Cassandra at Yammer

Michał Rutkowski (mrutkowski@yammer-inc.com)

Plan

- About Yammer
- What we wanted to change and why
- How we rolled out Cassandra
- What we’ve learned and what worked well

Plan

- About Yammer
- What we wanted to change and why
- How we rolled out Cassandra
- What we’ve learned and what worked well

Plan

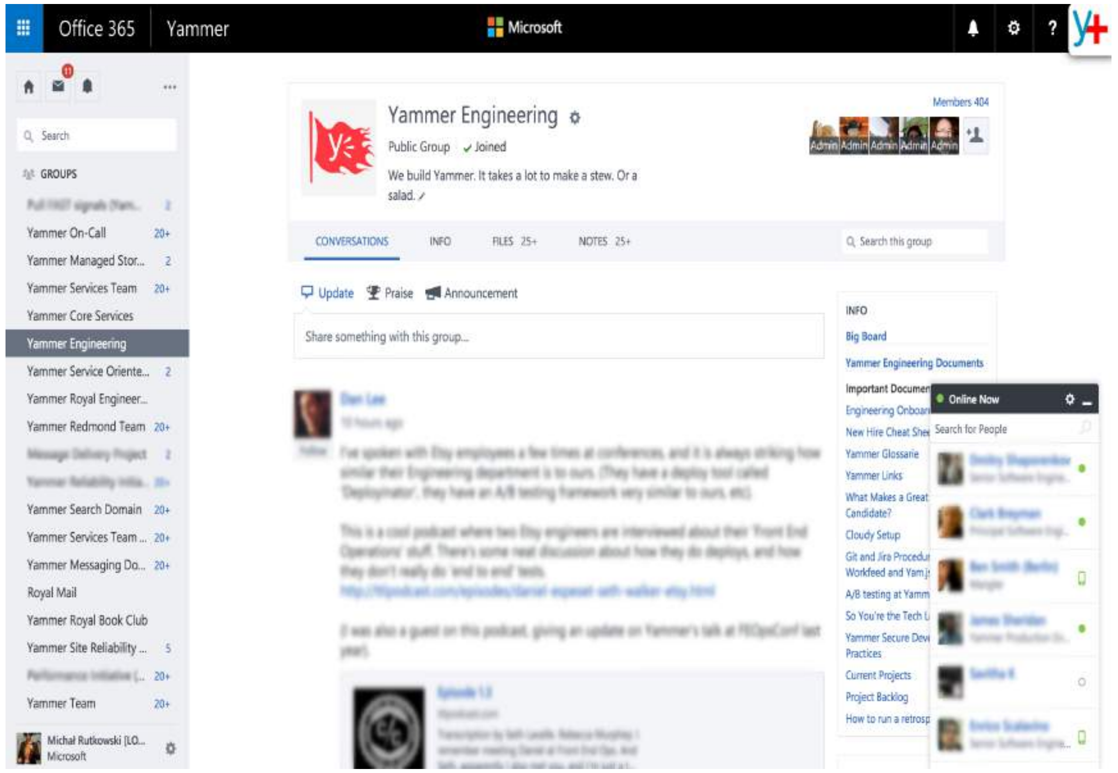
- About Yammer
- What we wanted to change and why
- How we rolled out Cassandra
- What we've learned and what worked well

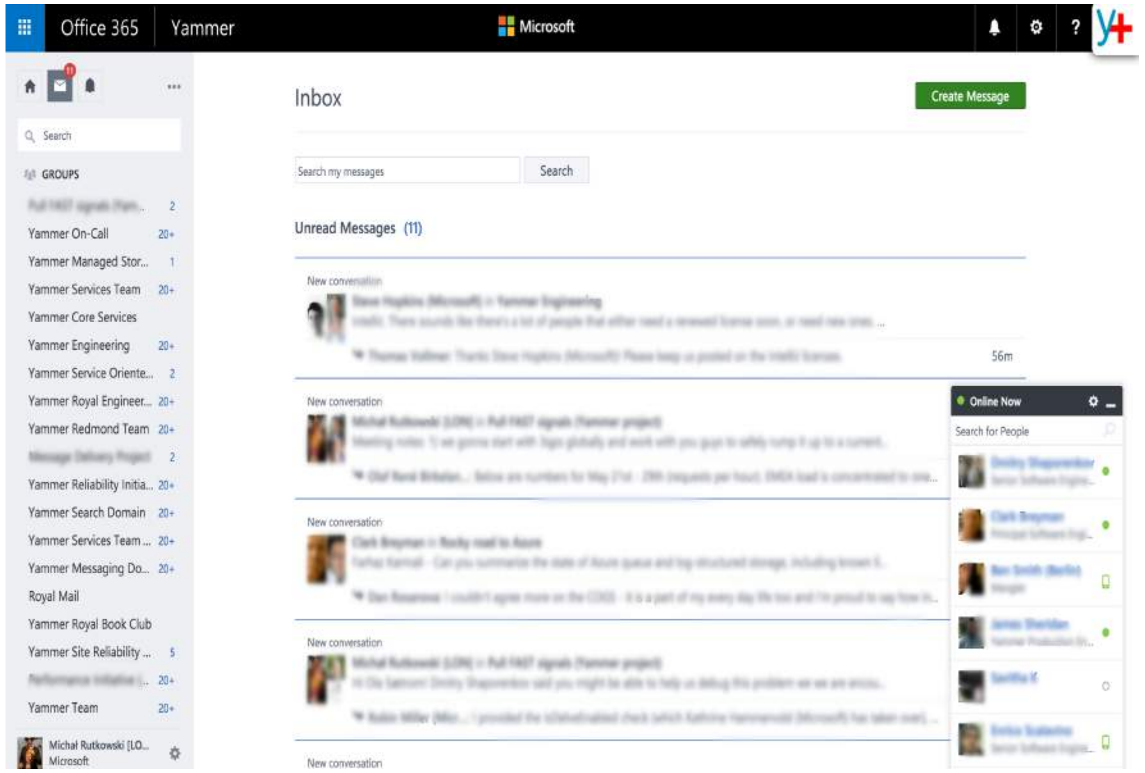
Plan

- About Yammer
- What we wanted to change and why
- How we rolled out Cassandra
- What we’ve learned and what worked well

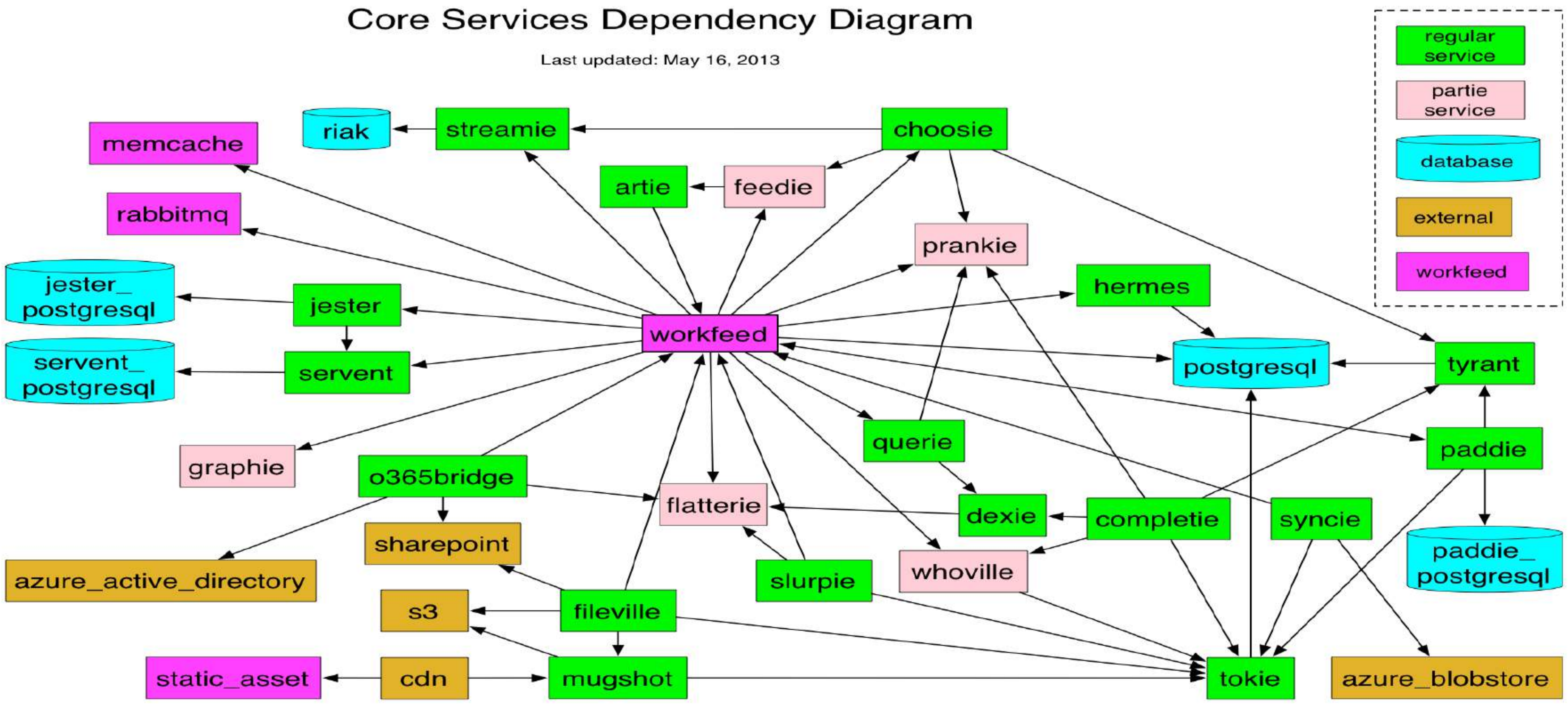
About Yammer

An Enterprise Social Network whose aim is to facilitate better and faster communication within an organization.





Yammer's Architecture



Yammer’s Operational Tooling

This obviously quite complex, so we have some tooling:

- CI – Team City and Jenkins
- Deployment – homegrown DW service
- Analytics – homegrown service for querying across multiple DBs
- Metrics – DW metrics, Kafka based collection pipeline and Wavefront for visualization
- Log aggregation (Logstash + Kibana)

Yammer’s Operational Tooling

This obviously quite complex, so we have some tooling:

- CI – Team City and Jenkins
- Deployment – homegrown DW service
- Analytics – homegrown service for querying across multiple DBs
- Metrics – DW metrics, Kafka based collection pipeline and Wavefront for visualization
- Log aggregation (Logstash + Kibana)

Yammer’s Operational Tooling

This obviously quite complex, so we have some tooling:

- CI – Team City and Jenkins
- Deployment – homegrown DW service
- Analytics – homegrown service for querying across multiple DBs
- Metrics – DW metrics, Kafka based collection pipeline and Wavefront for visualization
- Log aggregation (Logstash + Kibana)

Yammer’s Operational Tooling

This obviously quite complex, so we have some tooling:

- CI – Team City and Jenkins
- Deployment – homegrown DW service
- Analytics – homegrown service for querying across multiple DBs
- Metrics – DW metrics, Kafka based collection pipeline and Wavefront for visualization
- Log aggregation (Logstash + Kibana)

Yammer’s Operational Tooling

This obviously quite complex, so we have some tooling:

