Kubernetes Workshop

Kubernetes Training

11: Application Container Design

INNOQ

Motivation (for modularization)

Architecture Review Results

- Building features takes too long
- Architectural quality has degraded
- Technical debt is well-known and not addressed
- Deployment is way too complicated and slow
- Scalability has reached its limit
- Replacement would be way too expensive
- Too many dependencies

Conway's Law

"Organizations which design systems are constrained to produce systems which are copies of the communication structures of these organizations."

App characteristics

- Separate, runnable process
- Accessible via standard ports & protocols
- Shared-nothing model
- Horizontal scaling
- Fast startup & recovery

Microservice Characteristics

- Small
- each running in its own process
- lightweight communicating mechanisms (often HTTP)
- built around business capabilities
- independently deployable
- minimum of centralized management
- may be written in different programming languages
- may use different data storage technologies

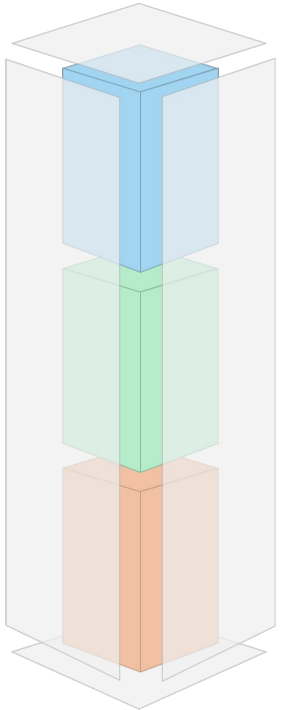
http://martinfowler.com/articles/microservices.html

SCS

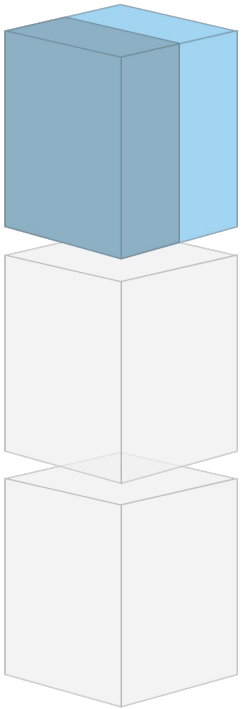
Self-contained Systems (SCS)

- SCS: Autonomous web application
- Optional service API (e.g. for mobile clients)
- Includes data & logic & persistence
- Might contain several microservices
- No shared UI between SCS
- No shared business code
- E.g. Otto, Kaufhof ...

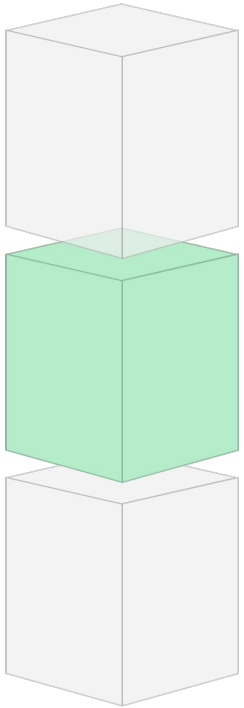
SCS



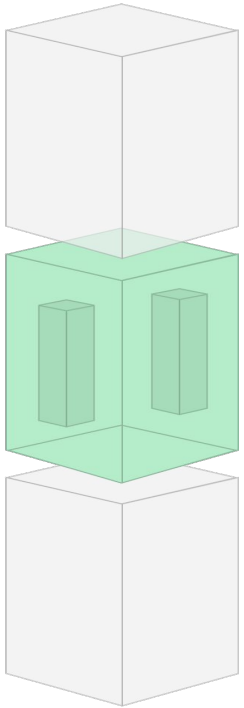
An SCS contains its own user interface, specific business logic and separate data storage



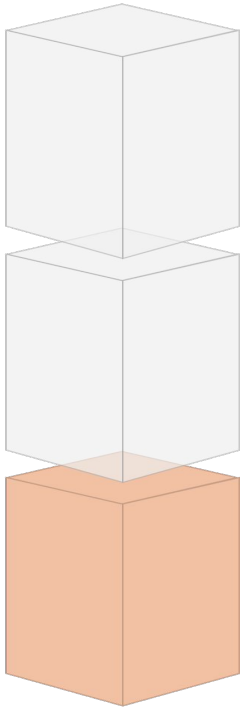
optional API e.g. for mobile



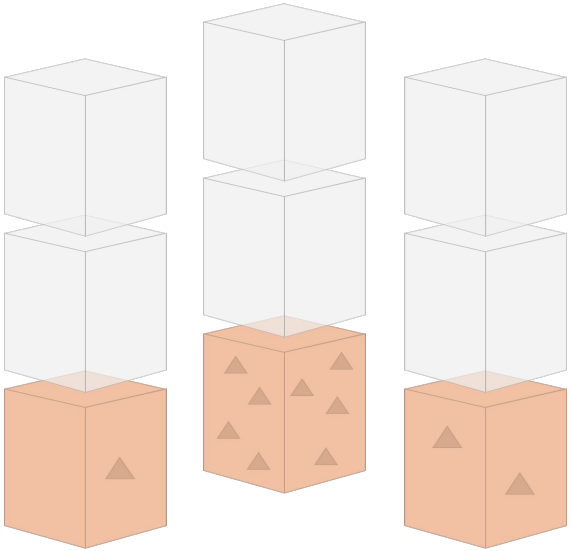
Logic only shared over a well defined interface.



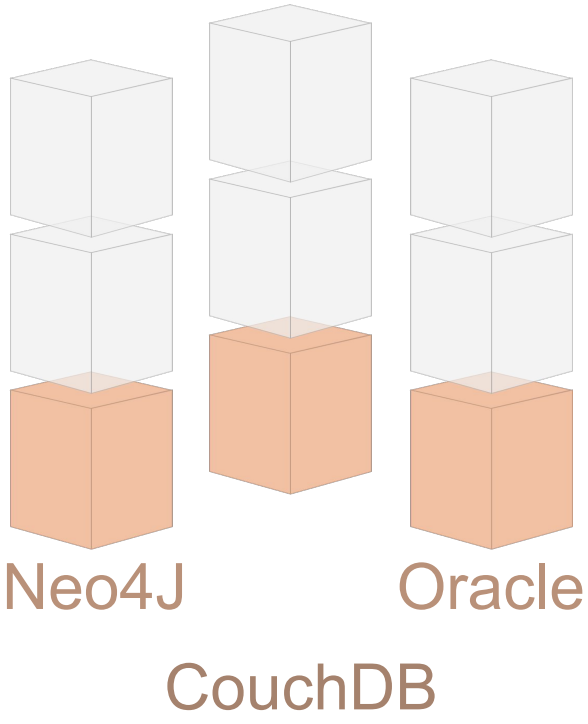
Business logic can consist of microservices



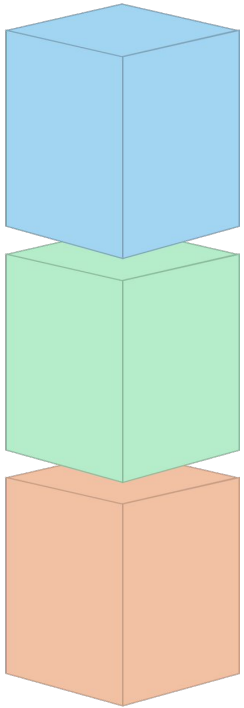
Every SCS brings its own data storage with its own (potentially redundant) data



Redundancies: tolerable as long as sovereignty of data by owning system is not undermined.

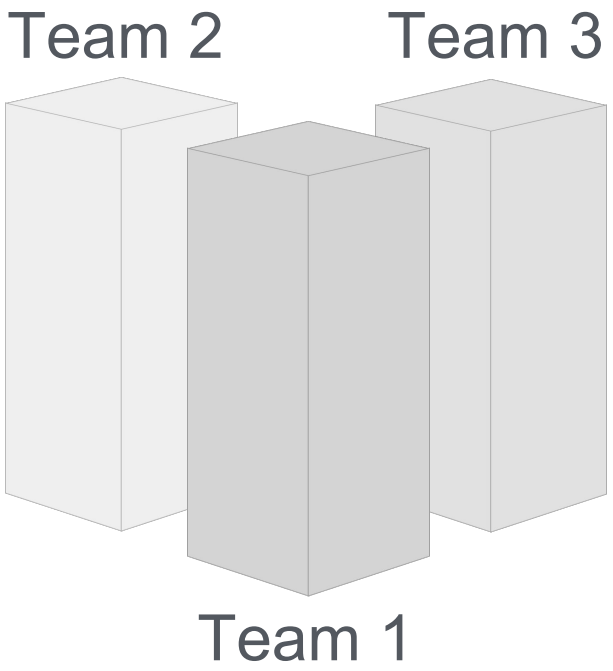


Enables polyglot persistence

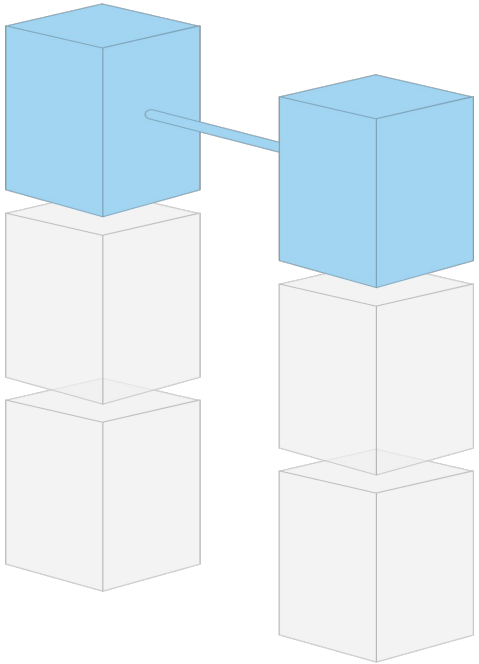


Technical decisions can be made independently from other systems

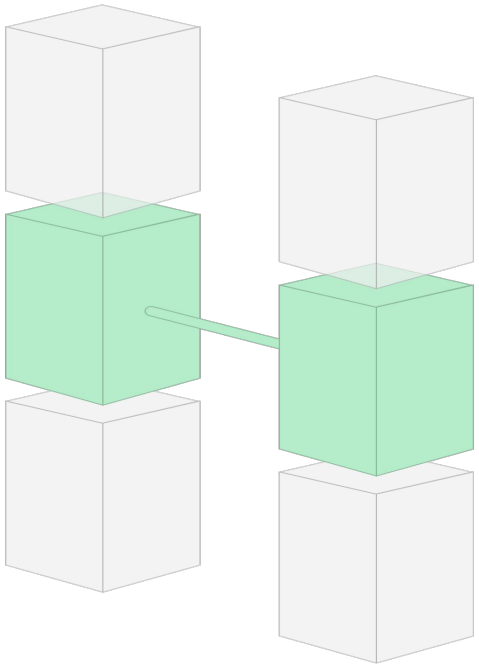
- Programming Language
- Frameworks
- Tooling
- Platform



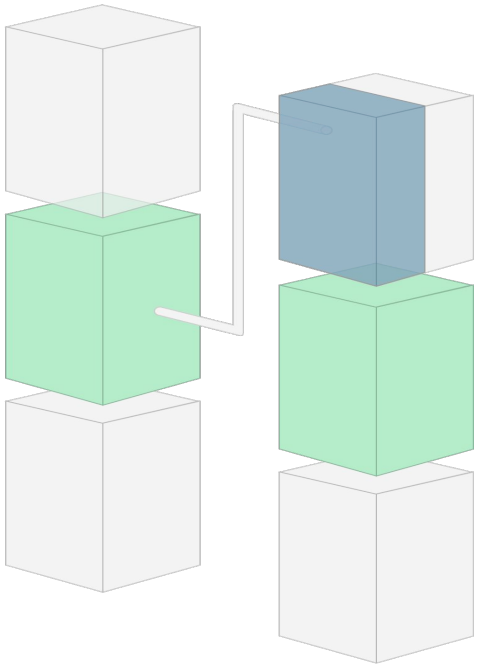
Domain scope enables development, operation and maintenance of SCS by a single team.



SCS
should be integrated via Frontend



Synchronous remote calls inside the business logic should be avoided.



Asynchronous

Remote calls reduce dependencies and prevent error cascades.

