Large-Scale Automated Storage on Kubernetes

Uber |

Matt Schallert @mattschallert

SRE @ Uber

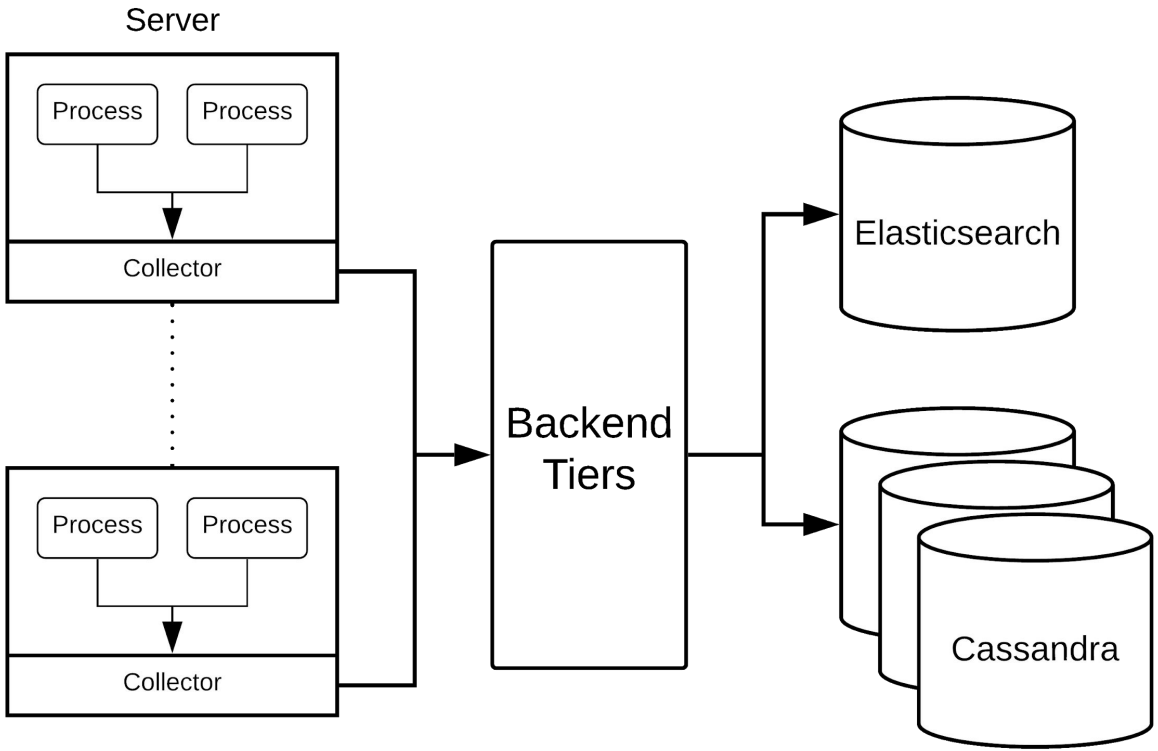
About Me

- SRE @ Uber since 2016
- M3 - Uber’s OSS Metrics Platform

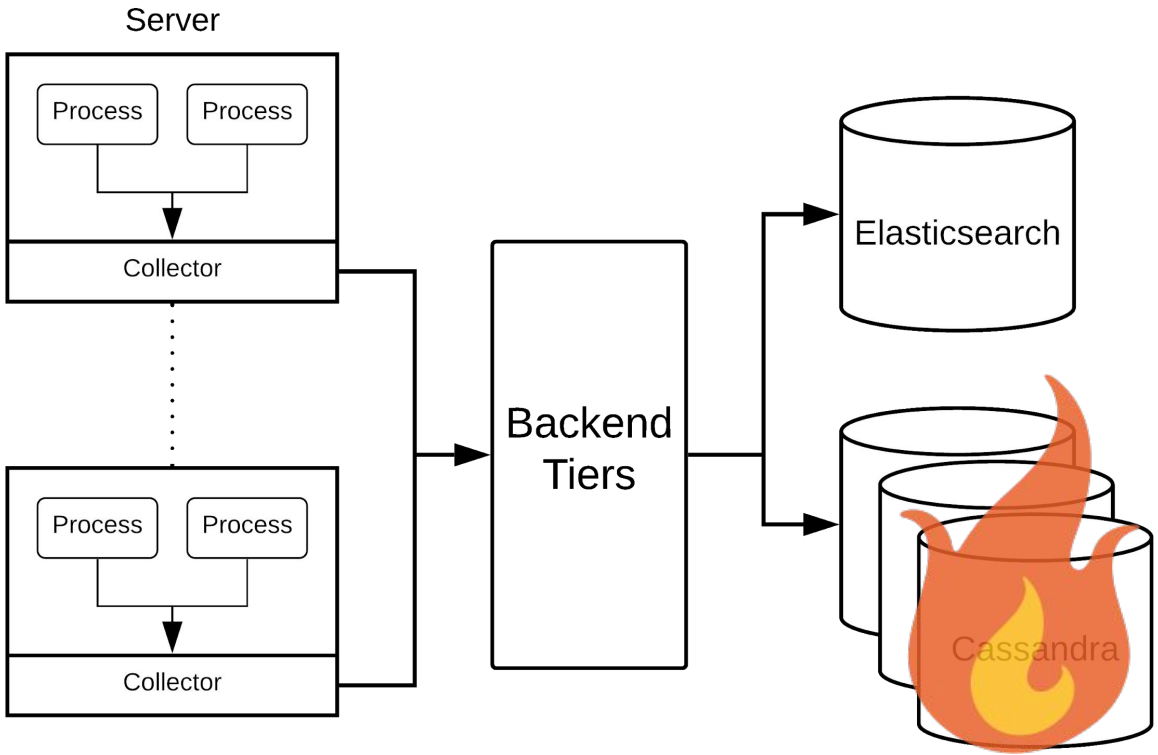
Agenda

- Background: Metrics @ Uber
- Scaling Challenges
- Automation
- M3DB on Kubernetes

2015



Mid 2016



Mid 2016

- Cassandra stack scaled to needs but was expensive
  - Human cost: operational overhead
  - Infra cost: lots of high-IO servers
- Scaling challenges inevitable, needed to reduce complexity

M3DB

- Open source distributed time series database
- Highly compressed (11x), fast, compaction-free storage
- Sharded, replicated (zone & rack-aware)
- Production late 2016

M3DB

- Reduced operational overhead
- Runs on commodity hardware with SSDs
- Strongly consistent cluster membership backed by etcd

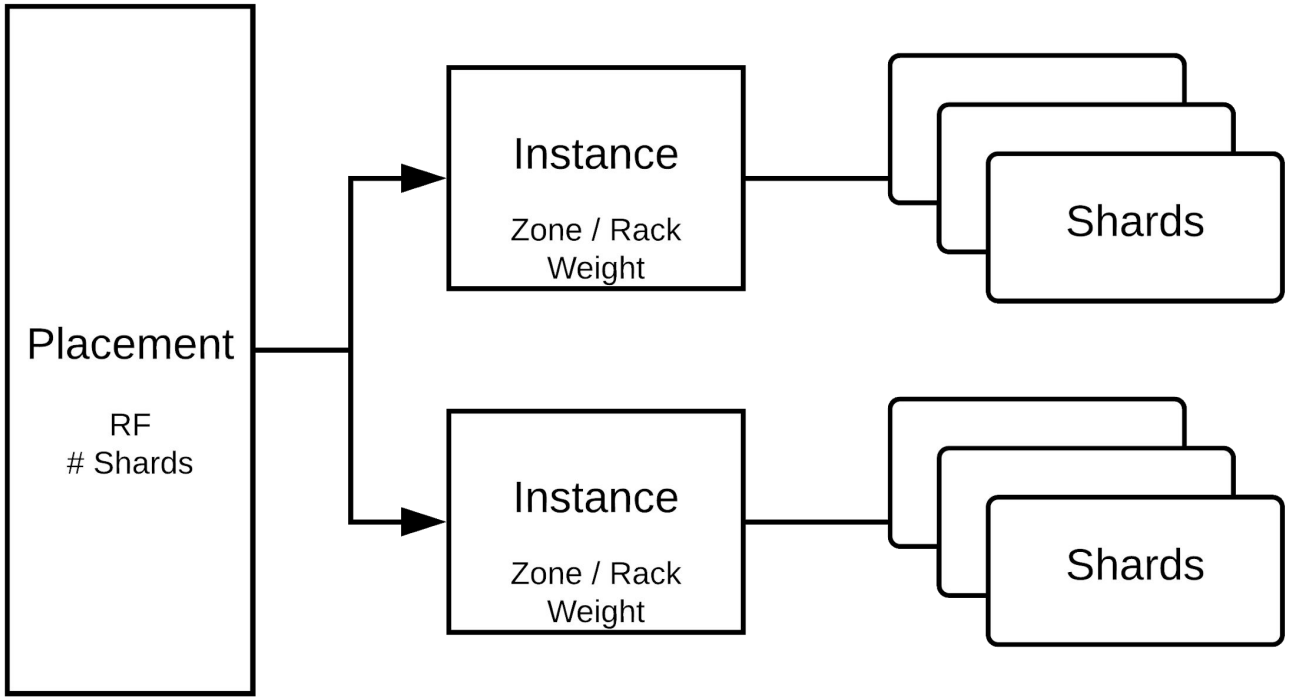
m3cluster

- Reusable cluster management libraries created for M3DB
  - Topology representation, shard distribution algos
  - Service discovery, runtime config, leader election
  - Etcd (strongly consistent KV store) as source of truth for all components
- Similar to Apache Helix
  - Helix is Java-heavy, we’re a Go-based stack

