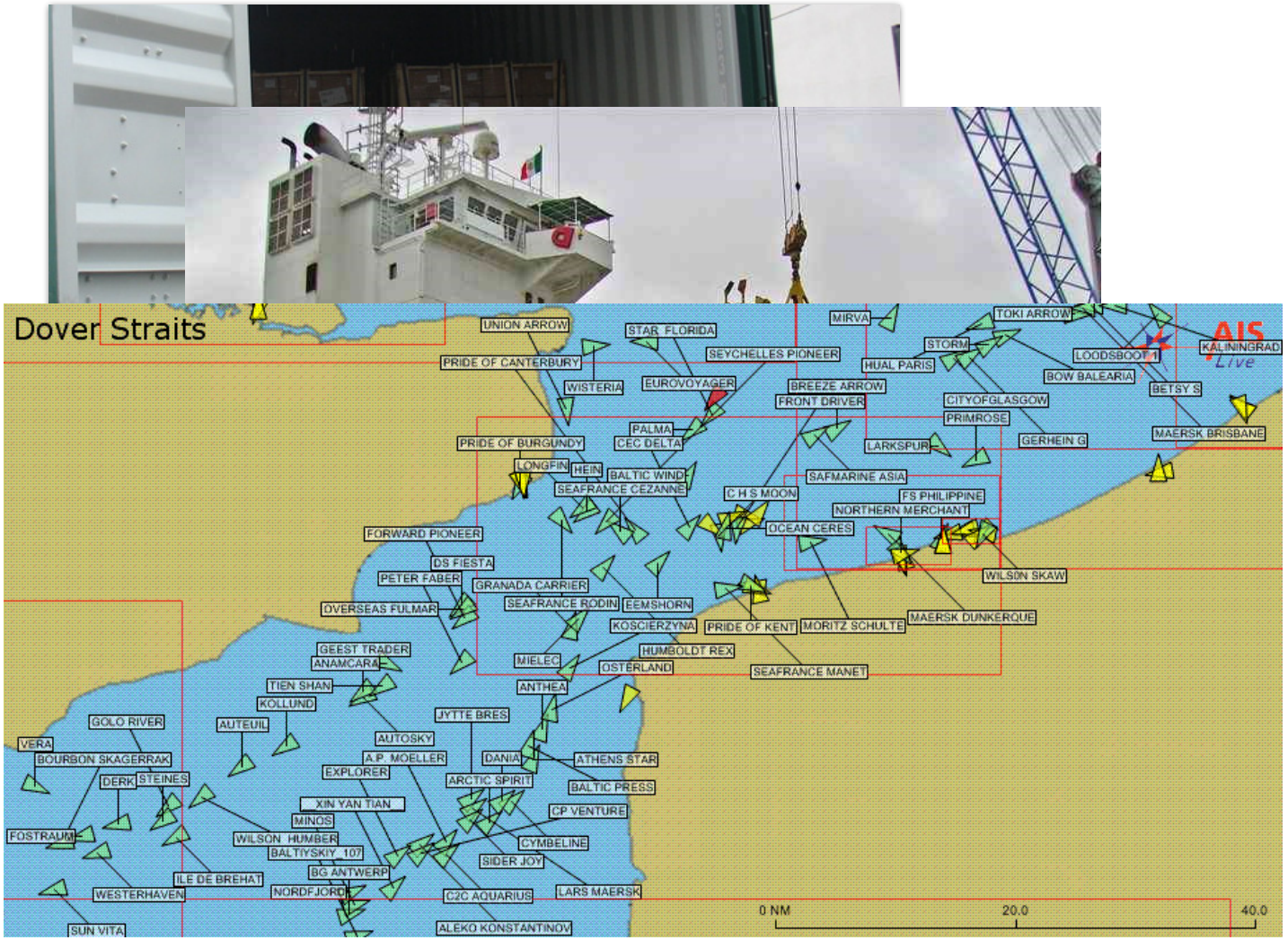
Architecting for Failure in a Containerized World

Tom Faulhaber
Infolace


长车
沪D6495挂
长车
沪D
6495挂


ZIM


Dover Straits
UNION ARROW
STAR FLORIDA
SEYCHELLES PIONEER
PRIDE OF CANTERBURY
WISTERIA
EUROVOYAGER
MIRVA
TOKI ARROW
STORM
HUAL PARIS
LOODSBOOT 1
KALININGRAD
BOW BALEARIA
BETSY S
BREEZE ARROW
FRONT DRIVER
CITYOFGLASGOW
PRIMROSE
PALMA
CEC DELTA
GERHEIN G
MAERSK BRISBANE
PRIDE OF BURGUNDY
LARKSPUR
LONGFIN
HEIN
BALTIC WIND
SAFMARINE ASIA
SEAFRANCE CEZANNE
C H S MOON
FS PHILIPPINE
NORTHERN MERCHANT
FORWARD PIONEER
OCEAN CERES
DS FIESTA
PETER FABER
GRANADA CARRIER
WILSON SKAW
OVERSEAS FULMAR
SEAFRANCE RODIN
EEMSHORN
KOSCIERZYNA
PRIDE OF KENT
MORITZ SCHULTE
MAERSK DUNKERQUE
GEEST TRADER
ANAMCARA
MIELEC
HUMBOLDT REX
OSTERLAND
SEAFRANCE MANET
TIEN SHAN
ANTHEA
KOLLUND
GOLO RIVER
JYTTE BRES
AUTEUIL
VERA
AUTOSKY
BOURBON SKAGERRAK
A.P. MOELLER
DANIA
ATHENS STAR
DERK
STEINES
EXPLORER
ARCTIC SPIRIT
BALTIC PRESS
XIN YAN TIAN
CP VENTURE
MINOS
FOSTRAUM
WILSON HUMBER
CYMBELINE
BALTIYSKIY 107
SIDER JOY
BG ANTWERP
ILE DE BREHAT
NORDFJORD
C2C AQUARIUS
LARS MAERSK
WESTERHAVEN
ALEKO KONSTANTINOV
SUN VITA
0 NM
20.0
40.0
AIS Live

How can container tech help us build robust systems?

Key takeaway: an architectural toolkit for building robust systems with containers

The Rules

Decomposition

Orchestration and Synchronization

Managing Stateful Apps

Simplicity

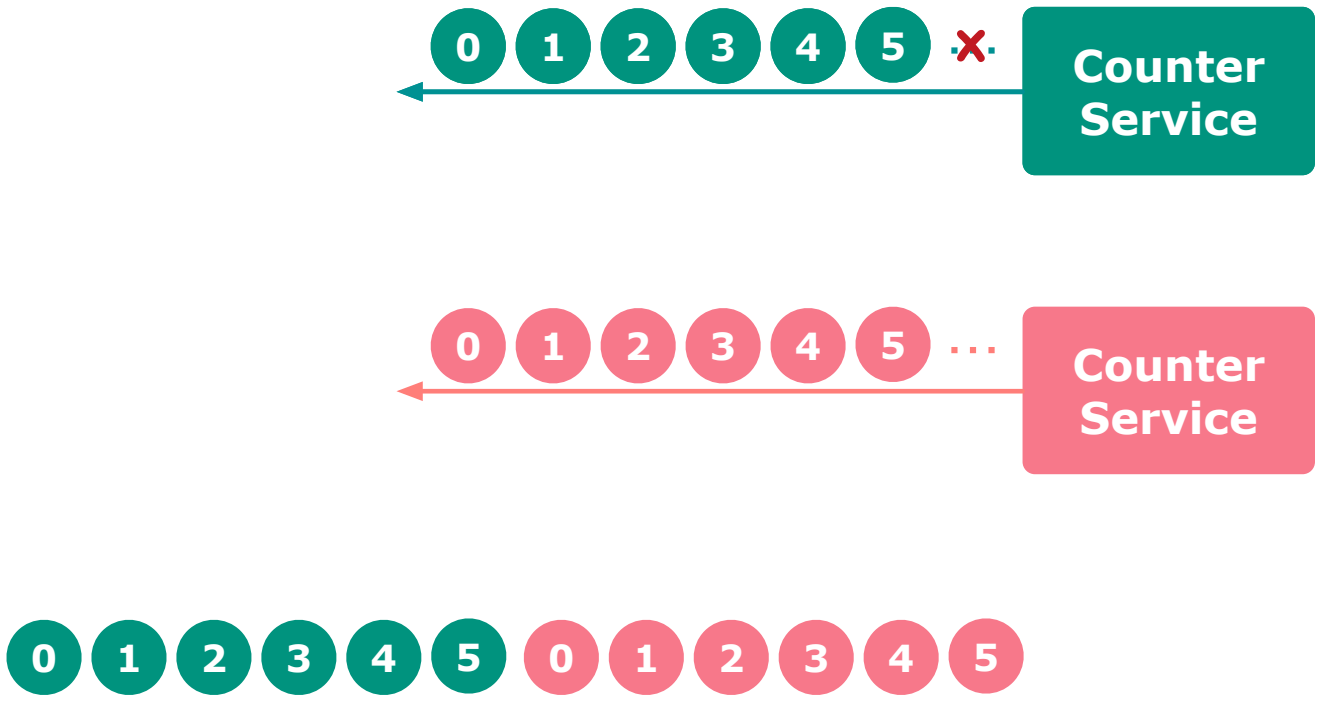
Simple means:
“Do one thing!”