SCS: Benefits

- Business logic for one domain in one SCS
- Change usually local to one SCS
- Less communication between Teams

- I think this should be the goal
- http://scs-architecture.org

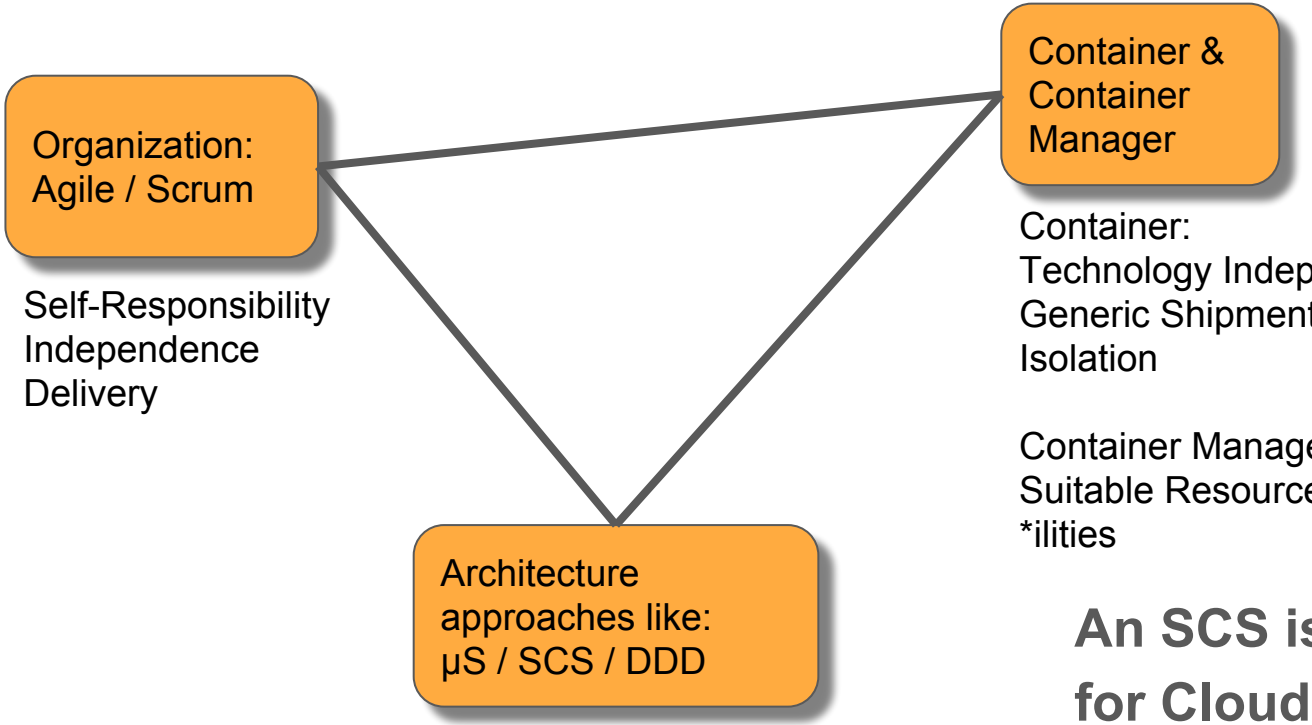
SCS: Conclusion

- SCS: autonomous application
- Might consist of Microservices
- Focus on UI Integration
- Almost completely independent
- Coarse-grained architecture approach

- Self-contained Systems are Microservices …
- that are not “micro”…
- and don’t have to be “services”

http://scs-architecture.org

The winning team



Self-Responsibility
Independence
Delivery

Container:
Technology Independance
Generic Shipment
Isolation

Container Manager:
Suitable Resource Abstraction
*ilities

An SCS is a great fit for Cloud Native Architectures

CQRS

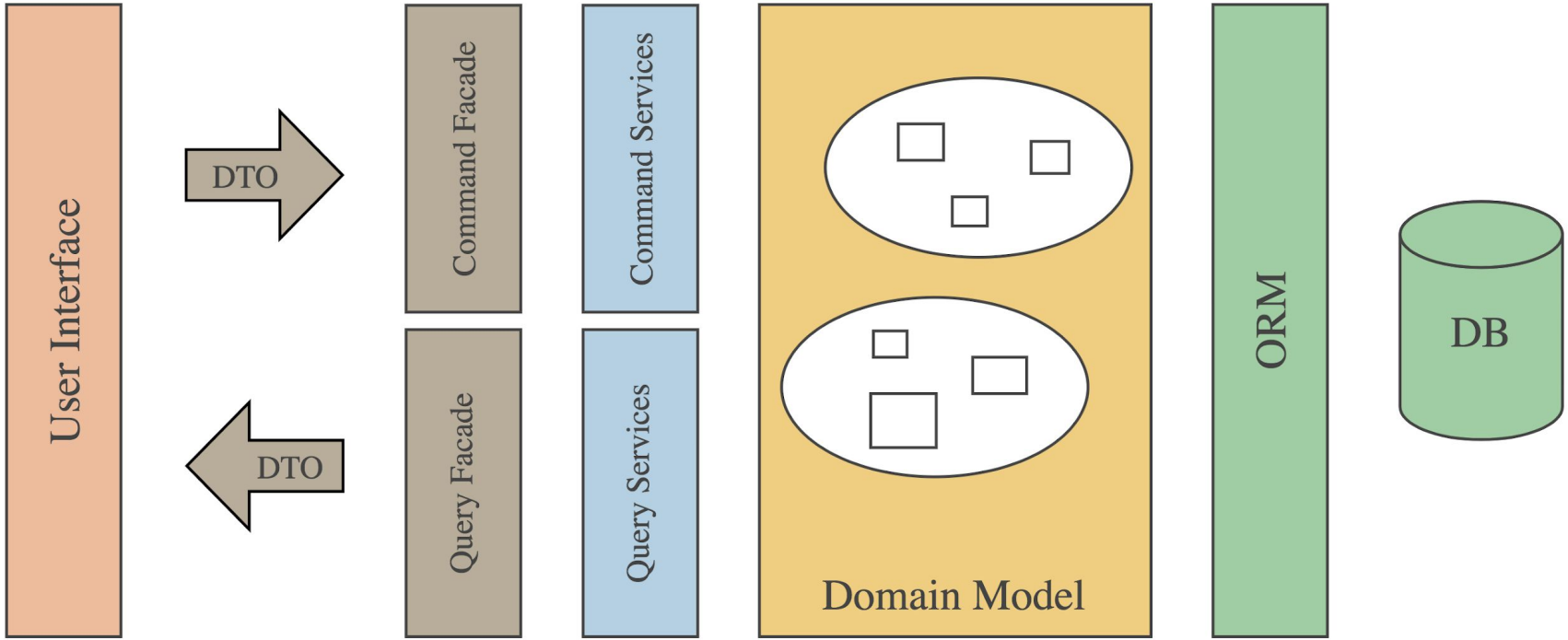
CQRS

```
interface CustomerService {
  void updateCustomer(Customer);
  CustomerList findCustomers(CustomerQuery);
  Customer getCustomer(ID);
  void deleteCustomer(ID);
}
```

CQRS

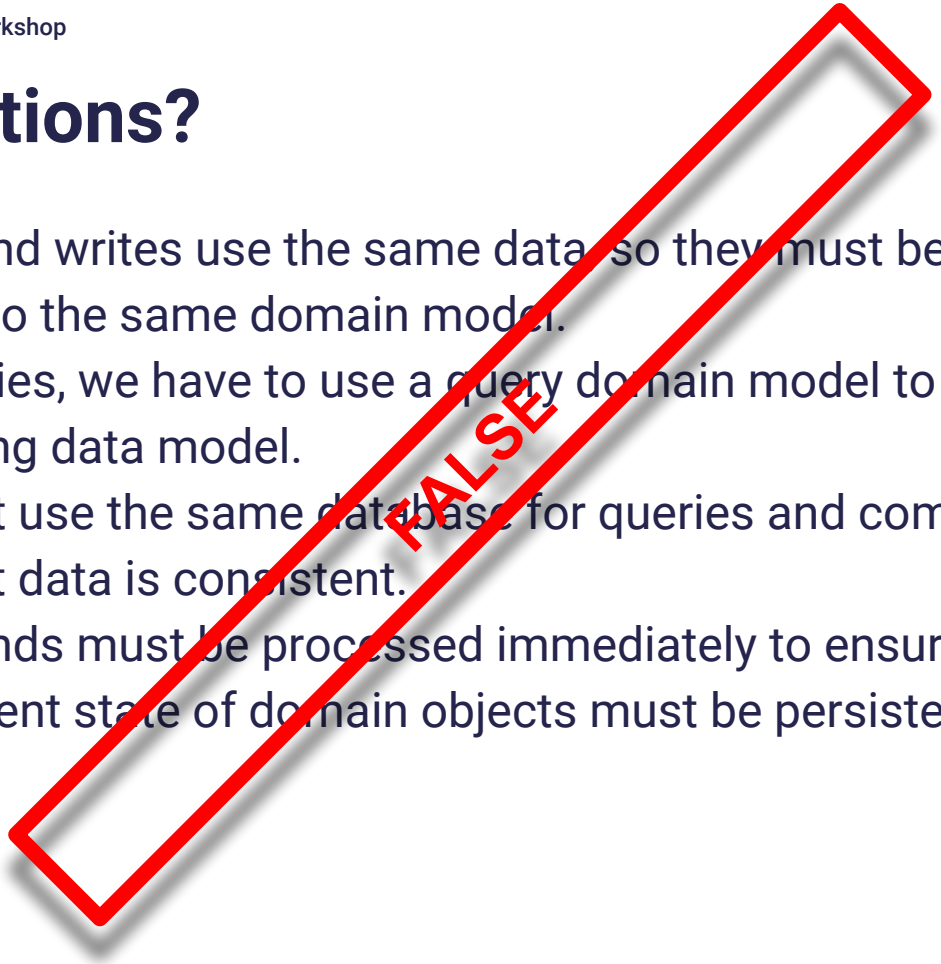
```
interface CustomerQueryService {
  CustomerList findCustomers(CustomerQuery);
  Customer getCustomer(ID);
}

interface CustomerCommandService {
  void updateCustomer(CustomerUpdateCommand);
  void deleteCustomer(ID);
}
```

CQRS Pattern Applied

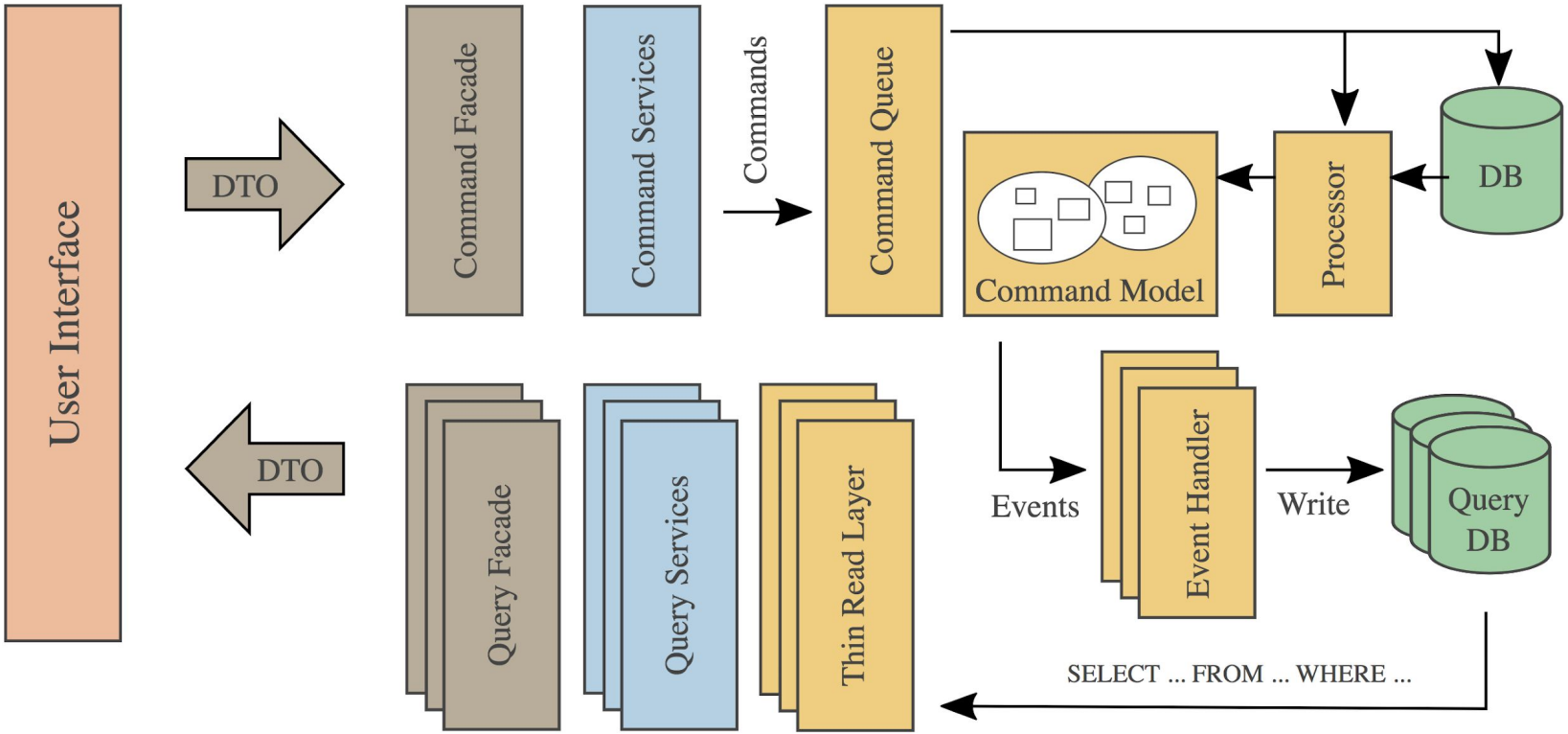


Assumptions?

