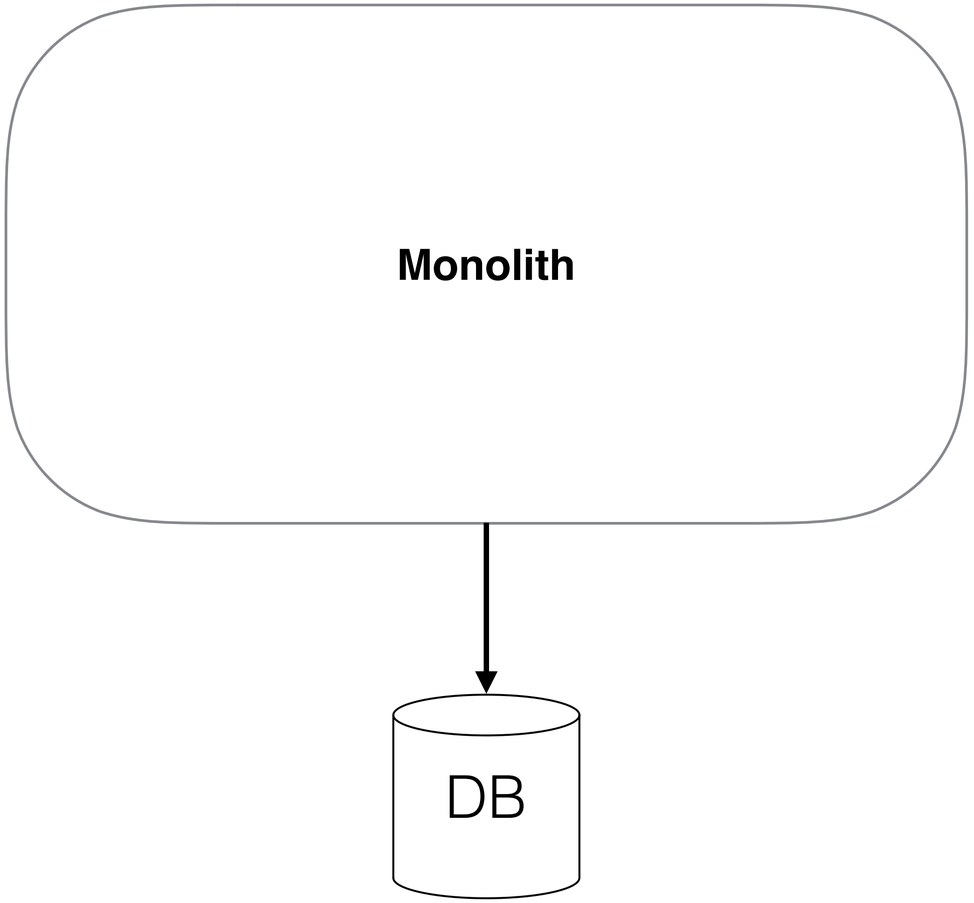
ACCESSING DATA



ACCESSING DATA

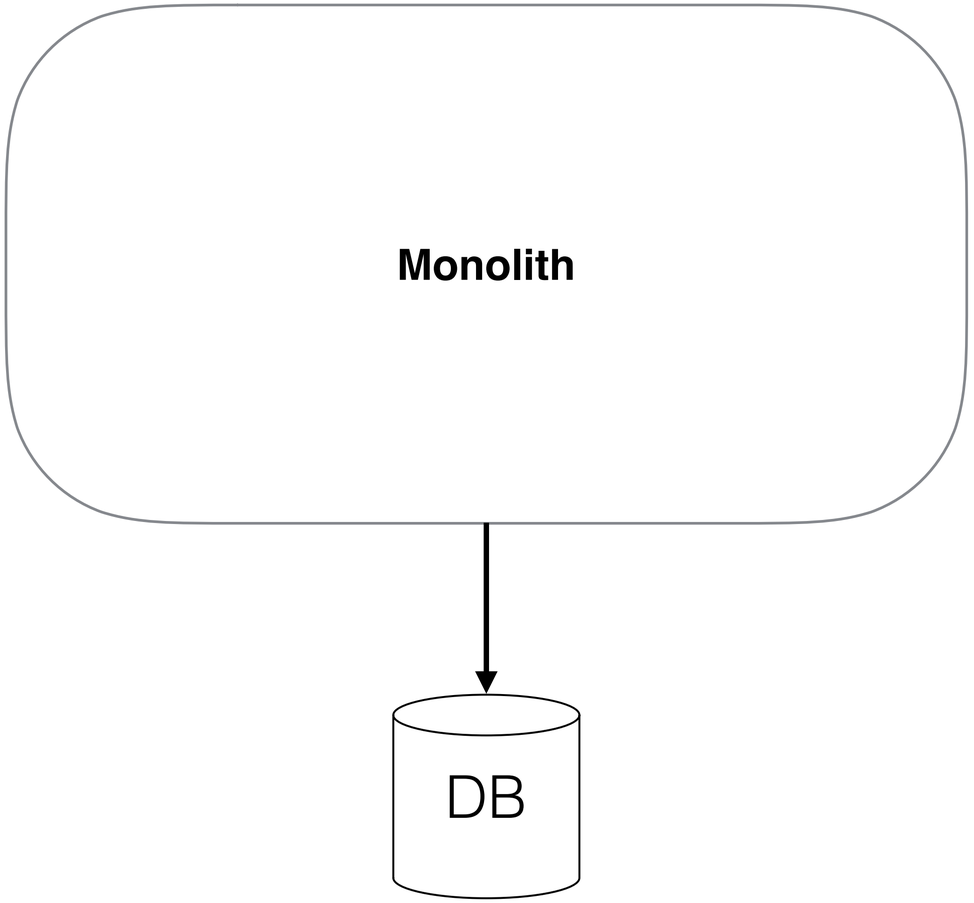


ACCESSING DATA



Our new service needs some data….

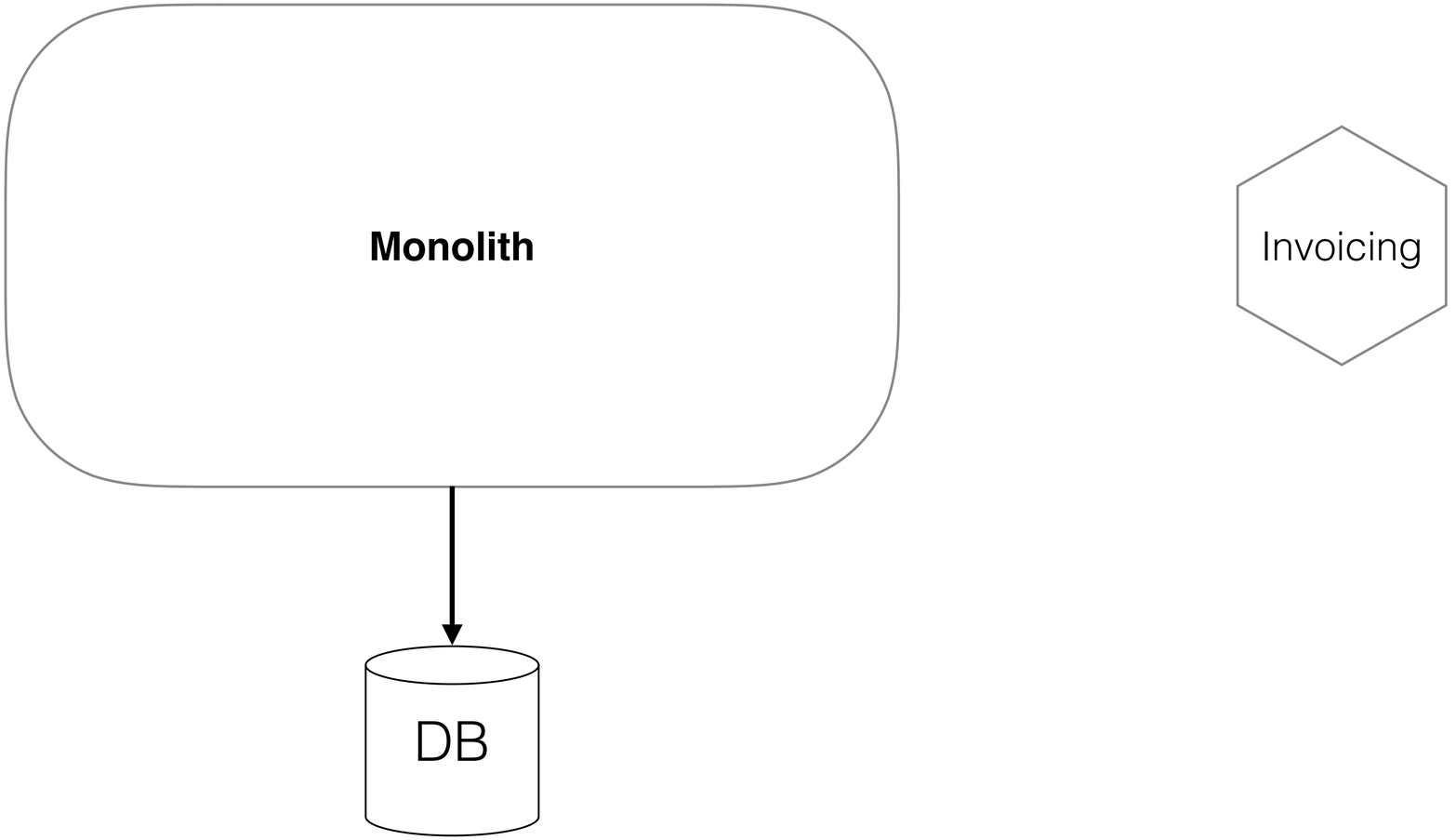
ACCESSING DATA



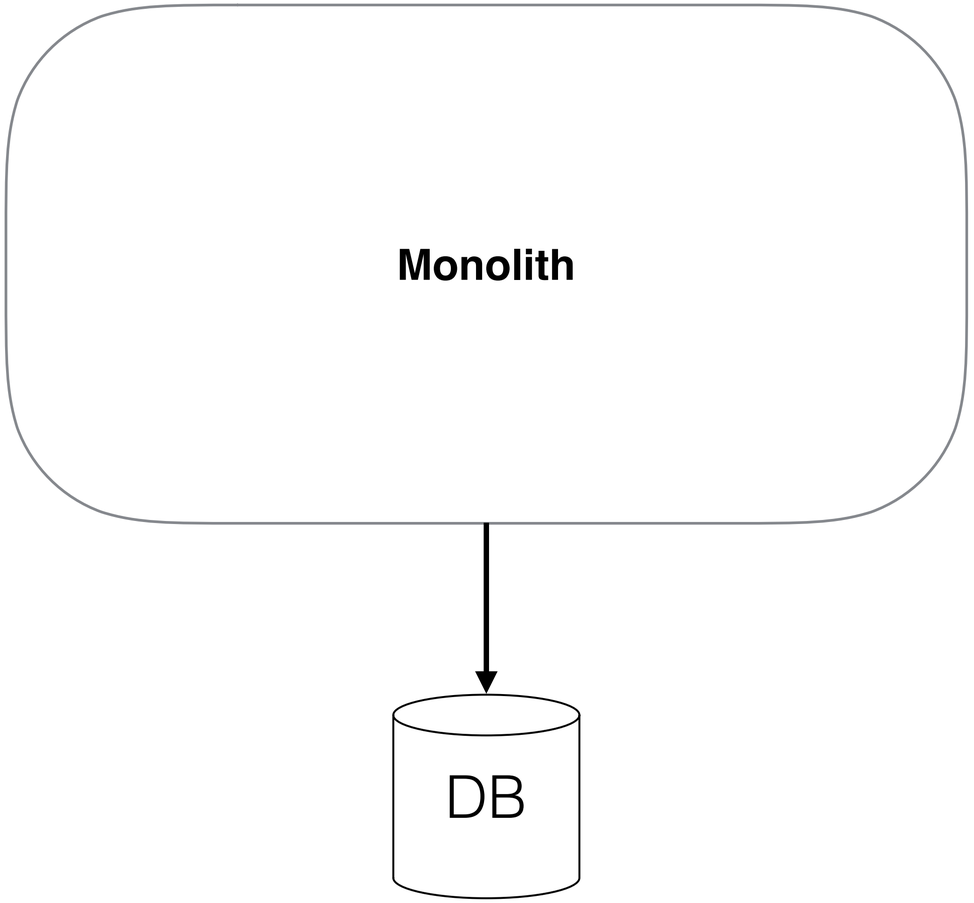
Our new service needs some data….

What are our options?

DATA REUSE?

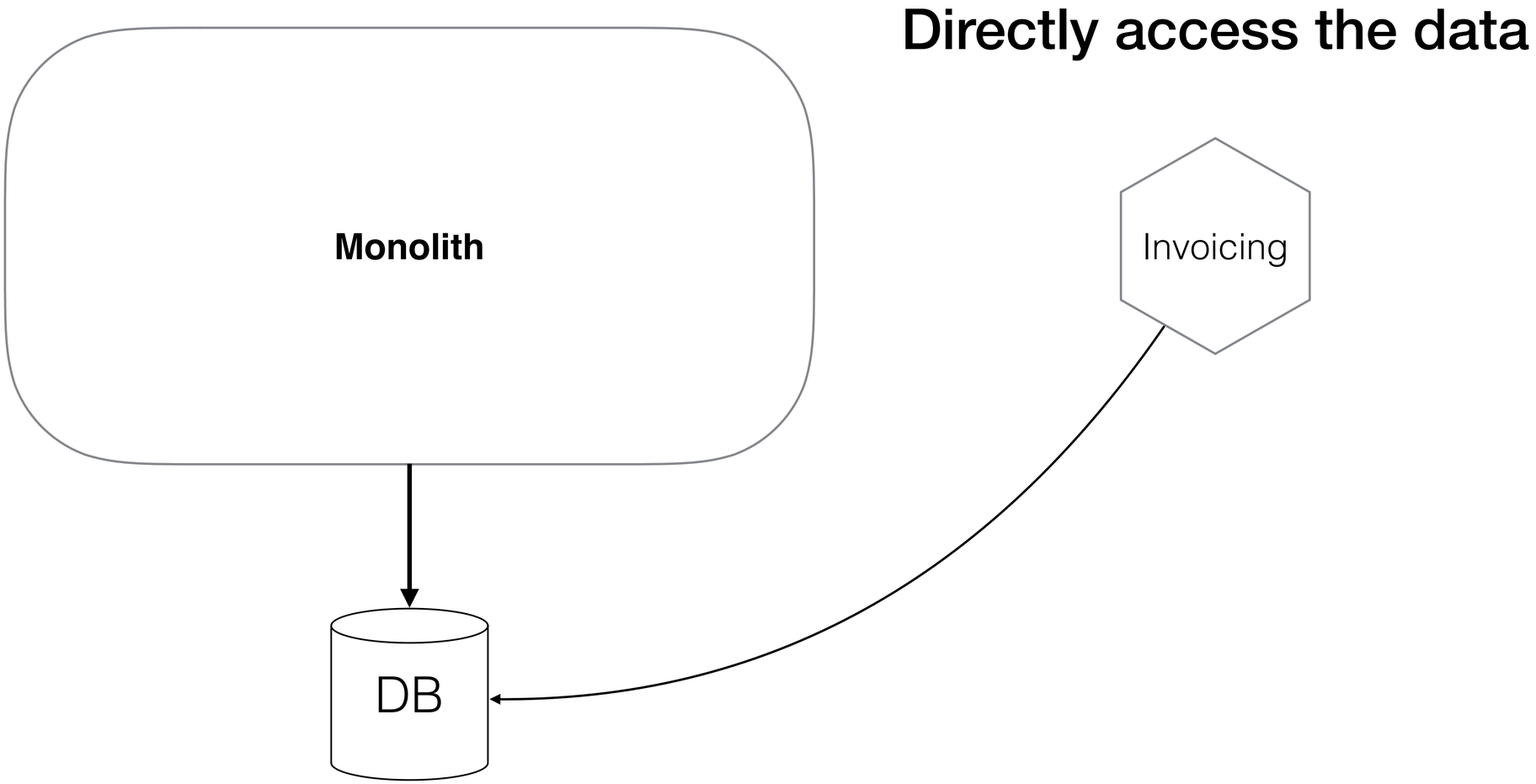


DATA REUSE?

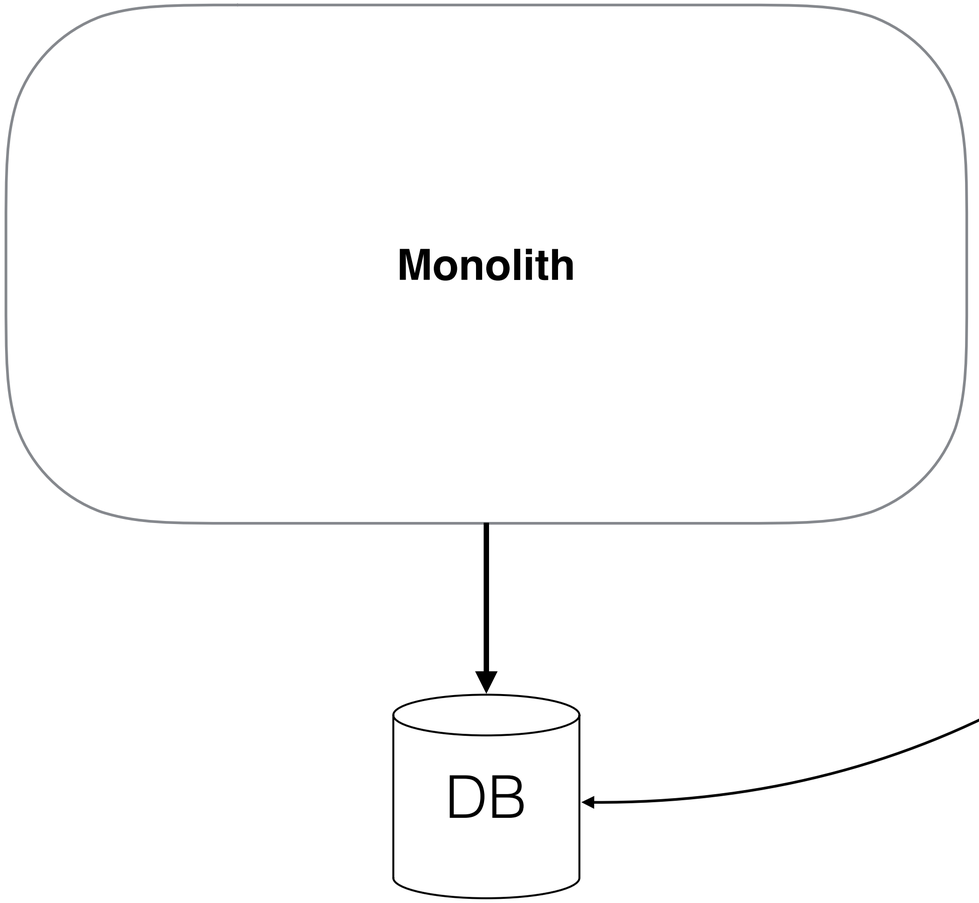


Directly access the data

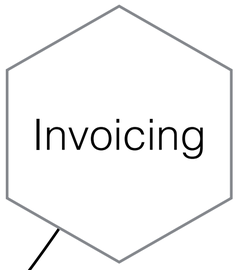
DATA REUSE?



DATA REUSE?

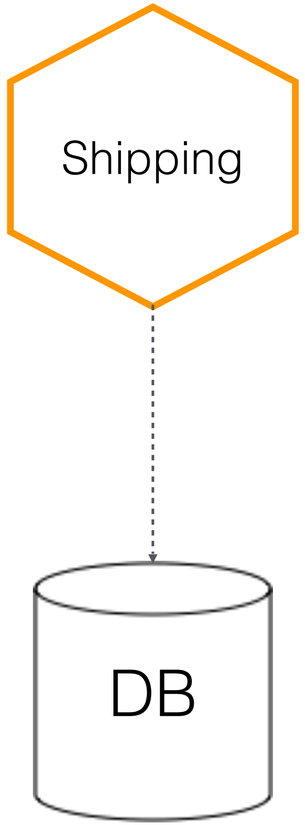


Directly access the data



But coupling!

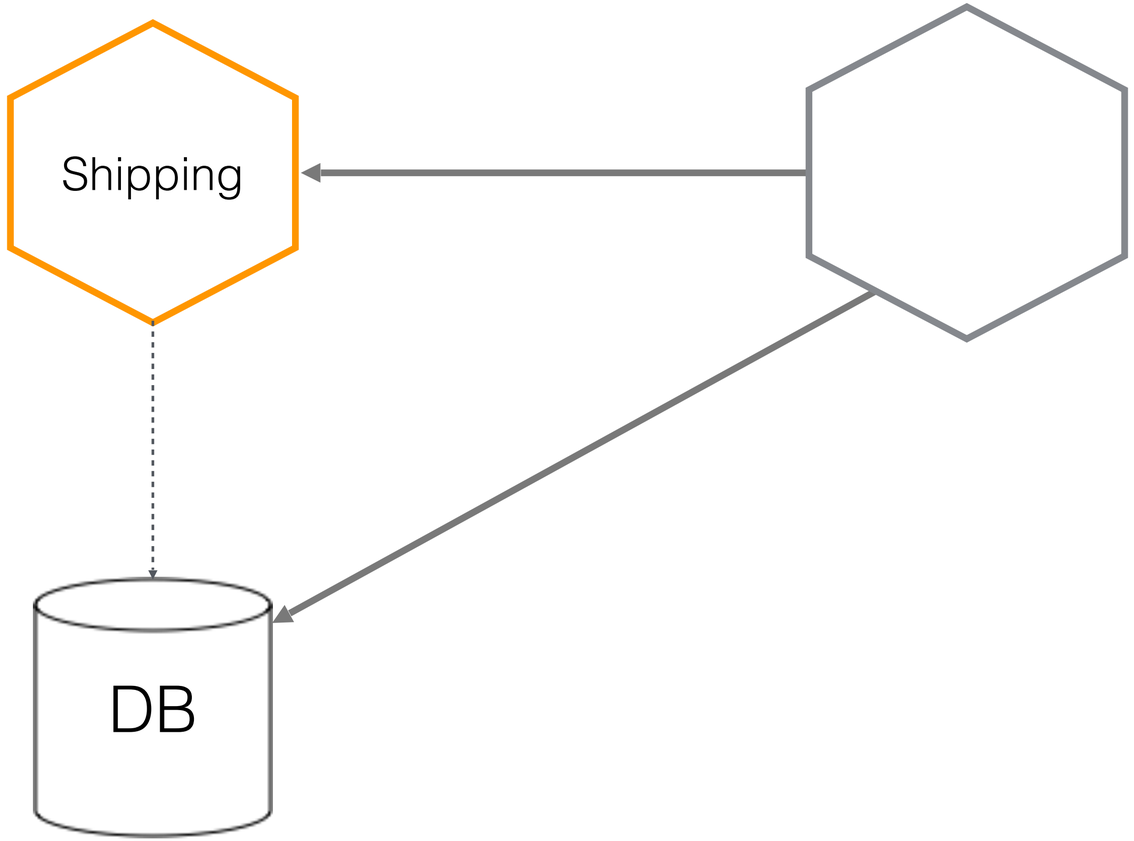
AVOID SHARING DATABASES!