The opposite of
simple is **complex**

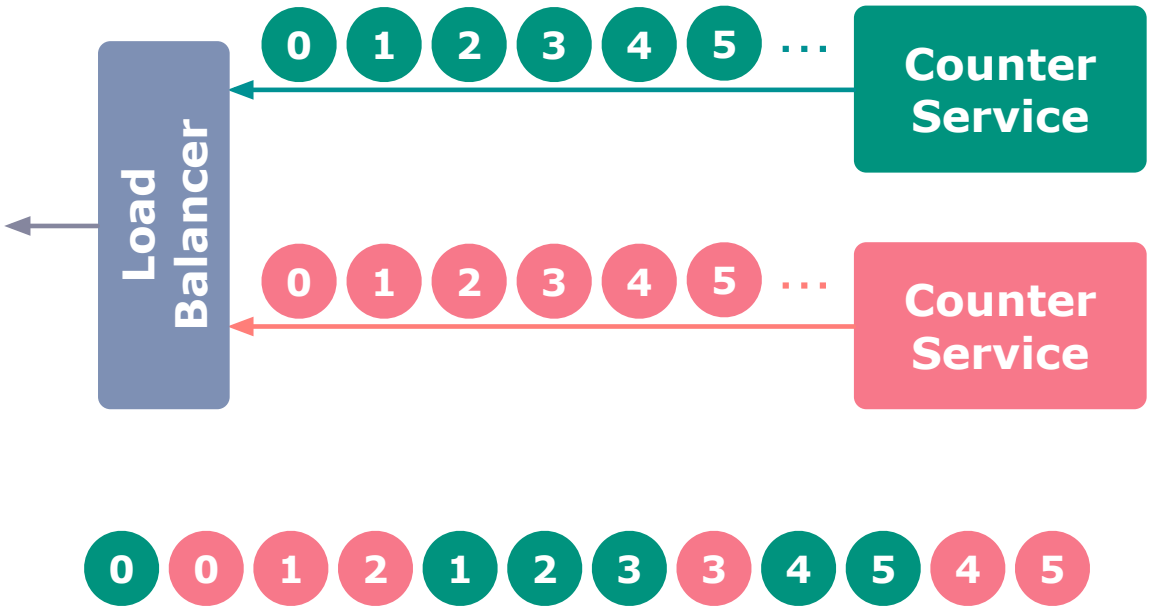
Complexity exists
within
components

Complexity exists
between
components

Example: a counter



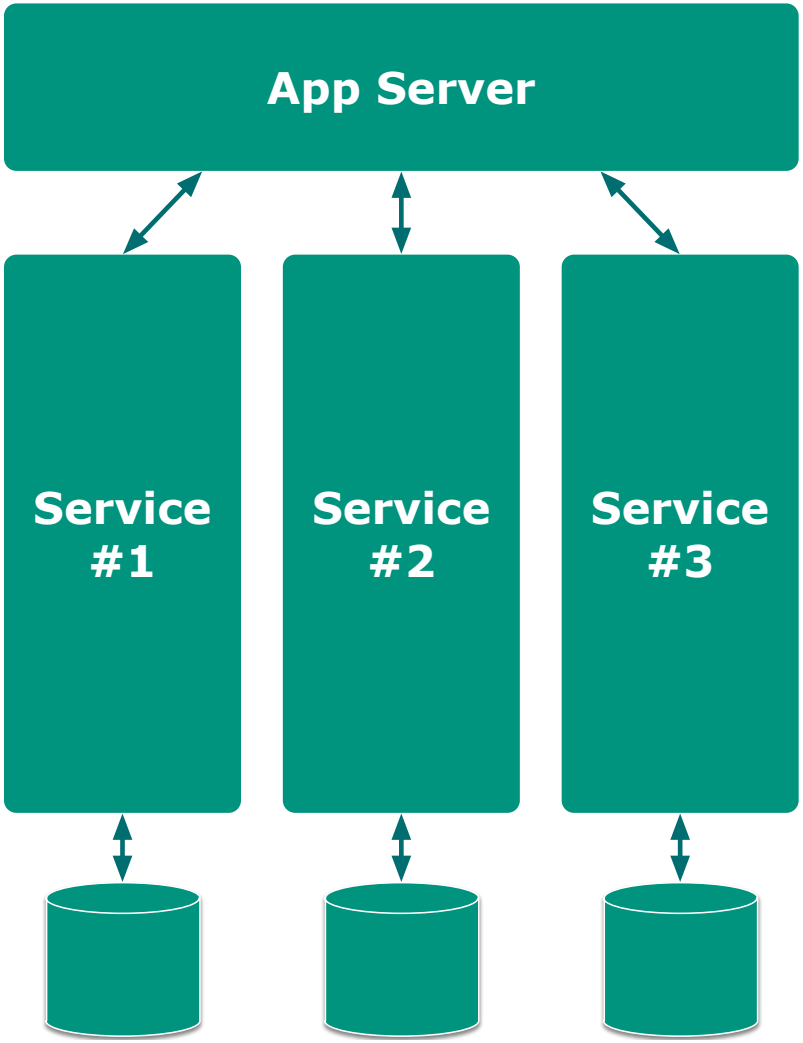
Example: a counter

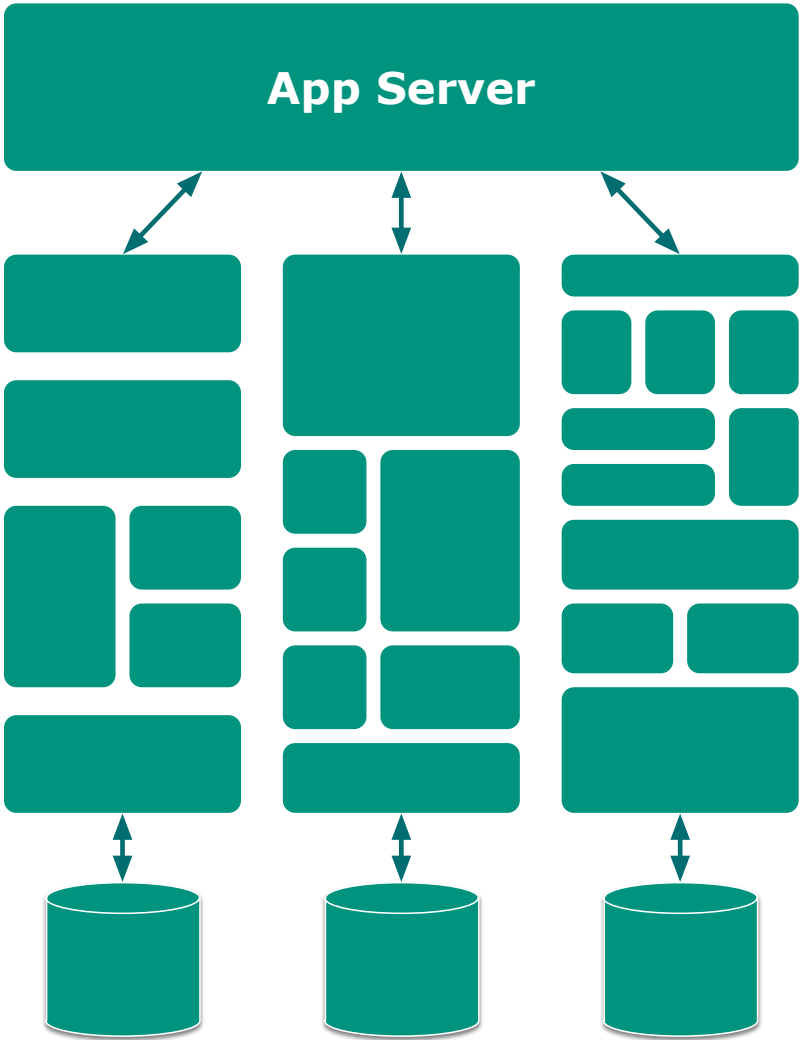


State + composition = complexity

