

# Modernizing Systems

---

with Microservices, Hystrix and RxJava

Holger Kraus, Arne Landwehr

W-JAX, München, Nov 4, 2015

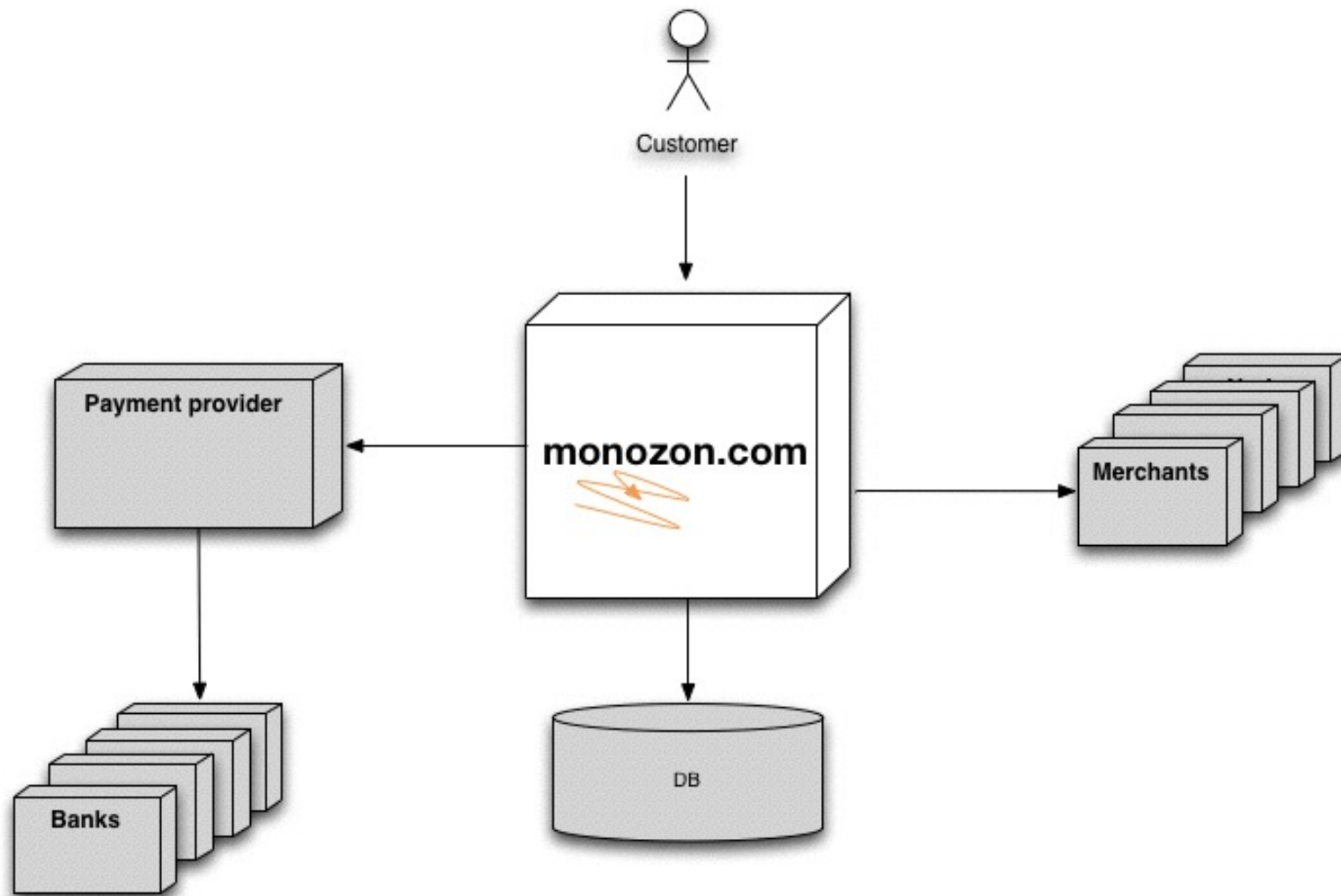


# A typical System

**monozon.com**



# The context



# Current problems

---

- › Maintenance is difficult
- › New features need a lot of time
- › Very unstable
- › Outdated technology
- › Doesn't scale

+ frustrated  
developers :(

# Current problems

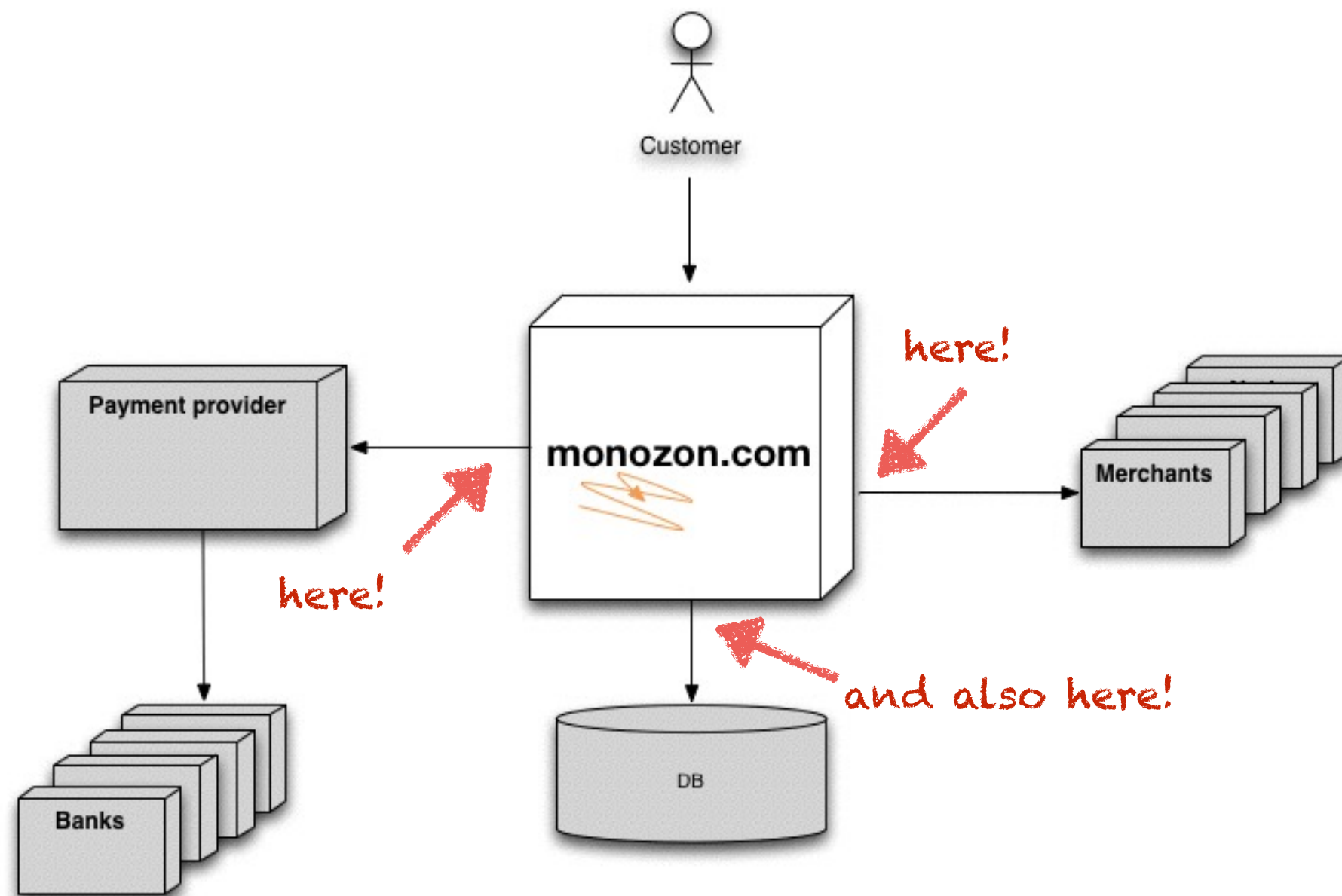
---

- › Maintenance is difficult
- › New features need a lot of time
- › Very unstable
- › Outdated technology
- › Doesn't scale

~~Microservices  
FTW!~~  
not yet ...

Stabilize first!

# External dependencies







# Cascading Failures



# Stability patterns

---

- › Timeouts
- › Circuit Breaker
- › Bulkhead



... Fail Fast, Steady State, Handshaking, Test Harness, Decoupling Middleware

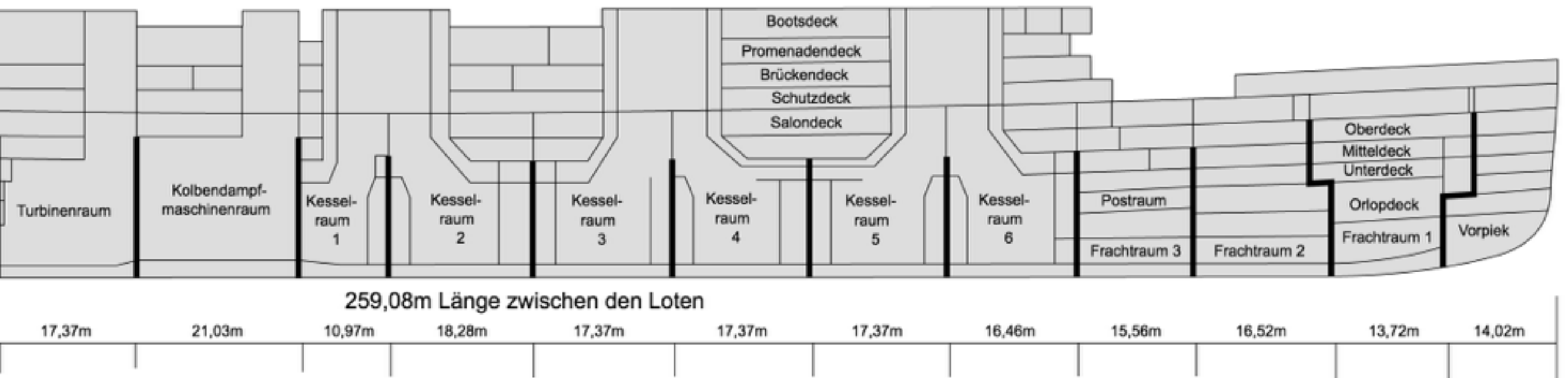
# Timeout



# Bulkheads



# Ships

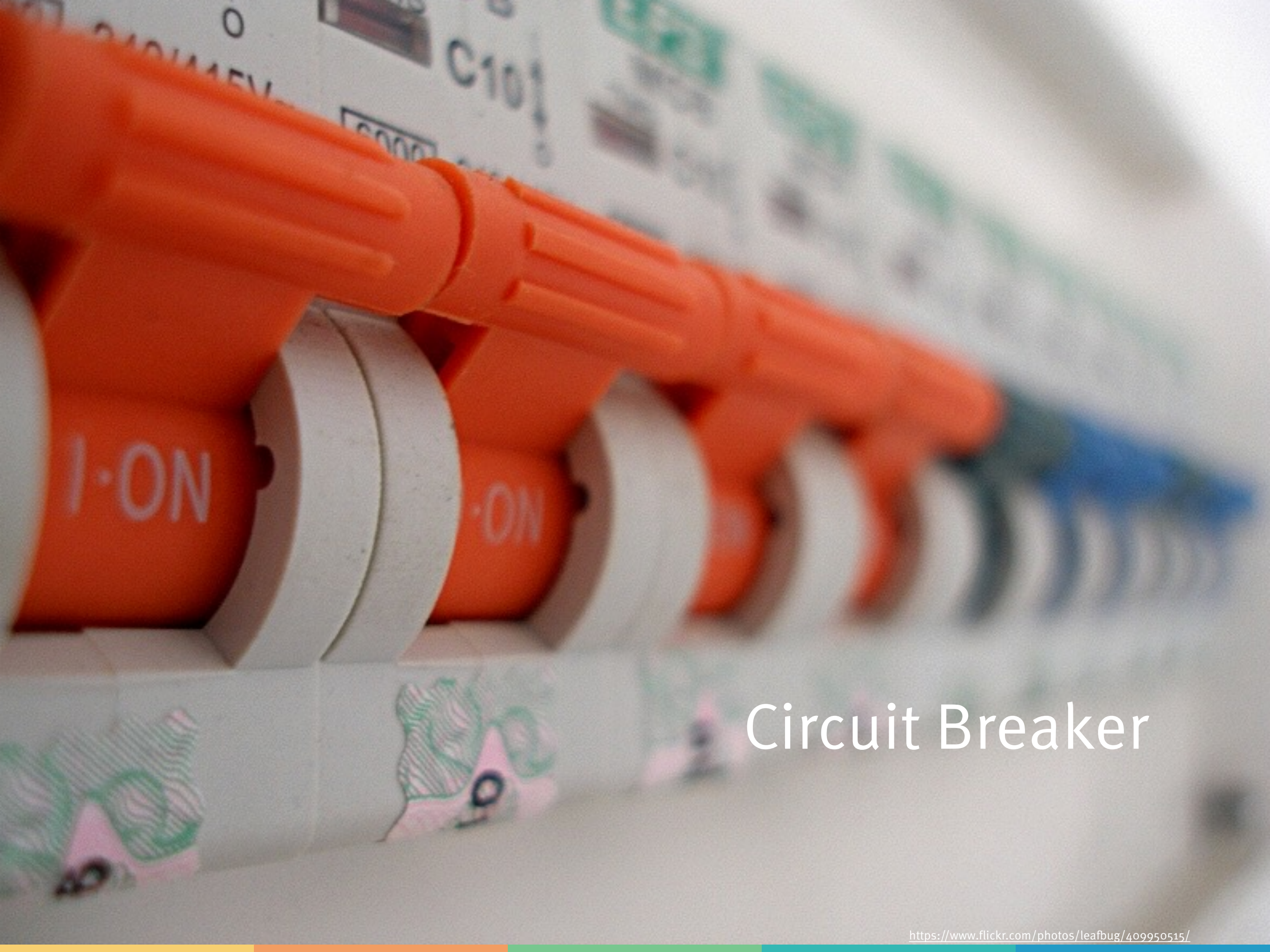


# Bulkheads and IT

---

- › Thread pools
- › Database connection pools
- › Instances
- › Server
- › Data center

