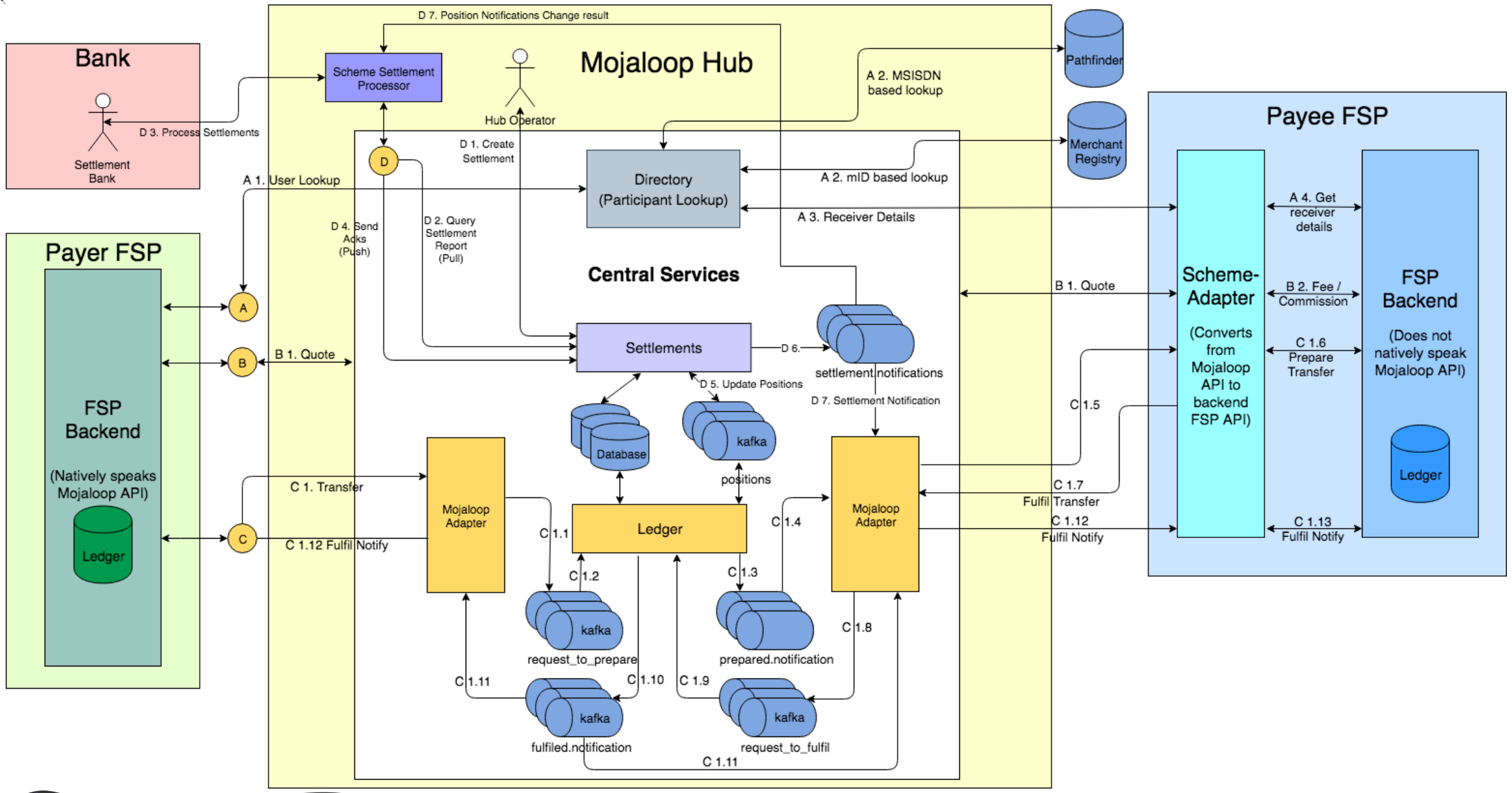
# Mojaloop

## PI9 Performance

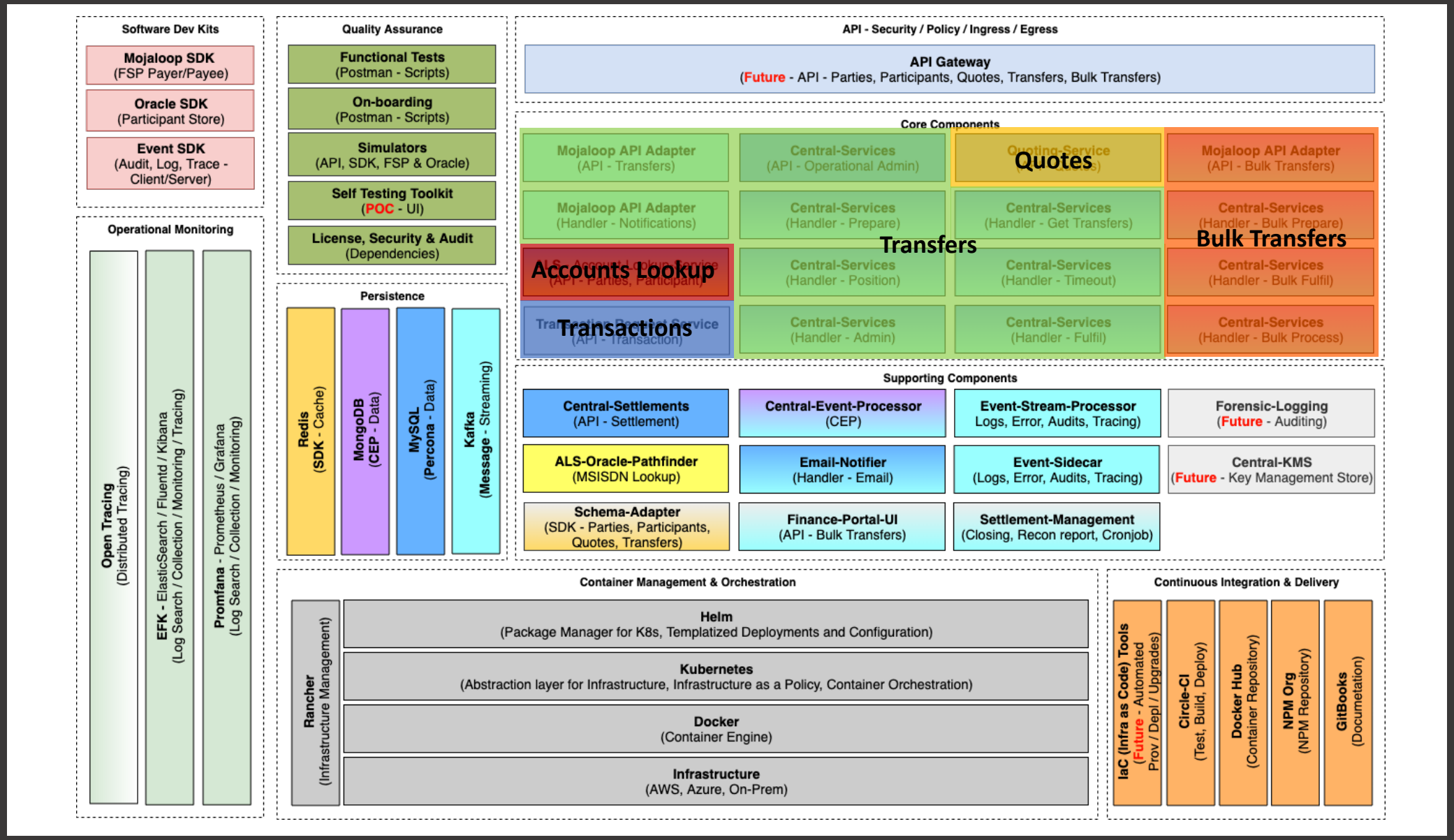
March 2020

ModusBox

# Architecture Overview: Mojaloop



# Component Architecture: Current PI-8 Architecture




March 2020

* Color-shading of core & supporting components match dependent persistent stores.

ModusBox

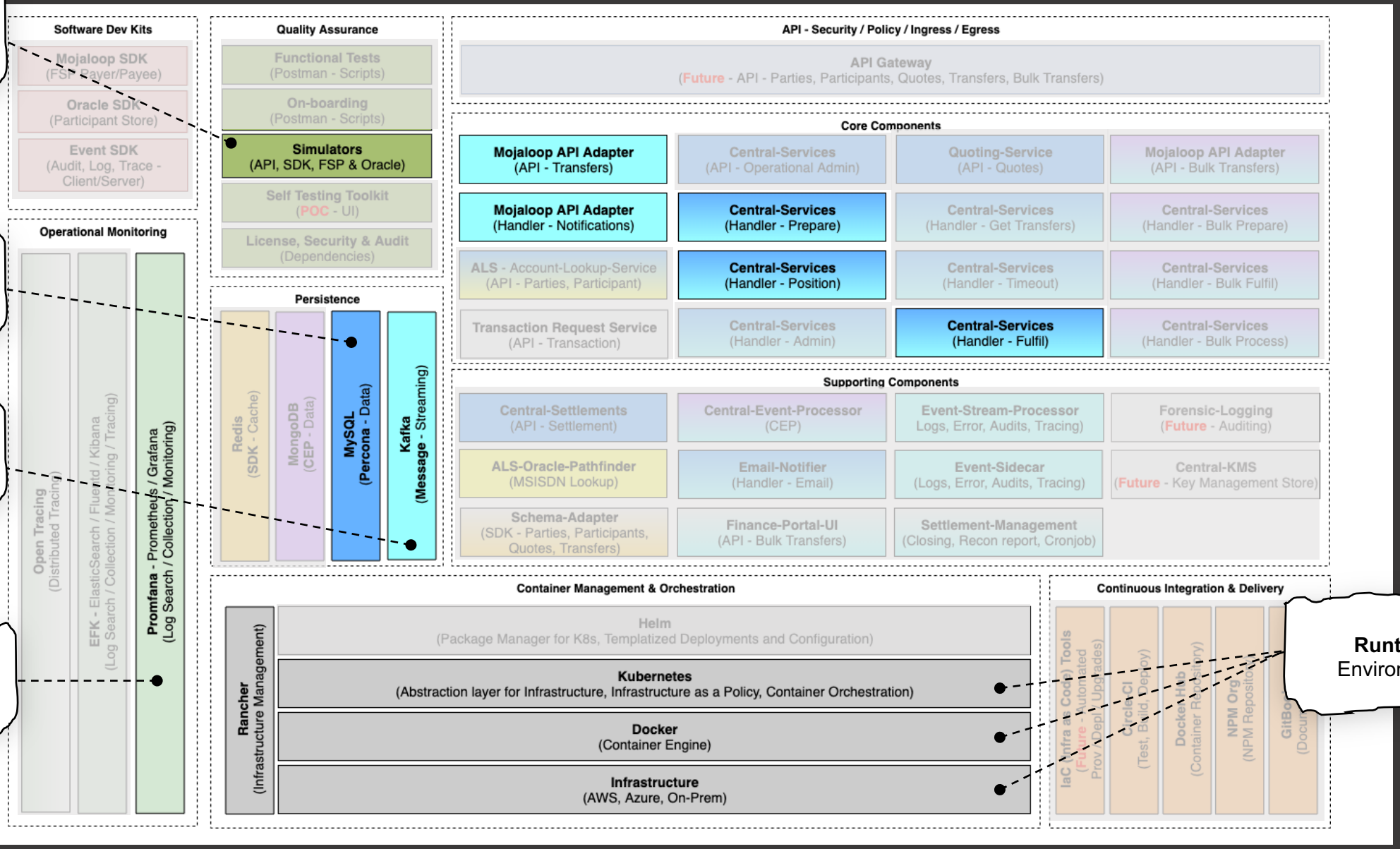
# Component Architecture: Scope of Performance Testing