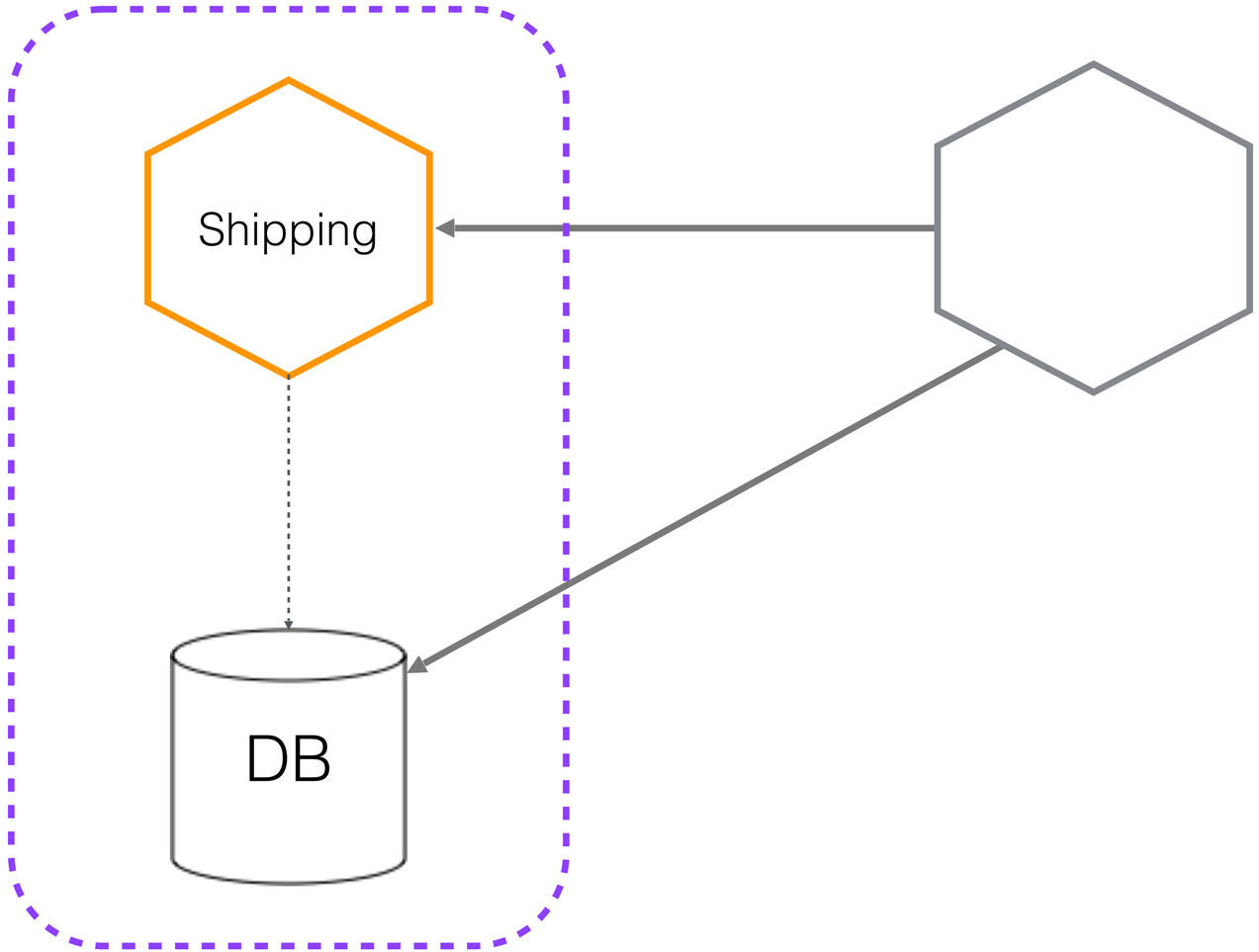
AVOID SHARING DATABASES!

AVOID SHARING DATABASES!

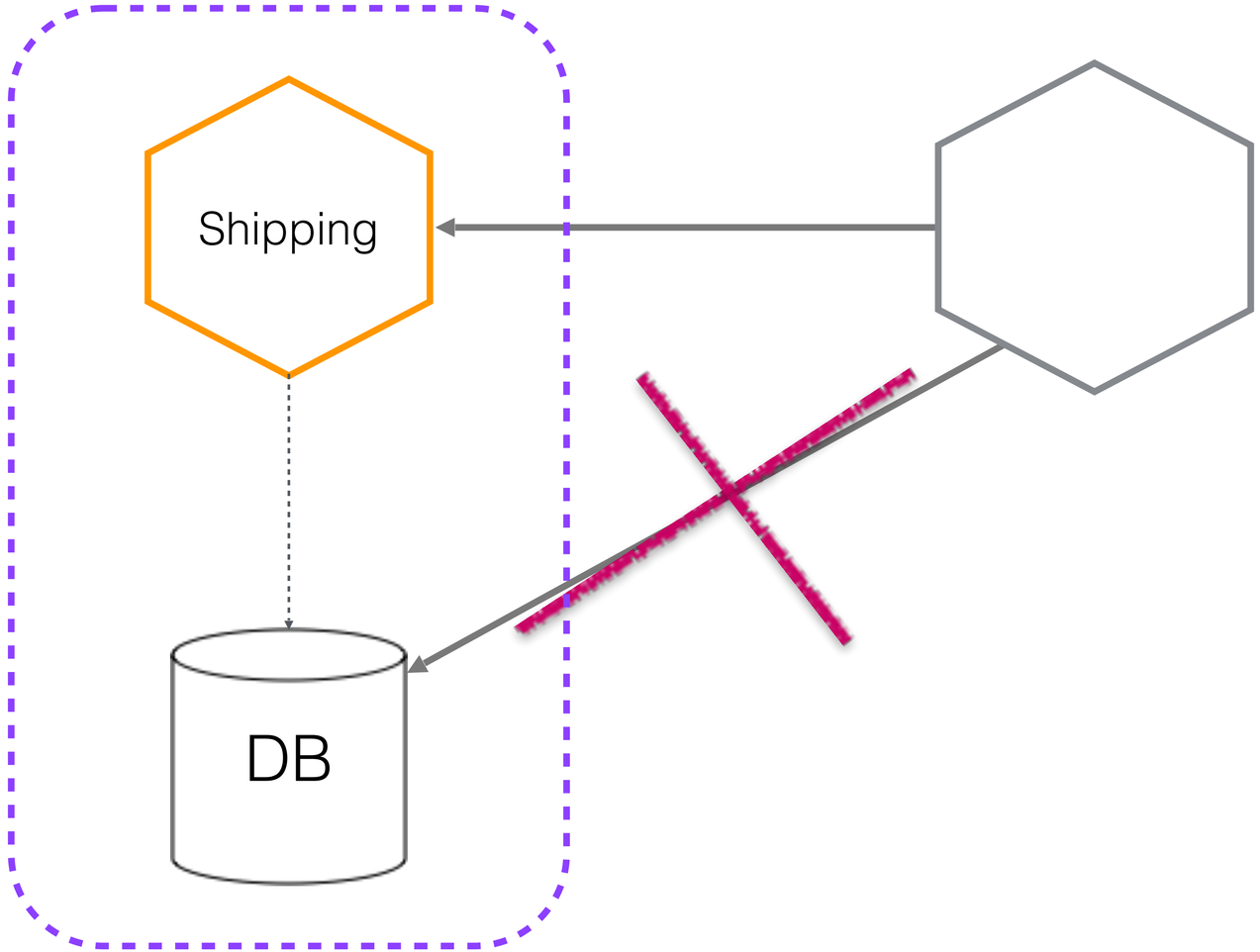
AVOID SHARING DATABASES!



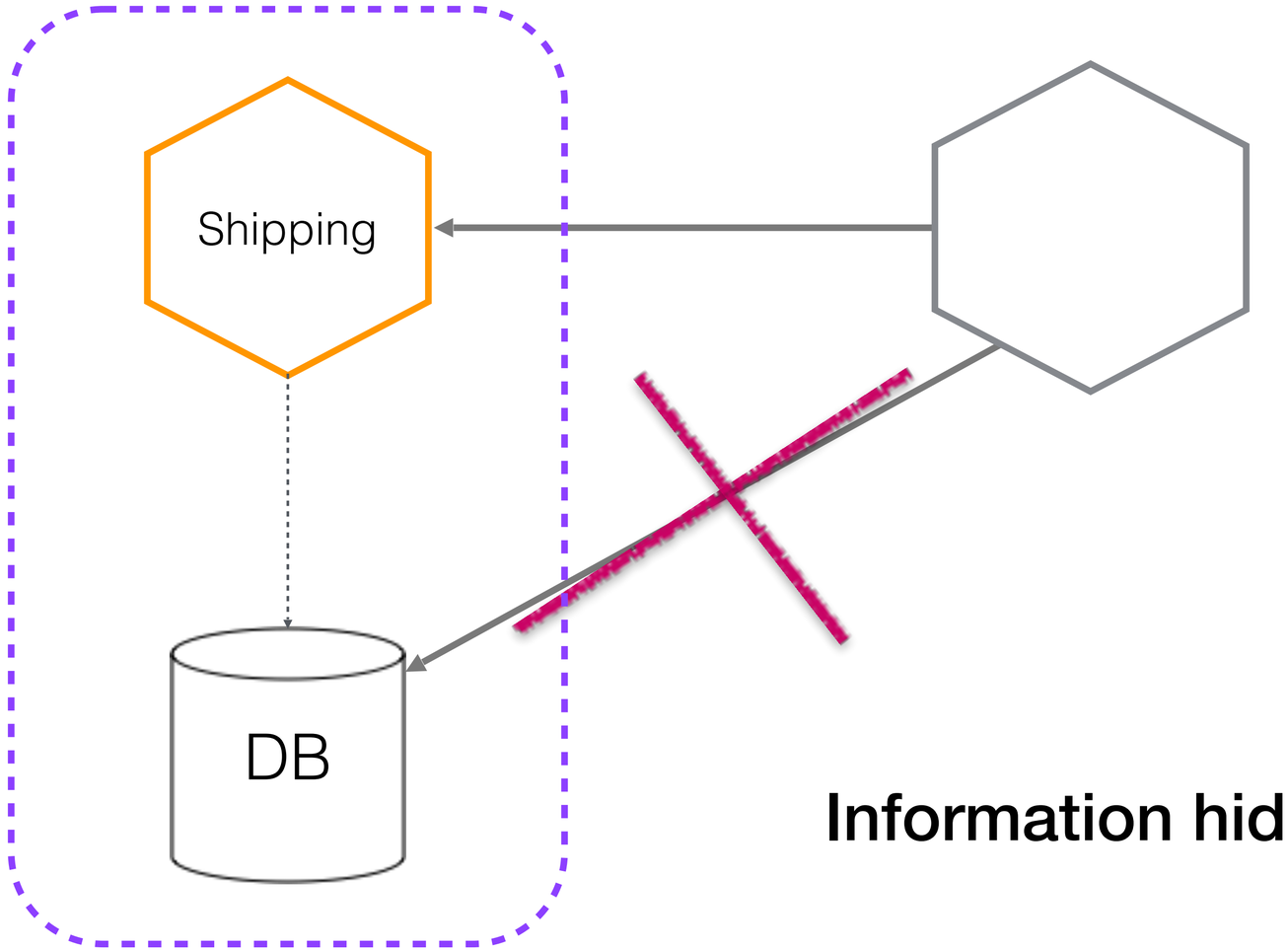
AVOID SHARING DATABASES!



AVOID SHARING DATABASES!

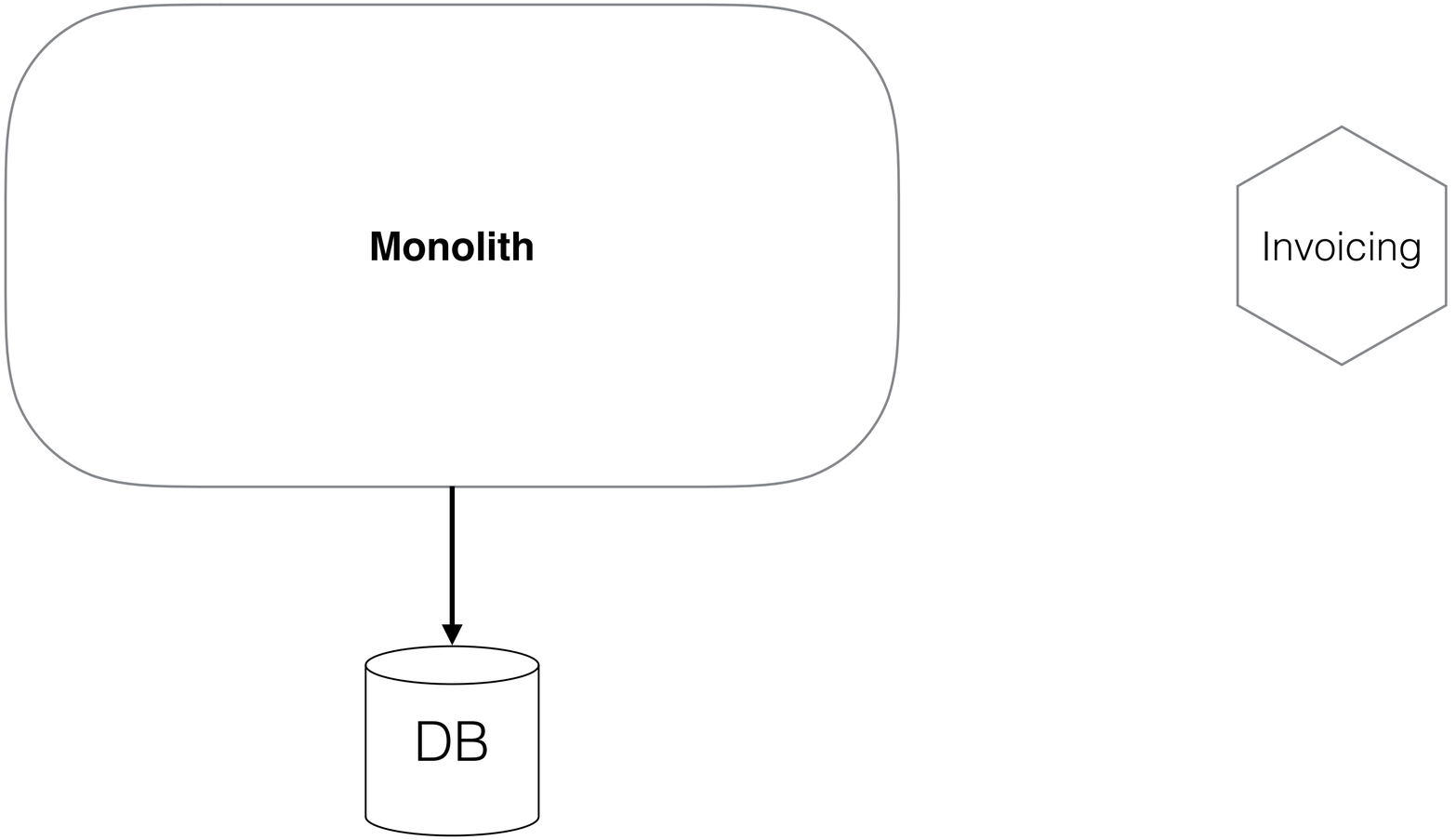


AVOID SHARING DATABASES!

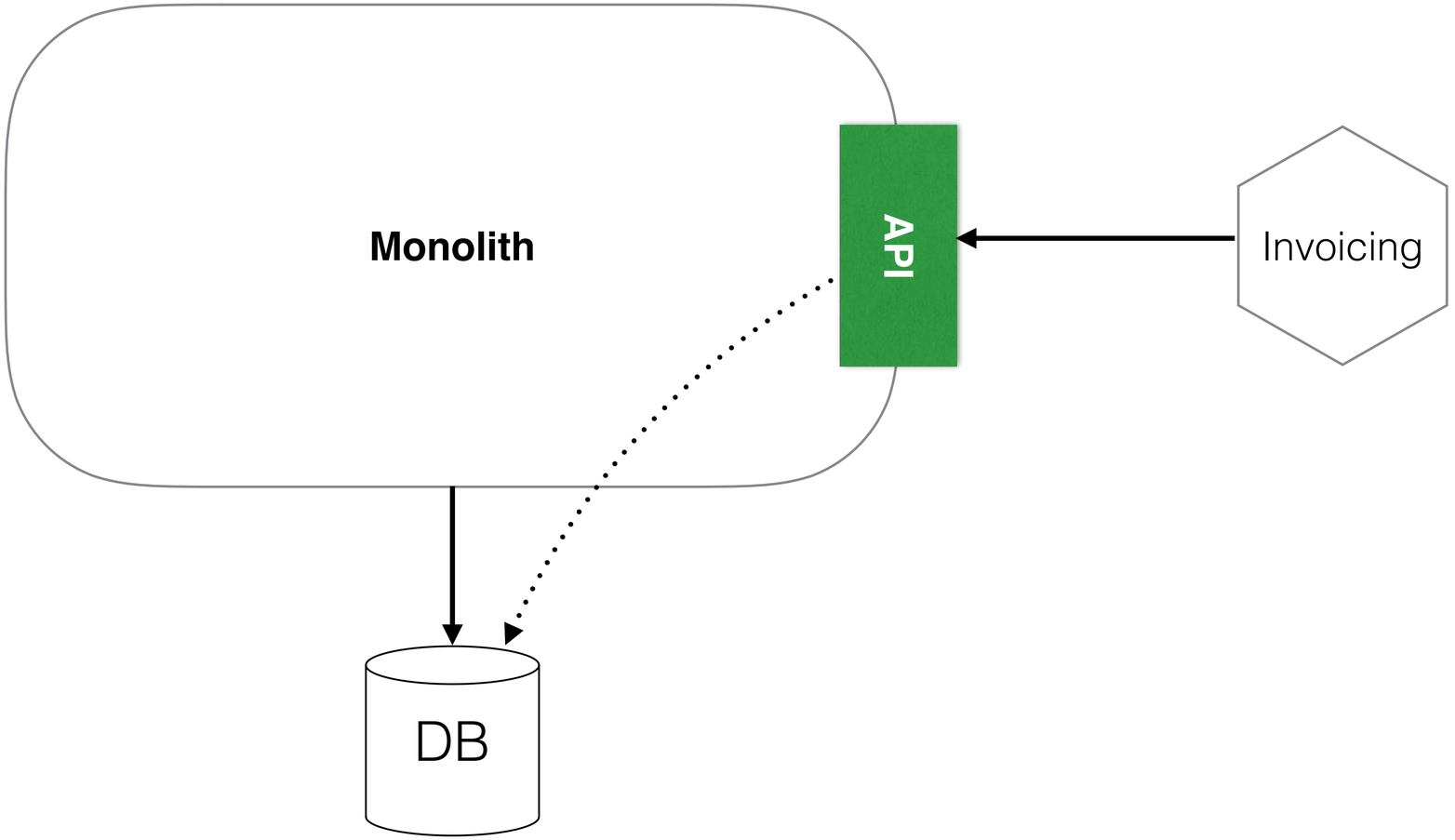


Information hiding!