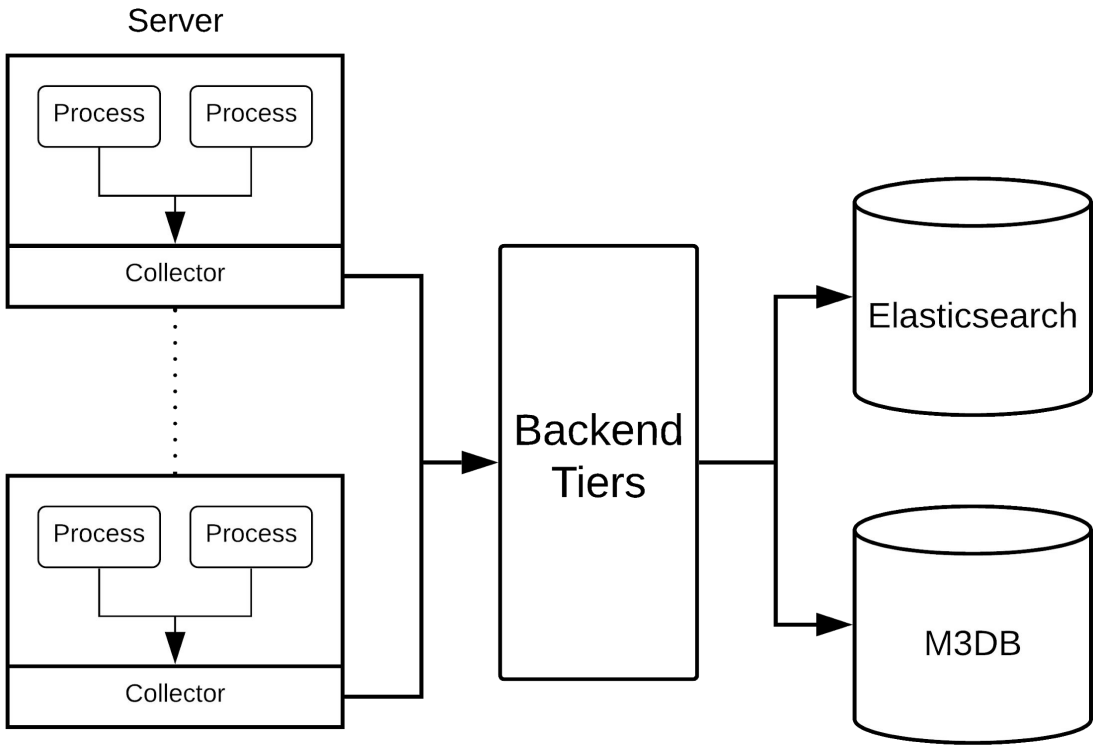
m3cluster

- Topologies represented as desired goal states, M3DB converges

m3cluster: Placement



Late 2016

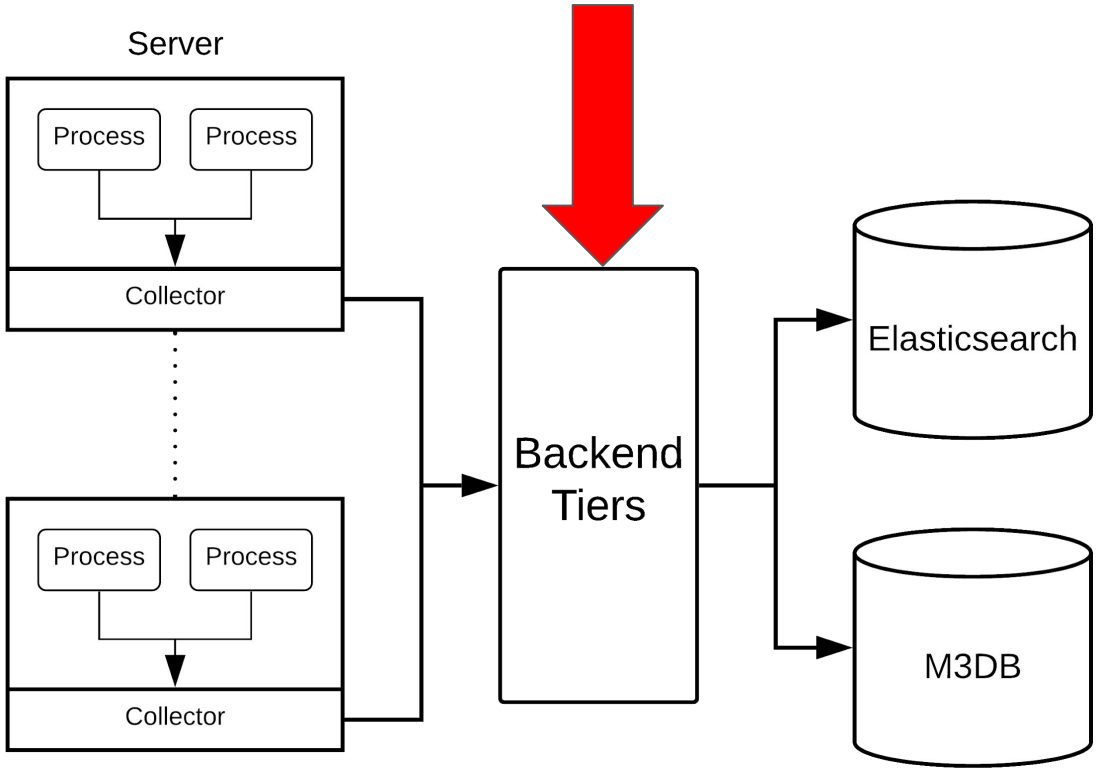


Post-M3DB Deployment

- M3DB solved most pressing issue (Cassandra)
  - Improved stability, reduced costs
- System as a whole: still a high operational overhead

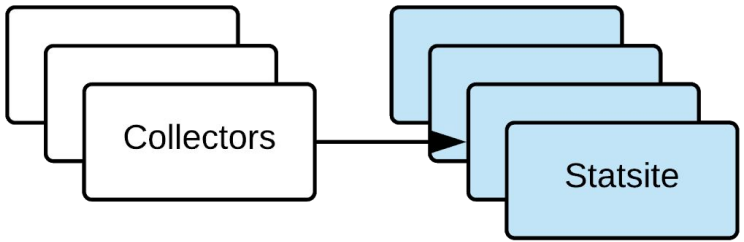
Fragmented representation of components

System Representation (2016)



System Representation (2016)

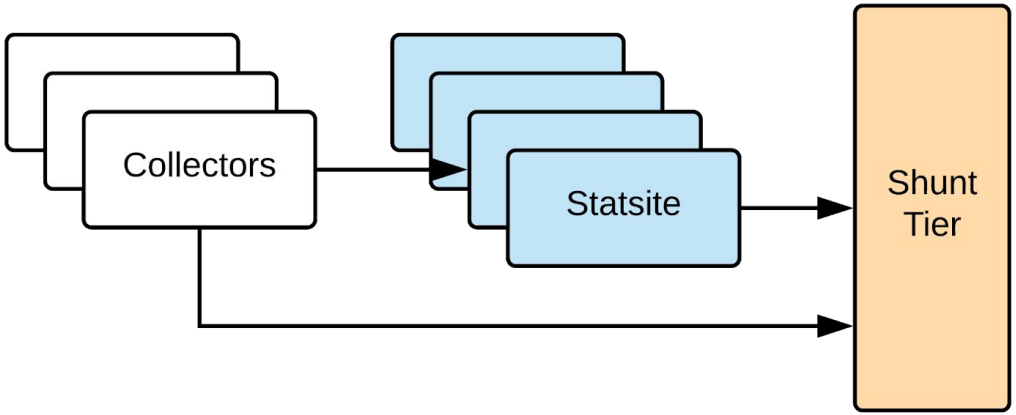
Config service



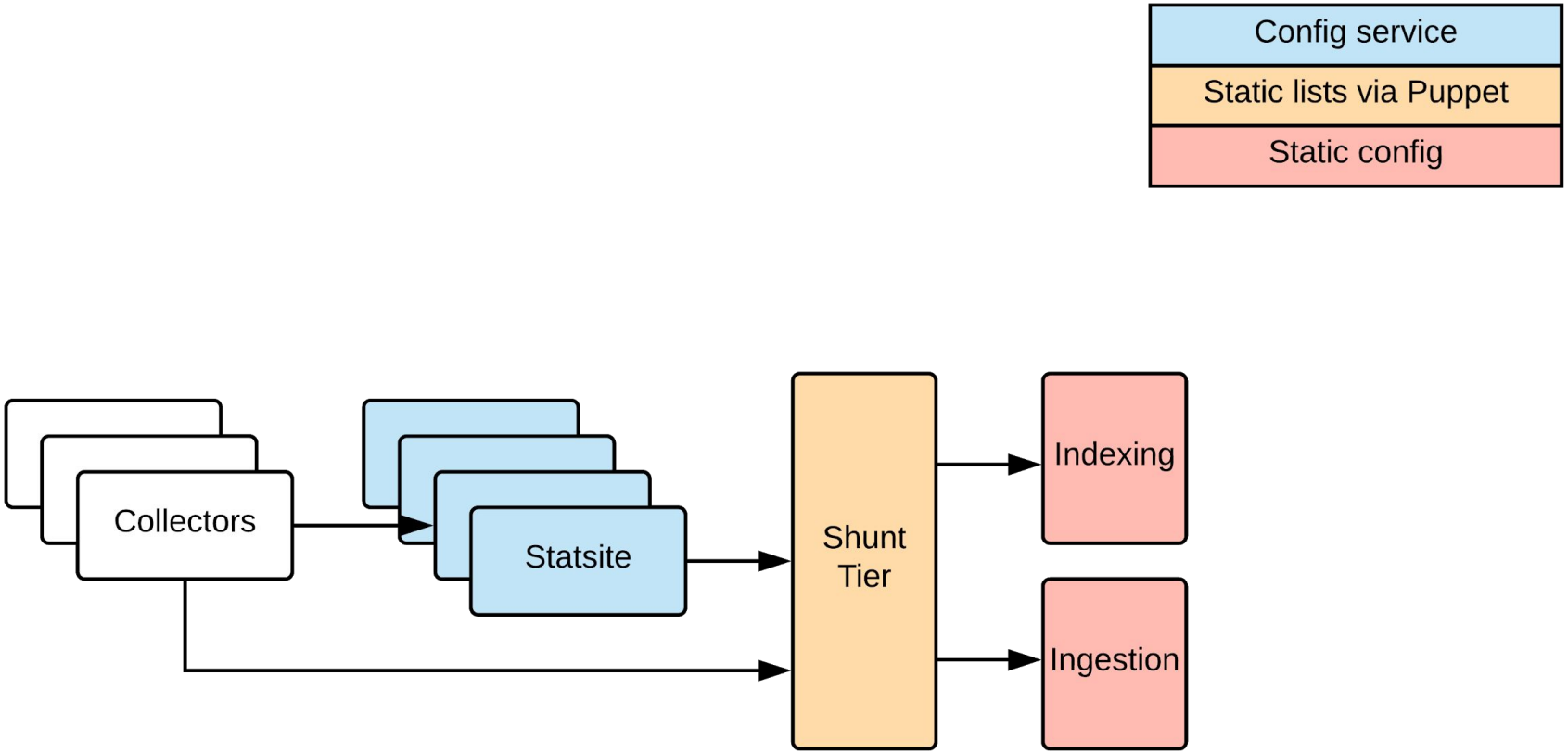
System Representation (2016)

Config service

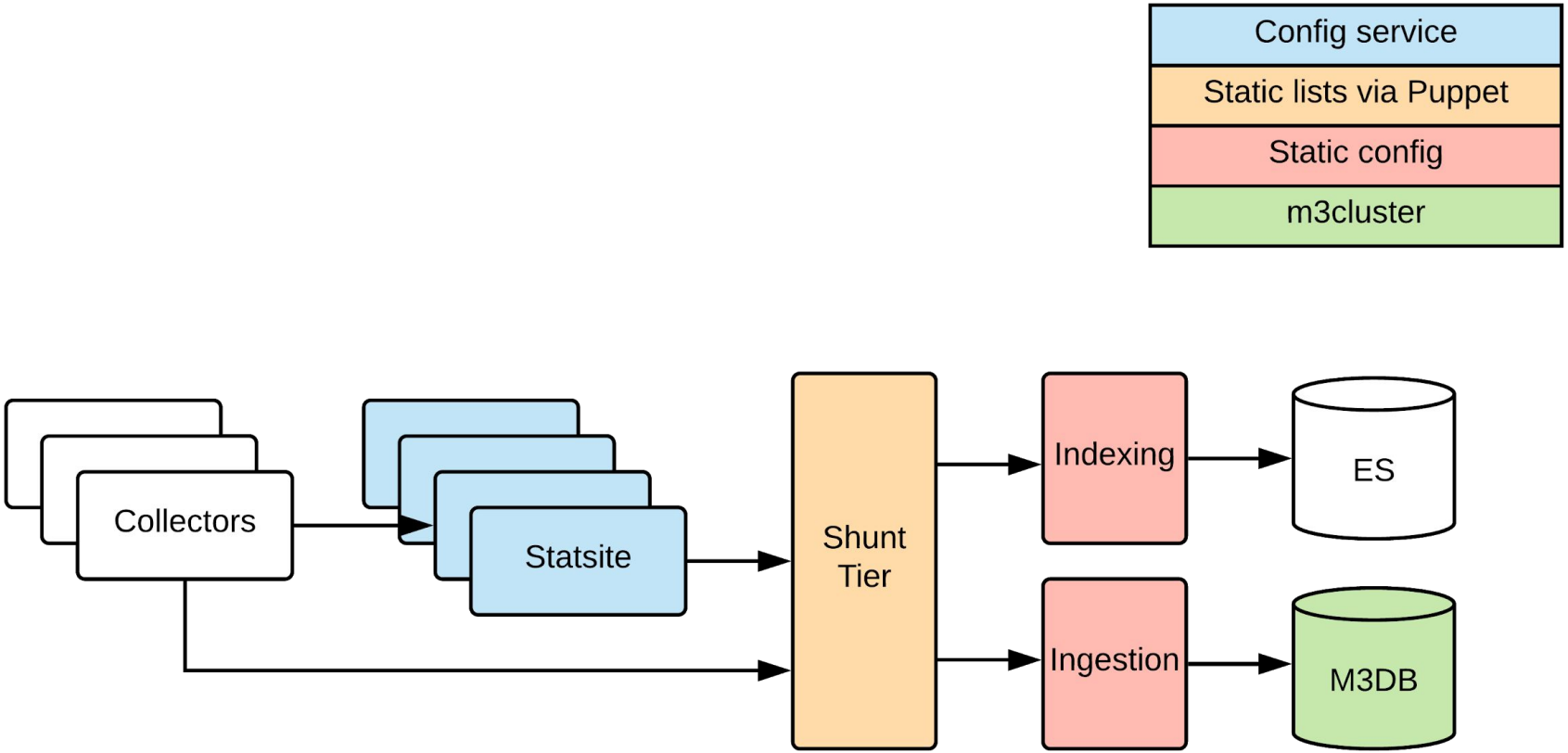
Static lists via Puppet



System Representation (2016)