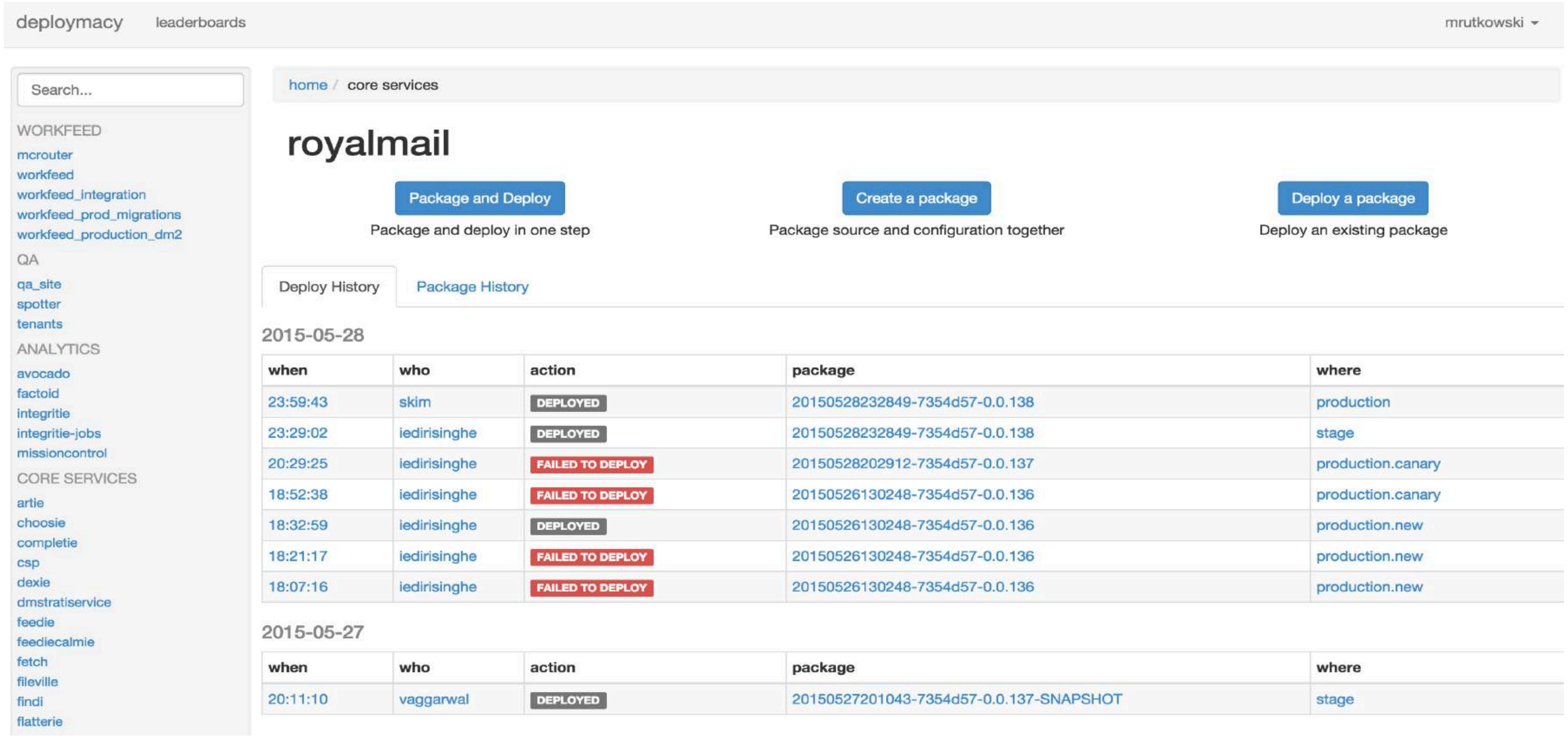
- CI – Team City and Jenkins
- Deployment – homegrown DW service
- Analytics – homegrown service for querying across multiple DBs
- Metrics – DW metrics, Kafka based collection pipeline and Wavefront for visualization
- Log aggregation (Logstash + Kibana)

Yammer’s Operational Tooling

This obviously quite complex, so we have some tooling:

- CI – Team City and Jenkins
- Deployment – homegrown DW service
- Analytics – homegrown service for querying across multiple DBs
- Metrics – DW metrics, Kafka based collection pipeline and Wavefront for visualization
- Log aggregation (Logstash + Kibana)

Deployment



deploymacy   leaderboards   mrutkowski

Search...

WORKFEED
- mcrouter
- workfeed
- workfeed_integration
- workfeed_prod_migrations
- workfeed_production_dm2

QA
- qa_site
- spotter
- tenants

ANALYTICS
- avocado
- factoid
- integritie
- integritie-jobs
- missioncontrol

CORE SERVICES
- artie
- choosie
- completie
- csp
- dexie
- dmstratiservice
- feedie
- feediecalmie
- fetch
- fileville
- findi
- flatterie

home / core services

royalmail

Package and Deploy — Package and deploy in one step

Create a package — Package source and configuration together

Deploy a package — Deploy an existing package

Deploy History | Package History

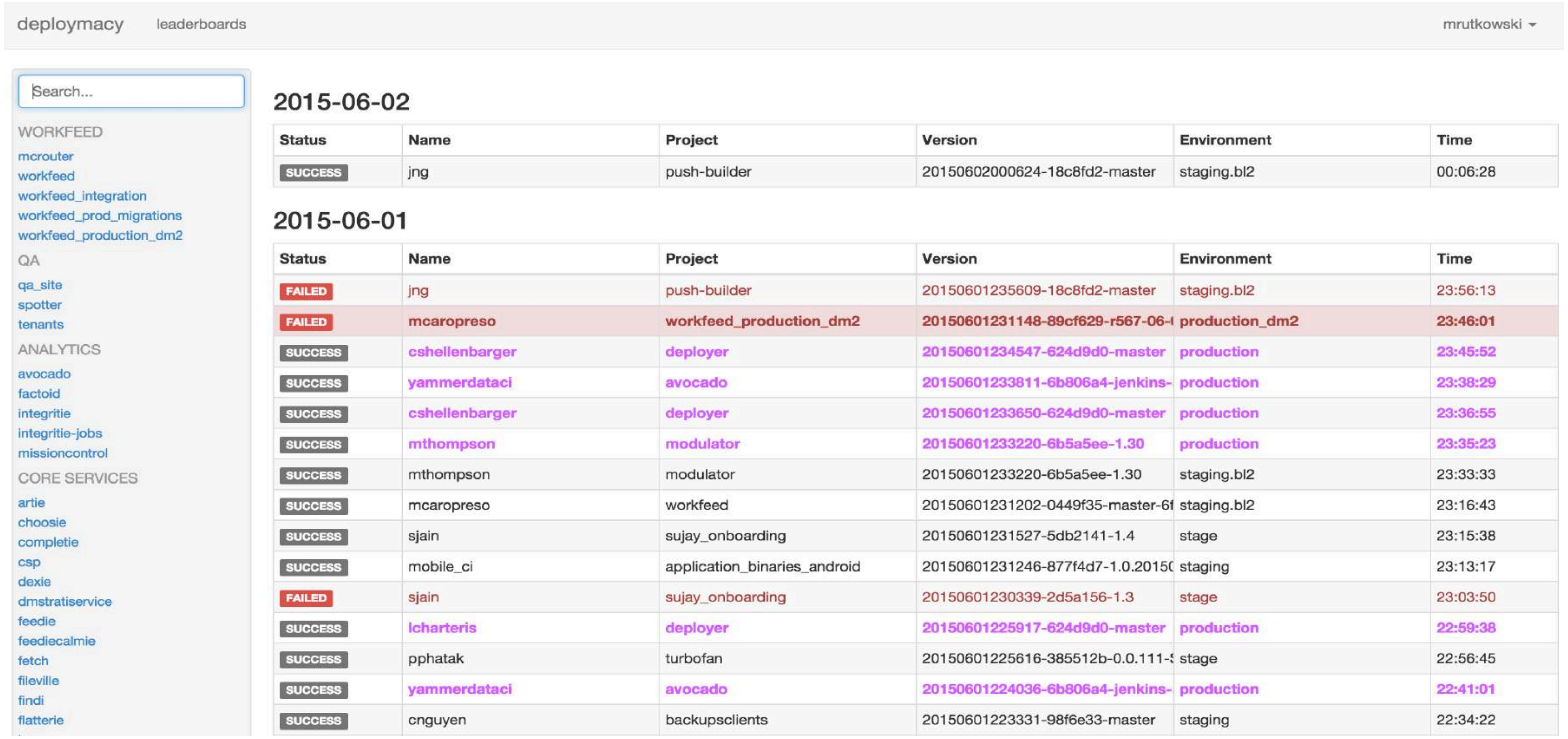
2015-05-28

when	who	action	package	where
23:59:43	skim	DEPLOYED	20150528232849-7354d57-0.0.138	production
23:29:02	iedirisinghe	DEPLOYED	20150528232849-7354d57-0.0.138	stage
20:29:25	iedirisinghe	FAILED TO DEPLOY	20150528202912-7354d57-0.0.137	production.canary
18:52:38	iedirisinghe	FAILED TO DEPLOY	20150526130248-7354d57-0.0.136	production.canary
18:32:59	iedirisinghe	DEPLOYED	20150526130248-7354d57-0.0.136	production.new
18:21:17	iedirisinghe	FAILED TO DEPLOY	20150526130248-7354d57-0.0.136	production.new
18:07:16	iedirisinghe	FAILED TO DEPLOY	20150526130248-7354d57-0.0.136	production.new

2015-05-27

when	who	action	package	where
20:11:10	vaggarwal	DEPLOYED	20150527201043-7354d57-0.0.137-SNAPSHOT	stage

Deployment



deploymacy　leaderboards　mrutkowski

Search...

WORKFEED
- mcrouter
- workfeed
- workfeed_integration
- workfeed_prod_migrations
- workfeed_production_dm2

QA
- qa_site
- spotter
- tenants

ANALYTICS
- avocado
- factoid
- integritie
- integritie-jobs
- missioncontrol

CORE SERVICES
- artie
- choosie
- completie
- csp
- dexie
- dmstratiservice
- feedie
- feediecalmie
- fetch
- fileville
- findi
- flatterie

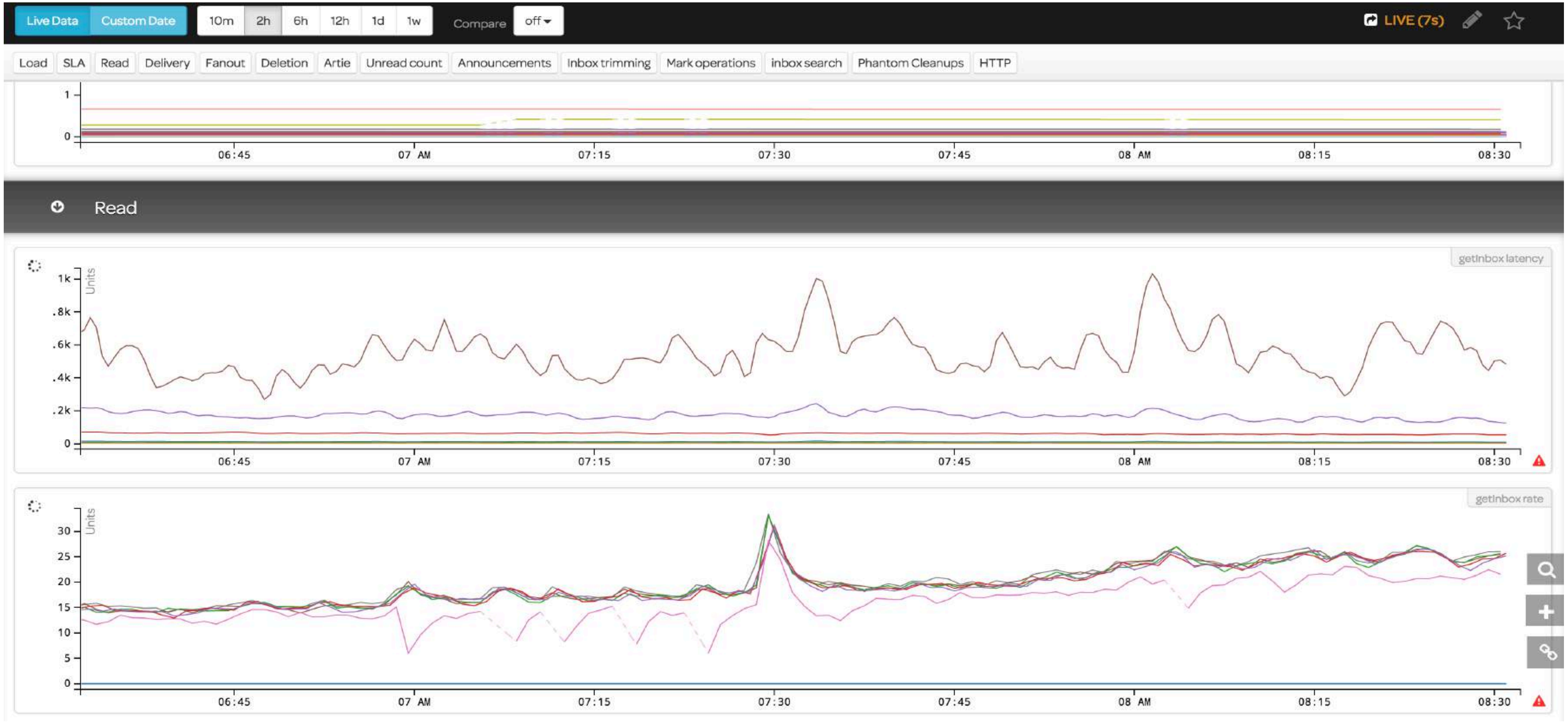
2015-06-02

Status	Name	Project	Version	Environment	Time
SUCCESS	jng	push-builder	20150602000624-18c8fd2-master	staging.bl2	00:06:28

2015-06-01

Status	Name	Project	Version	Environment	Time
FAILED	jng	push-builder	20150601235609-18c8fd2-master	staging.bl2	23:56:13
FAILED	mcaropreso	workfeed_production_dm2	20150601231148-89cf629-r567-06-	production_dm2	23:46:01
SUCCESS	cshellenbarger	deployer	20150601234547-624d9d0-master	production	23:45:52
SUCCESS	yammerdataci	avocado	20150601233811-6b806a4-jenkins-	production	23:38:29
SUCCESS	cshellenbarger	deployer	20150601233650-624d9d0-master	production	23:36:55
SUCCESS	mthompson	modulator	20150601233220-6b5a5ee-1.30	production	23:35:23
SUCCESS	mthompson	modulator	20150601233220-6b5a5ee-1.30	staging.bl2	23:33:33
SUCCESS	mcaropreso	workfeed	20150601231202-0449f35-master-6	staging.bl2	23:16:43
SUCCESS	sjain	sujay_onboarding	20150601231527-5db2141-1.4	stage	23:15:38
SUCCESS	mobile_ci	application_binaries_android	20150601231246-877f4d7-1.0.2015	staging	23:13:17
FAILED	sjain	sujay_onboarding	20150601230339-2d5a156-1.3	stage	23:03:50
SUCCESS	lcharteris	deployer	20150601225917-624d9d0-master	production	22:59:38
SUCCESS	pphatak	turbofan	20150601225616-385512b-0.0.111-	stage	22:56:45
SUCCESS	yammerdataci	avocado	20150601224036-6b806a4-jenkins-	production	22:41:01
SUCCESS	cnguyen	backupsclients	20150601223331-98f6e33-master	staging	22:34:22