Legacy **Simulator**

**Custom** and **RDS** Deployment

**Custom** Deployment

**Metrics** Dashboards & Reports




**Runtime** Environment

March 2020

* Legacy Simulator used

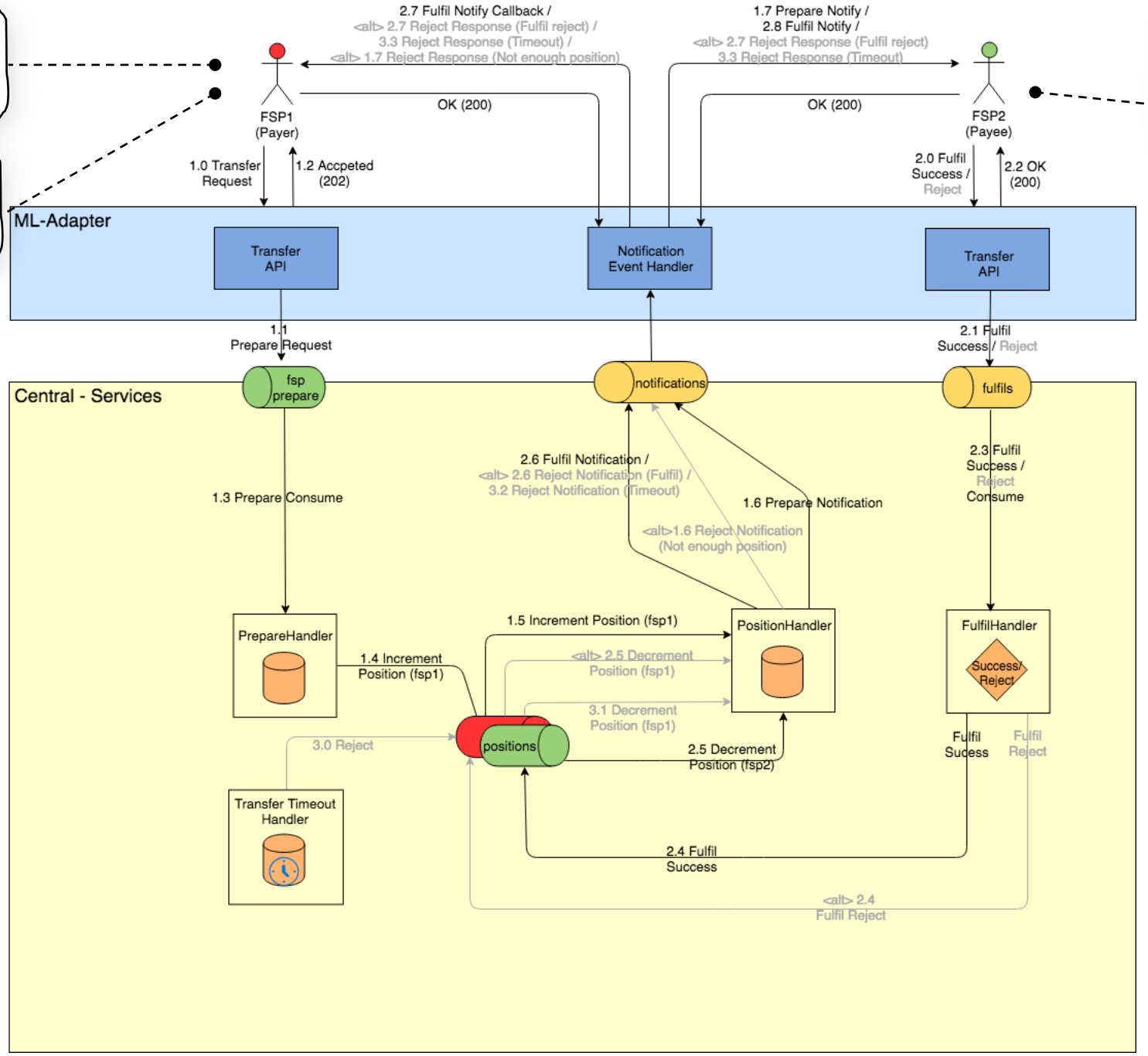
ModusBox

# Transfers Solution Design

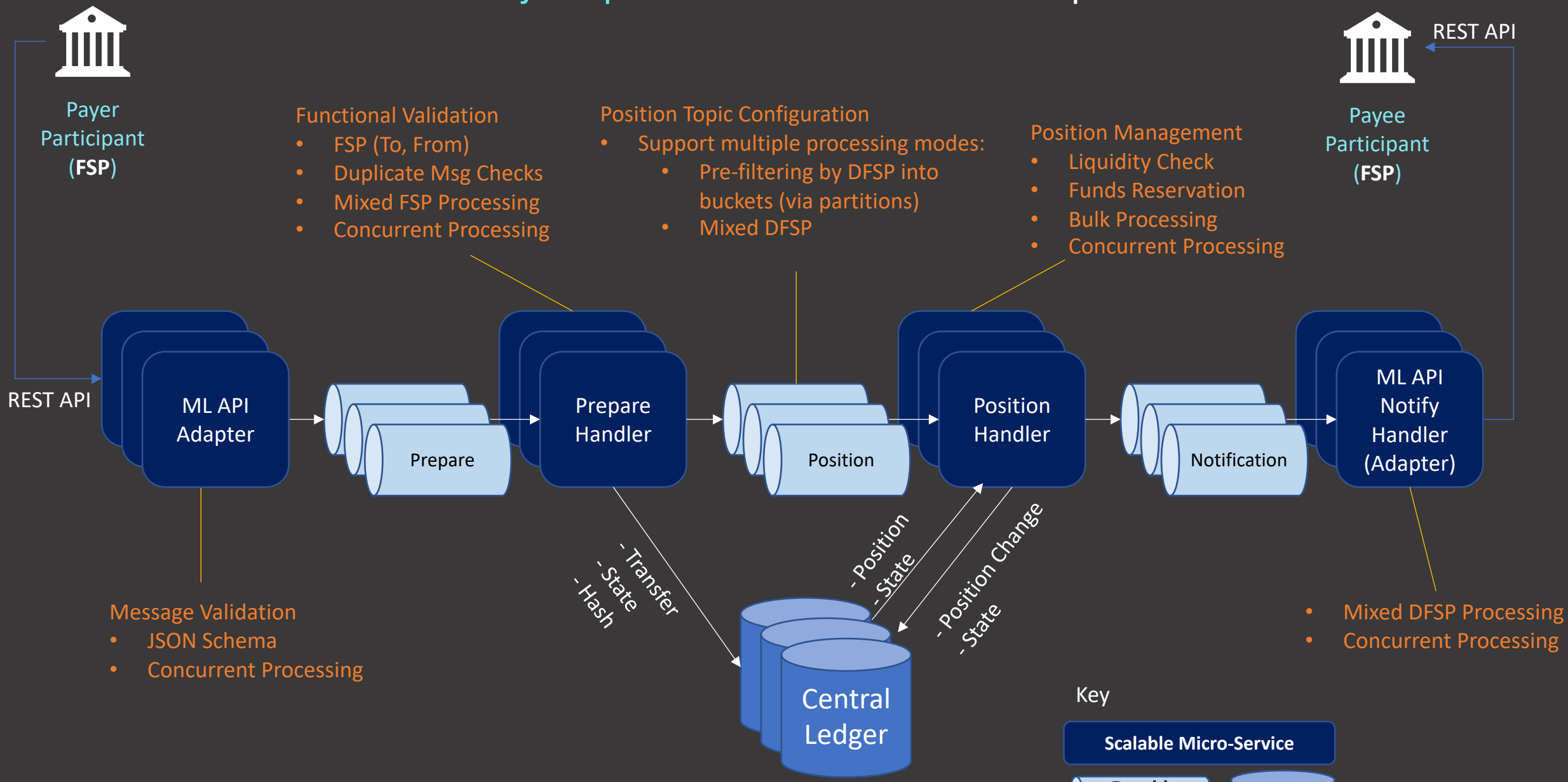
**JMeter** Initiates **Payer Request**

Legacy **Simulator** Receives **Fulfil Notify Callback**

Legacy **Simulator**
- handles **Payee processing**
- initiates **Fulfilment Callback**

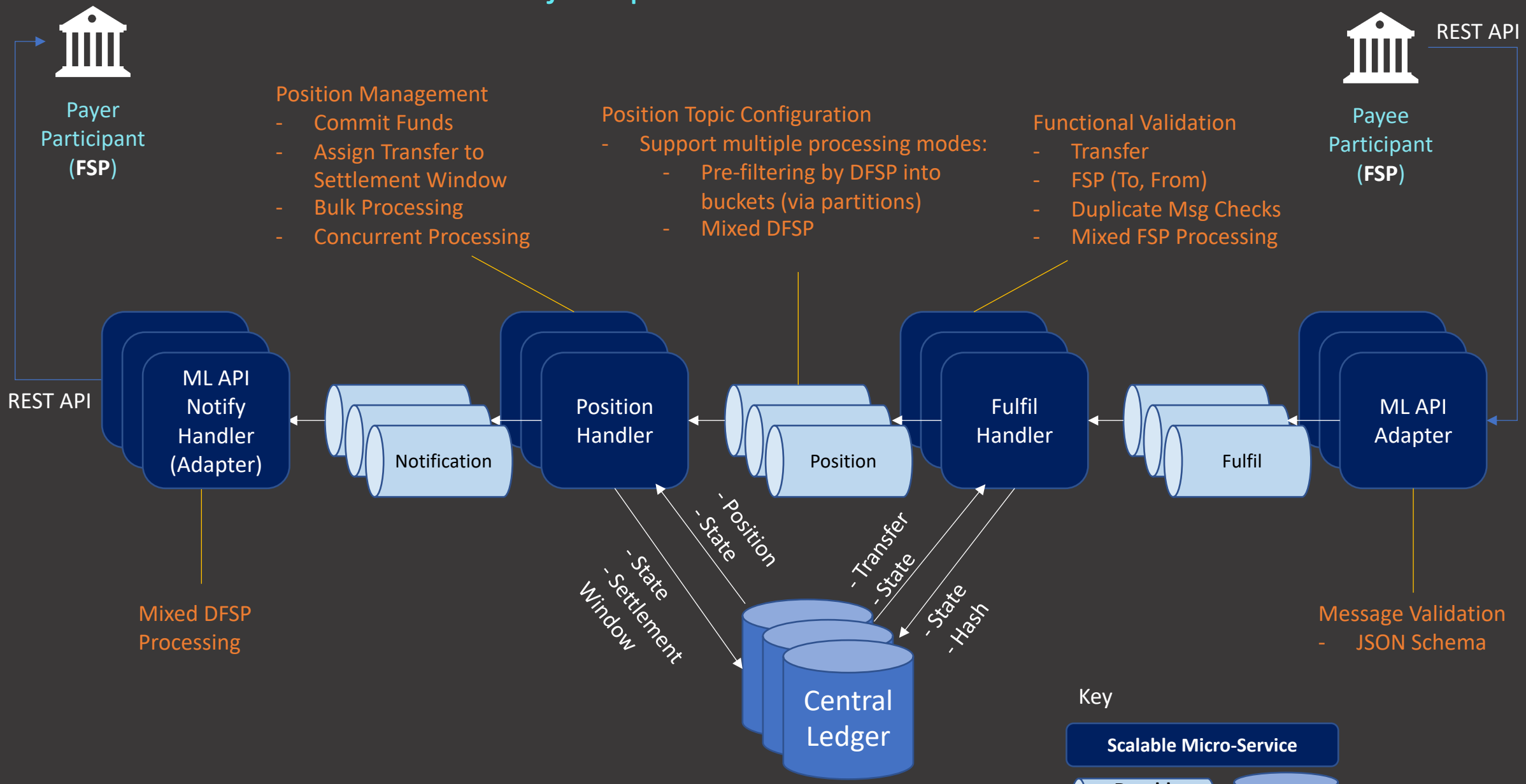


# Mojaloop Transfer Flow: Transfer Prepare




March 2020

# Mojaloop Transfer Flow: Transfer Fulfil




March 2020

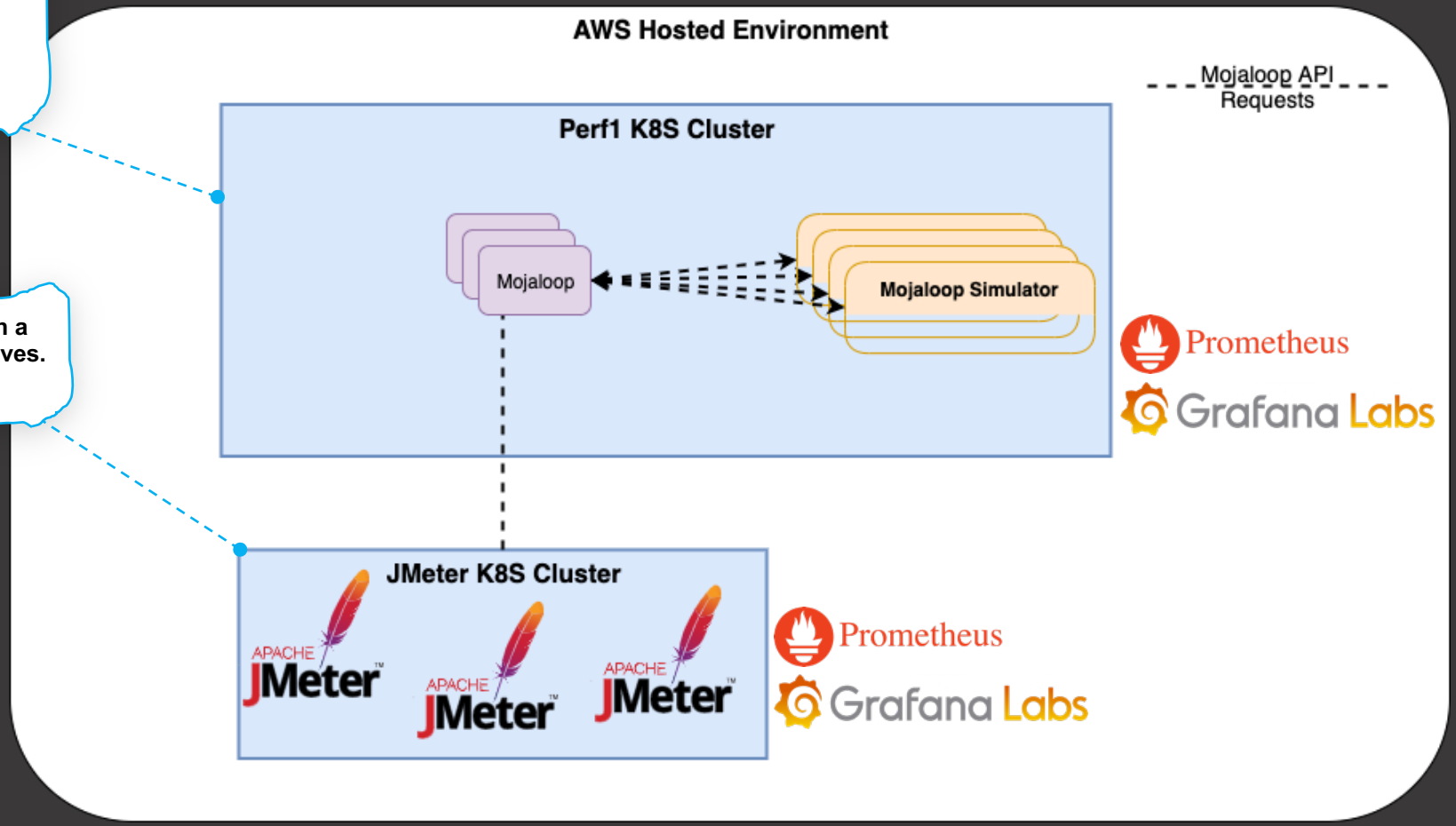
# Performance Environment

# Infrastructure Overview

**Perf1 Environment**

**Scalable Kubernetes Cluster**
1. **Partitioned nodes for DB, Kafka, Mojaloop, etc**
2. **NVME Storage for high disk IOP**