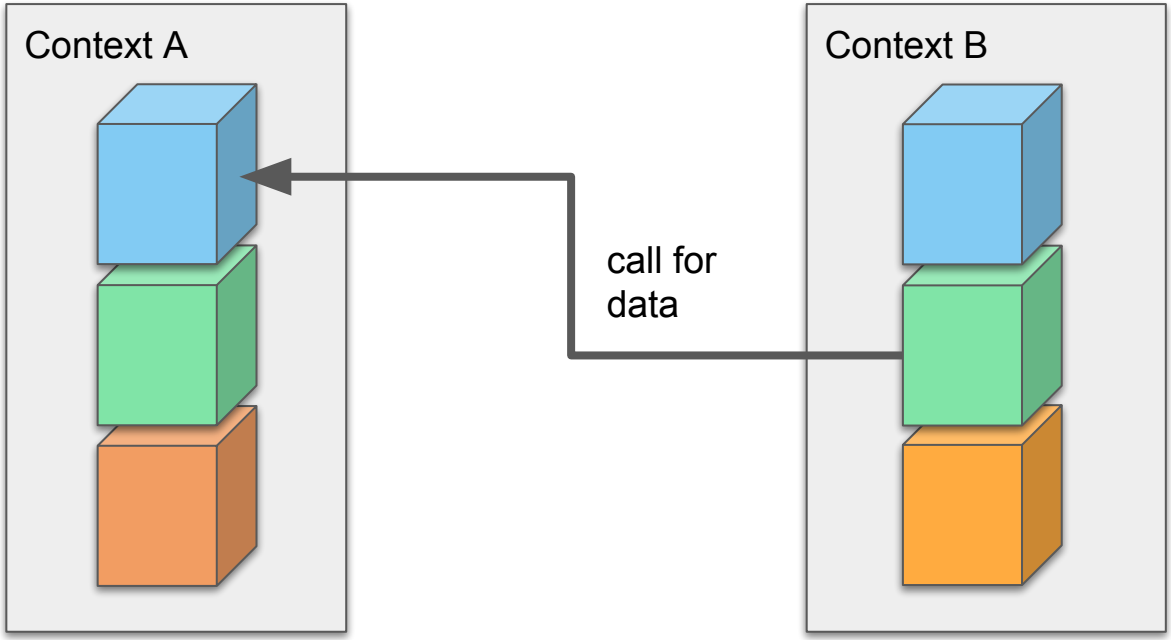
- Reads and writes use the same data, so they must be served from and applied to the same domain model.
- For queries, we have to use a query domain model to abstract from the underlying data model.
- We must use the same database for queries and commands to make sure that data is consistent.
- Commands must be processed immediately to ensure data consistency.
- The current state of domain objects must be persistent.



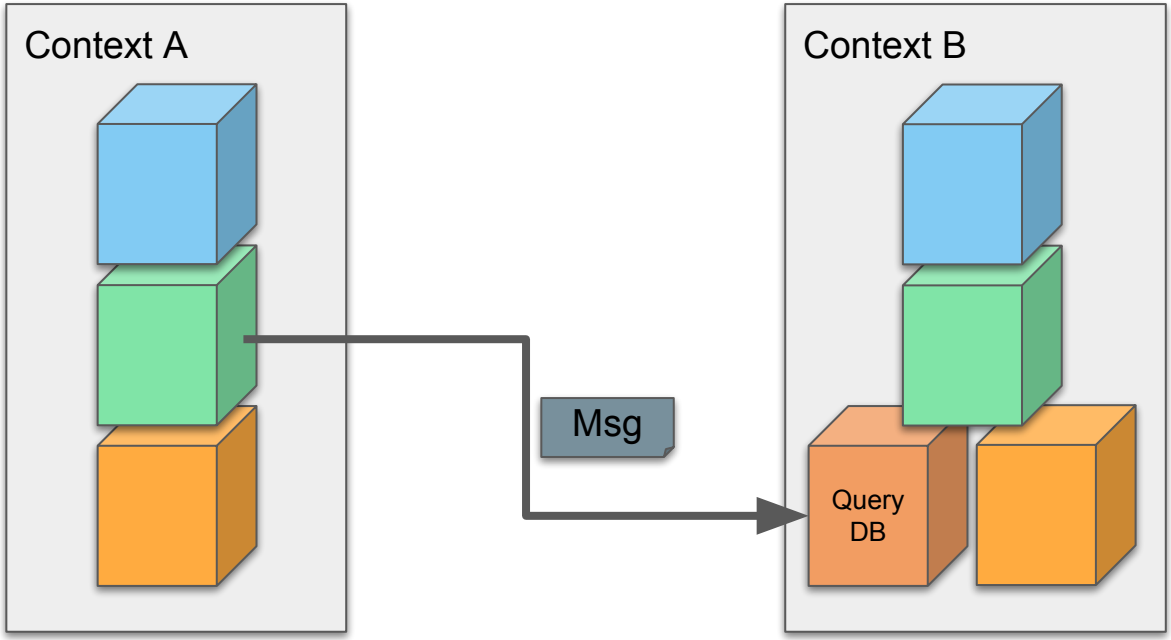
CQRS Pattern Applied



CQRS Pattern Applied (SCS)



CQRS Pattern Applied (SCS)



cloud-native journey

- Simplicity wins over complexity
- Distributed wins over centralized
- Security is woven into the network
- Routine operations are automated

Container &
Container Manager

Effect on SW Architecture?

Some general Container rules...

General Container Rules

- Modularize to Container
  - Runtime wise (logging, proxy, SSL termination. etc.)
  - Architecture wise (f.e. Use cases, context bounds, etc.)
- Plan your application
  - Logging
  - Volumes
  - Configuration / Environment
  - Base images
  - Scaling
- One image per service/process, one service per image

General Container Rules

- No image difference between DEV and PROD
- Keep Runtime state in Volumes (immutability)
- Add dependencies at image build time (self contained)
- Images should be as small as possible
  - Use `FROM Scratch` (Kernel only)
  - Small base images (f.e. Alpine)
  - Reuse as custom base images
- Resilient to dead connections (disposable container)

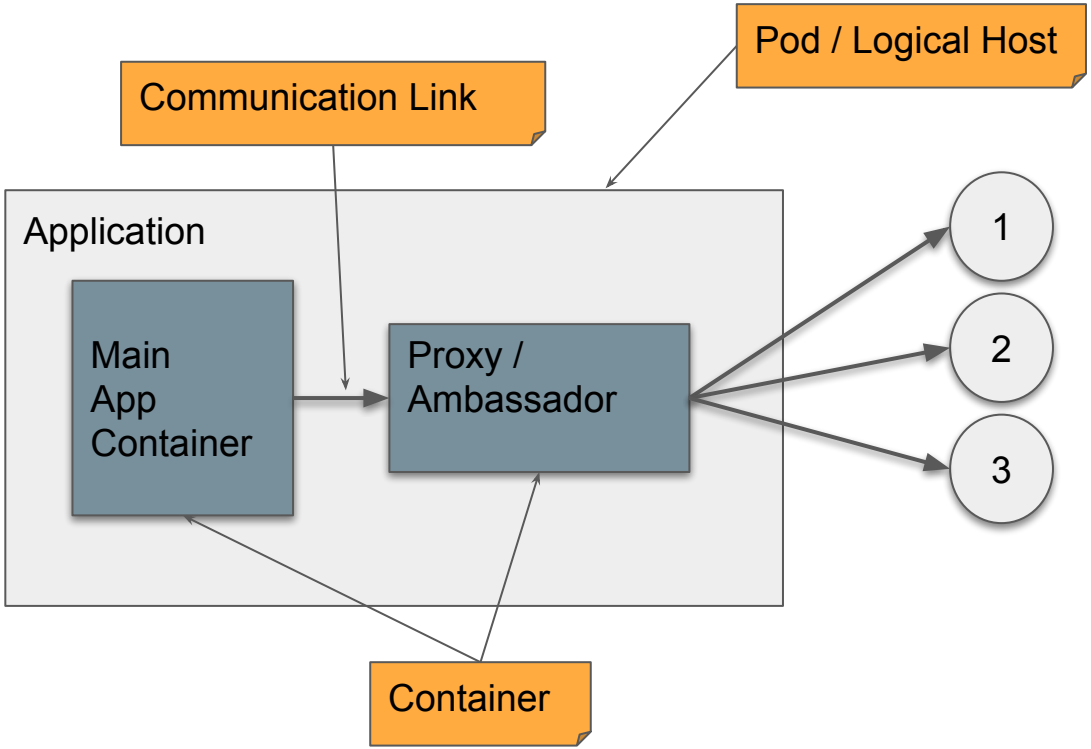
Are there Pattern for Container?

What are Container Pattern?

- Abstract away the low-level details
- Is valid for container as it is for OOP
- Reveal general reusable solutions to a commonly occurring problem
  - Simplifies reuse of images
- Can help to modularize on container level
  - Separation of Concerns
  - Isolation
- Perfectly fit to abstractions like Logical Hosts or Services (Container Manager)

Single Node, Multi Container Pattern

Ambassador Pattern



Ambassador Pattern

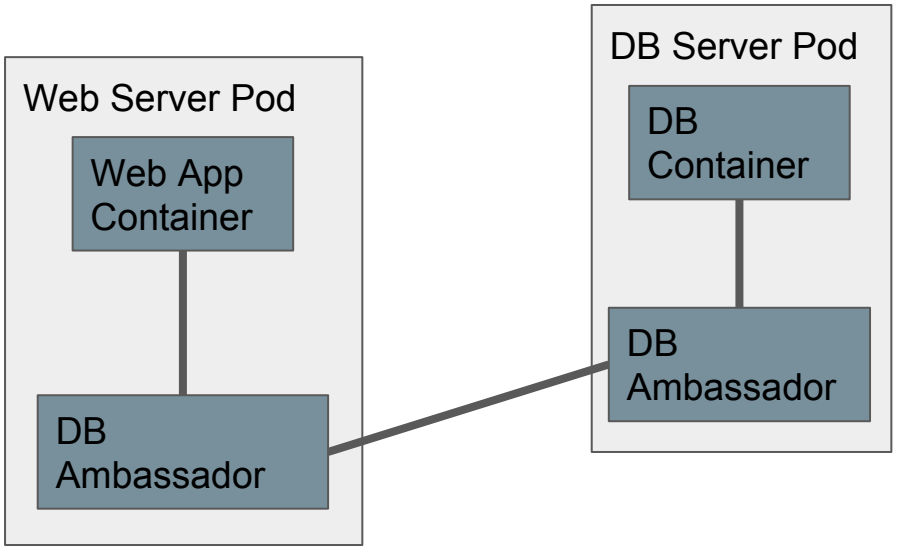
Can be used to abstract communication or infrastructure details

- Protocol switch
- Authorization
- Encryption
- Failover
- Circuit Breaker Pattern
- . . .

Ambassador Pattern Example

Need to rewire the Web Server to a different DB?

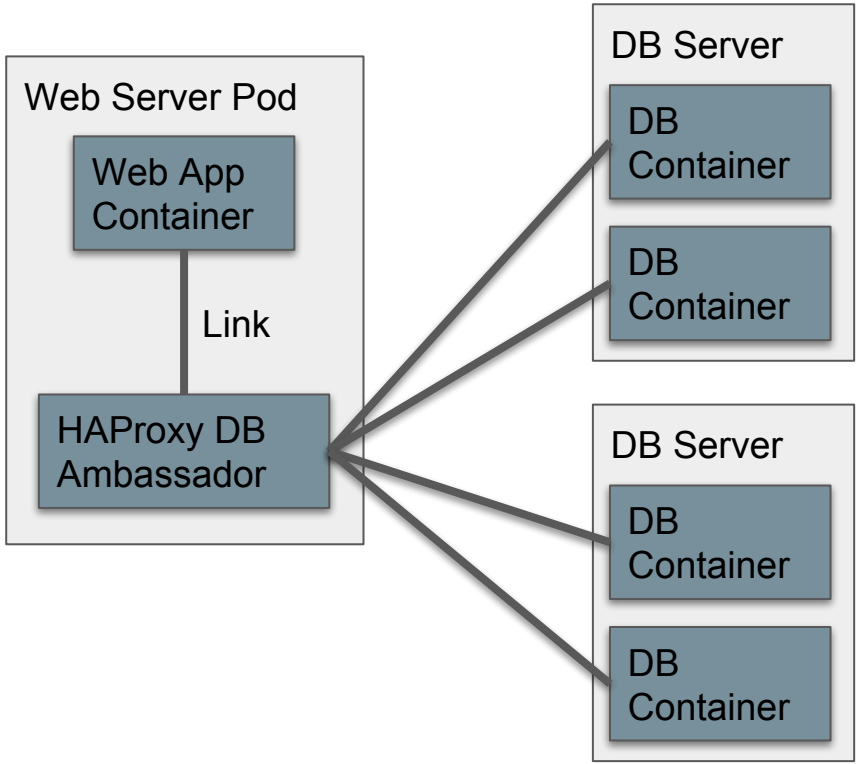
Just reconfigure and restart the DB Ambassador