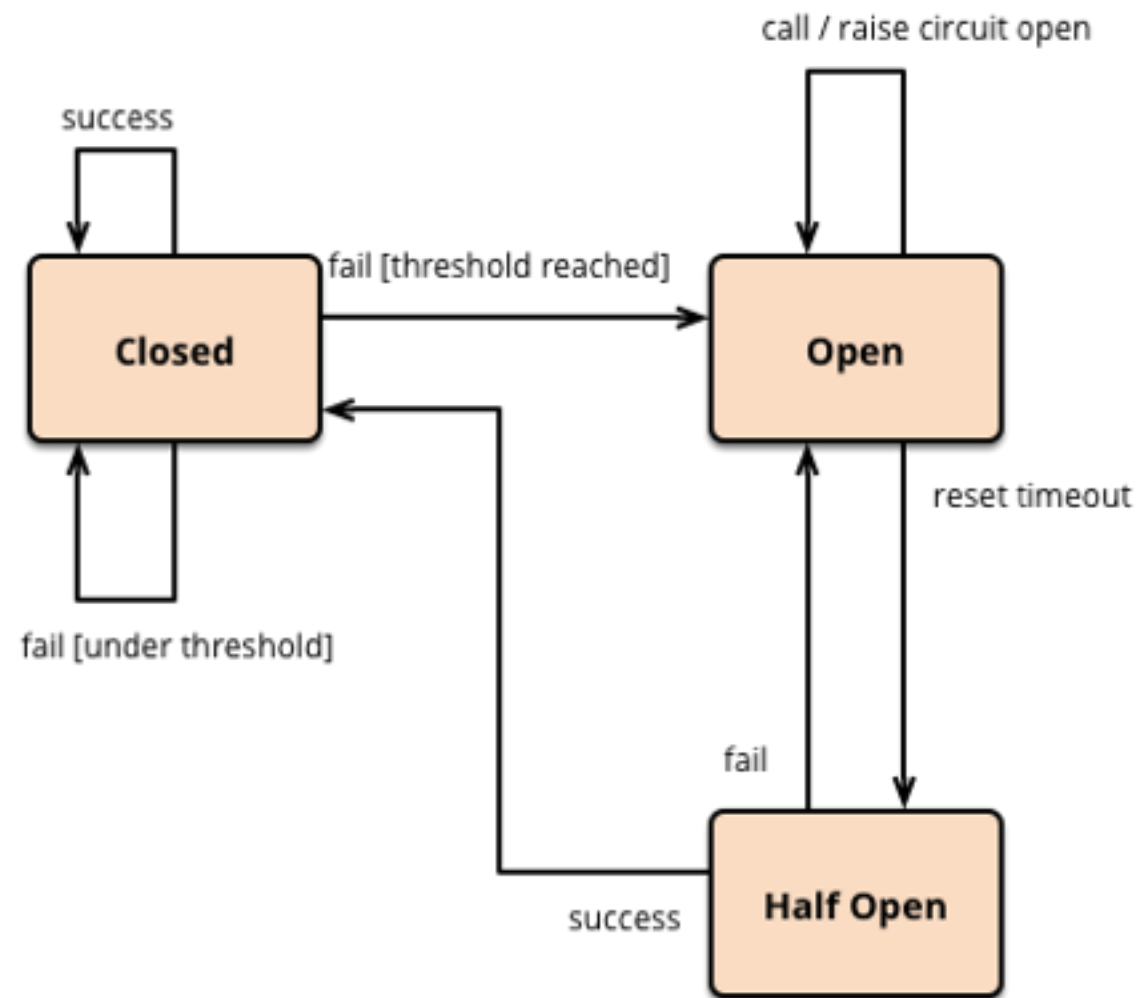




# Circuit Breaker



# Circuit Breaker



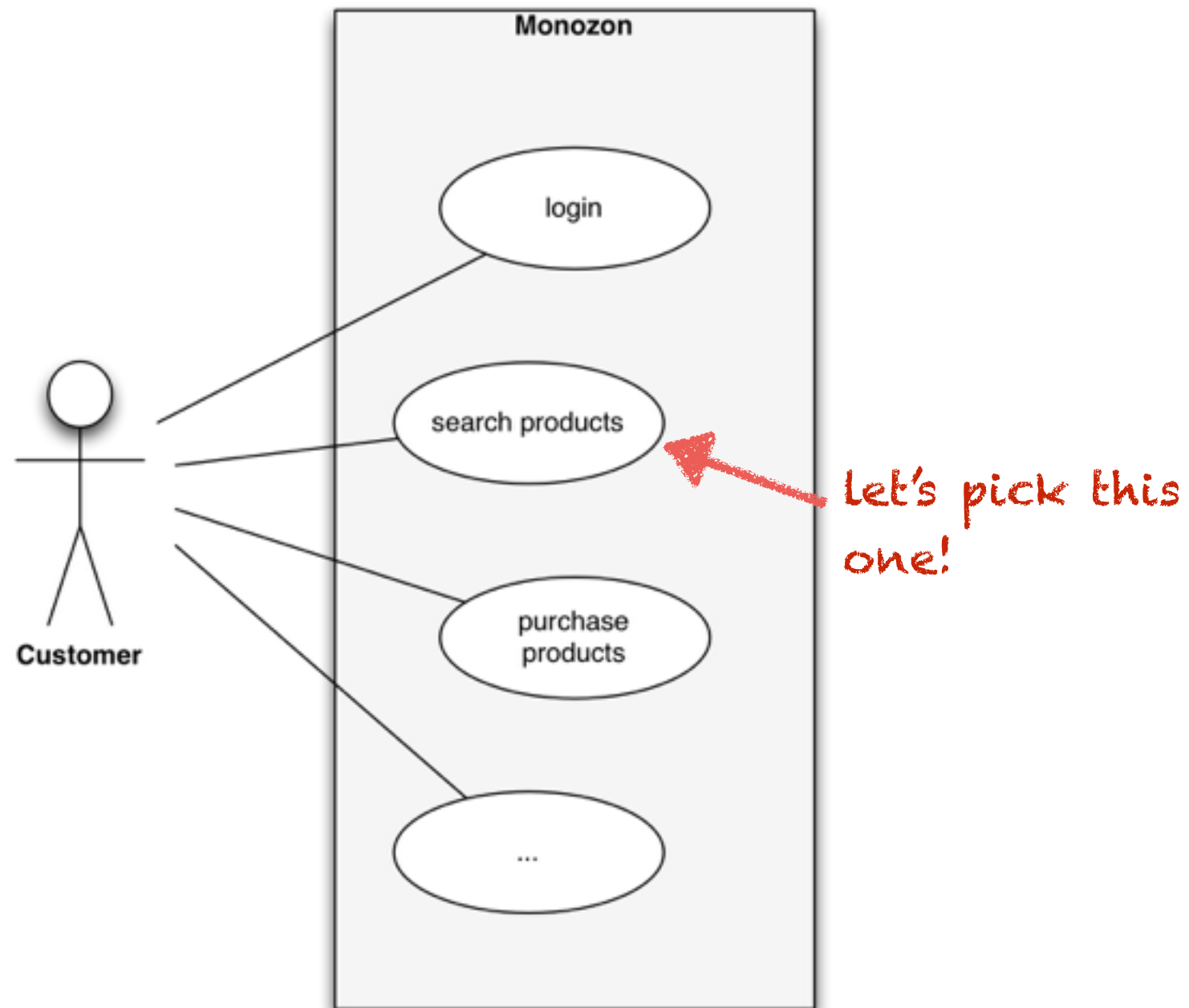
# Hystrix

---

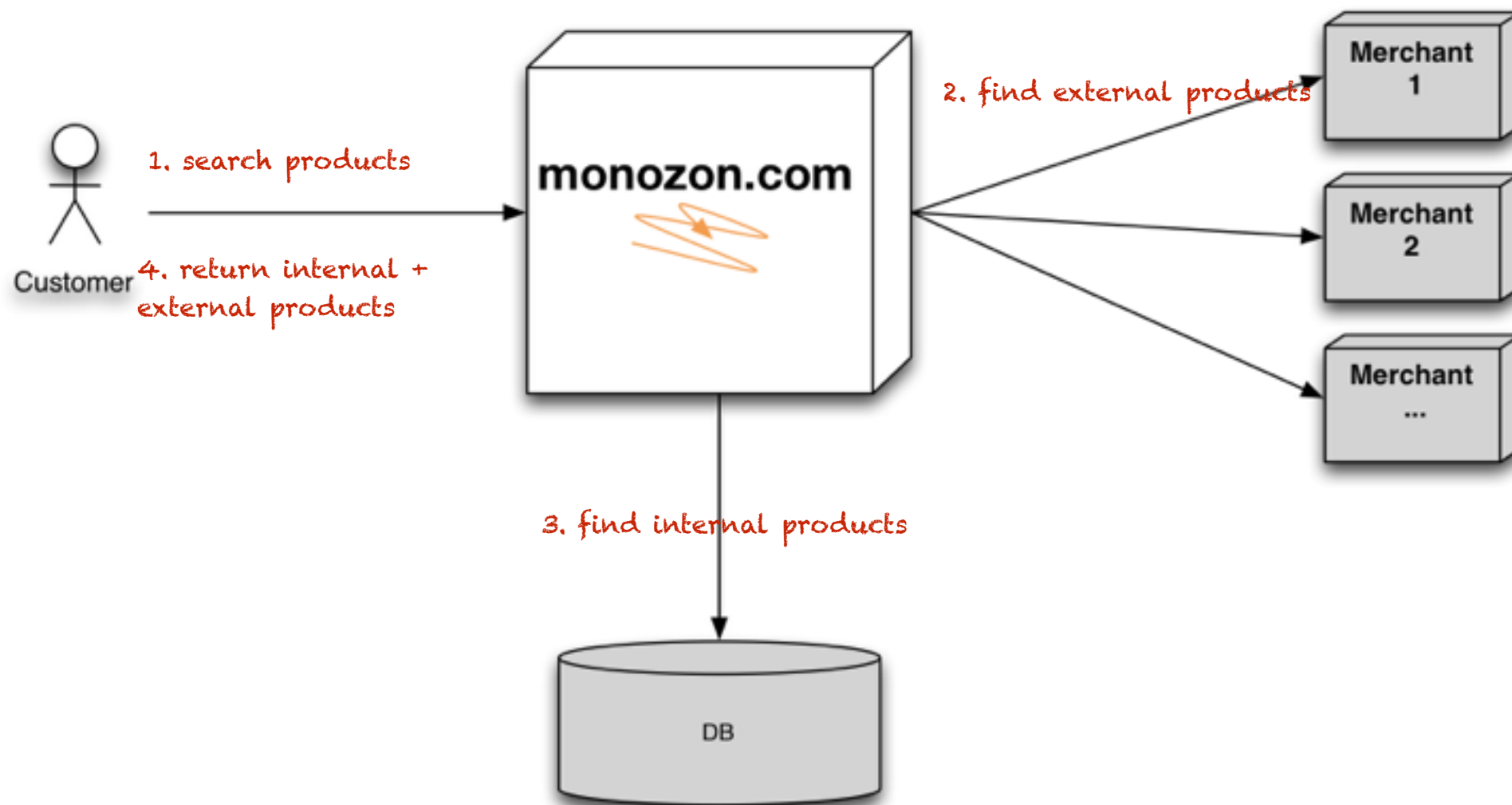
- › Library from Netflix
- › Resilience Library
- › Command Pattern
- › Metrics
- › Dashboard



# Use Cases



# Search Products



# Search Products

```
→ / http GET http://monozon:8080/products | jq '.'  
[  
  "internalProduct_1",  
  "internalProduct_2",  
  "merchant_1",  
  "merchant_2",  
  "merchant_3",  
  "merchant_4"  
]
```