System Representation (2016)



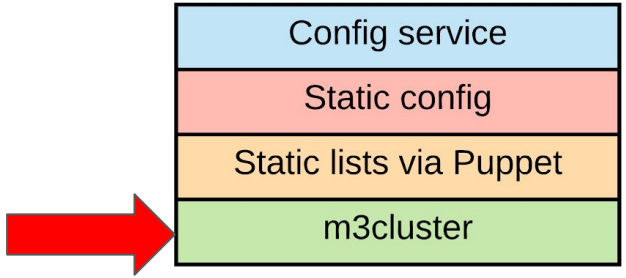
System Representation

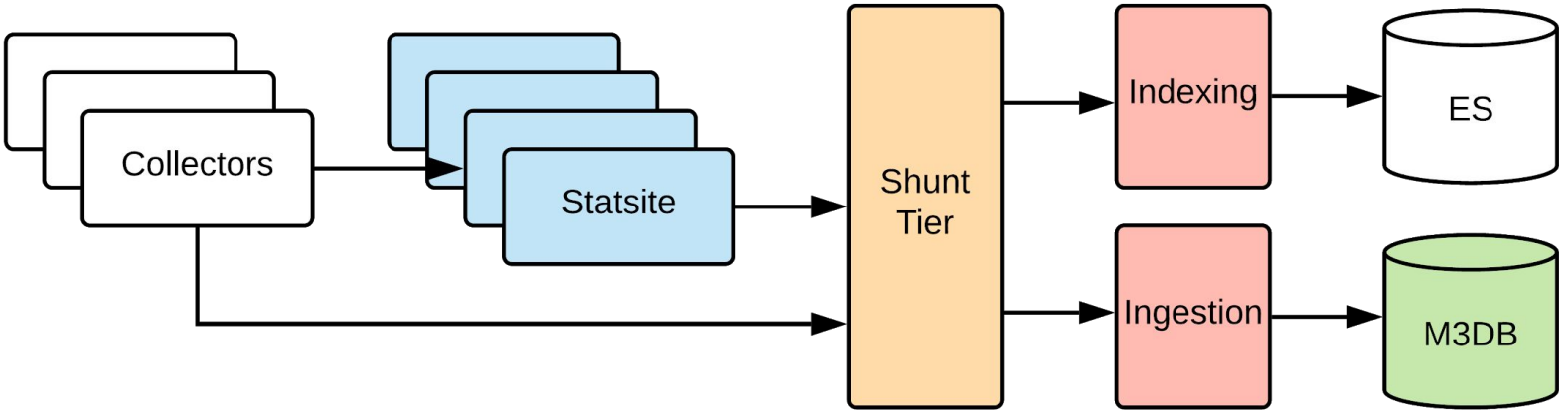
- Systems represented as…
  - Lists fetched from Uber config service
  - Static lists deployed via Puppet
  - Static lists coupled with service deploys
  - m3cluster placements

Fragmentation

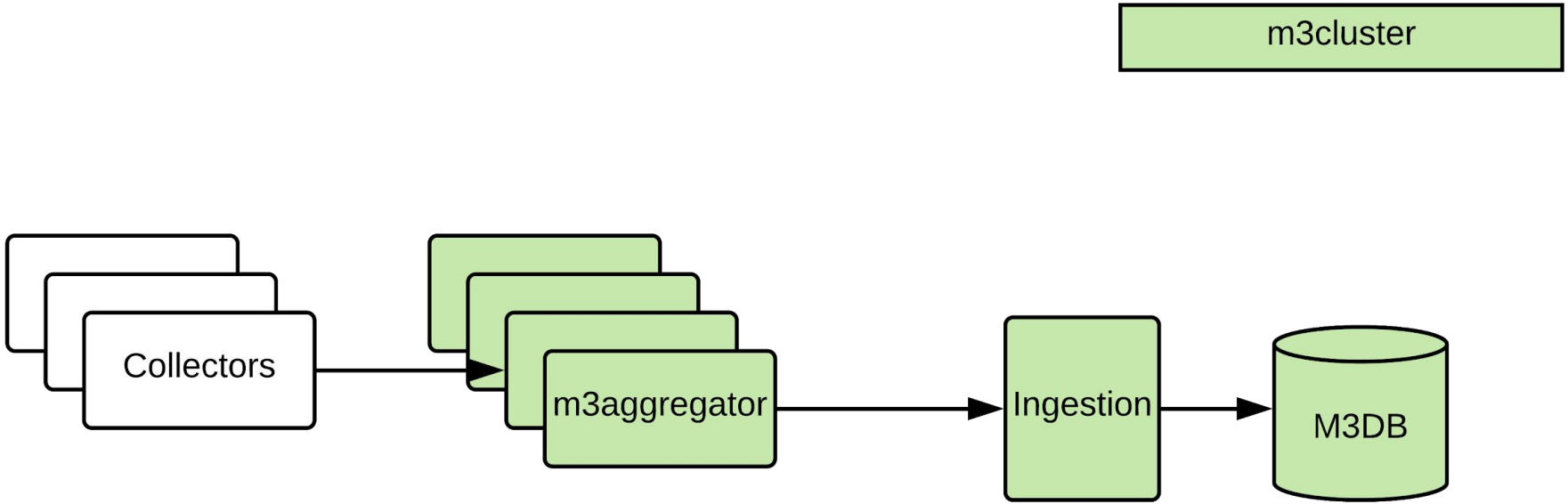
- Had smaller components rather than one unifying view of the system as a whole
- **Difficult to reason about**
  - “Where do I need to make this change?”
  - Painful onboarding for new engineers
- **Impediment to automation**
  - Replace host: update config, build N services, deploy

System Representation (2016)





System Representation: 2018



m3cluster benefits

- Common libraries for all workloads: stateful (M3DB), semi-stateful (aggregation tier), stateless (ingestion)
  - Implement “distribute shards across racks according to disk size” once, reuse everywhere
- Single source of truth: everything stored in etcd

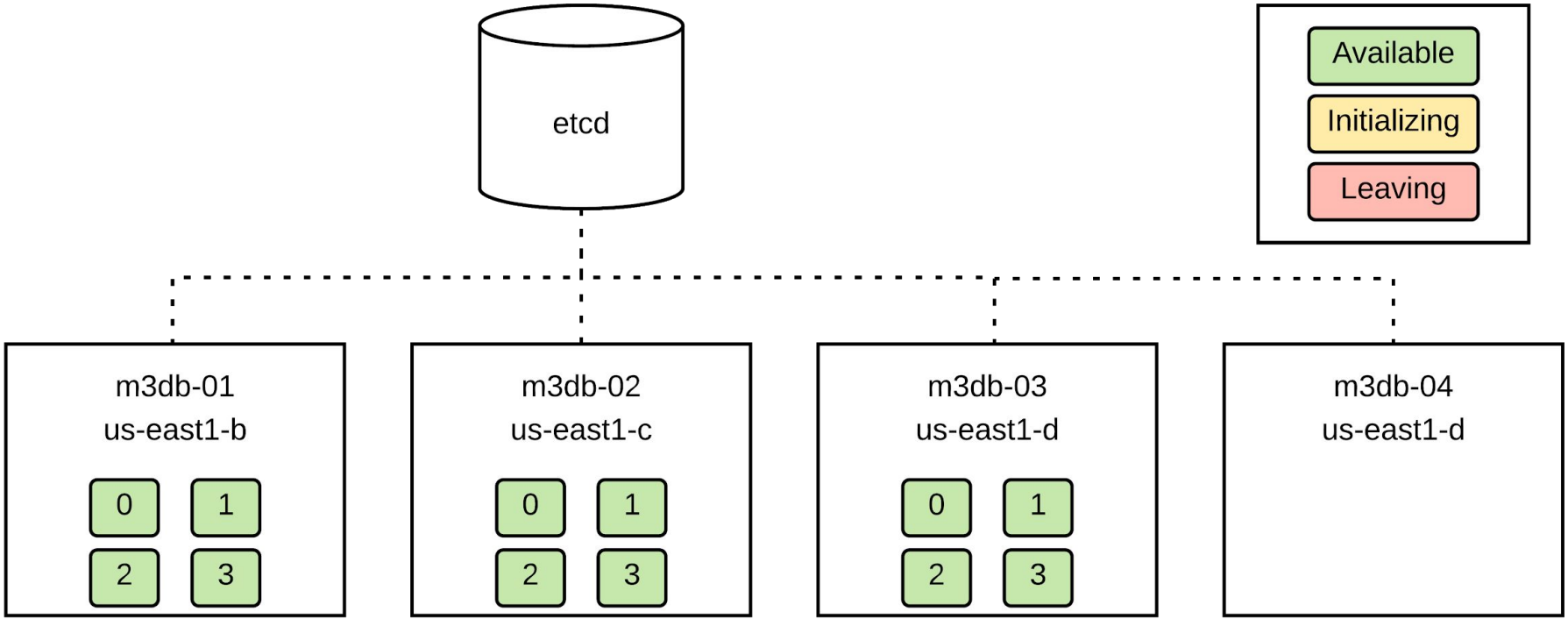
Operational Improvements

- Easier to reason about the entire system
  - Unifying view: Placements
- Operations much easier, possible to automate
  - Just modifying goal state in etcd
- One config per deployment (no instance-specific)
  - Cattle vs. pets: instances aren’t special, treat them as a whole