Metrics



Live Data
Custom Date
10m
2h
6h
12h
1d
1w
Compare
off
LIVE (7s)
Load
SLA
Read
Delivery
Fanout
Deletion
Artie
Unread count
Announcements
Inbox trimming
Mark operations
inbox search
Phantom Cleanups
HTTP
Read
getInbox latency
Units
getInbox rate
06:45
07 AM
07:15
07:30
07:45
08 AM
08:15
08:30

What we wanted change and why

- Extract Inbox feature from an existing service, that powered all messaging feeds
- To enable faster iteration on Inbox
- Find an alternative to a legacy DB that:
  - expensive to scale,
  - had a bad support story
  - wasn’t great for a cross-DC setup

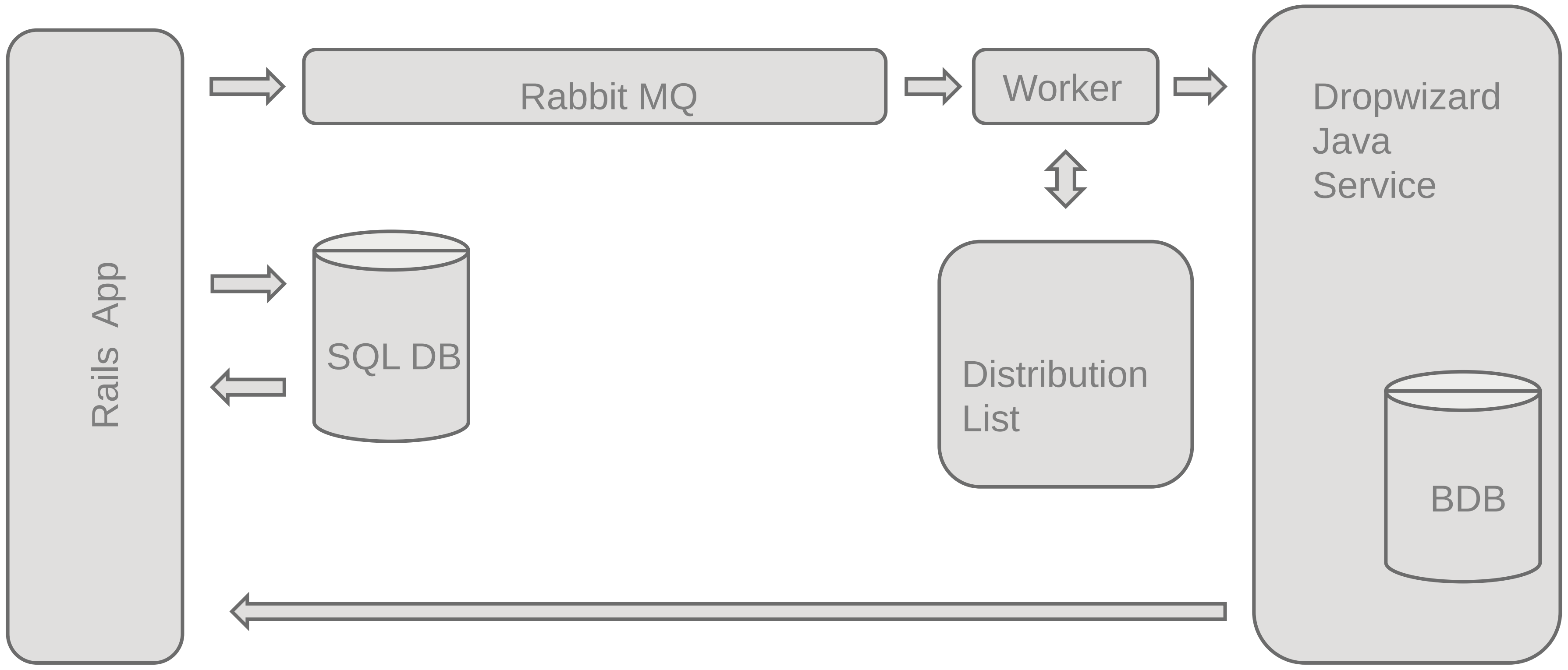
What we wanted change and why

- Extract Inbox feature from an existing service, that powered all messaging feeds
- To enable faster iteration on Inbox
- Find an alternative to a legacy DB that:
  - expensive to scale,
  - had a bad support story
  - wasn't great for a cross-DC setup

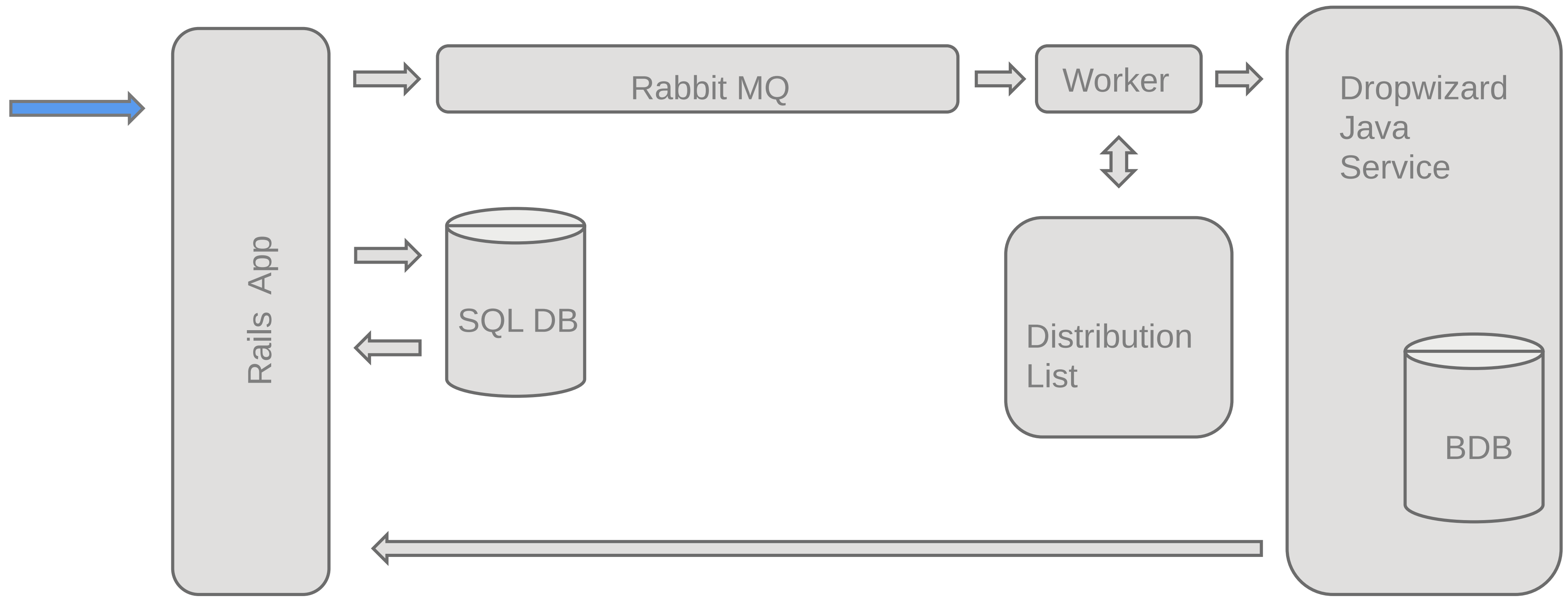
What we wanted change and why

- Extract Inbox feature from an existing service, that powered all messaging feeds
- To enable faster iteration on Inbox
- Find an alternative to a legacy DB that:
  - expensive to scale,
  - had a bad support story
  - wasn’t great for a cross-DC setup