# Call without Hystrix

```
private List<Product> findProducts(String query) {  
    ClientResponse clientResponse =  
        Client.create()  
            .resource("http://merchant1/products/")  
            .queryParams("query", query)  
            .get(ClientResponse.class);  
  
    return toProduct(clientResponse);  
}
```

*cascading failures incoming!*



# Simple Command

```
public class GetMerchant1Products
    extends HystrixCommand<List<Product>> {

    private final String query;

    public GetMerchant1Products(String query) {
        super(HystrixCommandGroupKey.Factory.asKey("merchant1"));
        this.query = query;
    }

    @Override
    protected List<Product> run() throws Exception {
        return findProducts(query);
    }
}
```

# Execute it!

```
public List<Product> findExternalProducts(String query) {  
    List<Product> productList =  
        new GetMerchant1Products(query).execute();  
  
    List<Product> merchant2Products =  
        new GetMerchant2Products(query).execute();  
  
    productList.addAll(merchant2Products);  
  
    return productList;  
}
```

# Execute it asynchronously

```
public List<Product> findExternalProducts(String query) {  
    Future<List<Product>> merchant1ProductsFuture =  
        new GetMerchant1Products(query).queue();  
  
    Future<List<Product>> merchant2ProductsFuture =  
        new GetMerchant2Products(query).queue();  
  
    List<Product> productList = new ArrayList<>();  
    productList.addAll(merchant1ProductsFuture.get());  
    productList.addAll(merchant2ProductsFuture.get());  
  
    return productList;  
}
```

# Fallback

```
@Override  
protected List<Product> run() throws Exception {  
    return findProducts(query);  
}
```

```
@Override  
protected List<Product> getFallback() {  
    return Collections.emptyList();  
}
```

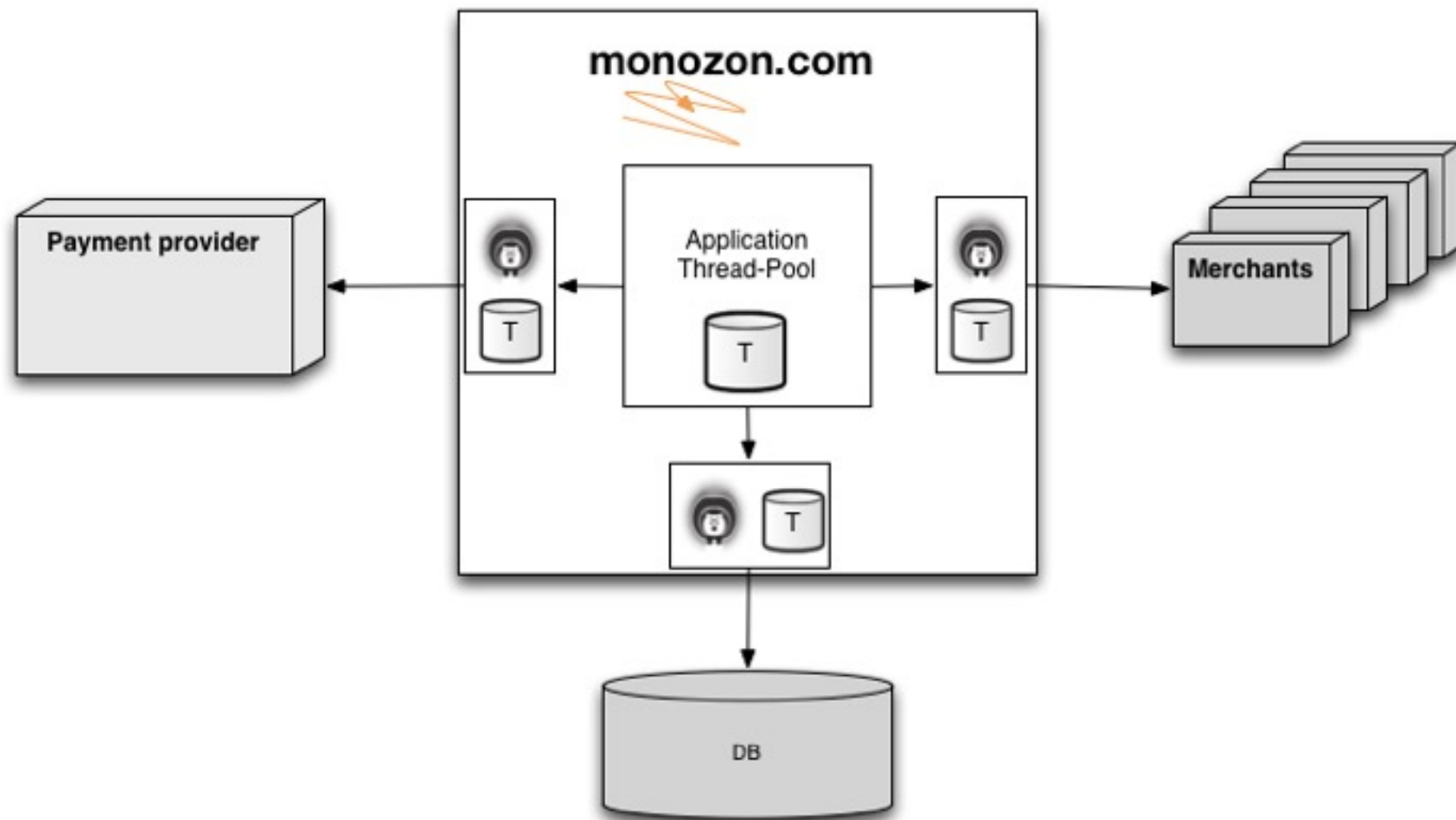
# In case Merchant 2 is down

```
→ / http GET http://monozon:8080/products | jq '.'  
[  
  "internalProduct_1",  
  "internalProduct_2",  
  "merchant_1",  
  "merchant_3",  
  "merchant_4"  
]
```



something is  
missing here

# The stabilized system





# Demo

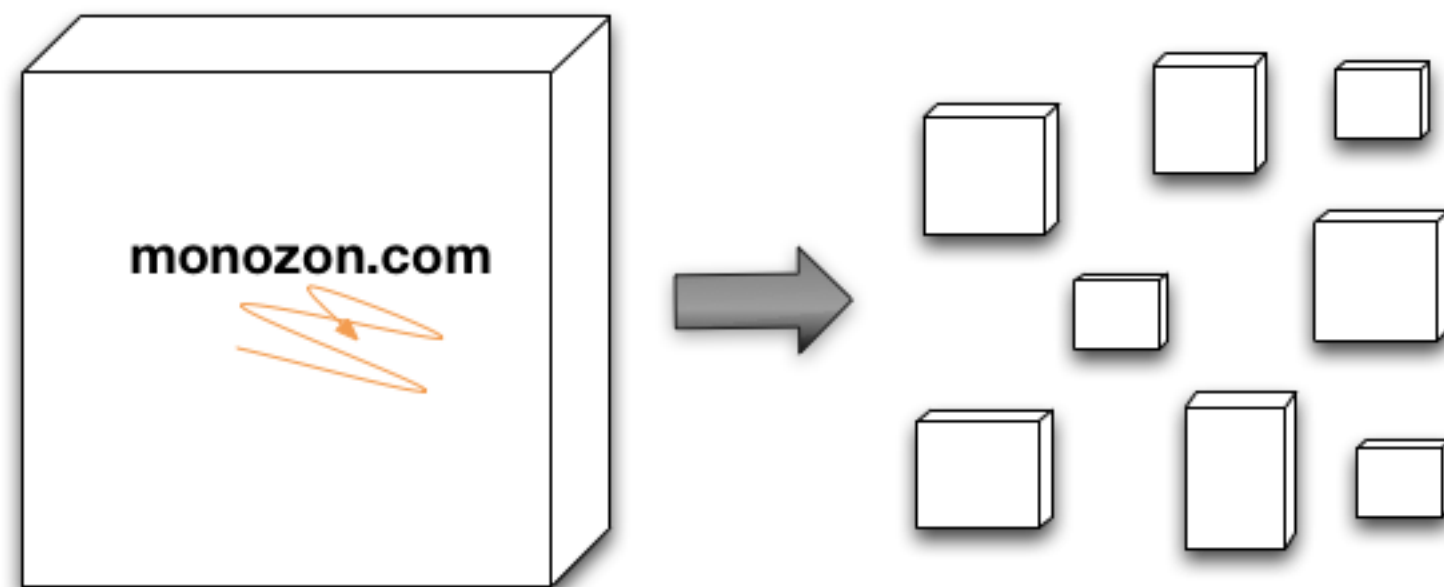
And now?

# Current Problems

---

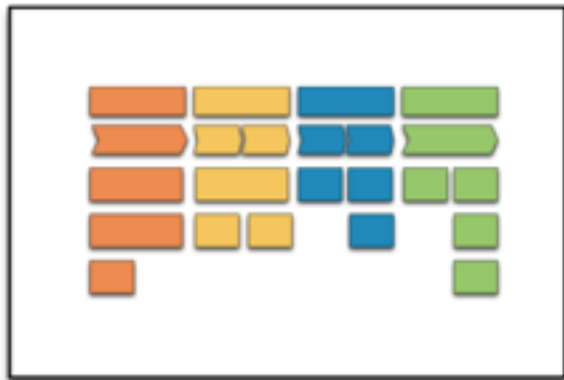
- › Maintenance is difficult
- › New features need a lot of time
- › ~~Very unstable~~ => enables further distribution
- › Outdated technology
- › Doesn't scale

# Microservices!

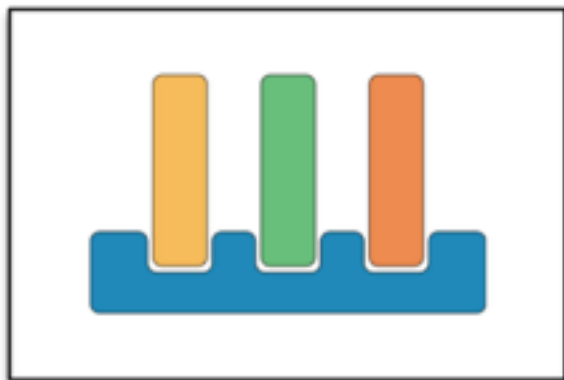


but how to get started ????

