BLUEGROUND

Transactional Outbox

with a twist

Chris Aslanoglou - Software Engineer

Java Hellenic User Group (JHUG) - December 2022

The problem

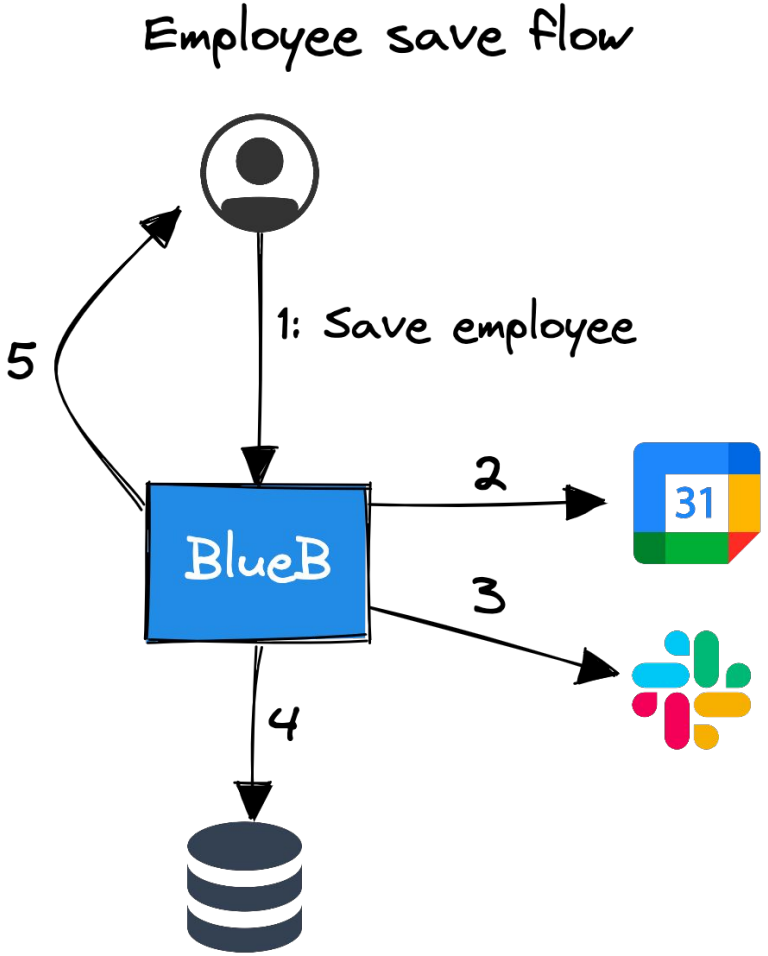
An example

Goal

1. Save an employee's details
2. Create new GCalendar
3. Fetch SlackId

What can go wrong? *famous last words*

- Transient network failures Google
- Same with Slack
- DB consistency failure => rollback
- *It's slooow*