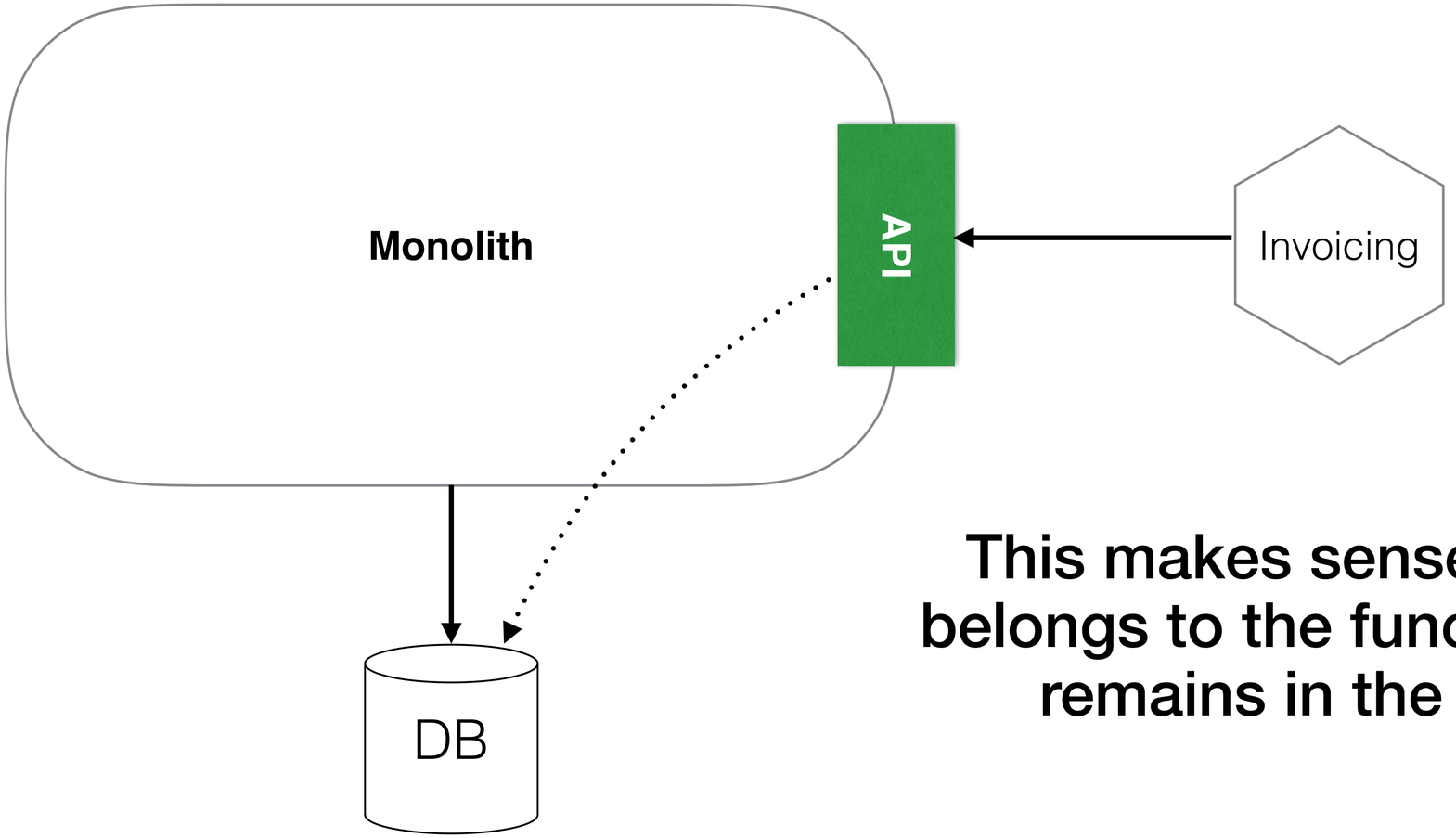
EXPOSE DATA IN THE MONOLITH



EXPOSE DATA IN THE MONOLITH

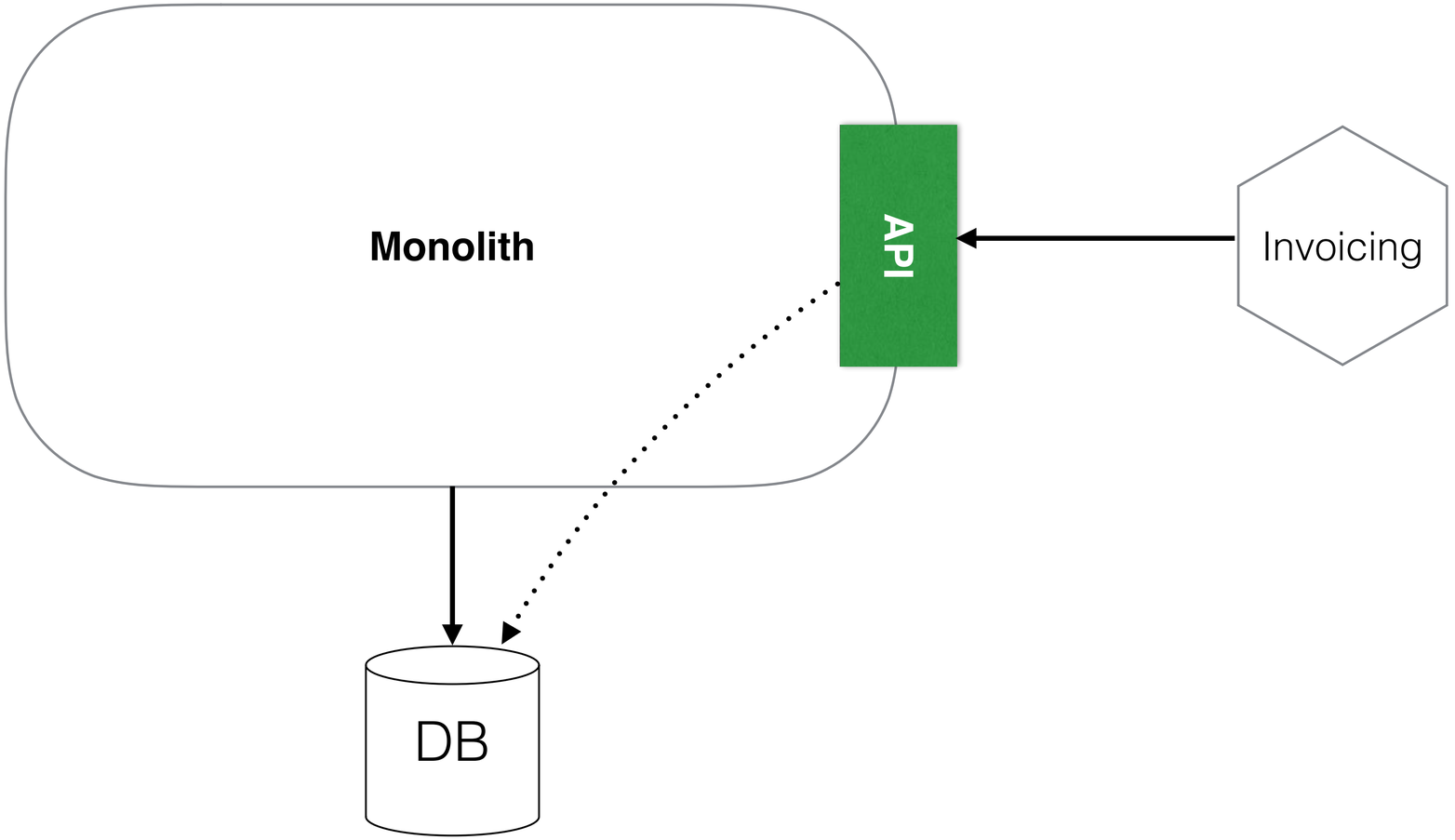


EXPOSE DATA IN THE MONOLITH

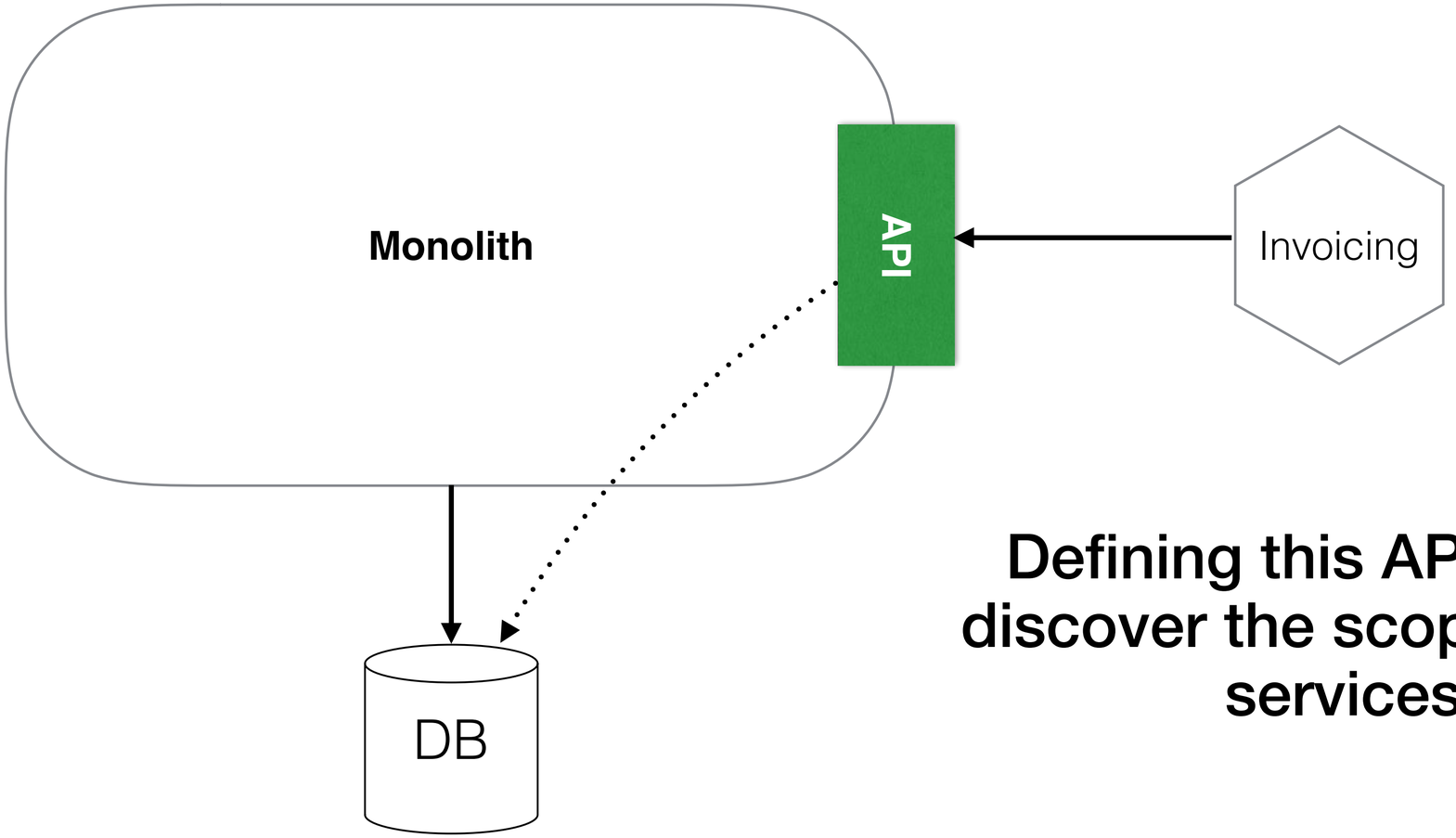


This makes sense if the data belongs to the functionality that remains in the monolith

EXPOSE DATA IN THE MONOLITH (CONT)

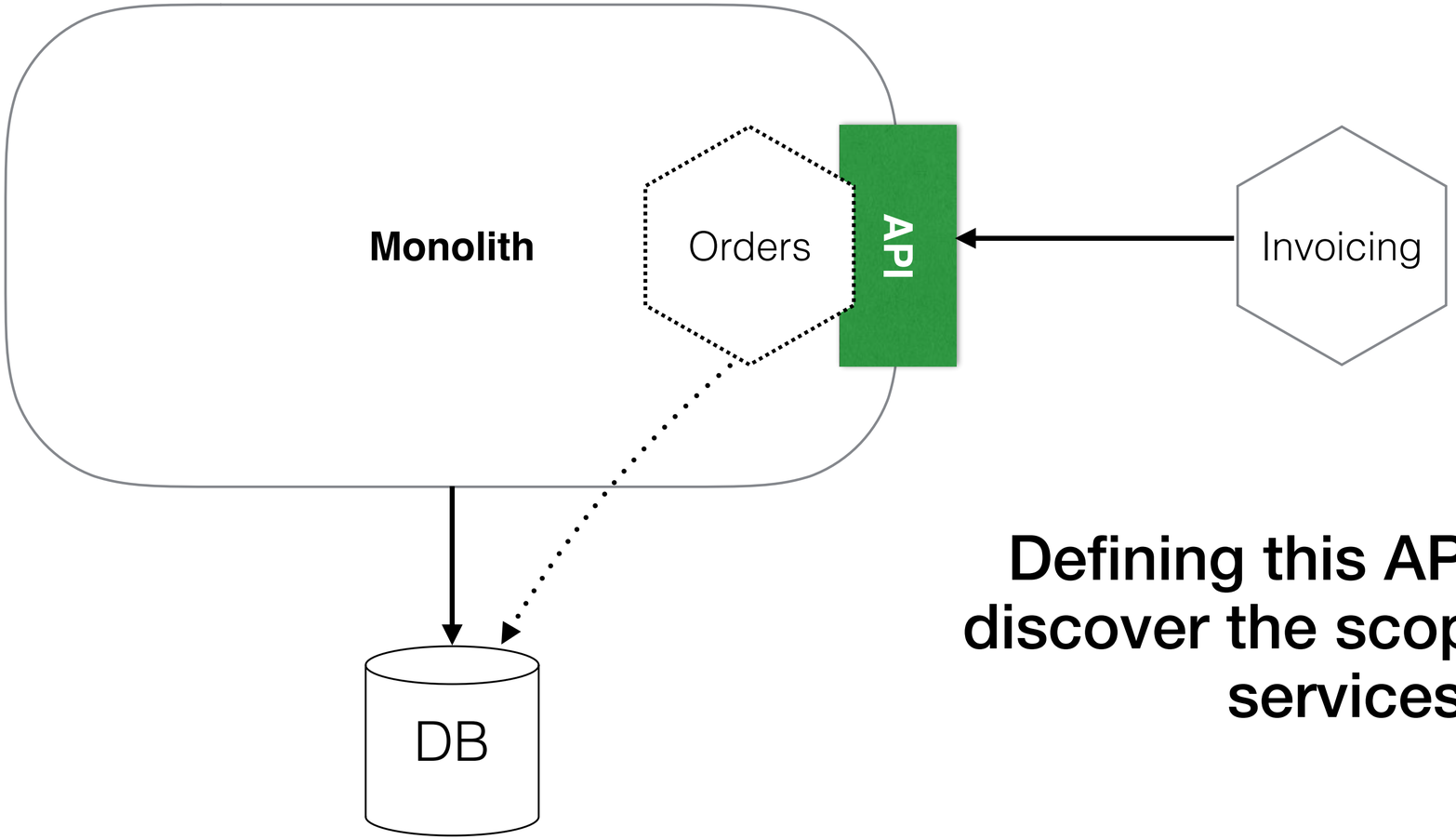


EXPOSE DATA IN THE MONOLITH (CONT)



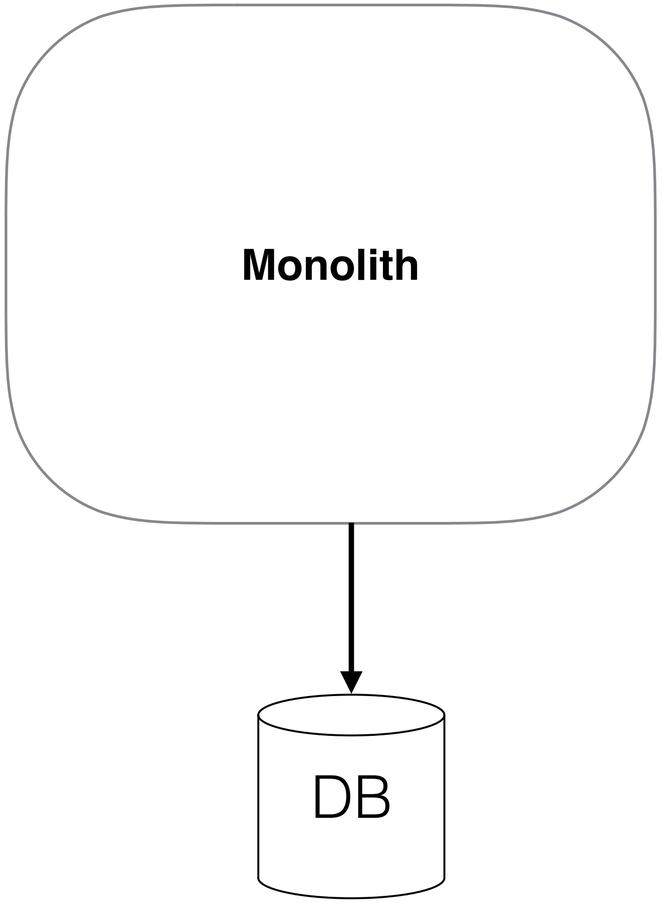
Defining this API can help discover the scope of further services…

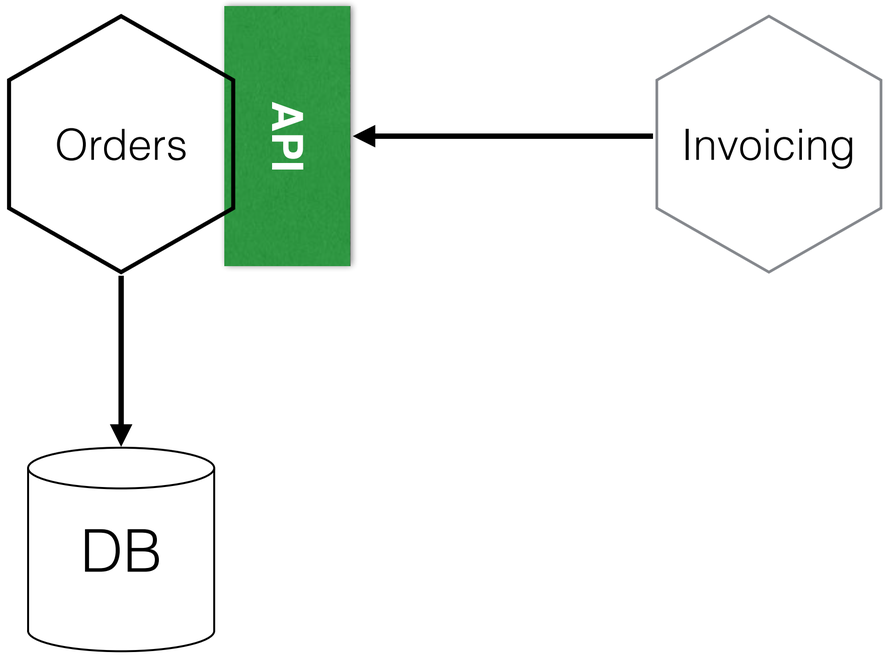
EXPOSE DATA IN THE MONOLITH (CONT)



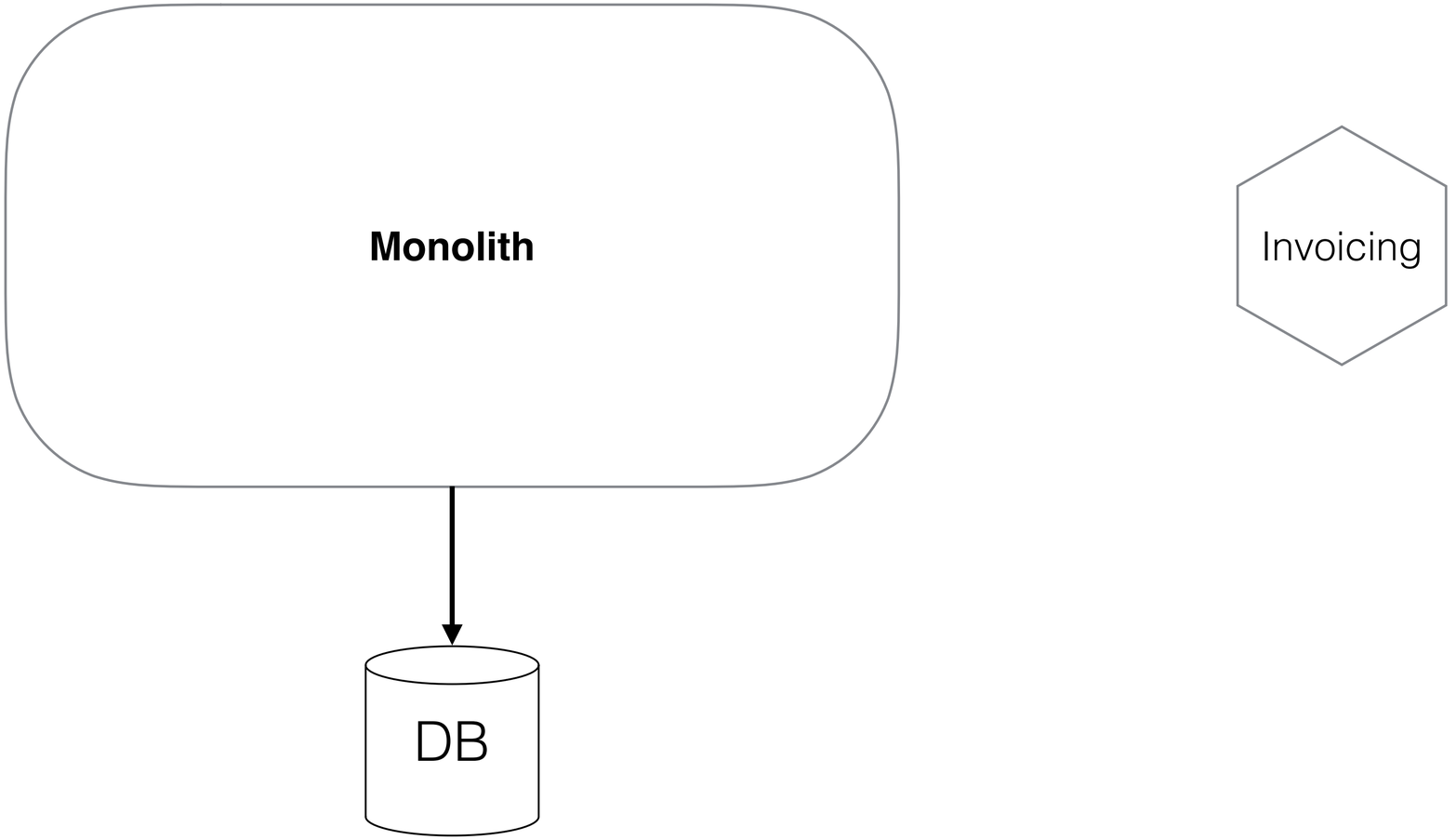
Defining this API can help discover the scope of further services…

EXPOSE DATA IN THE MONOLITH (CONT)

