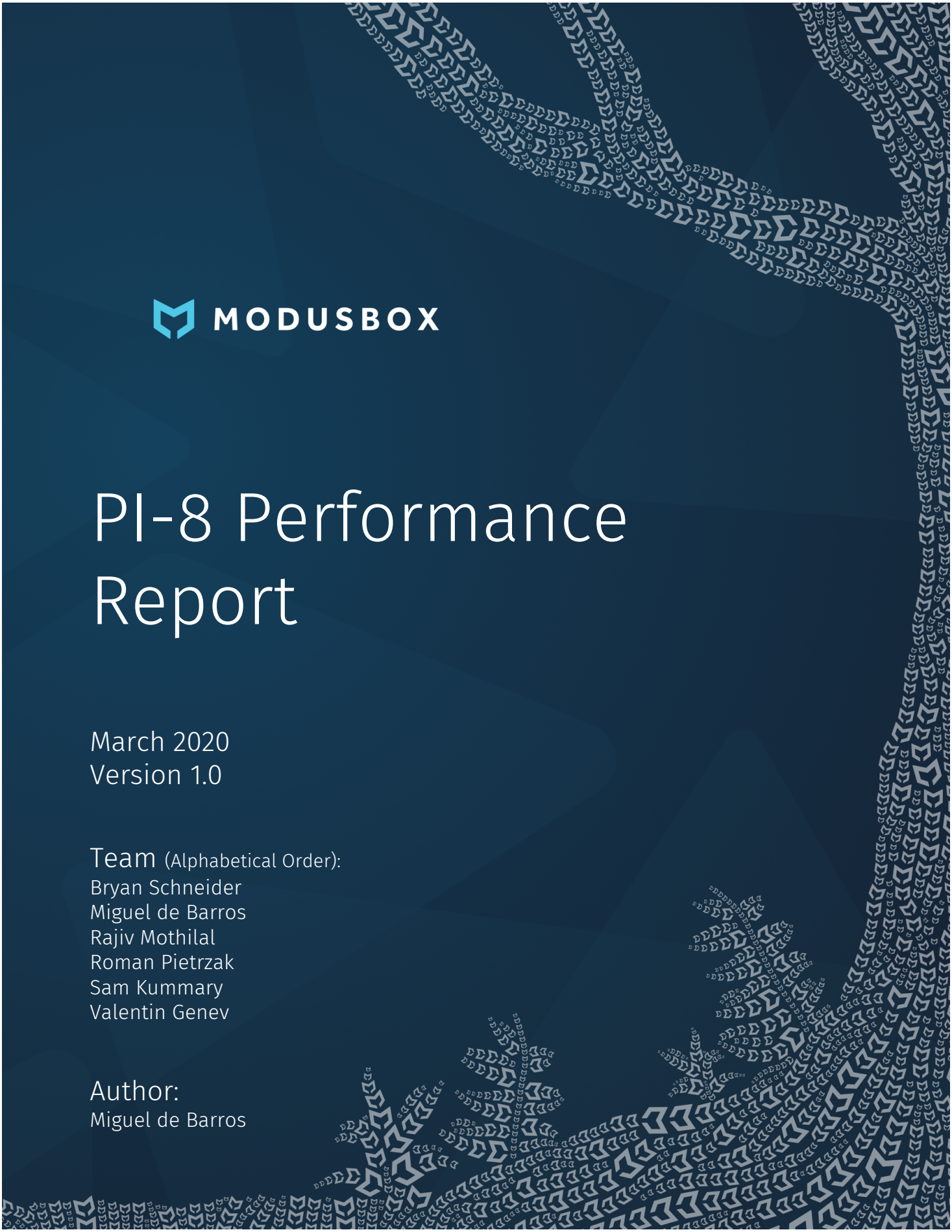


# PI-8 Performance Report

March 2020
Version 1.0

Team (Alphabetical Order):
Bryan Schneider
Miguel de Barros
Rajiv Mothilal
Roman Pietrzak
Sam Kummary
Valentin Genev

Author:
Miguel de Barros

# 1. Executive Summary

The Modusbox Team has spent much effort over PI-8 and PI-9 in understanding the performance characterization and capabilities of the Mojaloop system.

The goal being to prove that the Mojaloop system is able to achieve the following performance targets at a minimum:

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

*Table 1. Performance Targets Indicators*

In this report you will see the results of this effort, and that the Mojaloop System is able to surpass the above targets.

## 1.1. Approach

Several key activities took place during this effort:

1. Establish an environment that can be used for Performance Testing
2. Characterize the following Performance Test-Cases:
    a. Financial Transaction End-to-end
    b. Financial Transaction Prepare-only
    c. Financial Transaction Fulfil-only
    d. Individual Mojaloop Components
3. Instrument additional metrics to assist in the characterization of the Mojaloop System, and to identify bottlenecks
4. Implement quick-win performance improvements
5. Reporting on identified improvements in the following areas:
    a. Code
    b. Architecture
    c. Deployment (i.e. infrastructure)
6. Document Key Learnings

# 2. Environment

All environments are managed and provisioned by the existing Tanuki Rancher Services within the Mojaloop Open-Source AWS space. This allowed the environment to be flexible enough to scale as required to accommodate different performance test scenarios.