Part 1:
Decomposition

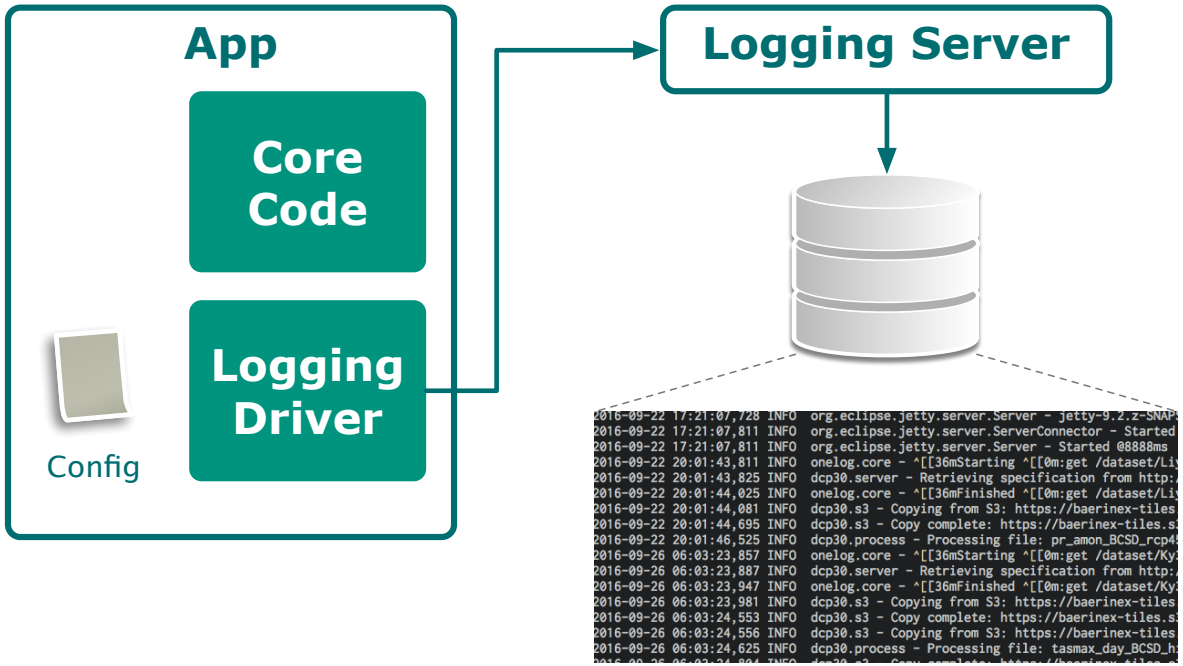
Rule:
Decompose vertically


App Server
Service #1
Service #2
Service #3

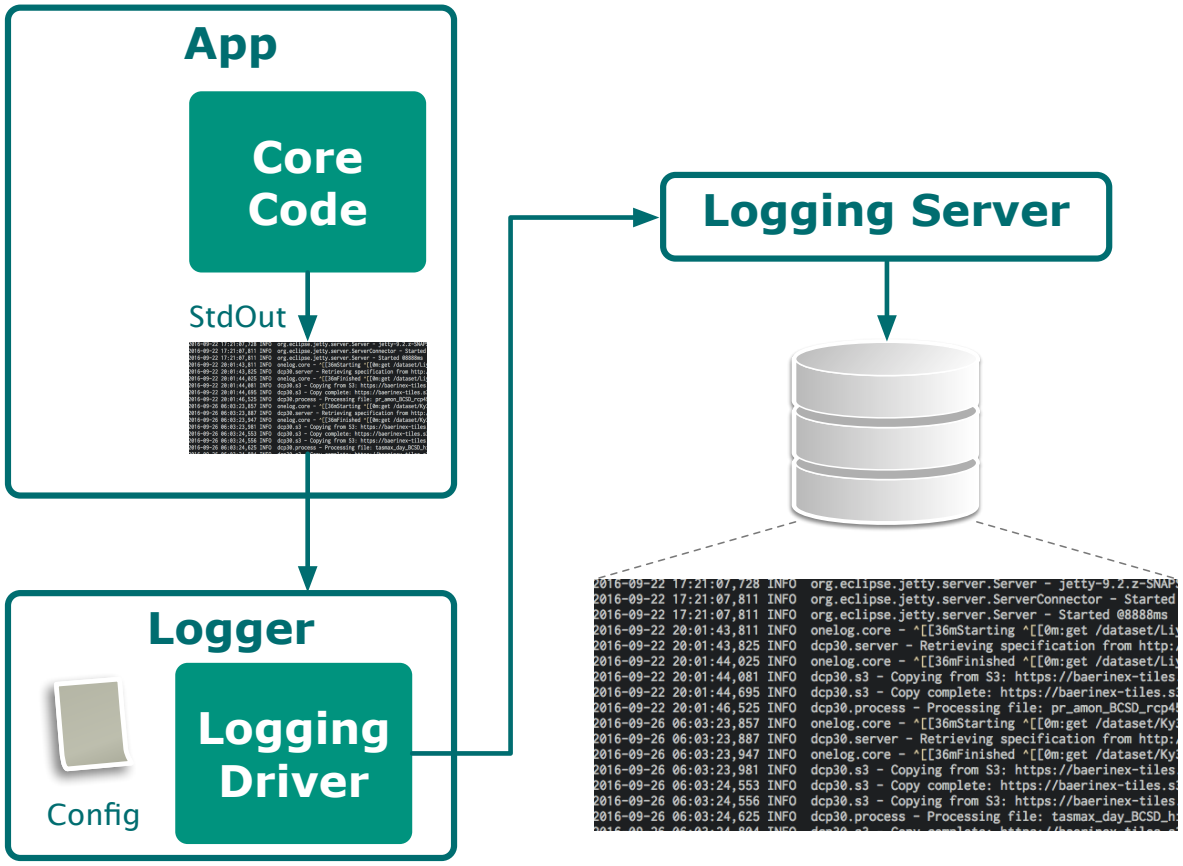

App Server

Rule:
Separation of concerns

Example: Logging

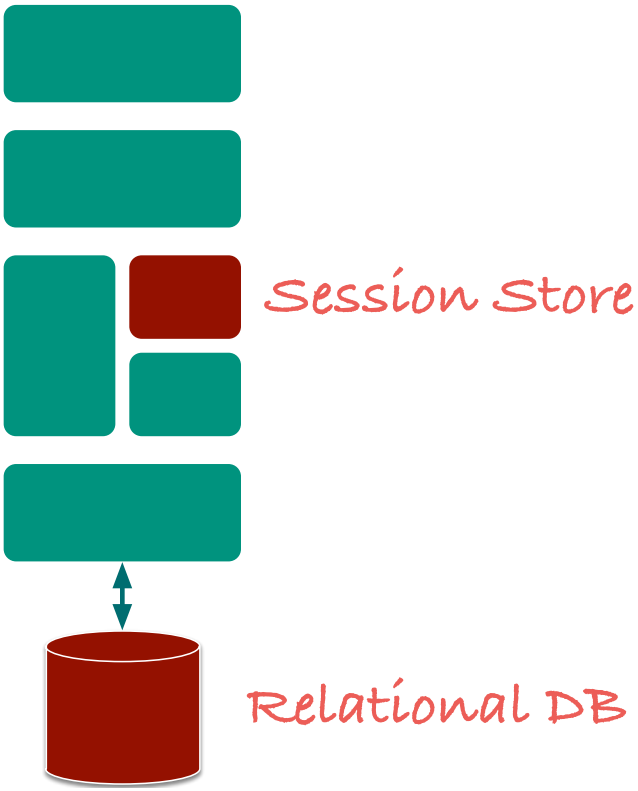


Example: Logging