Outbox pattern to the rescue

Outbox Pattern

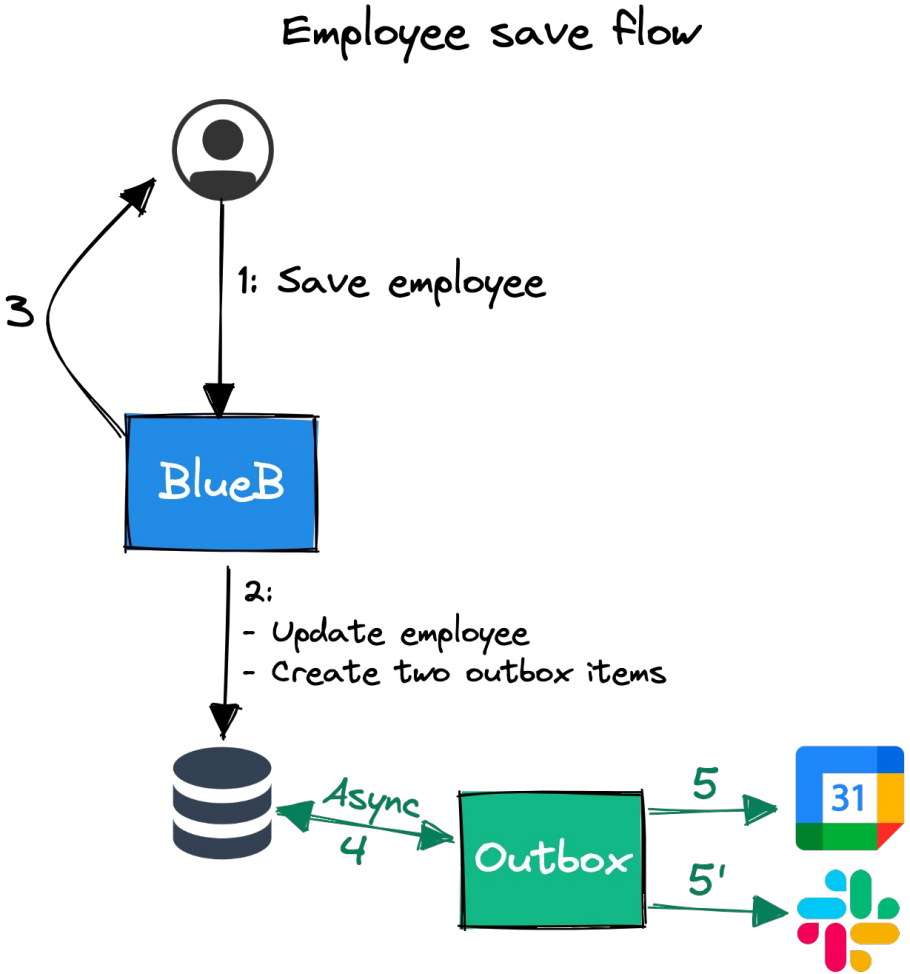
Q: What do we want to achieve?

A: Have our cake and *eventually* eat it

Well, kinda...

Reliably and atomically
update a DB record &
communicate with another service

Create a new Google calendar
if and only if
the “save employee” transaction succeeds