The following distinct Kubernetes environments were setup for the performance work:

a. Perf1-tanuki
b. JMeter-tanuki

| Application | Version |
|---|---|
| Kubernetes | 1.11.6 |
| Rancher | 2.1.6 (upgraded to 2.3.5) |
| Docker | 17.03.2ce |
| Operating System | Ubuntu 16.04.4 LTS |

*Table 2. Environment Versions*

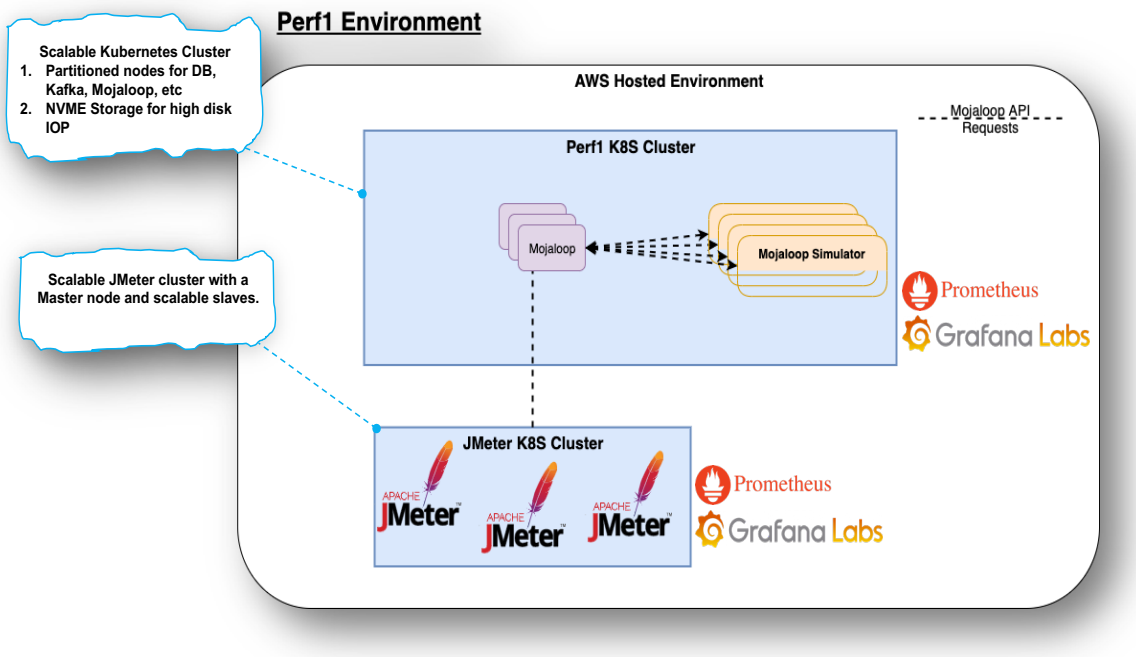


*Figure 1. AWS Hosted Environments*

## 2.1. Perf1-tanuki Kubernetes Cluster

This environment is dedicated to hosting all Mojaloop components, and any additional operational components (e.g. Prometheus, Grafana) that directly support them.

Specific compute pools were setup to isolate the workload by I/O and CPU requirements based on:

- Master Nodes – m4.large: Kubernetes Management Services
- General Nodes – i3.xlarge: NodeJS Micro-Services (e.g. Mojaloop components)
- I/O Nodes – i3.xlarge: Datastore Services (e.g. Kafka, MySQL, etc)
- Operational Nodes– m4.xlarge: Support Services (e.g. Prometheus, Grafana)

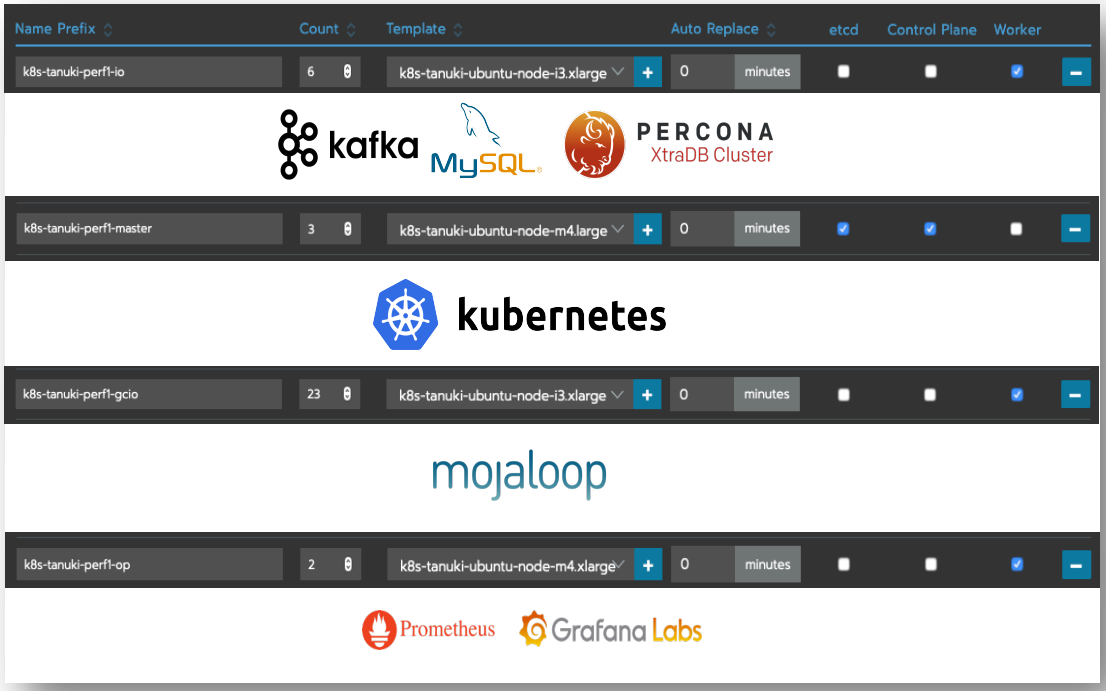


*Figure 2. Perf1-Tanuki Cluster*

## 2.2. JMeter-Tanuki Kubernetes Cluster

This environment was setup to host a JMeter cluster to drive the performance testing from an API perspective.