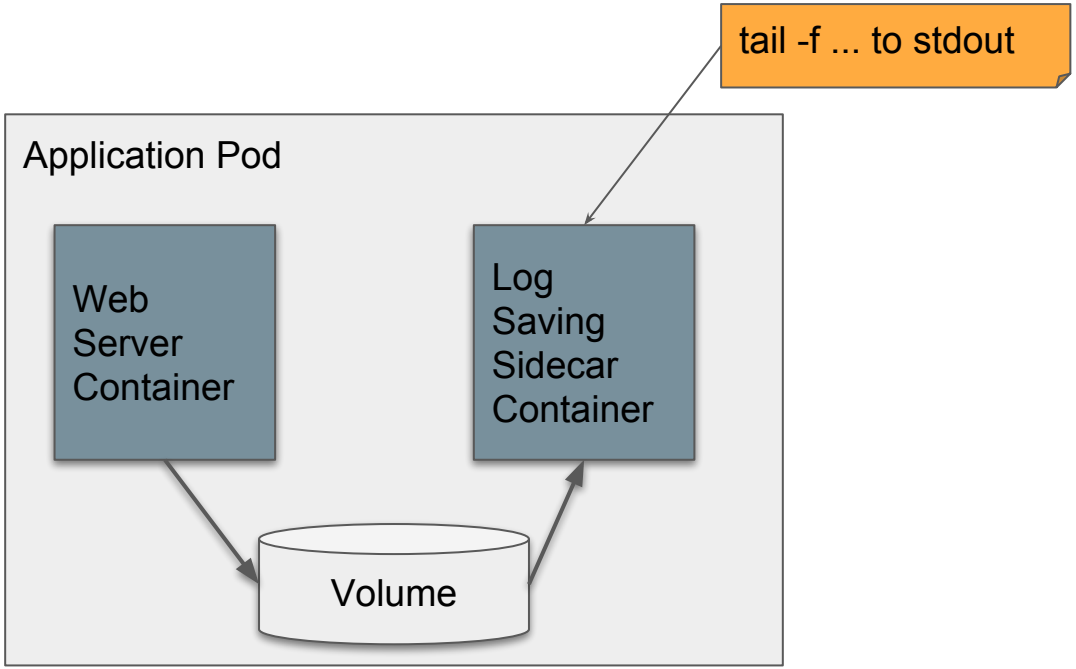
Ambassador Pattern Example

Also possible:

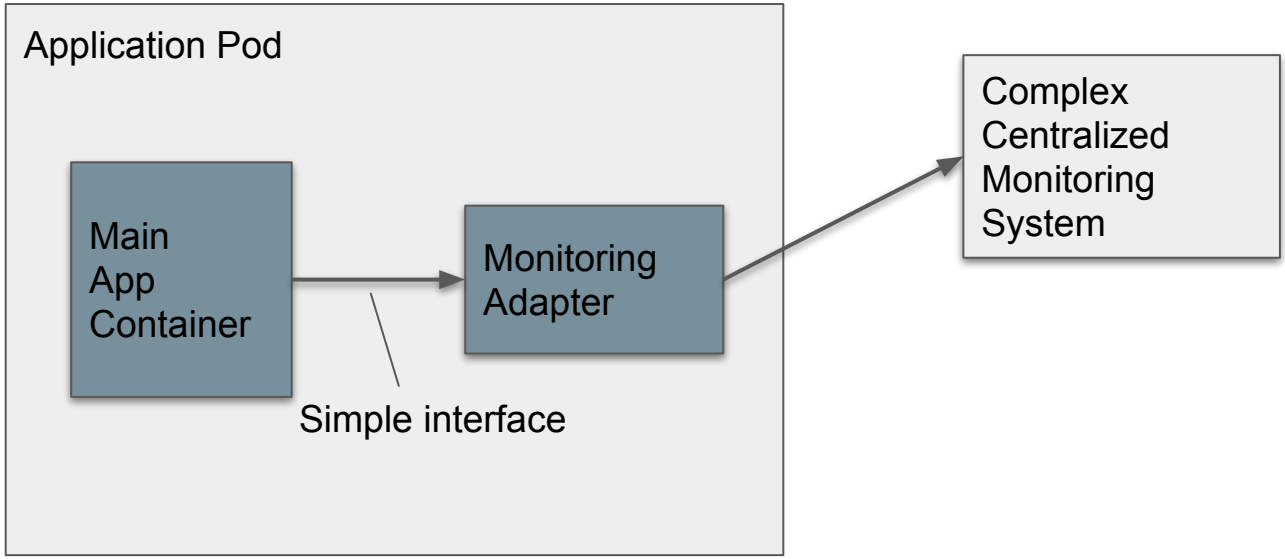
- Dynamic Scaling
- Failover
- etc.



Sidecar / Sidekick pattern

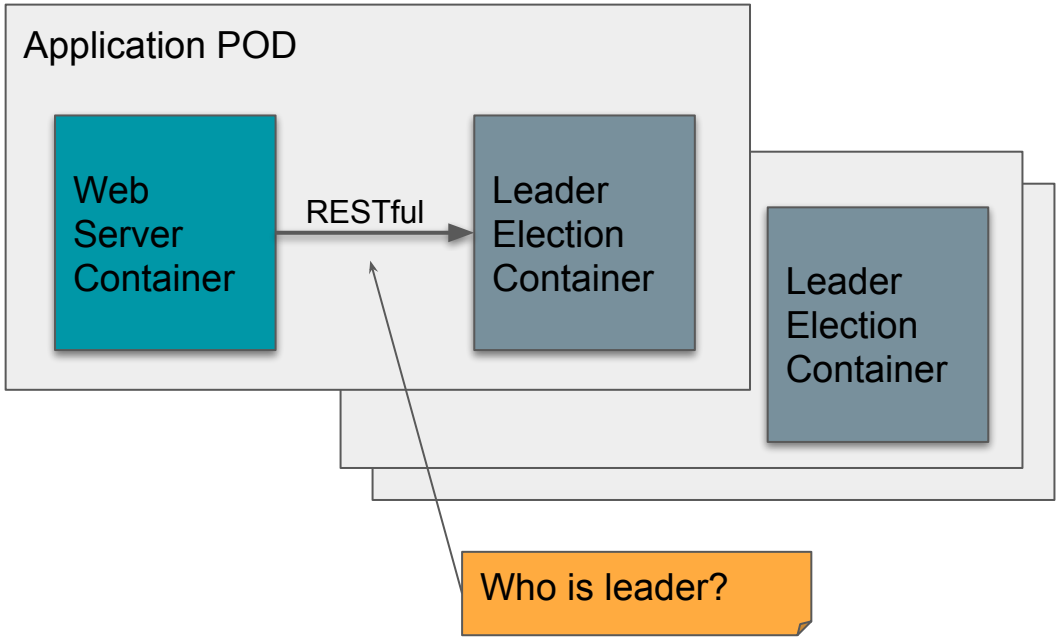


Adapter Pattern

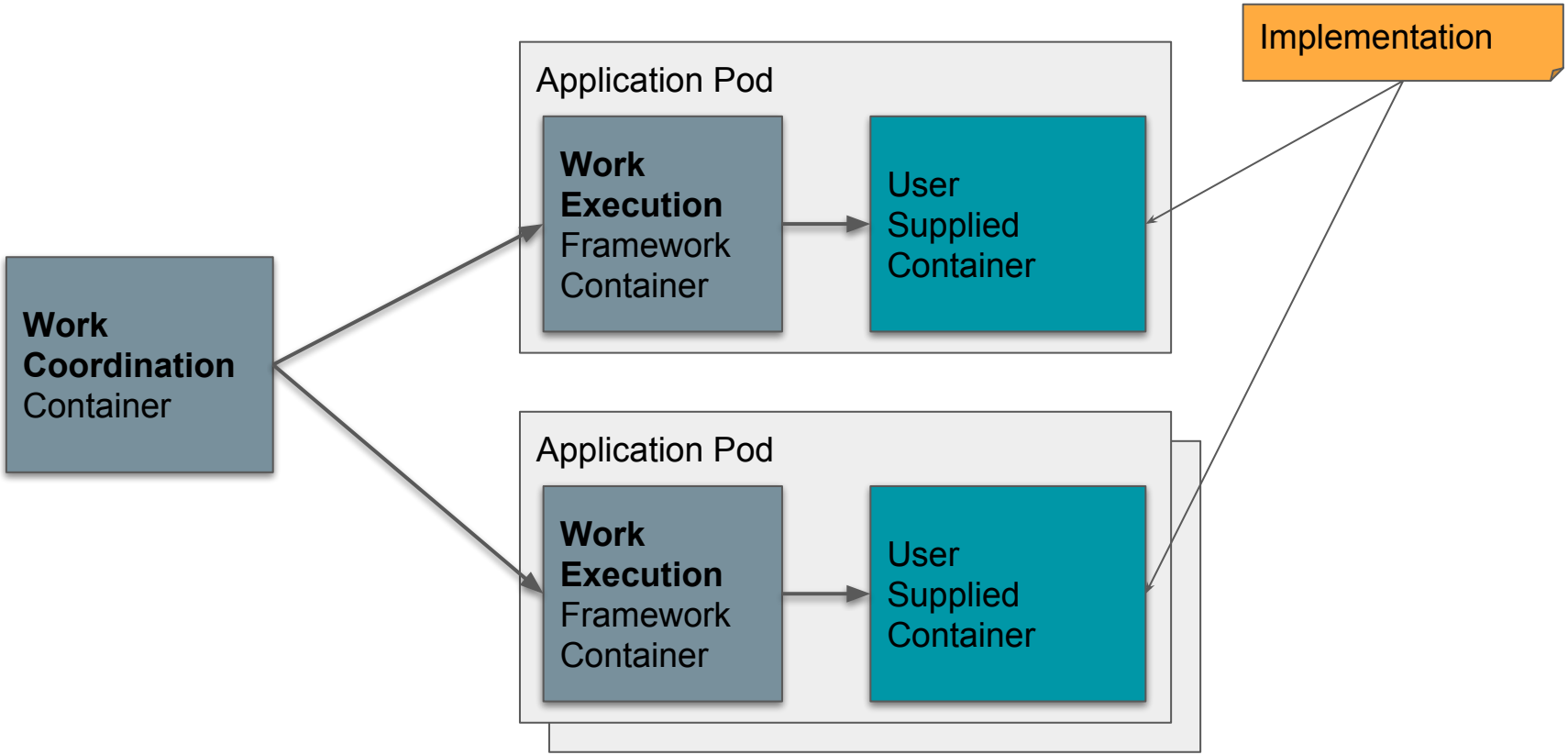


Multi Node Application Pattern

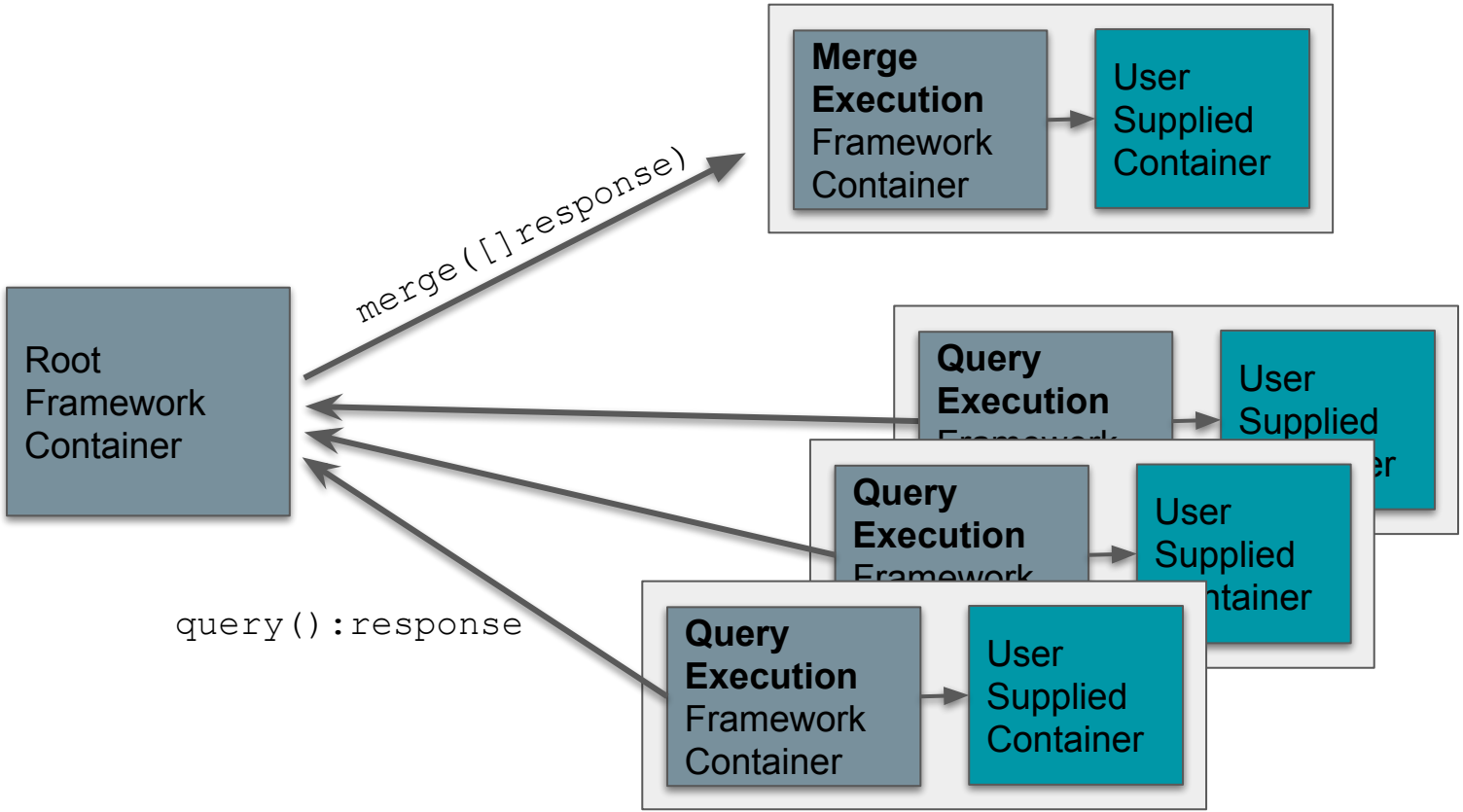
Leader Election (Conversation Pattern)