**Scalable JMeter cluster with a Master node and scalable slaves.**




## Environment

- Kubernetes Version: v1.11.6
- Rancher: v2.1.6
- Docker Version: 17.03.2-ce
- Kernel Version: 4.4.0-1052-aws
- Operating System: Ubuntu 16.04.4 LTS

# Infrastructure Overview – Perf1

| Name Prefix | Count | Template | Auto Replace | etcd | Control Plane | Worker |
|---|---|---|---|---|---|---|
| k8s-tanuki-perf1-io | 6 | k8s-tanuki-ubuntu-node-i3.xlarge | 0 minutes | ☐ | ☐ | ☑ |

kafka MySQL PERCONA XtraDB Cluster

| | | | | | | |
|---|---|---|---|---|---|---|
| k8s-tanuki-perf1-master | 3 | k8s-tanuki-ubuntu-node-m4.large | 0 minutes | ☑ | ☑ | ☐ |

kubernetes

| | | | | | | |
|---|---|---|---|---|---|---|
| k8s-tanuki-perf1-gcio | 23 | k8s-tanuki-ubuntu-node-i3.xlarge | 0 minutes | ☐ | ☐ | ☑ |

mojaloop

| | | | | | | |
|---|---|---|---|---|---|---|
| k8s-tanuki-perf1-op | 2 | k8s-tanuki-ubuntu-node-m4.xlarge | 0 minutes | ☐ | ☐ | ☑ |

Prometheus Grafana Labs

# Infrastructure Overview – JMeter

| Name Prefix | Count | Template | Auto Replace | etcd | Control Plane | Worker |
|---|---|---|---|---|---|---|
| k8s-tanuki-jmeter | 4 | k8s-tanuki-ubuntu-node-m4.xlarge | 0 minutes | ☑ | ☑ | ☑ |

mojaloop

# Performance Test Cases

# Performance: Test Cases

1. Financial Transaction End-to-End
2. Financial Transaction Prepare-only
3. Financial Transaction Fulfil-only
4. Individual Mojaloop Component Characterization
    1. Mojaloop Services & Handlers
    2. Mojaloop Streaming Architecture & Libraries
    3. Database

# Performance: Financial Transaction End-to-end

| Metric | Value | Comments |
|---|---|---|
| ***Financial Transactions per second (FPS)** | 301 FPS | 300 FPS is well over the T1 indicator of 200 FPS. **2x JMeter threads throttled to 150tps each. |
| ***Duration of Test** | 4800 seconds (80m) | Executed test longer than 60m to ensure consistency. |
| **Total Financial Requests** | 1442514 | Observed from JMeter metrics & verified from DB after the final Commit (Position Handler). |
| **Total Requests > 1s** | 0 | Calculated from 'moja_tx_transfer' metric. To be verified via DB after the final Commit (Position Handler). |
| ***% of Requests > 1s** | 0% | Calculated from 'moja_tx_transfer' metric. To be verified via DB after the final Commit (Position Handler). |
| **Average Financial Transaction Time** | 331ms | Observed from 'moja_tx_transfer' metric taken within a 1hour window. |
| **Minimum Financial Transaction Time** | 223ms | Observed from 'moja_tx_transfer' metric taken within a 1hour window. |
| **Maximum Financial Transaction Time** | 690ms | Observed from 'moja_tx_transfer' metric taken within a 1hour window. |

* ***Performance Targets for PI-8 based on the following indicators***

| # | Indicator | Target |
|---|---|---|
| **T1** | Financial Transactions Per Second (fps) | > 200 fps |
| **T2** | Time for performance run | >= 1 Hour sustained run |
| **T3** | % of transactions that took longer than a second | < 1% |

**Jmeter Configuration

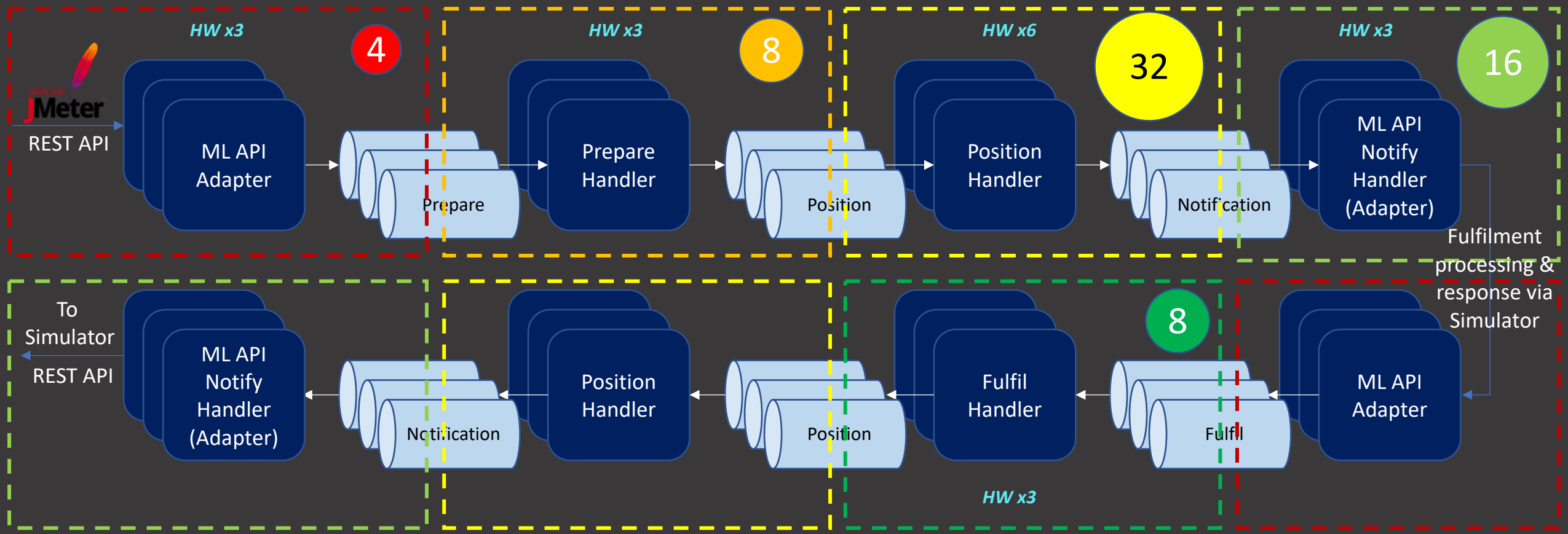
| Configuration | Value | Comments |
|---|---|---|
| **JMeter Slaves / Threads** | 2 | Each JMeter slave will execute a single Thread. |
| **JMeter Throttling** | 150 TPS | 2 Salves * 150TPS = 300 TPS Target |
| **JMeter Ramp-up** | 10 seconds | |
| **Number of FSPs** | 8 | |

# Performance: Financial Transaction End-to-end - Scaling

*- Pod Scaling*

*HW x#* *- Dedicated Kubernetes Node Workers*

** Components without the above two indicators will share scaling/resources/nodes with similarly colored components*



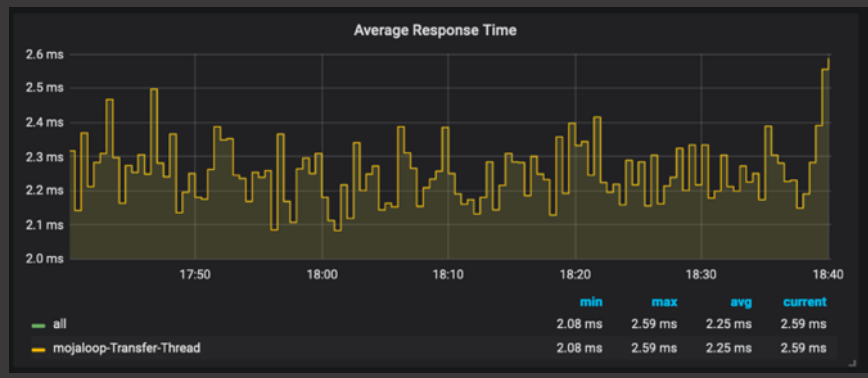

Environment Setup

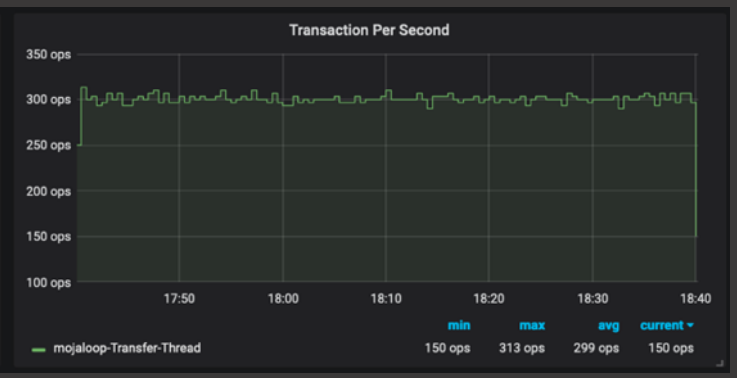
| Component | Pod Scale | Dedicate Node Workload Scale |
|---|---|---|
| ML-API-Adapter Service (API) | 4 | 3 |
| Central-Ledger Handler - Prepare | 8 | 3 |
| Central-Ledger Handler – Position | 32 | 6 |
| Central-Ledger Handler – Fulfil | 8 | 3 |
| ML-API-Adapter Handler - Notifications | 16 | 3 |

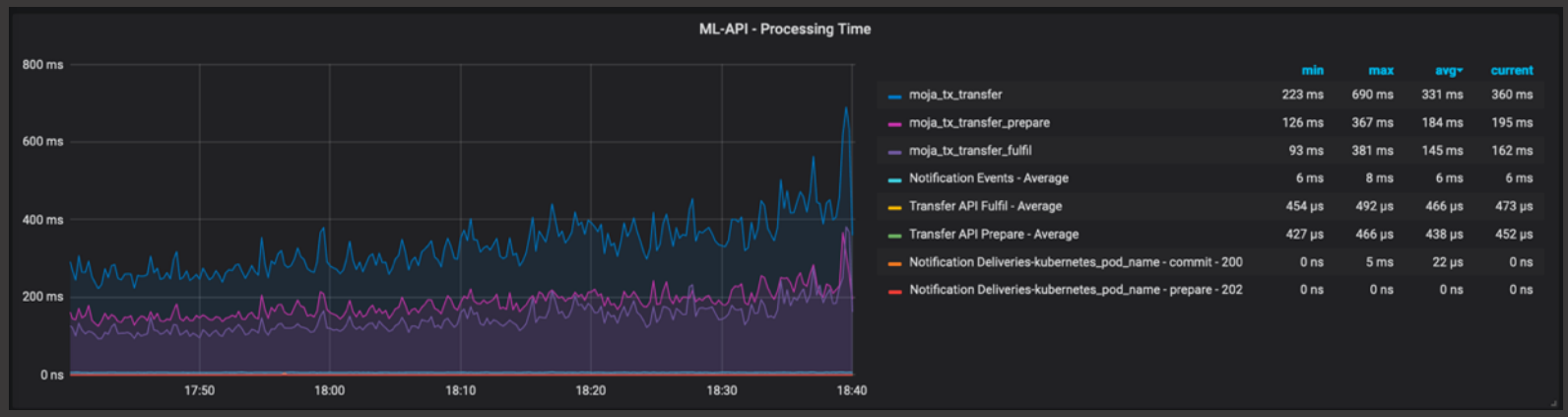
# Performance: Financial Transaction End-to-end Performance Enhancements