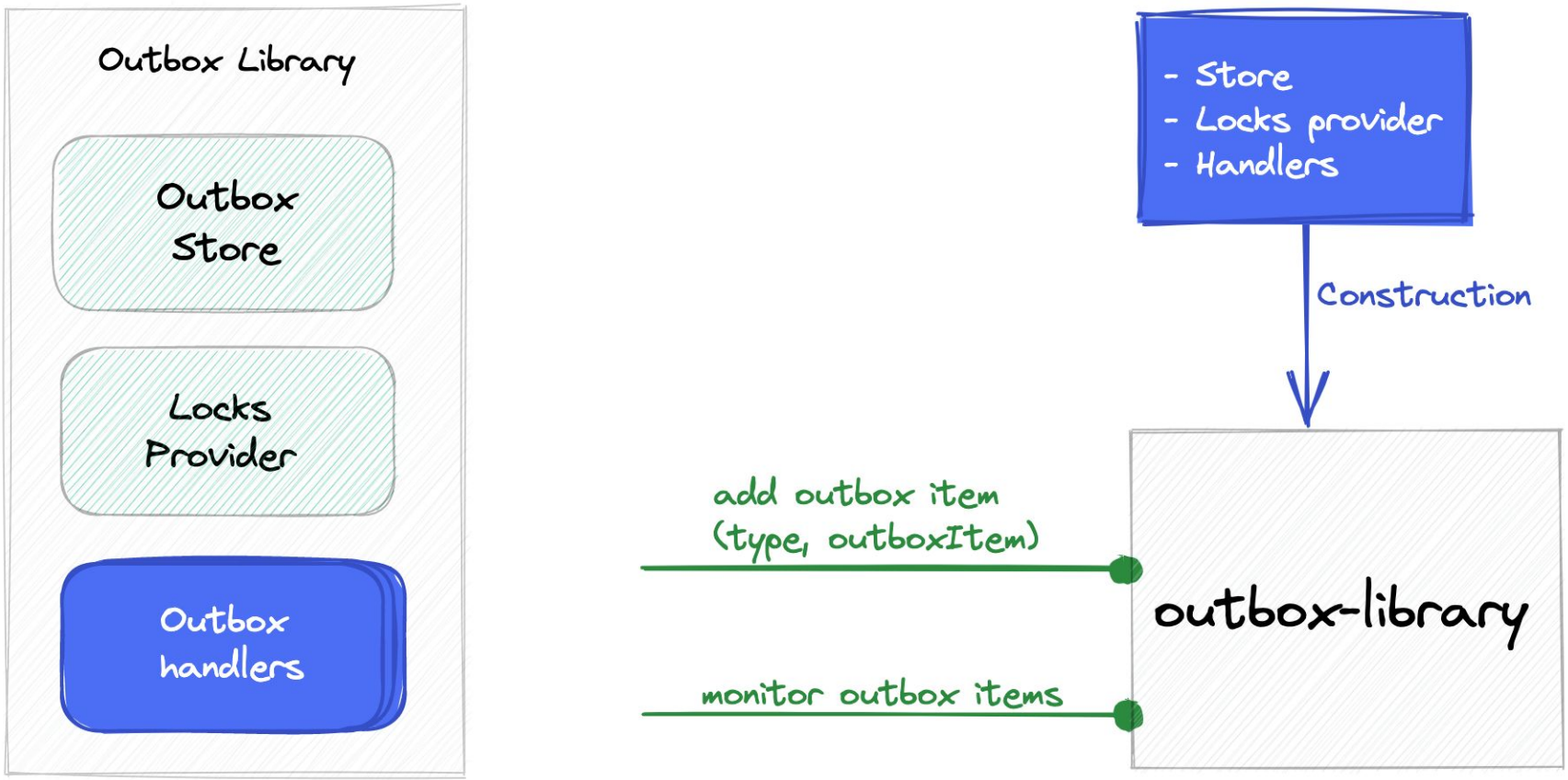
Our requirements

Our requirements

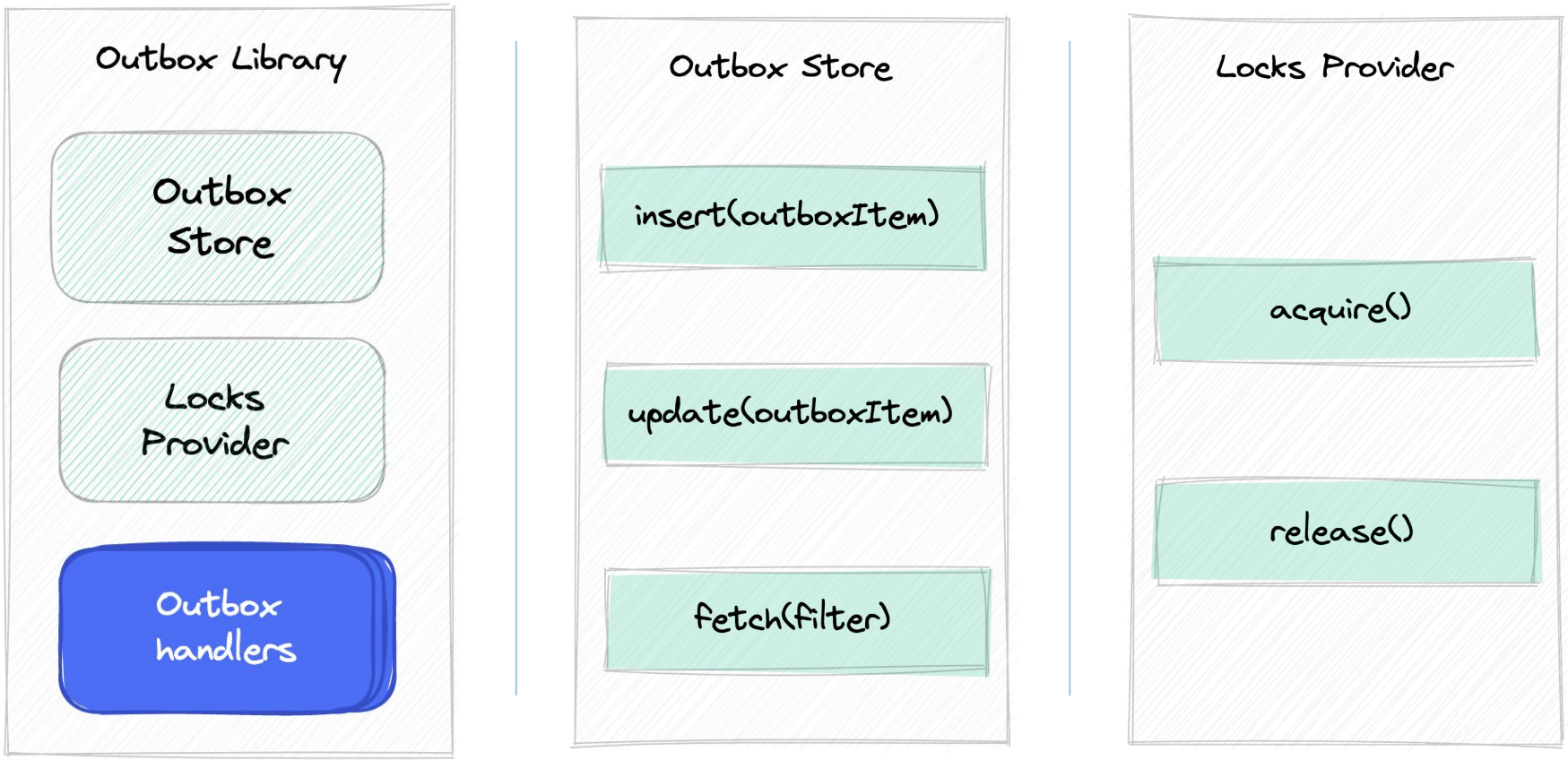
- A library to persist and handle generic outbox items
- At-least once processing semantics
- Per type failure handling granularity
- Agnostic to
  - Application framework
  - Data storage layer
  - Locks provider implementation