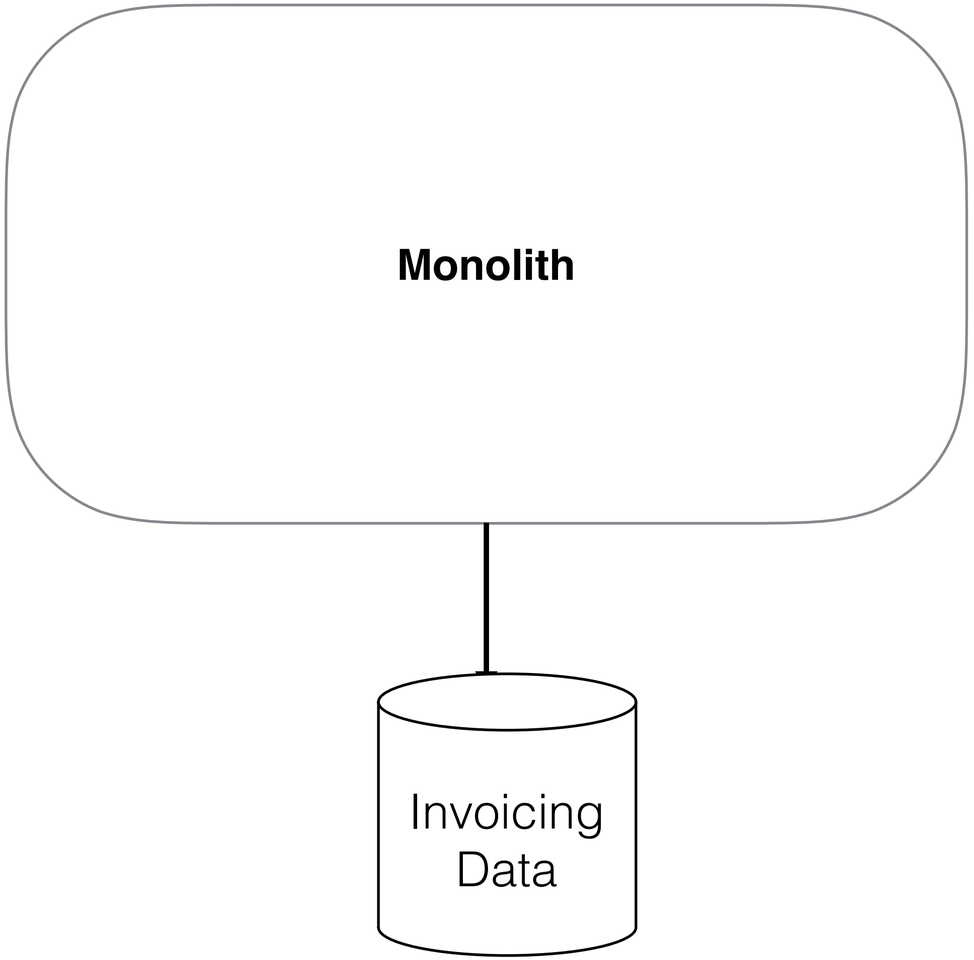




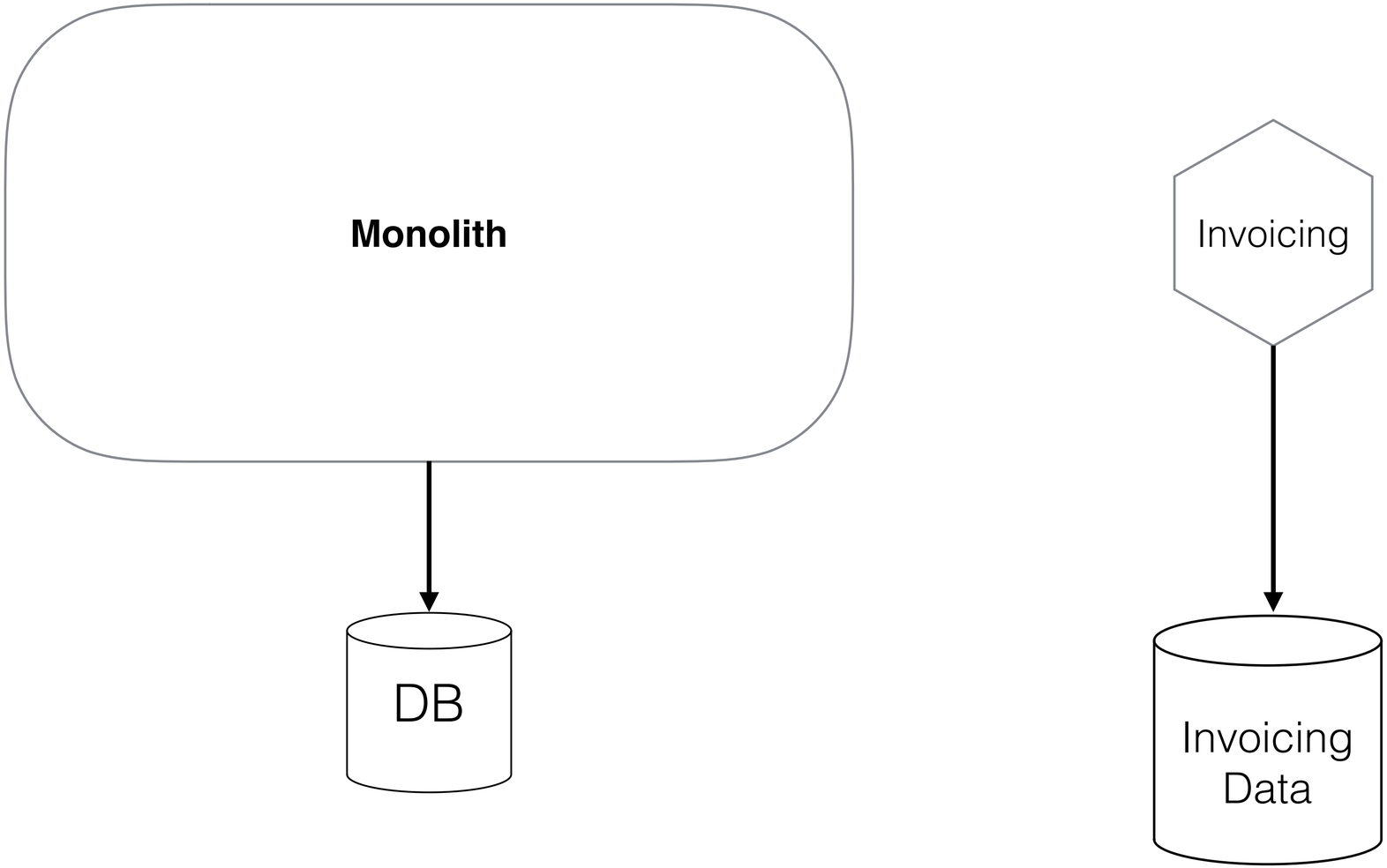
MOVE DATA TO THE NEW SERVICE



MOVE DATA TO THE NEW SERVICE

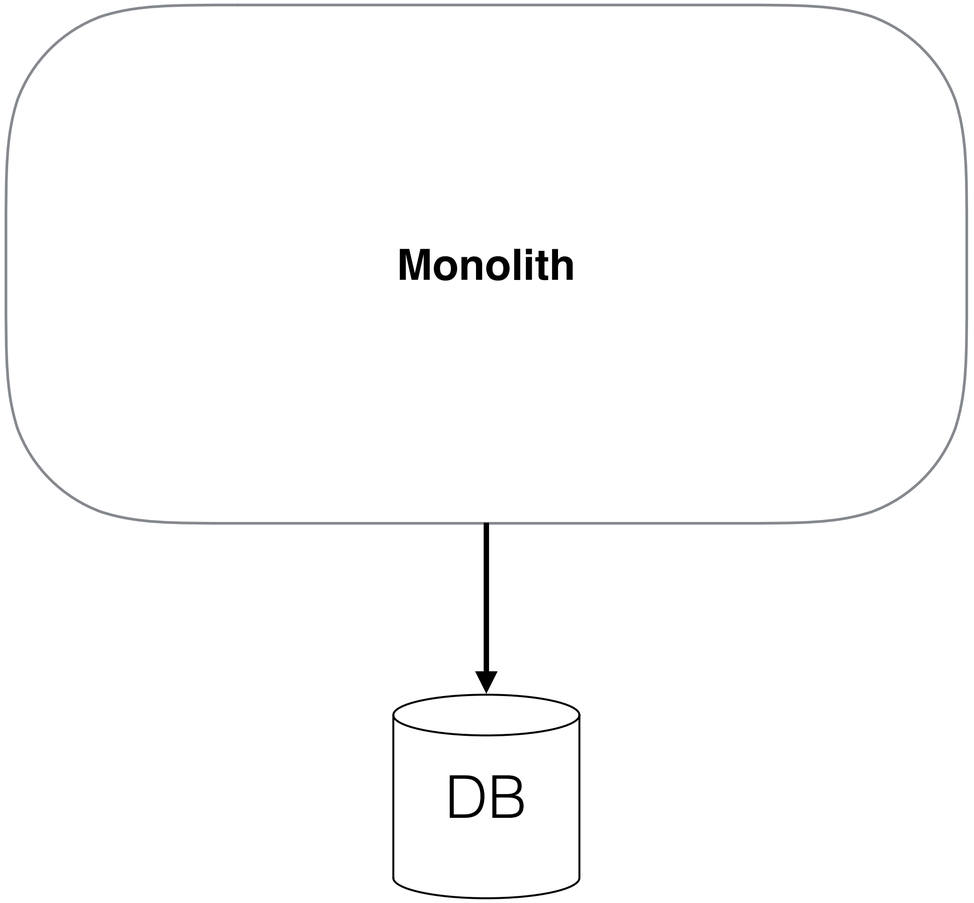


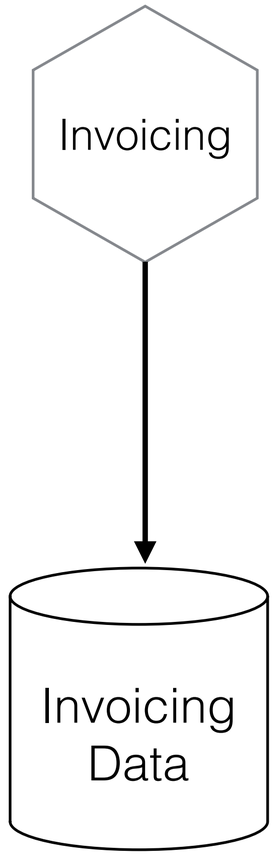
MOVE DATA TO THE NEW SERVICE



MOVE DATA TO THE NEW SERVICE

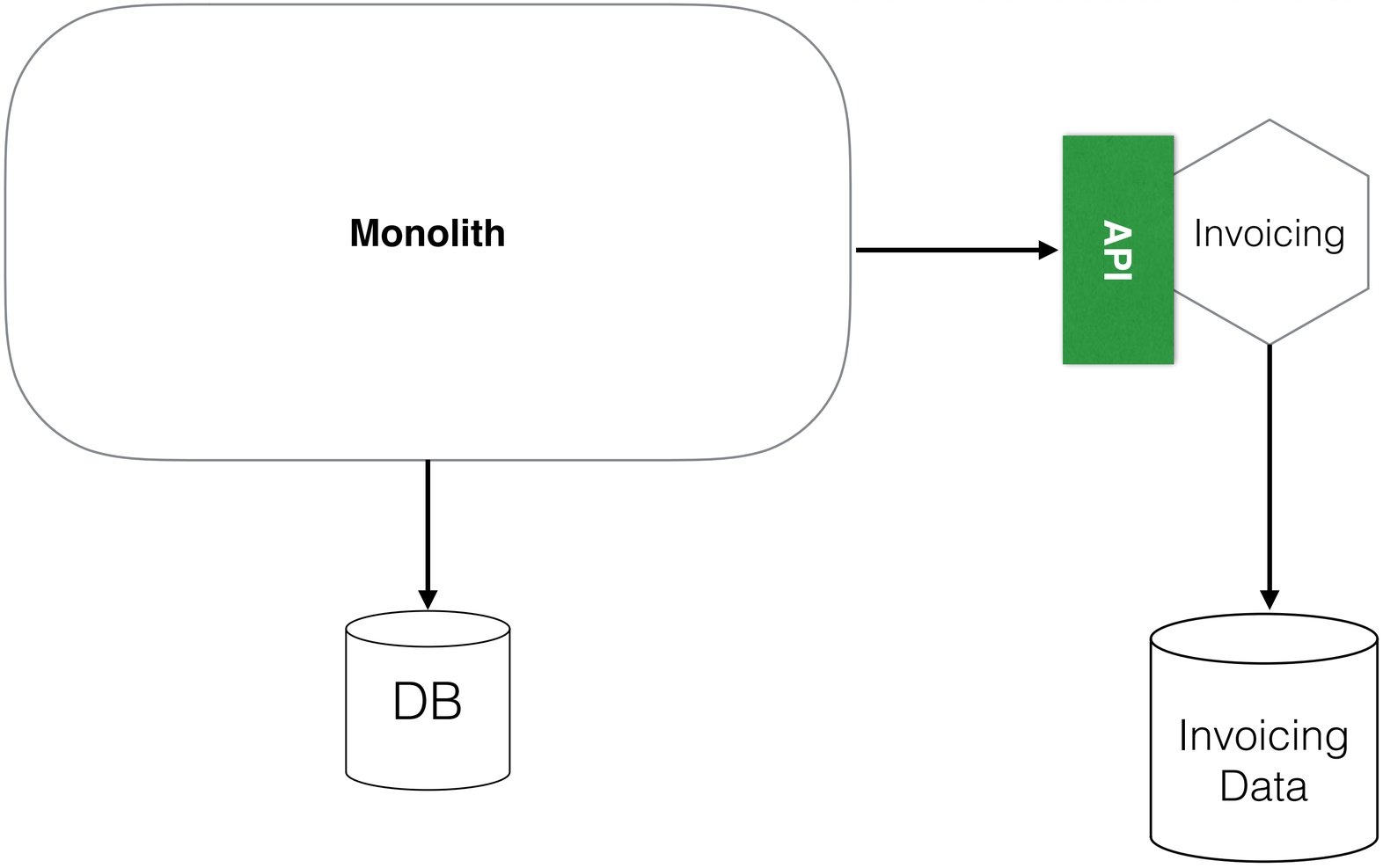
Monolith needs to be changed to source data from the new service



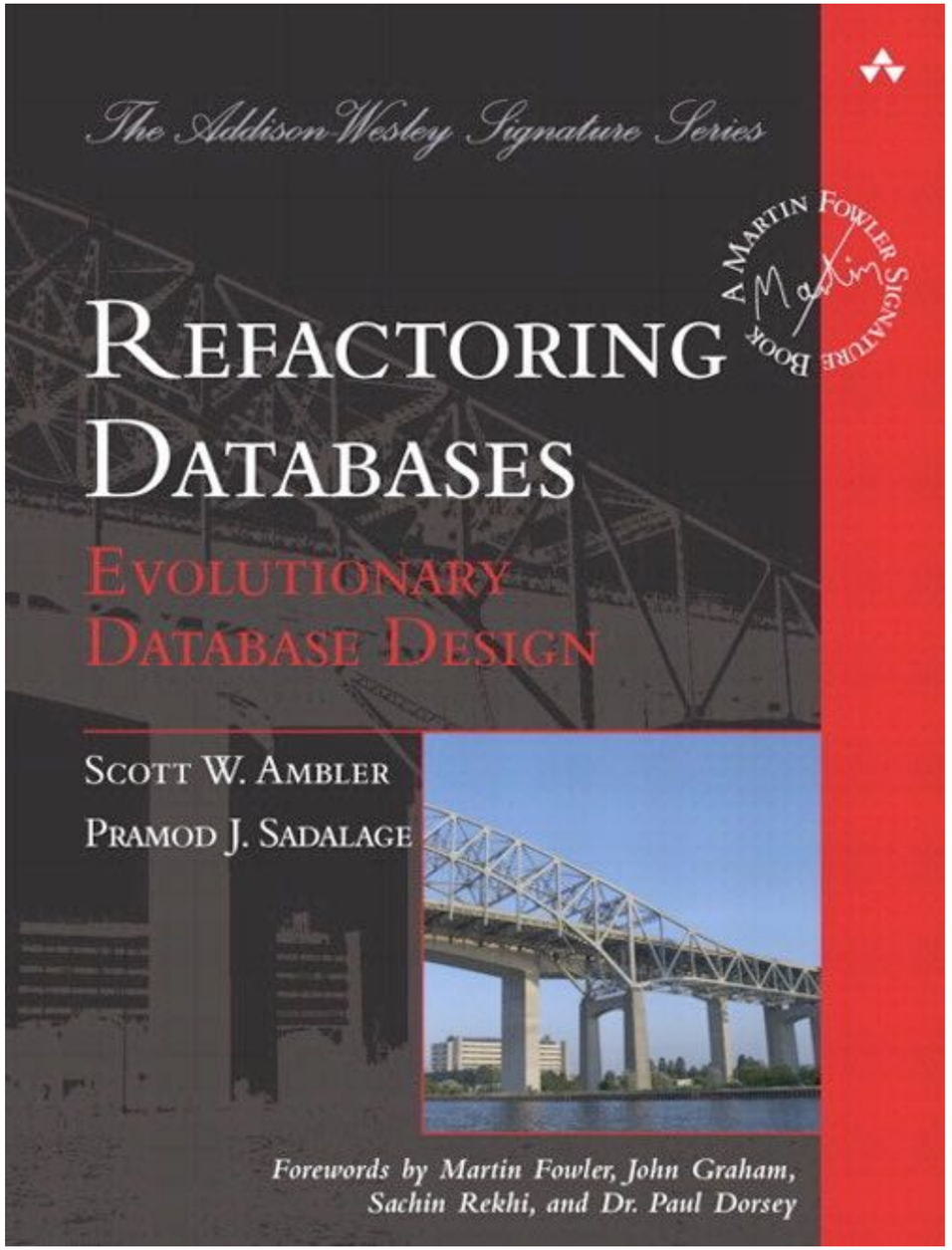


MOVE DATA TO THE NEW SERVICE

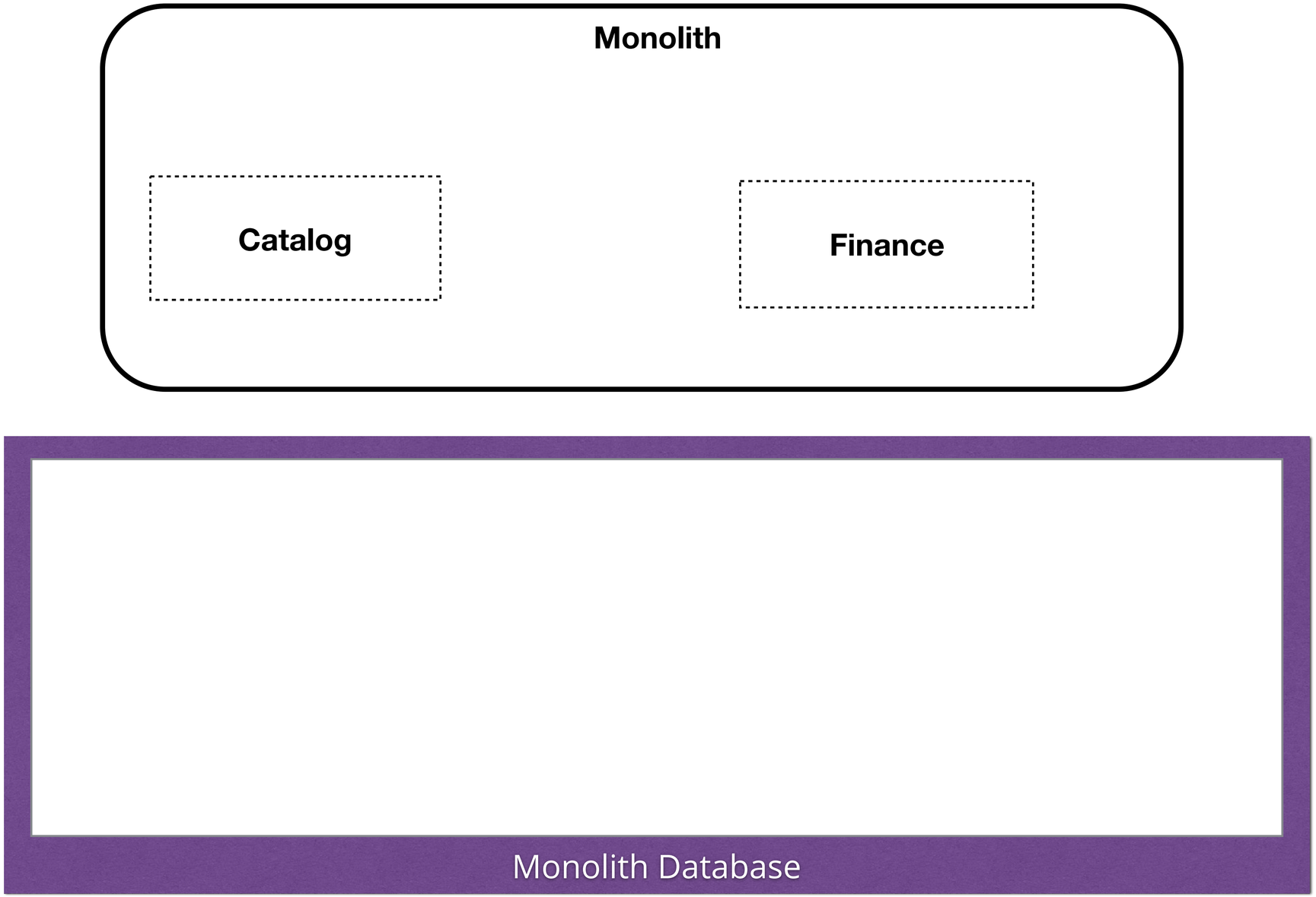
Monolith needs to be changed to source data from the new service



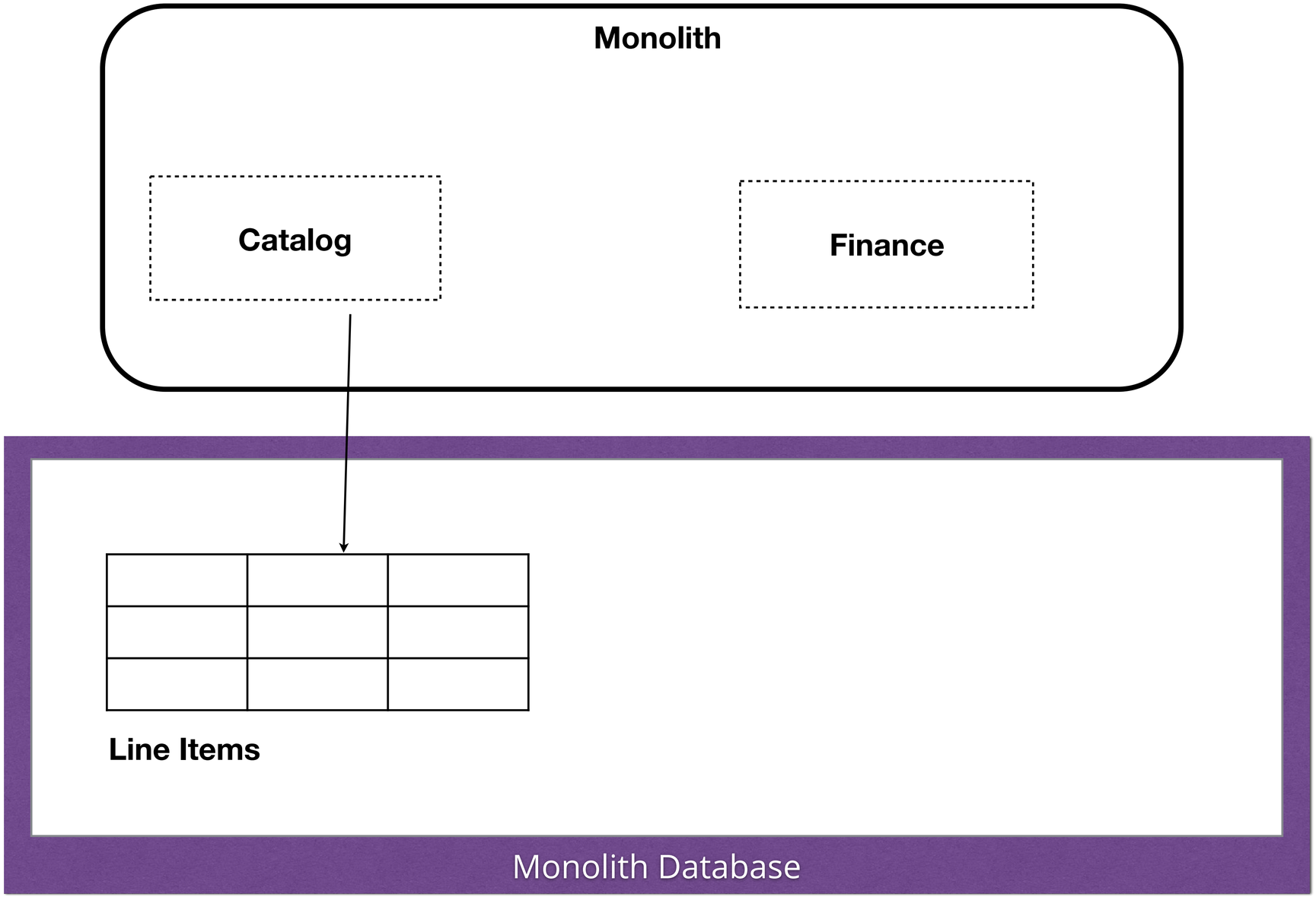
DB Refactoring Patterns


The Addison Wesley Signature Series
A Martin Fowler Signature Book
Refactoring Databases
Evolutionary Database Design
Scott W. Ambler
Pramod J. Sadalage
Forewords by Martin Fowler, John Graham, Sachin Rekhi, and Dr. Paul Dorsey