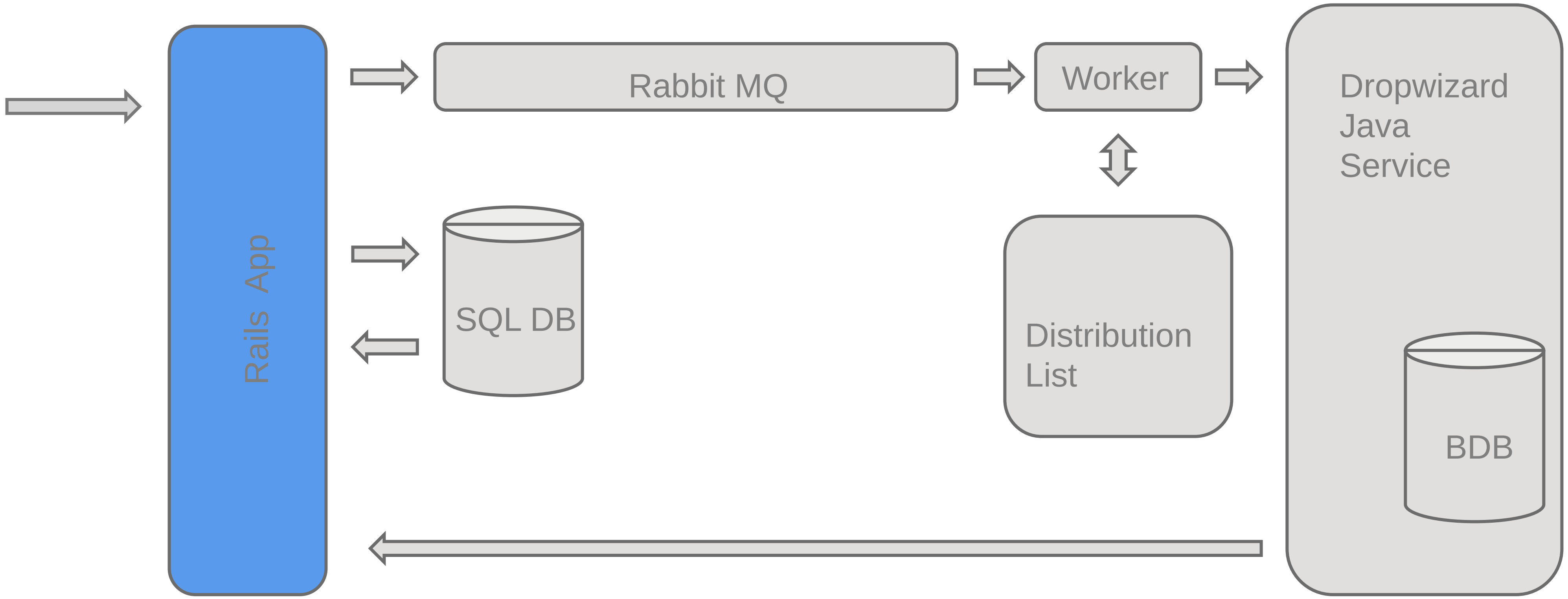
Overview


Rails App
Rabbit MQ
Worker
Dropwizard Java Service
SQL DB
Distribution List
BDB

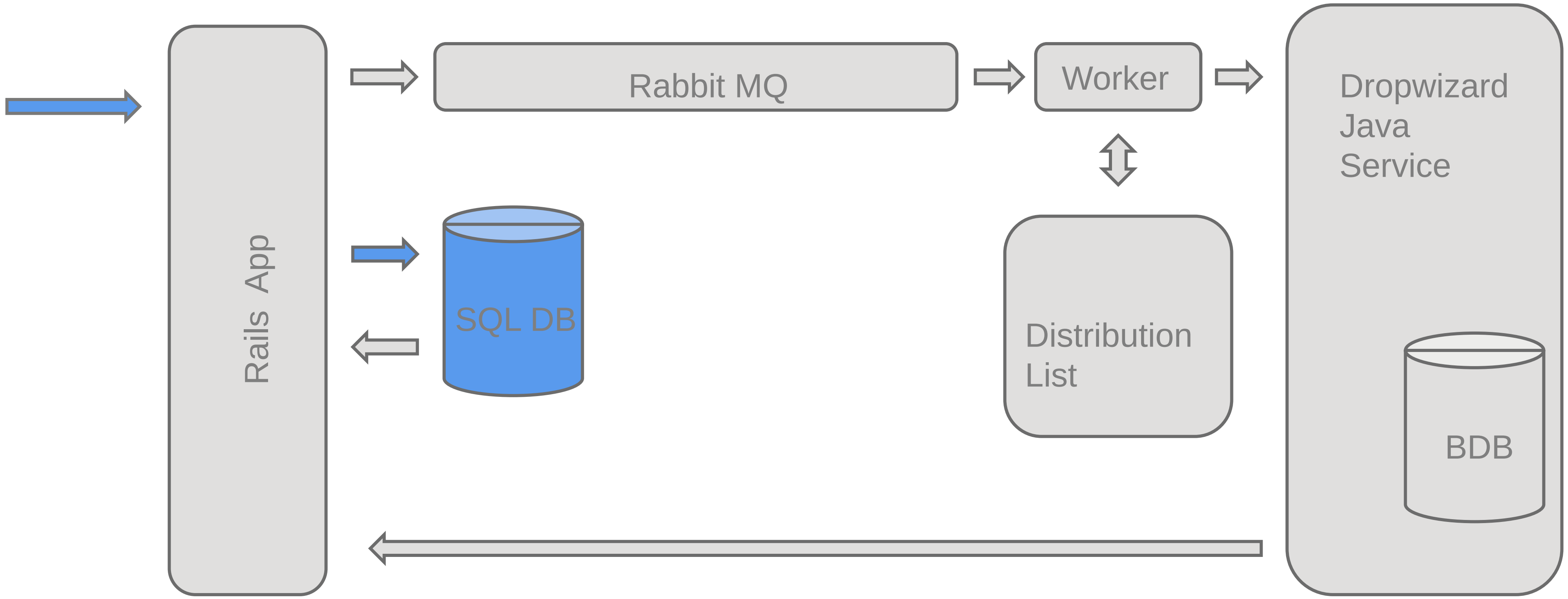
Write Path


Rails App
Rabbit MQ
Worker
Dropwizard Java Service
SQL DB
Distribution List
BDB

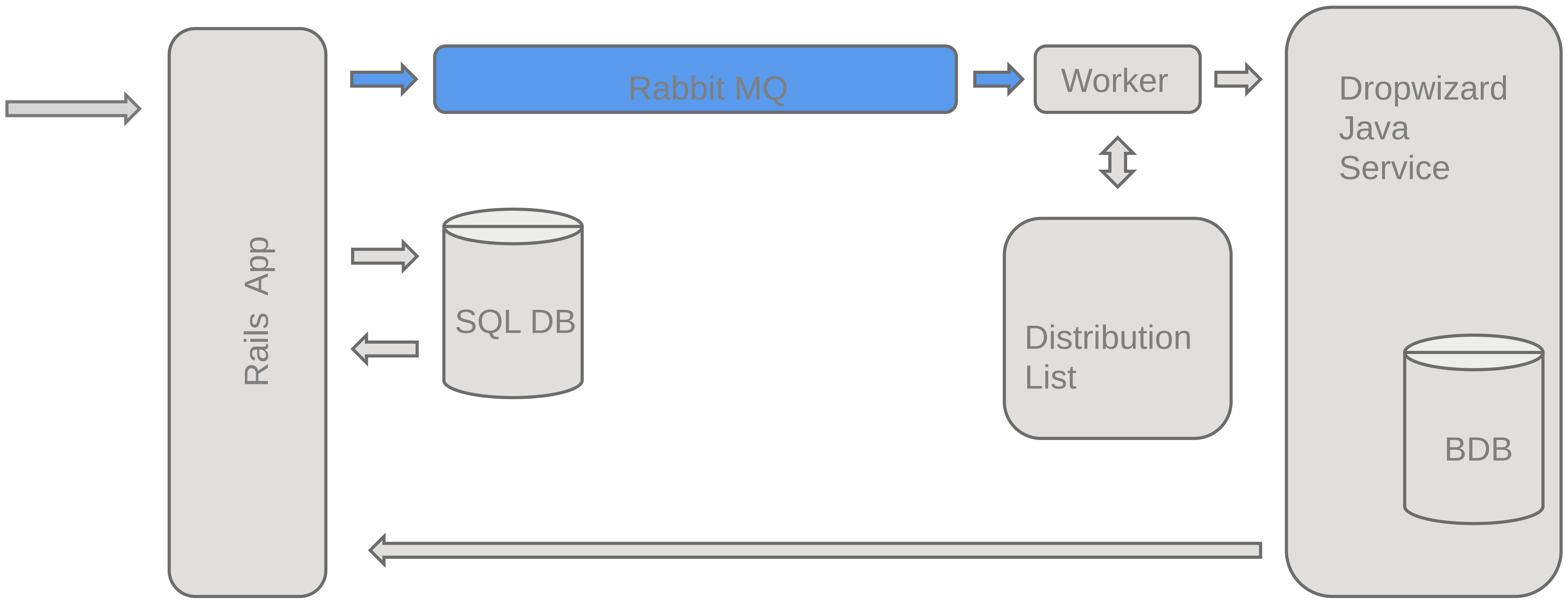
Write Path


Rails App
Rabbit MQ
Worker
Dropwizard Java Service
SQL DB
Distribution List
BDB

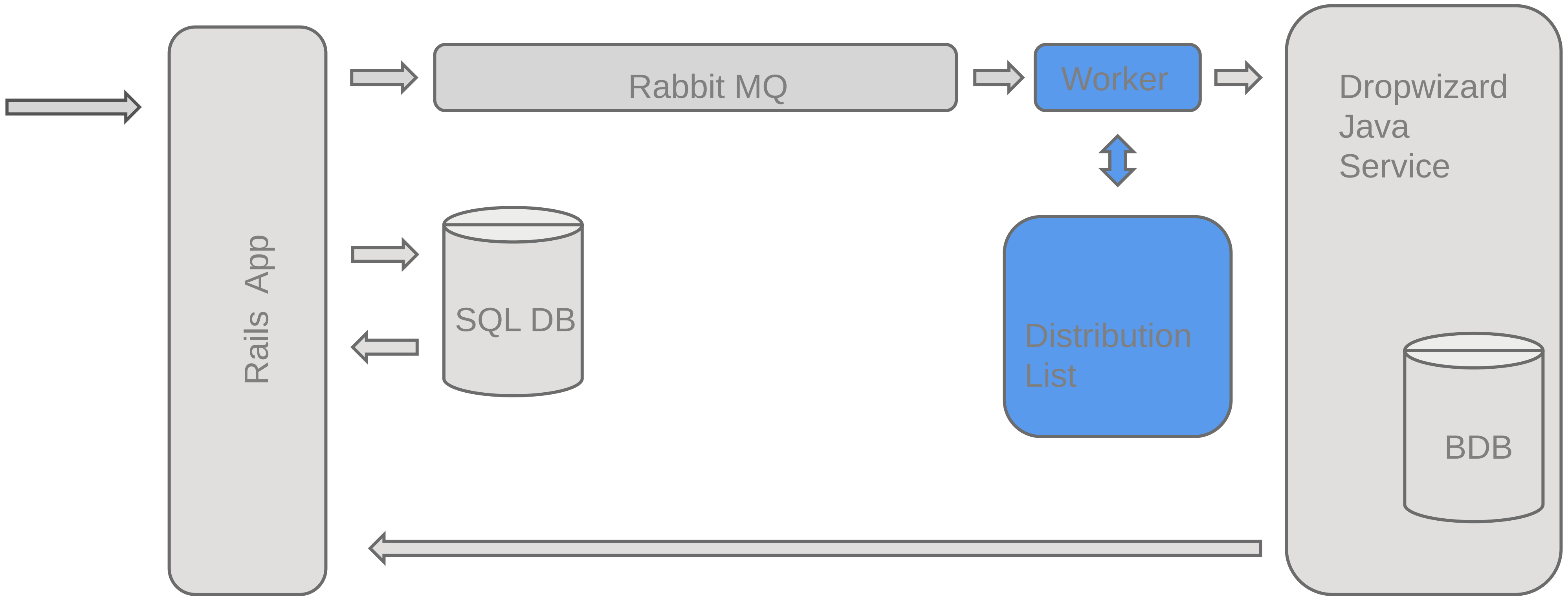
Write Path


Rails App
Rabbit MQ
Worker
Dropwizard Java Service
SQL DB