Only a single compute pool of m4.xlarge was setup which hosted both the Kubernetes management services and the JMeter workload. This proved to be adequate for our needs.

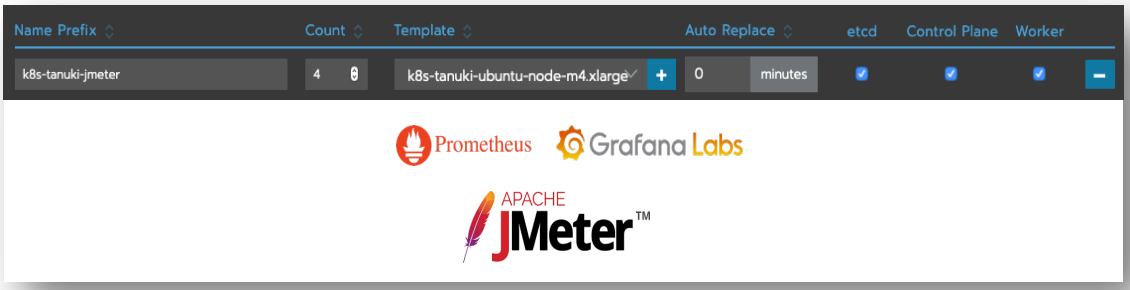


*Figure 3. JMeter-Tanuki Cluster*

# 3. Performance Test-Cases

## 3.1. Financial Transaction End-to-end

Table 3. shows the final End-to-end results for a Performance test executed against 8 FSPs, for a sustained period longer than the required 1-hour period:

| Metric | *Value | Comments |
|---|---|---|
| Financial Transactions per second (FPS) | 300 FPS | 300 FPS is well over the T1 indicator of 200 FPS. |
| Duration of Test | 4800 seconds (80m) | Executed test longer than 60m to ensure consistency. |
| Total Financial Requests | 1441897 | Currently calculated from DB after the final Commit (Position Handler). |
| Total Requests > 1s | 369 | Currently calculated from DB after the final Commit (Position Handler). |
| % of Requests > 1s | 0.03% | Currently calculated from DB after the final Commit (Position Handler). |
| Average Financial Transaction Time | 0.1698 | Currently calculated from DB after the final Commit (Position Handler). |
| Minimum Financial Transaction Time | 0.1s | Currently calculated from DB after the final Commit (Position Handler). |
| Maximum Financial Transaction Time | 6s | Currently calculated from DB after the final Commit (Position Handler). |

*Table 3. Financial Transaction End-to-End Results*

The above results were attained running the Mojaloop components with the following pod and hardware scaling factors:

| 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 |

*Table 4. Financial Transaction End-to-End scaling config*

In addition, all similar workloads would run on the same dedicate nodes (i.e. Prepare Handlers would be scaled over the same shared 3x Nodes (Physical Machines) within the Kubernetes Cluster.

### 3.1.1. Performance Enhancements

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

### 3.1.2. Post Transfers from JMeter

The following JMeter configuration in table 5. was used:

| 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 | |

*Table 5. Financial Transaction End-to-End JMeter Config*

Figure 4. shows the Average Response Time and Transactions per Second as captured from the JMeter metric dashboards:

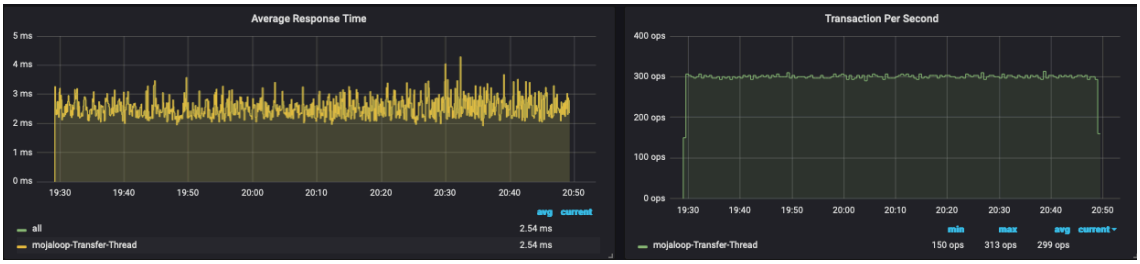

*Figure 4. Financial Transaction End-to-End JMeter Metrics*

### 3.1.3. Component Operations Per Second

The following two figures show the average Operations per Second for each of the Mojaloop Handlers. Figure 5, specifically for the ML-API-Adapter and the Notification Handler, with Figure 6 being for the Central-Services (e.g. Prepare, Position, and Fulfil Handlers).