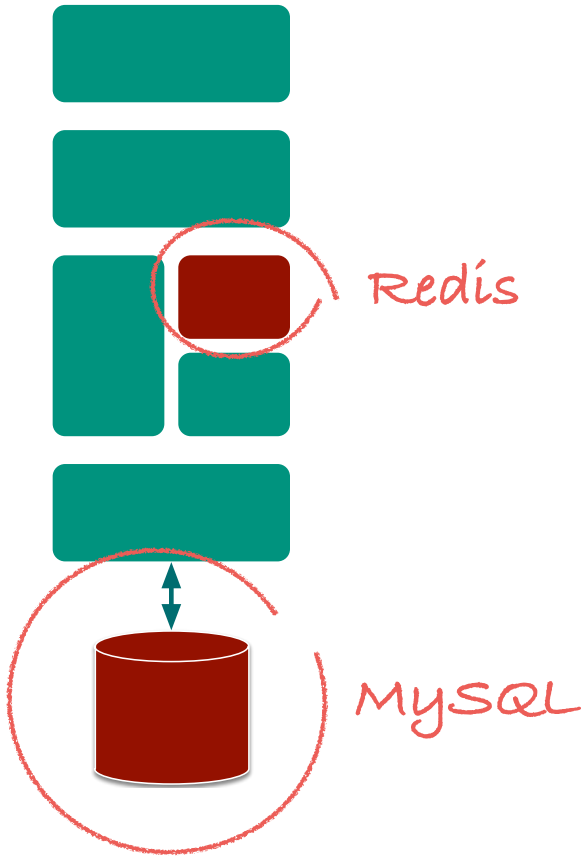


Aspect-oriented programming

Rule:
Constrain state


Session Store
Relational DB

Rule:
Battle-tested tools

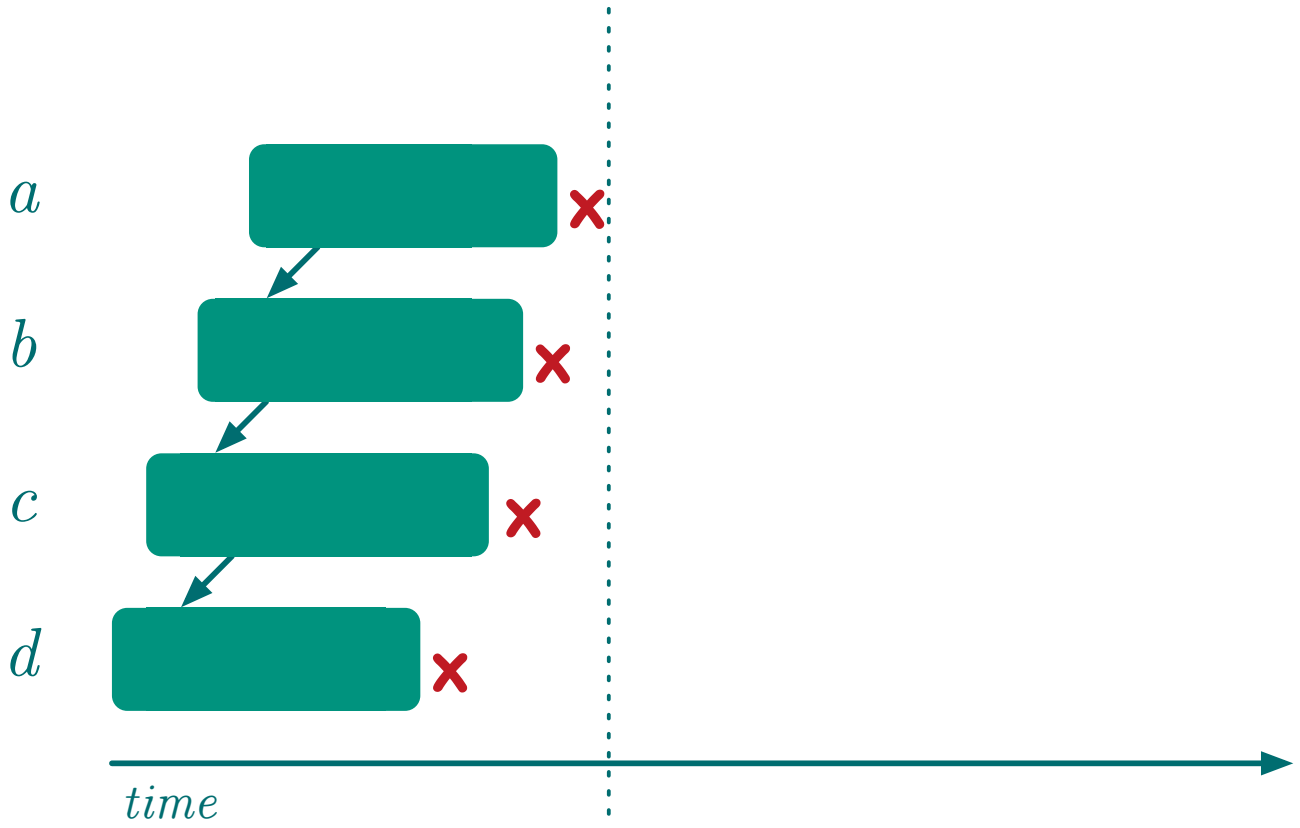

Redis
MySQL

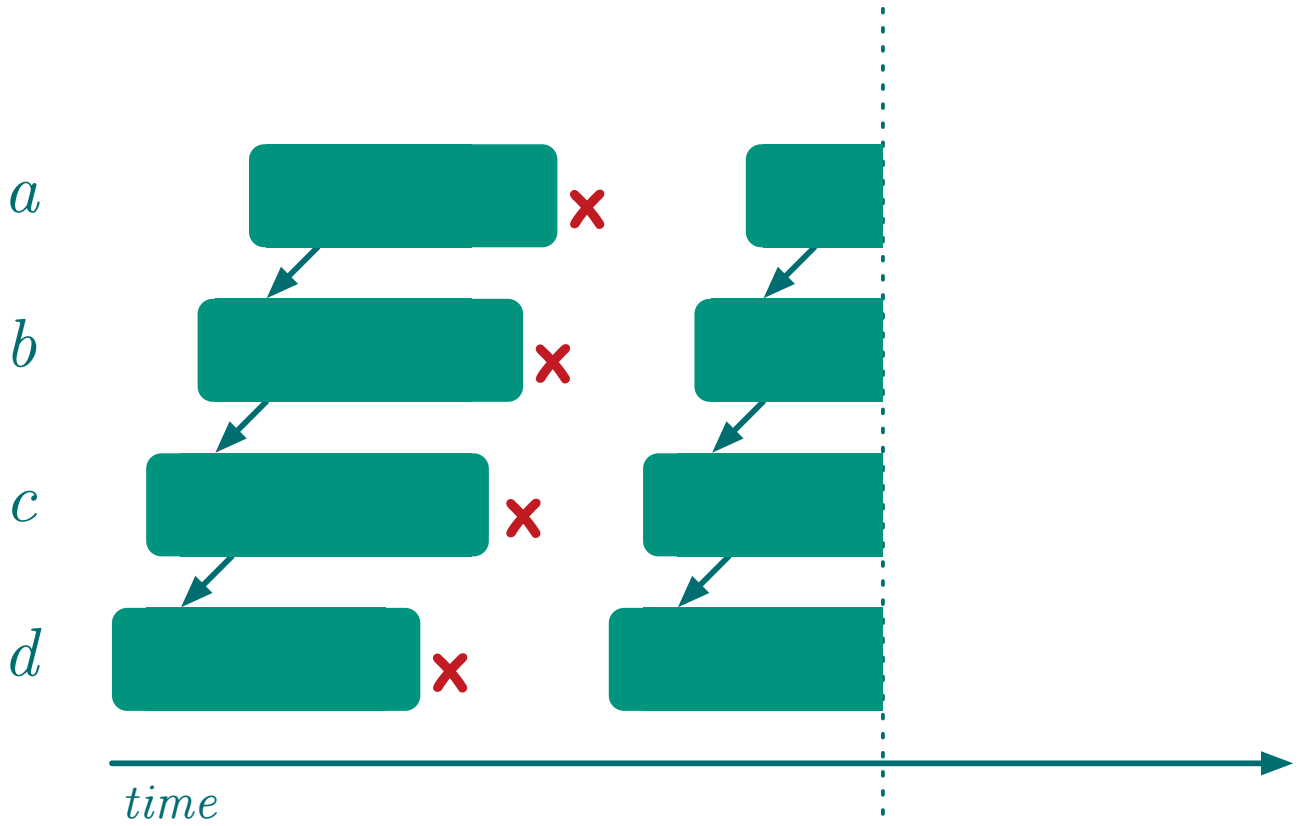
Rule:
High code churn
→Easy restart

Rule:
No start-up order!