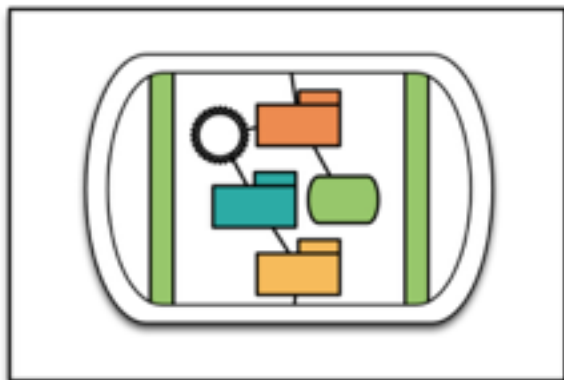
# Architectural Decisions



> Domain Architecture



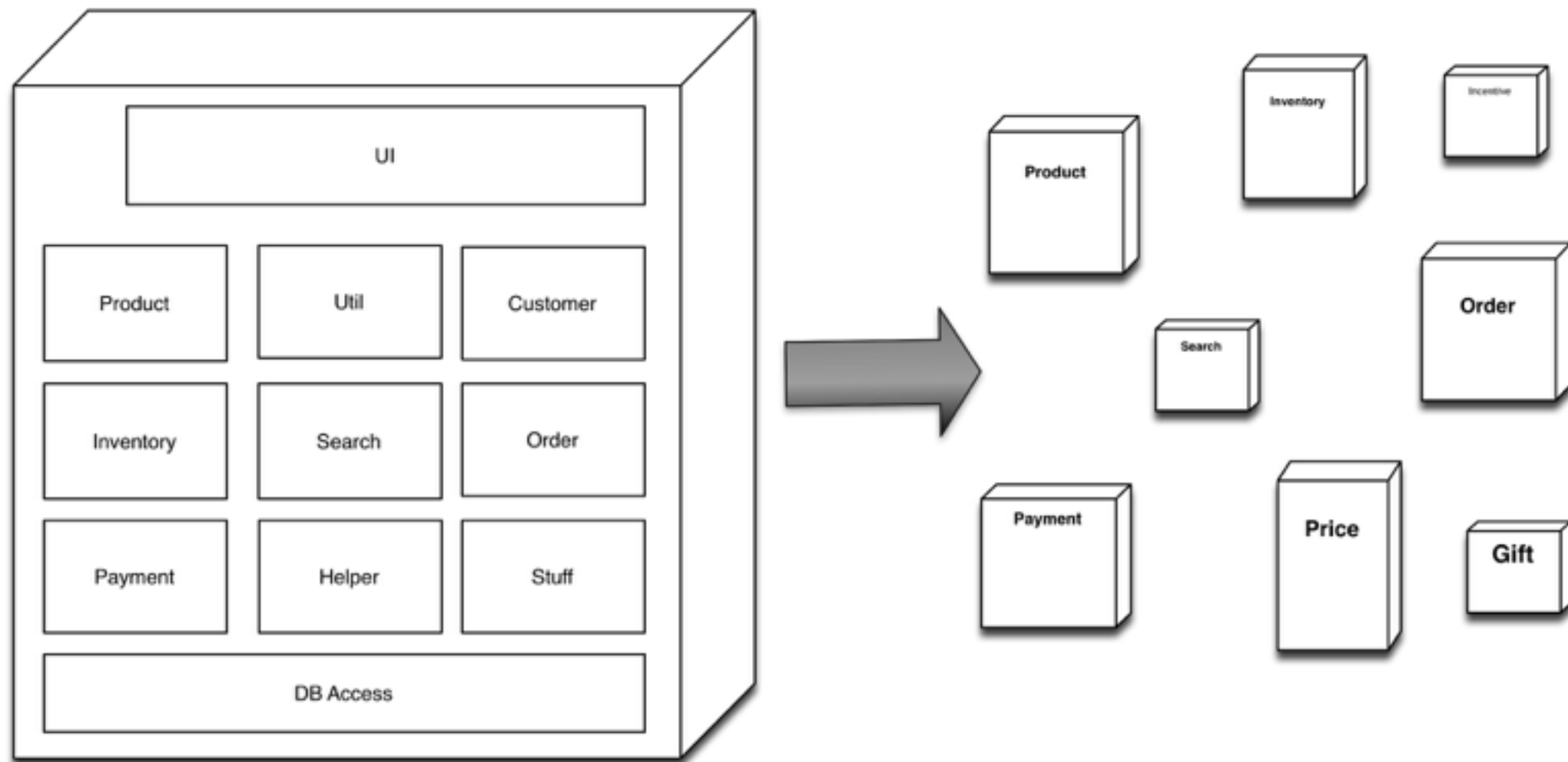
> Macro Architecture



> Micro Architecture

# Domain Architecture

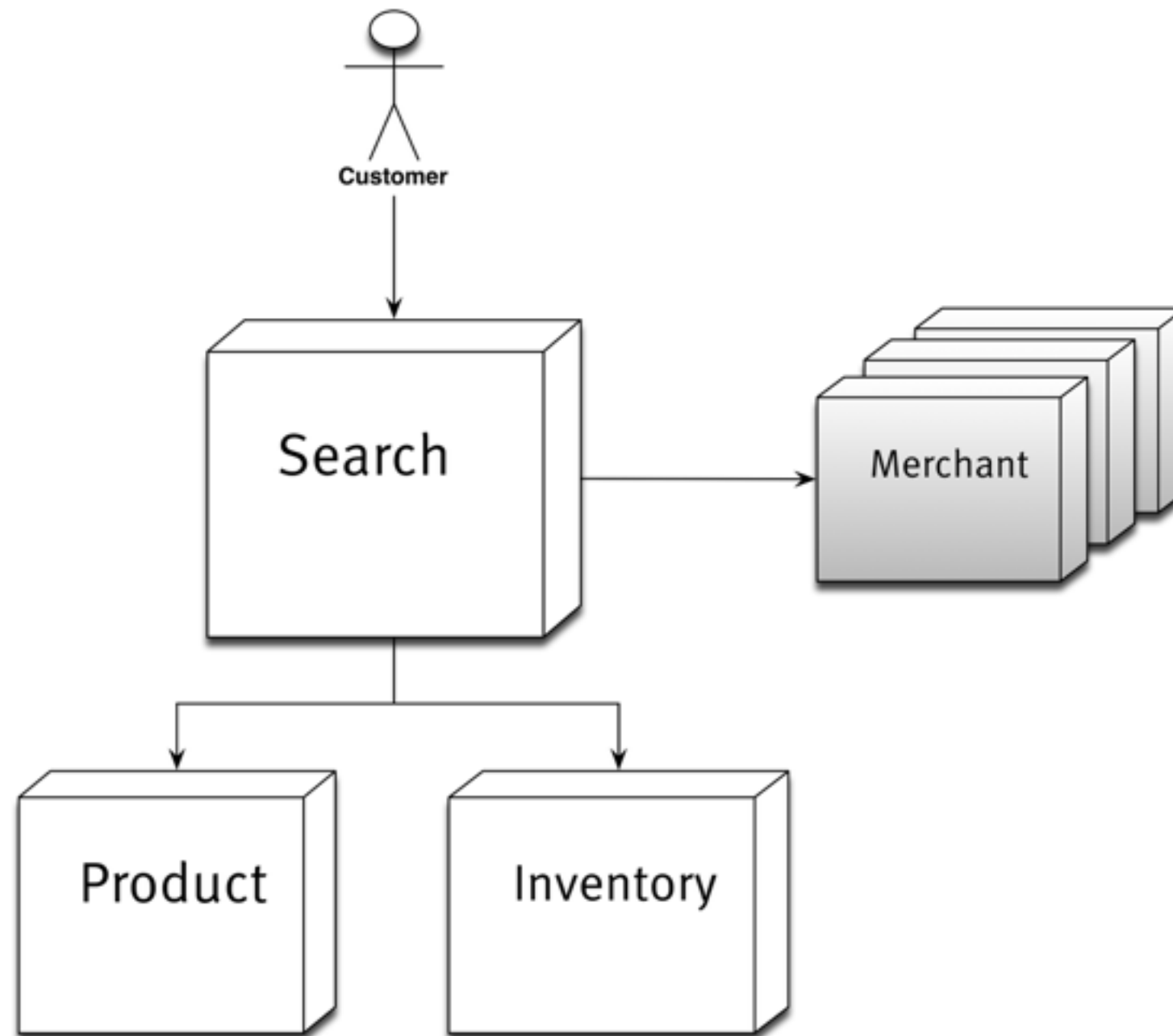
*which boxes do we need ?*



*let the monolith guide you!*



# Domain: Search



# Macro Architecture

*what's the same for all boxes ?*

---

- › Integration
- › Deployment
- › Formats
- › Protocols
- › Reduce Choices

Monozon:

= API + UI

= Docker

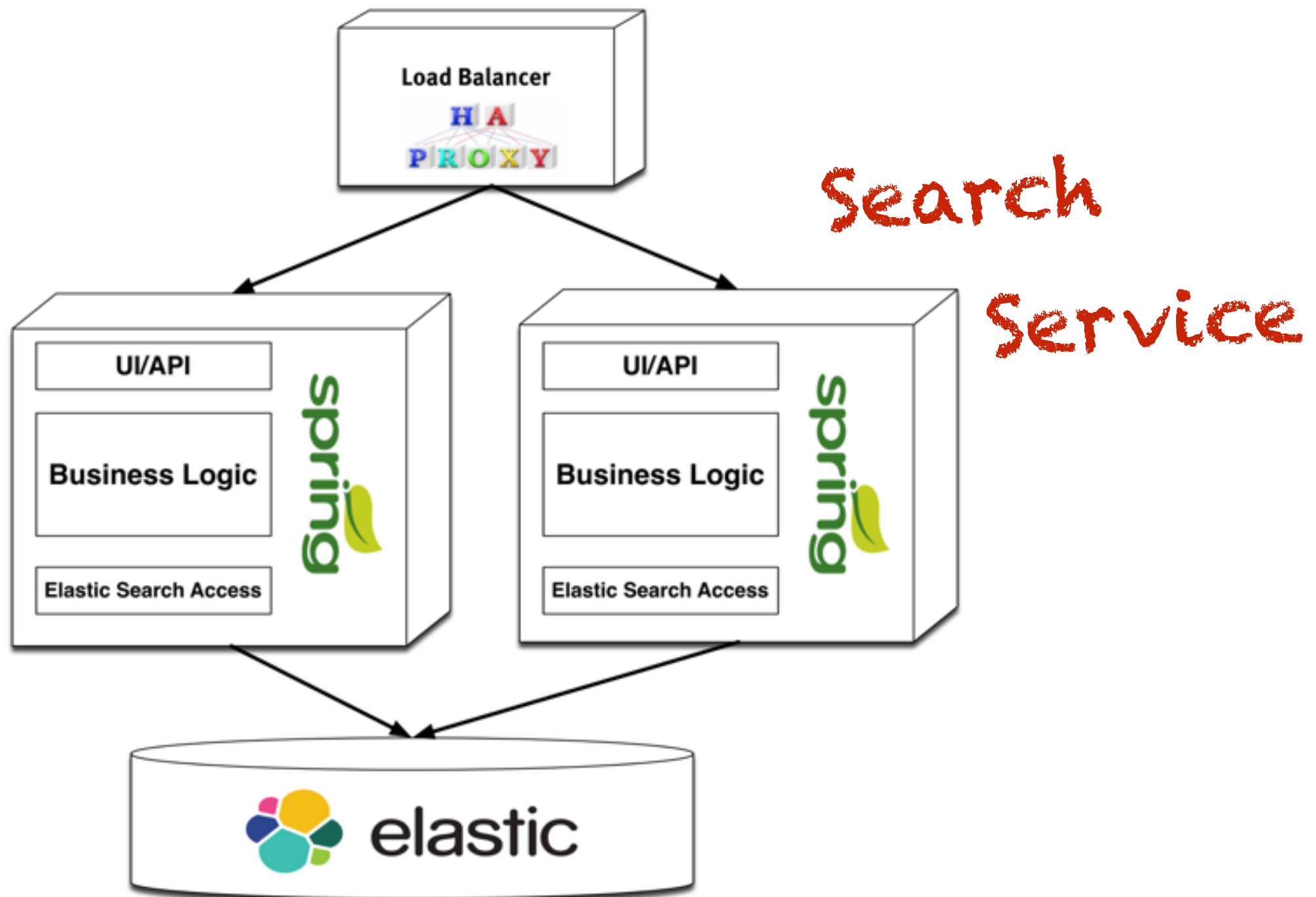
= JSON

= HTTP + AMQP

= Java, Go

*pick your  
own!*

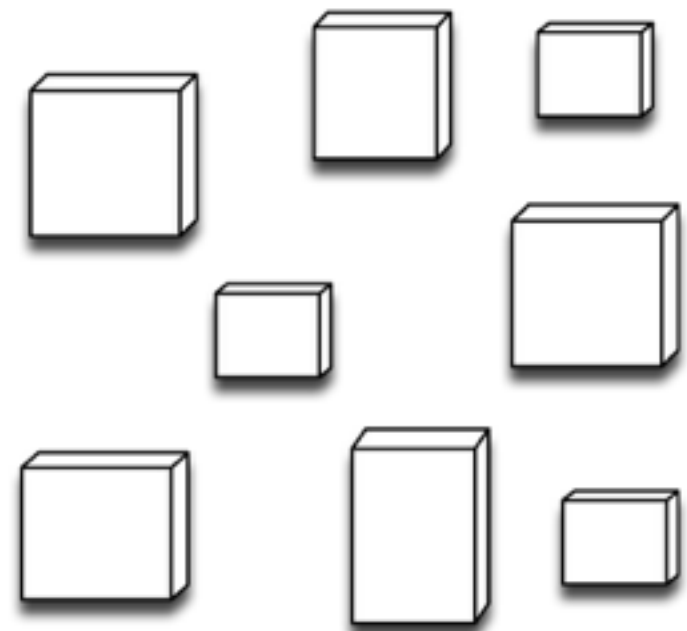
# Micro Architecture



# Migration Path



?