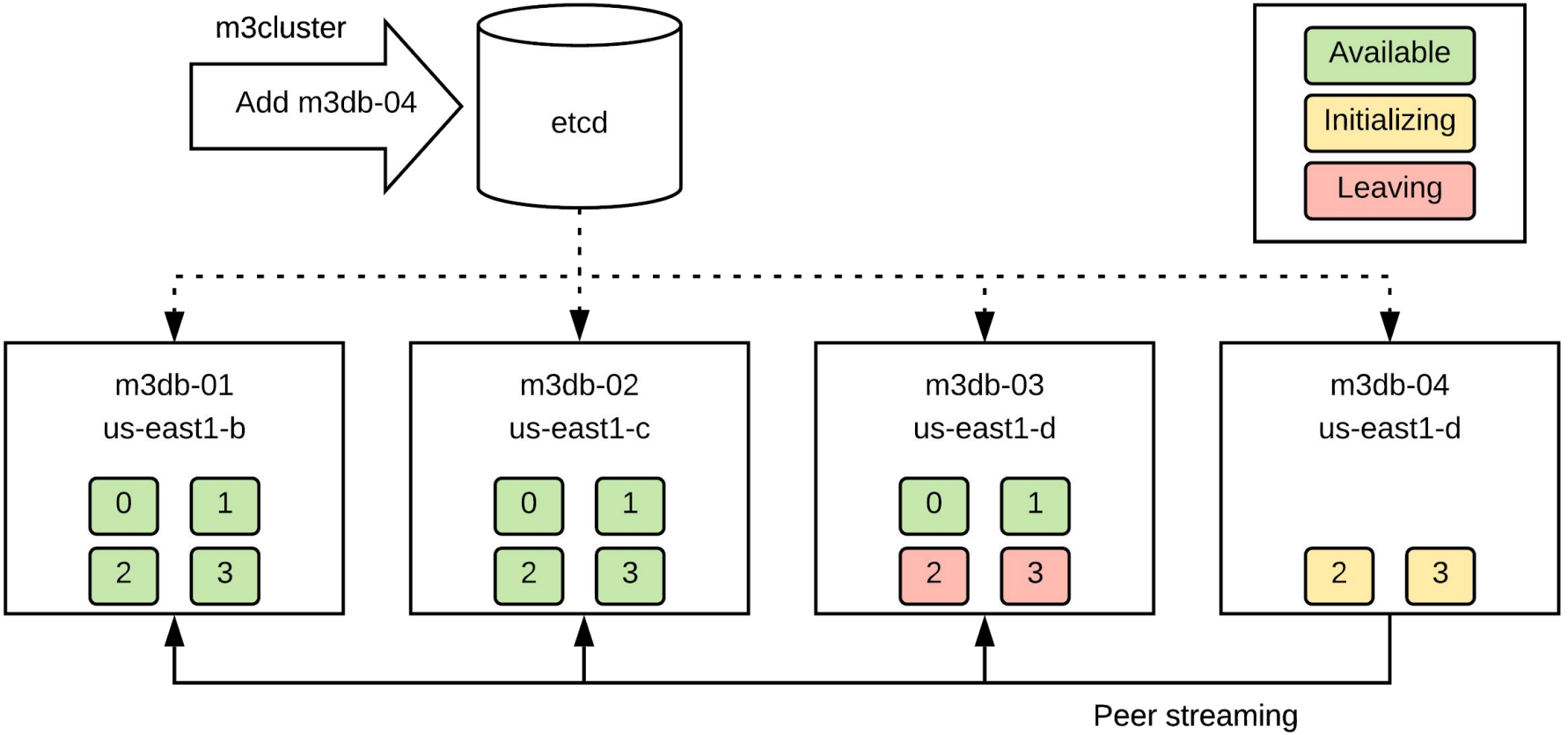
M3DB Goal State

- Shard assignments stored in etcd
  - M3DB nodes observe desired state, react, update shard state
- Imperative actions such as “add a node” are really changes in declarative state
  - Easy for both people or tooling to interact with

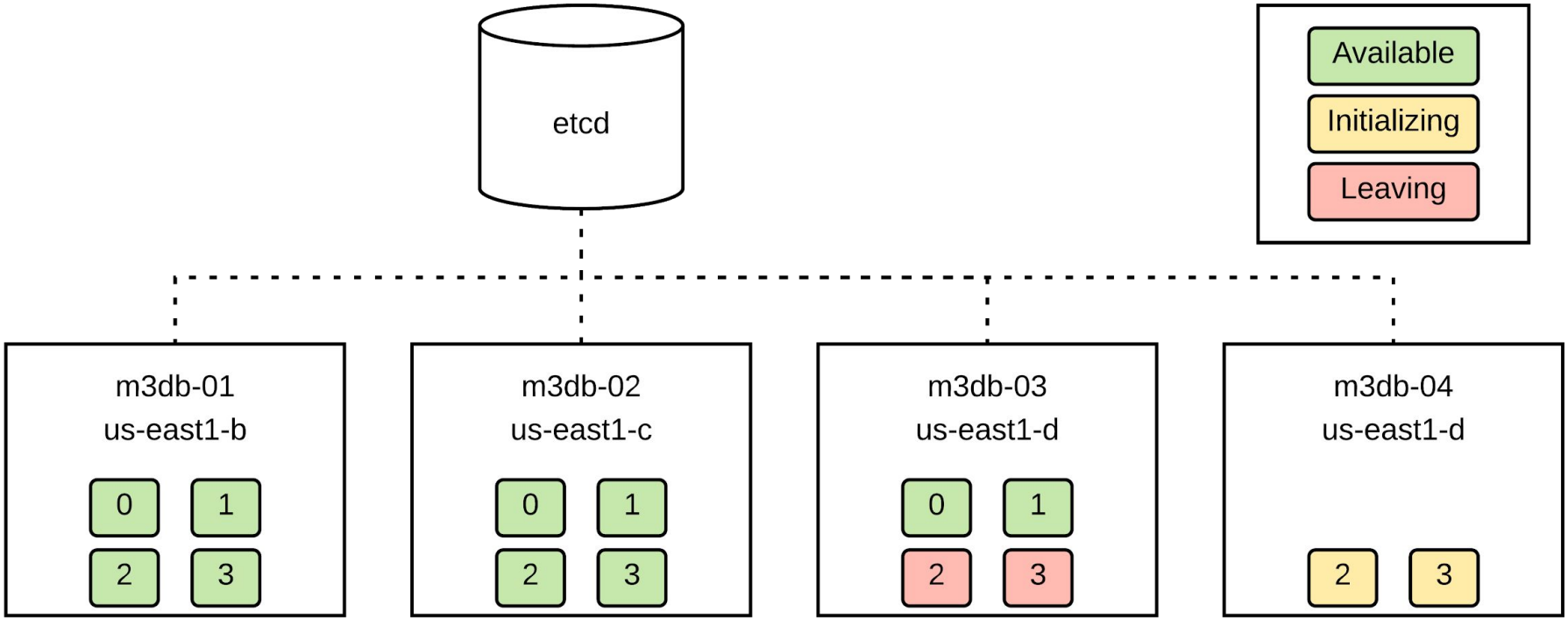
M3DB: Adding a Node



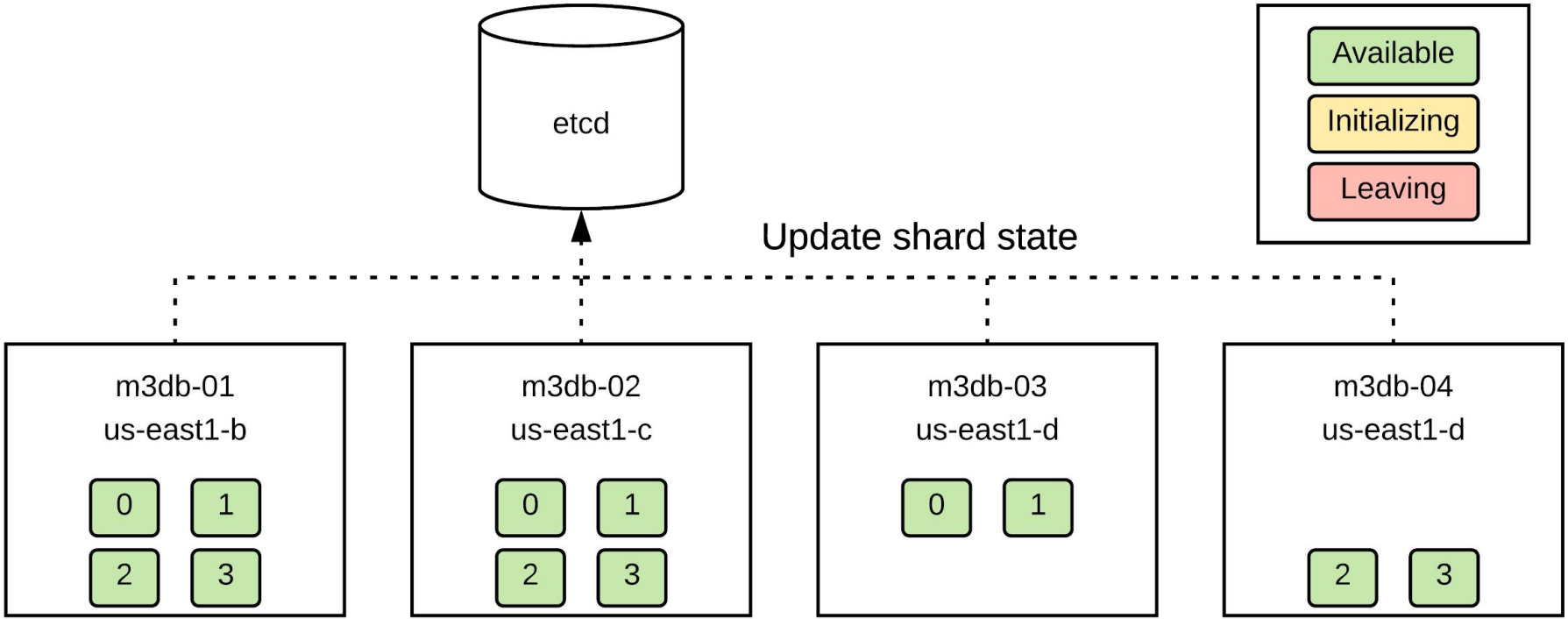
M3DB: Adding a Node



M3DB: Adding a Node



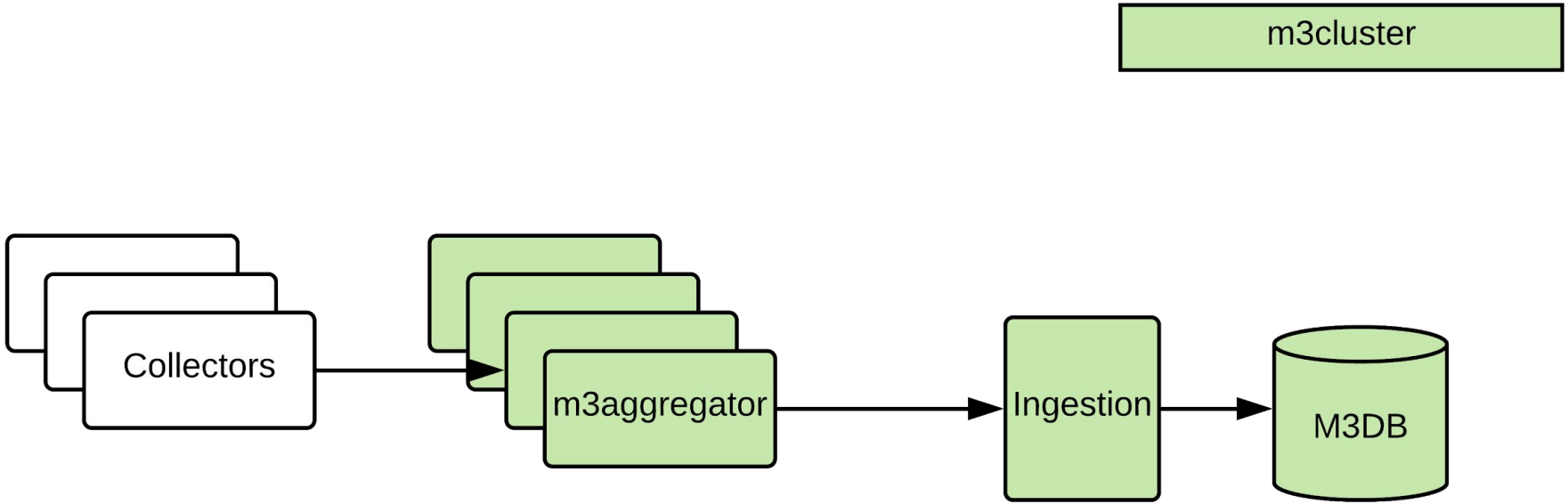
M3DB: Adding a Node



M3DB Goal State

- Goal-state primitives built with future automation in mind
- m3cluster interacts with single source of truth
- Vs. imperative changes:
  - No instance-level operations
  - No state or steps to keep track of: can always reconcile state of world (restarts, etc. safe)