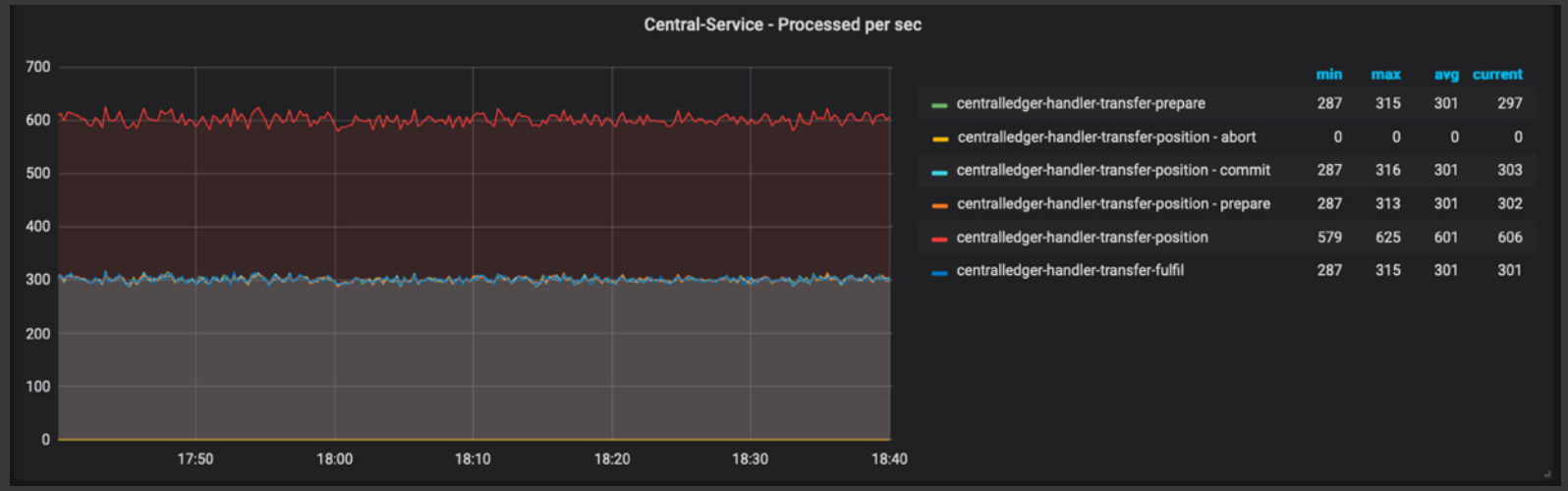
1. Caching of non-stateful data for Prepare, Position and Fulfil Handlers
    1. Participants Info
    2. Participant Accounts
    3. Participant Limits
2. Metrics
    a. Caching (Hit vs Miss)
    b. Prepare, Fulfil and Position Models (Database Access Objects)
    c. Prepare, Fulfil and Position Functional Domains
    d. Central-Services-Stream and underlying Node-Rdkafka Library
3. Streamlined logging on ML-API-Adapter, Central-Services and Central-Services-Stream
4. Optimized Duplicate-Check logic for both Prepare and Fulfil Handlers using an Insert-only algorithm
5. Balanced Handler scale/deployments for each workload (e.g. Prepare vs Position vs Fulfil Handlers)
6. Modified Kafka Configuration (auto-commit interval) to facilitate better monitoring
7. Deployed RDS to demonstrate how Mojaloop behaves against an optimized MySQL deployment with EBS provisioned with 2000 IOPS.

# Performance: Financial Transaction End-to-end - Metrics



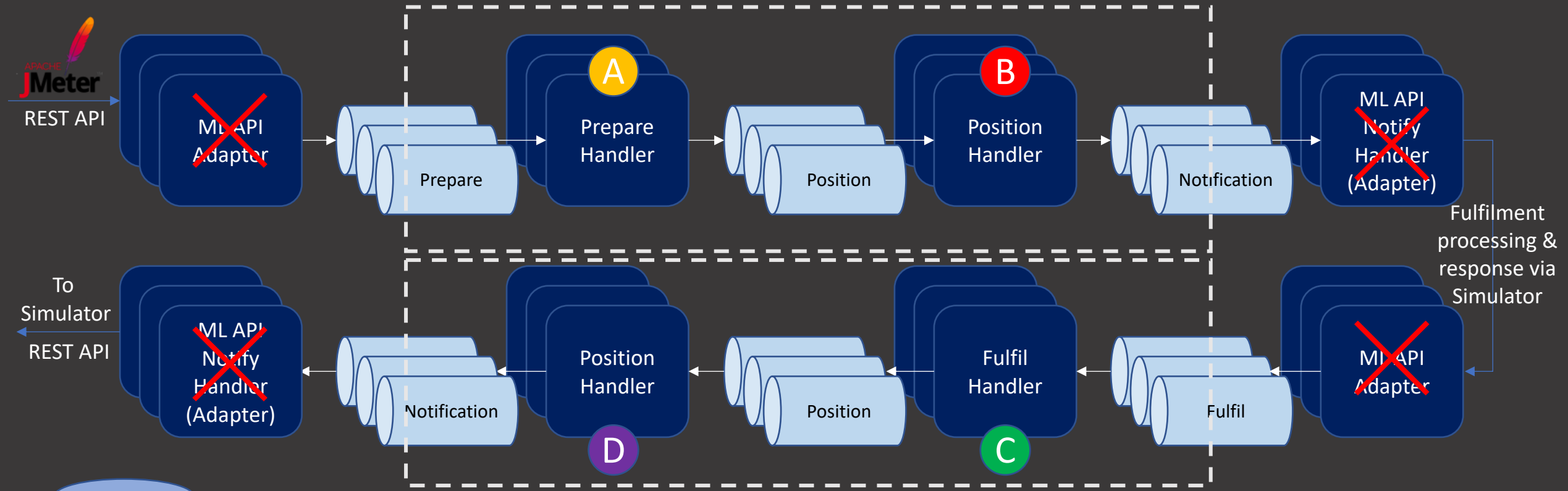

March 2020

# Performance: Financial Transaction Prepare & Fulfil Only

** OPS – Operations Per Second (Complete Handler Process per Message)*

| OPS | Scale | HW |
|---|---|---|
| 408* | 8 | 3 |

| OPS | Scale | HW |
|---|---|---|
| 408* | 16 | 3 |




| OPS | Scale | HW |
|---|---|---|
| 539* | 16 | 3 |

| OPS | Scale | HW |
|---|---|---|
| 539* | 8 | 3 |

# Performance: Characterization - Services & Handlers

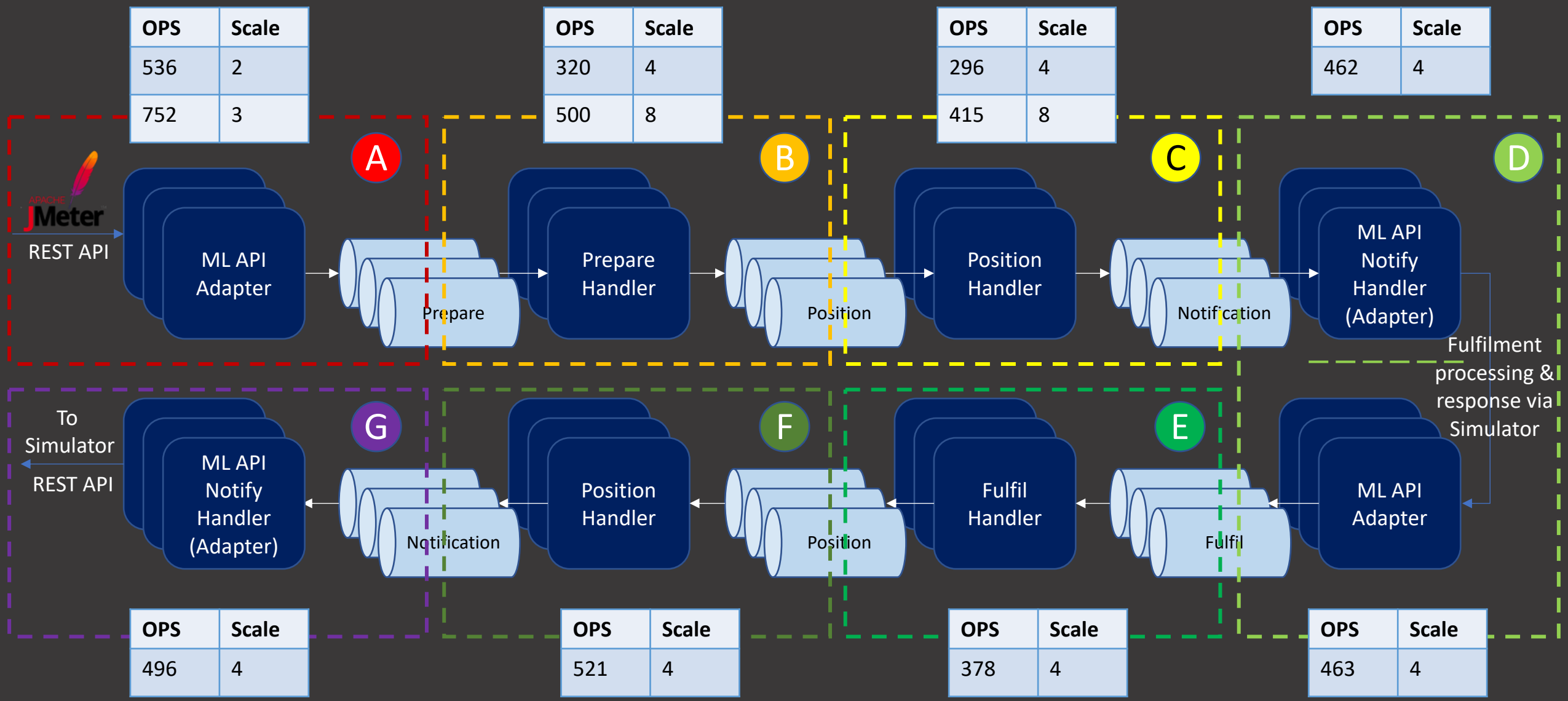
** OPS – Operations Per Second (Complete Handler Process per Message)*

| OPS | Scale |
|---|---|
| 536 | 2 |
| 752 | 3 |

| OPS | Scale |
|---|---|
| 320 | 4 |
| 500 | 8 |

| OPS | Scale |
|---|---|
| 296 | 4 |
| 415 | 8 |

| OPS | Scale |
|---|---|
| 462 | 4 |




| OPS | Scale |
|---|---|
| 496 | 4 |

| OPS | Scale |
|---|---|
| 521 | 4 |

| OPS | Scale |
|---|---|
| 378 | 4 |

| OPS | Scale |
|---|---|
| 463 | 4 |

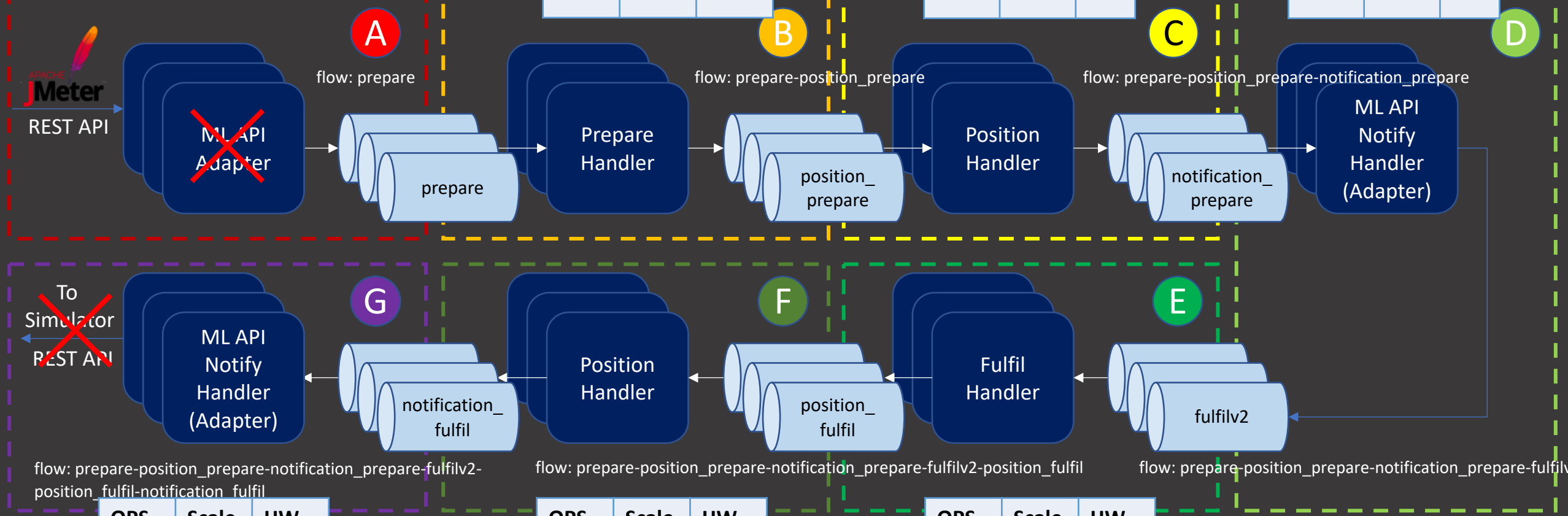
# Performance: Characterization - Streaming Architecture & Libs