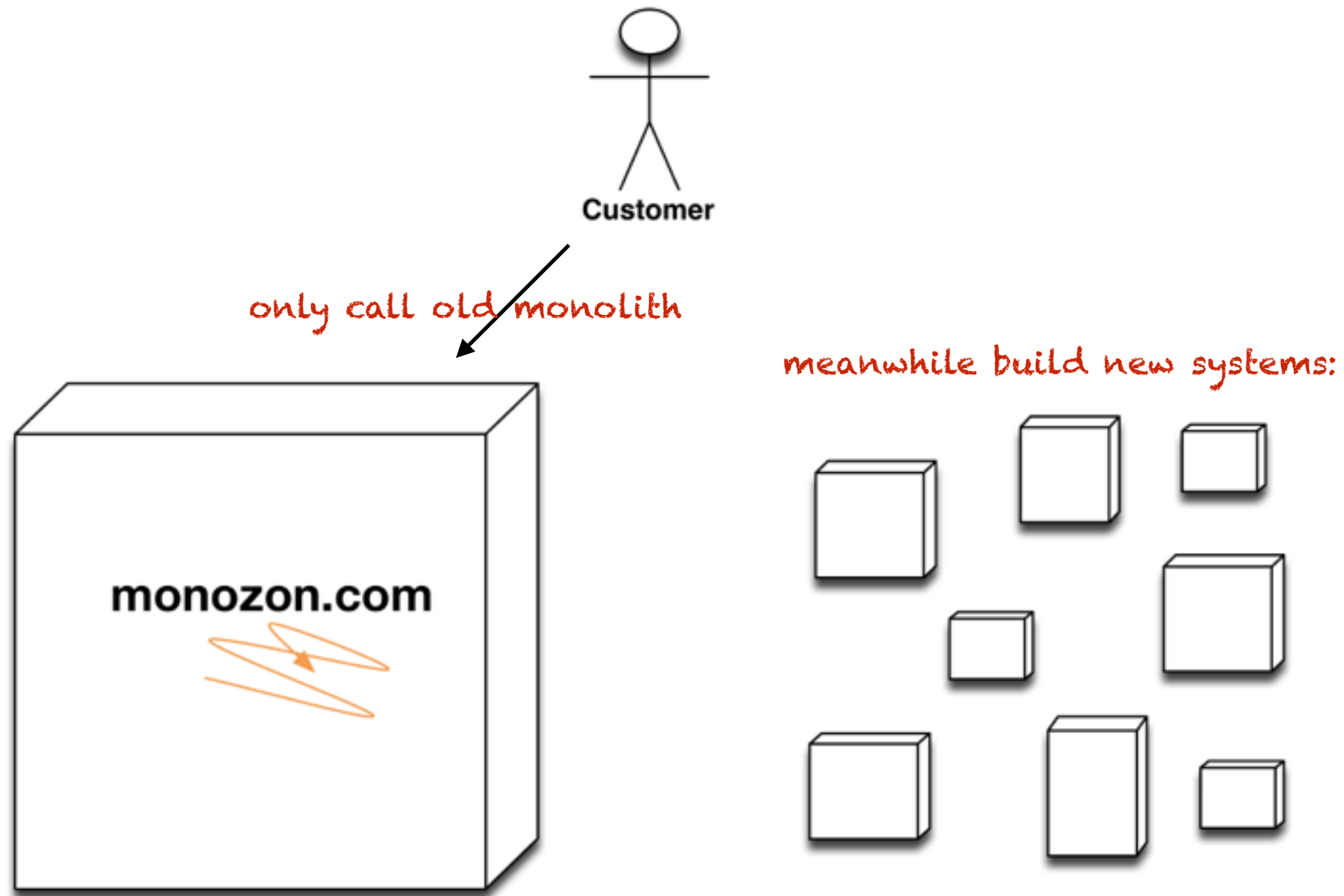


pick one

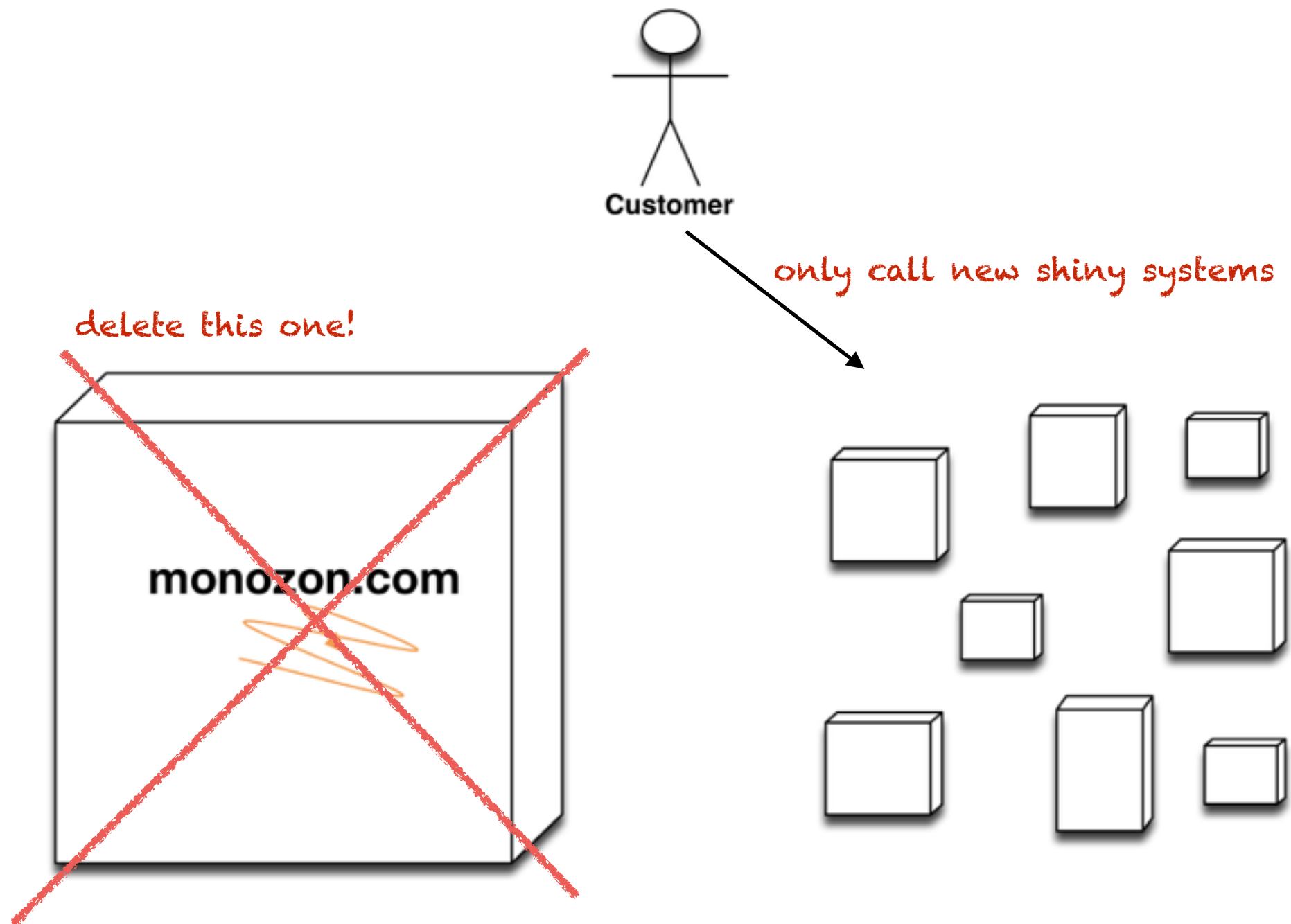
[big bang,  
strangler,  
wonder]