Work Queue Pattern



Scatter / Gather (Messaging Pattern)



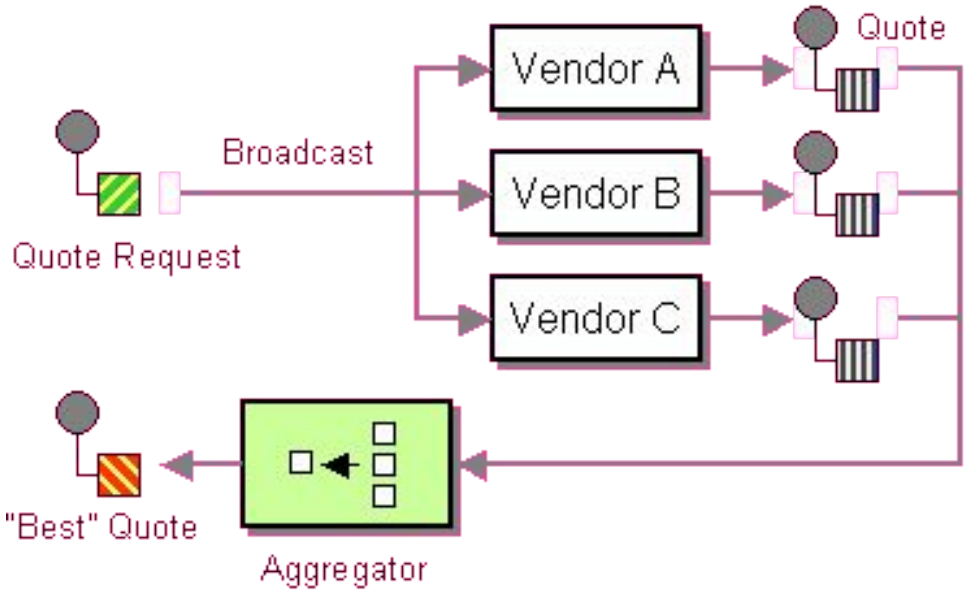
Example Application Container Design

Aggregate quotes from vendors to get the “best” one

Example: Best Quote

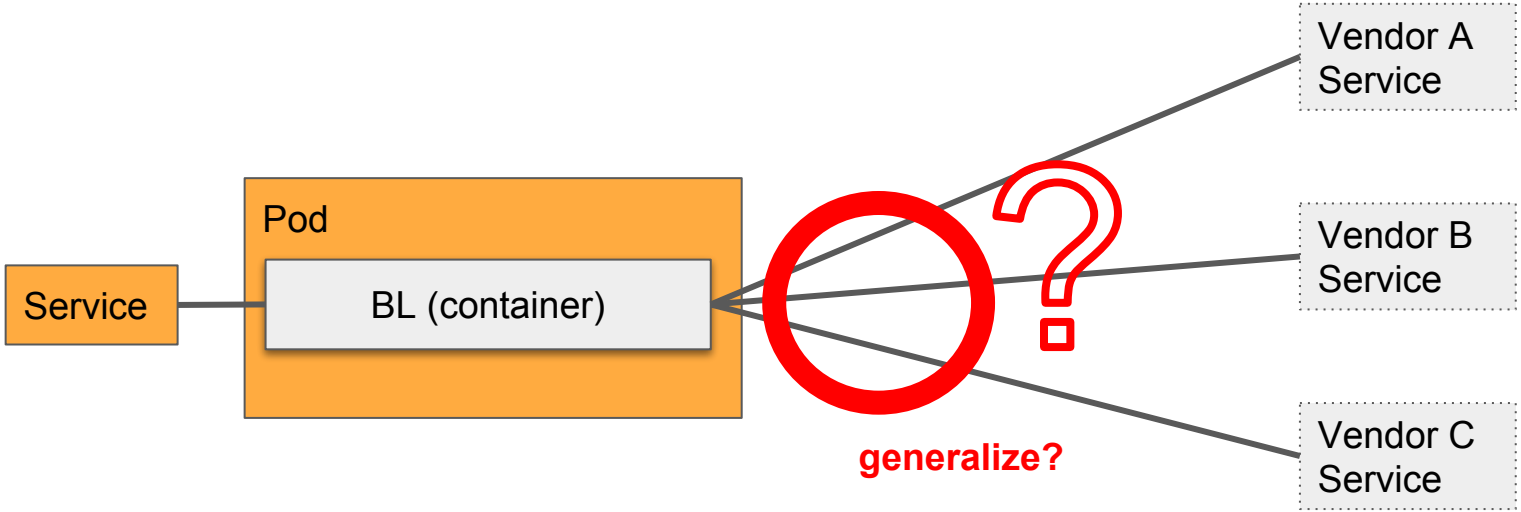
- Several external APIs to be called for quotes
- APIs can be HTTP/JSON, HTTP/XML, FTP/XLS etc.
- No standard semantics
- Best quote is not simply lowest price

Example: Best Quote

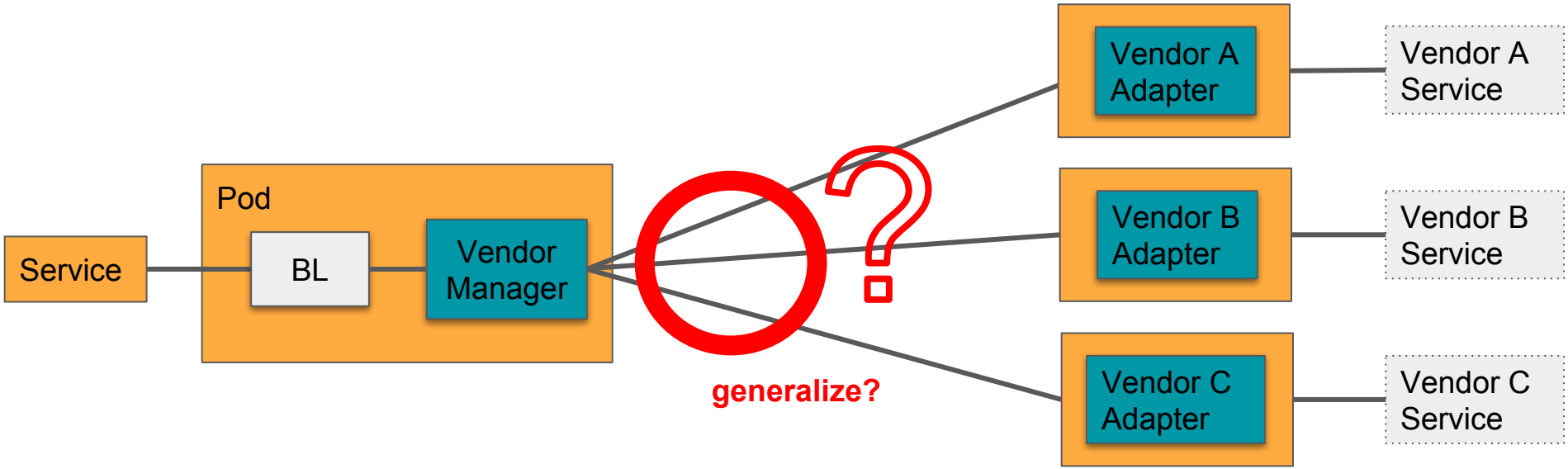


http://www.enterpriseintegrationpatterns.com/

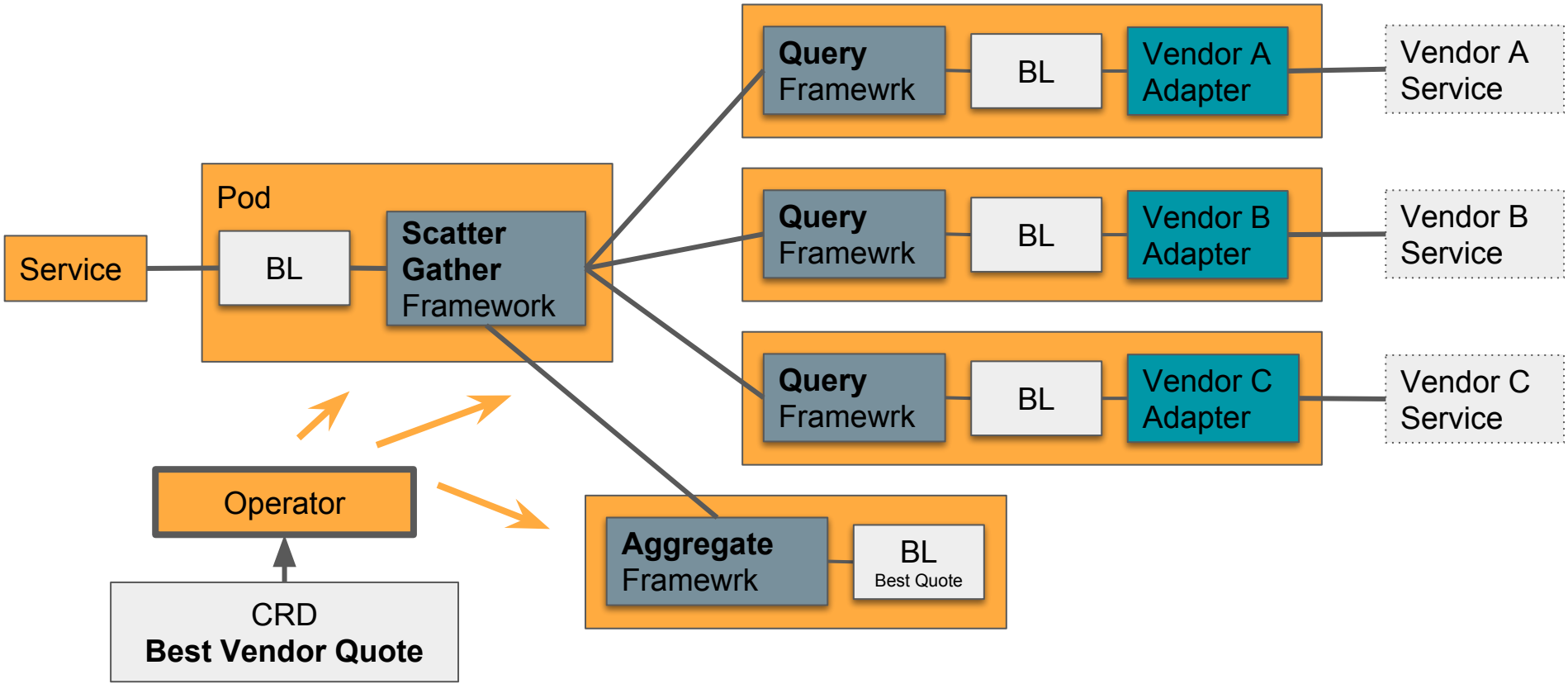
Example: Best Quote



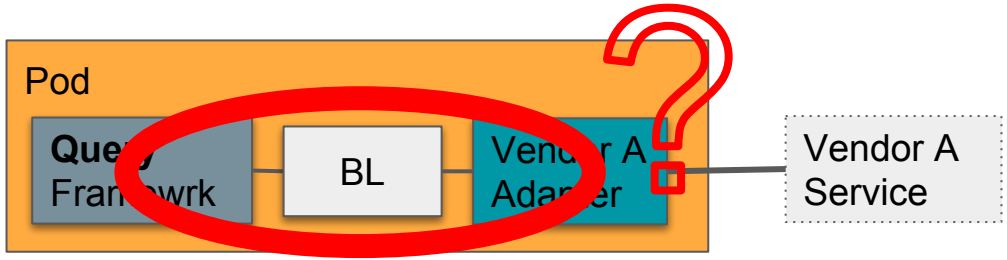
Example: Best Quote



Example: Best Quote



Example: Best Quote



- 1 POD
- 3 Container
- Calling each other by gRPC / REST

- 1 POD
- 3 Container
- “Calling” each other by messaging (f.e. Vert.x)

- 1 POD
- 1 Container (build by FROM inheritance)
- Calling each other by binary exec