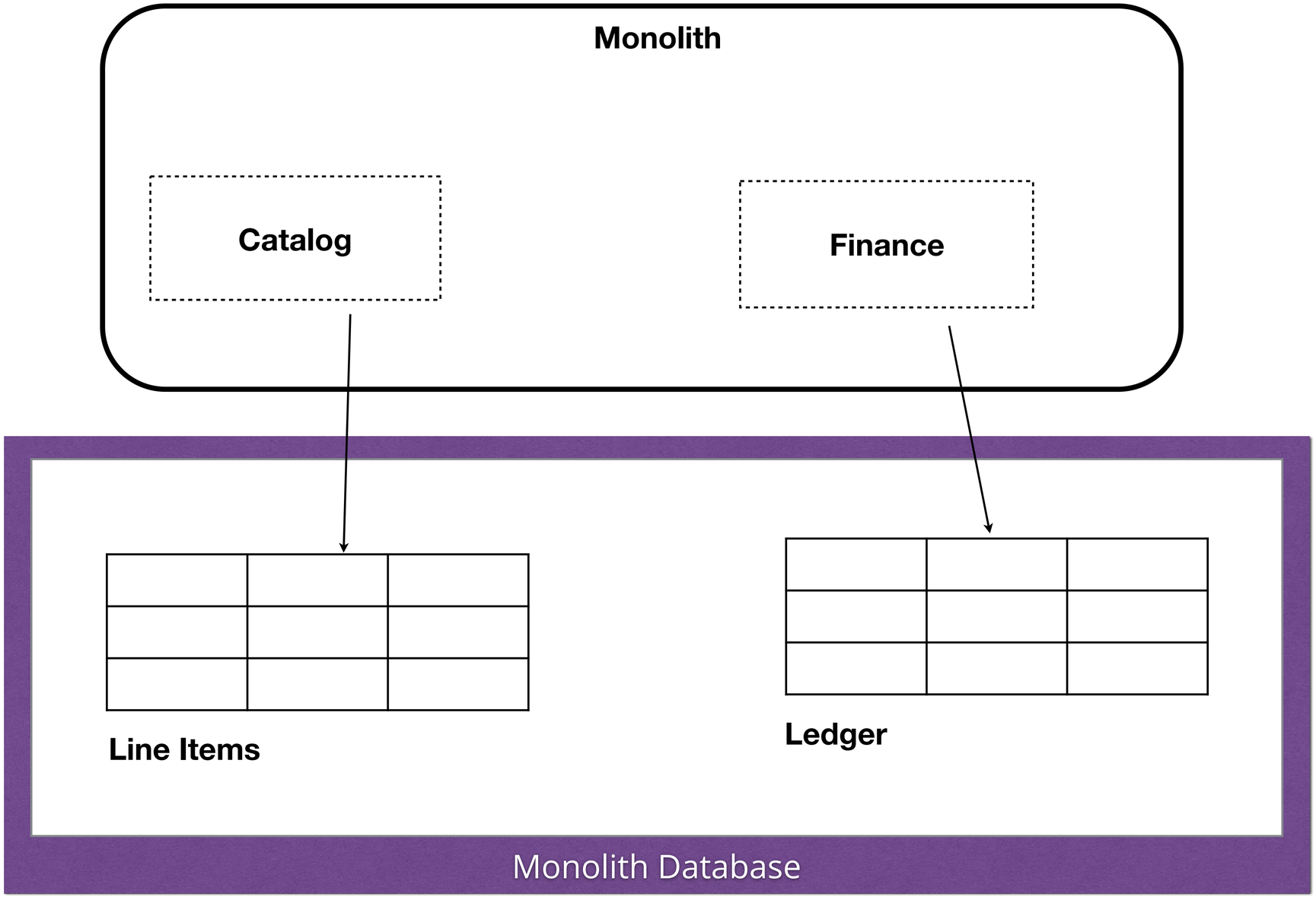
JOINS ACROSS TABLES



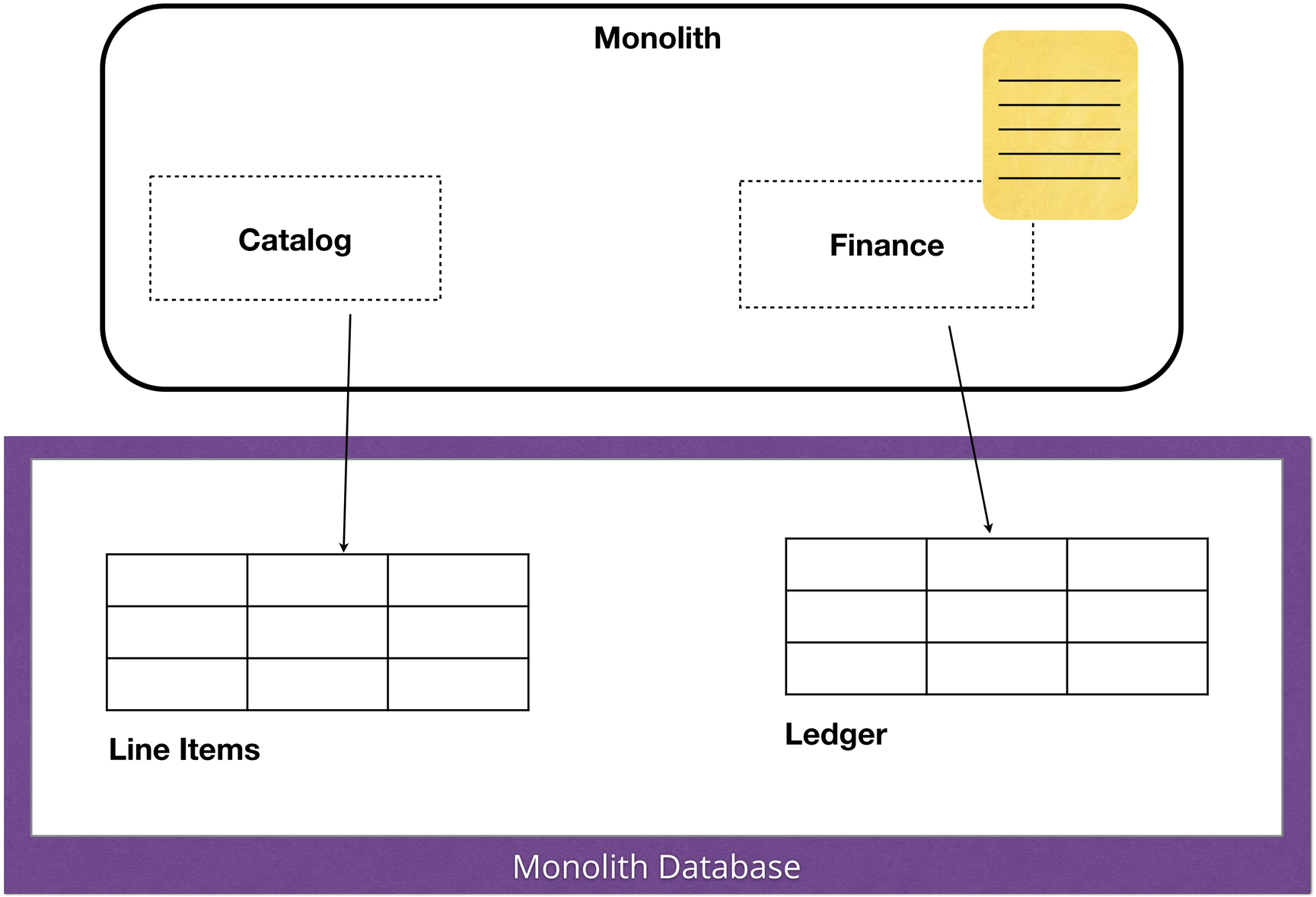
JOINS ACROSS TABLES



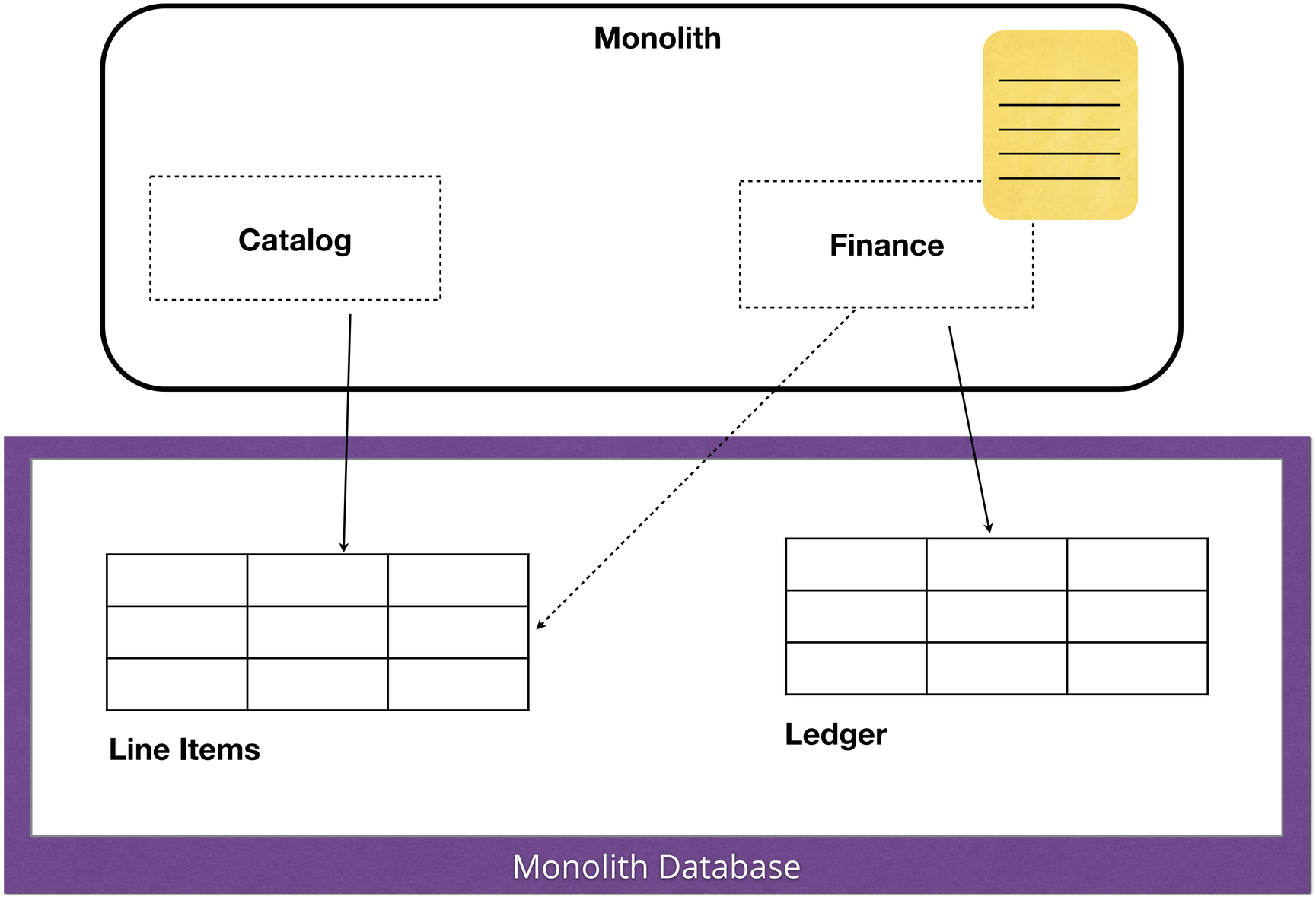
JOINS ACROSS TABLES



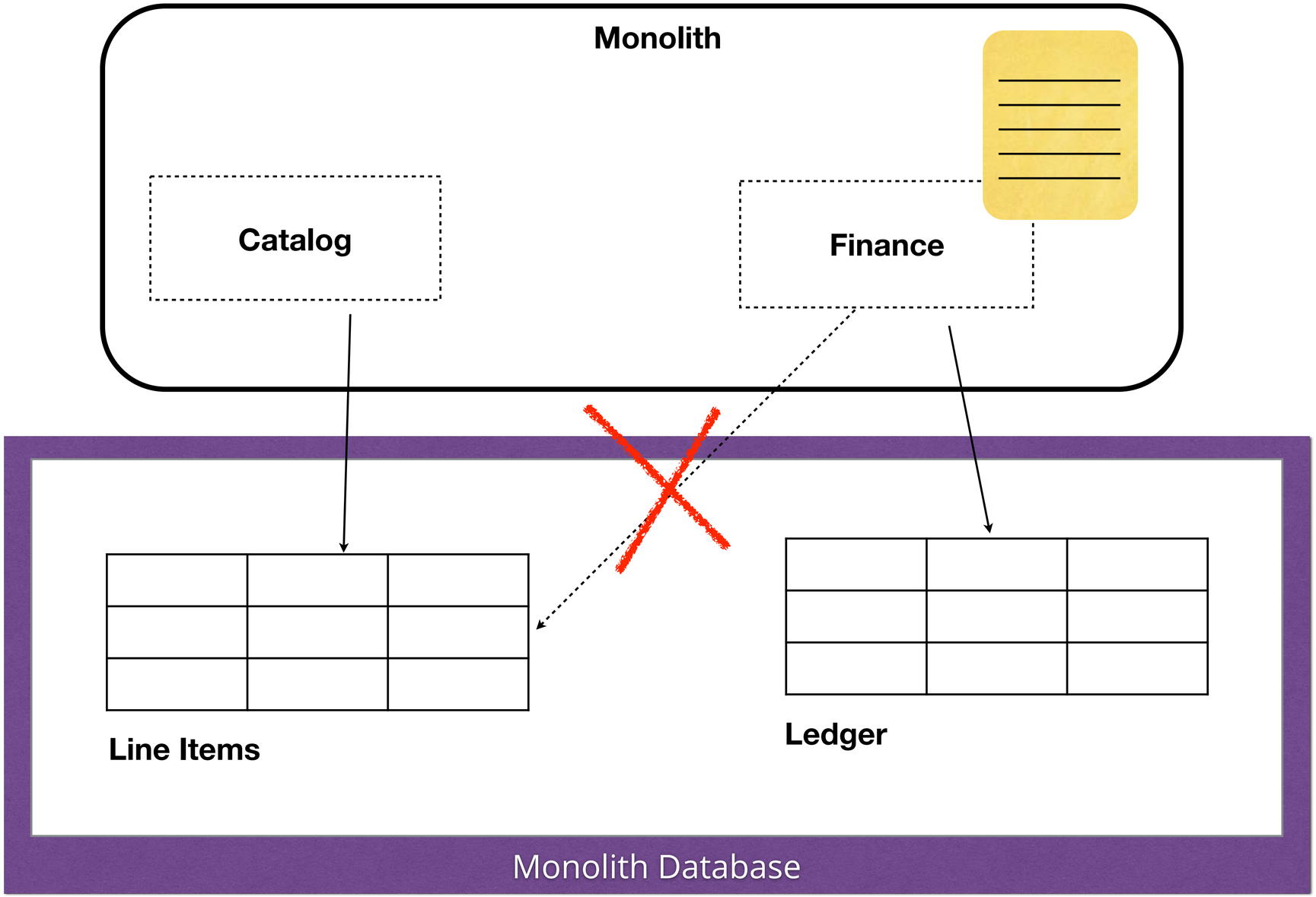
JOINS ACROSS TABLES



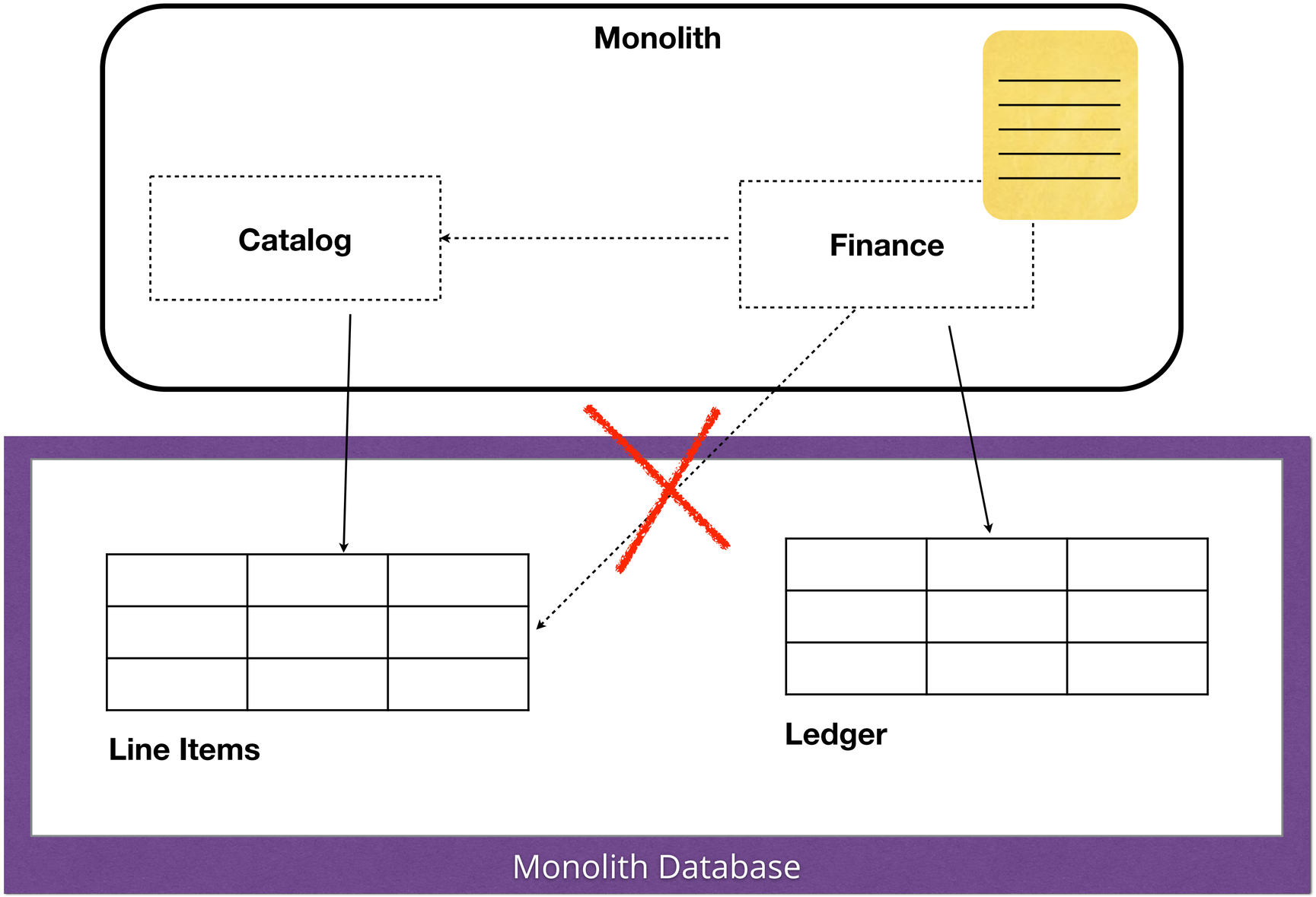
JOINS ACROSS TABLES

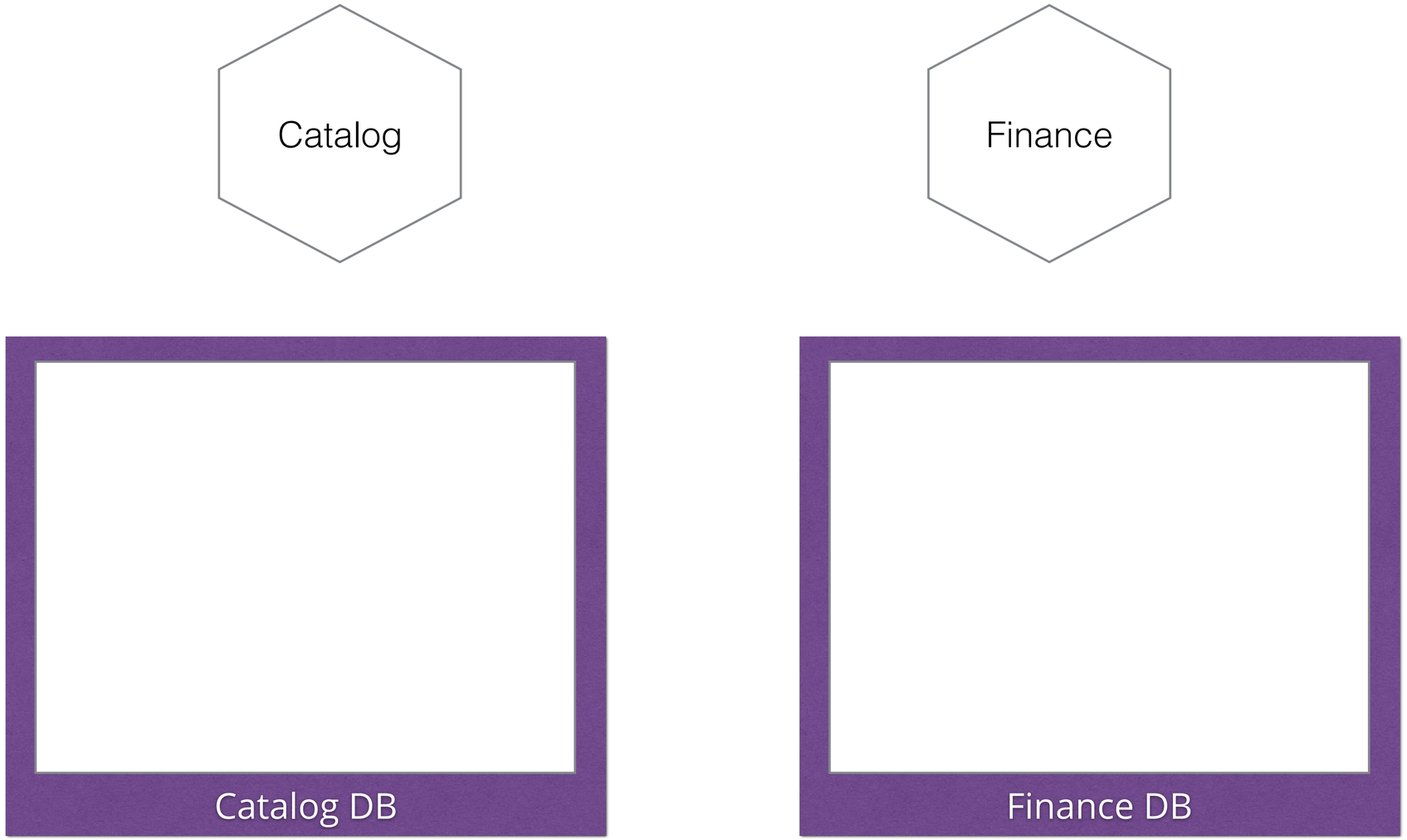


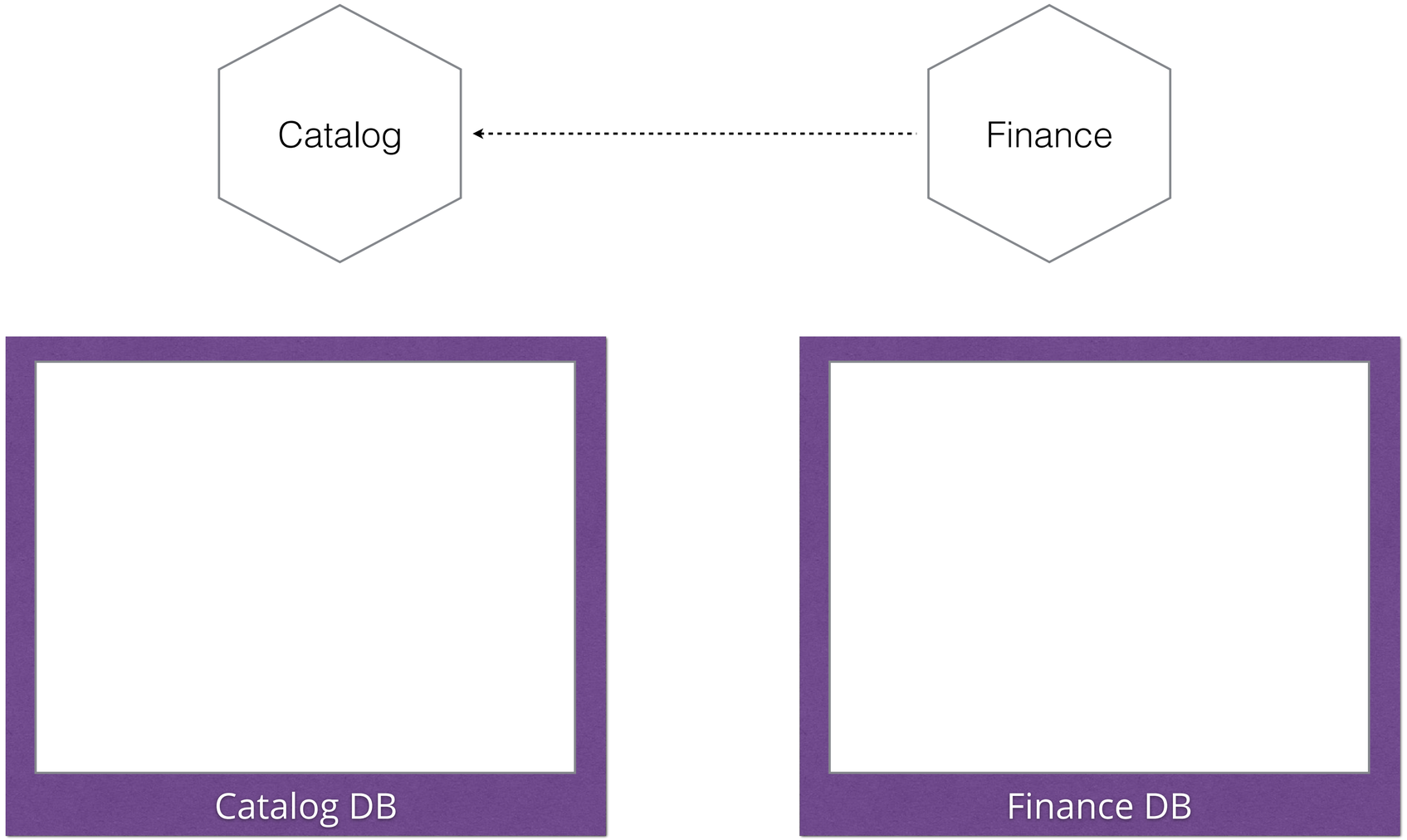
JOINS ACROSS TABLES

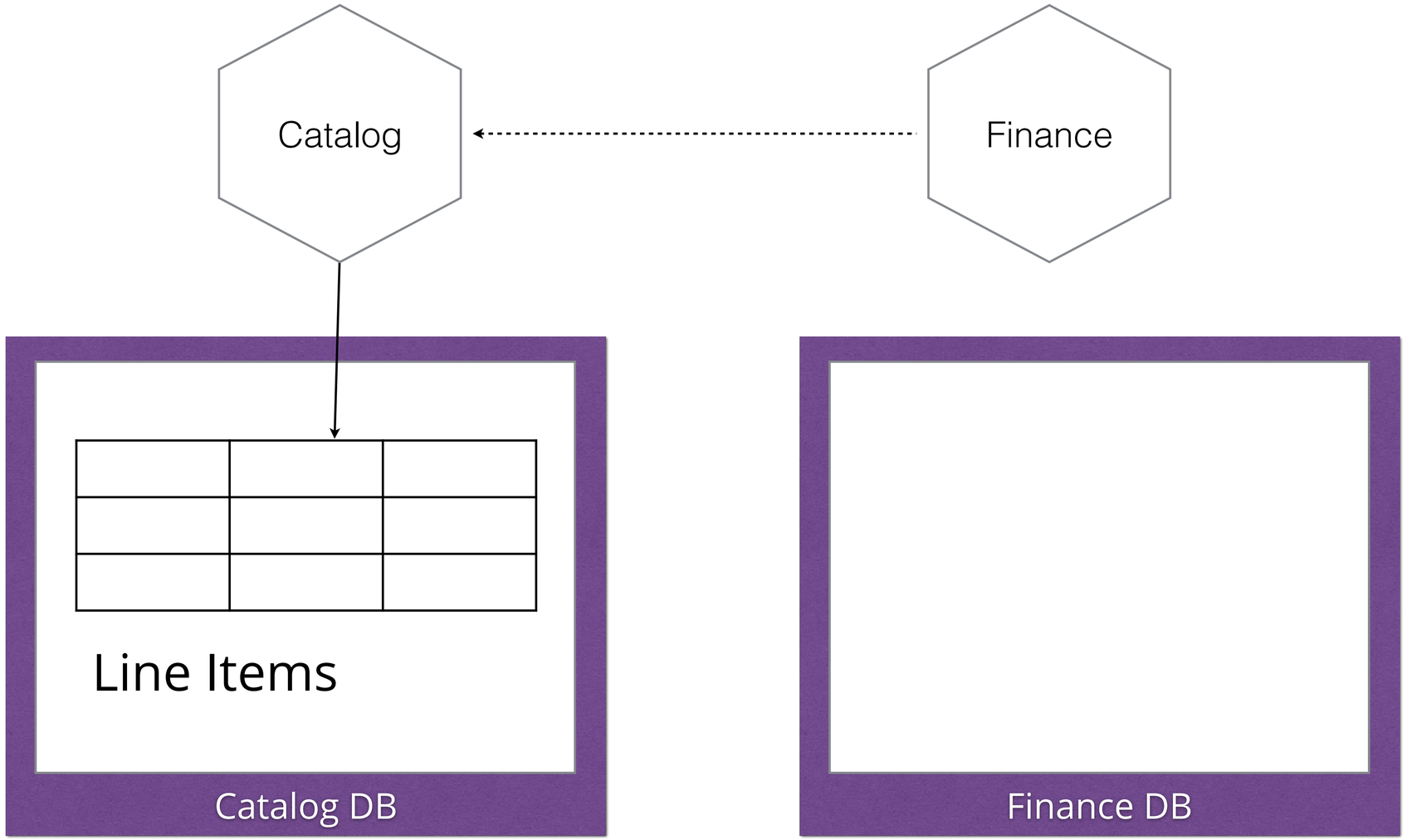


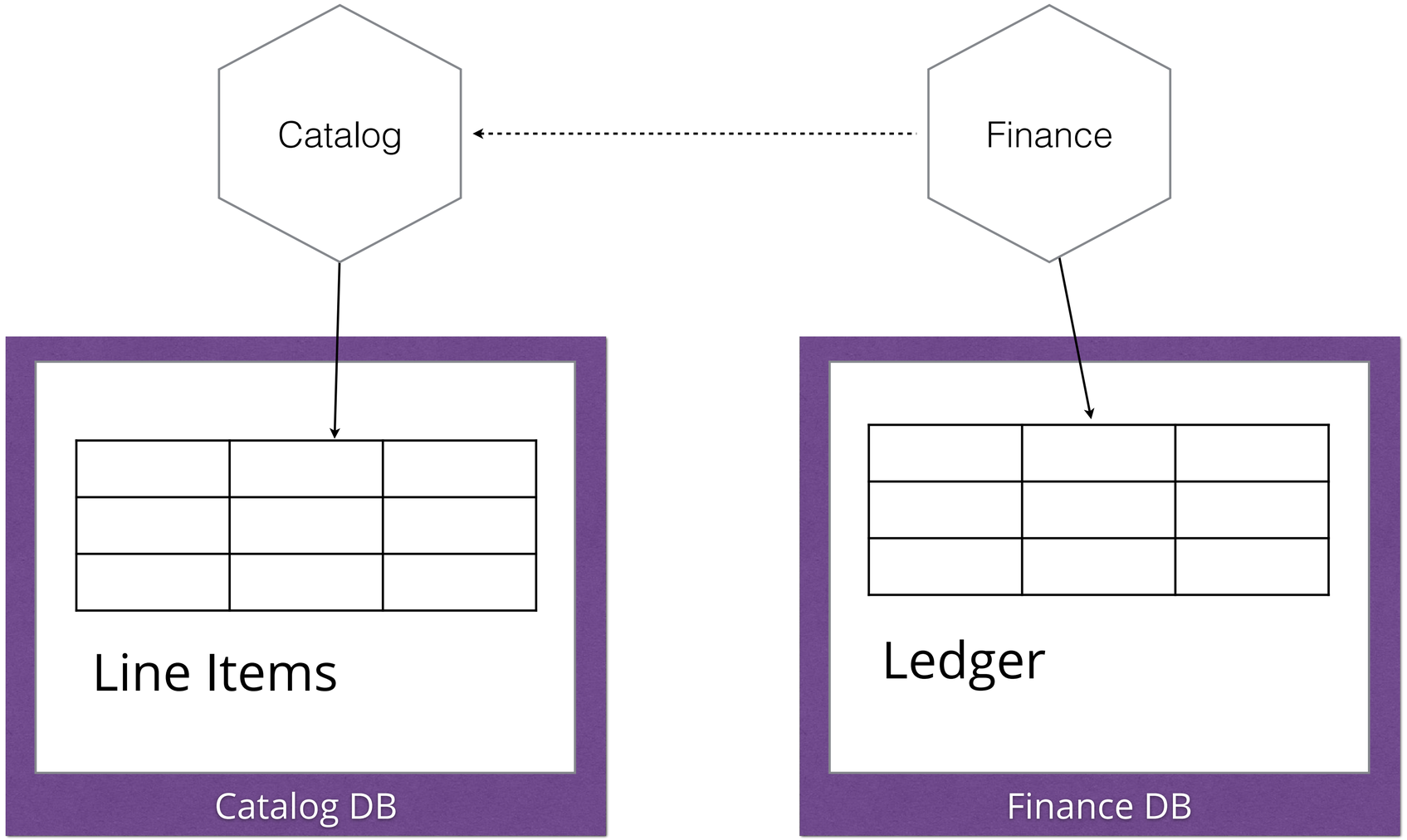
JOINS ACROSS TABLES




Catalog
Finance
Catalog DB
Finance DB


Catalog
Finance
Catalog DB
Finance DB


Catalog
Finance