Distribution List
BDB

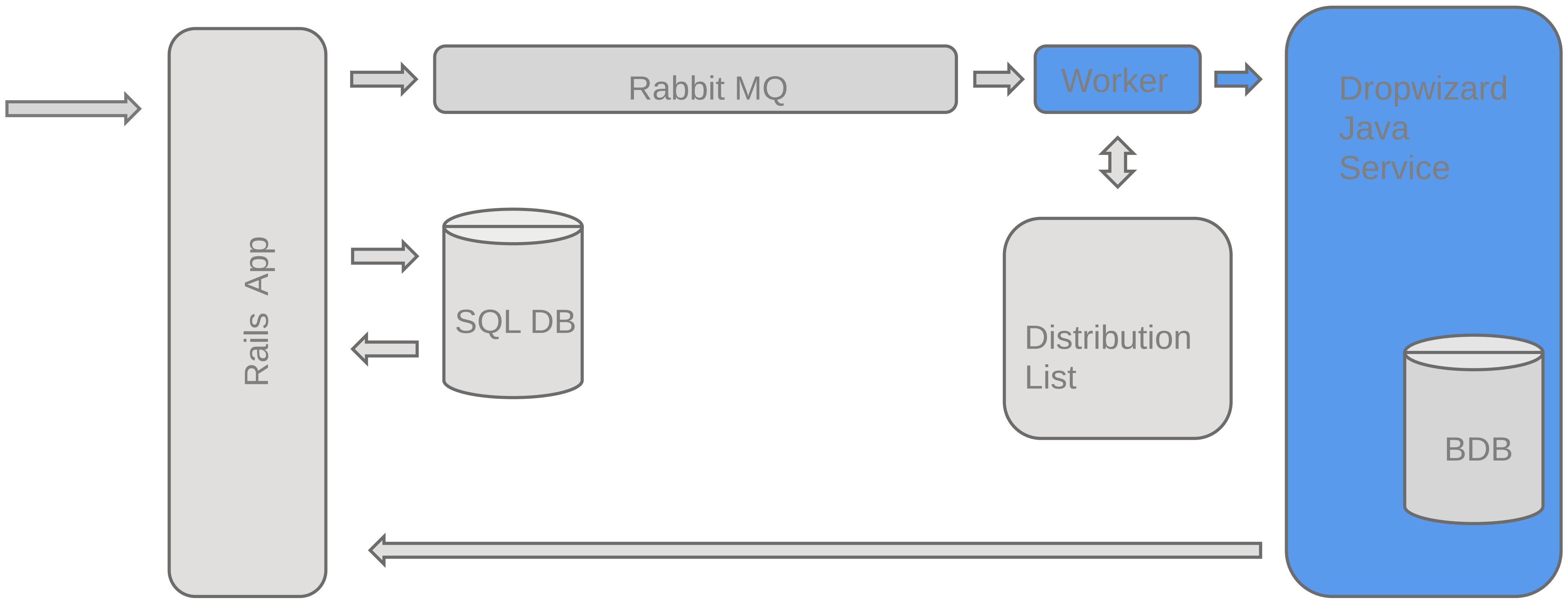
Write Path


Rails App
Rabbit MQ
Worker
Dropwizard Java Service
SQL DB
Distribution List
BDB

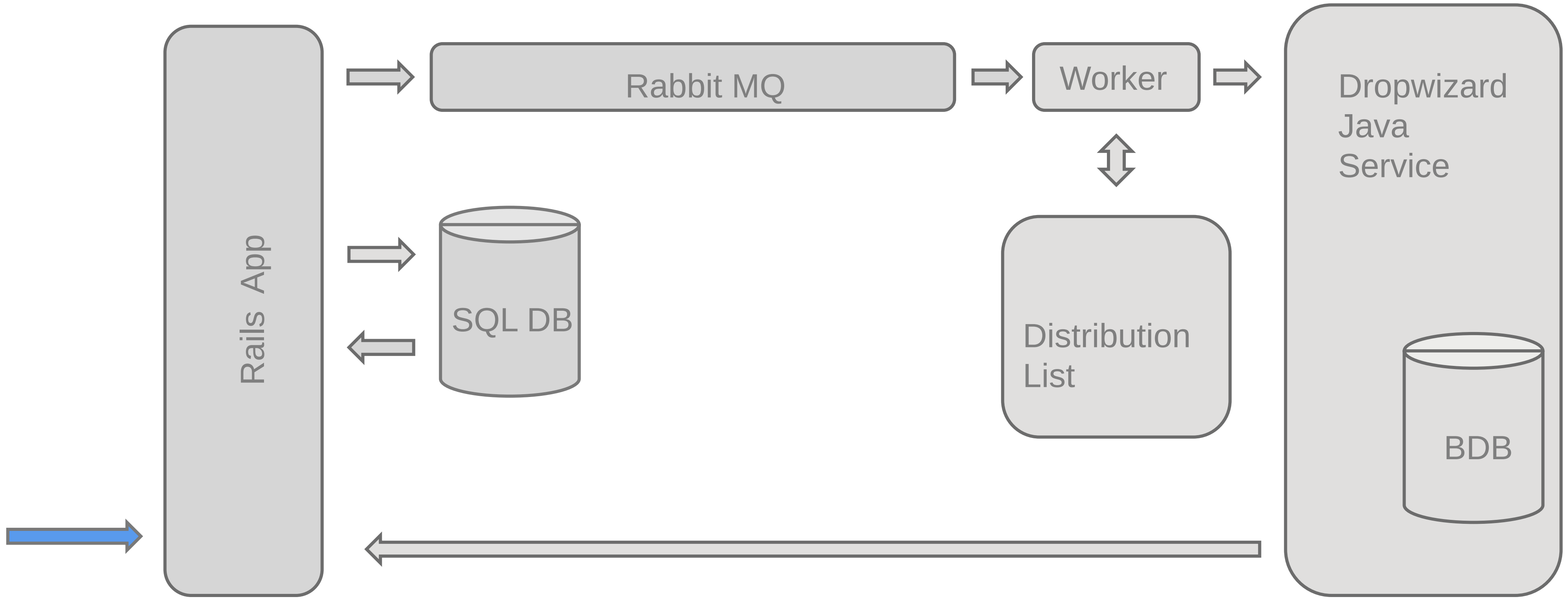
Write Path


Rails App
Rabbit MQ
Worker
Dropwizard Java Service
SQL DB
Distribution List
BDB

Write Path

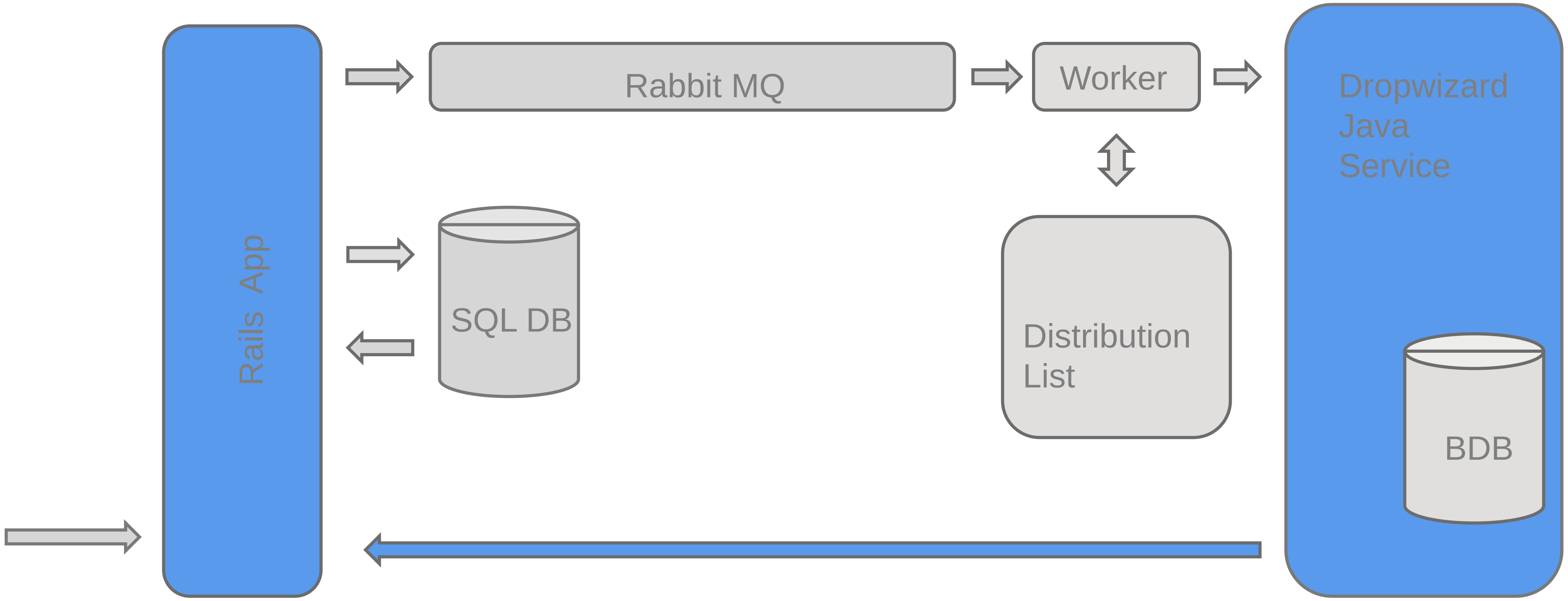

Rails App
Rabbit MQ
Worker
Dropwizard Java Service
SQL DB
Distribution List
BDB

Read Path

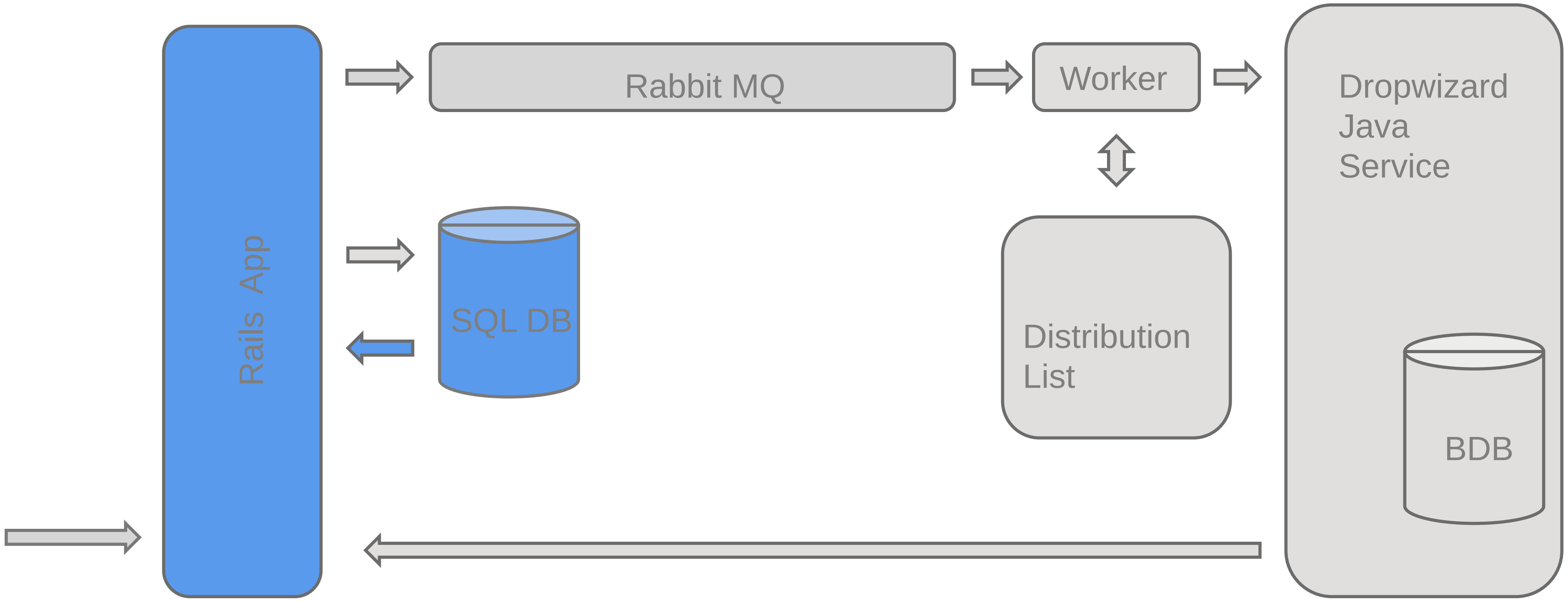


Rails App
Rabbit MQ
Worker
