CONTINO

Mainframe & Serverless Integration

How to Liberate the Data and Stay Competitive in the New World!

Federico Fregosi - VP of Engineering

CONTINO

Mainframe & Serverless Integration

Come liberare i dati e rimanere competitivi nel nuovo mondo!

Federico Fregosi - VP of Engineering

#WhoAmI

Federico Fregosi

Current position: **VP of Engineering, EMEA**

Past:

federico.fregosi@gmail.com

https://www.linkedin.com/in/federico-fregosi/

Large Payments Providers Are at a Crossroads

From the 90s, the payments industry made billions ($$$!) off the back of first-class reliability and resilience.

But modern tech is pushing the source of value beyond just reliability and resilience.

But there is a solution! Payment providers are sitting on a **DATA GOLD MINE i**n their mainframe.

This data contains insights that will power their product and customer strategies for years to come.

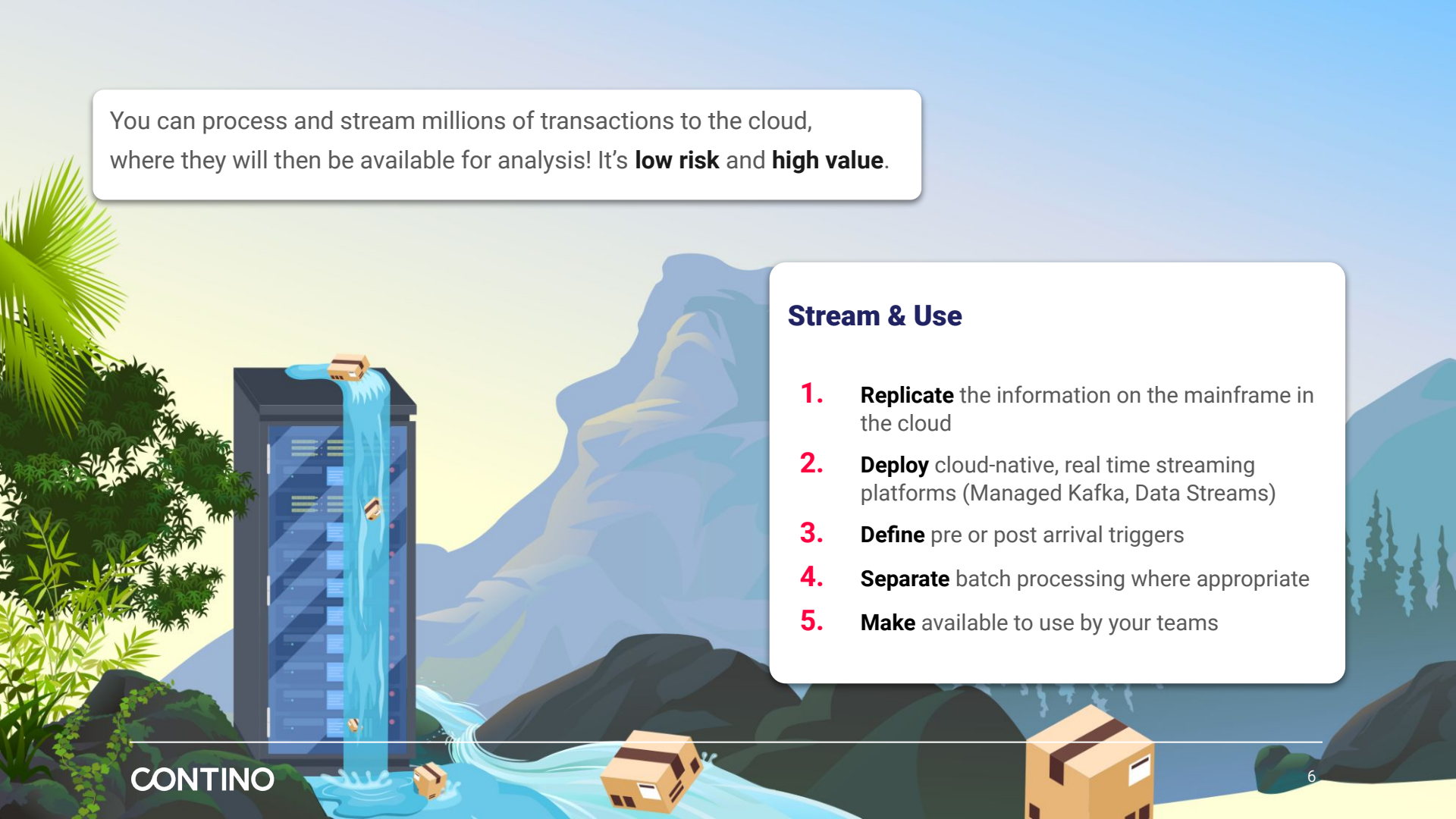
Getting to that data isn't easy.

