mojaloop foundation

PI-11 Scalability PoC update

October 2020

Milestones

1st - Prove scalability according to guiding principles

2nd - Introduce Event-Sourcing and CQRS

3rd - Report out document

Overview

Re-designed transfers and participants interactions using a distributed transaction

Built a distributed event-based transaction system based on the single responsibility principle - split transfers and participants

Implemented Event-Sourcing and CQRS

mojaloop foundation

Where we are since PI-10?

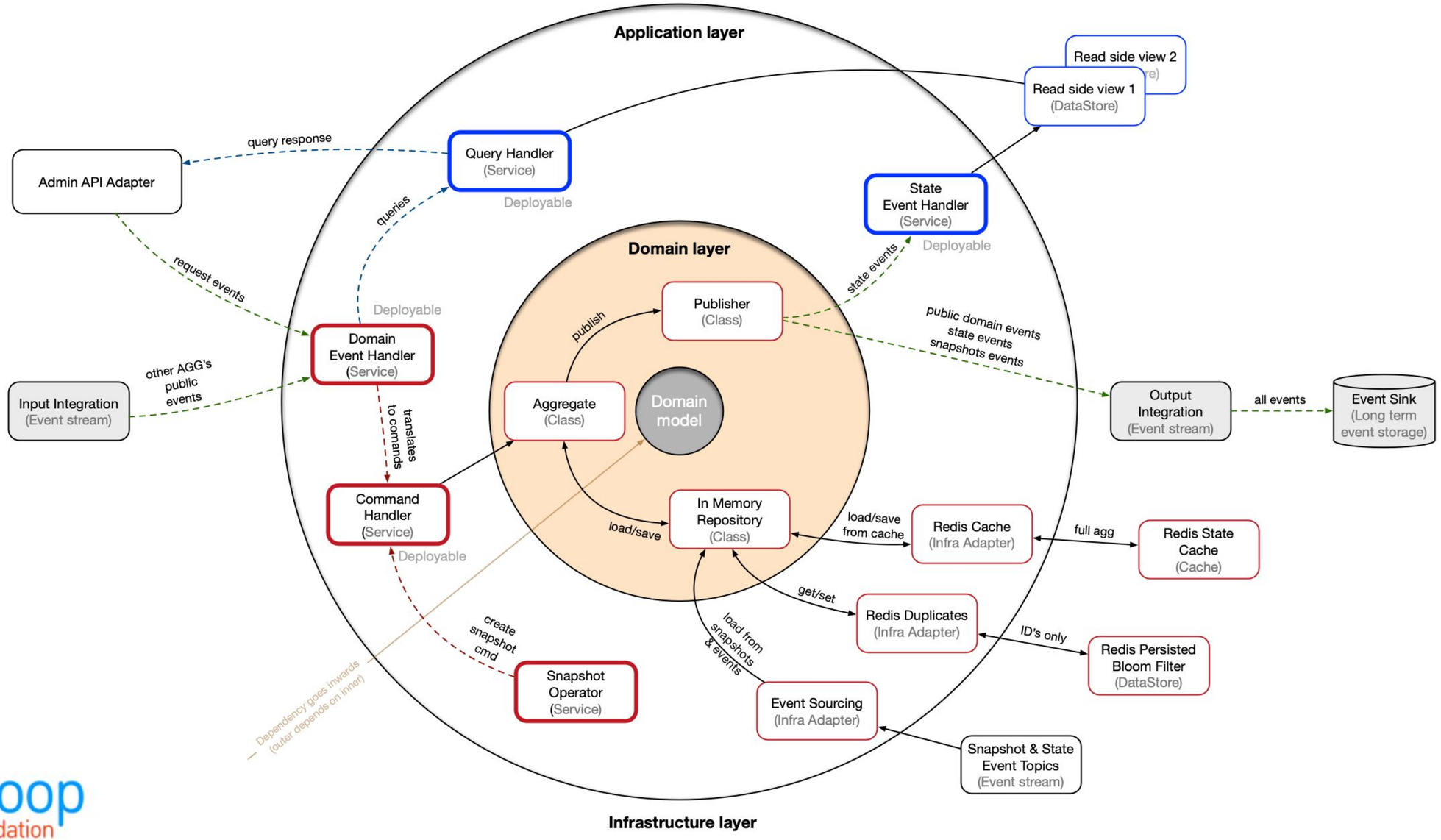
~~Properly test CQRS and Event Sourcing and execute load tests~~