In Figure 5 the metrics for the POST Transfer and PUT Transfer operations are represented by 'Transfers API Prepare' and 'Transfers API Fulfil' which are both running at 300ops.

We have also added a metric to capture the delivery OP/s via the Notification Handler to the FSP Simulator for both the Prepare and Fulfil paths as represented by the 'prepare' and 'commit' actions. This results in each achieving more than 300ops matching our JMeter throughput.

The 'Notification Event' representing the actual Notification Handler metric achieved double our JMeter throughput at 626ops. The reason for this being that it is an aggregate of both the Prepare and Fulfil flows.

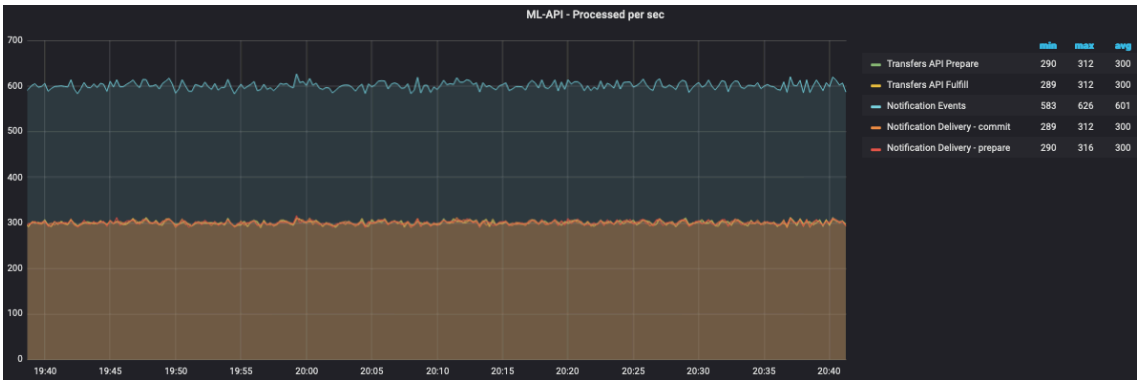


*Figure 5. Financial Transaction End-to-End Ops/s for ML-API-Adapter*

As expected, we see Prepare 'centralledger-handler-transfer-prepare' and Fulfil 'centralledger-handler-transfer-fulfil' handlers running at 300ops in Figure 6 once again matching the JMeter throughput.

Similarly, as discussed for Figure 5, the Position 'centralledger-handler-transfer-position' metric represents both the Prepare and Fulfil flows running at 600ops. We have also split Position metrics by the Prepare and Fulfil paths as represented by the 'prepare' and 'commit' actions. These are each running more than the 300ops as expected.

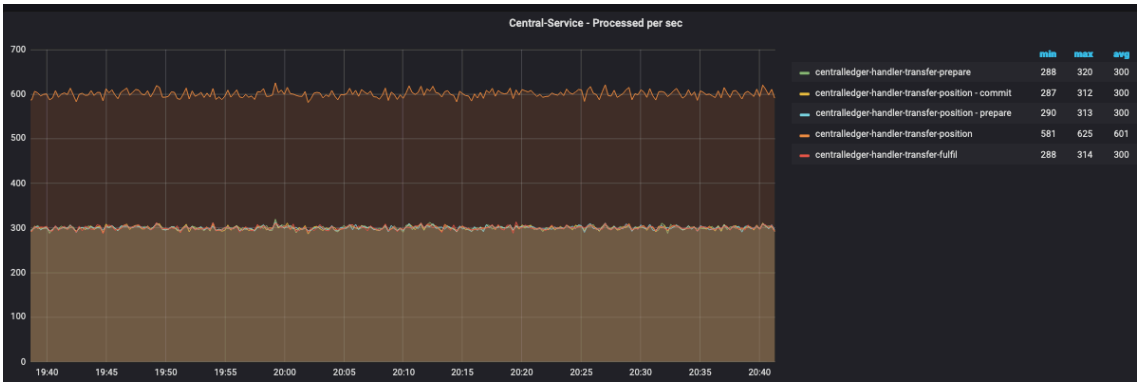


*Figure 6. Financial Transaction End-to-End Ops/s for Central Services (Central-Ledger)*

### 3.1.4. Component Processing Time

Figure 7 shows the processing time for each of the ML-API-Adapter components. The API Service component represented by the 'Transfer API Prepare' and 'Transfer API Fulfil' metric show minimal processing overhead.

The 'Notification Events' which represents the Notification Handler is the most expensive operation here sitting at around 5-7.5ms on average.