Payment Providers Need to Evolve!

You need to shift your business from delivering resilient, reliable transactions....

...to providing real-time integrated data-driven services.

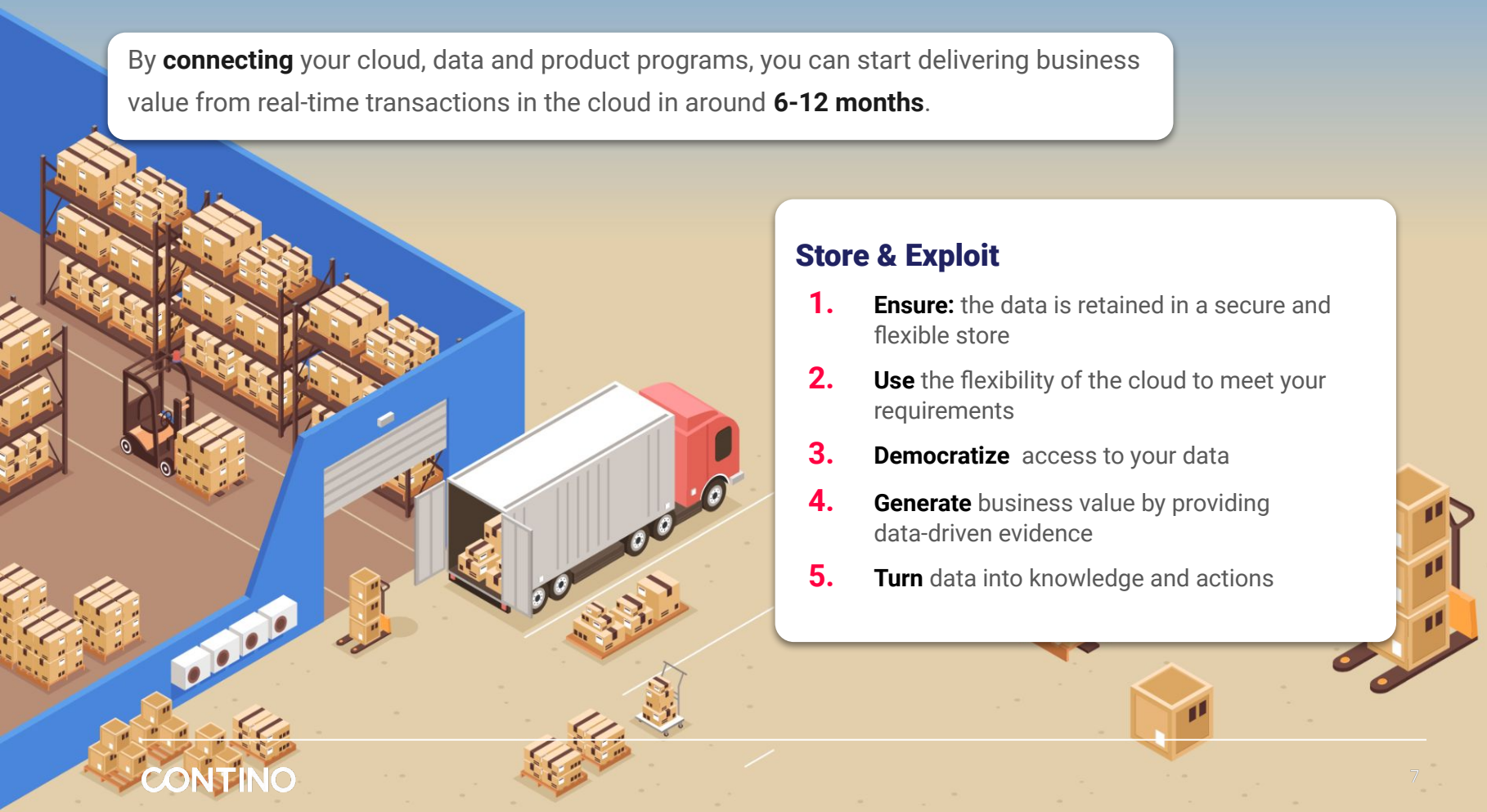
You can process and stream millions of transactions to the cloud, where they will then be available for analysis! It's **low risk** and **high value**.