System Representation: 2018



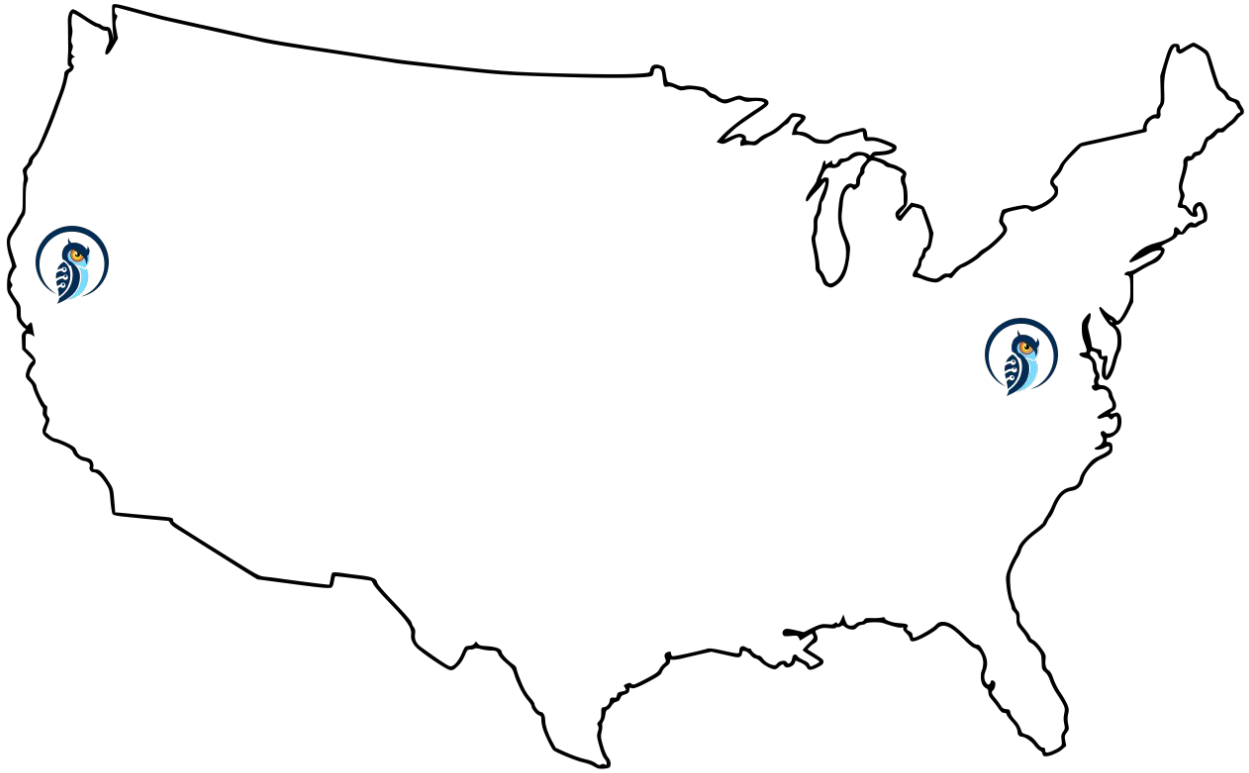
M3DB Operations

- Retooled entire stack as goal state-driven, dynamic
  - Operations simplified, but still triggered by a person
- M3DB requires more care to manage than stateless
  - Takes time to stream data

2016

2

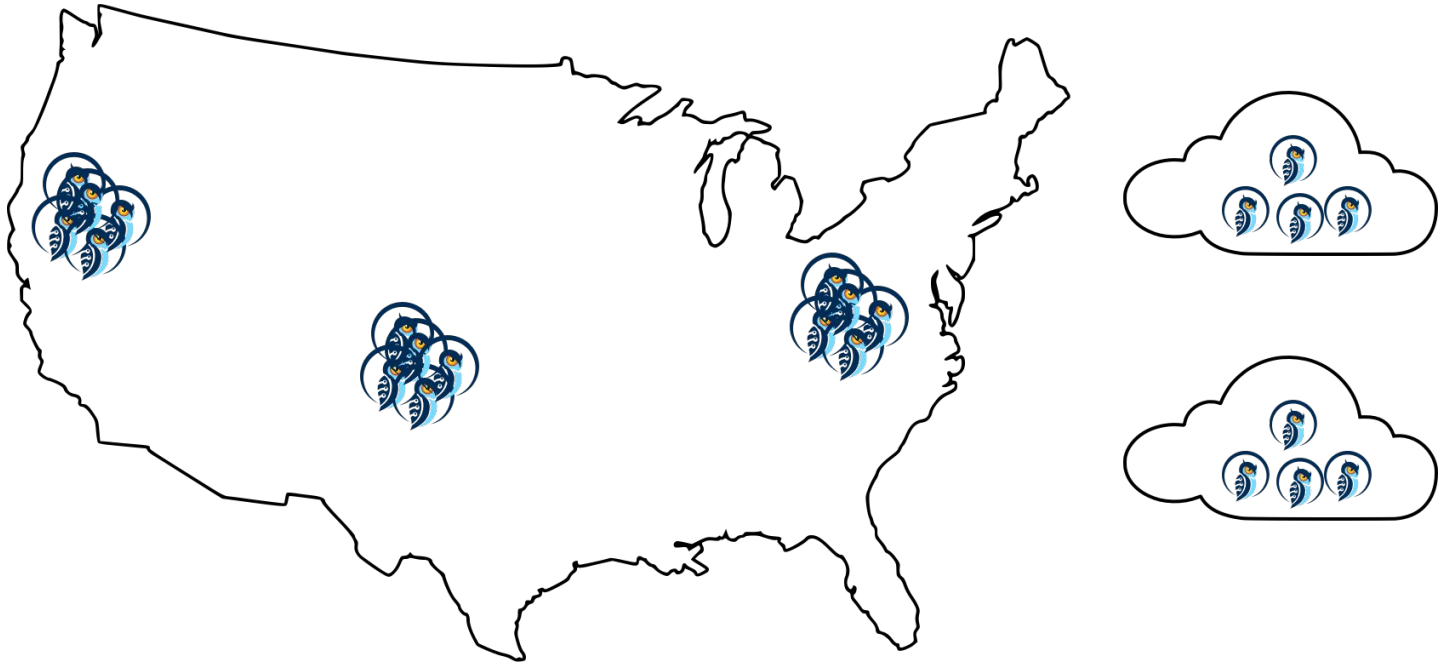
Clusters



2018

40+

Clusters



Automating M3DB

- Wanted goal state at a higher level
  - Clusters as cattle
- M3DB can only do so much; needed to expand scope
- Chose to build on top of **Kubernetes**
  - Value for M3DB OSS community
  - Internal Uber use cases

Kubernetes @ 35,000 ft

- Container orchestration system
- Support for stateful workloads
- Self-healing
- Extensible
  - CRD: define your own API resources
  - Operator: custom controller

Kubernetes Driving Principles

- Pod: base unit of work
  - Group of like containers deployed together
- Pods can come and go
  - Self-healing → pods move in response to failures
- No pods are special

Kubernetes Controllers

```
for {
  desired := getDesiredState()
  current := getCurrentState()
  makeChanges(desired, current)
}
```

Writing Controllers Guide

M3DB & Kubernetes

M3DB:

- Goal state driven
- Single source of truth
- Nodes work to converge

Kubernetes:

- Goal state driven
- Single source of truth
- Components work to converge

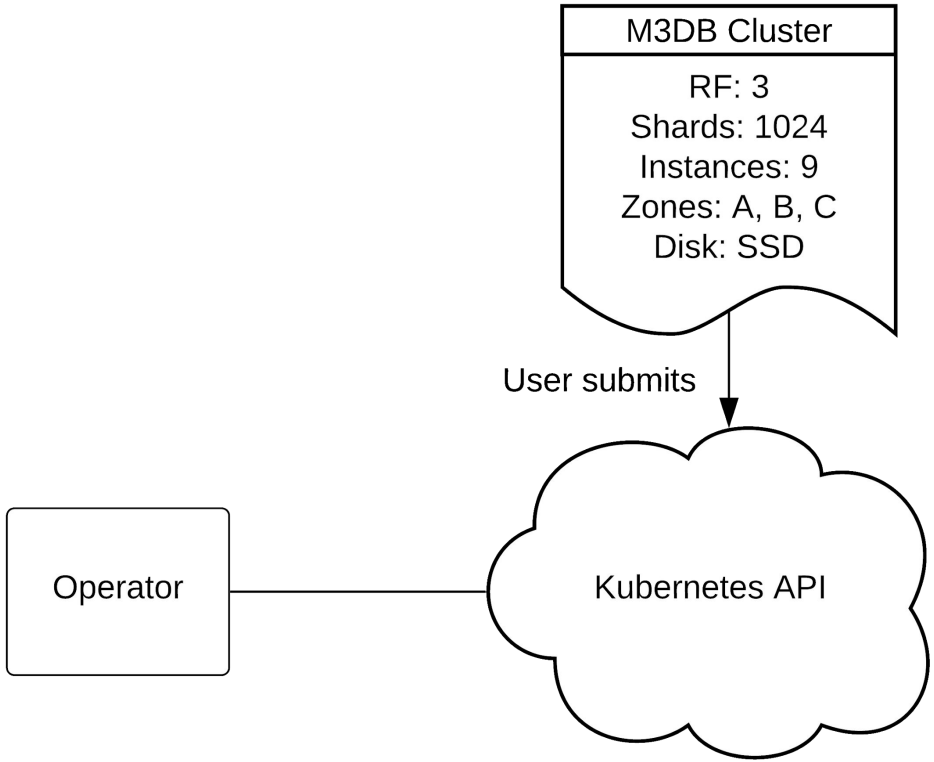
M3DB Operator

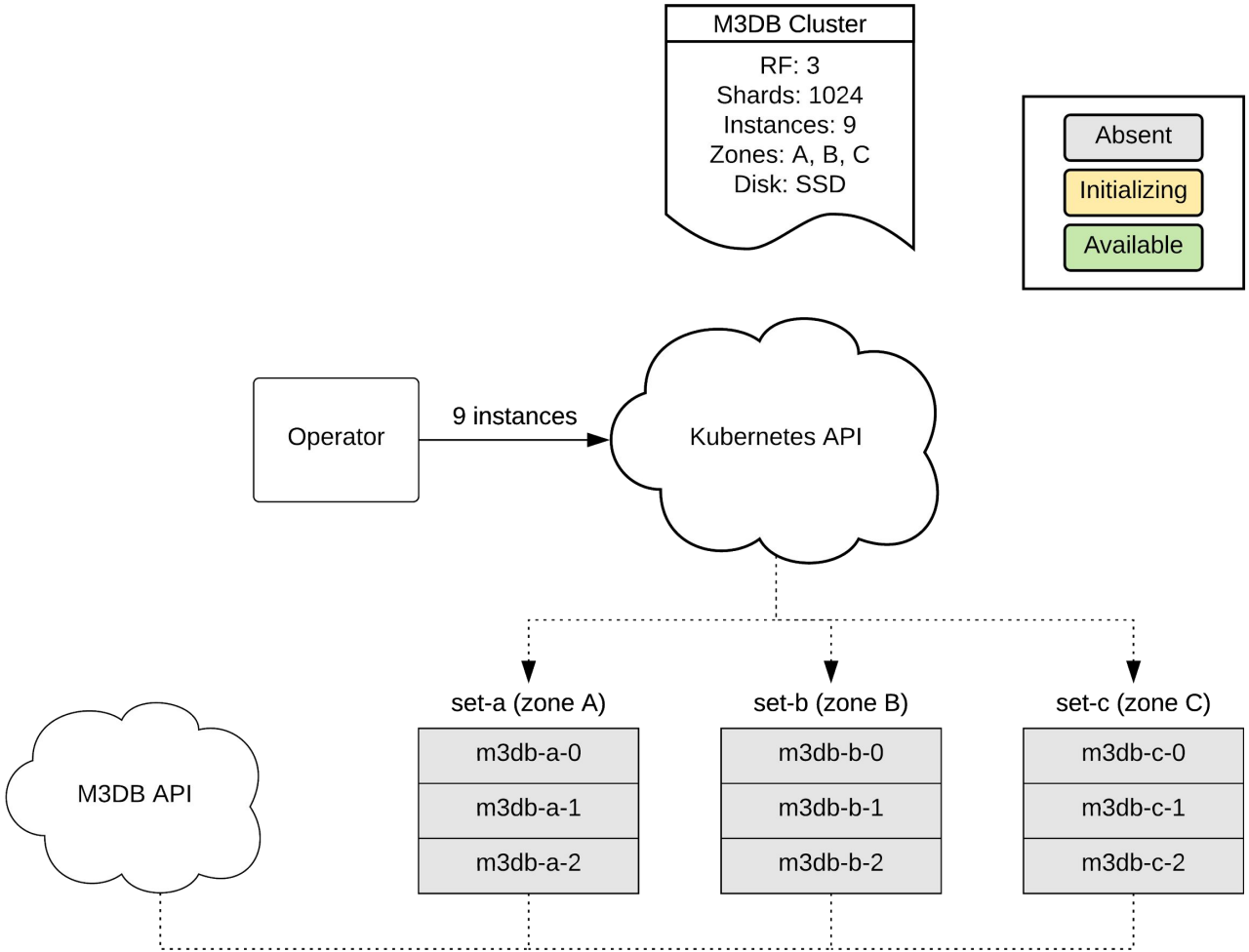
- Manages M3DB clusters running on Kubernetes
  - Creation, deletion, scaling, maintenance
- Performs actions a person would have to do in full cluster lifecycle
  - Updating M3DB clusters
  - Adding more instances

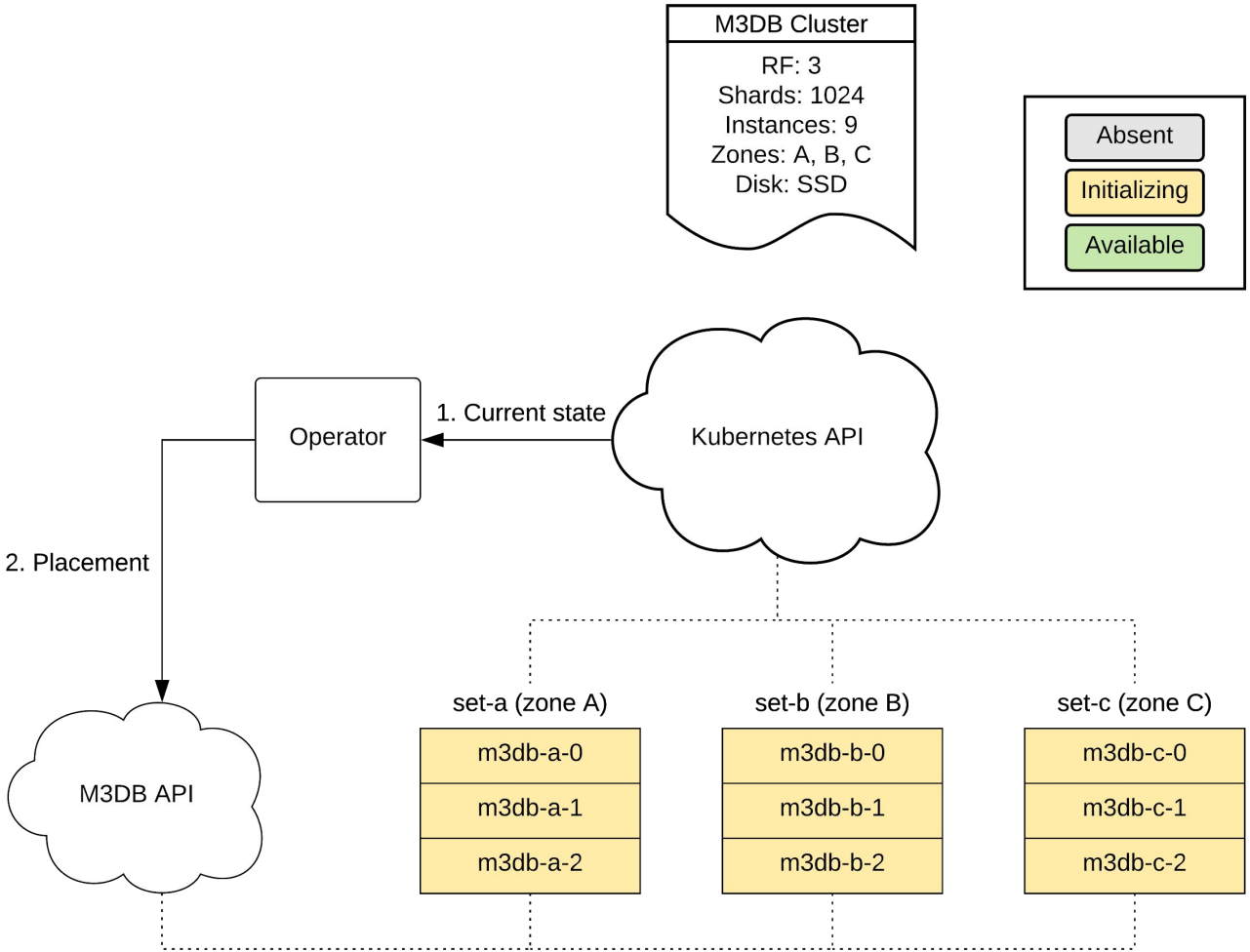
M3DB Operator

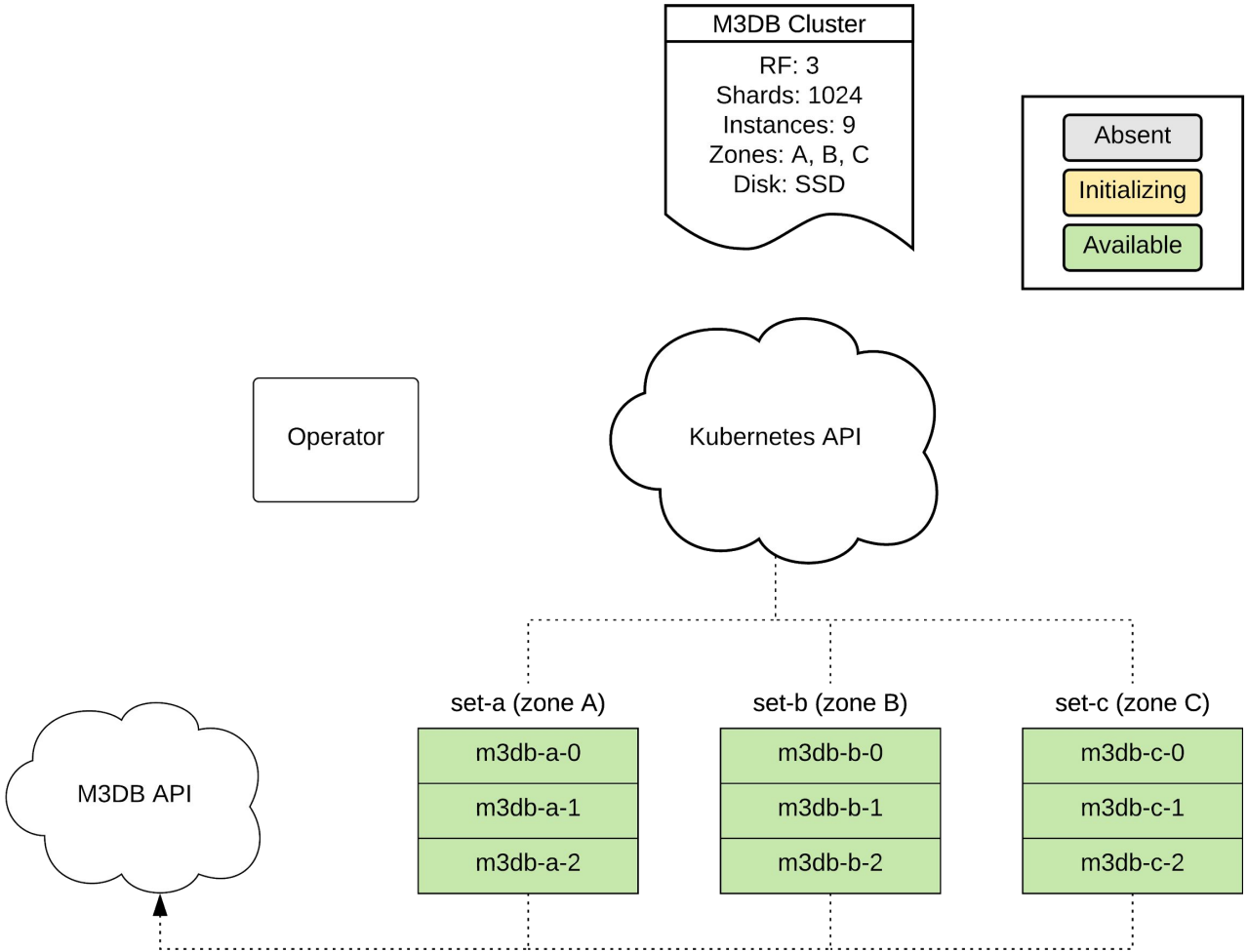
- Users define desired M3DB clusters
  - “9 node cluster, spread across 3 zones, with remote SSDs attached”
- Operator updates desired Kubernetes state, waits for convergence, updates M3DB desired state

M3DB Operator: Create Example









M3DB Operator: Day 2 Operations

- Scaling a cluster
  - M3DB adds 1-by-1, operation handles many
- Replacing instances, updating config, etc.
- All operations follow same pattern
  - Look at Kubernetes state, look at M3DB state, affect change

M3DB Operator: Success Factors

- Bridge between goal state-driven APIS (anti-footgun)
  - Can’t accidentally remove more pods than desired
  - Can’t accidentally remove two M3DB instances
  - Operator can be restarted, pick back up
- Share similar concepts, expanded scope
  - Failure domains, resource quantities, etc.