# Big Bang

a.k.a REVOLUTION

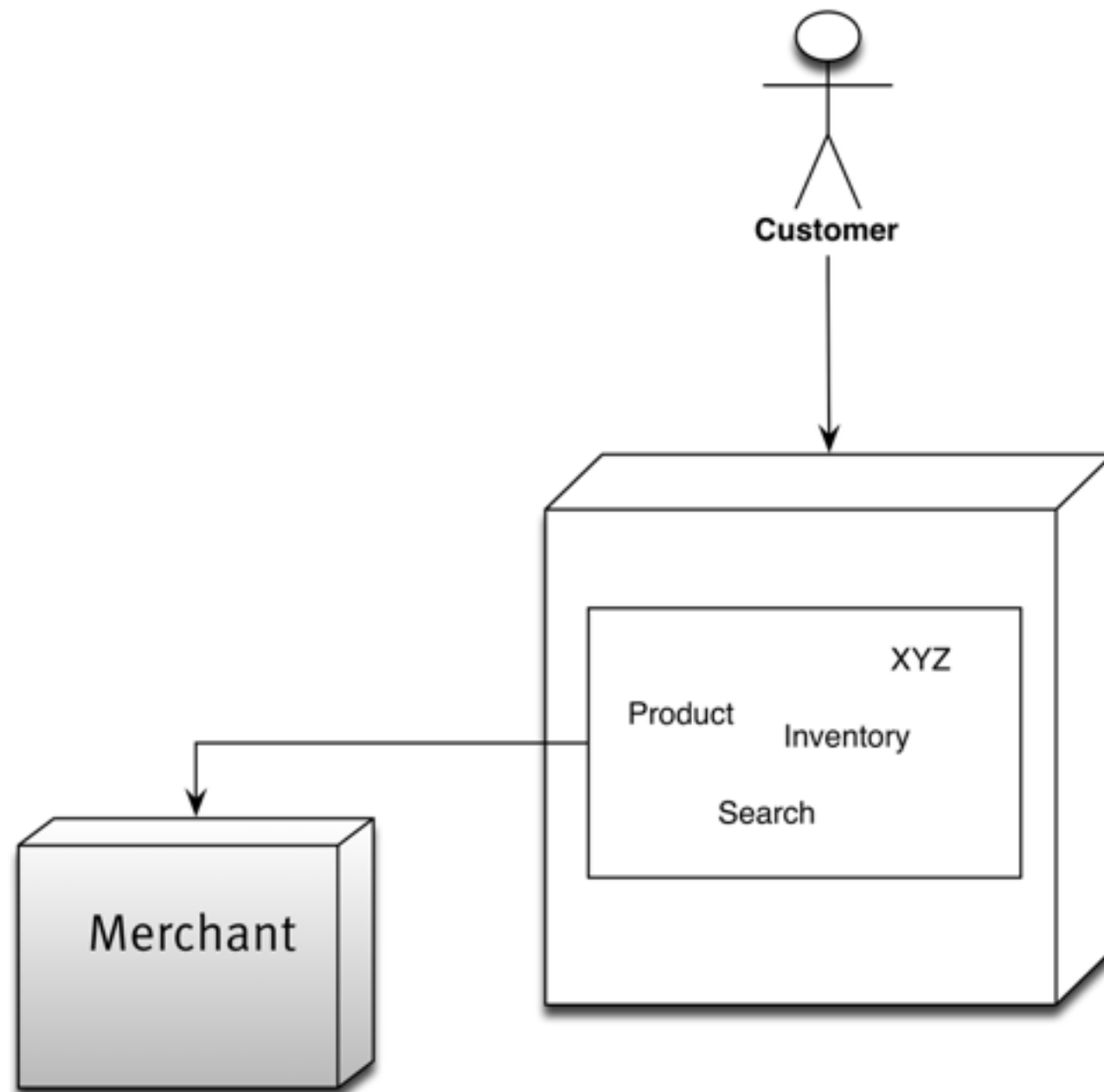


# Big Bang



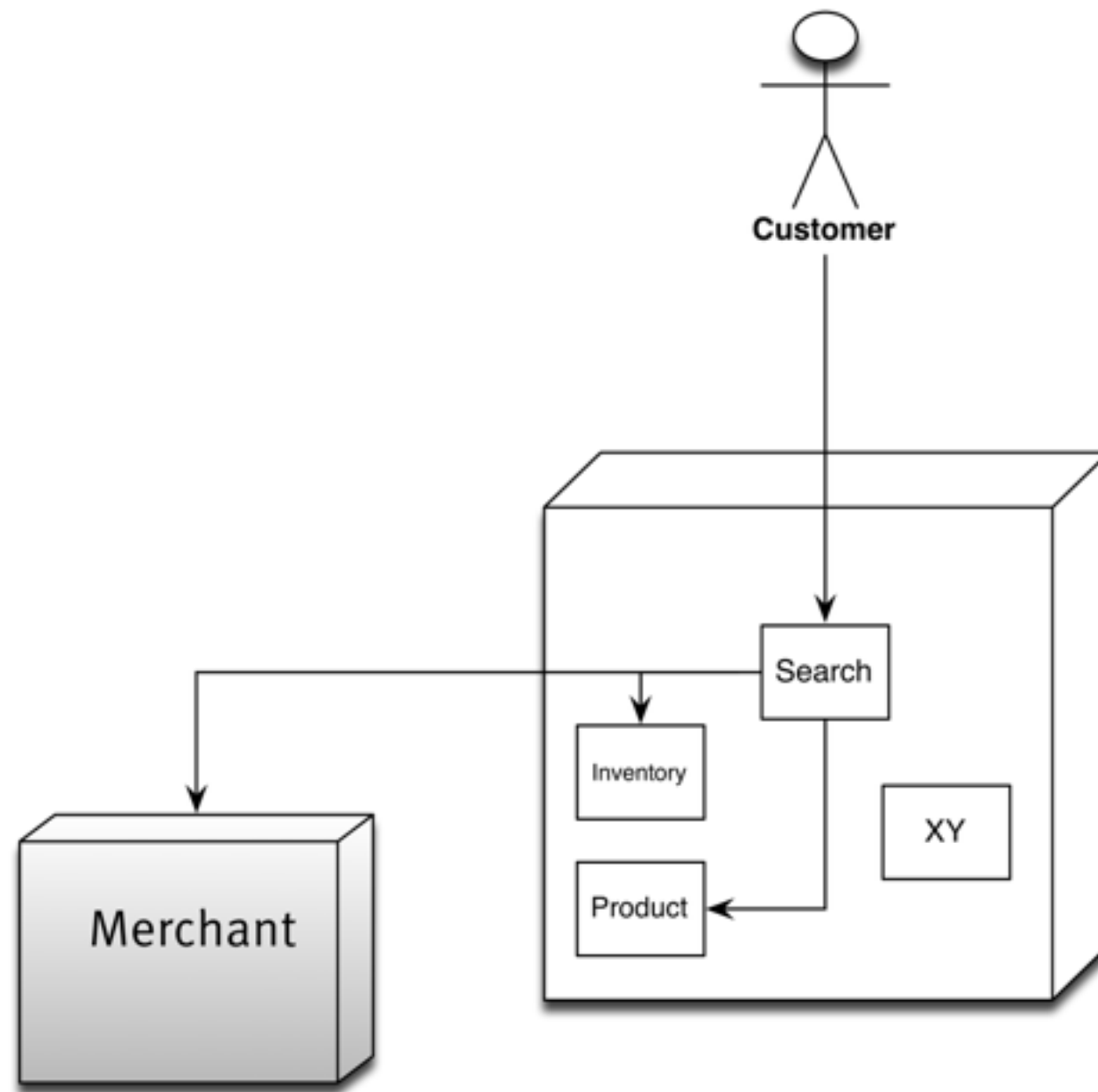
# Strangler

a.k.a EVOLUTION



your monolith a.k.a  
„Big Ball of Mud“

# Strangler



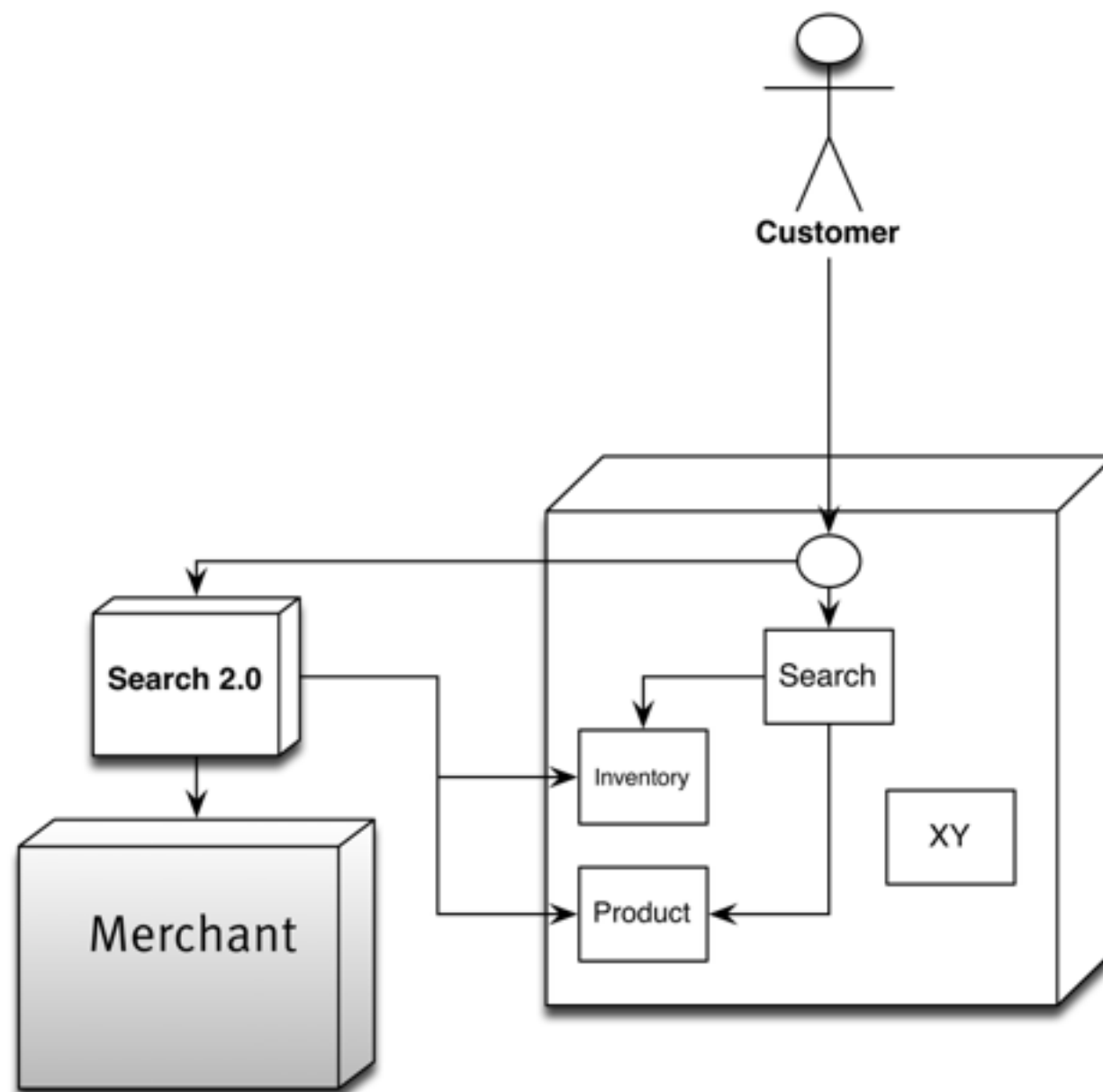
give your monolith  
internal structure  
and define interfaces

...

if possible.