Stream & Use

1. **Replicate** the information on the mainframe in the cloud
2. **Deploy** cloud-native, real time streaming platforms (Managed Kafka, Data Streams)
3. **Define** pre or post arrival triggers
4. **Separate** batch processing where appropriate
5. **Make** available to use by your teams

By **connecting** your cloud, data and product programs, you can start delivering business value from real-time transactions in the cloud in around **6-12 months**.

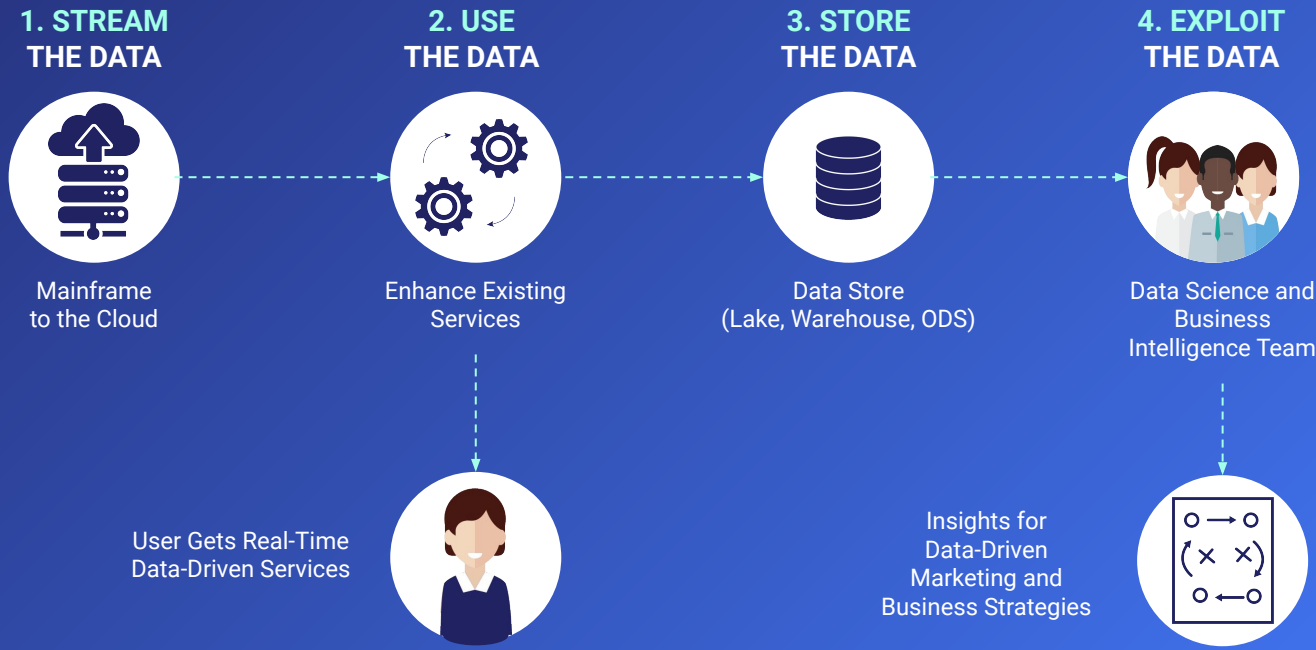


Store & Exploit

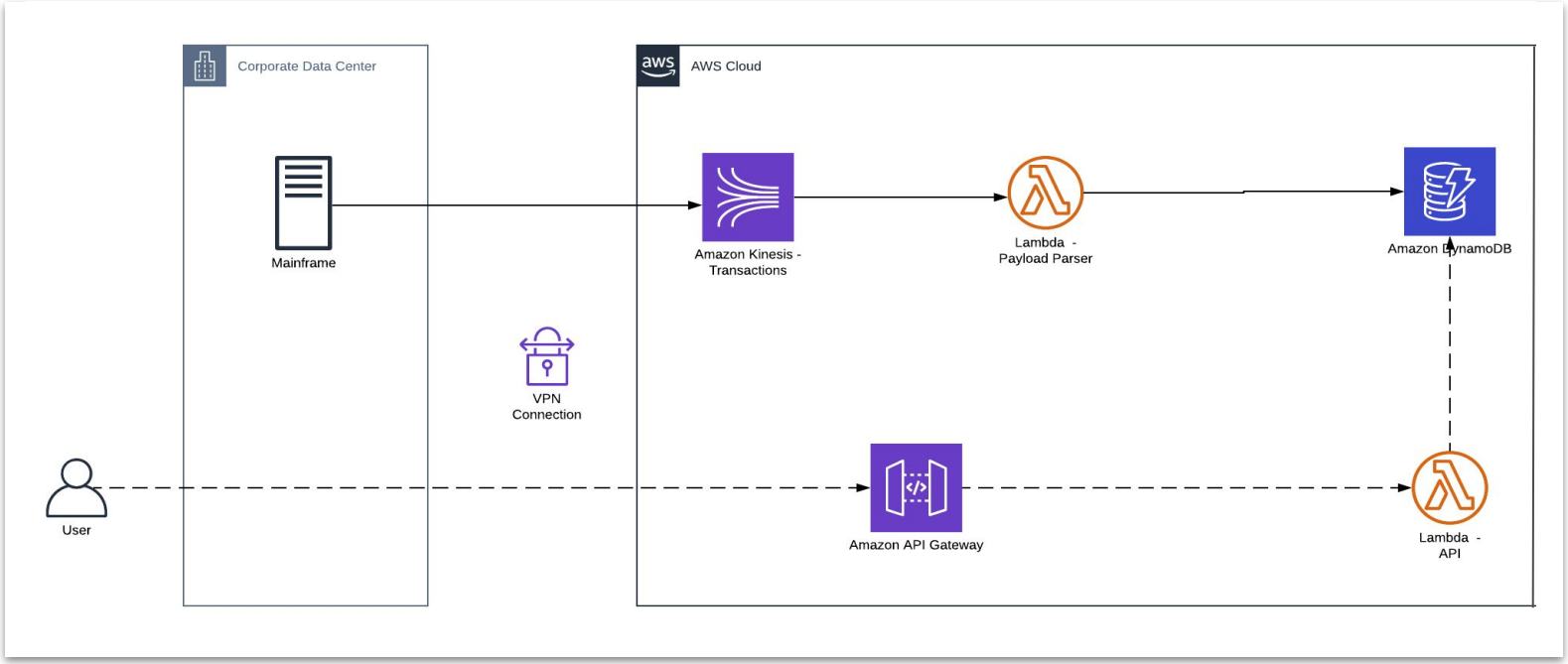
1. **Ensure:** the data is retained in a secure and flexible store
2. **Use** the flexibility of the cloud to meet your requirements
3. **Democratize** access to your data
4. **Generate** business value by providing data-driven evidence
5. **Turn** data into knowledge and actions

The Architecture

Live-Streaming Mainframe Data to the Cloud



The Architecture (Extremely Simplified)