~~Write a CQRS handler to keep necessary DB Data up to date~~

~~Test scaling to 32 or more FSPs (or up to 5000 FTPS)~~

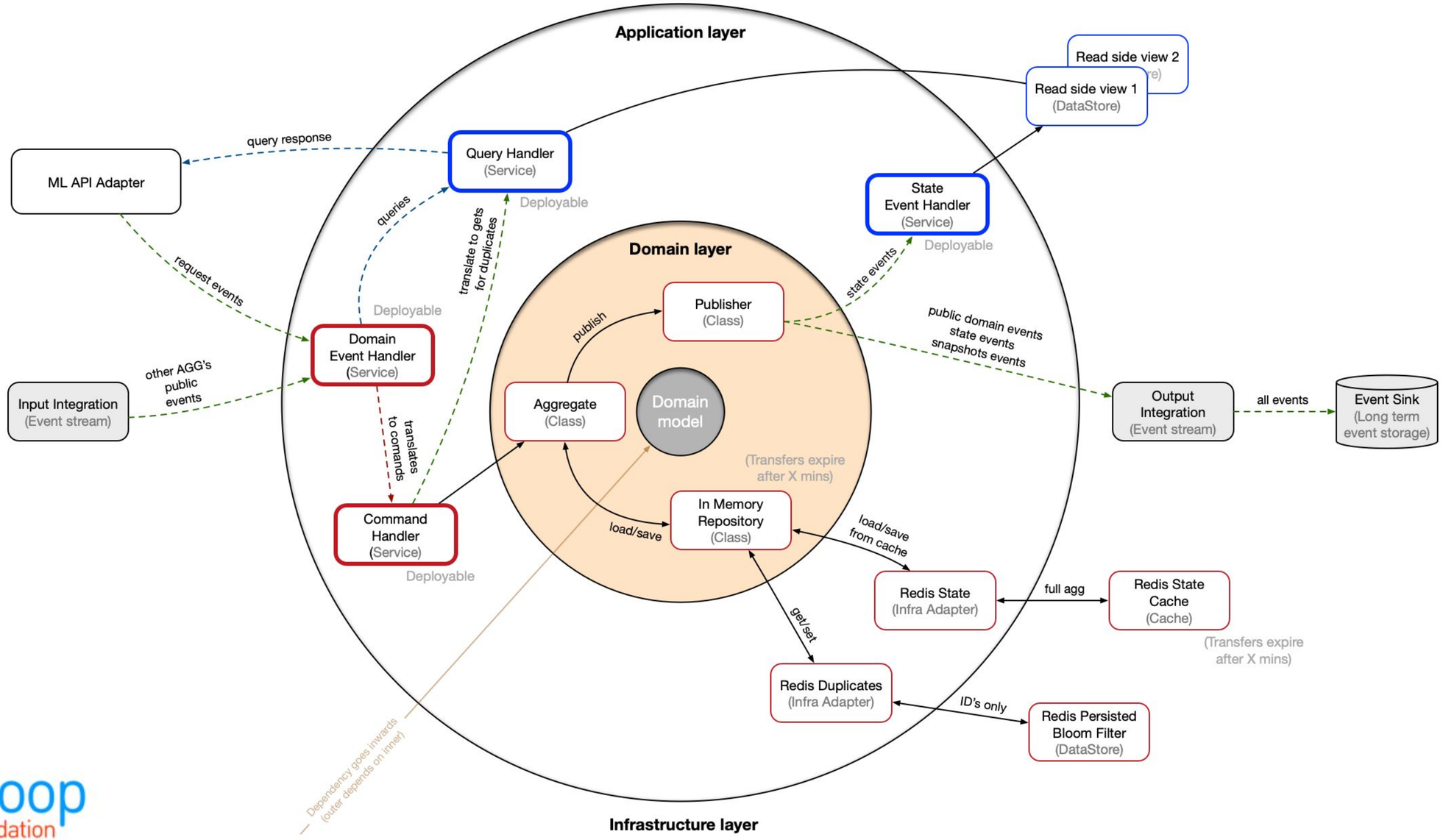
~~“Performance Scalability PoC Report”~~

Architecture Overview (Participants)



mojaloop foundation

Architecture Overview (Transfers)