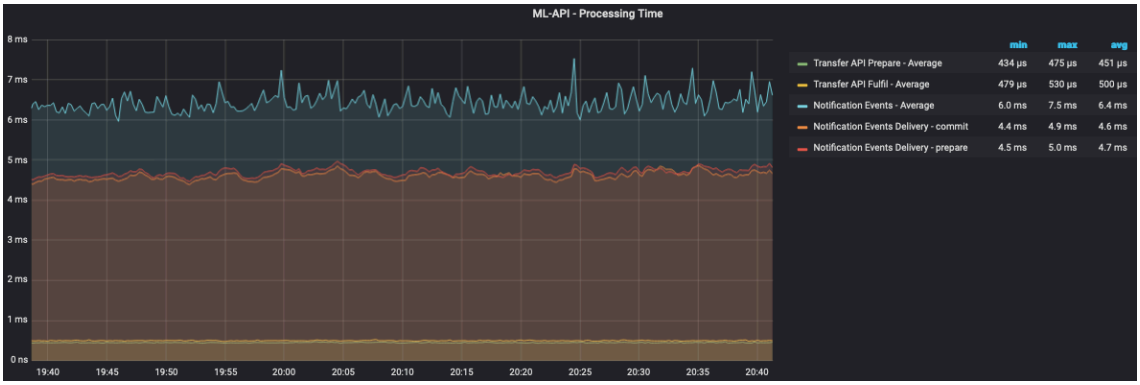


*Figure 7. Financial Transaction End-to-End Processing Time for ML-API-Adapter*

The processing time for Prepare, Fulfil and Position handlers are close at around 22-26ms, with the Position Handler proving to be the most expensive in Figure 8. It's worth noting here the slight increase in process times from the start to the end of the hour run. Some additional testing should be done on longer sustained runs to see if this trend is maintained.

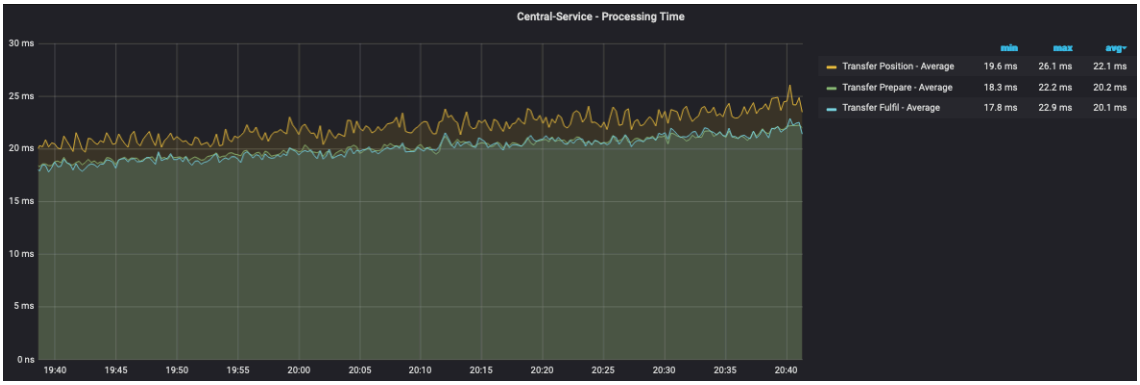


*Figure 8. Financial Transaction End-to-End Processing Time for Central Services (Central-Ledger)*

### 3.1.5. Messaging Topic Lag

Auto-commits were enabled due to the I/O contention introduced when using manual commits. As such the auto-commit interval was set to 100ms for this test, which would result in 30 transactions being committed after the interval elapsed.

The result being that 30 transfers could be re-processed as a duplicate-transfer event as the state of each transfer is validated. The Kafka index should be stored as part of the Transfer record and validated as part of the transfer state to fully circumvent this issue.

Figure 9. shows the sustained lag for each of Mojaloop's topic over the hour period being well under-control.

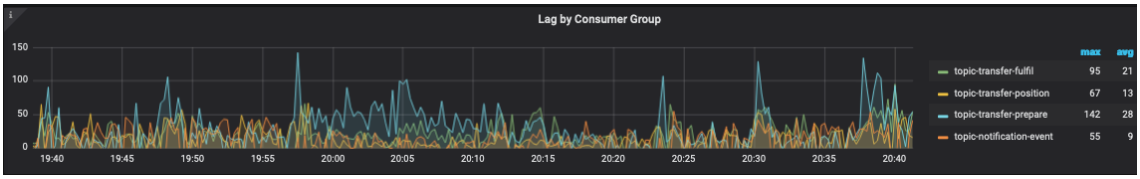


*Figure 9. Financial Transaction End-to-End Message Topic Lag*

## 3.2. Financial Transaction Prepare-only

The Prepare-only and Fulfil-only tests were executed by pre-loading the prepare or the fulfil topics and ensuring that the Prepare, Fulfil and Notification Handlers were disabled.

The desired Handler would then be enabled, allowing us to observe the isolated performance for the Central-Services for each of the legs.

Figure 10 shows the Prepare flow, with the Prepare and Position Handlers achieving 408tps.

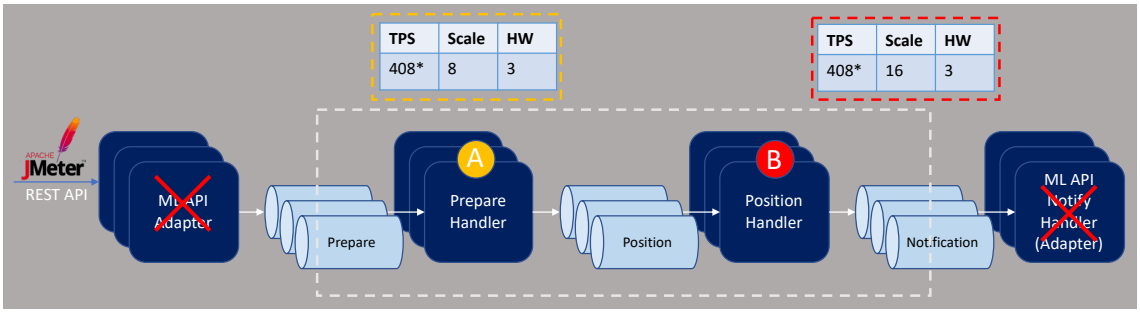


*Figure 10. Financial Transaction Prepare-only Flow (*Caching Disabled)*

## 3.3. Financial Transaction Fulfil-only

The Fulfil-only flow was able to achieve a slightly higher TPS at 539 for both the Fulfil and Position Handlers in comparison to Prepare-only flow (ref: Figure 10).