** OPS – Operations Per Second (Reading & Writing to Kafka)*
** Kafka is doing ~60k OPS in total*

| OPS | Scale | HW |
|---|---|---|
| 1.6k | 1 | 1 |
| 6.5k | 8 | 3 |

| OPS | Scale | HW |
|---|---|---|
| 1.5k | 1 | 1 |
| 5.5k | 8 | 3 |

| OPS | Scale | HW |
|---|---|---|
| 1.5k | 1 | 1 |
| 5.5k | 8 | 3 |




| OPS | Scale | HW |
|---|---|---|
| 1.5k | 1 | 1 |
| 5.5k | 8 | 3 |

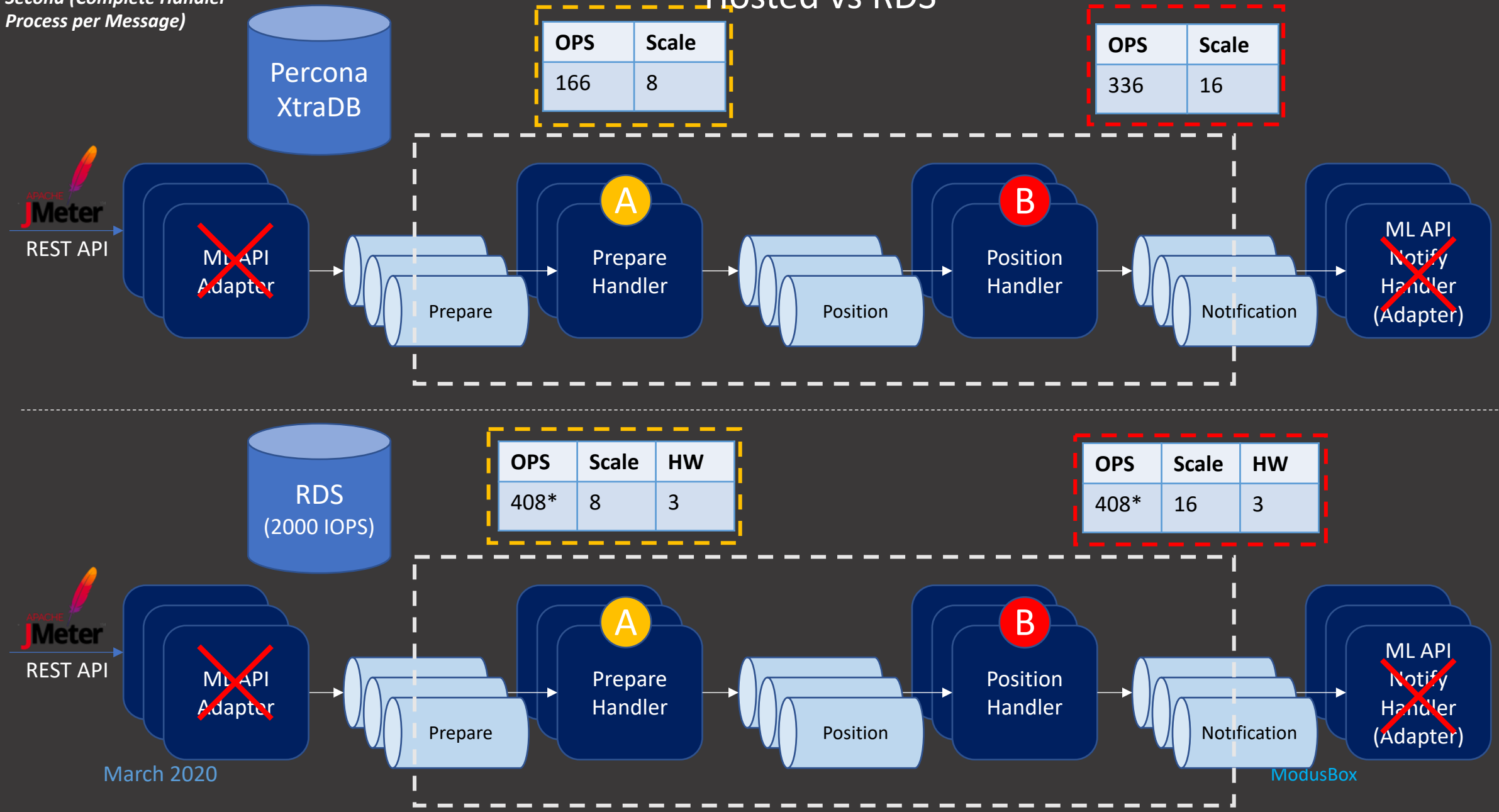
| OPS | Scale | HW |
|---|---|---|
| 1.5k | 1 | 1 |
| 5.5k | 8 | 3 |

| OPS | Scale | HW |
|---|---|---|
| 1.5k | 1 | 1 |
| 5.5k | 8 | 3 |

# Performance: Characterization - Database – Prepare-only Hosted vs RDS

** Caching disabled*

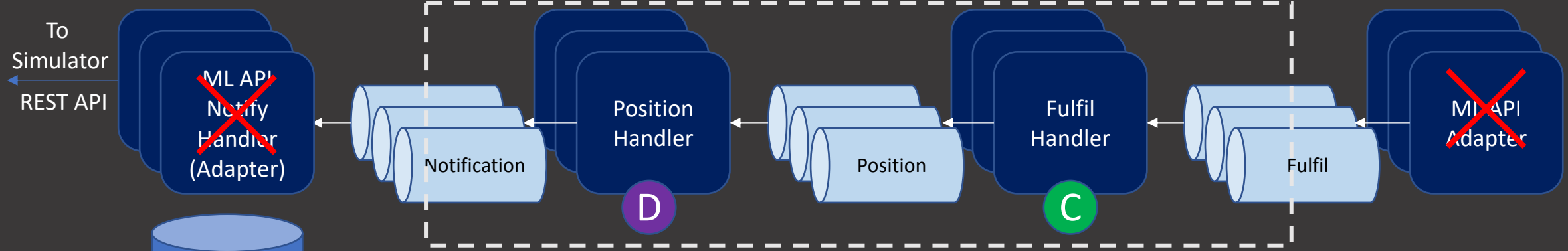
** OPS – Operations Per Second (Complete Handler Process per Message)*



# Performance: Characterization - Database – Fulfil-only Hosted vs RDS

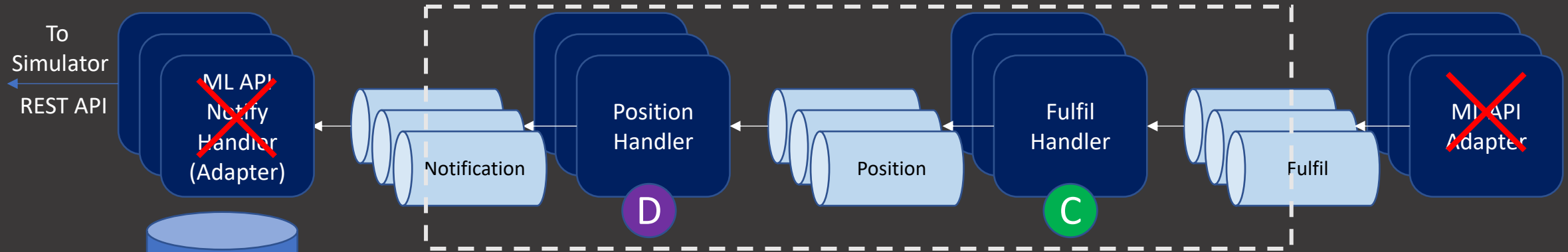
** Caching disabled*

** OPS – Operations Per Second (Complete Handler Process per Message)*




| OPS | Scale |
|---|---|
| 336 | 16 |

| OPS | Scale |
|---|---|
| 170 | 8 |

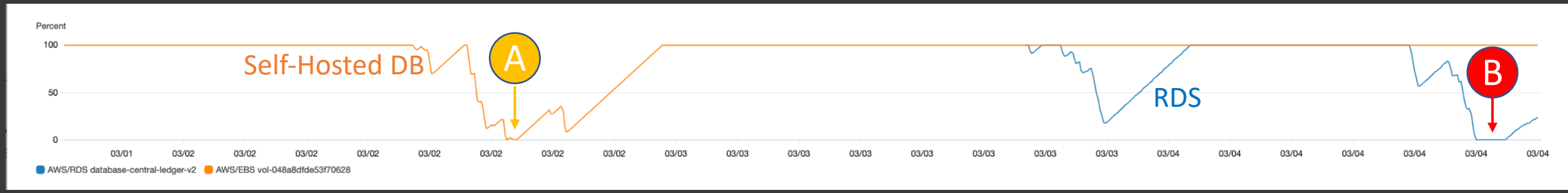



| OPS | Scale | HW |
|---|---|---|
| 539 | 16 | 3 |

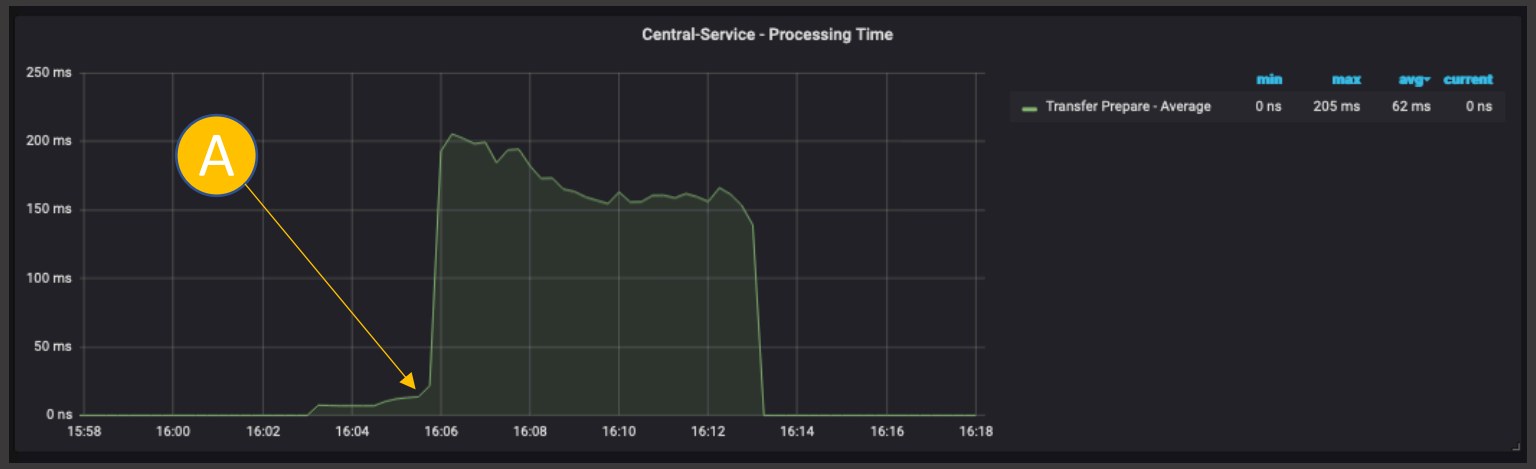
| OPS | Scale | HW |
|---|---|---|
| 539 | 8 | 3 |

# Performance: Database - IOPS Throttling

## Database - IOPS Budget Balance




## Observed Throttling on Prepare Handler




## Observed Throttling on AWS

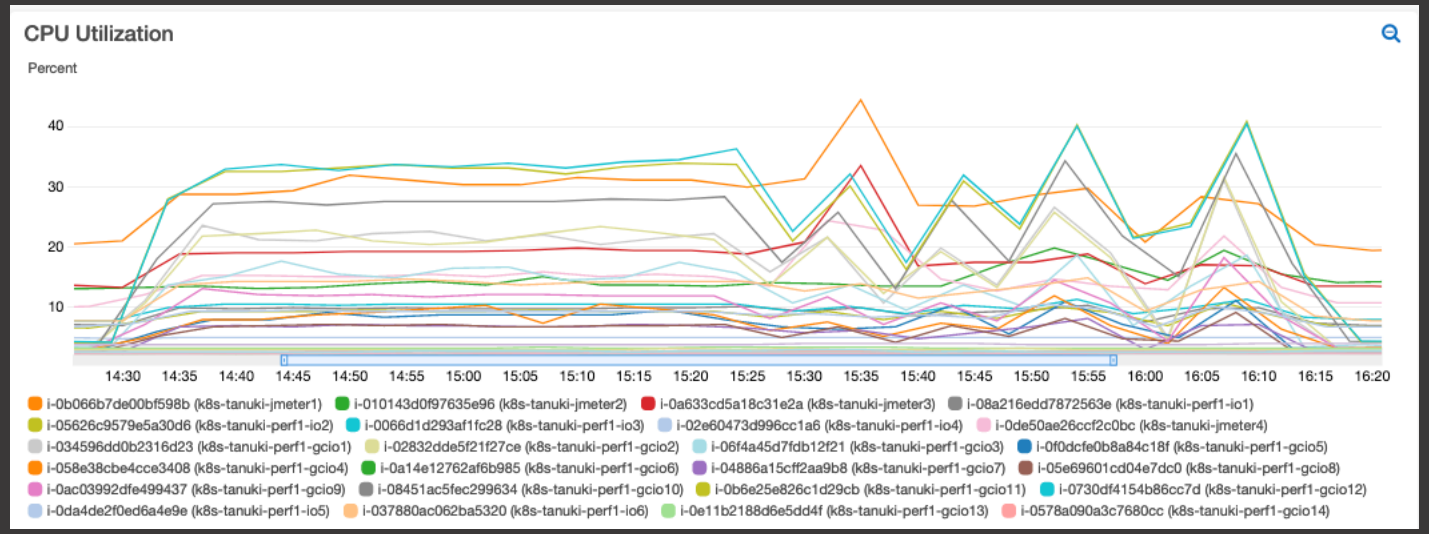
# Performance
# Known Issues

# Known Issue: Low CPU / Resource Usage

- Current CPU Usage is capped around 40% for each K8s-Node. Maximizing the resource usage is key for efficiency and to reduce infrastructure costs.