Quick Primer on Some Core AWS Services


AWS Lambda


Amazon API Gateway


Amazon Kinesis Data Firehose


Amazon Simple Notification Service


AWS Step Functions


AWS Snowball


AWS Fargate


Amazon Kinesis


Amazon Elasticsearch Service

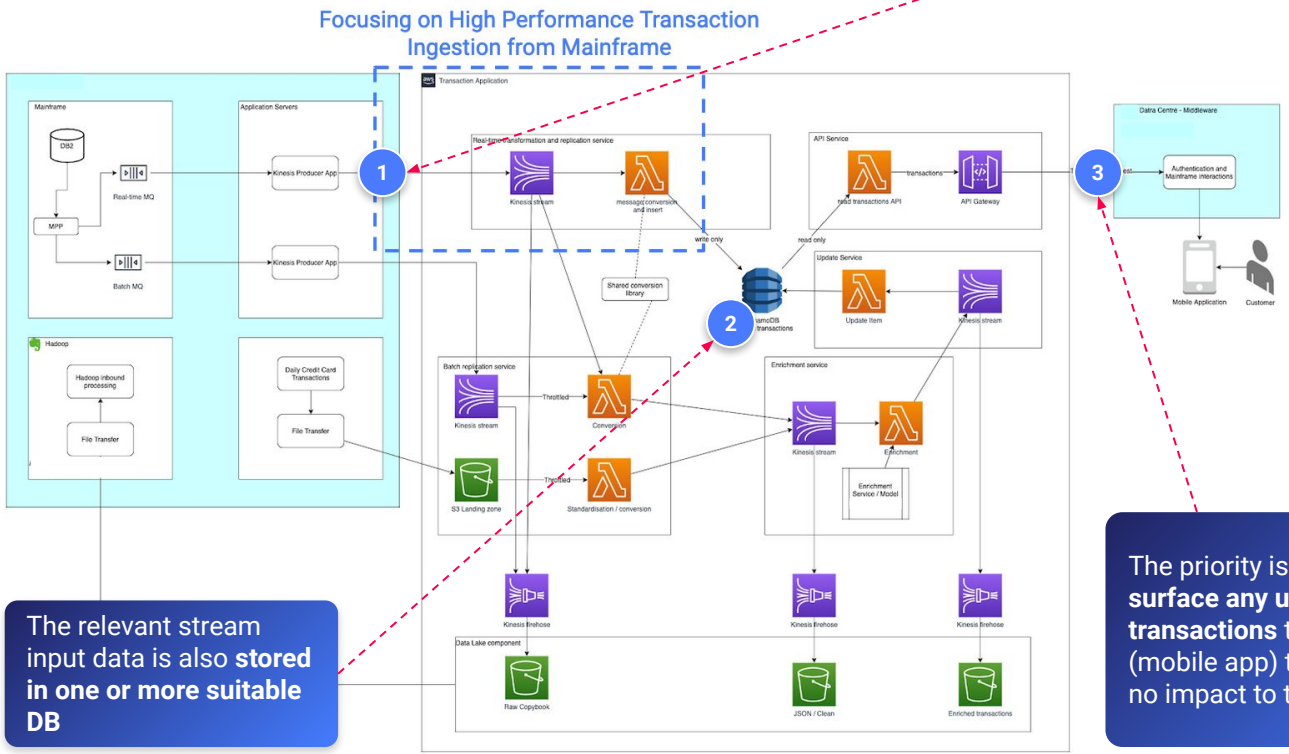

Amazon Simple Queue Service


Amazon RDS


Amazon DynamoDB

The Architecture (Not as Simple)

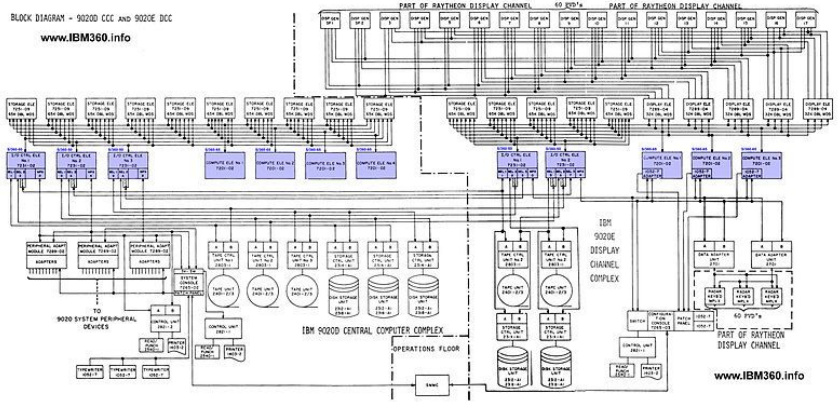
The focus is on providing **performance and accuracy** of the transactions stream initiated by a mainframe process



The relevant stream input data is also **stored in one or more suitable DB**

The priority is to **rapidly surface any updated transactions** to the users (mobile app) to prove there is no impact to the UX

Architecture at a Glance



- Serverless/Microservices architecture
- Lambda 80 %, Fargate 20%
- 3 Teams, multiple AWS accounts
- 9 Microservices, tens of functions
- Shared Kernel (lightly versioned / library)
- 120 rps API, latency <100 ms 95th percentile
- Event sourcing pattern with Data lake integration
- CI/CD with AWS services & Serverless Framework


The Actual Architecture... here...