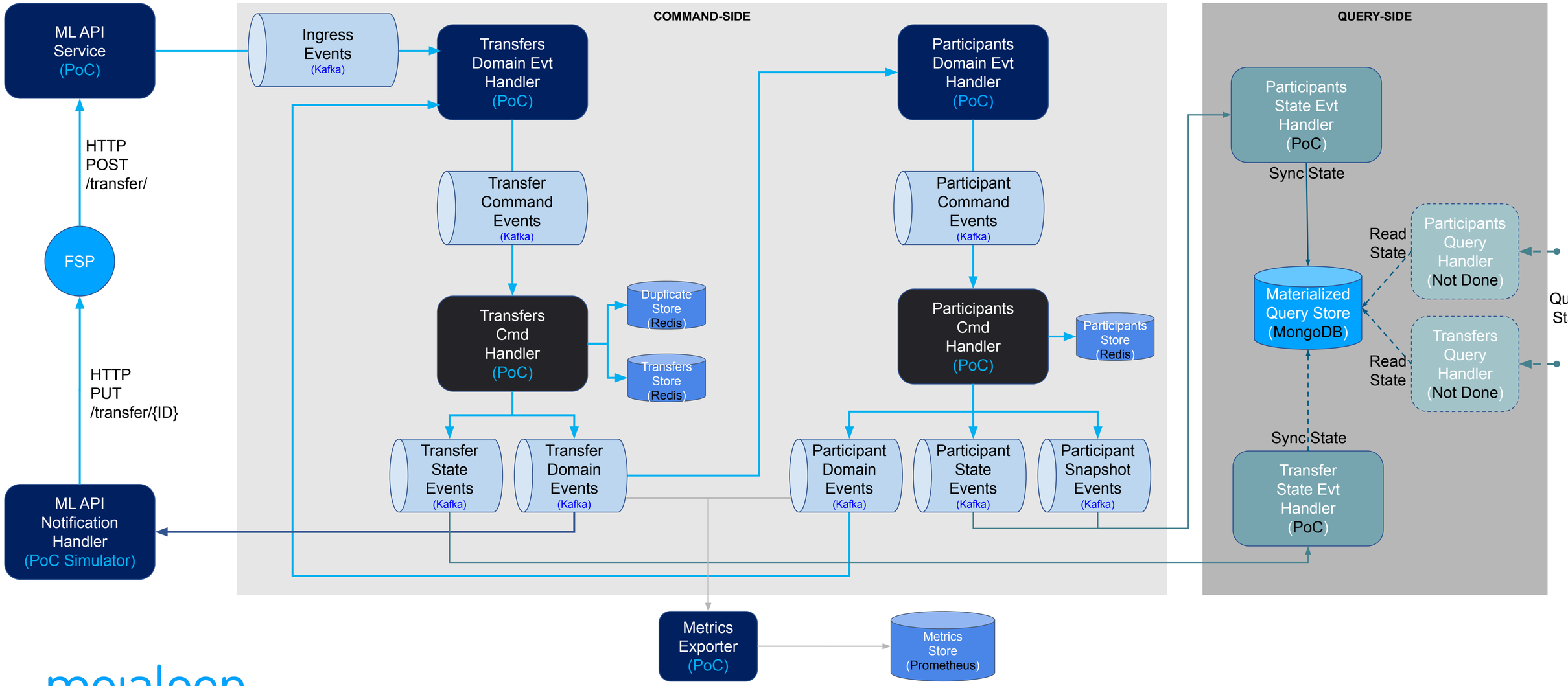
mojaloop foundation

How it flows (recap)

mojaloop foundation

Deployment Architecture

Available deployable Artifacts - Helm, Docker & Github



mojaloop foundation

Performance Summary

T#	Indicator	Cost Scale Unit (CSU)	8 FSP (ECache)	16 FSP (ECache)	32 FSP (ECache)
T3.1.1	Financial **Transactions Per Second** (FP/s) - **1000/5000 FP/s**	**1.001 FP/s**	**1.602k FP/s**	**3.002k FP/s** (+87%)	**5.12k FP/s** (+71%)
T3.1.2	**Time** for Run - **1H**	**1H 20M**	**1H 20M**	**1H 20M**	**1H 20M**
T3.1.3	**% of Transactions** that took **longer than a second - 1%**	**0.03%**	**0%**	**0%**	**0.22%**

Table 3.1.a. Performance Targets Summary

M#	Metric	Cost Scale Unit (CSU)	8 FSP	16 FSP	32 FSP
M3.1.1	Financial Transactions **Average** Response Time	**79 ms**	**63 ms**	**72 ms**	**43 ms**
M3.1.2	Financial Transactions **Minimum** Response Time	**56 ms**	**50 ms**	**57 ms**	**33 ms**
M3.1.3	Financial Transactions **Maximum** Response Time	**135 ms**	**98 ms**	**102 ms**	**81 ms**
M3.1.4	**Total** Financial Transactions	**3.662 mill**	**5.81 mill**	**11.05 mill**	**18.51 mill**