Optimizations:

- Single NodeJS Thread Performance
- Multiple NodeJS Thread Performance (e.g. Node Clustering)

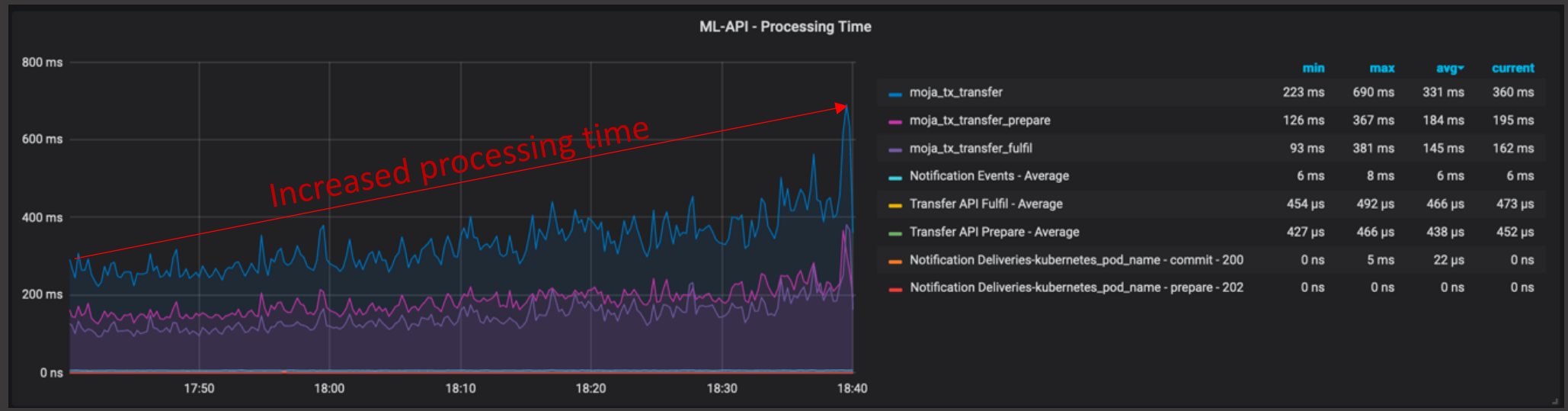


# Known Issue: Increasing Processing Time Trend

- Processing Times increase over time

Possible Causes:

- Correlation between Processing time increase vs table Size growth



# Known Issue: Performant Deployable Database

- Current Deployed and Configured DB as part of Helm deployments are unoptimized.

Example Optimizations:

- Resolve scaling issues (Refactor Transaction Logic to cater for Optimistic Locking)
- Queue Cache Size
- Configure full usage of Available Memory
- AWS – Ensure storage has guaranteed IOPS provisioned

Etc…

# Known Issue: Timeout Handler

**Current processing of timeouts:**

1. Retrieve transferStateChangeId up to which transfers have been processed during previous run (segment table)
2. Cleanup the list of expiring transfers (transferTimeout table) from finilised transfers [reads transferStateChange]
3. Determine current max transferStateChangeId up to which the current run will proceed (transferStateChange) [read]
4. Populate transferTimeout table with newly added transfers, which are not finilised [reads transferStateChange]
5. At this point transferTimeout holds records only for expiring transfers with their respective expiriationDate
6. Timeout transfers for which funds where not yet reserved by inserting multiple records in transferStateChange-EXPIRED_PREPARED [reads transferStateChange]
7. Mark for timeout transfer with reserved finds by inserting multiple records in transferStateChange-RESERVED_TIMEOUT [reads transferStateChange]
8. Store current max transferStateChangeId to be used for the next run (segment table)
9. Read extended info for all expired transfers, including latest state changes [reads transferStateChange]
10. Queue individual message for each transfer in the previous list to NotificationHandler/PostionHandler
11. For each transfer, PositionHandler reads from db info to change position [reads transferStateChange]
12. For each transfer, PositionHandler updates the position, writes transferStateChange and queues notification message
13. This process is currently configured to repeat every 15s

**Pitfalls:**

1. Too many multi record reads during TimeoutHandler from transferStateChange (6)
2. Two bulk inserts in transferStateChange happen inside a transaction during TimeoutHandler processing
3. PositionHandler updates position and inserts transferStateChange using transaction for each expiring transfer

**Disadvantages:**

1. Possible table level locks for transferStateChange
2. TimeoutHandler can not be scaled

**Proposed Enhancement:**

1. Create TimeoutPrepareHandler which does not use segment and transferTimeout tables, but rather a new table expiringTransfer (expiringTransferId, transferId, expirationDate, createdDate) indexed by expirationDate
2. PrepareHandler will create a new record in expiringTransfer for each incoming transfer
3. FulfilHandler will not delete records from expiringTransfer, but it is an option that might be considered
4. Upon run, TimeoutPrepareHandler will first delete all records for which a matching transferId exists in tranfserFulfilment and then select using index all transfers for which expirationDate < currentTimestamp. For each transfer in the selected set a message will be produced.
5. Create TimeoutProcessingHandler that will query current transfer state and
   a. Stop processing if transfer is finalized (and delete expiringTransfer entry), or
   b. Insert transferStateChange-EXPIRED_PREPARED and produce message to notification-topic (-expiringTransfer), or
   c. Insert transferStateChange-RESERVED_TIMEOUT and produce message to position-topic

In case of 5c PositionHandler processes the message as with the current TimeoutHandler.
6. PositionHandler to clear processed exprintTransfer etnry

**Advantages:**

1. Reads from transferStateChange are per transfer
2. TimeoutProcessingHandler could be scaled when necessary

**Disadvantages:**

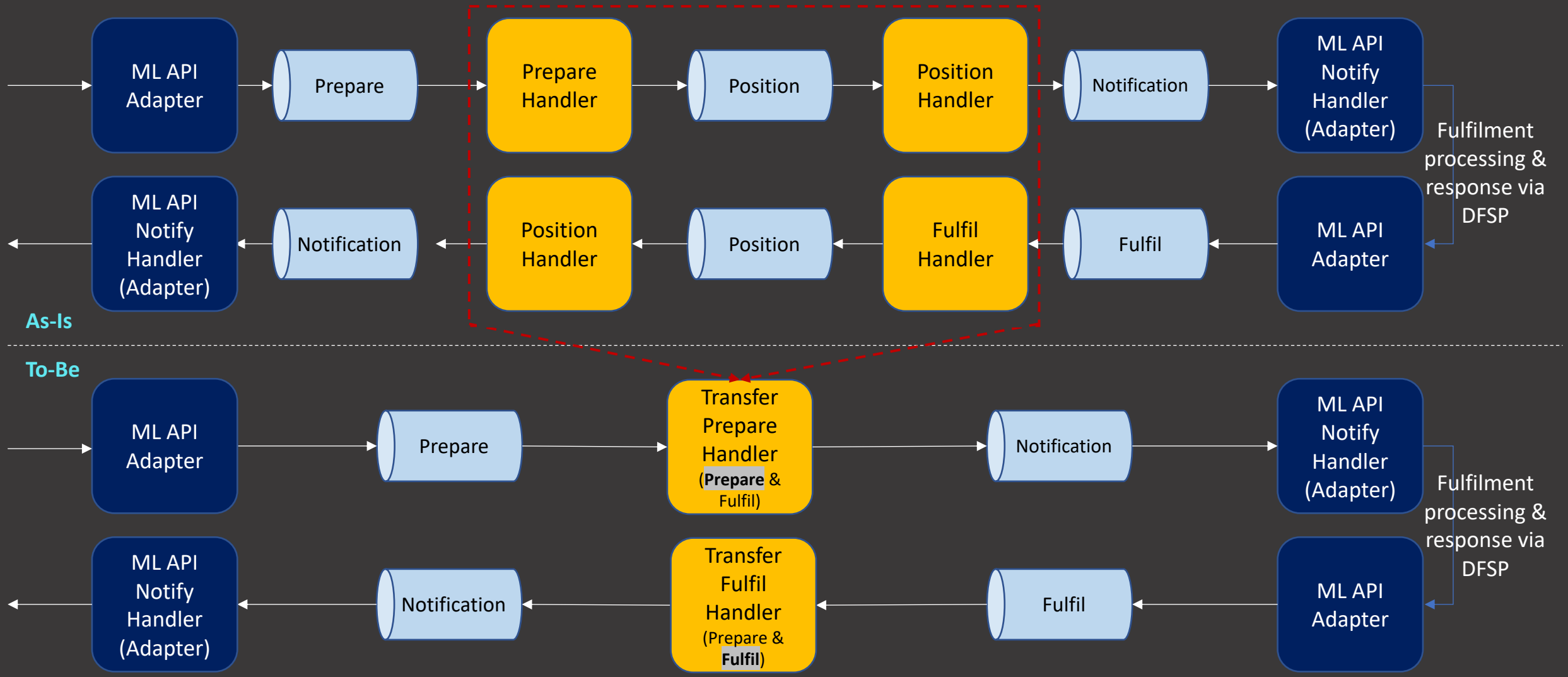
1. One extra insert during PrepareHandler processing, but expirationDate index maintenance should not be heavy as records get cleaned from it by TimeoutPrepareHandler

**Stories:**

1. https://github.com/mojaloop/project/issues/1089

# Performance
# Future Optimizations

# Future Optimization: Consolidated Prepare/Fulfil & Position Handlers




- Reduction in DB reads/write (state kept in memory for each leg – prepare & fulfil)
- Reduction in latency (less hops over network)

# Future Optimization: PRISM

## Meaning:

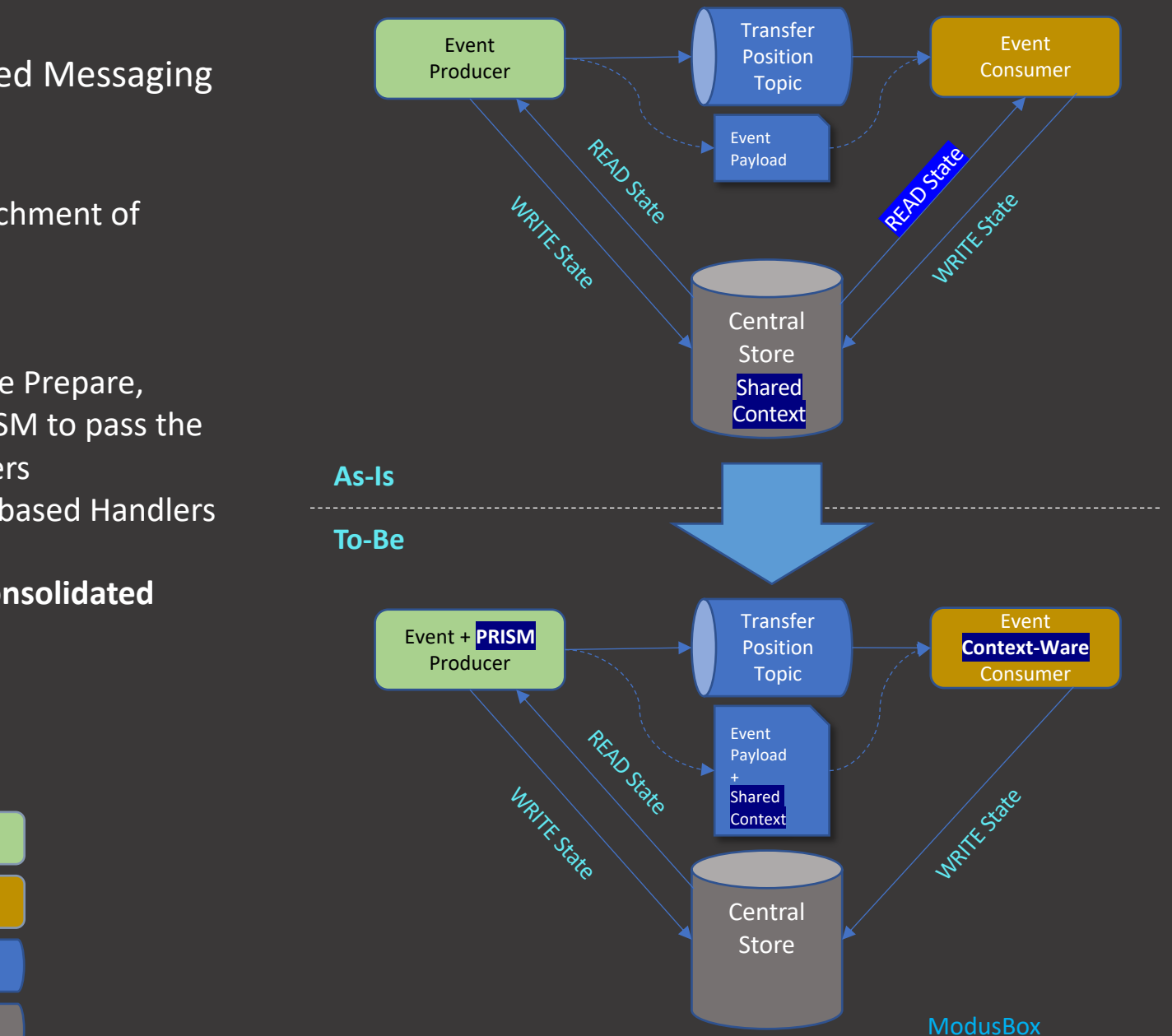
- PRISM - Persisted Replicated Internal Shared Messaging