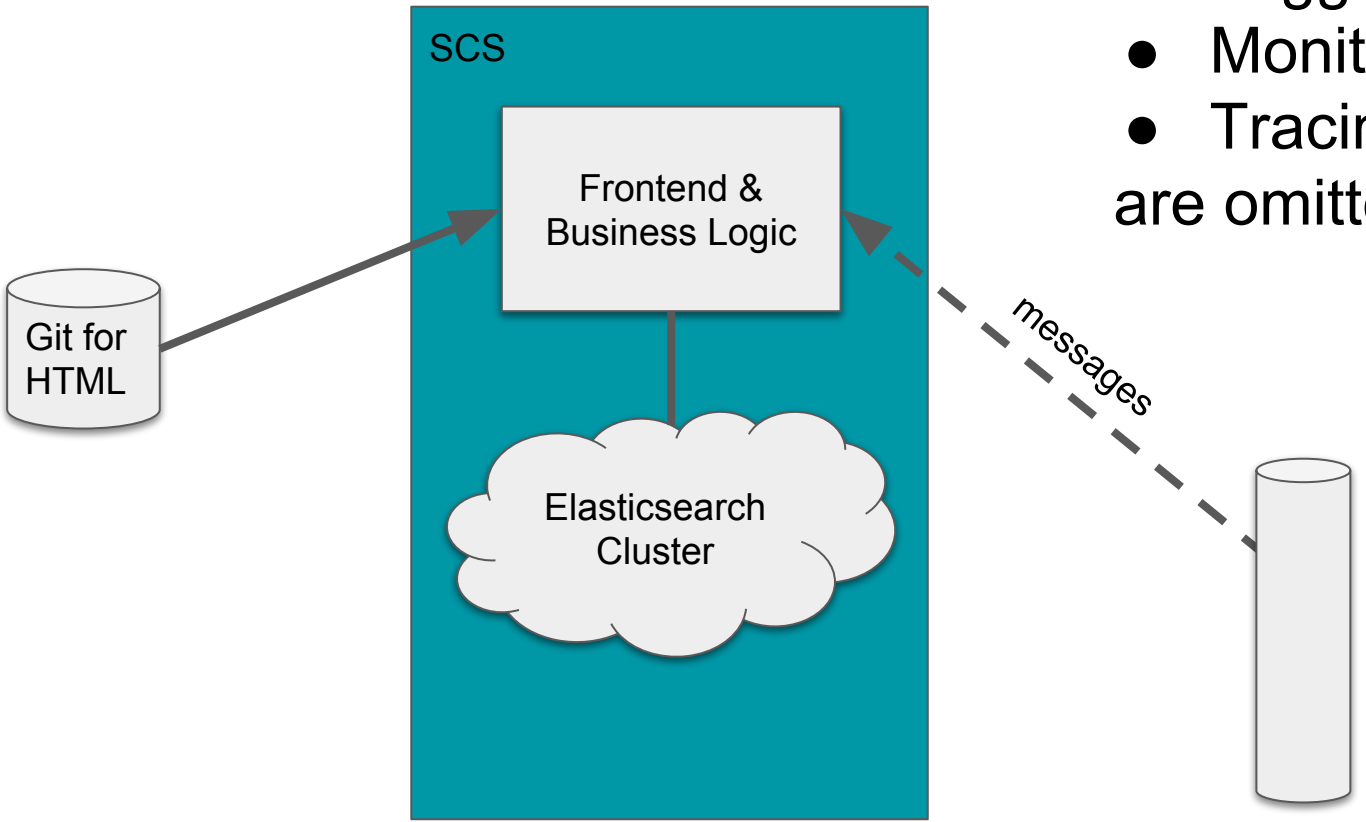
Example Application Container Design

FE with ES

Example: FE with ES

- F.e. Hotel Main Page
- New hotels are delivered by JSON messages (message system)
- Every consumed message is processed and stored as HTML in ES
- Static HTML is taken from a GIT repository