Table 3.1.b. Non-Indicator Performance Metrics Summary

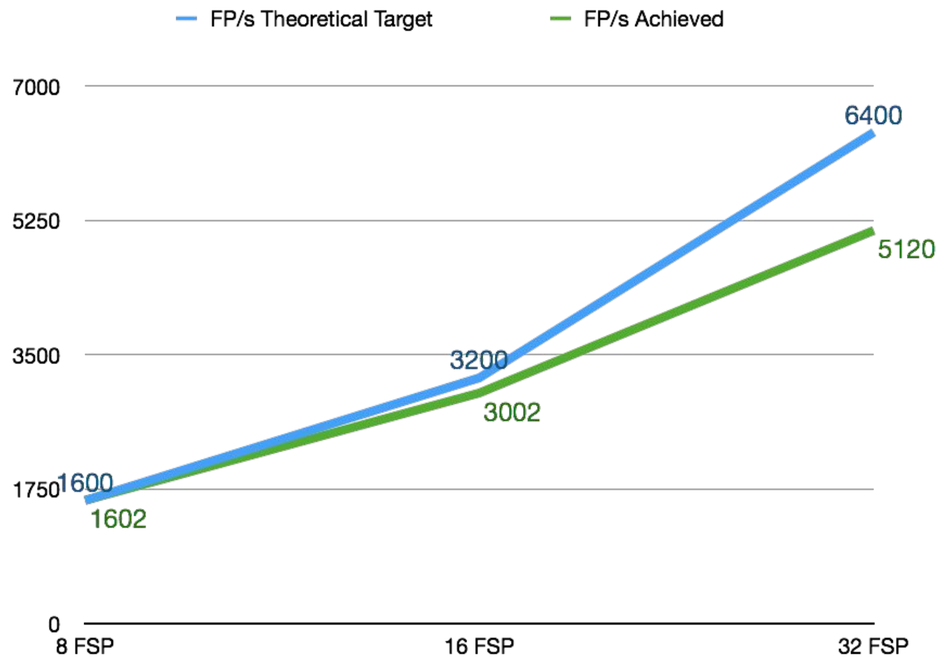


Figure 3.1.a. Scalability Comparison of Achieved Results against S# Targets using AWS Managed ElasticCache (ECache)

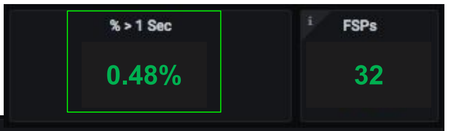
Performance Run: 32 FSPs (**self-hosted)

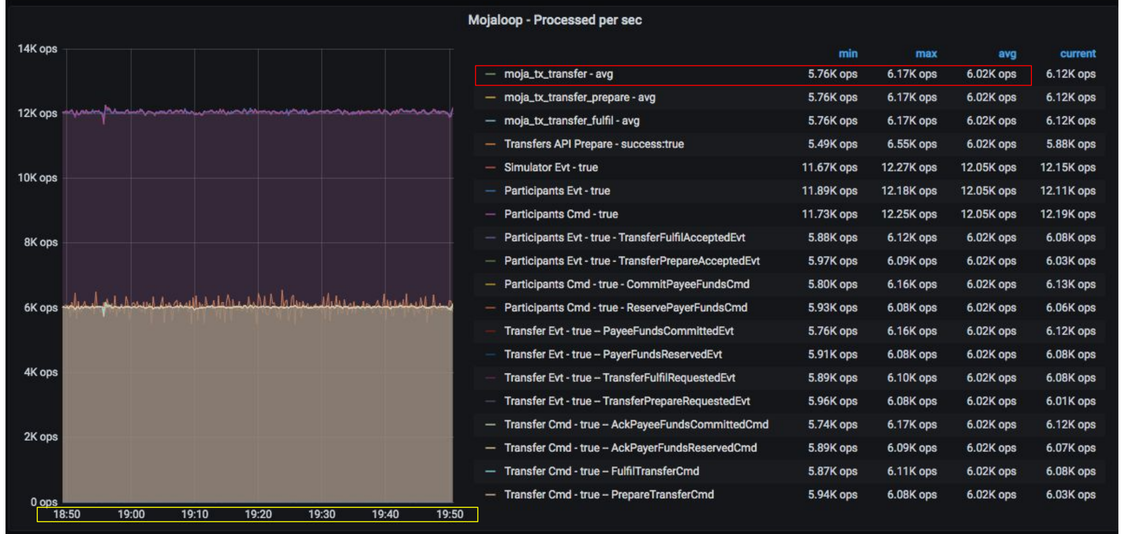
#	Indicators	Target	Actuals
T1	Financial Transactions Per Second (FTP/s)	>= 5000 FTP/s	6.02k FTP/s
T2	Time for performance run	>= 1 Hour sustained run	1 Hour
T3	% of transactions that took longer than a second	< 1%	0.48%