Line Items
Catalog DB
Finance DB

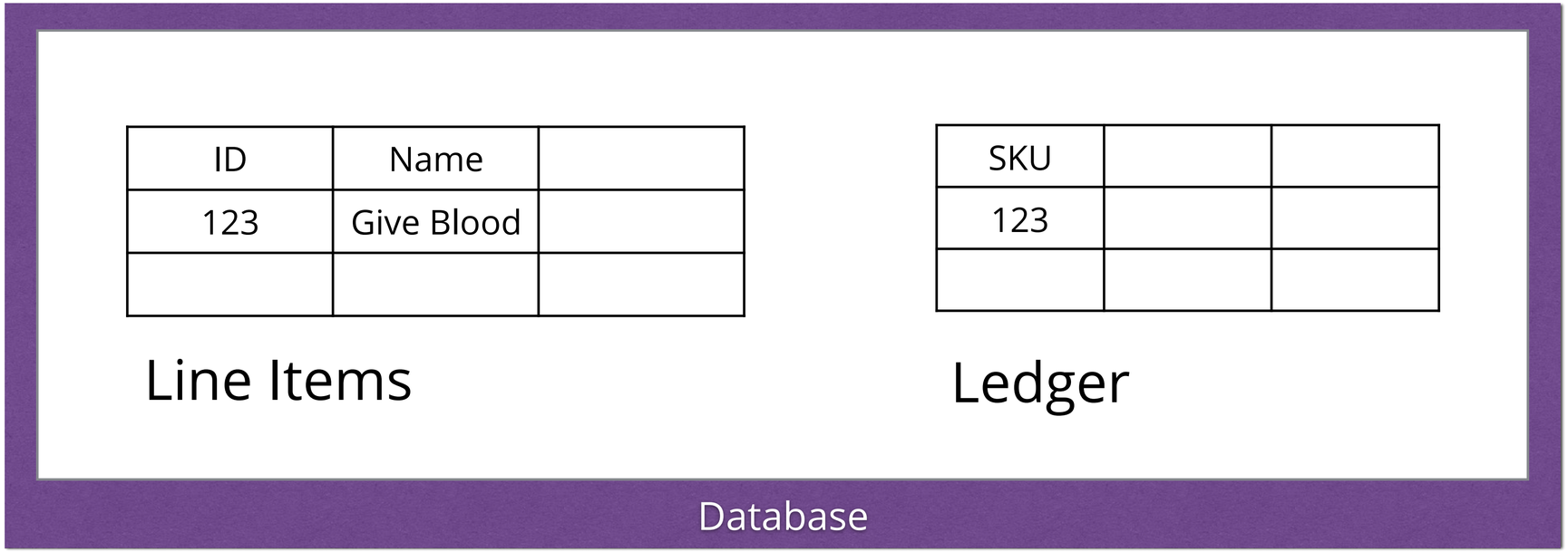

Catalog
Finance
Line Items
Catalog DB
Ledger
Finance DB

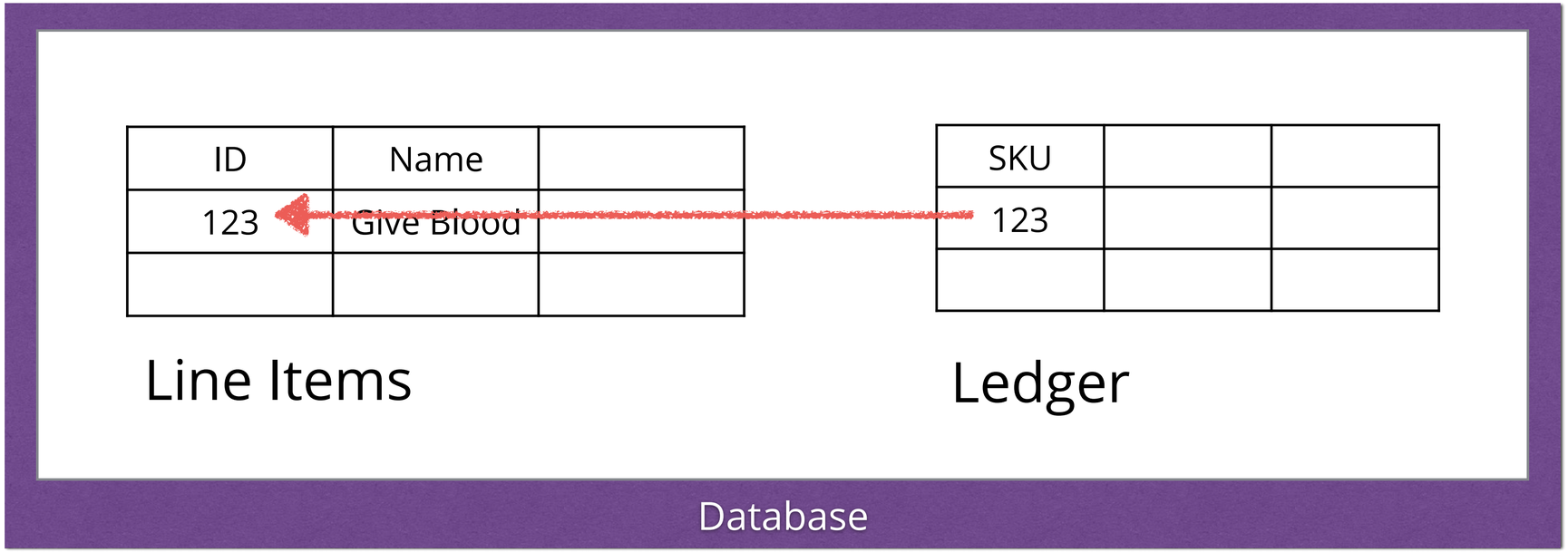

Database

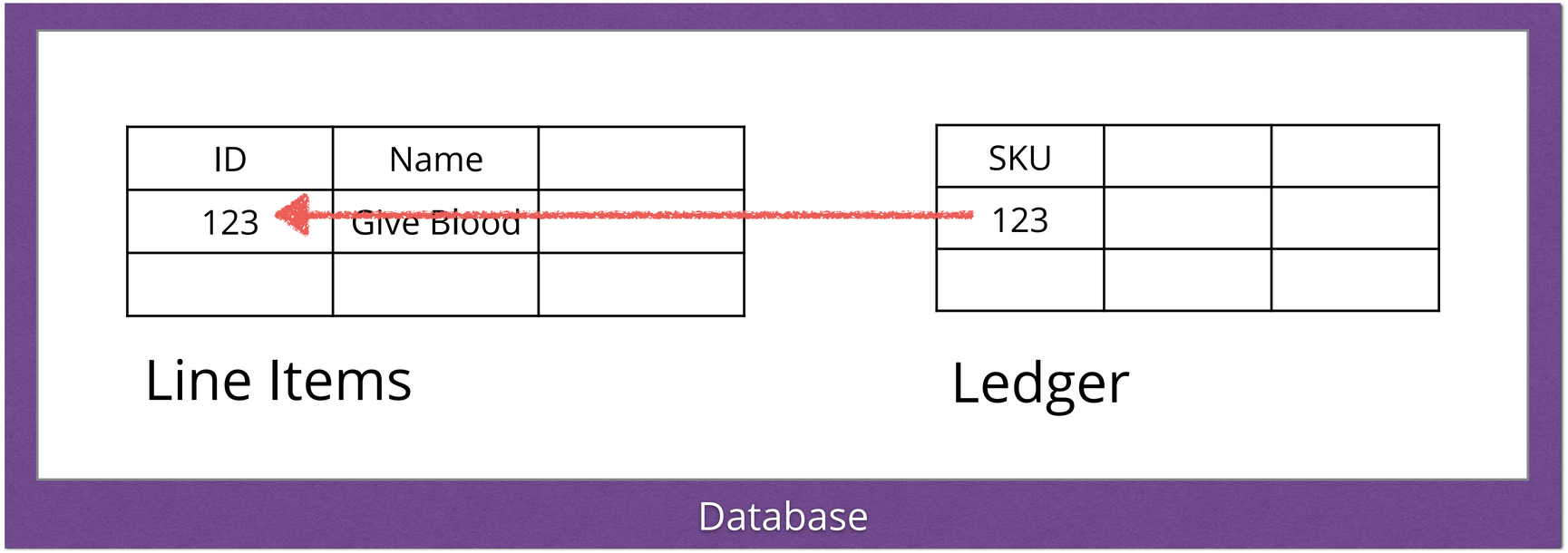
ID	Name	
123	Give Blood	

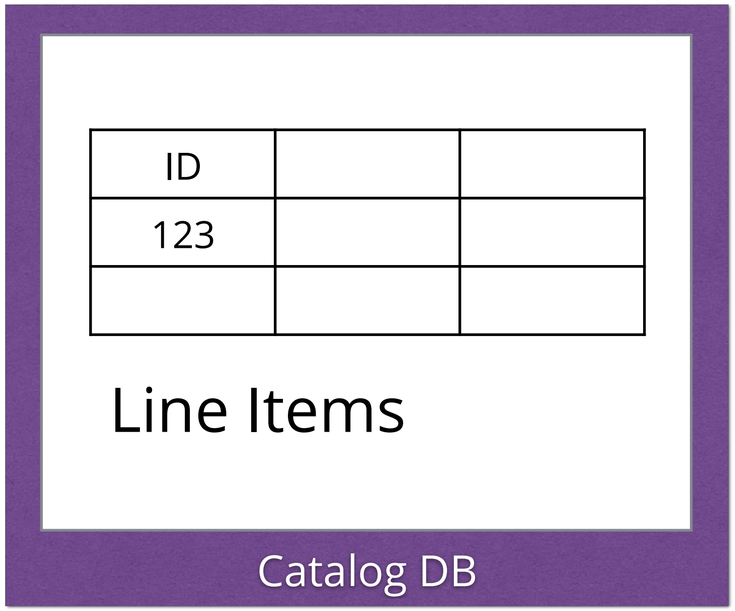
Line Items

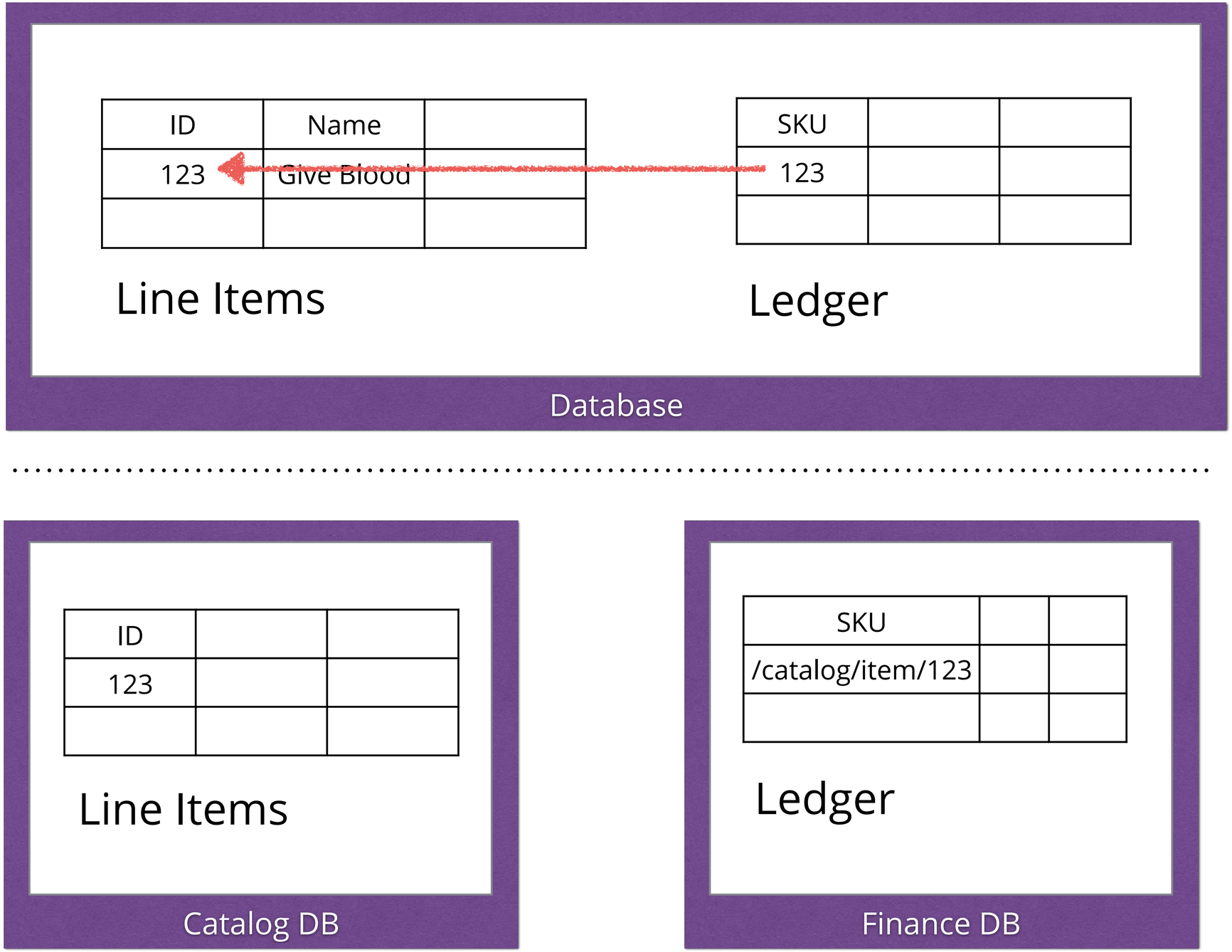
Database

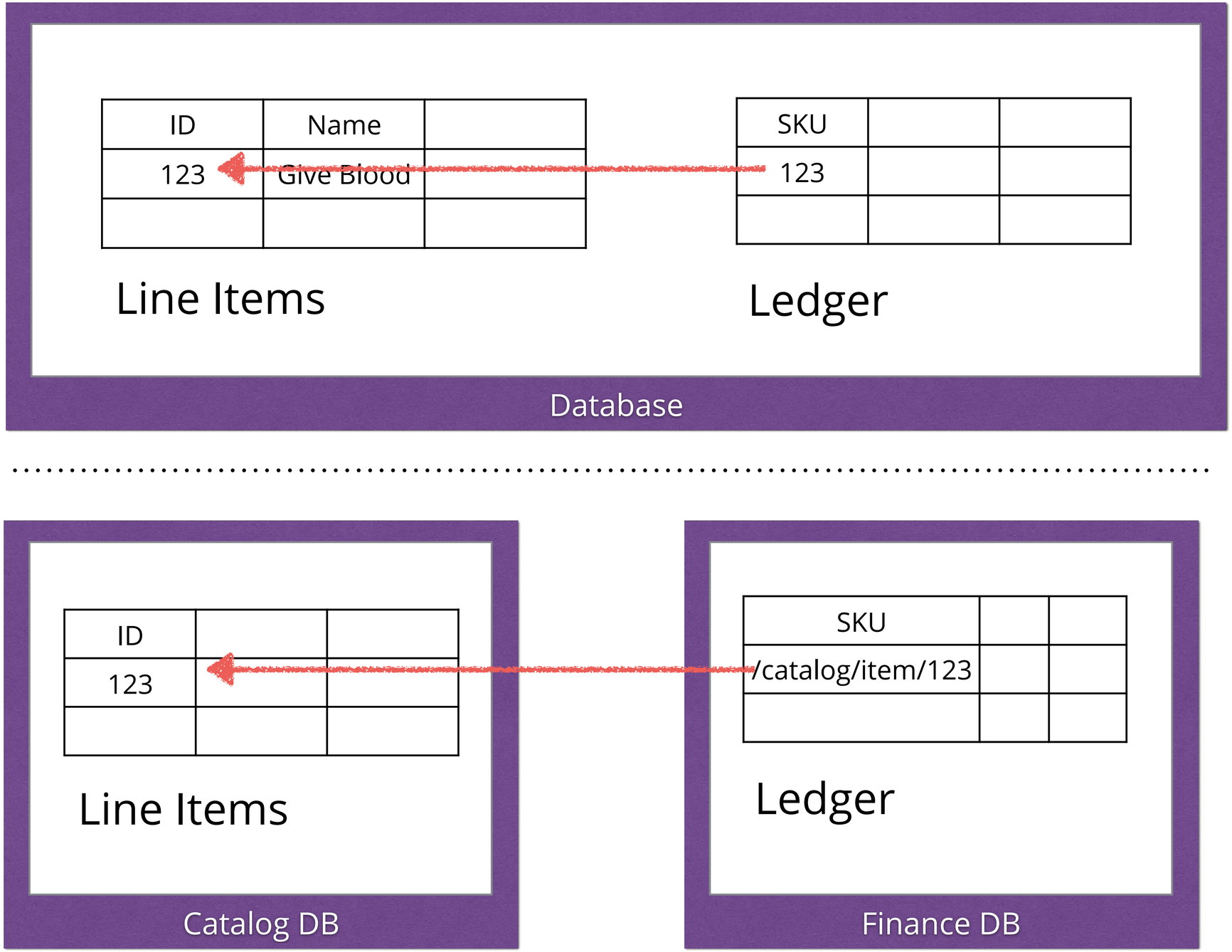

ID
Name
123
Give Blood
Line Items
SKU
123
Ledger
Database

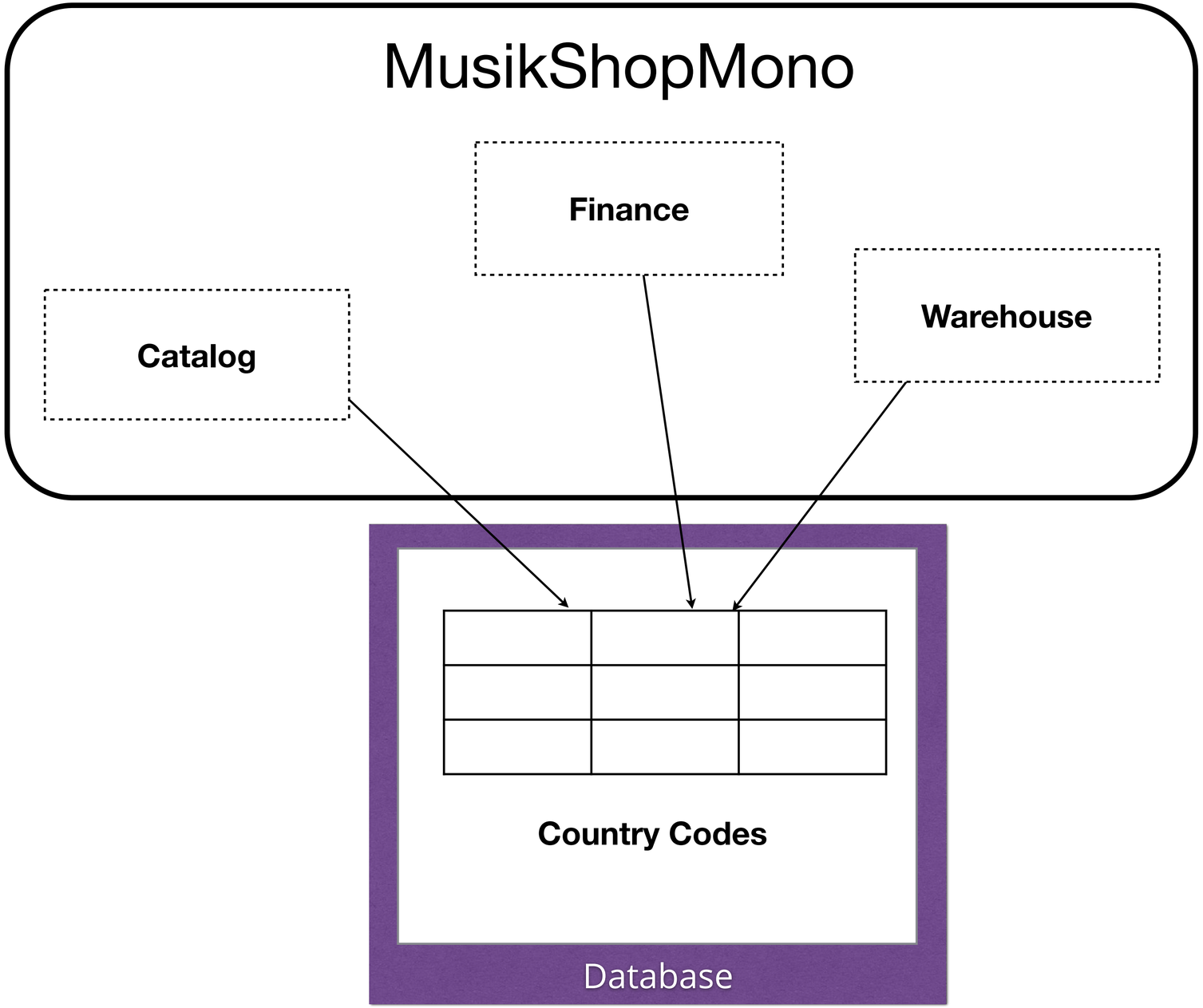

ID
Name
123
Give Blood
Line Items
SKU
123
Ledger
Database


ID
Name
123
Give Blood
SKU
123
Line Items
Ledger
Database