Microservices Boundaries

- First - Domain Model.
  Start identifying the aggregates

- Second - Operational needs.
  Who's going to run this?
  It had already been decided!

  (*Inverse Conway's Manoeuver!*)


Architectural
Highlights

Ensuring Correctness

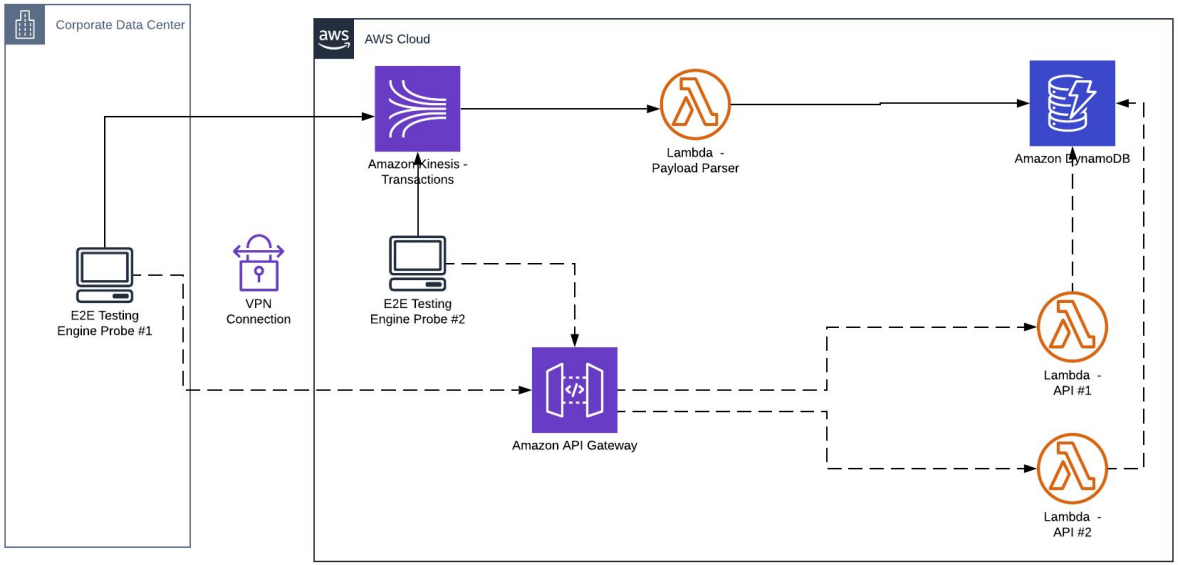
It's a distributed systems problem.This is financial transaction data; end users need correct values.

- Acceptable levels of service as SLOs (accuracy, speed, availability)
- Live End-To-End Testing System
- In-band and Out-of-Band Transaction Reconciliation System (TRS)

Acceptable Levels of Service (NFRs)

1. Define expected **correctness**: “99% of Get API calls for transactions, over a minute, will return correct data at a “certain” moment in time”
2. Define expected **consistency**: “99.99% of Get API calls for transactions, over a minute, will return correct data not-older-than 30 secs”
3. Define expected **latency**: “99% of Get API calls for transactions,over a minute, will return the most up-to-date data in less than 100ms”

End-To-End Testing System



- Adopts end-user perspective
- Verifies formal correctness
- Multiple Probes from different vantage points

- Continuous Testing
- Custom developed system (*7*)
- Failures trigger notifications
- Separate performance testing (*buy*)