Our approach

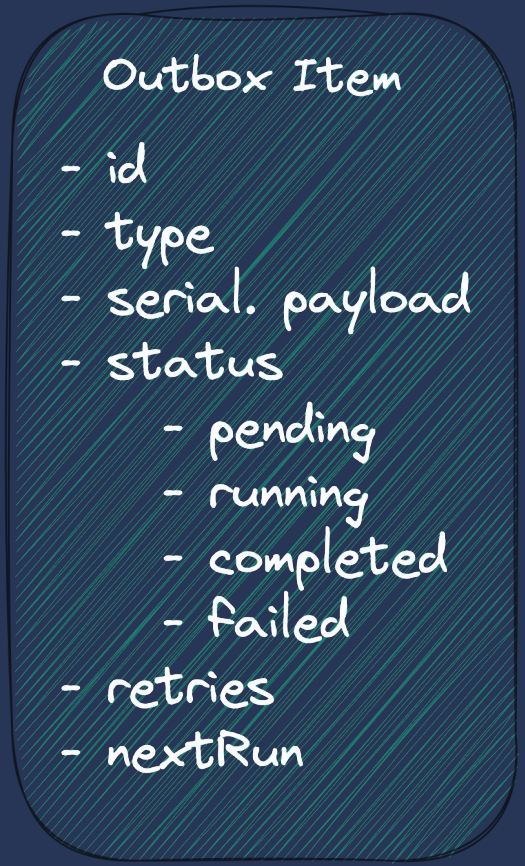
High Level Design - Core Library

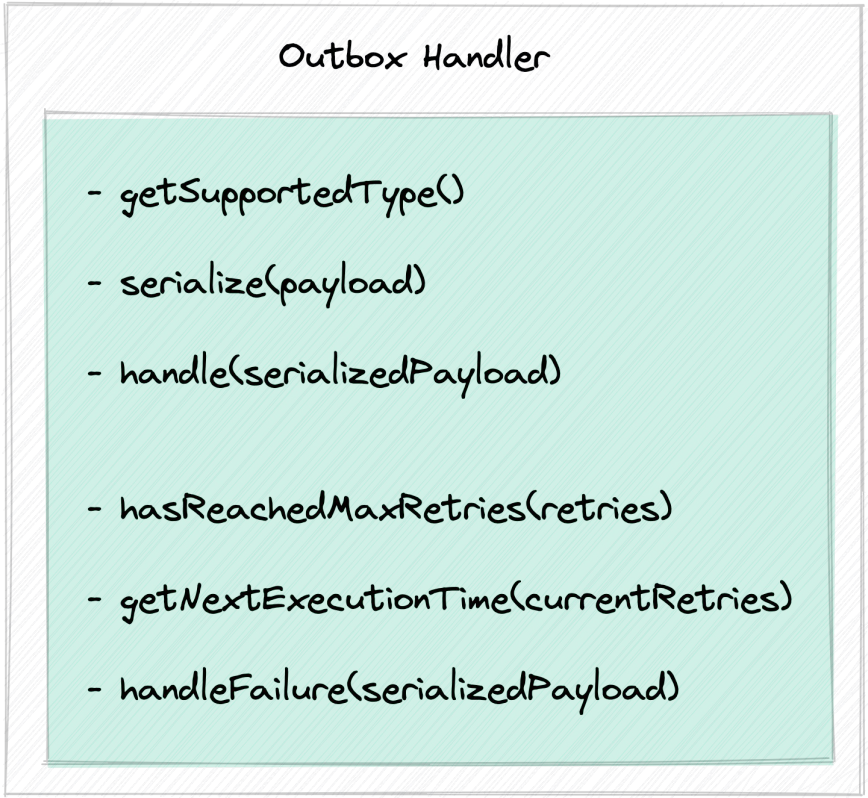


High Level Design - Required Interfaces



Handlers, Types, Items, o r, *how do I use this thing?*





Library instantiation

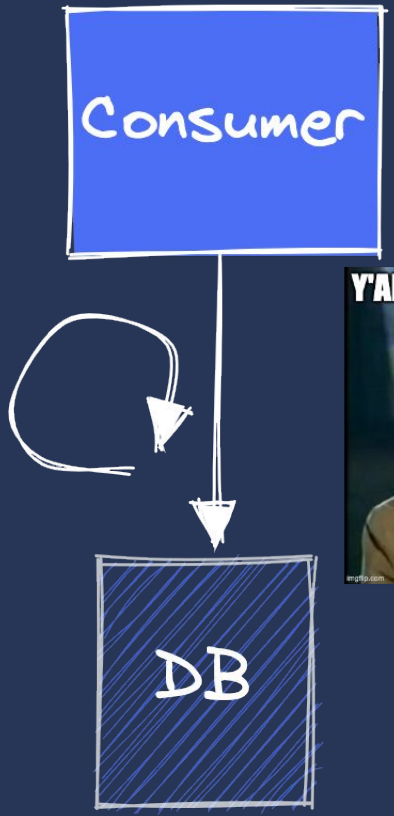
```
return TransactionalOutboxBuilder
    .make(clock)
    .withHandlers(outboxHandlers)
    .withLocksProvider(locksProvider)
    .withOutboxStore(outboxStore)
    .build()
```