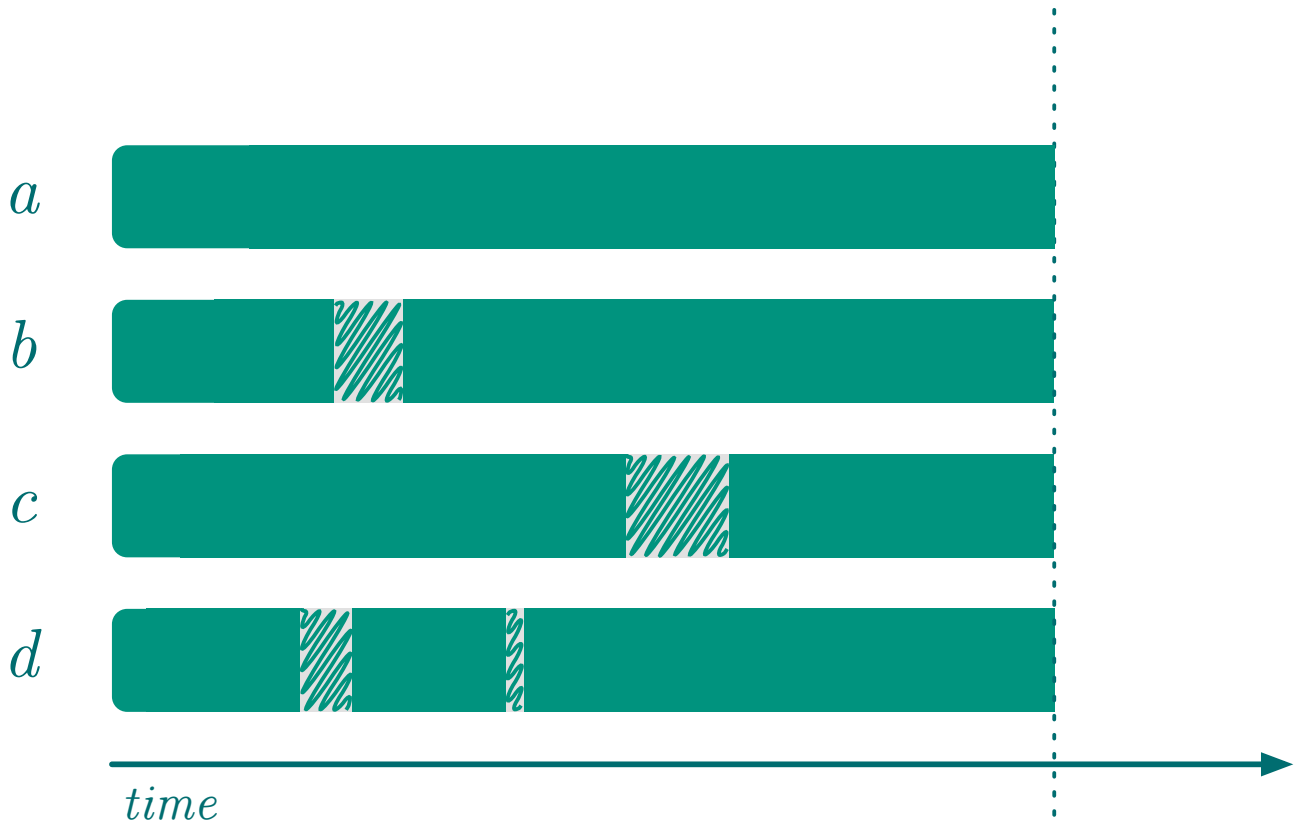

a
b
c
d
time

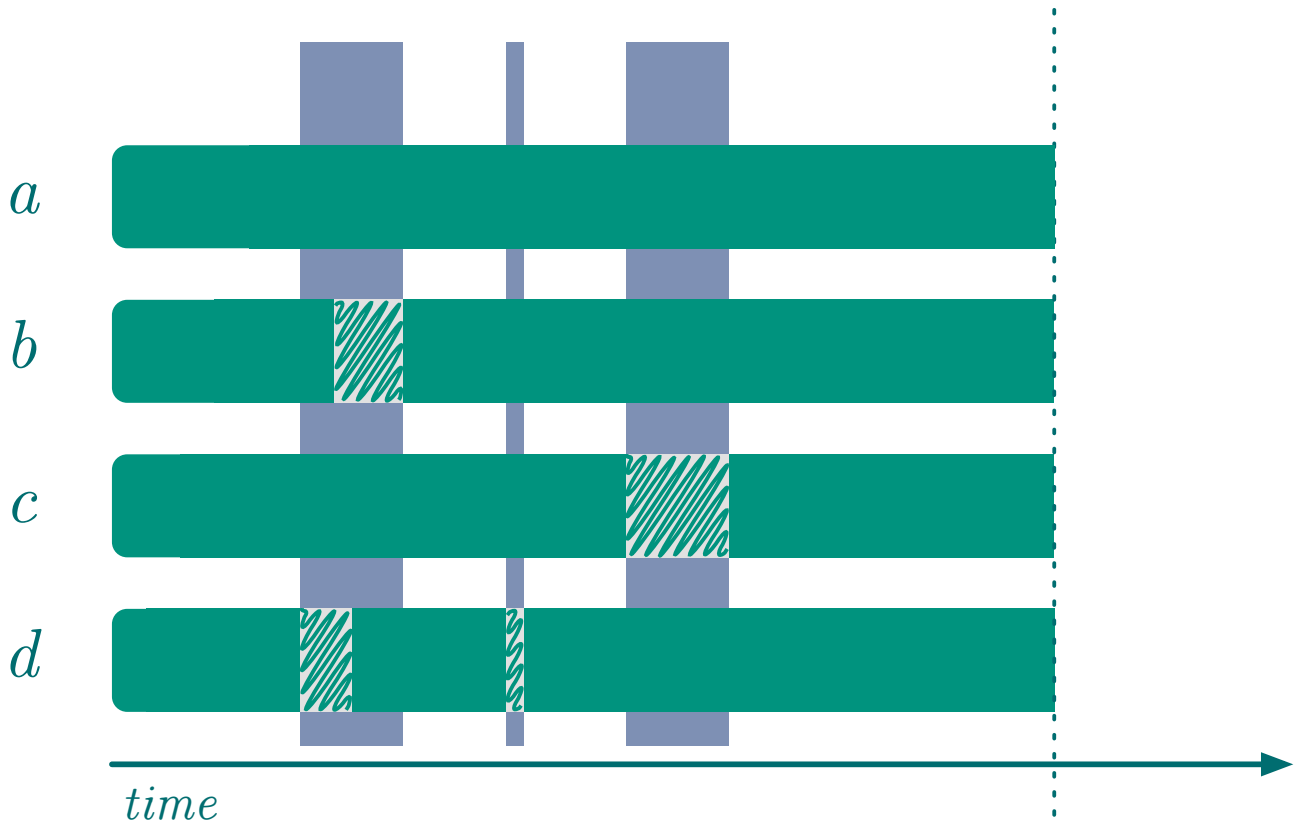

a
b
c
d
time


a
b
c
d
time


a
b
c
d
time


a
b
c
d
time


a
b
c
d
time


a
b
c
d
time

Rule:
Consider higher-order failure

The Rules

Decomposition

Decompose vertically
Separation of concerns
Constrain state
Battle-tested tools
High code churn, easy restart
No start-up order!
Consider higher-order failure