## Description:

- Kafka messaging system is used for stateful enrichment of messages to down-stream handlers

## Considerations:

- We can minimize unnecessary locks between the Prepare, Position, Fulfil and Timeout Handlers by using PRISM to pass the necessary information to the down-stream handlers
- General improvement to most Kafka Consumer based Handlers (e.g. Prepare, Positions, Fulfil, Notifications, etc)
- **Not applicable if Prepare/Fulfil Handlers are consolidated**

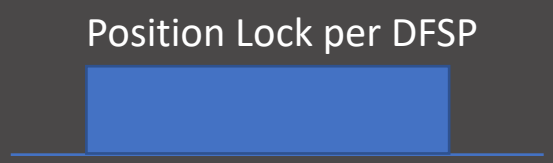


# Future Optimization: Position Handler Concurrency

Objective: Compete historic state record at time of rule execution, while never breeching liquidity checks

## 1. Position Handler Turnstile – single instance, bulk transfer set for processing

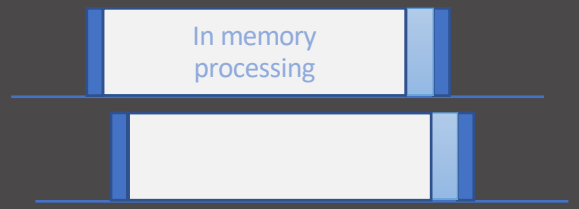
Single Position Handler




1. DFSP's Position is held in READ/UPDATE lock while processing transfer set read from topic
2. Means there can only be one active handler per DFSP

## 2. Position Reserved Handler – Reserved Position, multi-instances

Multiple Position Handlers




Position Reserve and Position Finalise, separated with in memory processing

1. Position/Liquidity is reserved for total of transfer set
2. After in memory processing:
   a. Finalised transfer states are INSERTed, and
   b. DFSP Liquidity is finalised with UPDATE
3. Works in mixed or per DFSP processing mode
4. Multiple active handlers

# Future Optimization: Review Architecture

## Open Discussion Topic

Architecture Assessment

Examples:

- CQRS / Event-Sourcing
- …

# Performance Roadmap

# Performance: Roadmap – Domains for Characterization

- Mojaloop
  - Account Lookup Services
  - Quoting Services
  - Settlement Services
  - Event Framework (Tracing, Auditing, etc)
  - EFK (Impact of Log ingestion on Mojaloop and underlying K8S infrastructure)
- Mojaloop SDK Components
  - Mojaloop-Simulator
  - Mojaloop-Scheme-Adapter
  - Mojaloop-SDK

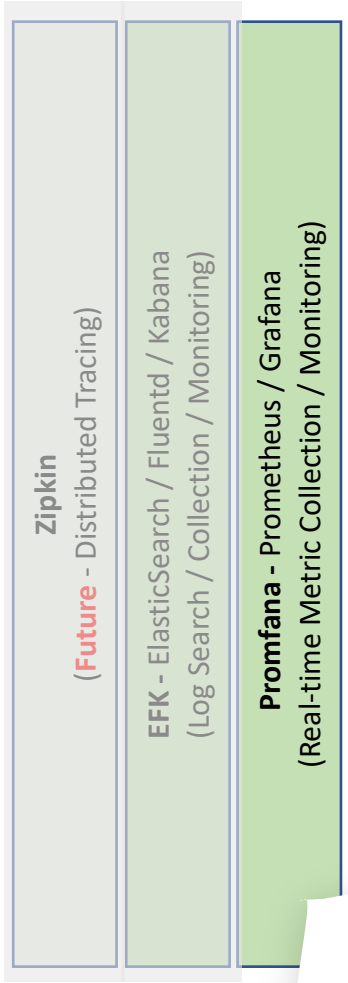
- Supporting Run-time Environment
  - Kubernetes
  - Kafka
  - Databases
- Hub Implementation
  - Security Controls
    - API Gateways (Ingres / Egress)
    - Encryption (TLS, JWS, etc)
    - Authentication

# Appendix

# Appendix

## Metrics & Instrumentation

# Ops Metric Instrumentation - Overview




## What is Metric Instrumentation?

Real-time operational visibility for:

- Performance
- Health
- Alerts

*Ref:* *http://prometheus.io*, *http://Grafana.com*

## What is Promfana?

Prometheus

Leading open-source instrumentation solution for monitoring

Grafana

The open platform for beautiful analytics and monitoring

## Why Promfana?

- Metric Instrumentation for Mojaloop
- Low overhead on nodejs (histograms + pull metric end-point)
- Real-time metric visualization for Performance and Health monitoring of the Mojaloop Stack

### Dimensional data

Prometheus implements a highly dimensional data model. Time series are identified by a metric name and a set of key-value pairs.

### Powerful queries

PromQL allows slicing and dicing of collected time series data in order to generate ad-hoc graphs, tables, and alerts.

### Great visualization

Prometheus has multiple modes for visualizing data: a built-in expression browser, Grafana integration, and a console template language.

### Efficient storage

Prometheus stores time series in memory and on local disk in an efficient custom format. Scaling is achieved by functional sharding and federation.

### Simple operation

Each server is independent for reliability, relying only on local storage. Written in Go, all binaries are statically linked and easy to deploy.

### Precise alerting

Alerts are defined based on Prometheus's flexible PromQL and maintain dimensional information. An alertmanager handles notifications and silencing.

### Many client libraries

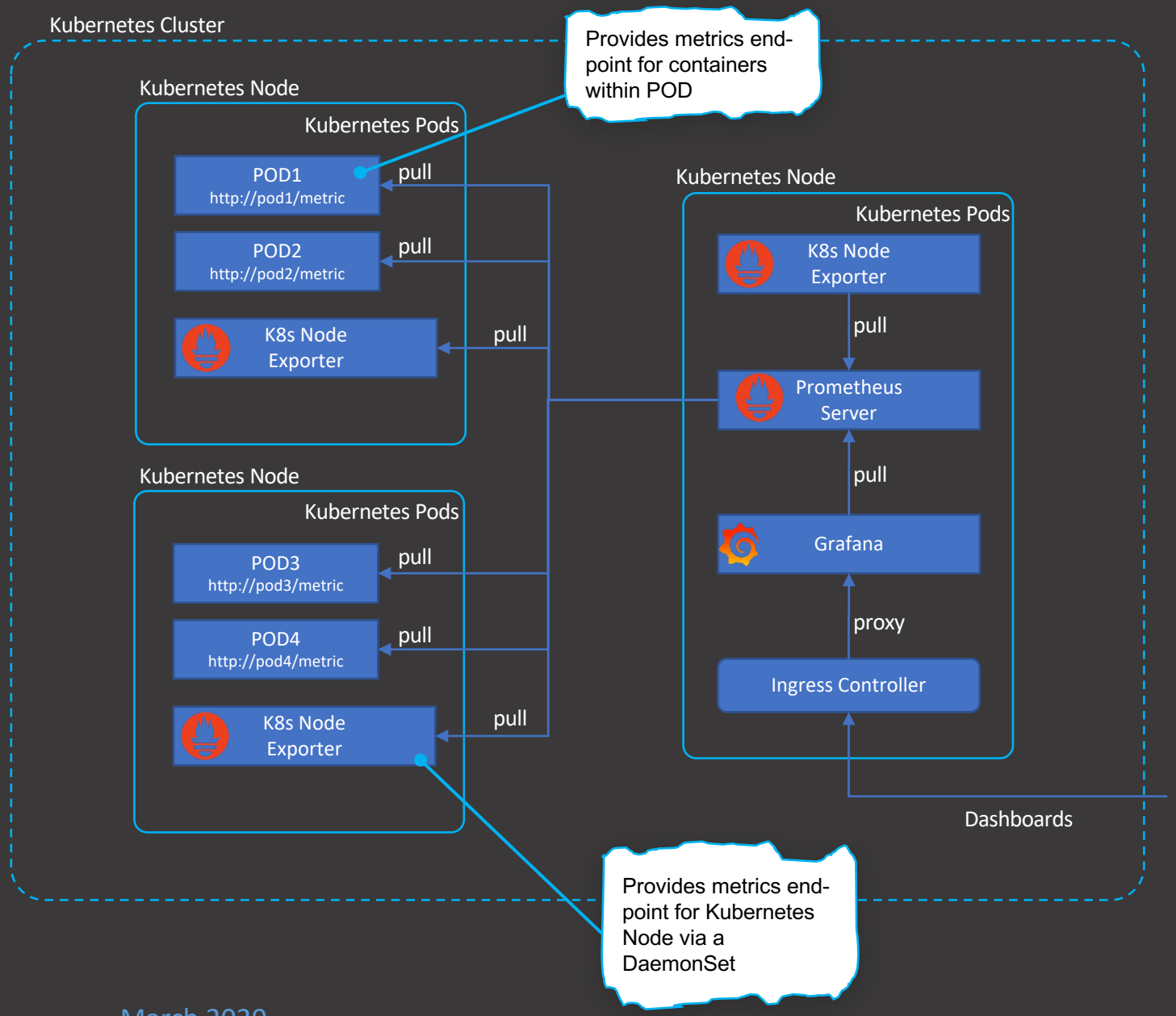
Client libraries allow easy instrumentation of services. Over ten languages are supported already and custom libraries are easy to implement.

### Many integrations

Existing exporters allow bridging of third-party data into Prometheus. Examples: system statistics, as well as Docker, HAProxy, StatsD, and JMX metrics.

CoreOS docker DigitalOcean SOUNDCLOUD PERCONA ERICSSON weaveworks

# Ops Metric Instrumentation – Architecture




## Requirements:

- Each pod to expose a HTTP API for /metrics operation to provide the instrumentation

## Enabling Collection of Metrics for a Pod

- Add annotation to Pod during deployment

```
annotations:
    prometheus.io/port: "8080"      <-- Specifies Port for /metrics collection
    prometheus.io/scrape: "true"    <-- Informs Prom Server that Pod is a target for collection
```