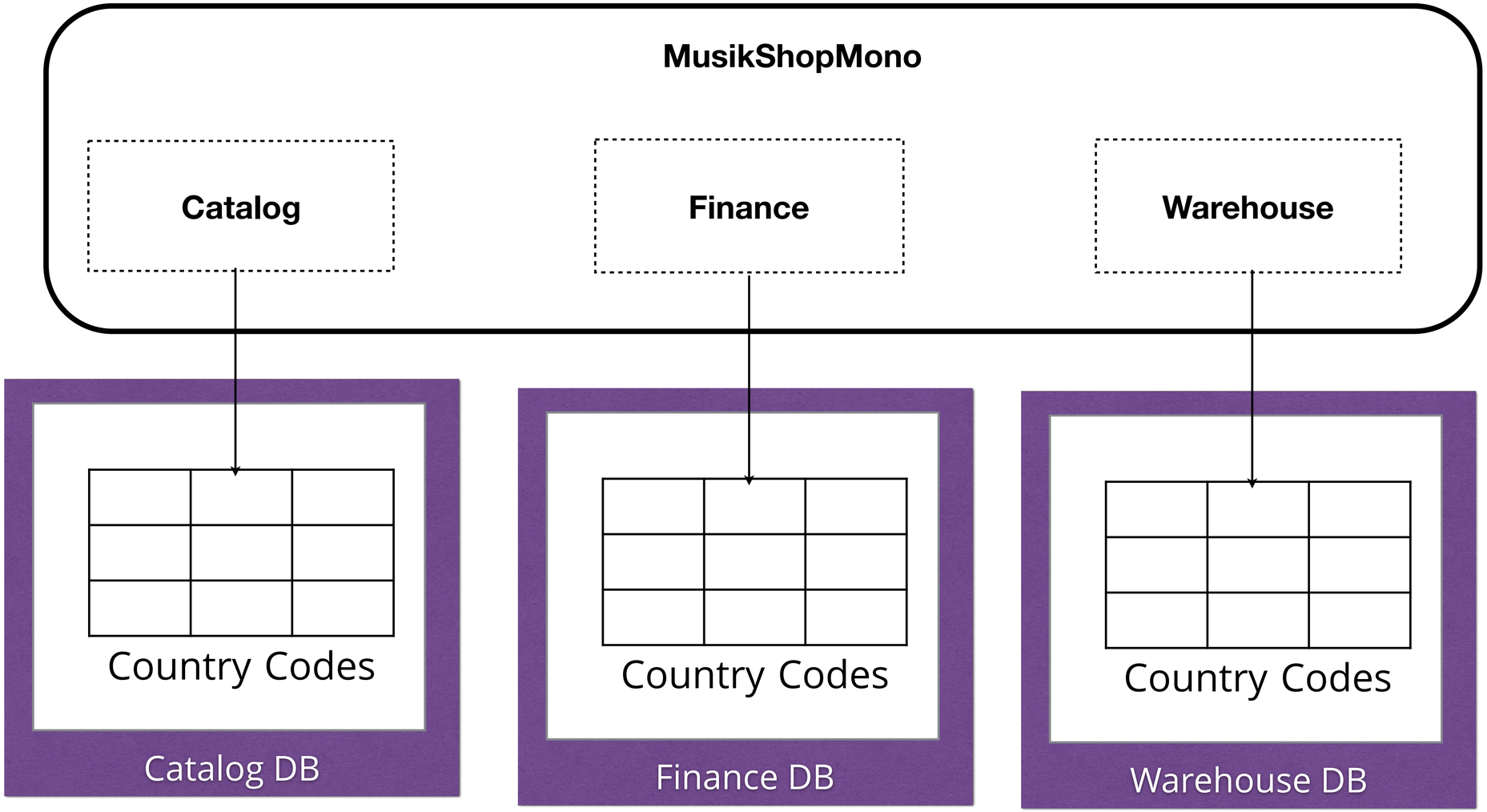

ID
123
Line Items
Catalog DB


ID
Name
123
Give Blood
SKU
123
Line Items
Ledger
Database
ID
123
Line Items
Catalog DB
SKU
/catalog/item/123
Ledger
Finance DB


ID
Name
123
Give Blood
SKU
123
Line Items
Ledger
Database
ID
123
SKU
/catalog/item/123
Line Items
Ledger
Catalog DB
Finance DB


MusikShopMono
Finance
Catalog
Warehouse
Country Codes
Database

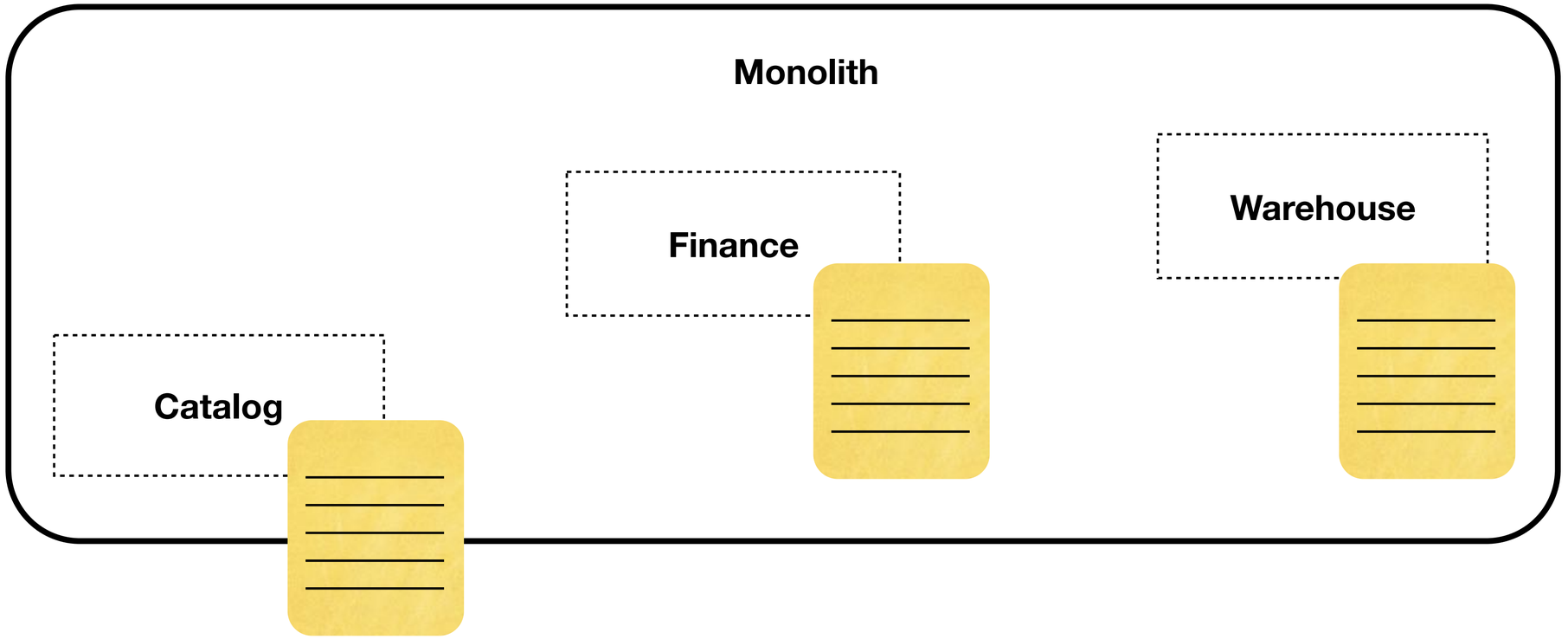
DUPLICATE THE DATA



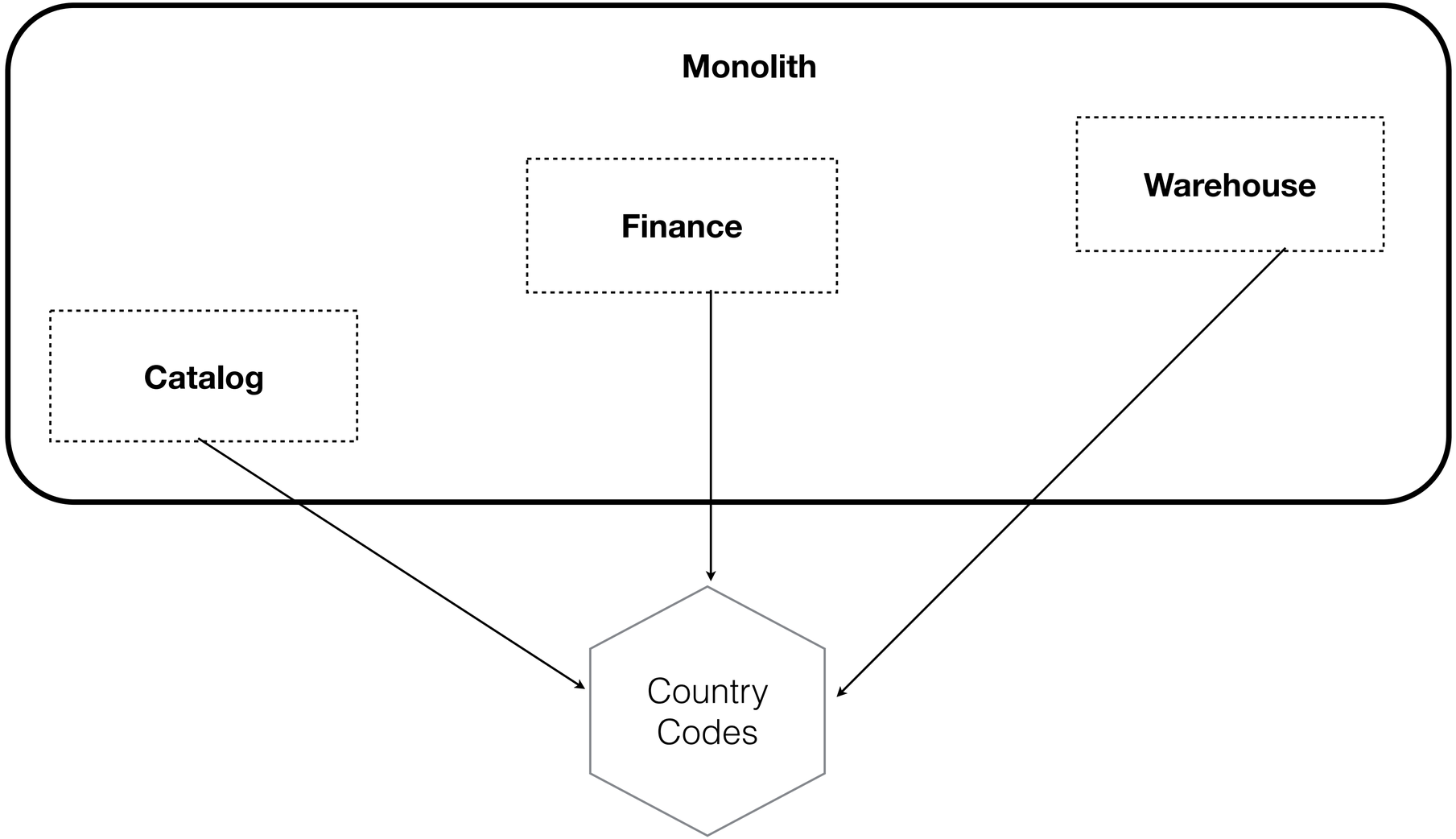
Duplication isn’t always bad…

…but watch for inconsistency

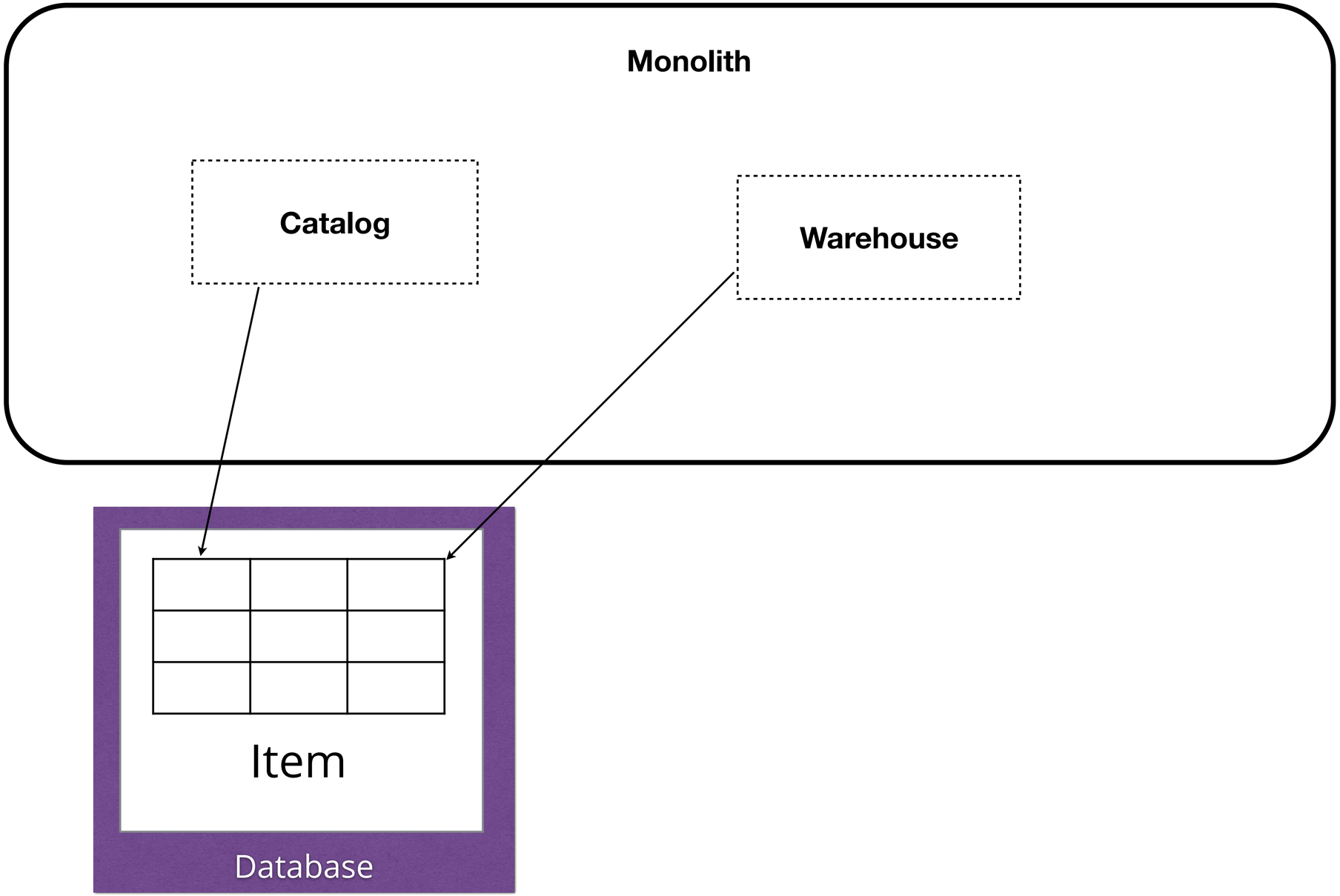
MOVE DATA INTO CONFIGURATION, OR LIBRARIES