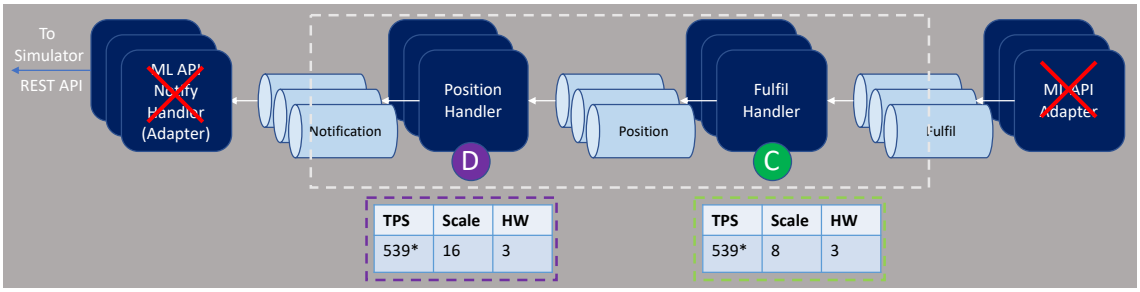


*Figure 11. Financial Transaction Fulfil-only Flow (*Caching Disabled)*

## 3.4. Individual Mojaloop Components

### 3.4.1. Mojaloop Services & Handlers

Figure 12 shows the individual characterization of Mojaloop's individual components for the Transfer Prepare and Fulfil flows. Each component was tested in isolation by ensuring that down-stream components were disabled, and topics were pre-loading where applicable.

Note: The data collected in Figure 12 was against the un-optimized baseline, which does not include the performance enhancements listed in 3.1.1.

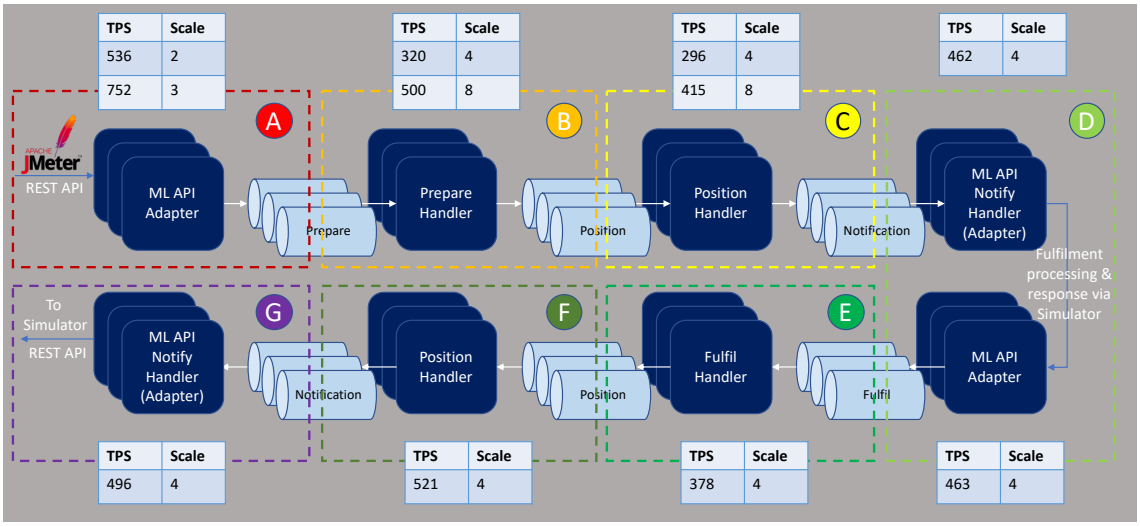


*Figure 12. Individual Mojaloop API And Handler TP/S*

## 3.4.2. Message Streaming Architecture and Libraries

As part of the characterization effort, the Central-Services-Streaming library was tested in a similar architectural configuration to Mojaloop. The outcome of these results indicate that Central-Services-Stream and Kafka are able to maintain a respectable 34,000tps for concurrent reads and writes.

Figure 13, shows a test driven by a pre-loaded Prepare topic. The simulated handlers are reading messages from a topic and writing to the next topic as depicted in the Figure. Similarly sized Mojaloop messages were used for this test.