Orchestration and Synchronization

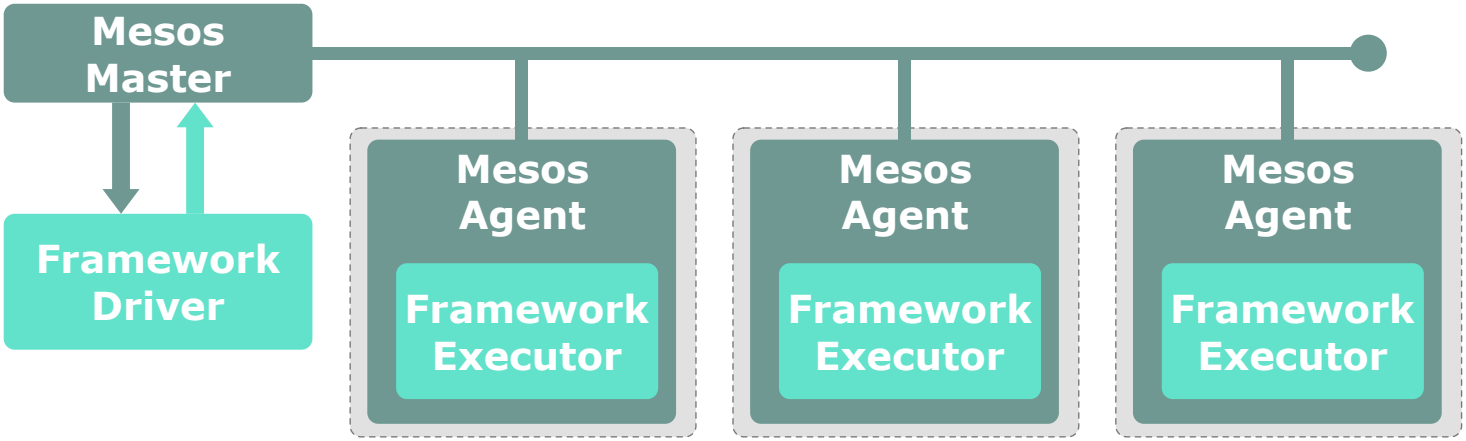
Managing Stateful Apps

Part 2:
Orchestration and Synchronization

Rule:
Use Framework Restarts

- Mesos: Marathon always restarts
- Kubernetes: `RestartPolicy=Always`
- Docker: Swarm always restarts

Rule:
Create your own framework


Mesos
Master
Framework
Driver
Mesos
Agent
Framework
Executor
Mesos
Agent
Framework
Executor
Mesos
Agent
Framework
Executor

Rule:

Use

Synchronized State

Synchronized State

Tools:

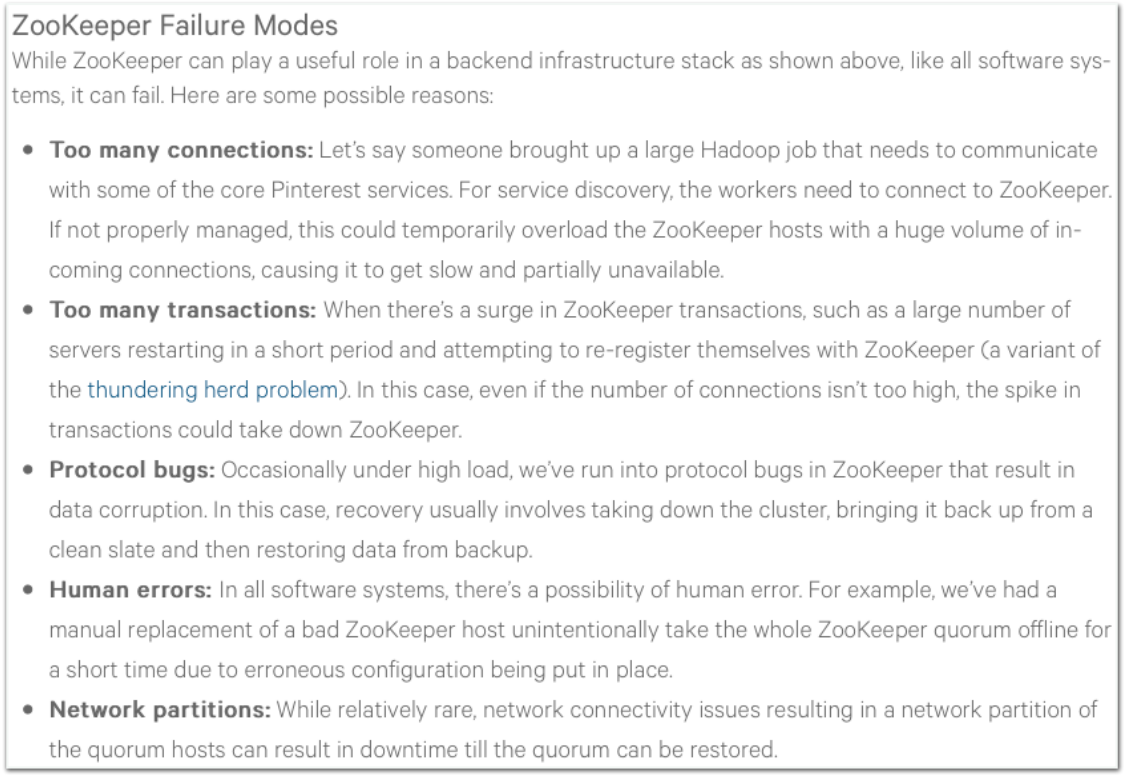
- zookeeper
- etcd
- consul

Patterns:

- leader election
- shared counters
- peer awareness
- work partitioning

Rule:
Minimize
Synchronized State

Even battle-tested state management is a headache.



ZooKeeper Failure Modes

While ZooKeeper can play a useful role in a backend infrastructure stack as shown above, like all software systems, it can fail. Here are some possible reasons:

- **Too many connections:** Let's say someone brought up a large Hadoop job that needs to communicate with some of the core Pinterest services. For service discovery, the workers need to connect to ZooKeeper. If not properly managed, this could temporarily overload the ZooKeeper hosts with a huge volume of incoming connections, causing it to get slow and partially unavailable.
- **Too many transactions:** When there's a surge in ZooKeeper transactions, such as a large number of servers restarting in a short period and attempting to re-register themselves with ZooKeeper (a variant of the thundering herd problem). In this case, even if the number of connections isn't too high, the spike in transactions could take down ZooKeeper.
- **Protocol bugs:** Occasionally under high load, we've run into protocol bugs in ZooKeeper that result in data corruption. In this case, recovery usually involves taking down the cluster, bringing it back up from a clean slate and then restoring data from backup.
- **Human errors:** In all software systems, there's a possibility of human error. For example, we've had a manual replacement of a bad ZooKeeper host unintentionally take the whole ZooKeeper quorum offline for a short time due to erroneous configuration being put in place.
- **Network partitions:** While relatively rare, network connectivity issues resulting in a network partition of the quorum hosts can result in downtime till the quorum can be restored.