Lessons Learned: M3

- Finding common abstractions made mental model easier
  - Implementation can be separate, interface the same
- + Goal-state driven approach simplified operations
  - External factors (failures, deploys) don’t disrupt steady state

Lessons Learned: Kubernetes

- Embracing Kubernetes principles in M3DB made Kubernetes onboarding easier
  - No pods are special, may move around
  - Self-healing after failures
- Instances as cattle, not pets
  - No instance-level operations or config

Considerations

- Any orchestration system implies complexity
  - **Your own context**: is it worth it?
  - How does your own system respond to failures, rescheduling, etc.?
- Maybe not a good fit if have single special instances
  - M3DB can tolerate failures

Considerations

- Requirements & guarantees of your system will influence your requirements for automation
  - M3DB strongly consistent: wanted strongly consistent cluster membership

Recap

- Building common abstractions for all our workloads

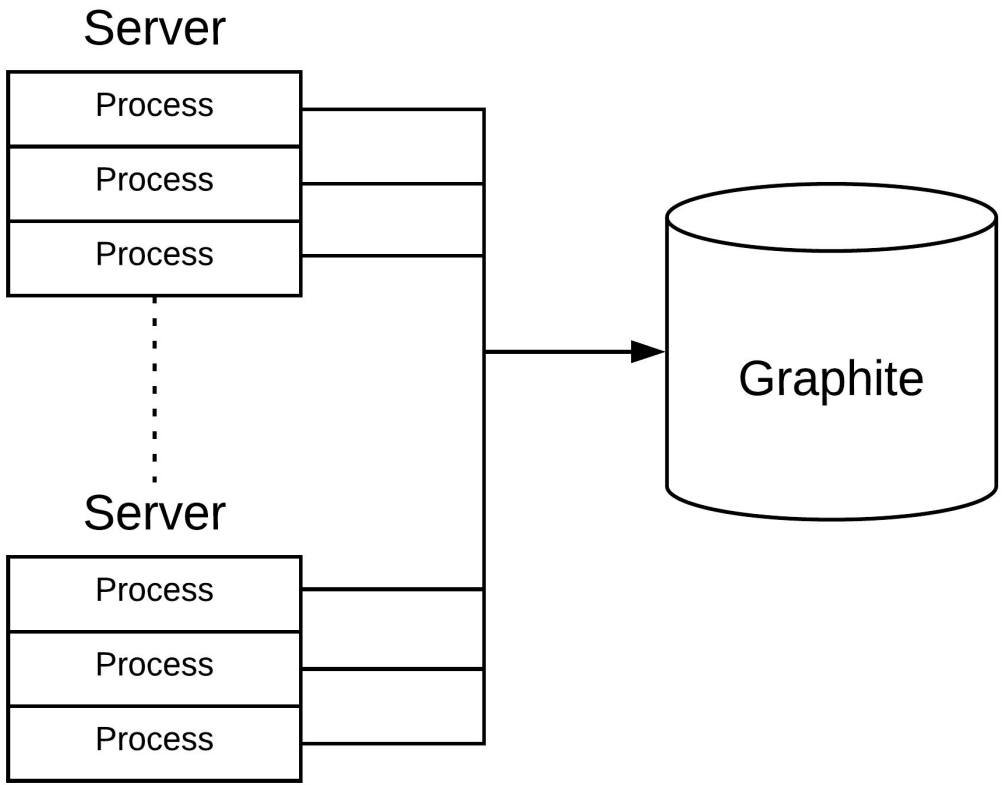
  eased mental model
- Following goal-state driven approach eased automation
- Closely aligned with Kubernetes principles

Thank you!

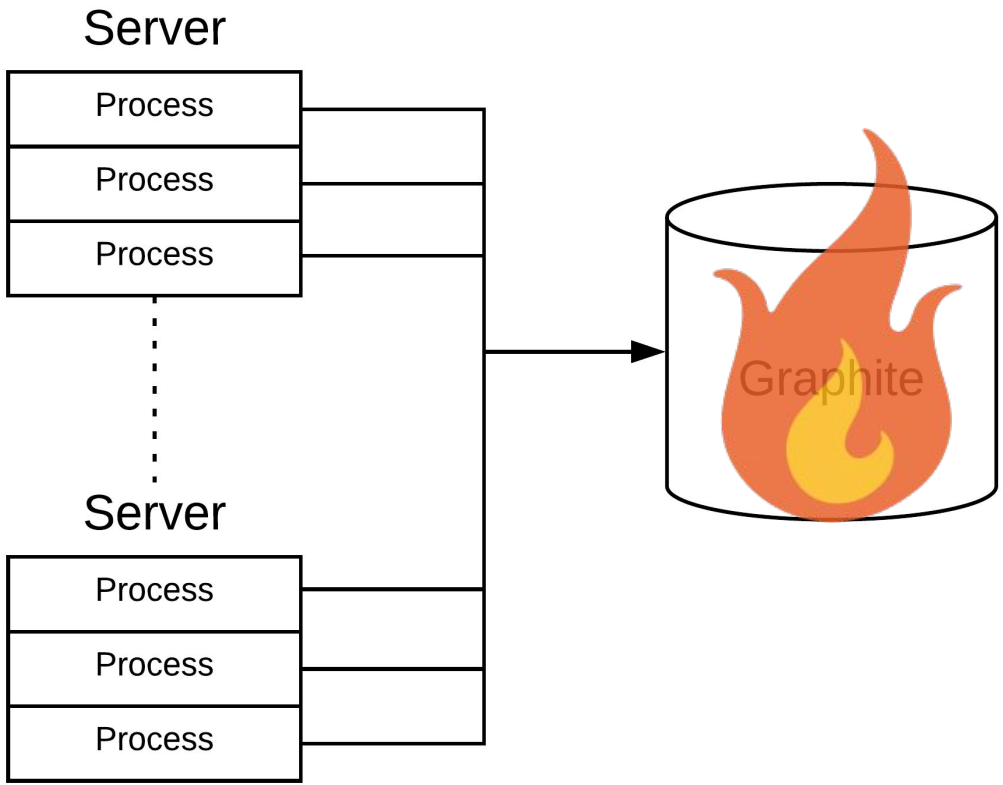
- github.com/m3db/m3
  - M3DB, m3cluster, m3aggregator
  - github.com/m3db/m3db-operator
- m3db.io & docs.m3db.io
- Please leave feedback on O'Reilly site!

@mattschallert

Pre-2015


Server
Process
Process
Process
Server
Process
Process
Process
Graphite

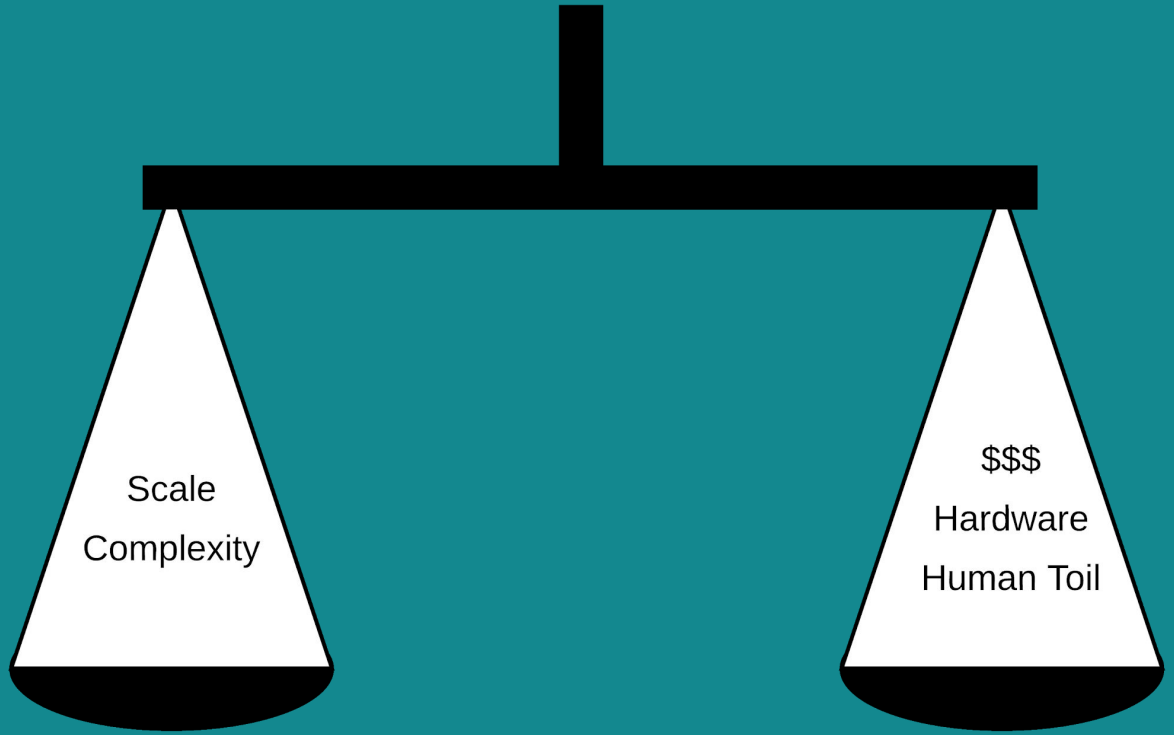
Early 2015


Server
Process
Process
Process
Server
Process
Process
Process
Graphite

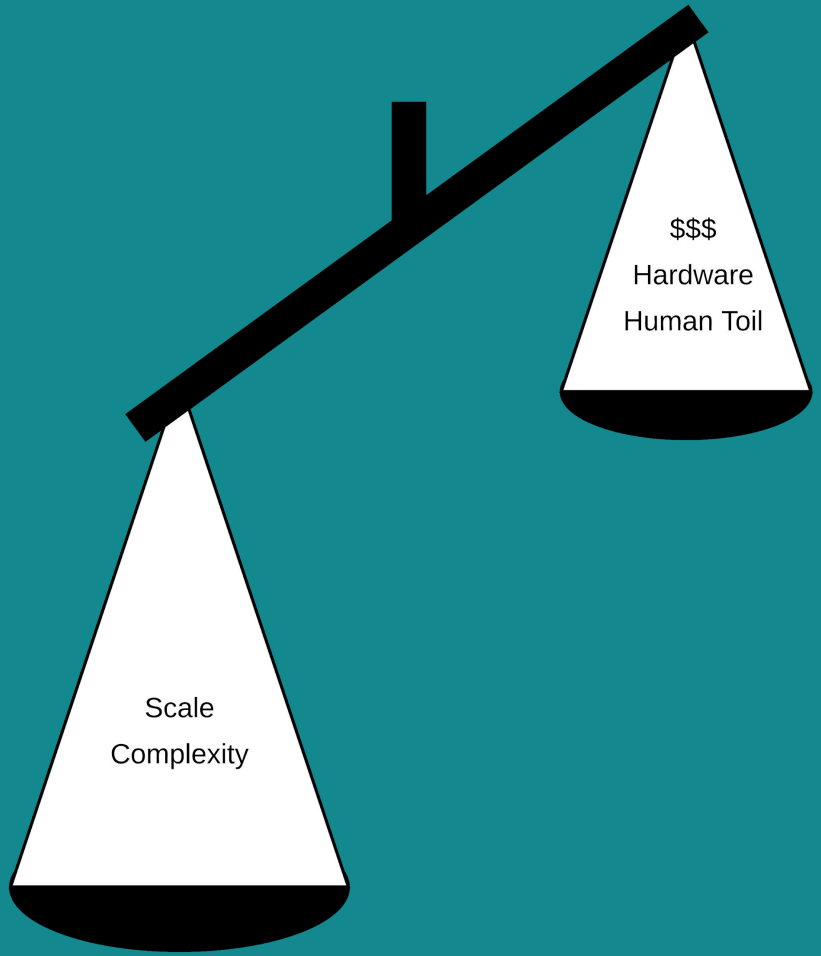
M3DB Operator: Today

- [THIS SLIDE IS SKIPPED]
- https://github.com/m3db/m3db-operator
- Managing 1 M3DB clusters ~as easy as 10