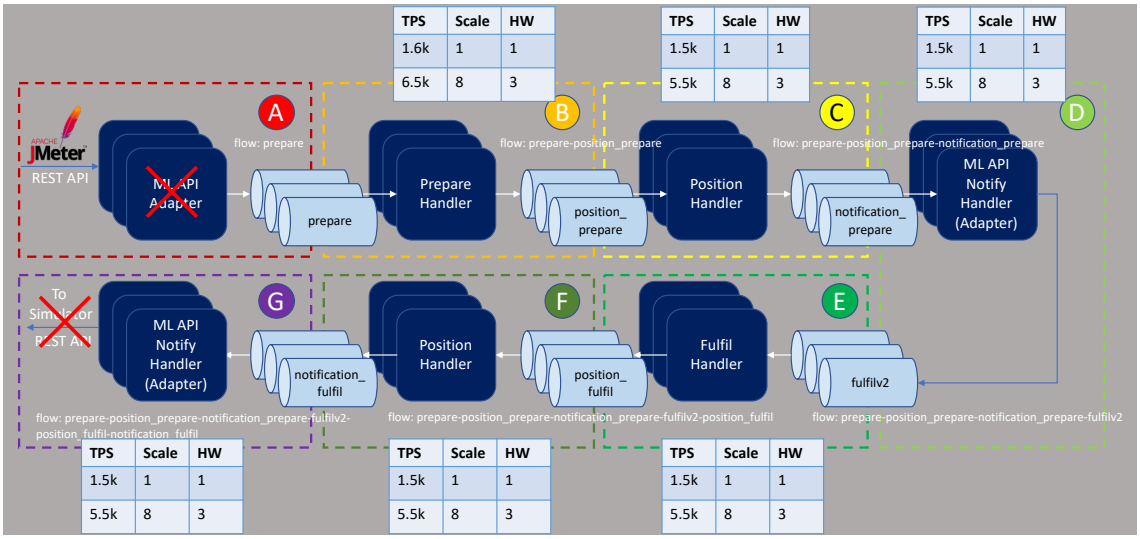


*Figure 13. Simulated Mojaloop Architecture Message Streaming TP/S*

### 3.4.3. Database

For the Database Characterization, an isolated Prepare Handler was re-factored to generate random Transfers with no dependency on Kafka for the ingestions of messages. This allowed us to identify a core resource constraint with regard to DB and IO Throttling.

Figure 14 shows a fairly common occurrence during this performance testing effort which has proven to be difficult to diagnose. In the first diagram we have an isolated Prepare Handler which is scaled from 1 to 8 pods (ref Figure 16.) mid-run.

The Op/s climbs from just over 100ops to a max of 655ops, at which point we see a drastic throttling back down to around 60ps.

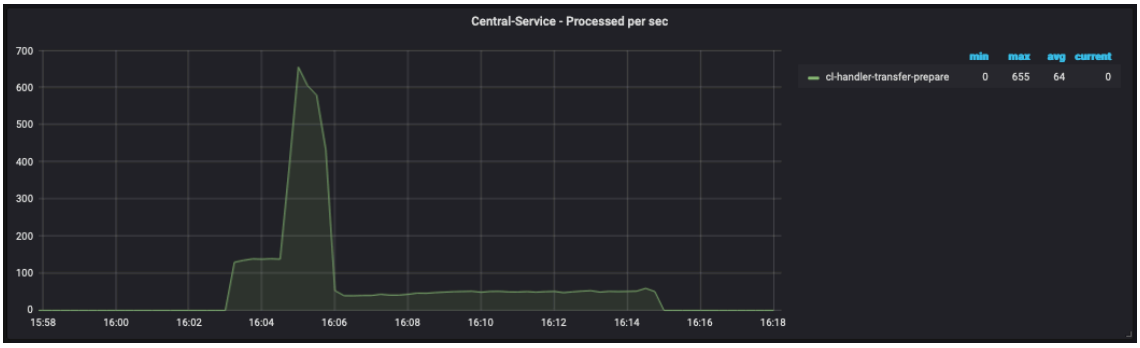


*Figure 14. Isolated Prepare Handler Op/s*

In Figure 15, we can see that the Processing Time for the Handler was under 50ms, which was sustained more or less until the scale of 8 pods was achieved. The Processing Time increased above 150ms as throttling was enforced by AWS.