(Source: http://blog.cloudera.com/blog/2014/03/zookeeper-resilience-at-pinterest/)

The Rules

Decomposition

Decompose vertically
Separation of concerns
Constrain state
Battle-tested tools
High code churn, easy restart
No start-up order!
Consider higher-order failure

Orchestration and Synchronization

Use framework restarts
Create your own framework
Use synchronized state
Minimize synchronized state

Managing Stateful Apps

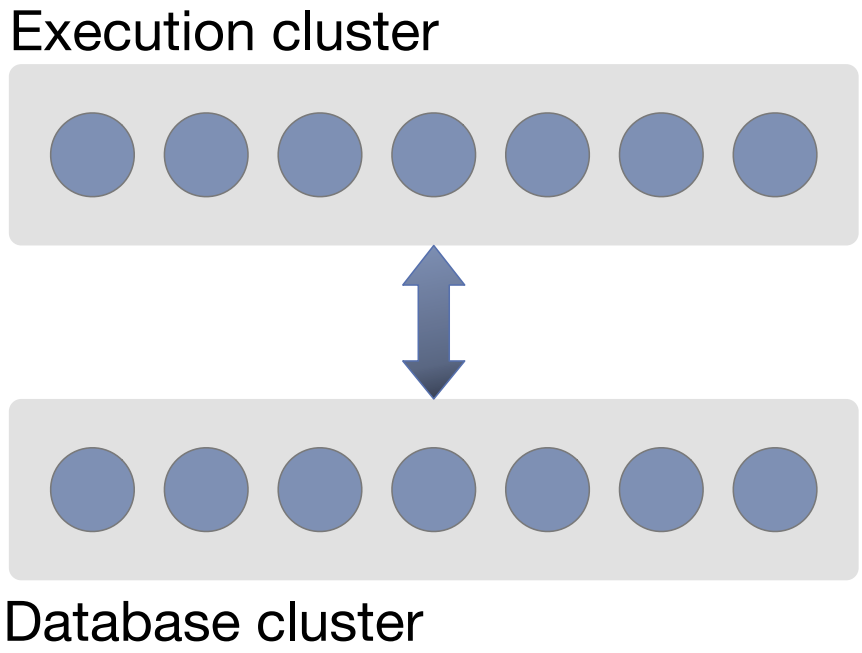
Part 3:
Managing Stateful Apps

Rule (repeat!):
Always use battle-tested tools!

(State is the weak point)

Rule:
Choose the DB architecture

Option 1: External DB



Option 1: External DB

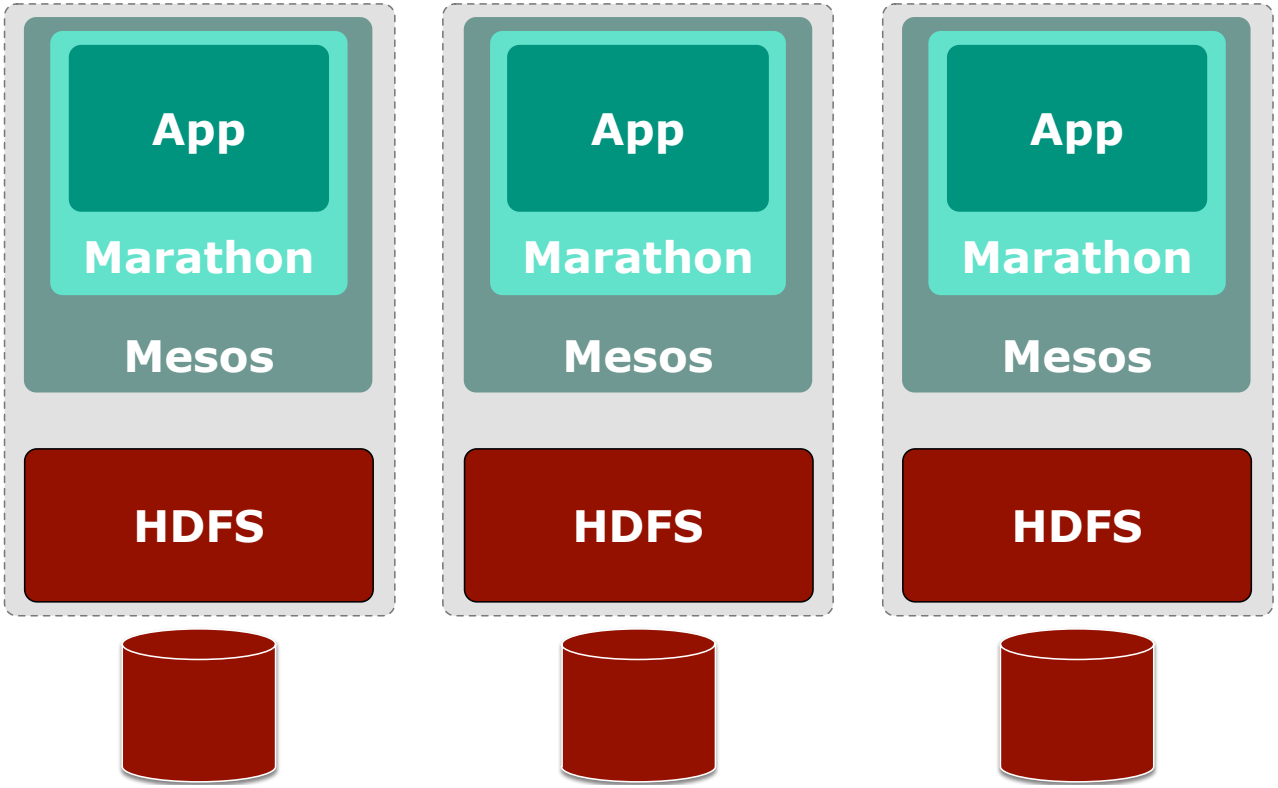
Pros

- Somebody else's problem!
- Can use a DB designed for clustering directly
- Can use DB as a service

Cons

- Not really somebody else's problem!
- Higher latency/no reference locality
- Can't leverage orchestration, etc.