Two Transaction Reconciliation Systems

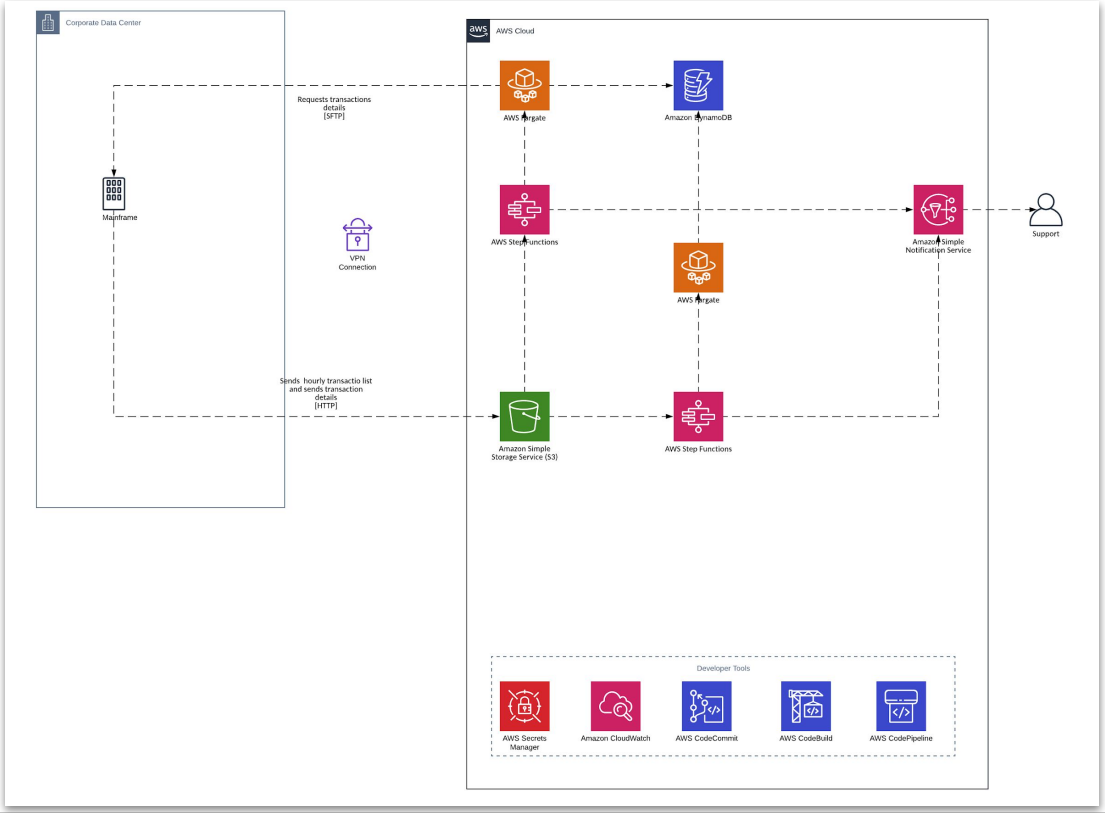
In-Band

- Uses streaming transactional data
- Idempotency tags
- Retries on AWS side
- Repeats on the mainframe
- Monitoring and alerting enabled
- With High-Water mark