Item addition

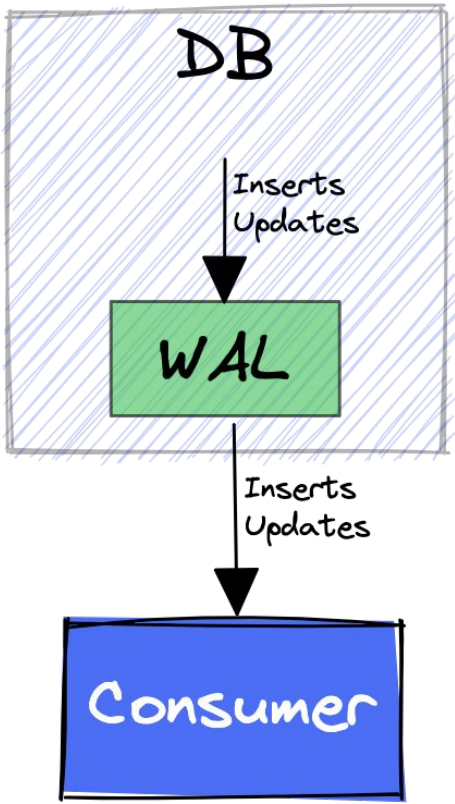
```
private void addGoogleCalendarOutboxItem(User user) {
  var payload = new GoogleCalendarCreationOutboxPayload(
      user.getId(),
      user.getFullname()
  );
  outbox.add(GOOGLE_CALENDAR_CREATE, payload);
}
```

Ok, we saved the items, now what?

Polling Consumer



Change-Data-Capture



Polling consumer call

```
@Scheduled(cron = "\${app.outbox.cron}")
@Transactional
fun run() {
  outbox.monitor()
}
```

Polling Consumer Outline

```
1. fun monitor() {
2.   locksProvider.acquire()
3.
4.   val items = store.fetch(eligibleFilter)
5.
6.   markForProcessing(items)
7.   items.map { store.update(it) }
8.
9.   items.forEach { item ->
10.    executor.execute(
11.      OutboxItemProcessor(item, handlers[item.type], store)
12.    )
13.  }
14.
15.  locksProvider.release()
16. }
```

Polling Consumer Outline

```
1. fun monitor() {
2.   locksProvider.acquire()
3.
4.   val items = store.fetch(eligibleFilter)
5.
6.   markForProcessing(items)
7.   items.map { store.update(it) }
8.
9.   items.forEach {
10.    executor.execu
11.      OutboxItemPr
12.    )
13.  }
14.
15.  locksProvider.re
16. }
```

```
1. fun markForProcessing(items) {
2.   items.map {
3.     it.status = OutboxStatus.RUNNING
4.     val now = Instant.now(clock)
5.     it.lastExecution = now
6.     it.rerunAfter = now.plus(rerunAfterDuration)
7.   }
8. }
```

OutboxItemProcessor (thread) Outline

```
1. fun run() {
2.   try {
3.     handler.handle(item.payload)
4.     item.status = OutboxStatus.COMPLETED
5.
6.   } catch (exception: Exception) {
7.     if (handler.hasReachedMaxRetries(item.retries) {
8.       handleTerminalFailure(exception)
9.     }
10.    else {
11.      handleRetryableFailure(exception)
12.    }
13.  } finally {
14.    store.update(item)
15.  }
16. }

// handleTerminalFailure ⇒ item.status = FAILED && handler.handleFailure(item.payload)

// handleRetryableFailure ⇒
//   1. item.nextRun = handler.getNextExecutionx(item.retries)
//   2. item.retries += 1
//   3. item.status = OutboxStatus.PENDING
```

Required API implementation

Store

- Simple data access layer abstraction
- **Insert**

  TX semantics of outbox.add caller
- **Fetch**

  Read only TX
- **Update**

  Requires new TX
  - Outbox item update committed before thread execution

    => Avoid concurrent updates
  - In thread: always commit in new TX

Locks Provider

- Requirements
  - Blocking exclusive locks
  - Automatic release (crash-failure)
- Implemented with PostgreSQL advisory locks
  - Lock semantics prescribed by application
  - DB session connection scoped
    - Released after connection is closed (crash-failure liveness)
  - Blocking