Dropwizard Java Service
SQL DB
Distribution List
BDB

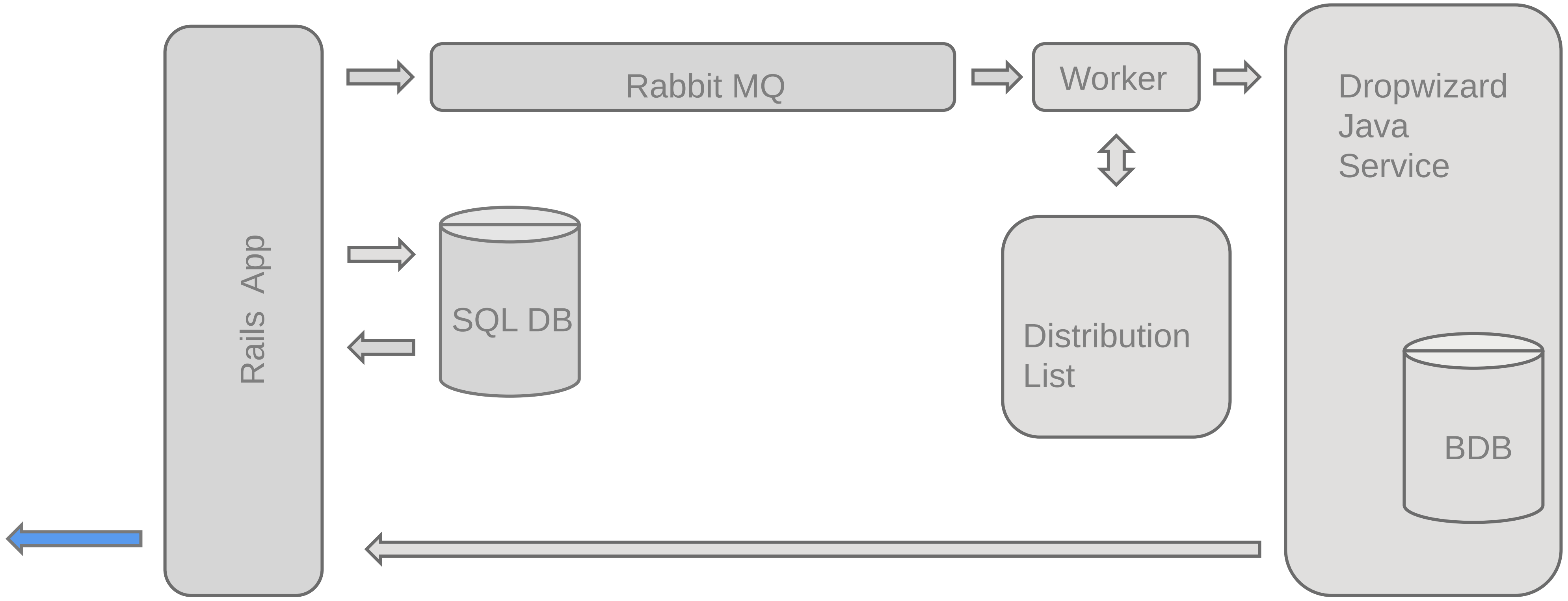
Read Path


Rails App
Rabbit MQ
Worker
Dropwizard Java Service
SQL DB
Distribution List
BDB

Read Path


Rails App
Rabbit MQ
Worker
Dropwizard Java Service
SQL DB
Distribution List
BDB

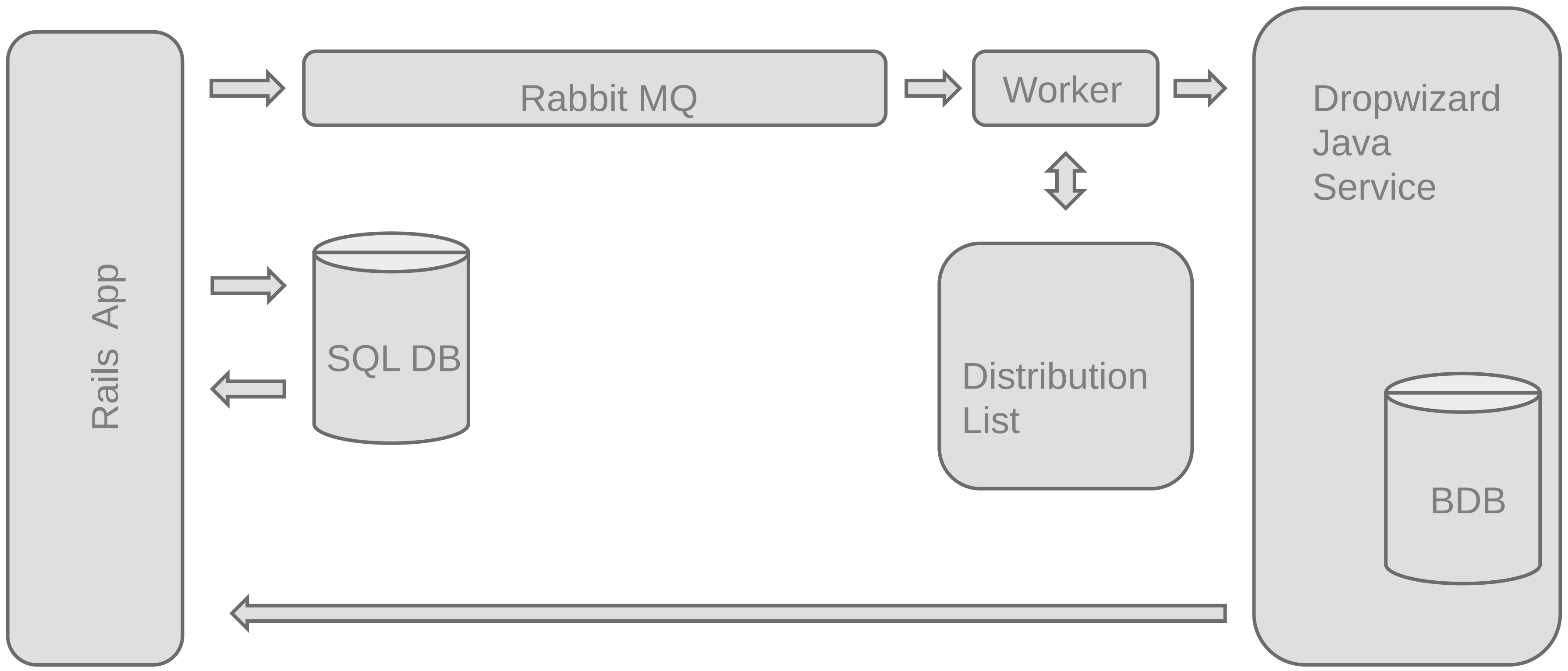
Read Path


Rails App
Rabbit MQ
Worker
Dropwizard Java Service
SQL DB
Distribution List
BDB

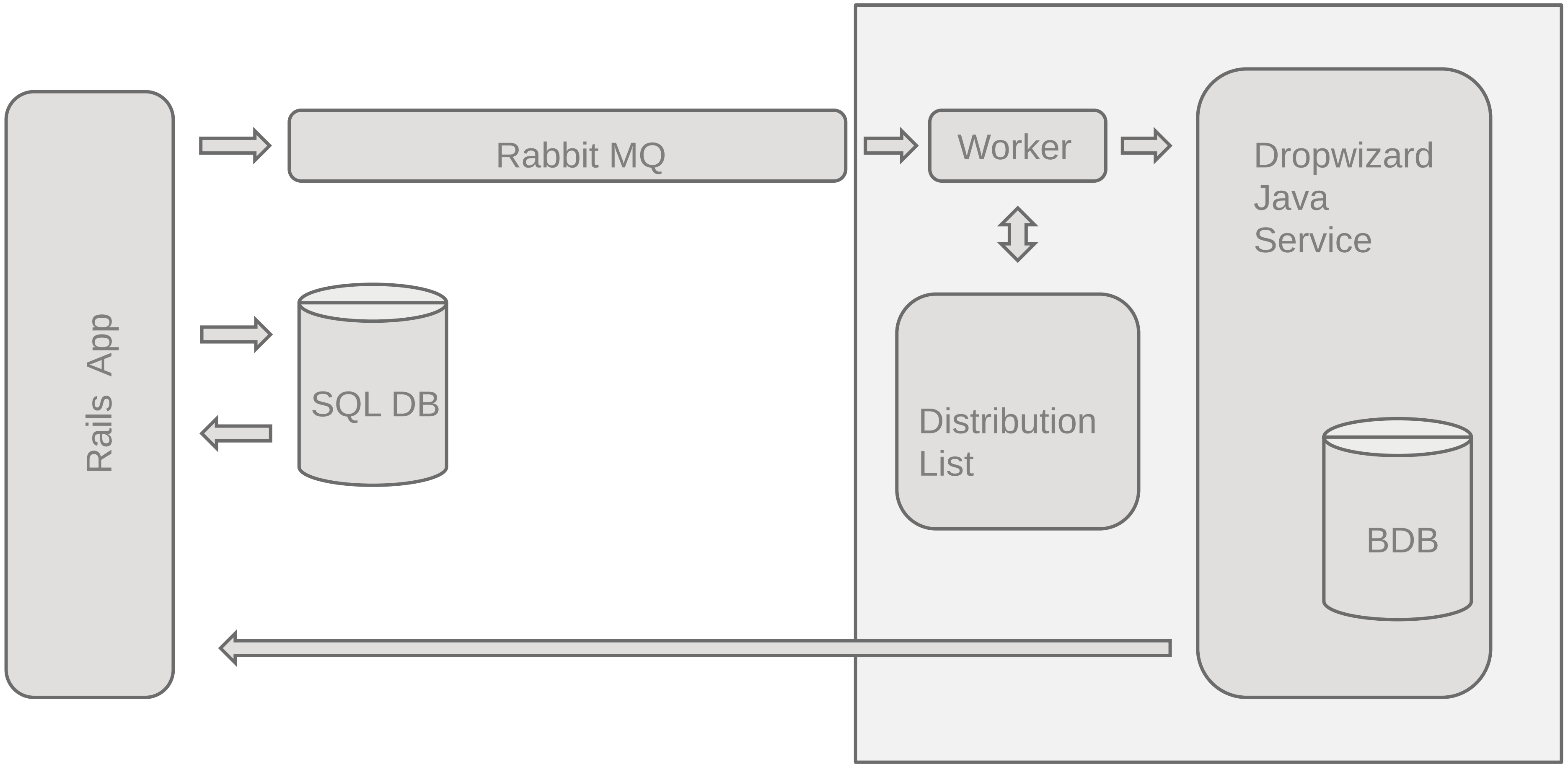
Goal

So what part of the system did we want to change and how?