Late 2015

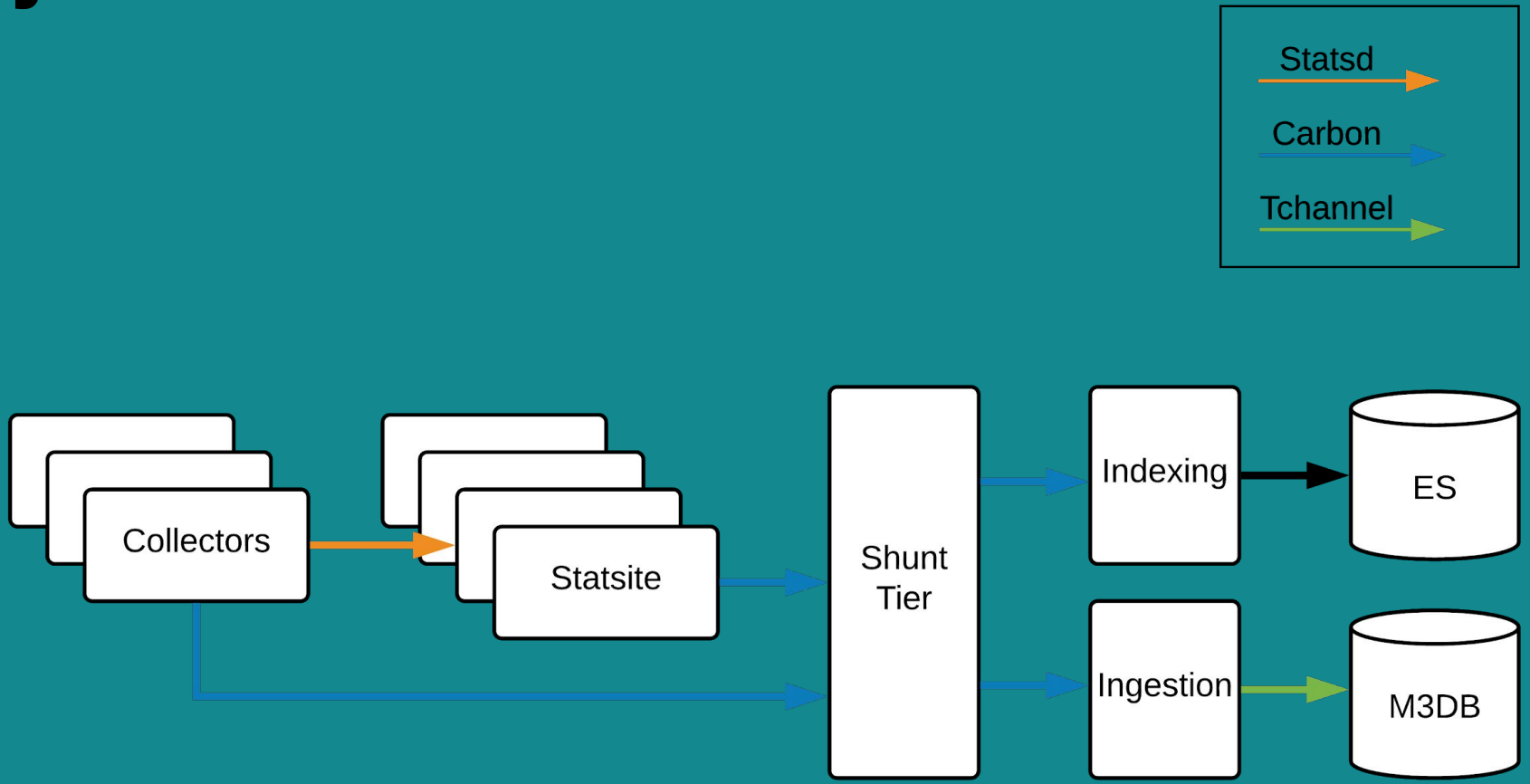


Mid 2016



Post-M3DB Deployment

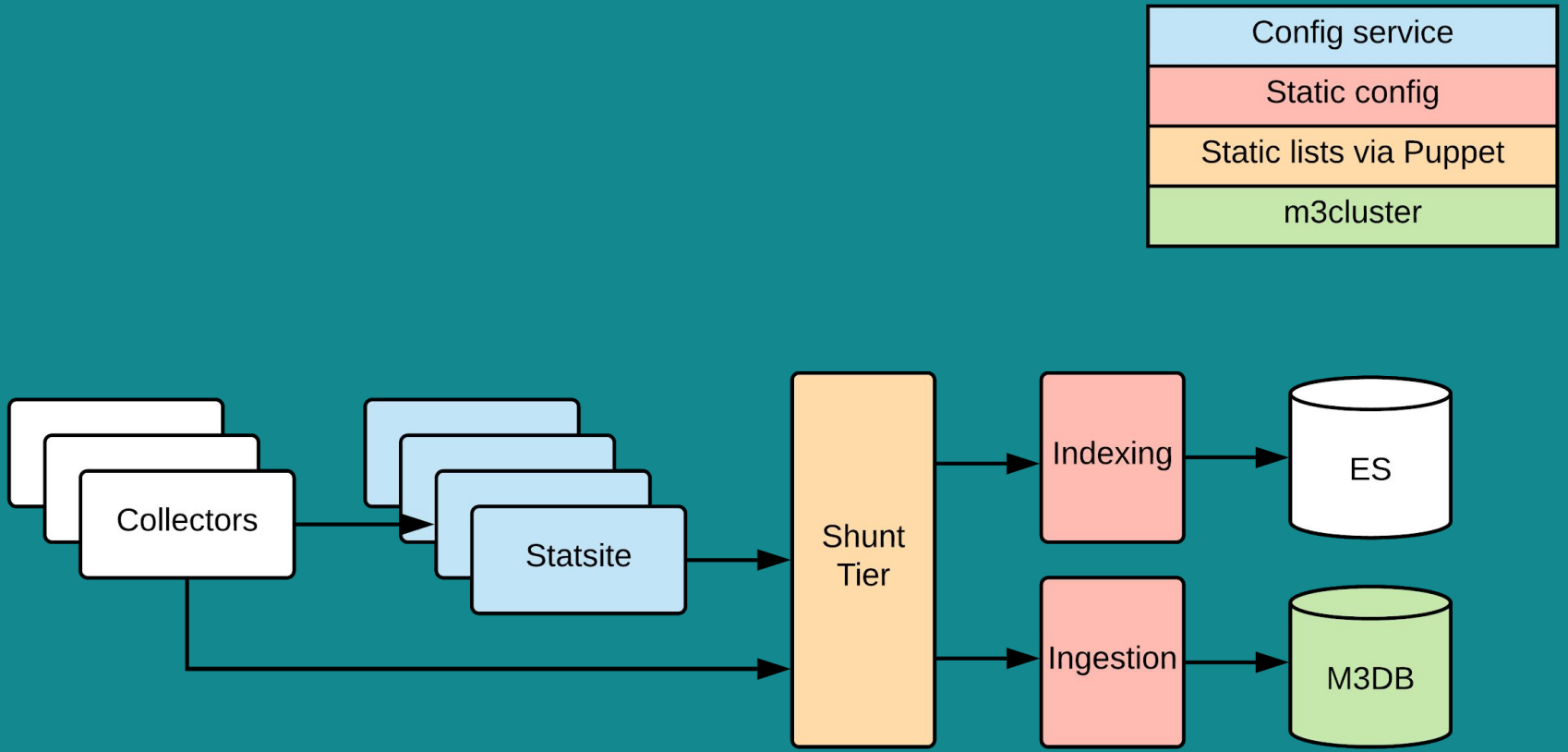
System Interfaces



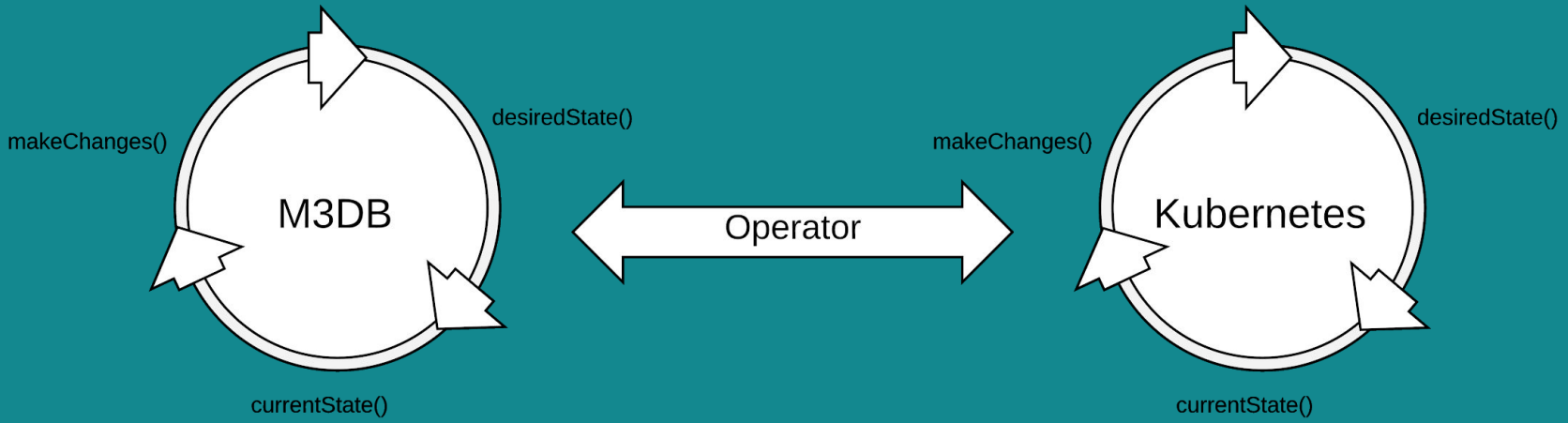
System Interfaces

- Ingestion path used multiple text-based line protocols
  - Difficult to encode metric metadata
  - Redundant serialization costs
- Storage layers used various RPC formats

System Representation (2016)



M3DB Operator [SLIDE SKIPPED]



ROUGH DRAFT DIAGRAM

Edge vs. Level Triggered Logic

- [Transition to Kubernetes, possibly using edge vs. level triggered]
  - Level triggered: reacting to current state. Actions such as “replace this node” can’t be missed.
  - Edge triggered: reacting to events. Events can be missed.
- [Maybe something about idempotency requirement ?]

m3cluster: Placement

```
message Placement {
 map<string, Instance> instances
 uint32 replica_factor
 uint32 num_shards
 bool is_sharded
}
```

```
message Instance {
 string id
 string isolation_group
 string zone
 uint32 weight
 string endpoint
 repeated Shard shards
 ...
}
```

m3cluster: Shard

```
message Shard {
 uint32 id = 1;
 ShardState state = 2;
 string source_id = 3;
}
```

```
enum ShardState {
 INITIALIZING = 0;
 AVAILABLE = 1;
 LEAVING = 2;
}
```

M3cluster: all

```
enum ShardState {
 INITIALIZING = 0;
 AVAILABLE = 1;
 LEAVING = 2;
}
```

```
message Shard {
 uint32 id = 1;
 ShardState state = 2;
 string source_id = 3;
}
```

```
message Instance {
 string id = 1;
 string isolation_group = 2;
 string zone = 3;
 uint32 weight = 4;
 string endpoint = 5;
 repeated Shard shards = 6;
 ...
}
```

```
message Placement {
 map<string, Instance> instances = 1;
 uint32 replica_factor = 2;
 uint32 num_shards = 3;
 bool is_sharded = 4;
}
```

Prerequisites for Orchestration

- Consistency of operations
  - Many actors in the system
  - Failures can happen, may re-reconcile states
  - Kubernetes & m3cluster leverage etcd

Lessons Learned

- [Lessons learned orchestrating M3DB generally]
- [Learnings specific to Kubernetes]