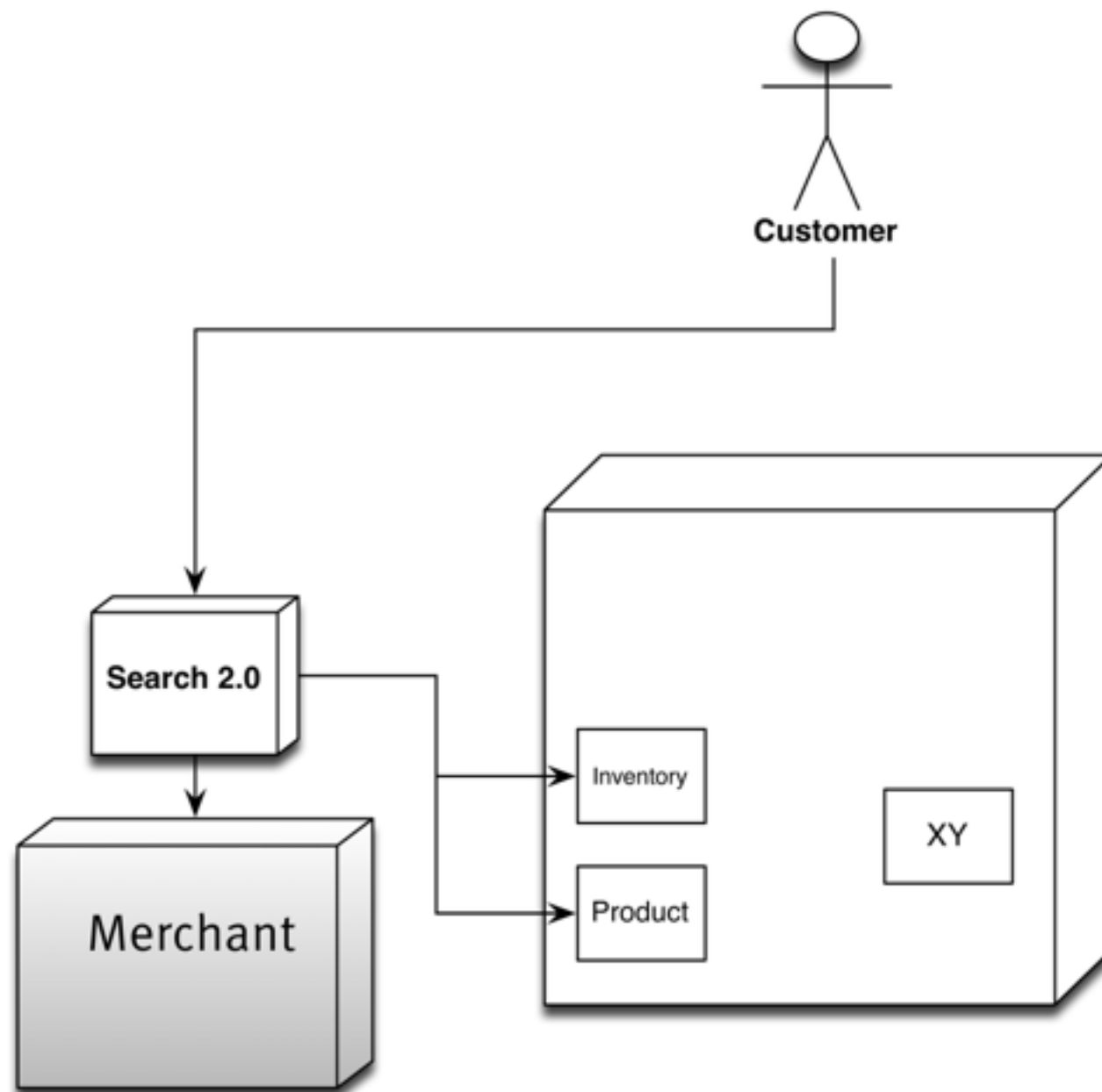


# Strangler



Extract your new service and redirect calls to it

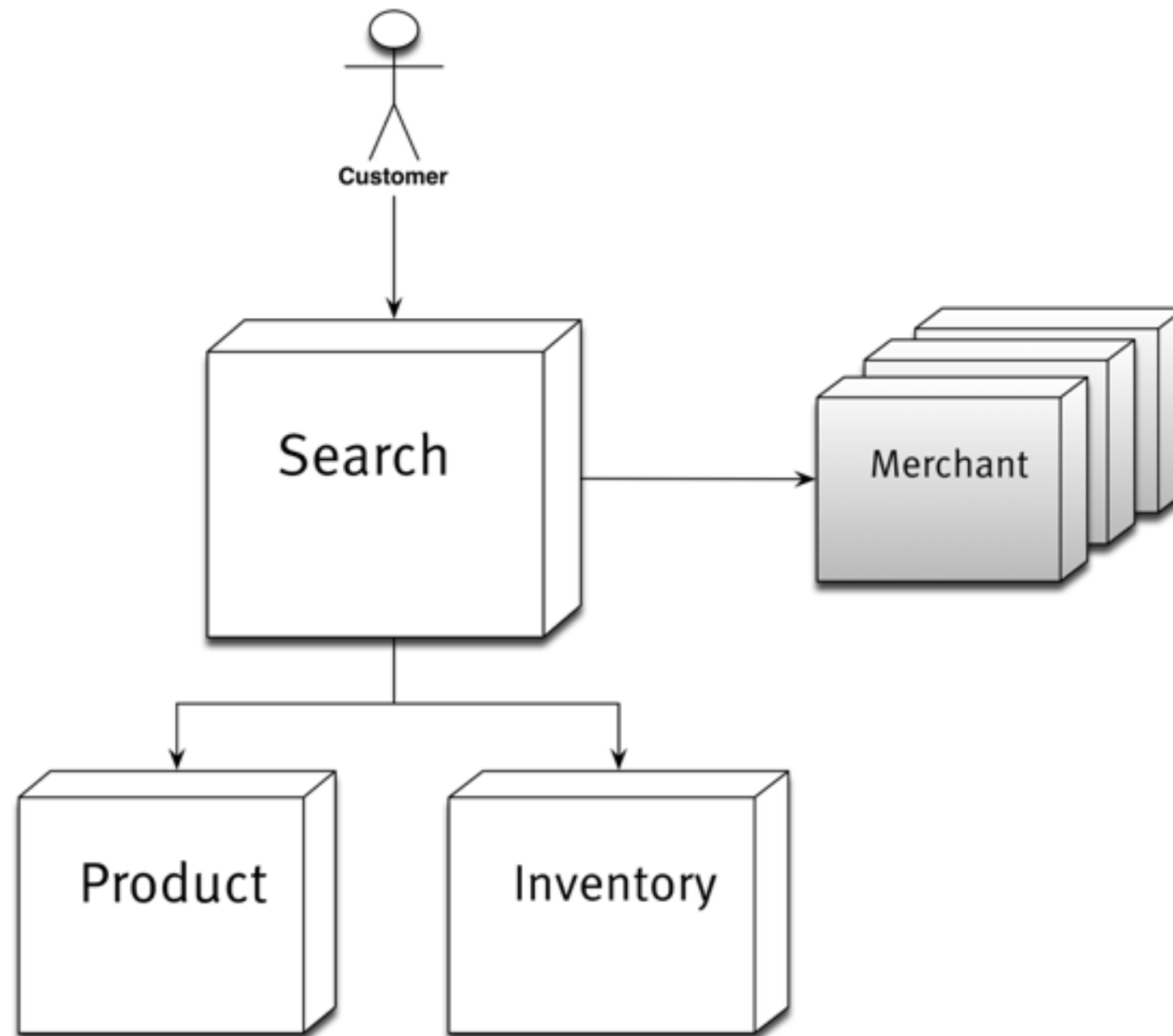
# Strangler



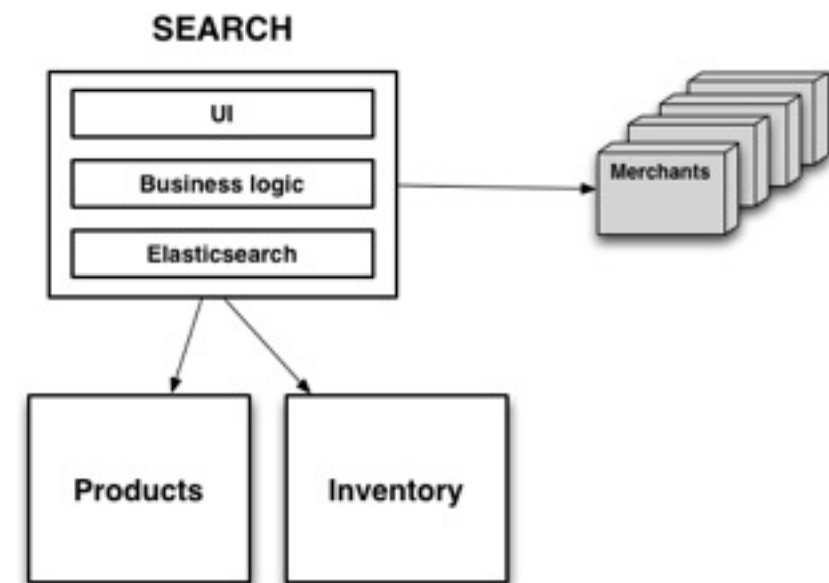
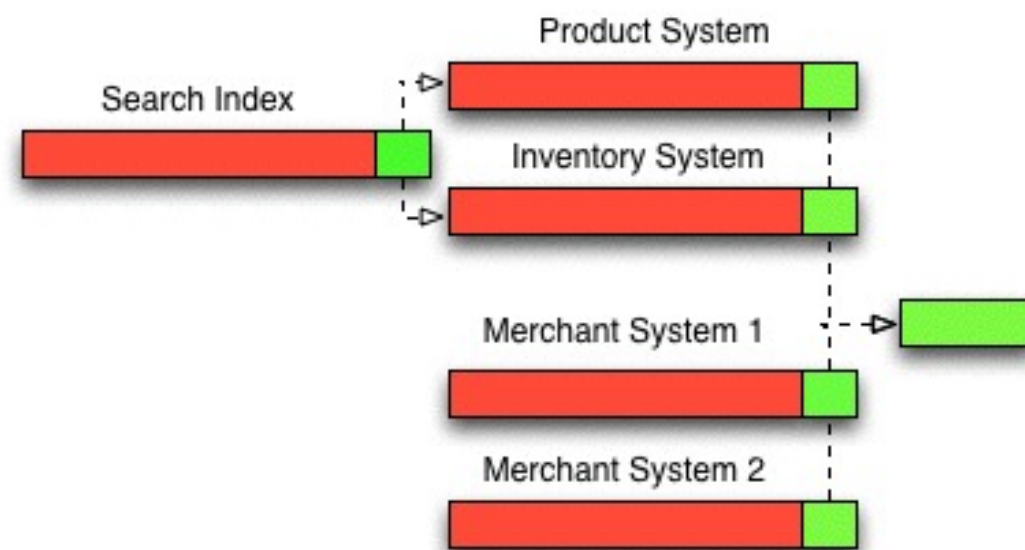
Once unused strangle  
the old code

Is it really so easy?

# Time to connect boxes



# Synchronous vs. Async/Parallel



# Try this with futures

```
Future<List<Long>> productIndexFuture =  
    getProductIndexFuture(query);  
  
List<Long> productIdList = productIndexFuture.get();  
  
List<FuturePair> futurePairList = new ArrayList<>();  
for (Long productId : productIdList) {  
    Future<Product> productFuture = retrieveProductFromProductSystem(productId);  
    Future<Long> quantityFuture = retrieveQuantityFromInventoryService(productId);  
    futurePairList.add(new FuturePair(productFuture, quantityFuture));  
}  
  
List<SearchResult> searchResultList = new ArrayList<>();  
for (FuturePair futurePair : futurePairList) {  
    Product product = futurePair.getProductFuture().get();  
    Long quantity = futurePair.getQuantityFuture().get();  
    SearchResult searchResult = new SearchResult(product, quantity);  
    searchResultList.add(searchResult);  
}
```

blocking!

blocking!

# Time for RxJava

---

- › Reactive Extensions for the JVM
- › Asynchronous streams
- › Elements of
  - › Iterator pattern
  - › Observable pattern
  - › Functional programming



**Iterable**

*pull*

T next()

throws Exception  
returns;

**Observable**

*push*

onNext(T)

onError(Exception)

onCompleted()

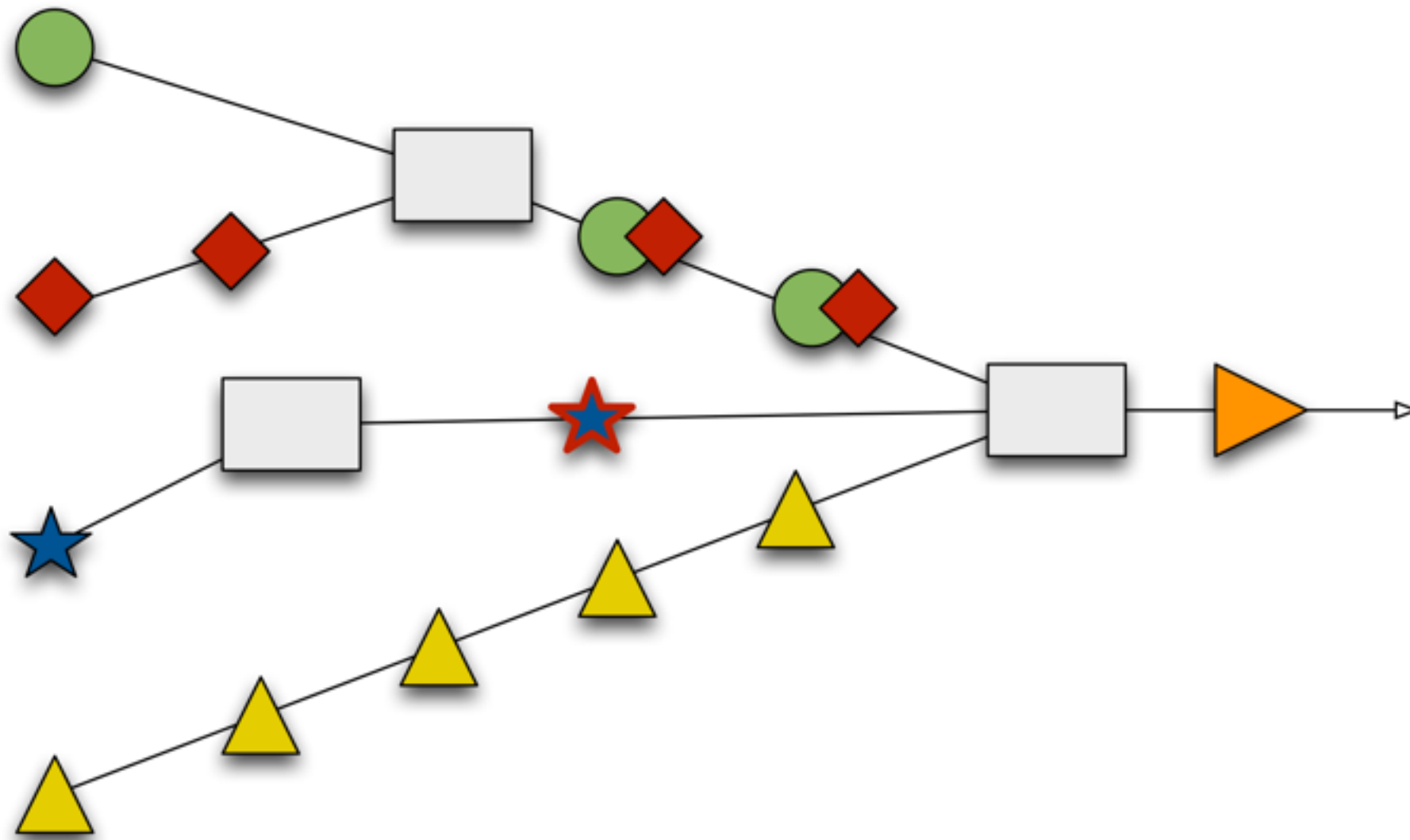


Everything is a stream!





# RxJava in one picture



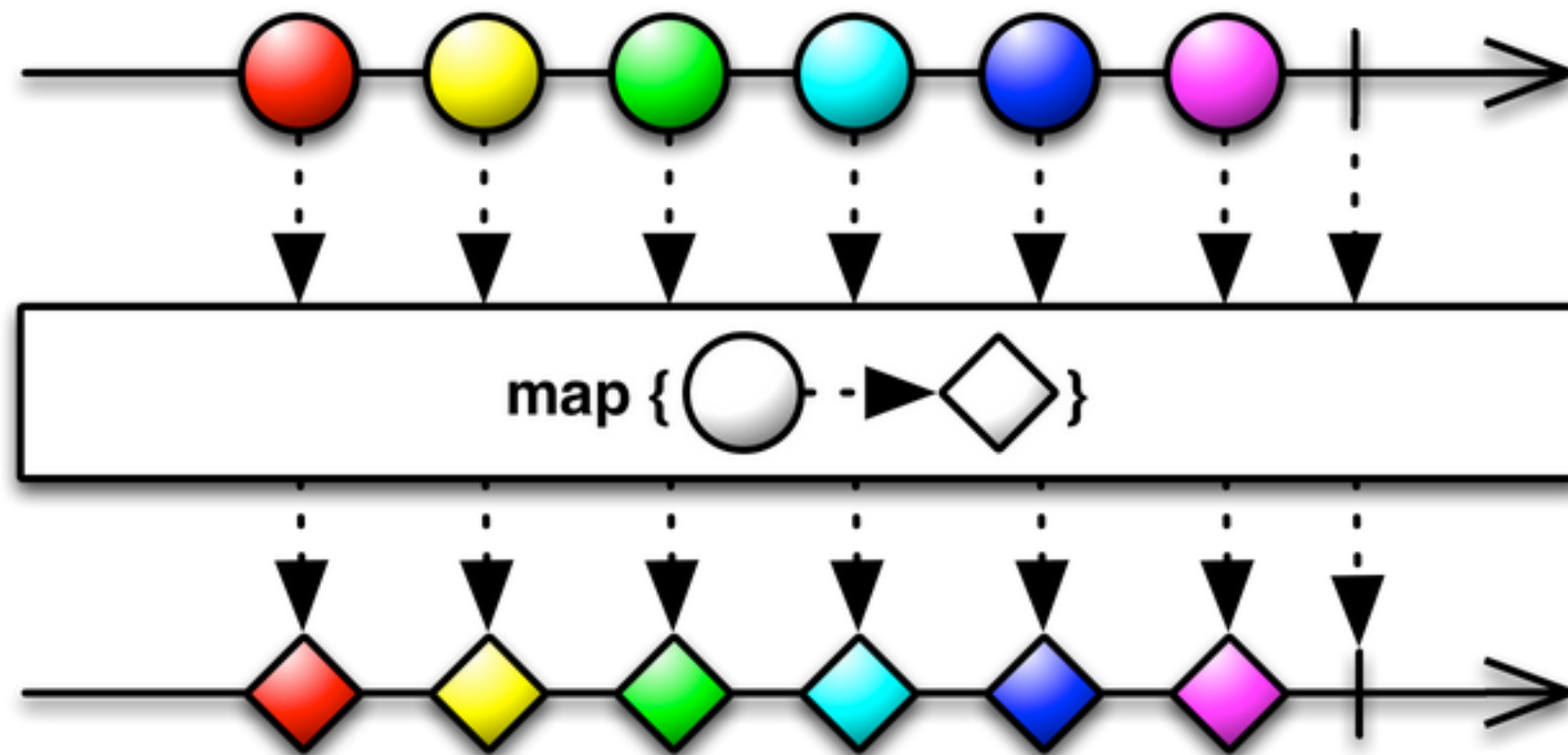
# Creating Observables

```
Observable.just("Book A", "Book B", "Book C");
```

```
Observable.from(findProducts(query));
```

```
Observable.create(o -> {  
    for (Merchant2Product product : findProducts(query)) {  
        o.onNext(product);  
    }  
    o.onCompleted();  
});
```