Option 2: Run on Raw HW



Option 2: Run on Raw HW

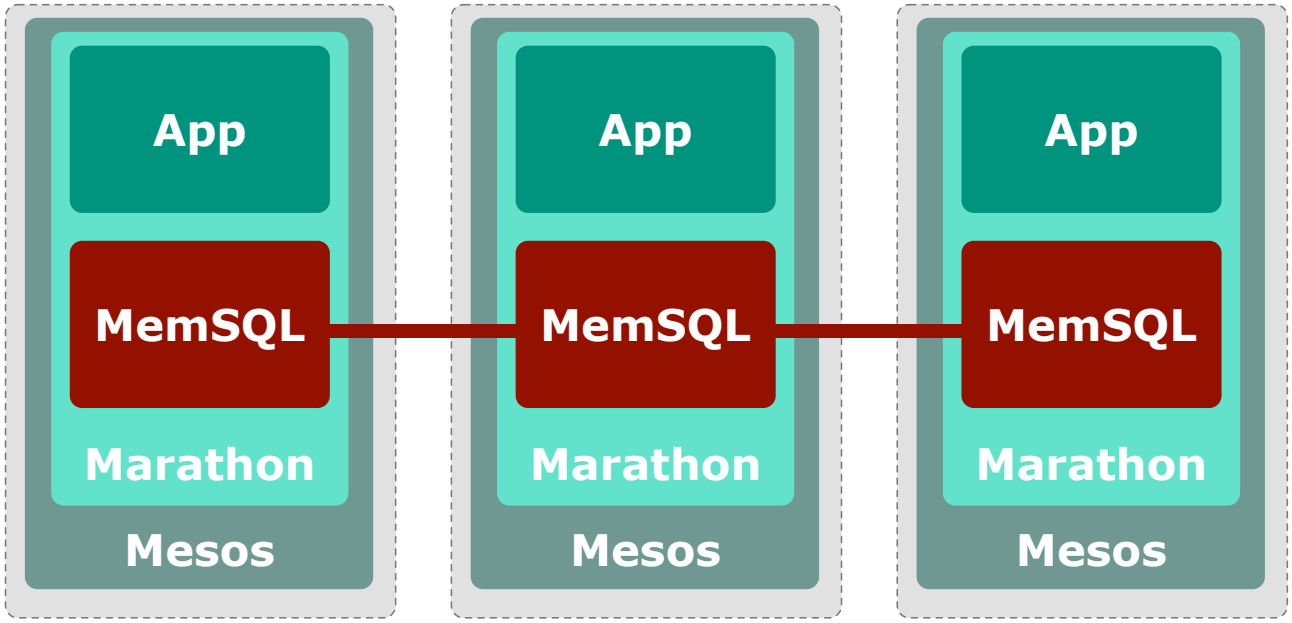
Pros

- Use existing recipes
- Have local data
- Manage a single cluster

Cons

- Orchestration doesn't help with failure
- Increased management complexity

Option 3: In-memory DB



Option 3: In-memory DB

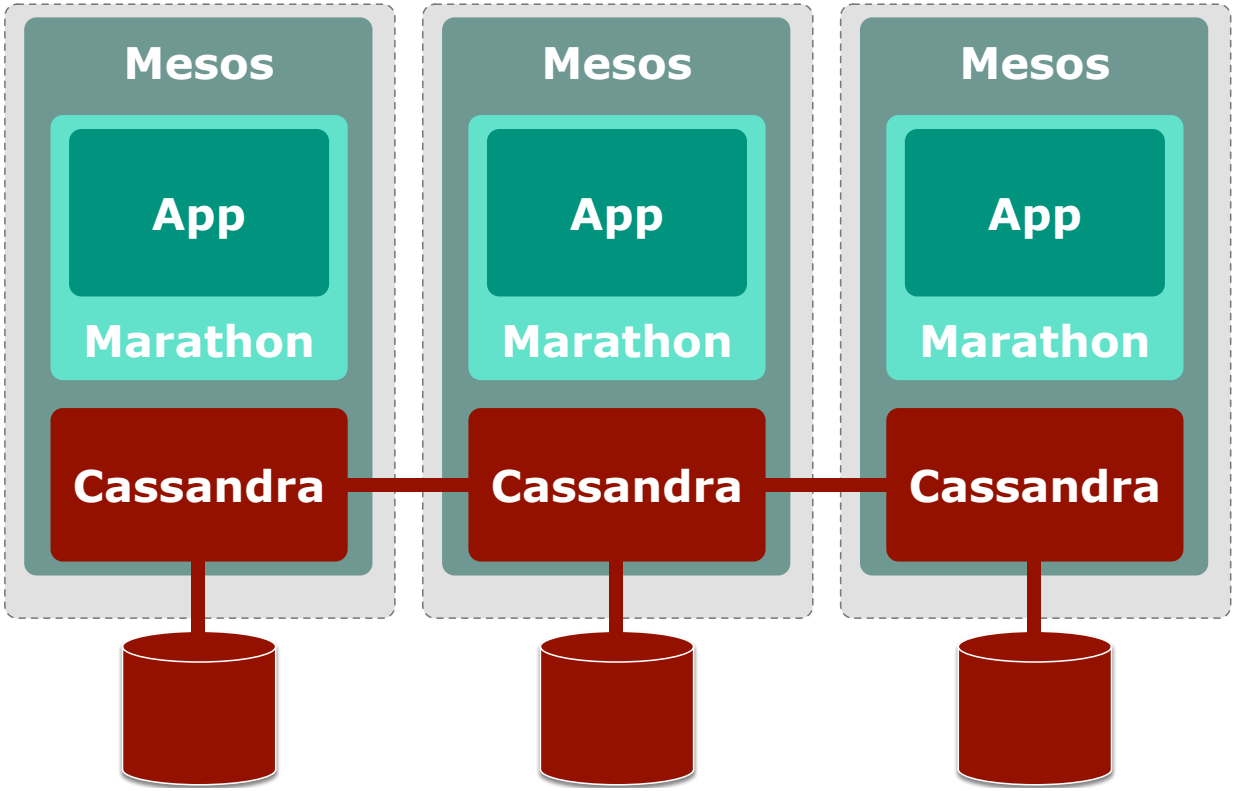
Pros

- No need for volume tracking
- Fast
- Have local data
- Manage a single cluster

Cons

- Bets all machines won't go down
- Bets on orchestration framework

Option 4: Use Orchestration



Option 4: Use Orchestration

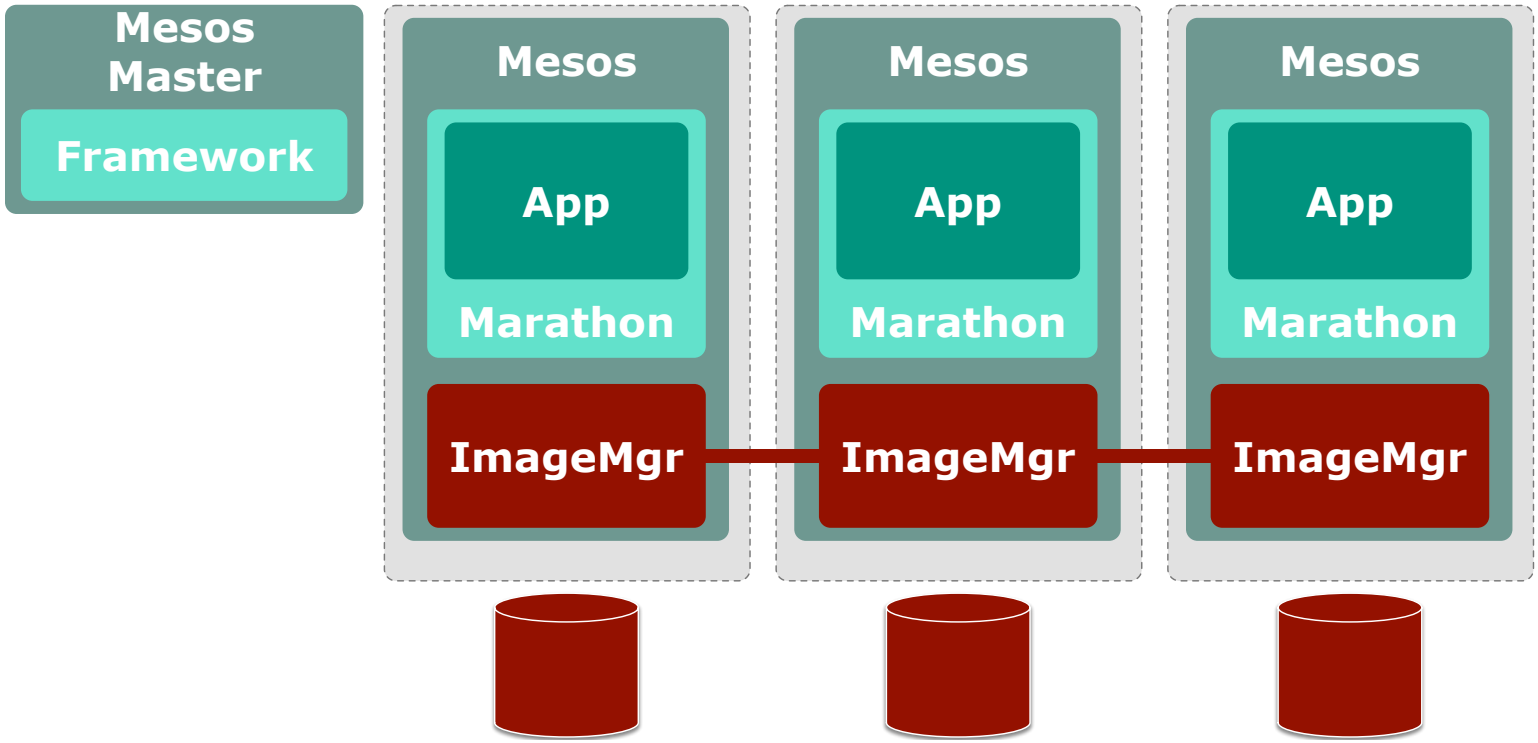
Pros

- Orchestration manages volumes
- One model for all programs
- Have local data
- Single cluster

Cons

- Currently the least mature
- Not well supported by vendors

Option 5: Roll Your Own



Option 5: Roll Your Own

Pros

- Very precise control
- You decide whether to use containers
- Have local data
- Can be system aware

Cons

- You're on your own!
- Wedded to a single orchestration platform
- Not battle tested

Rule:
Have replication

The Rules

Decomposition

Decompose vertically
Separation of concerns
Constrain state
Battle-tested tools
High code churn, easy restart
No start-up order!
Consider higher-order failure

Orchestration and Synchronization

Use framework restarts
Create your own framework
Use synchronized state
Minimize synchronized state

Managing Stateful Apps

Battle-tested tools
Choose the DB architecture
Have replication

Fin

References

- Rich Hickey:
  “Are We There Yet?” (https://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey)
  “Simple Made Easy” (https://www.infoq.com/presentations/Simple-Made-Easy-QCon-London-2012)

- David Greenberg, Building Applications on Mesos, O’Reilly, 2016

- Joe Johnston, *et al.*, Docker in Production: Lessons from the Trenches, Bleeding Edge Press, 2015

The Rules

Decomposition

Decompose vertically
Separation of concerns
Constrain state
Battle-tested tools
High code churn, easy restart
No start-up order!
Consider higher-order failure

Orchestration and Synchronization

Use framework restarts
Create your own framework
Use synchronized state
Minimize synchronized state

Managing Stateful Apps

Battle-tested tools
Choose the DB architecture
Have replication