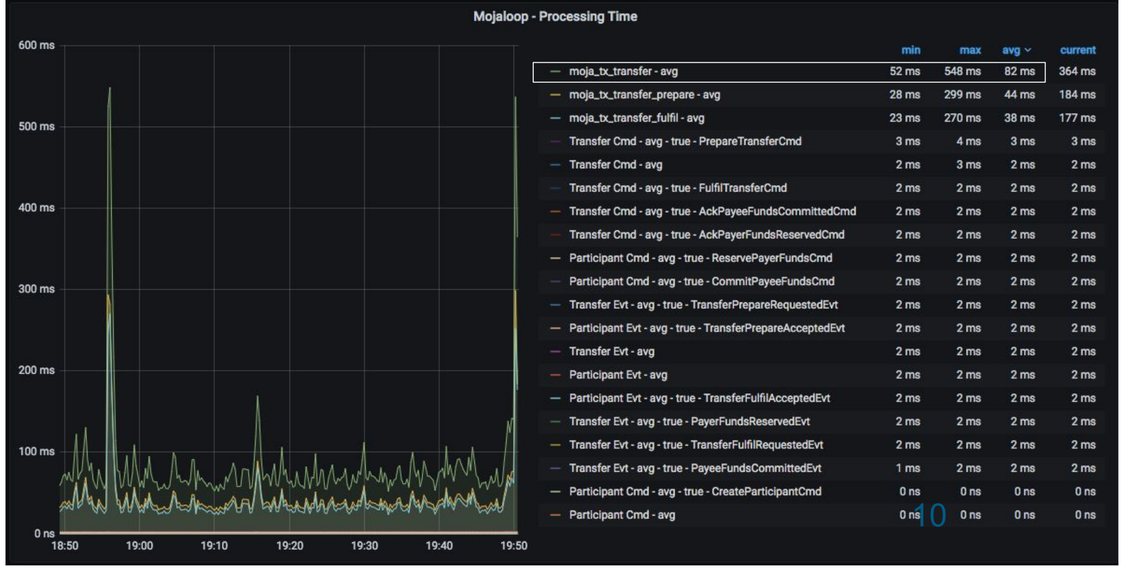
Component	Instance Scale	Infrastructure Type	Infrastructure Scale
Transfers Evt Handler	80	c5.2xlarge	28
Transfers Cmd Handler	80		
Participants Evt Handler	32		
Participants Cmd Handler	32		
ML-API-Adapter Service	24	c5.2xlarge	2

Component	Instance Scale	Infrastructure Type	Infrastructure Scale
Kafka	9	c5.9xlarge	9
Zookeeper	3		
Redis (Clustered - Self-hosted - Master only - No replication - Evenly Distributed Key space)	5 (cache) 1 (persistence)	i3.xlarge	6

Component	Instance Scale	Infrastructure Type	Infrastructure Scale
PoC Simulator - Payee	32	c5.2xlarge	2
PoC Simulator - Metrics Exporter	24	c5.2xlarge	2







mojaloop foundation

Notes:
* Results achieved without hardware optimization
** Fully Self-hosted (Redis, Kafka)

Scalability

We are close to linear.