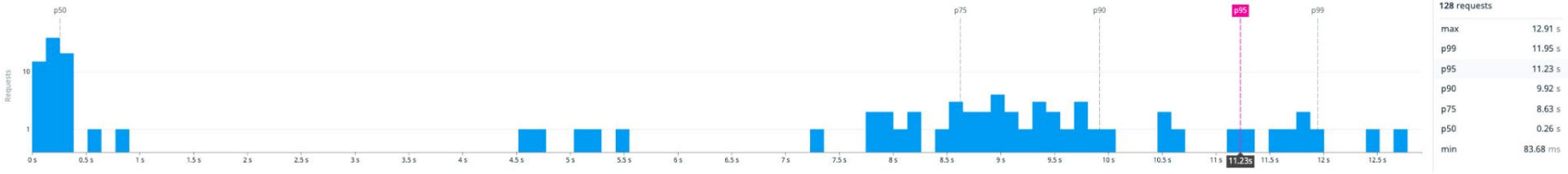
```
Acquire: select pg_advisory_lock(:id)

Release: select pg_advisory_unlock(:id)
```


GAINZ
imgflip.com

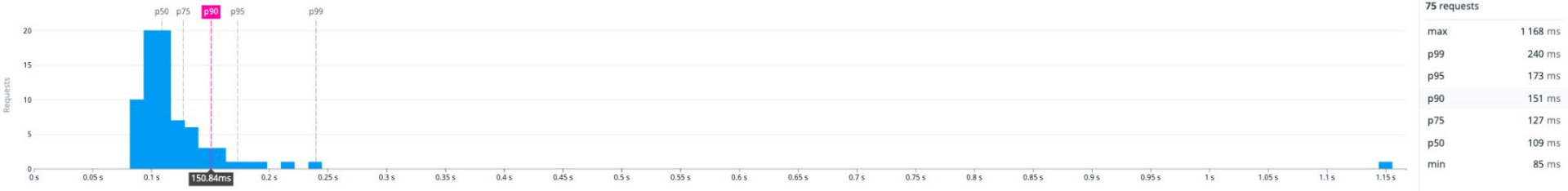
User update latencies

Historic data before outbox release



Latency distribution. Note that Y-axis has logarithmic scale.

Historic data after the release



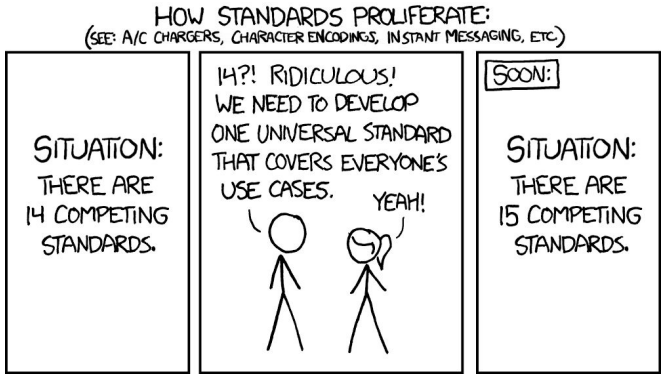
Latency distribution. Note that Y-axis has logarithmic scale.

What we've gained

- Reduced boilerplate code for new outbox items handling
    - Handler delegates to existing business flows
- Retries out of the box
- Per-type failure handling
- Learned much along the way

Related work

- Eventuate Tram (Richardson, microservices.io), quite thorough solution
  - More moving parts
    - message broker
    - their Change-Data-Capture microservice: requires Zookeeper for leader election when running as a cluster for high availability
- gruelbox/transaction-outbox, great alternative
  - Missing flexibility on a per-outbox type handlers (i.e., for failure, retry strategies) (e.g., they provide a single failure handler)
- Also, XKCD-927 and above all a learning opportunity



Concluding

Future work

- Configurable retry strategies (linear, exponential back-off)
- Immediate outbox item handling (e.g., via a post-commit TX hook or a CDC consumer)
- Evaluate co-routines instead of Executor service
- Extract to a library
  - core
  - spring-outbox: wrapper on-top core with store and locks implementations

Team

- Thanasis Polydoros
- Apostolis Kiraleos

Thank you