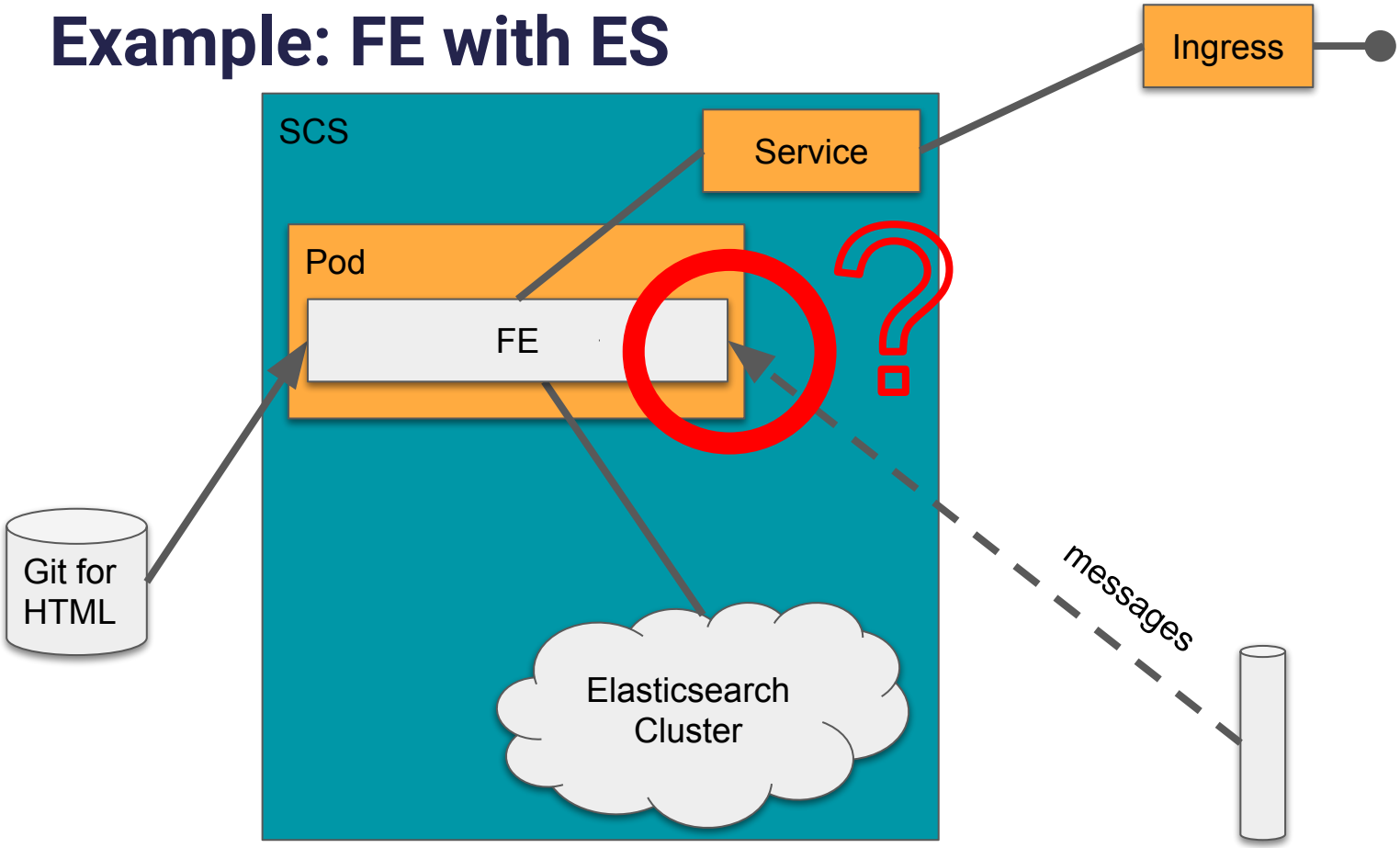
Example: FE with ES



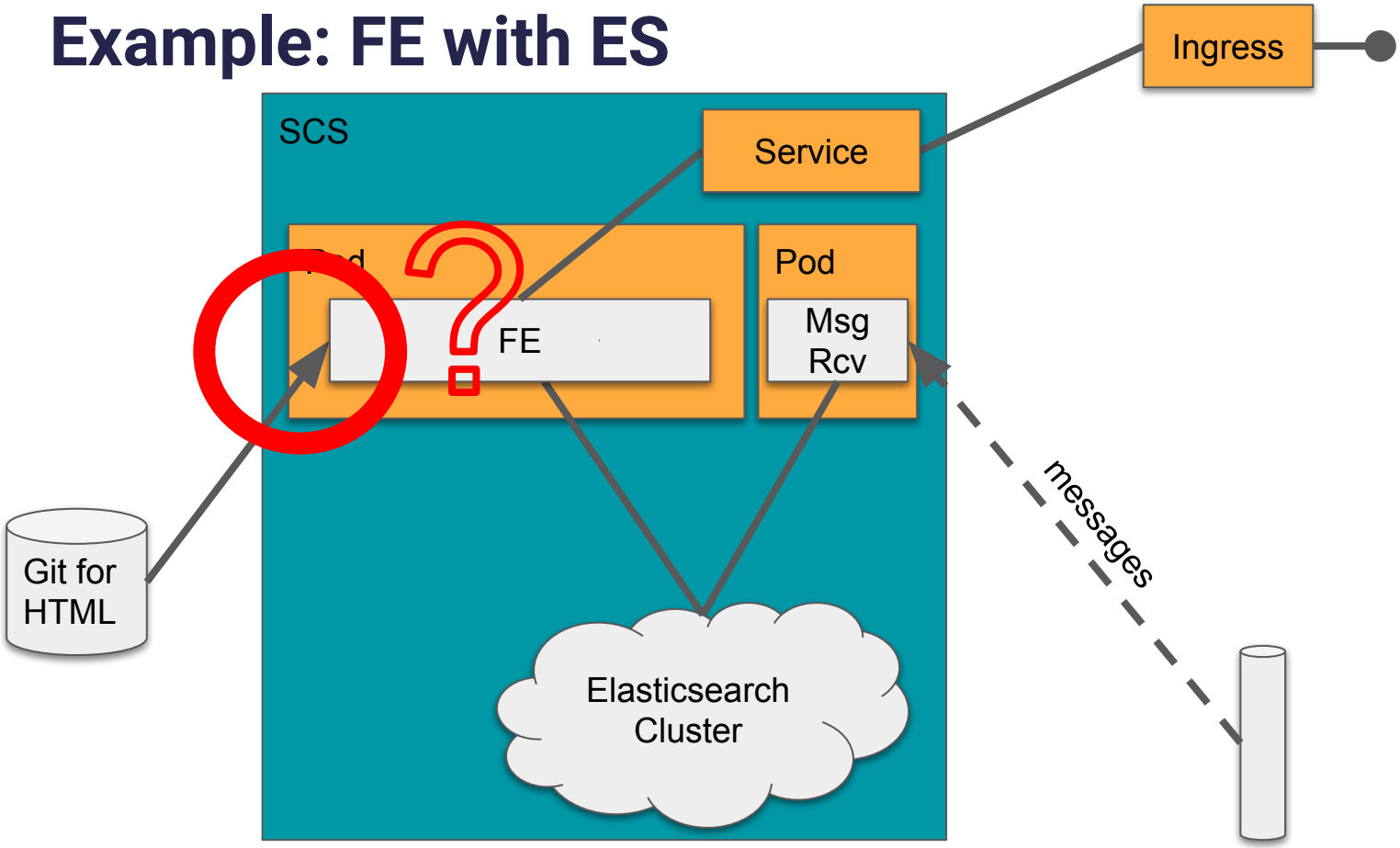
- Logging
- Monitoring
- Tracing

are omitted

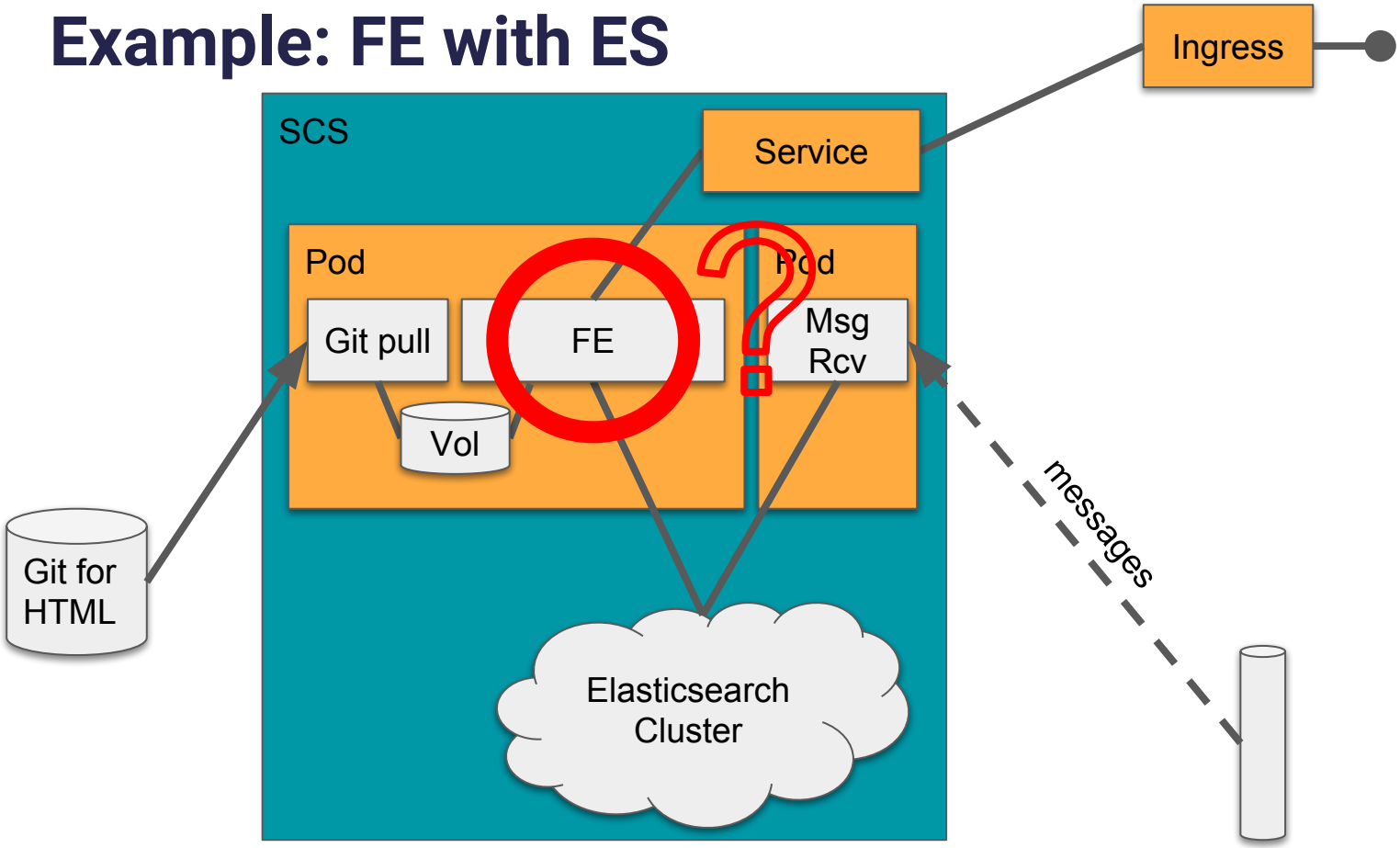
Example: FE with ES



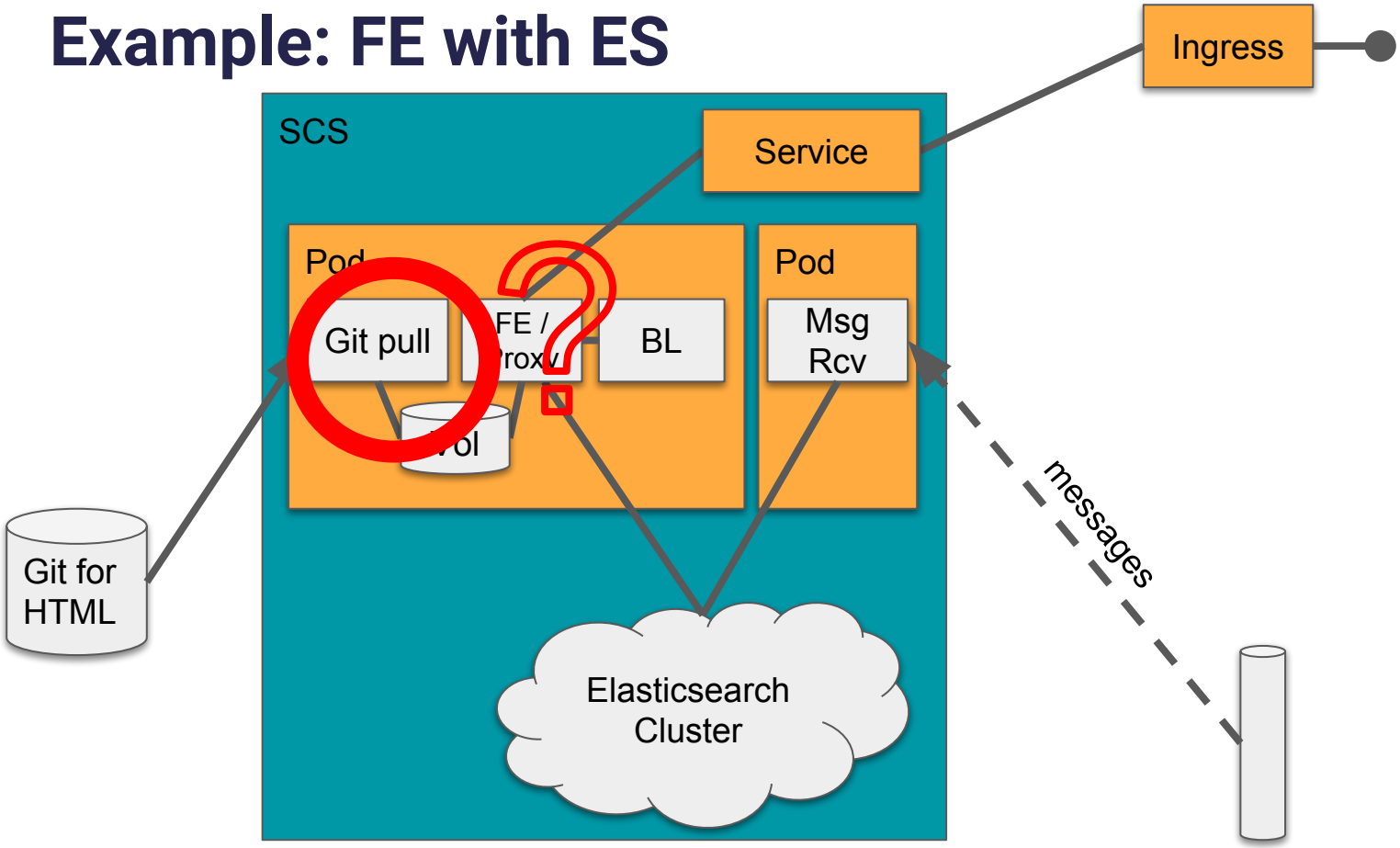
Example: FE with ES



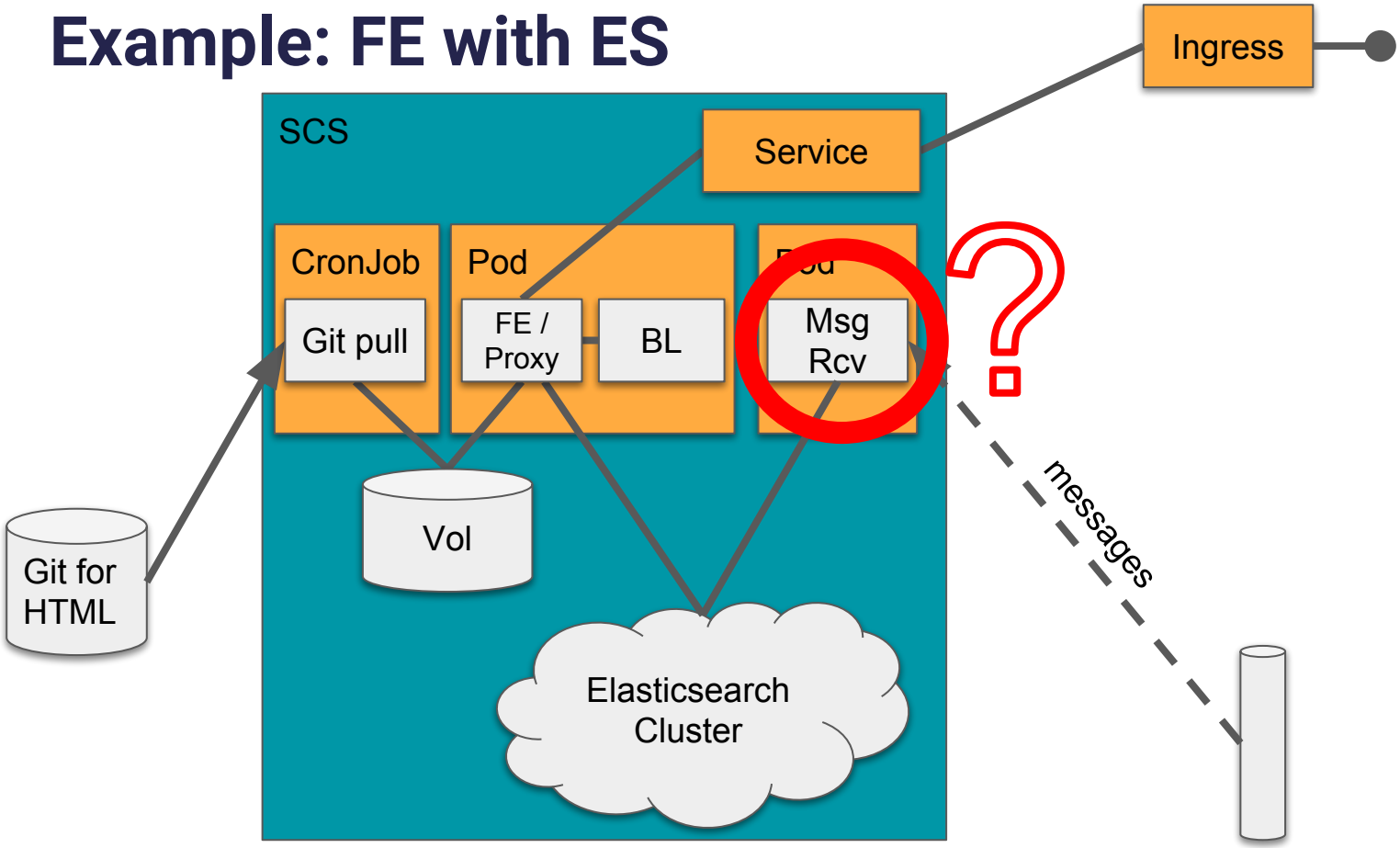
Example: FE with ES



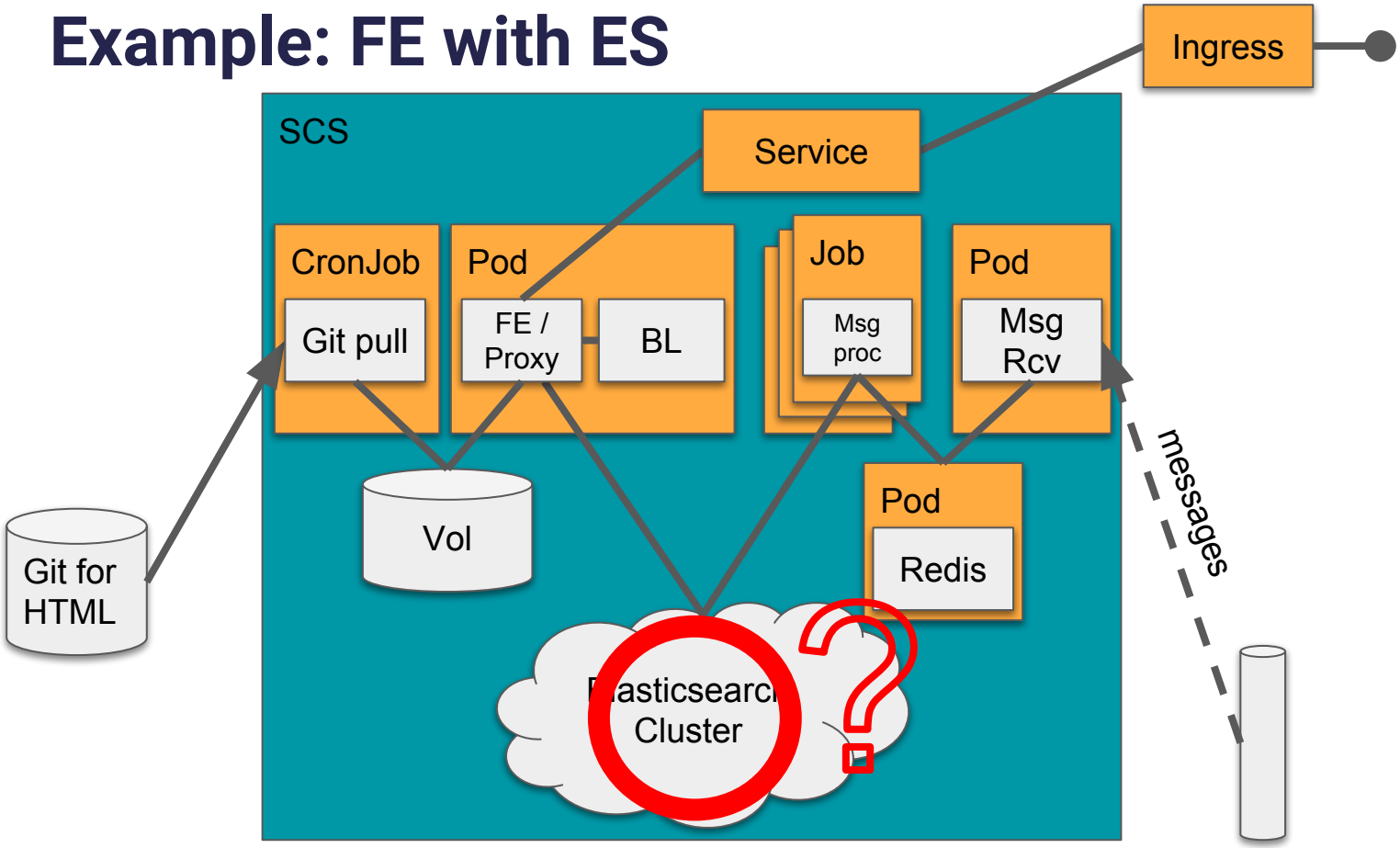
Example: FE with ES



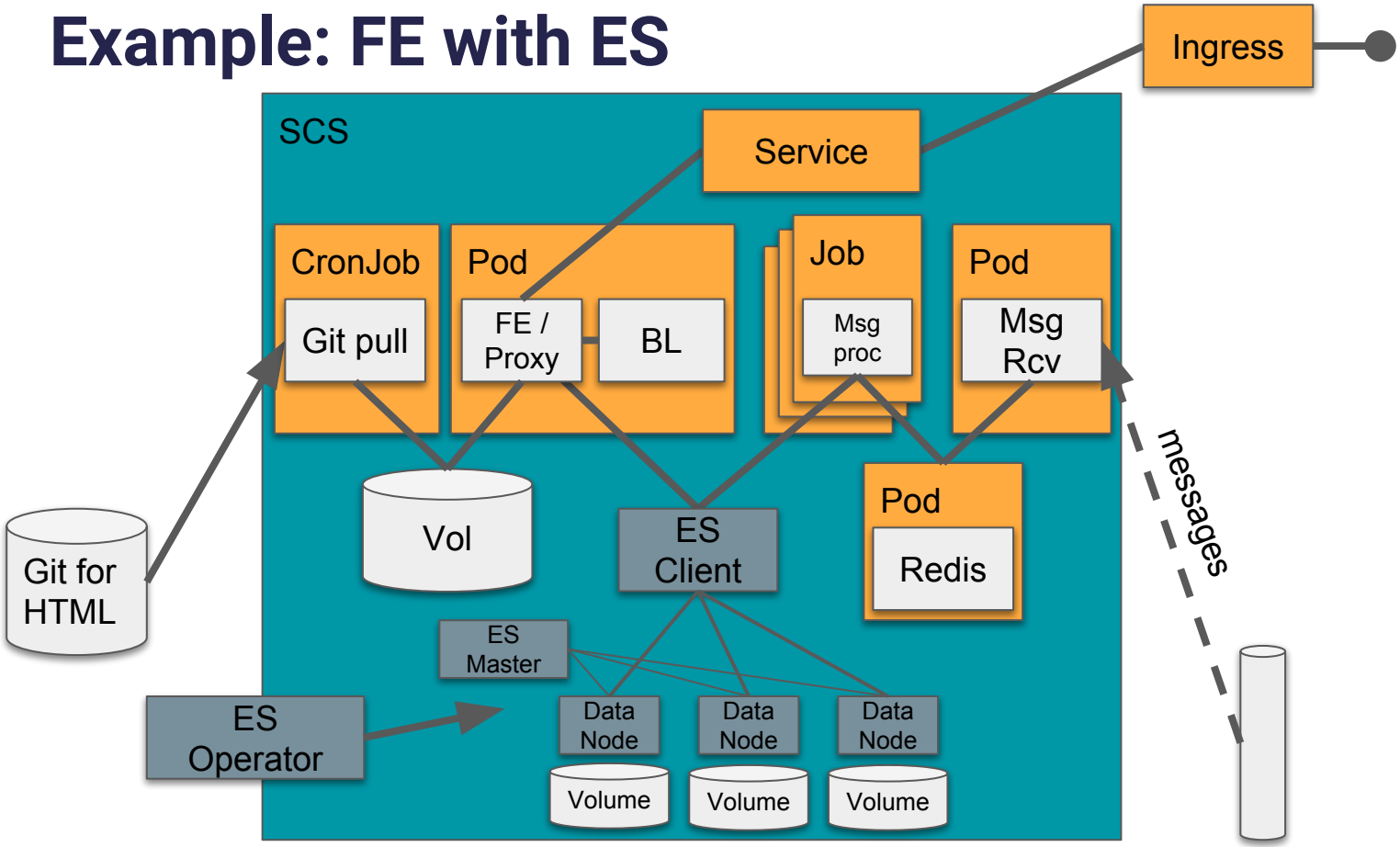