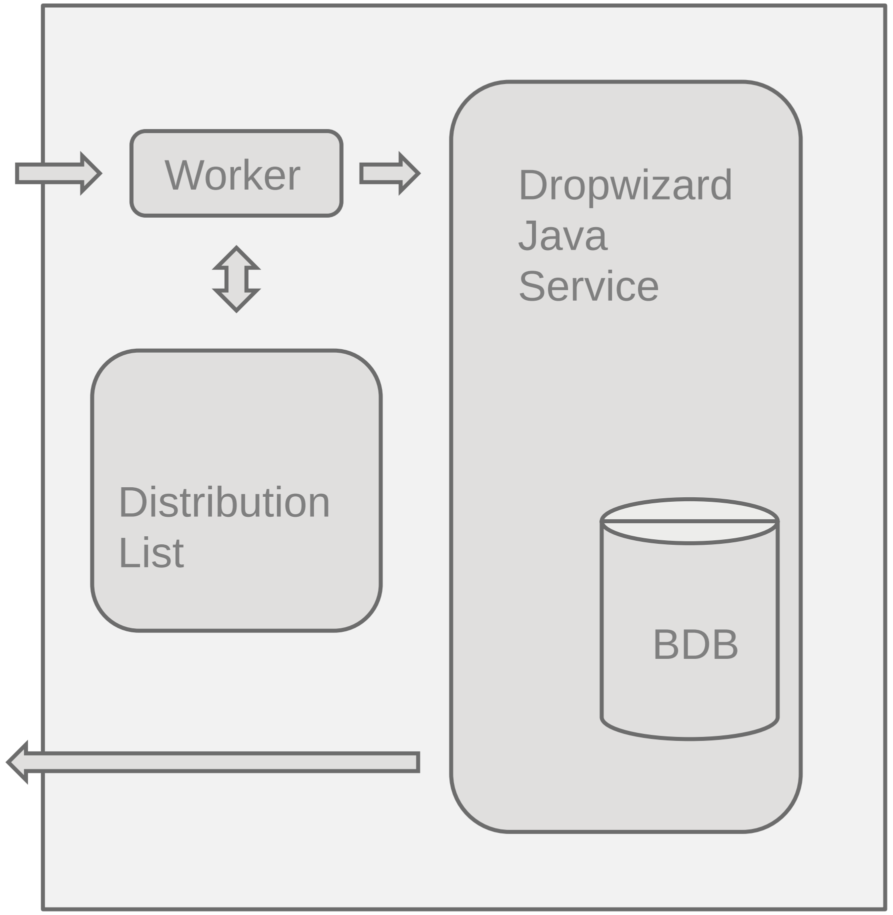
Goal


Rails App
Rabbit MQ
Worker
Dropwizard Java Service
SQL DB
Distribution List
BDB

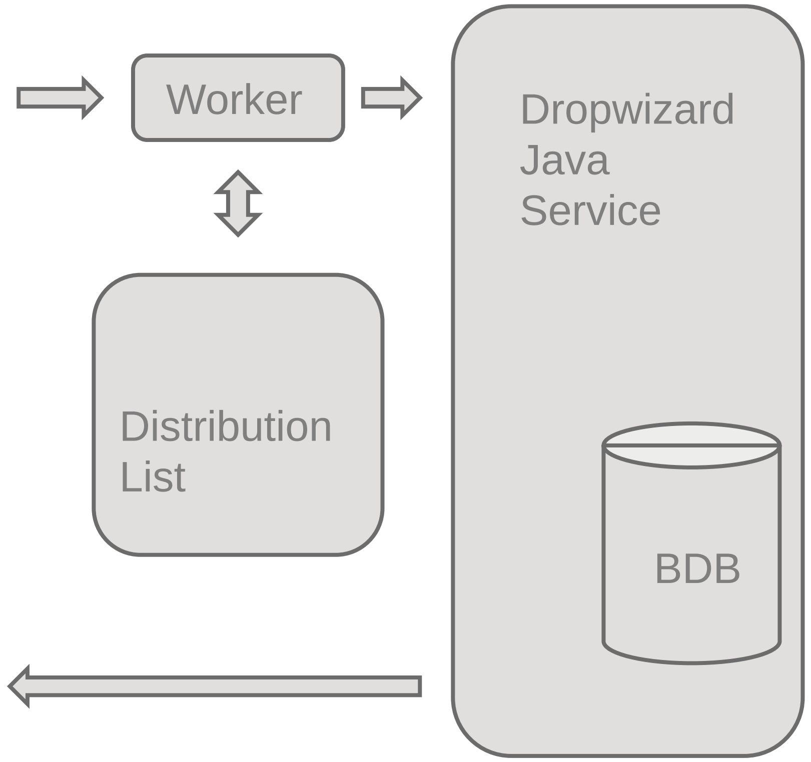
Goal


Rails App
Rabbit MQ
SQL DB
Worker
Distribution List
Dropwizard Java Service
BDB

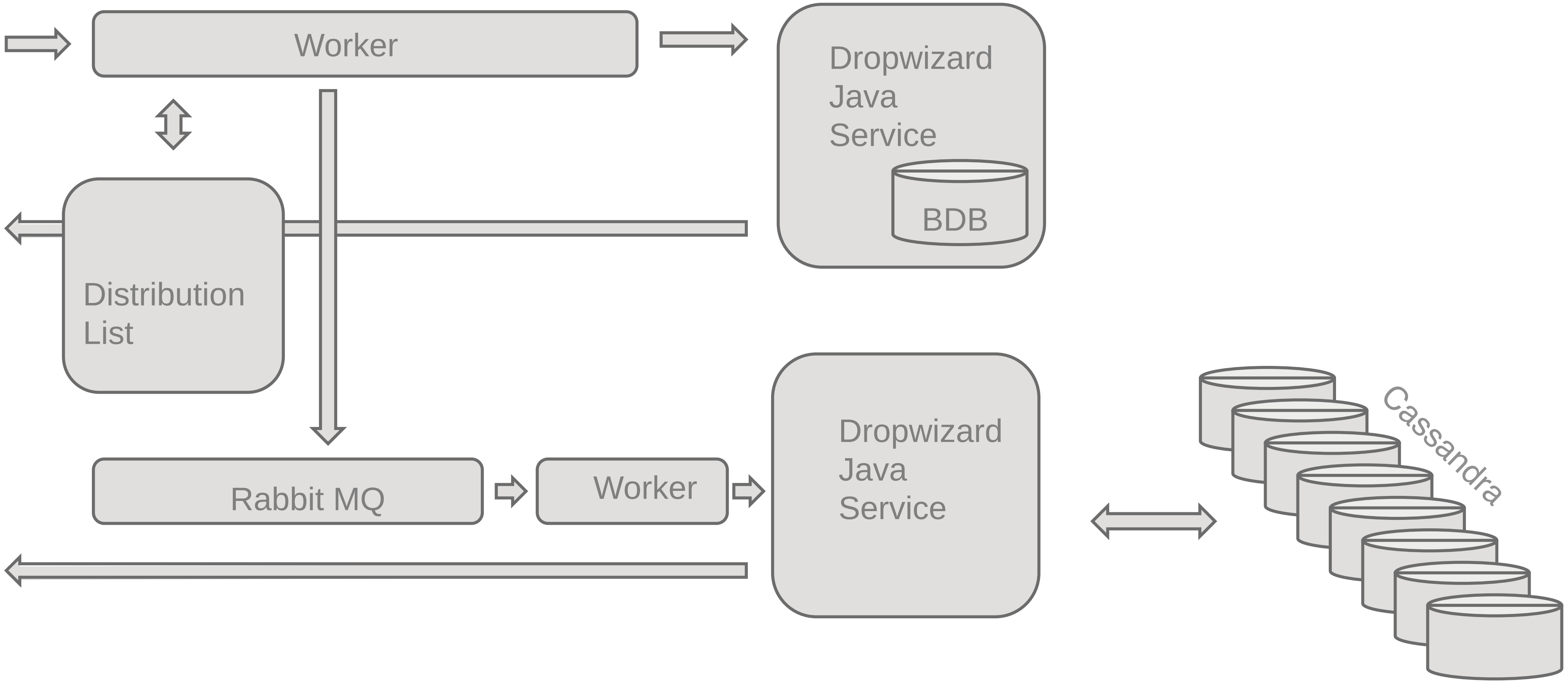
Goal



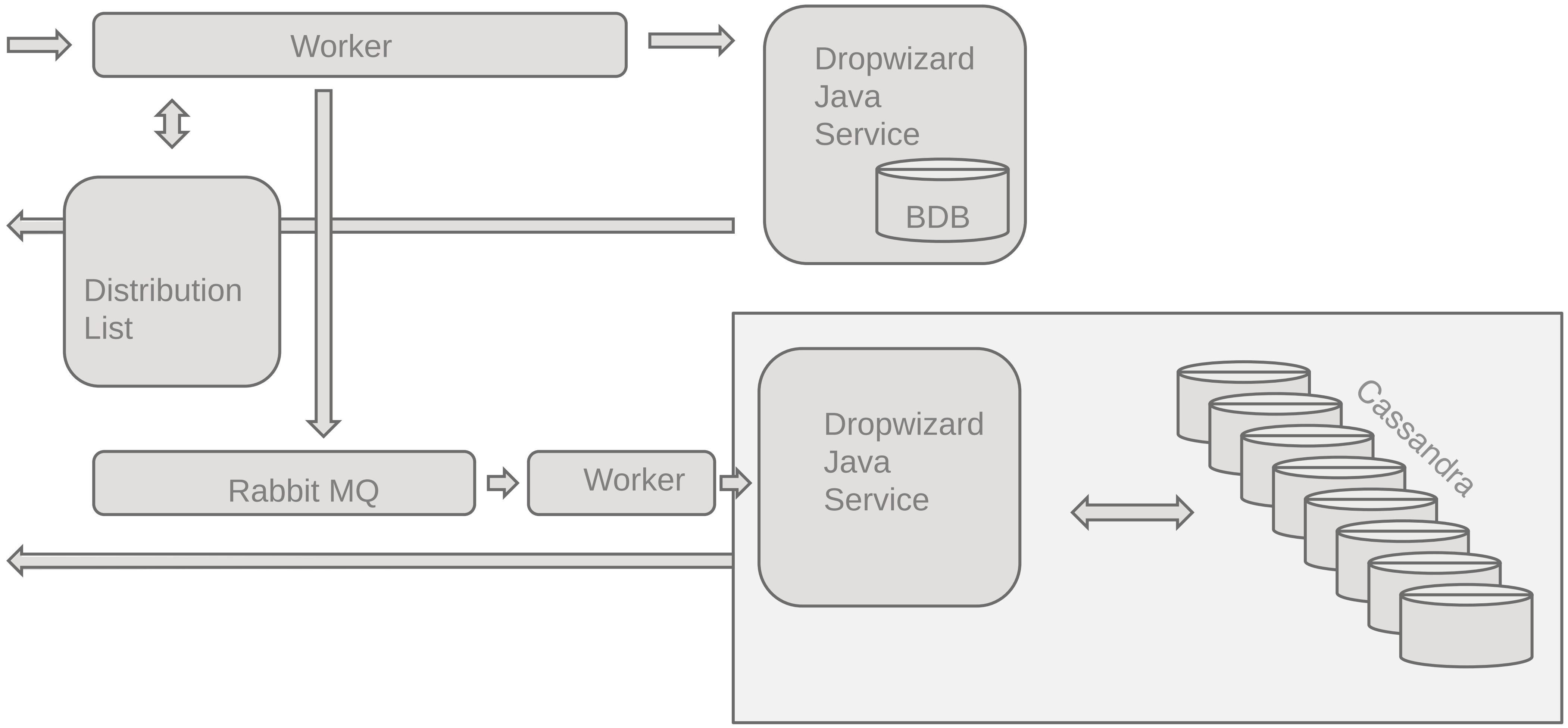
Goal



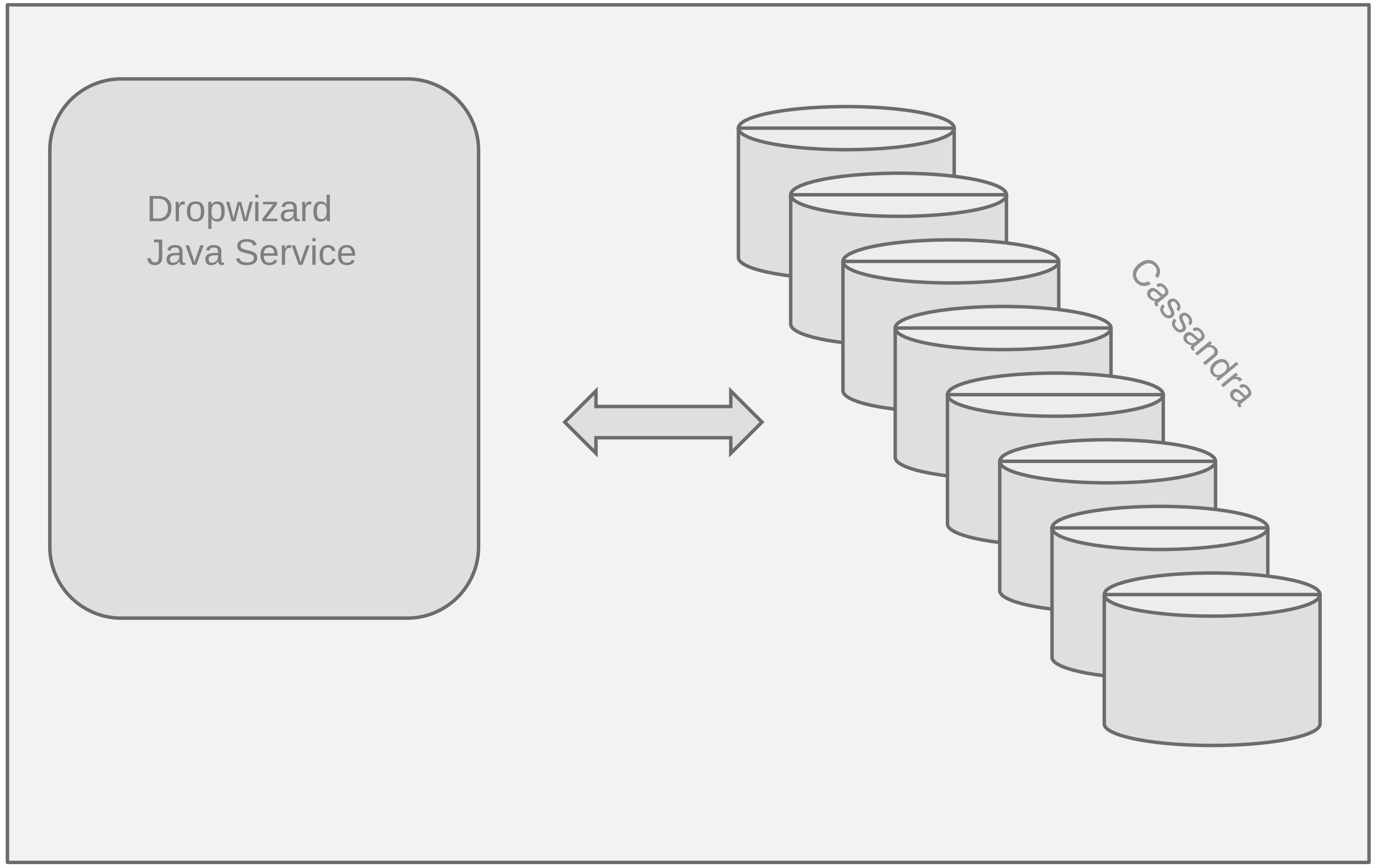
Goal


Worker
Dropwizard Java Service
BDB
Distribution List
Rabbit MQ
Worker
Dropwizard Java Service
Cassandra

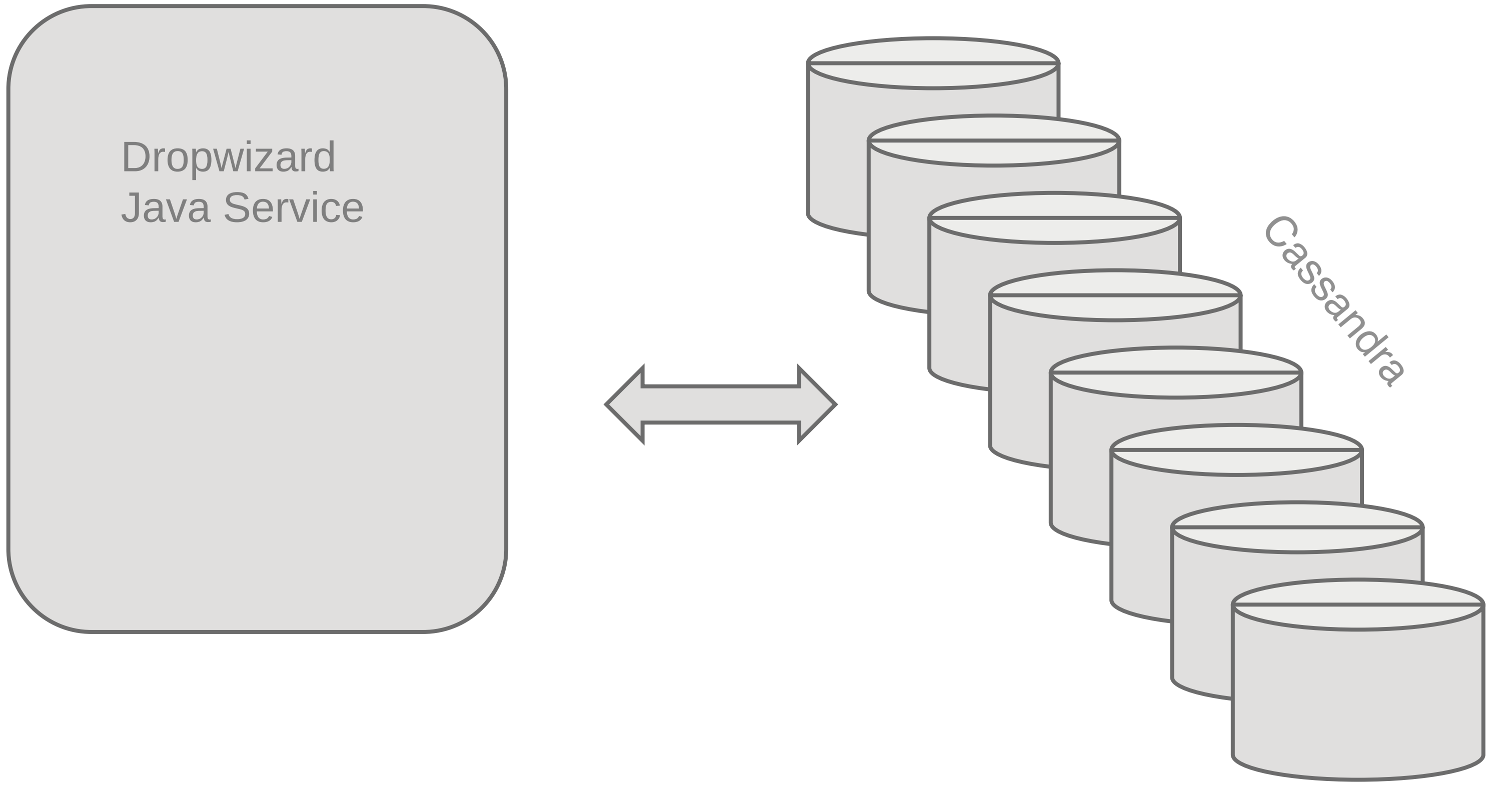
Goal


Worker
Dropwizard Java Service
BDB
Distribution List
Rabbit MQ
Worker
Dropwizard Java Service
Cassandra

Goal



Goal



Methodology

- Capture the API and semantics in integration tests
- Use production traffic to capacity plan and load test
  - shadow deploy and double dispatch
  - migrate data early
  - run verification tasks
- Assume we are going to make mistakes, so make data migration cheap:
  - bad modeling
  - missed use cases

Methodology

- Capture the API and semantics in integration tests
- Use production traffic to capacity plan and load test
  - shadow deploy and double dispatch
  - migrate data early
  - run verification tasks
- Assume we are going to make mistakes, so make data migration cheap:
  - bad modeling
  - missed use cases

Methodology

- Capture the API and semantics in integration tests
- Use production traffic to capacity plan and load test
  - shadow deploy and double dispatch
  - migrate data early
  - run verification tasks
- Assume we are going to make mistakes, so make data migration cheap:
  - bad modeling
  - missed use cases

What we knew

- Inbox is read heavy: 500mln requests/day
- We fan-out on write:
  - 50mln individual user deliveries/day
  - “announcements spikes” of up to 300K deliveries from one msg
- We needed tech that will be good for reads, but could also provide RT delivery in face of massive fan-outs.

What we knew

- Inbox is read heavy: 500mln requests/day
- We fan-out on write:
  - 50mln individual user deliveries/day
  - “announcements spikes” of up to 300K deliveries from one msg
- We needed tech that will be good for reads, but could also provide RT delivery in face of massive fan-outs.