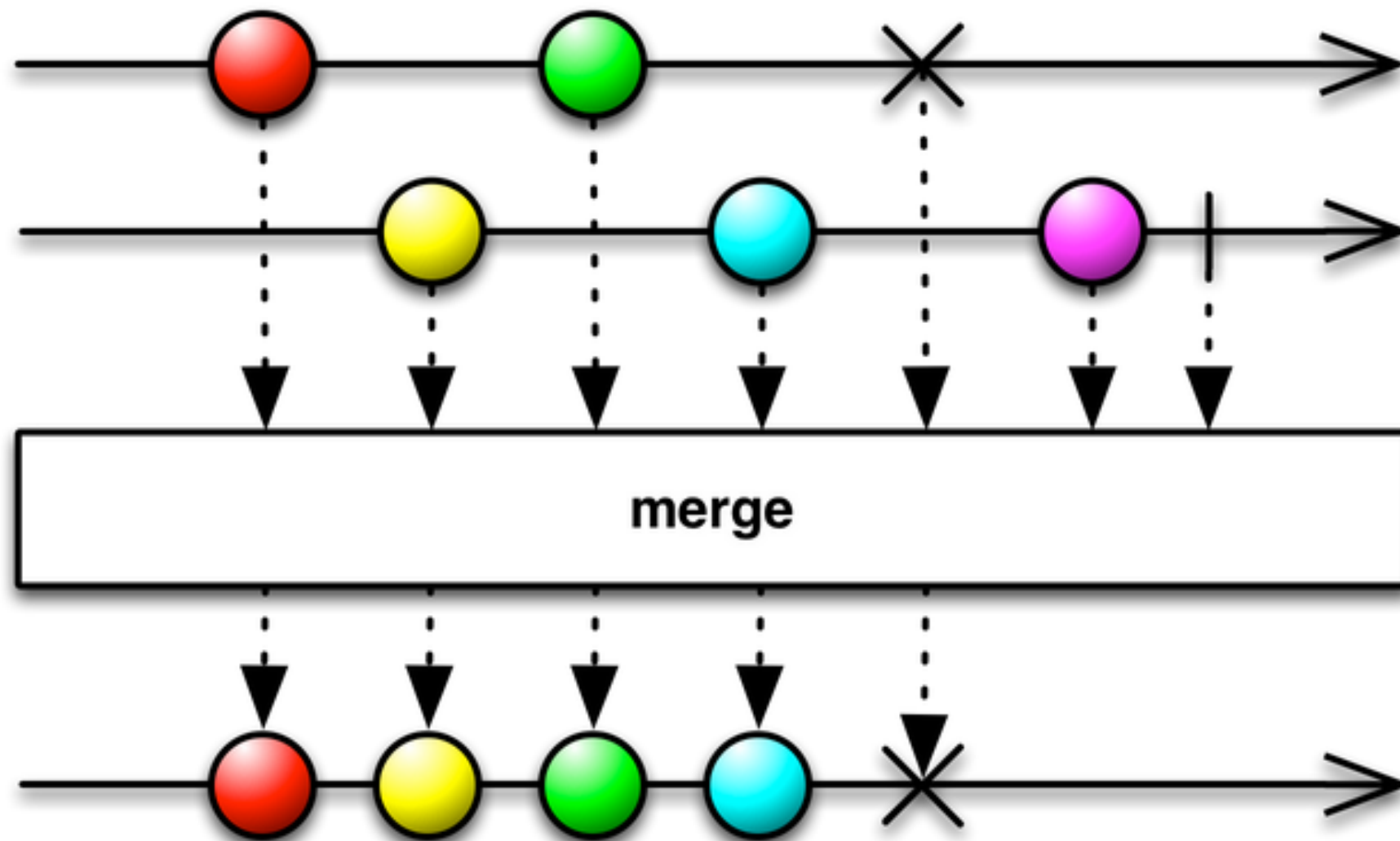
# Transforming with map



# map in action

```
Observable<Merchant2Product> productM20bservable =  
    Observable.from(findProducts(query));  
  
Observable<SearchResult> searchResultM20bservable =  
    productM20bservable.map(this::toSearchResult);
```

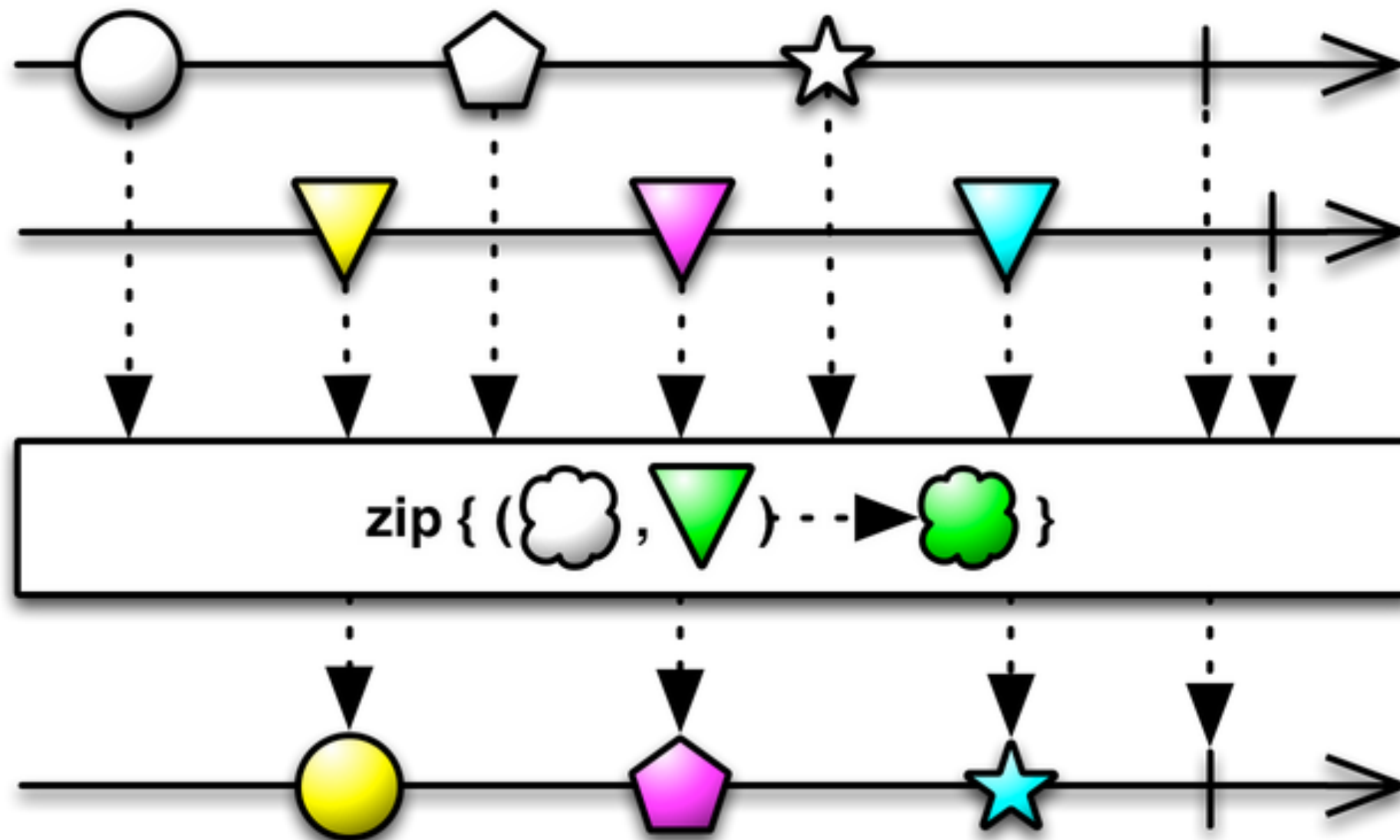
# Combining with merge



# merge in action

```
Observable<SearchResult> mergedSearchResultObservable =  
    |   searchResultM1Observable.mergeWith(searchResultM2Observable);
```

# Combining streams with zip

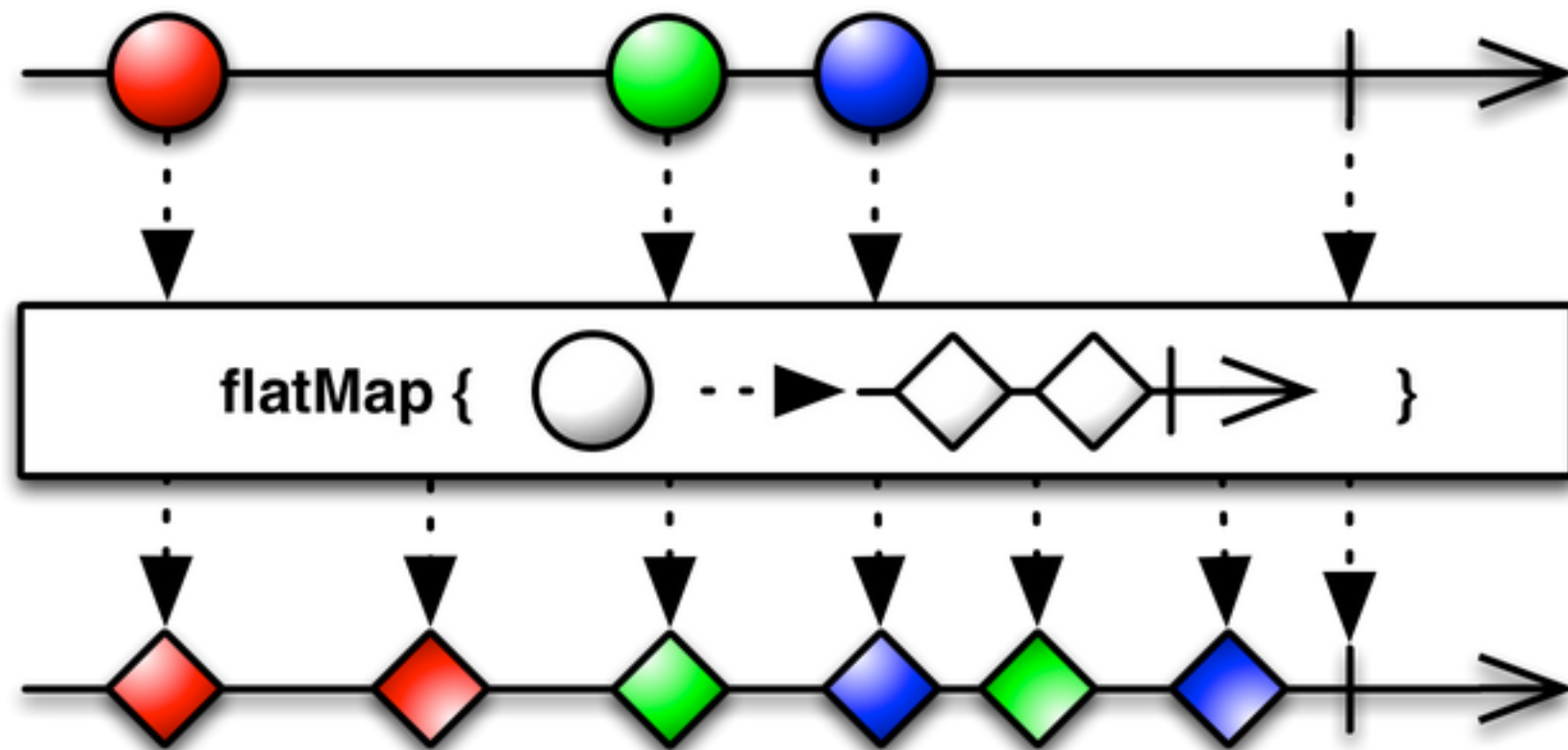




# zip in action

```
private Observable<SearchResult> productDetails(Long productId) {  
    Observable<Product> productObservable =  
        retrieveProductFromProductSystem(productId);  
  
    Observable<Long> quantityObservable =  
        retrieveQuantityFromInventoryService(productId);  
  
    return Observable.zip(productObservable,  
        quantityObservable, SearchResult::new);  
}
```

# Collecting details with flatMap



# flatMap in action

```
public Observable<SearchResult> findInternalProducts(String query) {  
    Observable<Long> productIndexObservable =  
        getProductIndexObservable(query);  
  
    return productIndexObservable  
        .flatMap(this::productDetails);  
}
```

# Why not map?

```
public Observable<Observable<SearchResult>> findInternalProducts(String query) {  
    Observable<Long> productIndexObservable =  
        getProductIndexObservable(query);  
  
    return productIndexObservable  
        .map(this::productDetails);  
}
```

# Concurrency

```
Observable<Merchant2Product> productM2Observable =  
    Observable.from(findProducts(query))  
        .subscribeOn(Schedulers.io());
```

# Easier with Hystrix

```
public class GetMerchant2Products
    extends HystrixCommand<List<Merchant2Product>> {

    private final String query;

    public GetMerchant2Products(String query) {
        super(HystrixCommandGroupKey.Factory.asKey("merchant2"));
        this.query = query;
    }

    @Override
    public List<Merchant2Product> run() {
        return findProducts(query);
    }
}
```

# Converting into a stream

```
Observable<List<Merchant2Product>> productM2ListObservable =  
    new GetMerchant2Products(query).observe();  
  
Observable<Merchant2Product> productM2Observable =  
    productM2ListObservable.flatMap(observable::from);
```



# Returning a result

```
public List<SearchResult> getSearchResults(String query) {  
    Observable<SearchResult> searchResults =  
        searchService.findInternalProducts(query)  
            .mergeWith(searchService.findExternalProducts2(query));  
  
    Iterator<SearchResult> searchResultIterator  
        = searchResults.toBlocking().getIterator();  
  
    return Lists.newArrayList(searchResultIterator);  
}
```

# Summary

---

- › Use Hystrix to stabilize your system!
- › Use RxJava to increase the amount of async/parallel processes in an easy way!
- › Introduce Microservices to get control over your system again!
- › Have fun 😊

Our goal was to make  
you curious!

Find your own way!

# Thank you!

[innoc.com](http://innoc.com)

inno**Q**