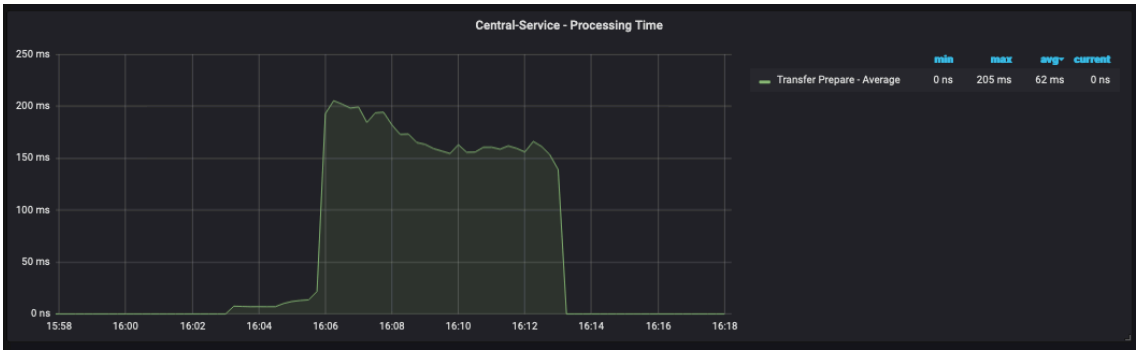


*Figure 15. Isolated Prepare Handler Processing Time*

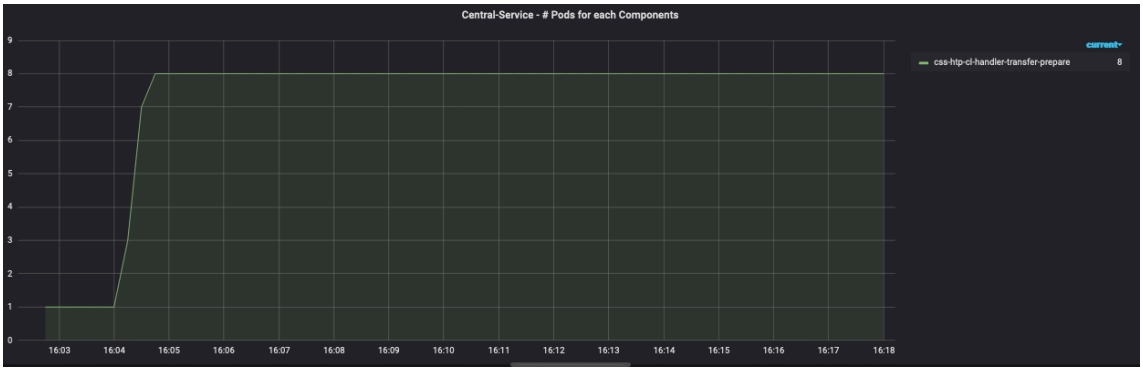


*Figure 16. Isolated Prepare Handler Pod Scale*

We were able to prove the AWS Platform as the cause for the throttling by observing the IOPS Burst Budget metric for our Database provisioned AWS Block Storage Volume as shown in Figure 17. The throttling is enforced once the budget hits 0% during the test interval.

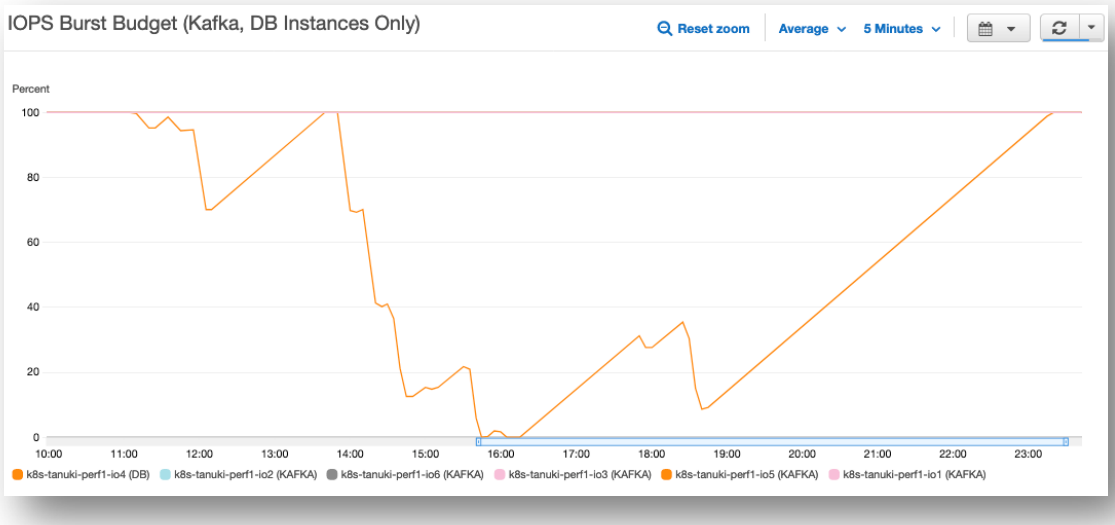


*Figure 17. CloudWatch Metrics for Perf1-Tanuki Environment IOPS Burst Budget*

The result being that we have temporarily provisioned an AWS Managed SQL Database (RDS) to test & demonstrate how Mojaloop behaves against an optimized MySQL deployment, with guaranteed provisioned 2000 IOPS (i.e. no IOPS Burst Budget or Throttling). We would expect the results in 3.1 to improve proportionally to the amount of guaranteed IOPS provisioned in conjunction with a scaled Mojaloop deployment.

Future effort will need to spend on performance tuning an on-prem MySQL cluster deployment as part of a Mojaloop deployment to achieve similar or better results to the RDS deployment on the AWS platform. These learnings would then be documented, and serve as a basis for other target platforms.

# 4. Conclusion

This report has documented our current understanding of Mojaloop's Performance Characterization and capabilities by the following approach:

1. Established an environment that can be used for Performance Testing (ref: 2)
2. Characterized the following Performance Test-Cases:
    a. Financial Transaction End-to-end (ref: 3.1)
    b. Financial Transaction Prepare-only (ref: 3.2)
    c. Financial Transaction Fulfil-only (ref: 3.3)
    d. Individual Mojaloop Components (ref: 3.4)
3. Instrumented additional metrics to assist in the characterization of the Mojaloop System, and to identify bottlenecks (ref: 3.1.1)
4. Implemented quick-win performance improvements (ref: 3.1.1)
5. Reported on identified improvements in the following areas:
    a. Code (ref: 3.1.1)
    b. Architecture (ref: 3.1.1)
    c. Deployment (i.e. infrastructure) (ref: 3.1.1)
6. Documented Key Learnings (ref 3.1.1, 3.4.3)

It has also shown that the goals outlined in Table 1. (reproduced below) were achieved and surpassed as described in section 3.1:

| # | Indicator | Target | Achieved |
|---|---|---|---|
| T1 | Financial Transactions Per Second | > 200 fps | **300** |
| T2 | Time for performance run | >= 1 Hour sustained run | **80m** |
| T3 | % of transactions that took longer than a second | < 1% | **0.03%** |

*Table 6. Performance Targets Indicators (ref: Table 1)*