What we knew

- Inbox is read heavy: 500mln requests/day
- We fan-out on write:
  - 50mln individual user deliveries/day
  - “announcements spikes” of up to 300K deliveries from one msg
- We needed tech that will be good for reads, but could also provide RT delivery in face of massive fan-outs.

First Phase – Choose the DB

We considered Riak and Cassandra as both:

- are sharded and replicated,
- work well cross-DC, and
- have a support story

We chose Cassandra over Riak because it did not force us to do a Read-Modify-Write of the whole inbox on message delivery.

First Phase – Choose the DB

We considered Riak and Cassandra as both:

- are sharded and replicated,
- work well cross-DC, and
- have a support story

We chose Cassandra over Riak because it did not force us to do a Read-Modify-Write of the whole inbox on message delivery.

Second Phase

Get something working!

- Provision Hardware
- Decide on a RESTful service API
- Get a build that tests the API and hits Cassandra
- Start implementing against our data model

Inbox - how it works?

- Stores threads addressed/watched by the user
- Threads ordered by most recently replied to
- Thread contents isn’t actually stored in this service
- On message post:
  - we deliver the message to every inbox
  - this amounts to updating `last_message_id`
- On read:
  - paginate (most recent messages first)
  - filter (read/unread)

Inbox - how it works?

- Stores threads addressed/watched by the user
- Threads ordered by most recently replied to
- Thread contents isn’t actually stored in this service
- On message post:
  - we deliver the message to every inbox
  - this amounts to updating `last_message_id`
- On read:
  - paginate (most recent messages first)
  - filter (read/unread)

Inbox - how it works?

- Stores threads addressed/watched by the user
- Threads ordered by most recently replied to
- Thread contents isn't actually stored in this service
- On message post:
  - we deliver the message to every inbox
  - this amounts to updating `last_message_id`
- On read:
  - paginate (most recent messages first)
  - filter (read/unread)

Inbox - how it works?

- Stores threads addressed/watched by the user
- Threads ordered by most recently replied to
- Thread contents isn’t actually stored in this service
- On message post:
  - we deliver the message to every inbox
  - this amounts to updating `last_message_id`
- On read:
  - paginate (most recent messages first)
  - filter (read/unread)

Inbox - how it works?

- Stores threads addressed/watched by the user
- Threads ordered by most recently replied to
- Thread contents isn’t actually stored in this service
- On message post:
  - we deliver the message to every inbox
  - this amounts to updating `last_message_id`
- On read:
  - paginate (most recent messages first)
  - filter (read/unread)

First Design

Inbox Table:

- Partitioned by: `inbox_id`
- Primary keyed: `(inbox_id, last_message_id)`
- Secondary indices for filtering, e.g. `(is_read)`

Thread Table:

- Secondary Index partitioned by: `thread_id`
- Used for storing thread metadata needed for delivery
- Heavily used CRDT sets

First Design

Inbox Table:

- Partitioned by: `inbox_id`
- Primary keyed: `(inbox_id, last_message_id)`
- Secondary indices for filtering, e.g. (`is_read`)

Thread Table:

- Secondary Index partitioned by: `thread_id`
- Used for storing thread metadata needed for delivery
- Heavily used CRDT sets

First Design

It was great:

- All our tests were passing and we covered for all the edge cases
- It fitted well with our usage patterns
- It was self healing in the presence of out-of-order deliveries or system partitions

Except that…. it brought our migration task to a halt.

First Design

It was great:

- All our tests were passing and we covered for all the edge cases
- It fitted well with our usage patterns
- It was self healing in the presence of out-of-order deliveries or system partitions

Except that…. it brought our migration task to a halt.

First Design

It was great:

- All our tests were passing and we covered for all the edge cases
- It fitted well with our usage patterns
- It was self healing in the presence of out-of-order deliveries or system partitions

Except that…. it brought our migration task to a halt.

First Design – What went wrong

We discovered that:

- Secondary indices are slow as hell!
- CRDTs are OKish for small infrequently updated things, but not for our subscription lists.

Secondly:

- The cost of our conveniently sorted data was heavy reliance on deletes – a NO NO in Cassandra world.

First Design – What went wrong

We discovered that:

- Secondary indices are slow as hell!
- CRDTs are OKish for small infrequently updated things, but not for our subscription lists.

Secondly:

- The cost of our conveniently sorted data was heavy reliance on deletes – a NO NO in Cassandra world.

What to do now?

We expected that kind of thing - we were only just learning to use Cassandra and wanted to use prod traffic to benchmark our solutions.

What was important is that this did not affect our API and that the semantics were captured in integration tests.

We could use the tests and the metrics we had to quickly iterate on the implementation.

What to do now?

We expected that kind of thing - we were only just learning to use Cassandra and wanted to use prod traffic to benchmark our solutions.

What was important is that this did not affect our API and that the semantics were captured in integration tests.

We could use the tests and the metrics we had to quickly iterate on the implementation.

What to do now?

We expected that kind of thing - we were only just learning to use Cassandra and wanted to use prod traffic to benchmark our solutions.

What was important is that this did not affect our API and that the semantics were captured in integration tests.

We could use the tests and the metrics we had to quickly iterate on the implementation.

Second Design

- Forget all the Cassandra Extras and design around it’s strengths.
- Understand feature requirements better and leverage that in your model (analytics):
  - we do not need to hold all the data, just recent stuff (Search)
  - 5000 entries is only 75KB, and that covers 4 years for an active user.
  - We don’t need to be that exact

Second Design

- Forget all the Cassandra Extras and design around it’s strengths.
- Understand feature requirements better and leverage that in your model (analytics):
  - we do not need to hold all the data, just recent stuff (Search)
  - 5000 entries is only 75KB, and that covers 4 years for an active user.
  - We don’t need to be that exact

Second Design

Inbox Table:

- Partitioned by: `inbox_id`
- Primary keyed: `(inbox_id, thread_id)`
- Mutable metadata: (`is_read, last_message_id`)

Thread Table:

- Secondary Index partitioned by: `thread_id`
- Used for storing thread metadata needed for delivery

Second Design

Inbox Table:

- Partitioned by: `inbox_id`
- Primary keyed: `(inbox_id, thread_id)`
- Mutable metadata: (`is_read, last_message_id`)

Thread Table:

- Secondary Index partitioned by: `thread_id`
- Used for storing thread metadata needed for delivery

Second Design

- This will be mutation heavy: use Leveled Compaction
- There will be races on updates:
  - on active threads it doesn't matter – we just order and filter
  - on less active ones, races are negligible and users can correct
- On read:
  - read the whole inbox, sort and then filter
  - data is small, so it will be actually OK
  - trim excess data (delete)

Second Design

- This will be mutation heavy: use Leveled Compaction
- There will be races on updates:
  - on active threads it doesn't matter – we just order and filter
  - on less active ones, races are negligible and users can correct
- On read:
  - read the whole inbox, sort and then filter
  - data is small, so it will be actually OK
  - trim excess data (delete)

Second Design

- This will be mutation heavy: use Leveled Compaction
- There will be races on updates:
  - on active threads it doesn’t matter – we just order and filter
  - on less active ones, races are negligible and users can correct
- On read:
  - read the whole inbox, sort and then filter
  - data is small, so it will be actually OK
  - trim excess data (delete)

Second Design

This time we got to production and passed the migration step.

It was even working in the shadow mode, modulo some data inconsistencies from missed use cases.

However… read performance was very varied, and below our expectations.

Second Design

This time we got to production and passed the migration step.

It was even working in the shadow mode, modulo some data inconsistencies from missed use cases.

However… read performance was very varied, and below our expectations.

Second Design

This time we got to production and passed the migration step.

It was even working in the shadow mode, modulo some data inconsistencies from missed use cases.

However… read performance was very varied, and below our expectations.

What now?

Look at usage metrics in more detail.

Out of the 500mln queries we see a day more than 450mln are for an unread count.

Actually, this happens to be a very small value:
- P95: < 100
- P999 < 1000

Materialize this query in a separate table (just unread stuff)

What now?

Look at usage metrics in more detail.

Out of the 500mln queries we see a day more than 450mln are for an unread count.

Actually, this happens to be a very small value:

- P95: < 100
- P999 < 1000

Materialize this query in a separate table (just unread stuff)

What now?

Look at usage metrics in more detail.

Out of the 500mln queries we see a day more than 450mln are for an unread count.

Actually, this happens to be a very small value:

- P95: < 100
- P999 < 1000

Materialize this query in a separate table (just unread stuff)

What now?

Look at usage metrics in more detail.

Out of the 500mln queries we see a day more than 450mln are for an unread count.

Actually, this happens to be a very small value:
- P95: < 100
- P999 < 1000

Materialize this query in a separate table (just unread stuff)

Where did this get us?

Write latency of:

- < 100ms for regular messages
- < 10s for the massive spikey announcements

Read Latency of:

- P99 < 250ms, P999 < 500ms – for the whole inbox
- P999 < 30ms – for unread count

Where did this get us?

Write latency of:

- < 100ms for regular messages
- < 10s for the massive spikey announcements

Read Latency of:

- P99 < 250ms, P999 < 500ms – for the whole inbox
- P999 < 30ms – for unread count

We shipped! – Thank You

Any Questions?

Not so fast!

Free months later, on the first day of my summer holiday!

Yammer is down!

- The site is down
- Our service is on fire!
- 3 Cassandra nodes are on fire

Not so fast!

Free months later, on the first day of my summer holiday!

Yammer is down!

- The site is down
- Our service is on fire!
- 3 Cassandra nodes are on fire

What went wrong?

Turns out having an HA service and an HA DB doesn’t make your site HA!

At the root of it was a massive inbox that was receiving tons of updates but was never read. This meant never trimmed.

Leveled compaction didn’t like it!

What went wrong?

Turns out having an HA service and an HA DB doesn’t make your site HA!

At the root of it was a massive inbox that was receiving tons of updates but was never read. This meant never trimmed.

Leveled compaction didn’t like it!

What went wrong?

Turns out having an HA service and an HA DB doesn’t make your site HA!

At the root of it was a massive inbox that was receiving tons of updates but was never read. This meant never trimmed.

Leveled compaction didn’t like it!

But there were deeper problems

- Our trimming strategy didn’t account for overgrowing inboxes` impact on compaction.
- Our main app didn’t have circuit breakers and timeouts
- The service itself didn’t control it’s resources (timeouts/threads) to ensure HA.

But there were deeper problems

- Our trimming strategy didn’t account for overgrowing inboxes` impact on compaction.
- Our main app didn’t have circuit breakers and timeouts
- The service itself didn’t control it’s resources (timeouts/ threads) to ensure HA.

But there were deeper problems

- Our trimming strategy didn’t account for overgrowing inboxes` impact on compaction.
- Our main app didn’t have circuit breakers and timeouts
- The service itself didn’t control it’s resources (timeouts/ threads) to ensure HA.

Fixes

Cassandra

- Added probabilistic and async trimming on delivery
- Dropped `gc_grace_period` and increased repair frequency in favor of small but frequent ones.

Service

- Bulkheads: all logical service operations are time-bound and have individual threadpools to ensure capacity.

Application: rolled out circuit breakers

Fixes

Cassandra

- Added probabilistic and async trimming on delivery
- Dropped `gc_grace_period` and increased repair frequency in favor of small but frequent ones.

Service

- Bulkheads: all logical service operations are time-bound and have individual threadpools to ensure capacity.

Application: rolled out circuit breakers

Fixes

Cassandra

- Added probabilistic and async trimming on delivery
- Dropped `gc_grace_period` and increased repair frequency in favor of small but frequent ones.

Service

- Bulkheads: all logical service operations are time-bound and have individual threadpools to ensure capacity.

Application: rolled out circuit breakers

Is it fixed now?

We still have some pending tasks we are working on:

- ensuring repairs are successful
- ensuring a single bad node (not dead but very slow) doesn't take down the cluster

But importantly, with the current setup a problem in Cassandra:

- doesn't take the site down
- only users whose data is on problematic nodes are affected

Is it fixed now?

We still have some pending tasks we are working on:

- ensuring repairs are successful
- ensuring a single bad node (not dead but very slow) doesn’t take down the cluster

But importantly, with the current setup a problem in Cassandra:

- doesn’t take the site down
- only users whose data is on problematic nodes are affected

What worked well

We were able to iterate and fix problems very quickly:

- integration tests allowed us to ship to prod with confidence
- shadow deploy gave us great feedback on design
- existing metrics/analytics aided our design choices
- having an easy to run migration allowed us to quickly iterate on the data model

What we’ve learned

Even for a big organization introducing a technology bears a high cost:

- Getting a working model in prod took only 3 months
- Ironing out operations will take at least 1 year
  - Understanding the system
  - Firefighting and fixing
  - Training

Support helps, but the above still holds.

What we’ve learned

Even for a big organization introducing a technology bears a high cost:

- Getting a working model in prod took only 3 months
- Ironing out operations will take at least 1 year
  - Understanding the system
  - Firefighting and fixing
  - Training

Support helps, but the above still holds.

Thank you – this time for real

Any questions?