Example: FE with ES



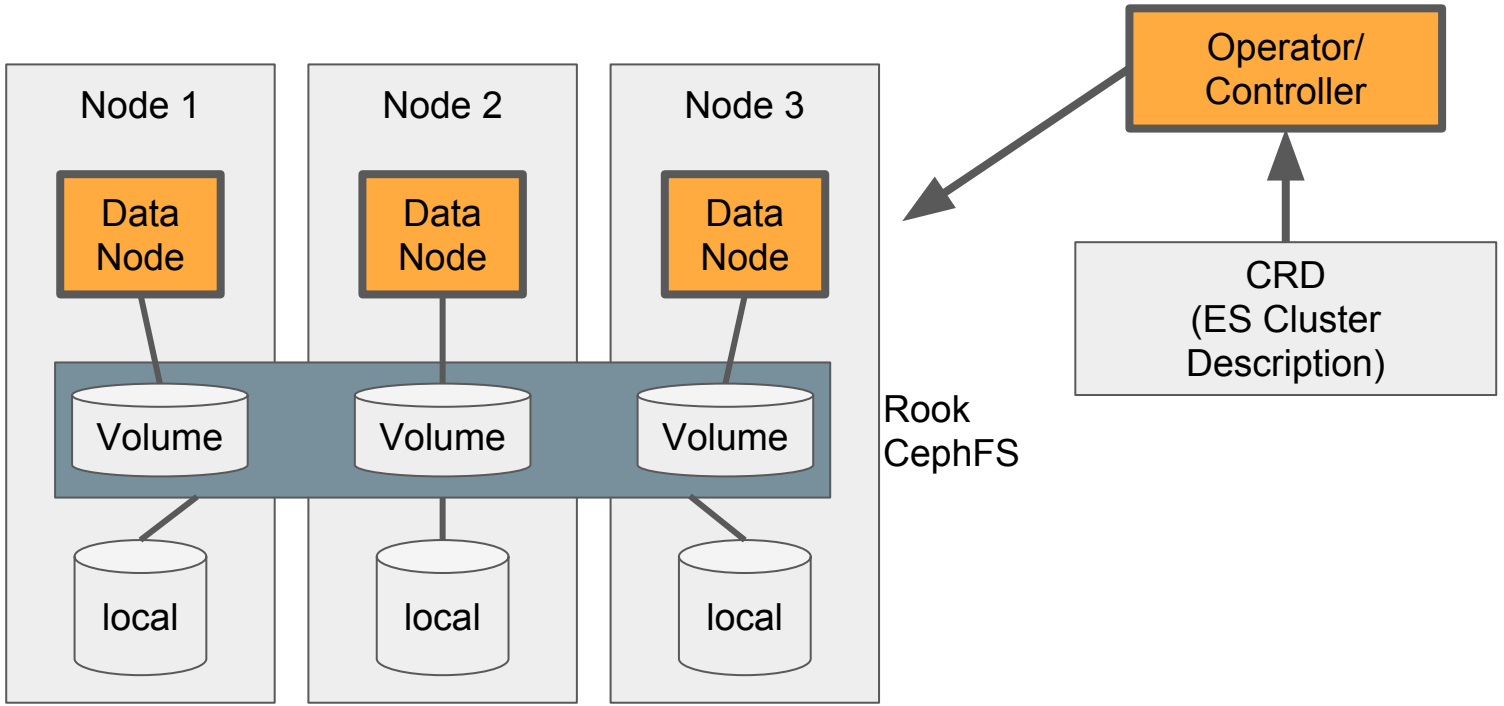
Example: FE with ES



Example: FE with ES



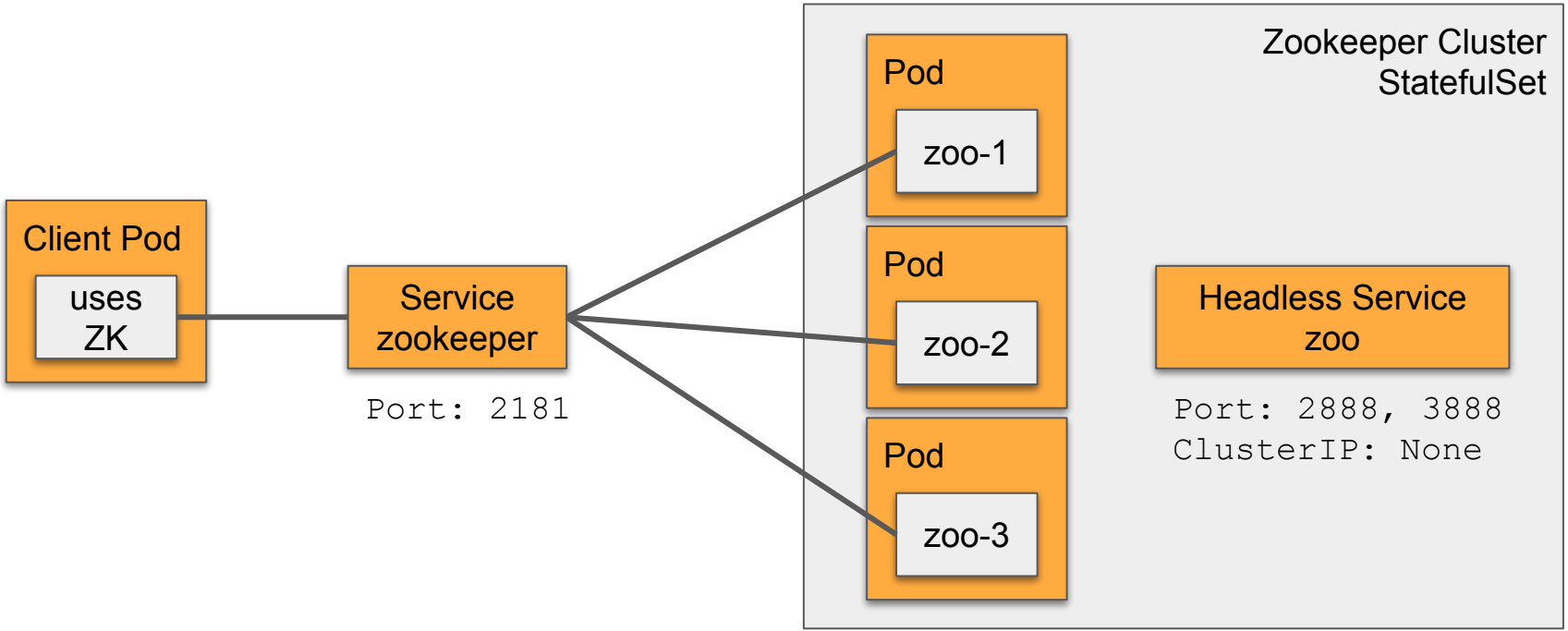
Elasticsearch Operator



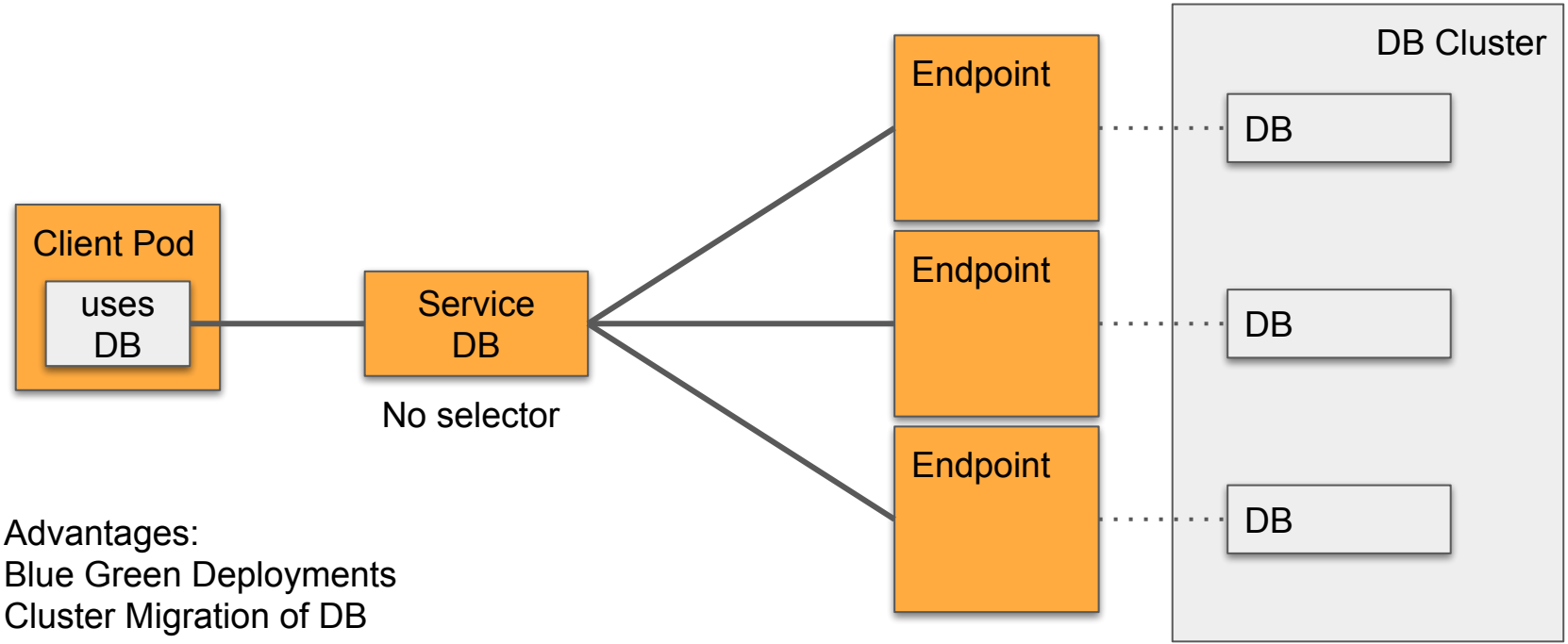
Example Application Container Design

Service Pattern Examples

Service Pattern: Intern, Extern Services



Service Pattern: Proxy by Endpoints



Advantages:
Blue Green Deployments
Cluster Migration of DB
. . .