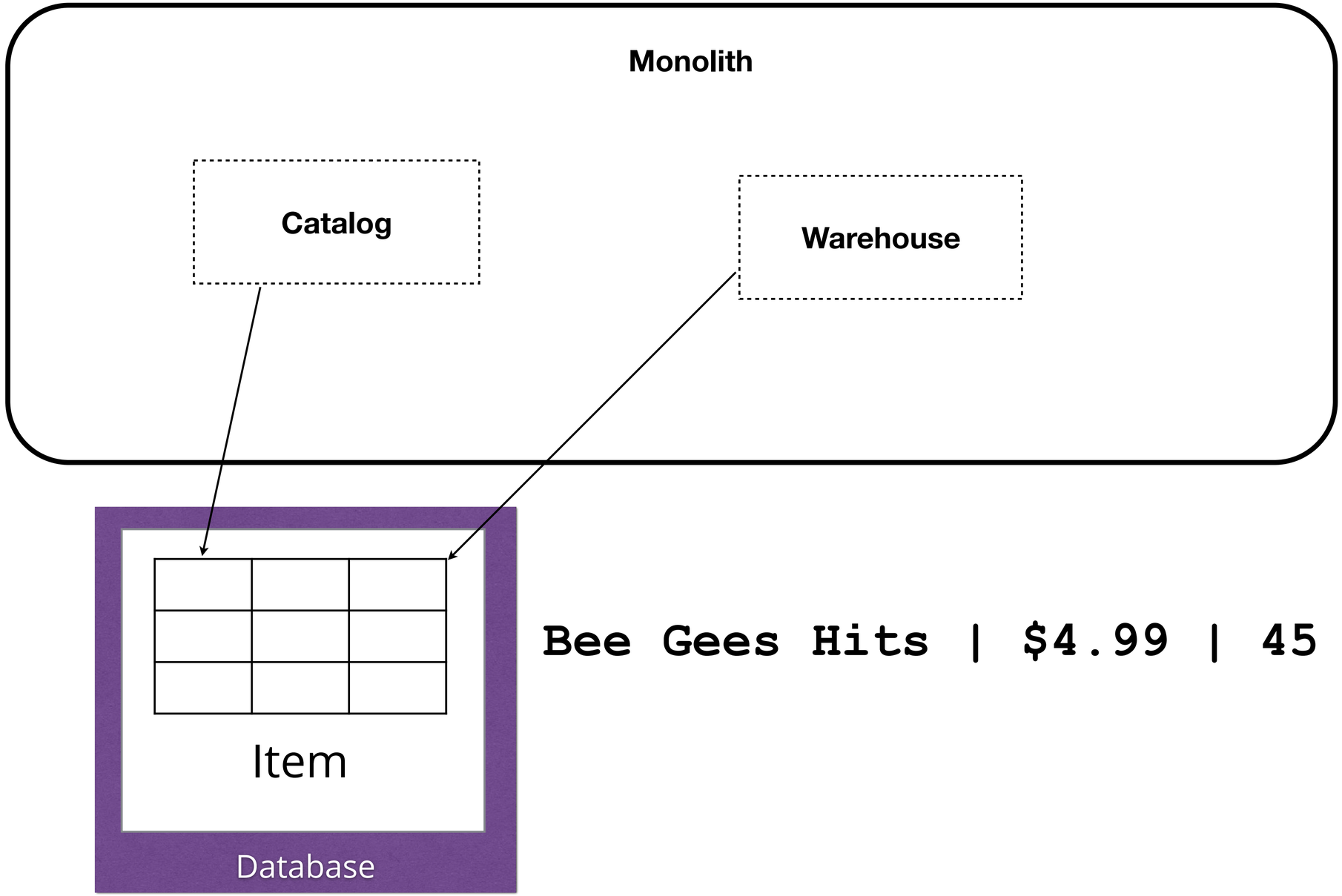
EXTRACT A SERVICE



SPLITTING TABLES

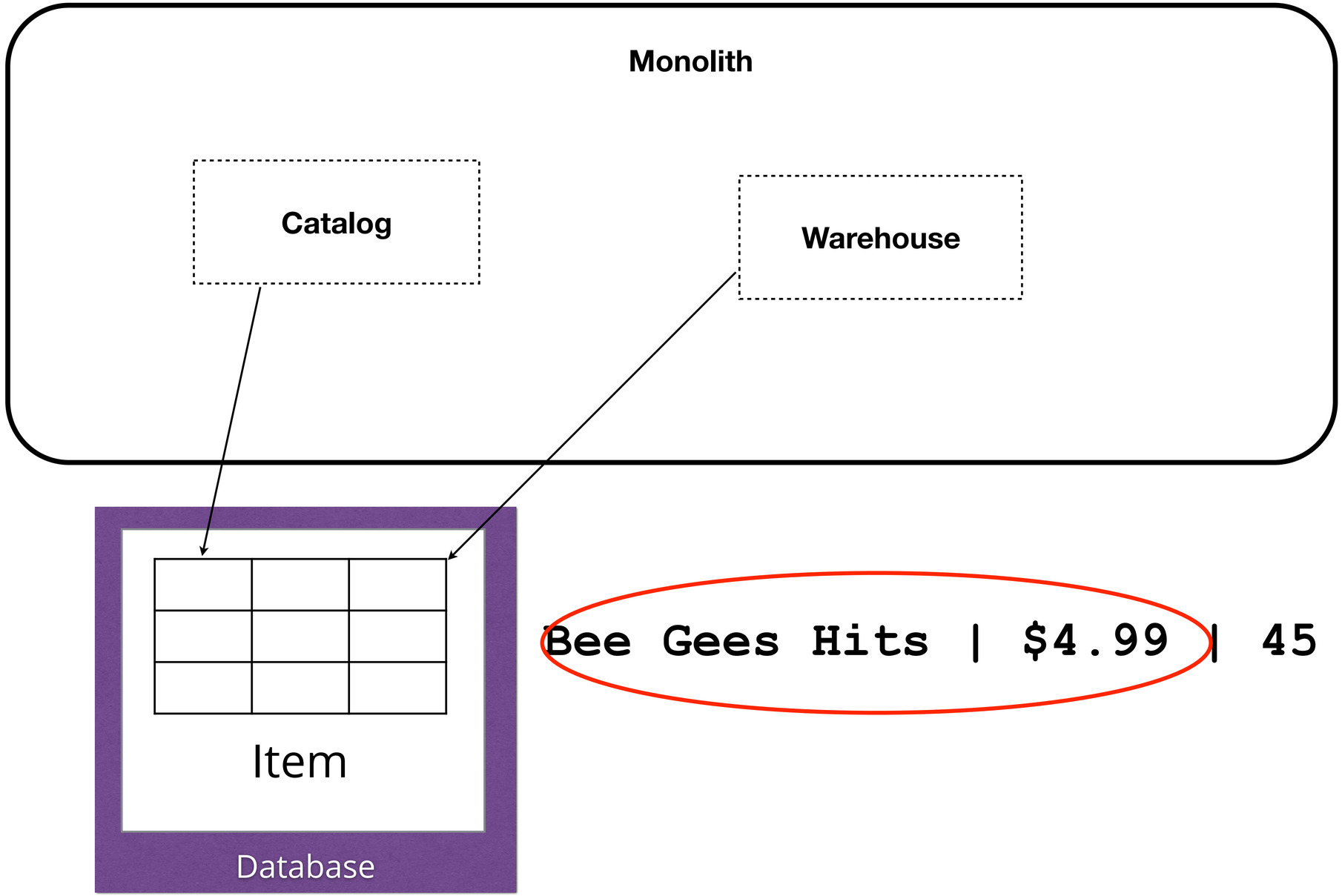


SPLITTING TABLES

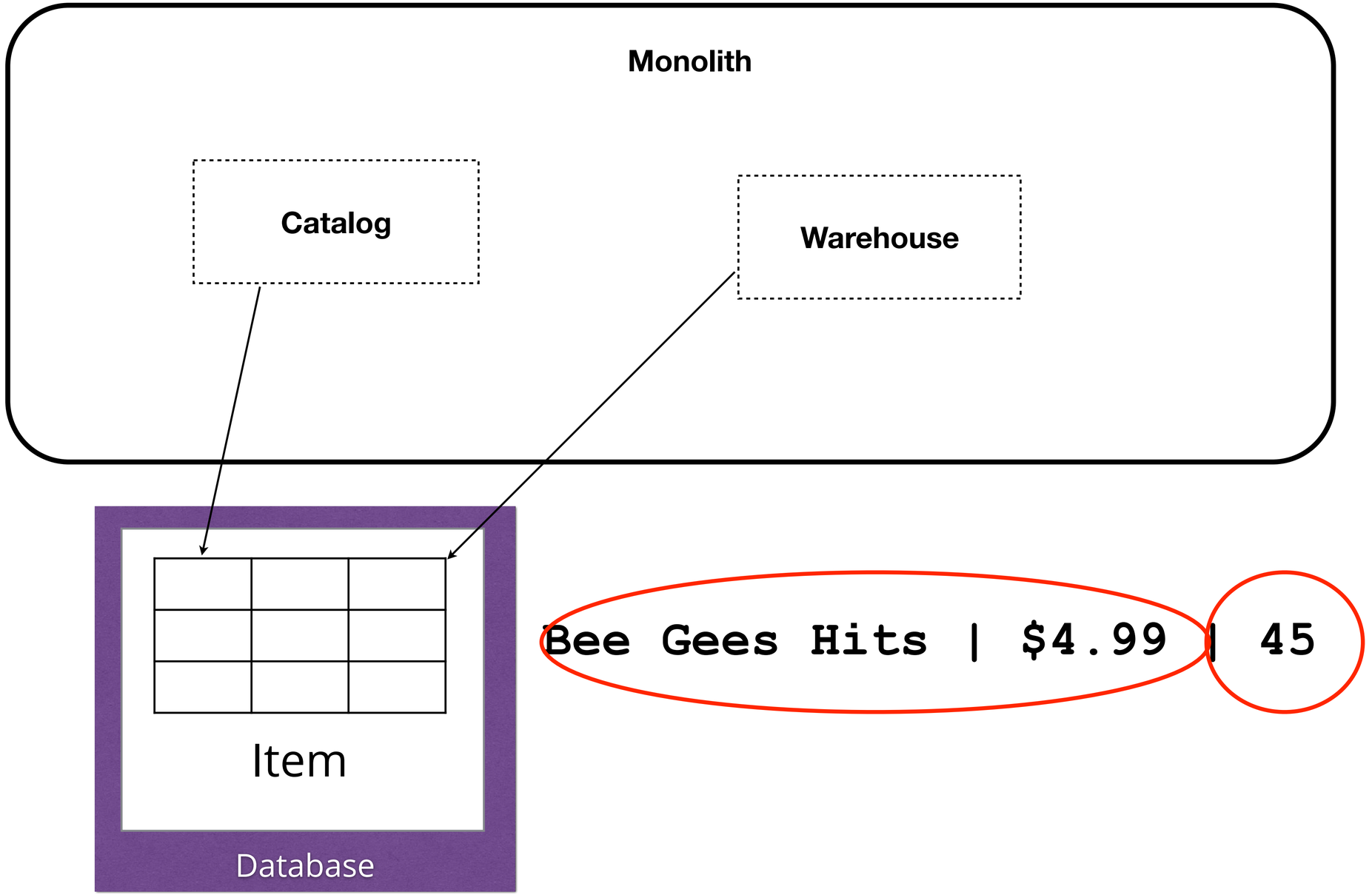


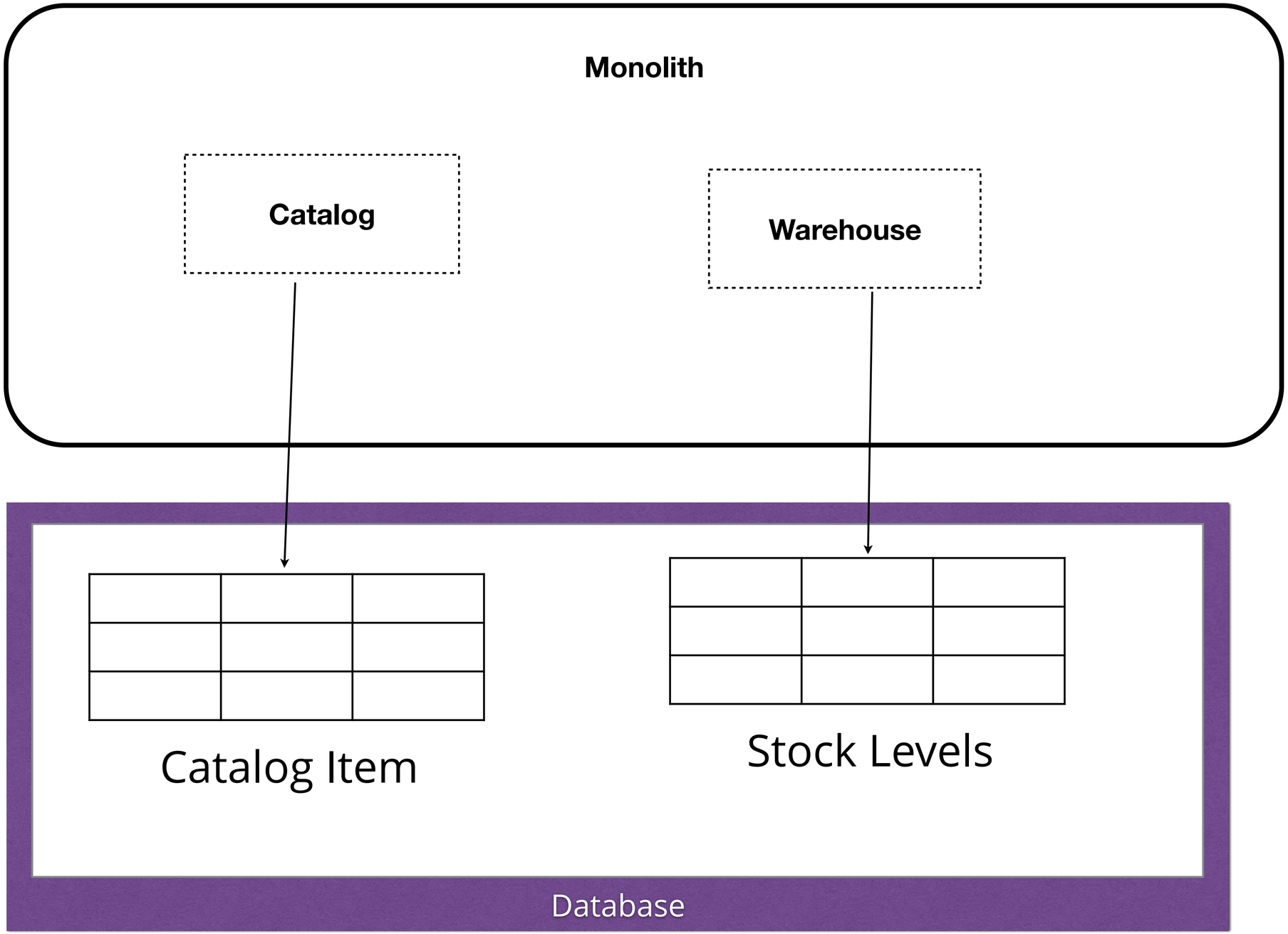

```
Bee Gees Hits | $4.99 | 45
```

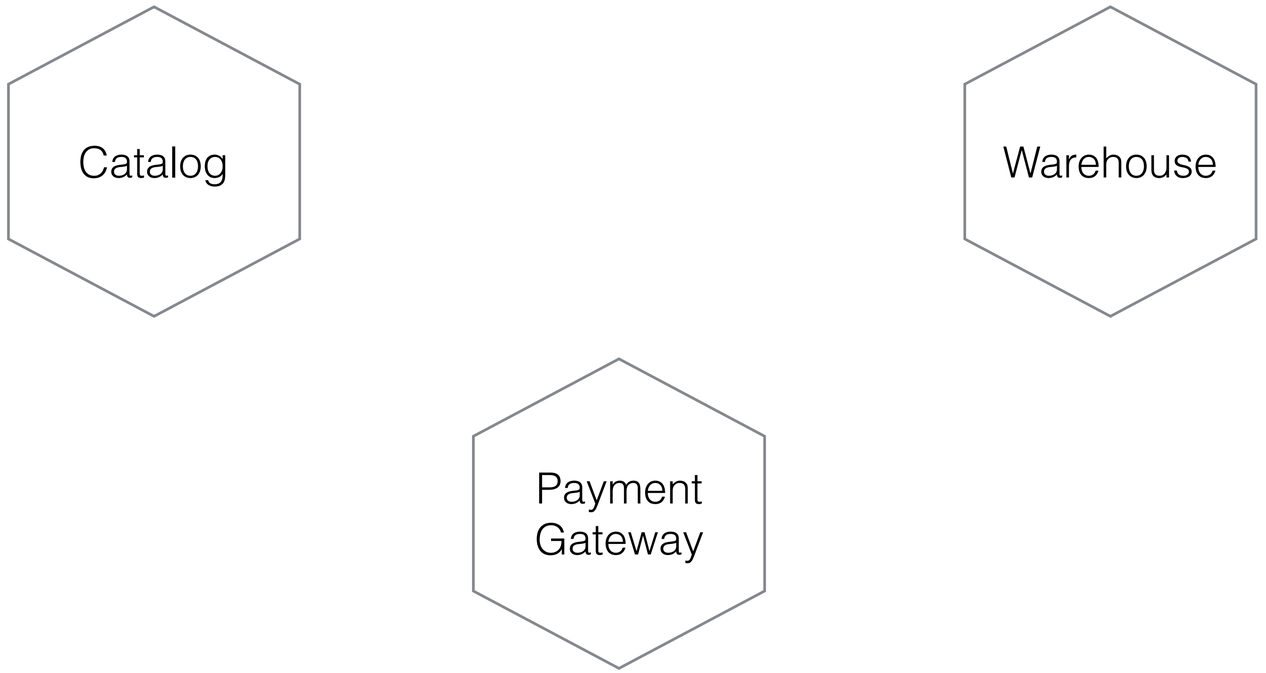
SPLITTING TABLES

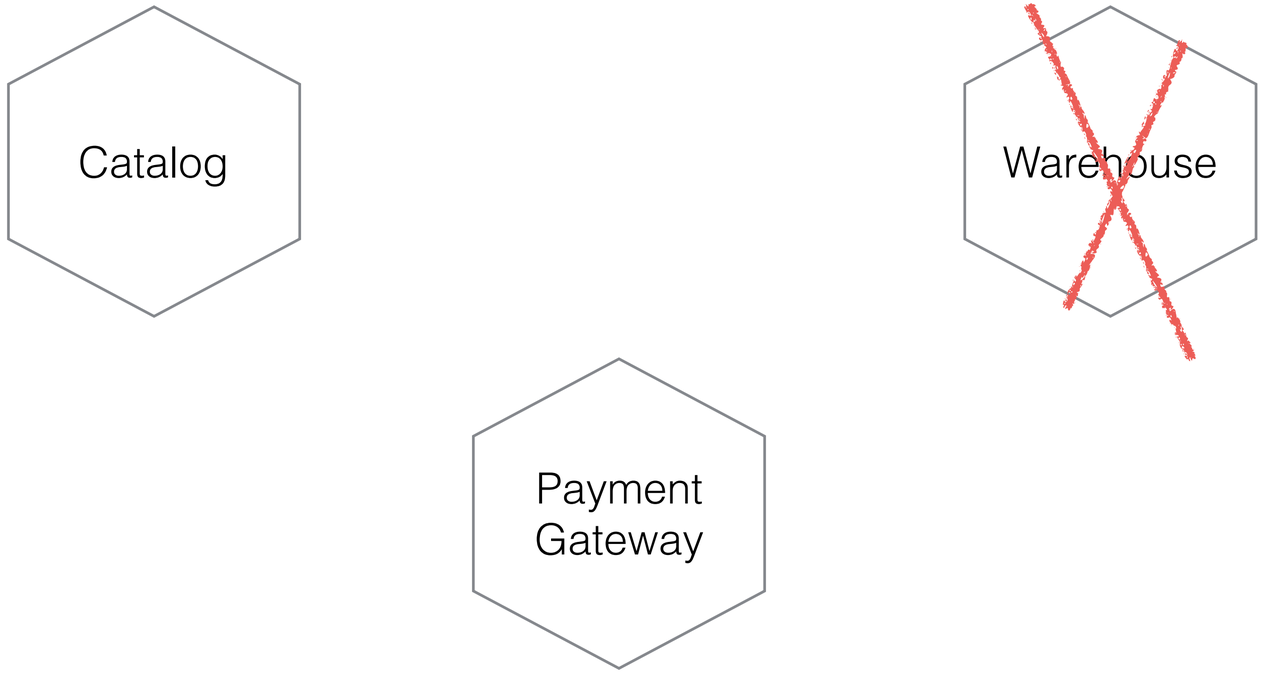


SPLITTING TABLES

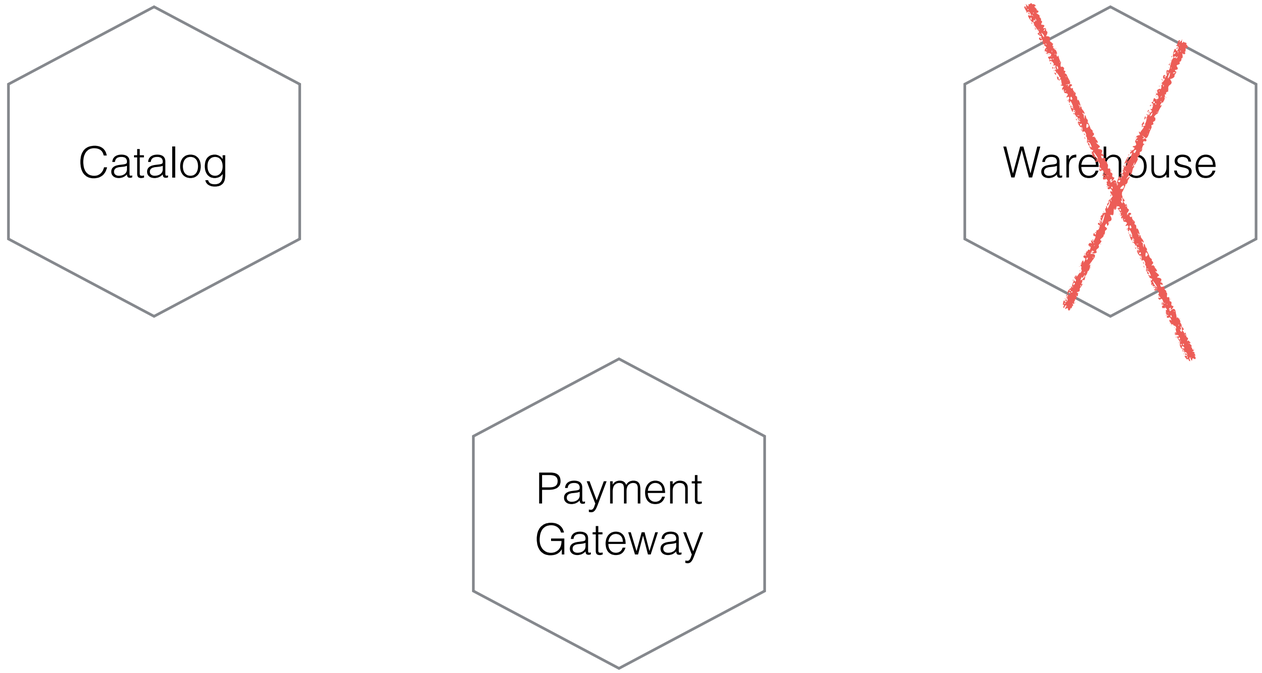



Monolith
Catalog
Warehouse
Catalog Item
Stock Levels
Database

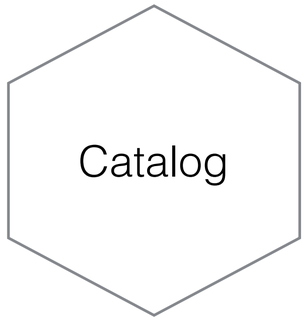

Catalog
Warehouse
Payment
Gateway

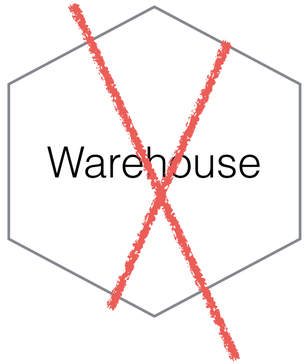

Catalog
Warehouse
Payment
Gateway

We can find out how much something costs…



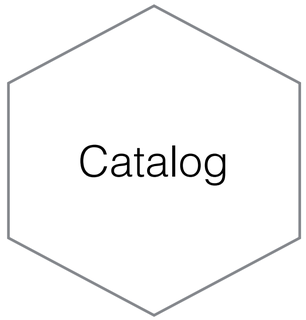
We can find out how much something costs…



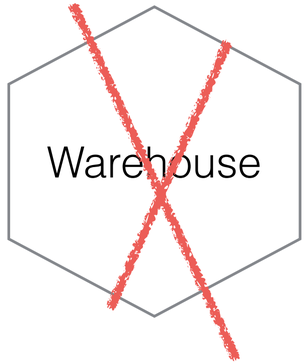


…and can take payment…

We can find out how much something costs…



…but can't check stock levels!



…and can take payment…

Should we keep selling CDs in this situation?

CAP Theory

CAP Theory

In a partition, you have to tradeoff between consistency and availability

Take the money?

Take the money?

Favouring availability over consistency

Don’t take the money?

Don’t take the money?

Favouring consistency over availability

This doesn’t have to be black and white…

This doesn’t have to be black and white…

```
Last view of stock: 1 hour ago
# Items:            100
```

This doesn’t have to be black and white…

```
Last view of stock: 1 hour ago    Sell item
# Items:            100
```

This doesn’t have to be black and white…

```
Last view of stock: 1 hour ago    Sell item
# Items:            100

Last view of stock: 1 hour ago
# Items:            1
```

This doesn’t have to be black and white…

```
Last view of stock: 1 hour ago
# Items:            100
```

Sell item

```
Last view of stock: 1 hour ago
# Items:            1
```

Don’t sell item

This doesn’t have to be black and white…

```
Last view of stock: 1 hour ago
# Items:            100
```

Sell item

```
Last view of stock: 1 hour ago
# Items:            1
```

Don’t sell item

```
Last view of stock: 2 days ago
# Items:            100
```

This doesn’t have to be black and white…

```
Last view of stock: 1 hour ago     Sell item
# Items:            100

Last view of stock: 1 hour ago     Don’t sell item
# Items:            1

Last view of stock: 2 days ago     Don’t sell item
# Items:            100
```

Memories, Guesses, and Apologies

Rate this article ☆☆☆☆☆

Pat Helland May 15, 2007

Well, here I am blogging on the bus with my newly installed Windows Live Writer!!!

This blog is a text version of a five minute "Gong Show" presentation I did at CIDR (Conference on Innovative Database Research) on Jan 8,2007.

All computing can be considered as: "Memories, Guesses, and Apologies". This is a personal opinion about how computers suck. Furthermore, it offers additional opinions about how we can take advantage of their sucki-ness. Lets dig into this...

Newton and Einstein

It used to be that we thought of computing as one big-ass mainframe. The database folks only thought about the database. Transactions (and transactional serializability) offered a crisp and clear perspective of how time marches forward uniformly. When working on transaction T(i), any other transaction T(j) can be perceived as occurring before T(i) or after T(i). If T(i) and T(j) are concurrently processed, the transaction system ensures that either order is correct without modifying the semantics. This offers a crisp and clear perspective of now. Time marches forward like a clock exactly as Newton envisaged his universe.

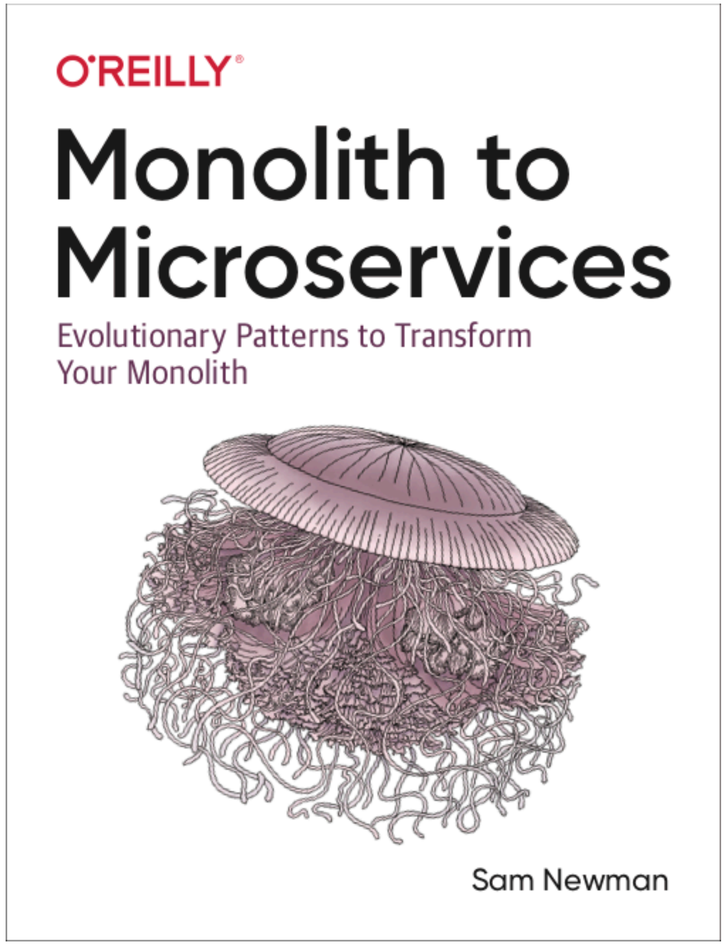
Nowadays, we have lots and lots of computers. Big ones, small ones, connected, disconnected, occasionally connected, etc. These computers each have their own perspective of time. When you see data, it is unlocked and an artifact of the past. *Time is subjective* with many different notions of now. This is very much the way Einstein revamped our understanding of the universe.

Moving to SOA is like moving from Newton's Universe to Einstein's Universe.

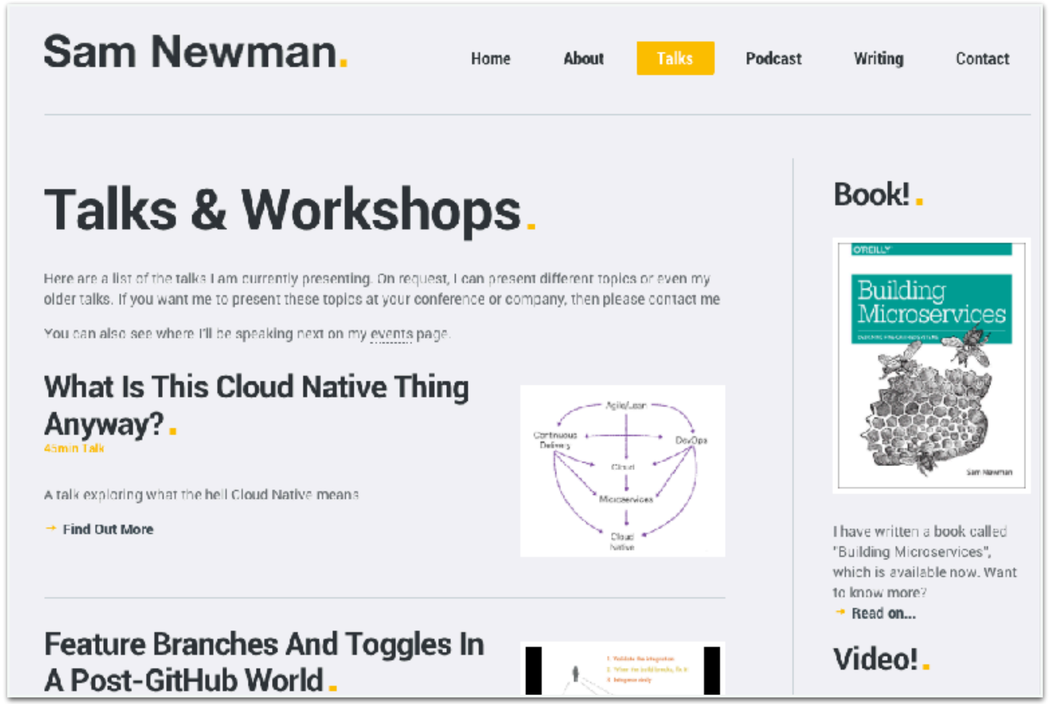
https://blogs.msdn.microsoft.com/pathelland/2007/05/15/memories-guesses-and-apologies/

Distributed systems that require strong consistency are hard to scale

THANKS!



Sam Newman & Associates



@samnewman

https://samnewman.io/