Theoretical limit is the participants -> we can do ~*400 **OP/s per participant -> 200 ***FTP/s per Participant

Participant Cmd Handler Scale (1-to-1 to FSP)	Total OP/s (Aggregated Handlers)	Total FP/s (End-to-end Financial Transfer)
5 (CSU)	2000	**1000**
8	3200	**1,600**
16	6400	**3,200**
32	12,800	**6,400**
48	192,000	**9,600**
96	38,400	**19,200**
192	76,800	**36,400**

Table 4.2.1.a. Scaling Factor for ParticipantCmdHandler

Notes:
* Based on c5.2xlarge
** OP/s - Operations per Second
*** FP/s - Financial Transactions per Second

To achieve even more performance per participant:

1. Kafka Message Encoding-Parsing ← Reduce encoding/parsing overhead
2. Separation of Participant Metadata, Account and End-point Information ← Optimize data persistence, & message sizes
3. Virtual positions (sharding the position through different participant handlers) ← Partition FSP position for Parallelism
4. Bloom Filters for Duplicate Checks ← Reduce space requirements for duplicates
5. Batching by FSP (workloads are already in an appropriate FSP bucket - i.e. partitions) ← Optimize processing throughput
6. Provision hardware with faster CPUs (scaling up to achieve a higher throughput) ← Increase processing muscle
7. Tiger Beetle ← Optimize data store

What we achieved against objectives

1. Min Deploy: 1,000 FTP/s, 99% < 1 sec, 1 hour durability, on minimum hardware footprint (required minimum HW footprint to be defined by the PoC) - **Achieved: 8x FSPs for 1x CSU fully self-hosted (Kafka & Redis)**

2. Sub-linear cost to scale for each single unit increment of capacity - i.e. **cost scale unit (CSU)**. The hardware cost of one increment is significantly less than the minimum deployment footprint. The architecture isolates the high-load components to scale separately from the low load ones. - **Achieved: Near-linear performance & cost scalability**

3. Demonstrated Scaled Perf: nearly 5,000 FTP/s, 99% < 1 sec, 1 hour durability for 5 CSU, plus min deploy unit. Proving point (2) - **Achieved: 6020 FTP/s - 6x Performance @ ~8 CSU using self-hosted**

4. Understand how much the PoC design scales, max scaling before 99% < 1 sec fails or error rate increases (optional) - **This is a moving target, we keep getting more: hitting 6k FTP/s**

PI-11 Learnings

1. Limit Event-Sourcing Usage to Participants only
2. Command-Side Caching of Transfers
3. GZIP Compression
4. Redis-Cluster (Sharding) to overcome Network & IOPs limitation

Cost Scale Unit (CSU) - AWS Cost

Central Ledger
- Participants + Transfers - 7x c5.xlarge
- ML-API-Adapter - 1x c5.2xlarge

Infrastructure
- Kafka cluster - 5x r5.xlarge
- Redis - 3x i3.xlarge

Others
- Simulators - 1x c5.2xlarge

7	c5.xlarge
5	r5.xlarge
3	i3.xlarge
2	c5.2xlarge
17	**Total**

2,323 USD / month

8 DFSPs handling 1000 financial transactions per sec

Ireland - 1yr reservation / no upfront - https://calculator.aws/#/estimate?id=279063344e3d2e6372a00c36d469f91f11d4e303

mojaloop foundation

Conclusion

1. Better horizontal scaling by segregating the different functions across different services and datastores
2. Separate transfers and participants lead to easier partition of datastores
3. Better maintainability / extensibility
4. This design shifts the burden from a shared DB to distributed streaming (kafka), separate (mongodb) read stores and (redis) caches - which are horizontally scalable by design
5. Reduced latency and faster response times
6. Retrofitting is simple and a migration/bootstrapping of data is feasible

Performance Scalability Report

Table of Contents - v0.1-draft - pending community review

Team:

Crosslake
- Pedro Sousa Barreto <pedrob@crosslaketech.com>

ModusBox
- Miguel de Barros <miguel.debarros@modusbox.com>
- Roman Pietrzak <roman.pietrzak@modusbox.com>

Coil
- Donovan Changfoot <donovan.changfoot@coil.com>

Next Steps

1. Community & Design-Authority to provide comments to the report document
2. Design-Authority to provide recommendation
3. Final decision:

    1. Productionize PoC
        - Roadmap, Delivery & Resource Plan for Productionization
        - Production-grade Designs
        - Some productionization work (not dependant on Final Production Designs)

    2. Integrate learnings into existing code-base
        - Select Key Learnings to implement based on available time & resources

How can you help?

We need help reviewing the

“Performance Scalability PoC Report”

Team work

Amar Ramachandran *(Modusbox)*

Donovan Changfoot *(Coil)*

Miguel de Barros *(Modusbox)*

Pedro Barreto *(Crosslake)*

Roman Pietrzak *(Modusbox)*

Sam Kummary *(Modusbox)*

mojaloop
foundation

Q&A

Code at
https://github.com/mojaloop/poc-architecture

mojaloop
foundation