- This can be configured in each Helm chart by enabling Metrics

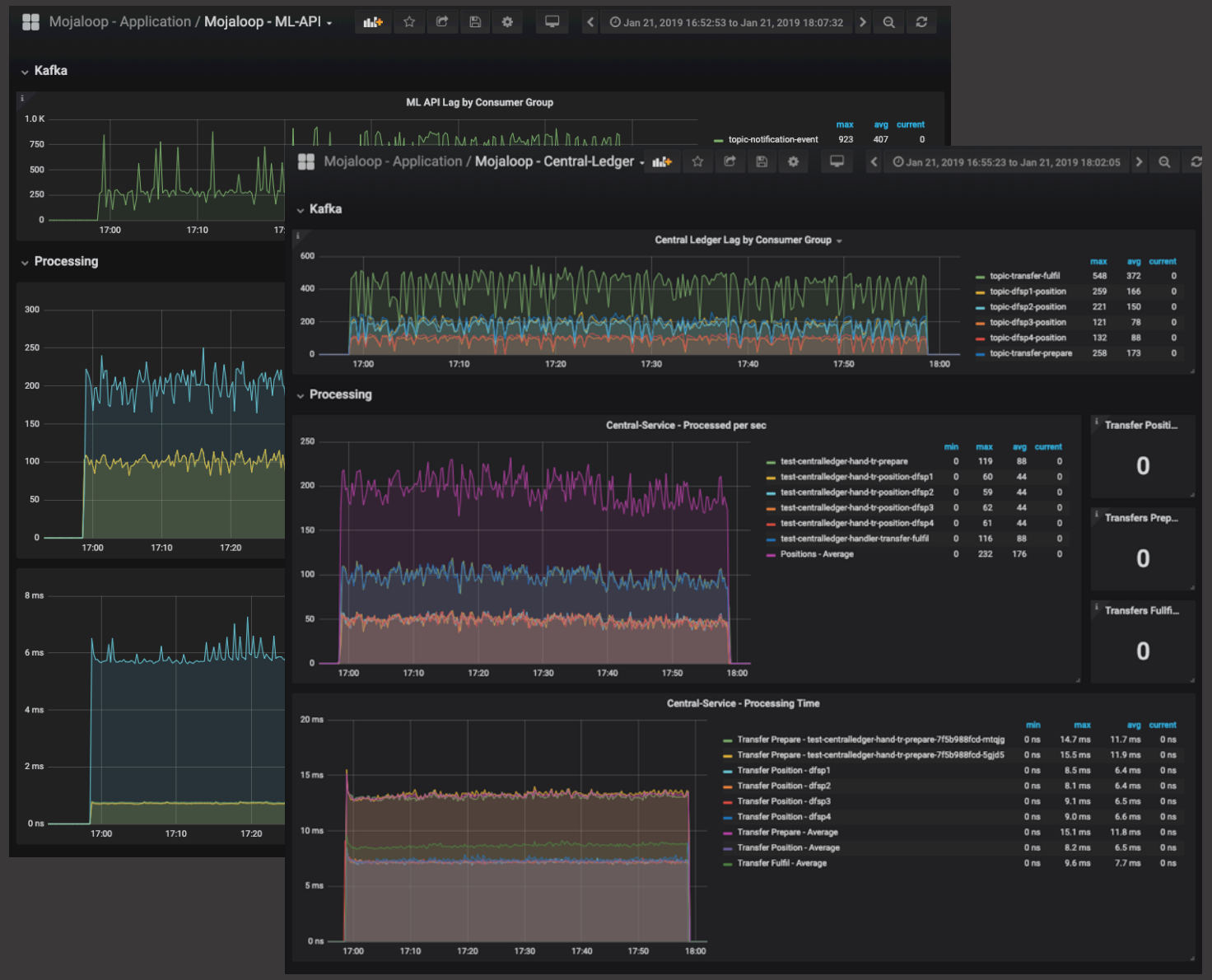
```
metrics:
    enabled: true      <-- By default this has been 'disabled'
```

## Metrics Supported Components:

| | | |
|---|---|---|
| **Mojaloop-API-Adapter** (Transfer Prepare & Fulfil) | **Mojaloop-API-Adapter** (Handler - Notifications) | |
| **Central-Services** (Handler - Prepare) | **Central-Services** (Handler - Positions) | **Central-Services** (Handler - Fulfil) |
| **Simulator** (QA Simulator) | | |

March 2020

# Ops Metric Instrumentation – Dashboards




## Documentation

http://mojaloop.io/helm/monitoring/

## Mojaloop Application

- ML-API-Adapter -- nodejs + application
- Central-Ledger -- nodejs + application
- Simulators -- nodejs + application

## Data Store

- MySQL Overview
- PXC Galera Overview
- PXC Galera Graphs

## Messaging

- Kafka Cluster Overview
- Kafka Topic Overview

## Kubernetes

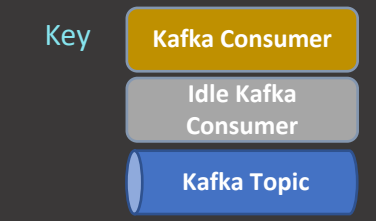
- Clusters
- Deployments

**DEMO**

# Appendix

## Pre-PI-8 Performance Optimizations

# Optimization – Position Handler Processing Modes

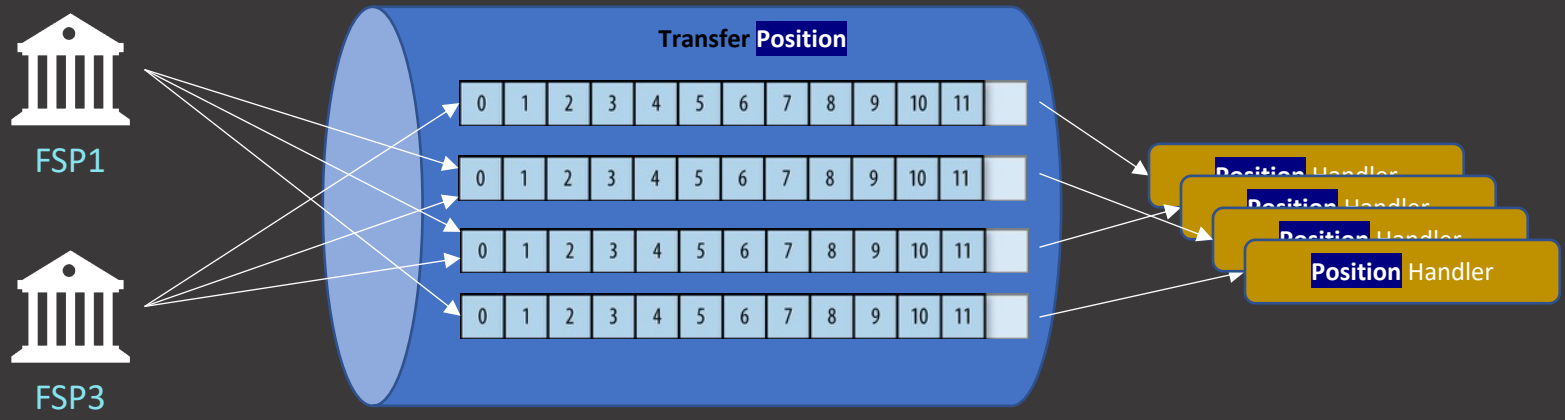



## Pre-filtering of DFSPs into Buckets

- Partition strategy is keyed for Positions.
- Message keys are used to route each message to a specific Partition (Bucket).
- Dedicated handlers for DFSP specific workloads (Future: Easier In-memory processing)
- *Possible issue/limitation*:
  - Limited scalability by number of FSPs
  - Possible idle handlers (cost implication)
  - Requires process for increasing partitions on Kafka (possible downtime, migration strategy, or temporal performance hit).


## Mixed DFSP Processing

- Partition strategy can be configured as Keyed (same as PI4) or Random
- Improved scalability (no limit)
- All Handlers are utilized (efficient hardware usage)
- Future possibility of consolidating handlers (prepare, position & fulfil)
- *Possible issue/limitation*:
  - Increased contention & locking on the database persistence layer.



# Optimization – IO

## Description:

- Slow response times observed on Handlers due to high IO

## Resolution:

- Handlers:
  - Manual Commits were causing excessive high IO thread contention which was drastically reduced by enabling auto-commits
  - Optimized Kafka configurations to reduce unnecessary IO
  - Loggers were moved into an async process (20% improvement)

## Considerations:

- Additional logic will need to be added in Handlers as the auto-commits allows for re-processing of messages. The handlers will not re-apply any business logic as the "state" of the transactions will have moved on, however it is possible that there will be unwanted notifications sent to the FSPs. Thus we will have to store the Kafka index for each message to validate if the message is a re-request from FSP or a re-processing of a previous message.

## PoC Producer - Commit = Manual

**Number** of unique matched entries: **10000**
Total difference of all requests in milliseconds: 1308
Shortest response time in millisecond: 0
Longest response time in millisecond: 3
**Mean**/The average time a transaction takes in millisecond: **0.1122**
Standard **deviation** in milliseconds: **0.3206**
Number of entries that took longer than a second: 0
% of entries that took longer than a second: 0.00%
Average **transactions per second**: **7645.26**

## PoC Producer - Commit = Auto

**Number** of unique matched entries: **10000**
Total difference of all requests in milliseconds: 1234
Shortest response time in millisecond: 0
Longest response time in millisecond: 3
**Mean**/The average time a transaction takes in millisecond: **0.1023**
Standard **deviation** in milliseconds: **0.3080**
Number of entries that took longer than a second: 0
% of entries that took longer than a second: 0.00%
Average **transactions per second**: **8103.73**

## PoC Consumer - Commit = Manual

**Number** of unique matched entries: **10000**
Total difference of all requests in milliseconds: 16154
Shortest response time in millisecond: 0
Longest response time in millisecond: 3
**Mean**/The average time a transaction takes in millisecond: **0.1116**
Standard **deviation** in milliseconds: **0.3187**
Number of entries that took longer than a second: 0
% of entries that took longer than a second: 0.00%
Average **transactions per second**: **619.04**

## PoC Consumer - Commit = Auto

**Number** of unique matched entries: **10000**
Total difference of all requests in milliseconds: 8852
Shortest response time in millisecond: 0
Longest response time in millisecond: 3
**Mean**/The average time a transaction takes in millisecond: **0.1018**
Standard **deviation** in milliseconds: **0.3063**
Number of entries that took longer than a second: 0
% of entries that took longer than a second: 0.00%
Average **transactions per second**: **1129.69**

1. Partially Implemented in PI3 Master Branch (async Loggers were deferred by DA →
   a. https://github.com/mojaloop/design-authority/issues/11
   b. https://www.dropbox.com/s/cu9vam5jlqsea35/Screenshot%2020199-10-09%2012.31.55.png?dl=0
2. DA decision was to defer async logging until the auditing framework (e.g. Event Framework) was in place.
3. Sync vs Async logging can be configured by each Components & Helm chart

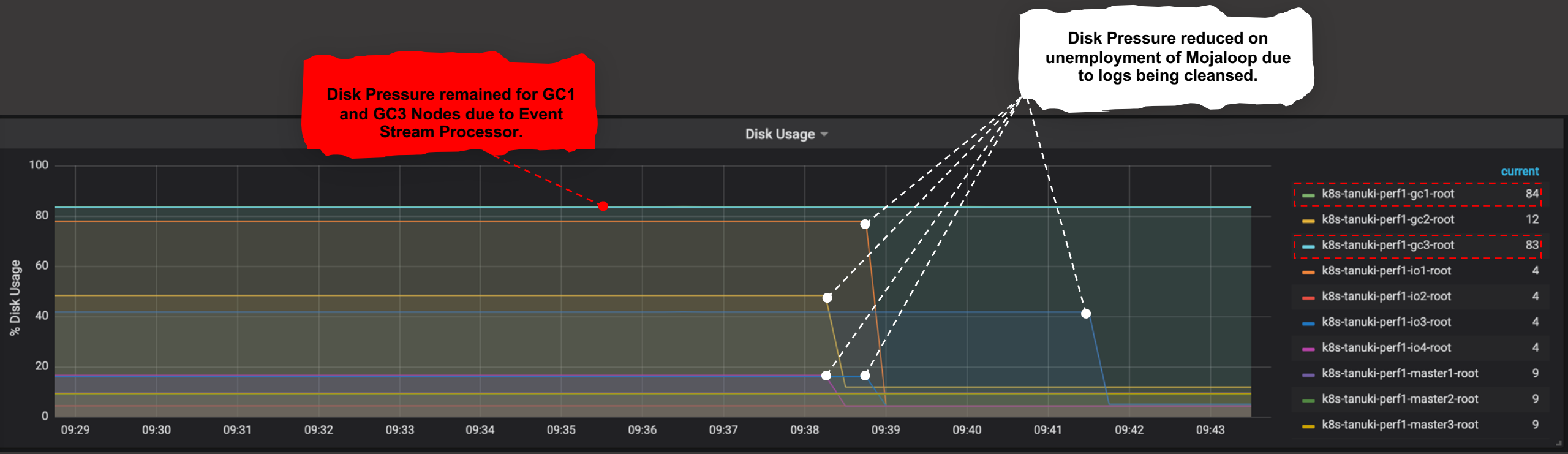
# Optimization – K8S Disk Pressure

## Description:

- High disk space usage on specific K8S Node due to excessive log file sizes.
- FluentD (Log collector) process experienced issues in collecting large log files
- Verbosity of Event logging identified to be the main cause

## Resolution:

- Event-Sidecar: Verbosity of Event Stream logs reduce to display meta-data only.
- Event-Stream-Processor: Verbosity of Event Stream logs reduce to display meta-data only.



# Optimization – General Resolved Issues

## Caching of Long-Lived-Temporal Data

1. Optimization – Caching of FSP Participant Information. ← Reduction in DB QPS

## SDK-Schema-Adapter

1. Stability - Connection Management for Redis Cache (https://github.com/mojaloop/sdk-scheme-adapter/pull/103)
2. Stability - Timeout subscribers failing (https://github.com/mojaloop/sdk-scheme-adapter/pull/108)
3. Stability - Redis cache data retention issue (https://redis.io/commands/expire - https://github.com/mojaloop/project/issues/1134)

## Central-Services-Stream

1. Stability - Kafka Client Producers were failing with “Full Queue Error” (https://github.com/mojaloop/project/issues/1114)

## Grafana Dashboard Metric Miss-match

1. Monitoring – Fixed metric calculation on JMeter Dashboard for TPS calculation (did not match ML-API rate)