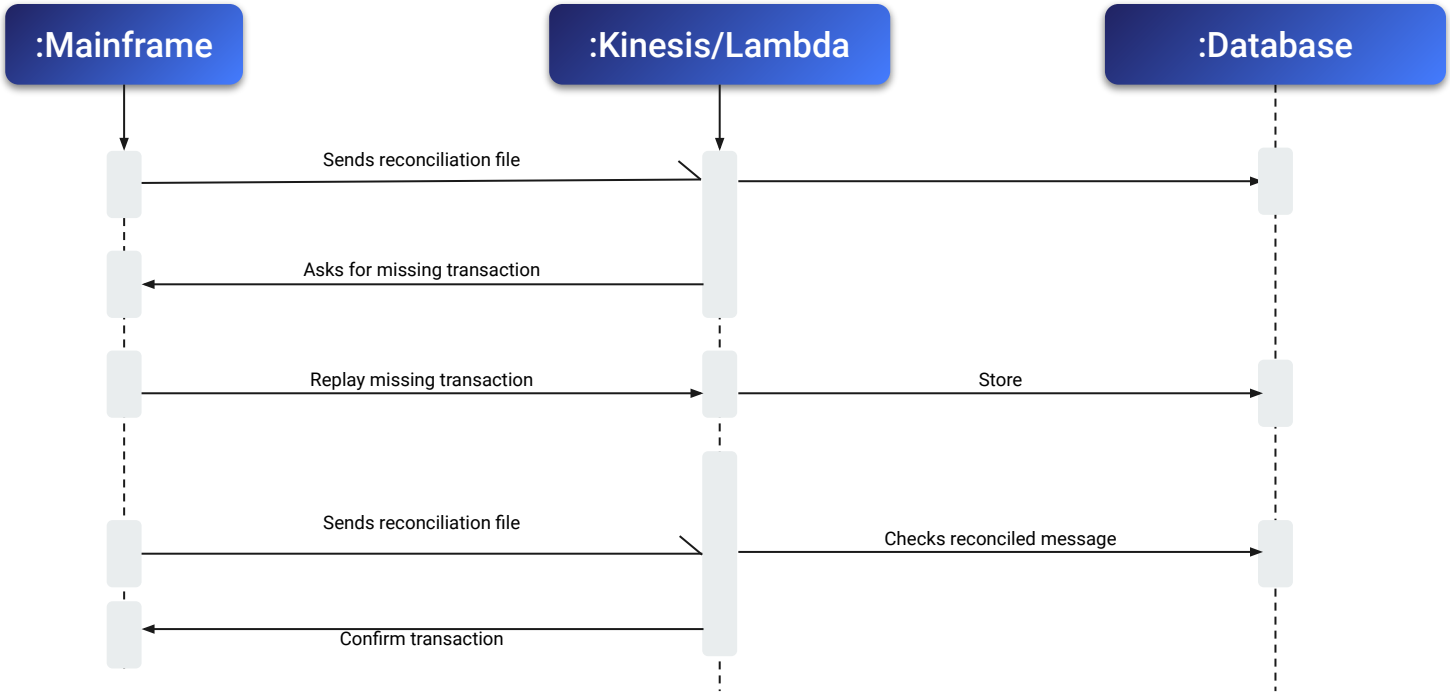
Out-Of-Band

- Complex system with Fargate and AWS Step Functions
- Uses a different datasource (batch) and processes transactions in a different way
- With automated and manual resolution
- Monitoring and alerting enabled
- With High-Water mark

Transaction Reconciliation System

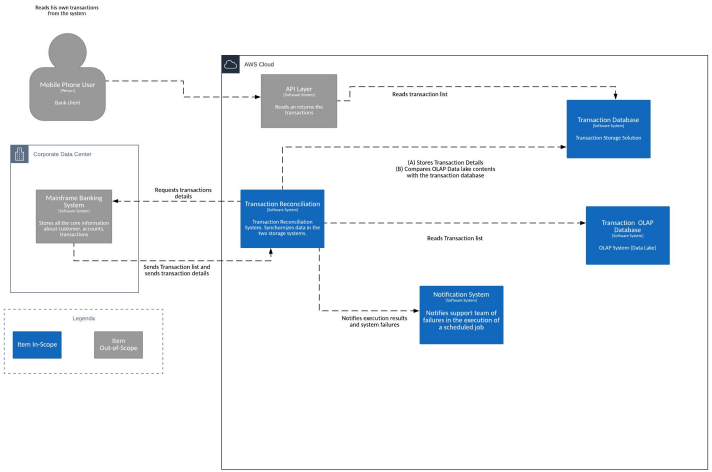


Transaction Reconciliation System

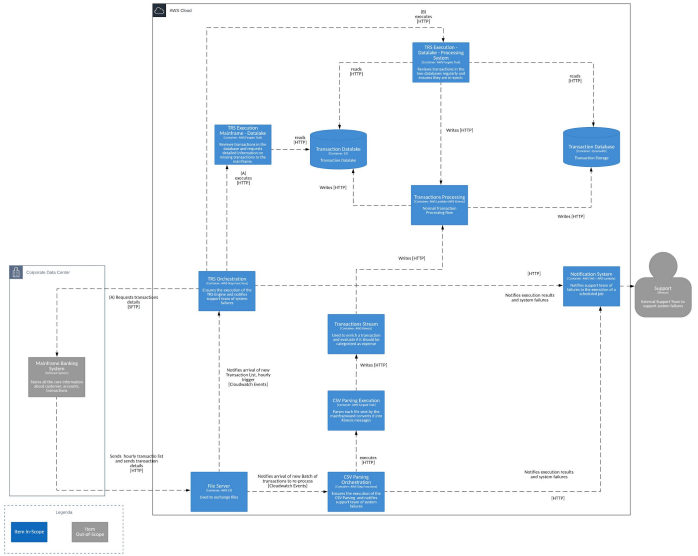


UML Sequence Diagrams

Transaction Reconciliation System - C4 Model

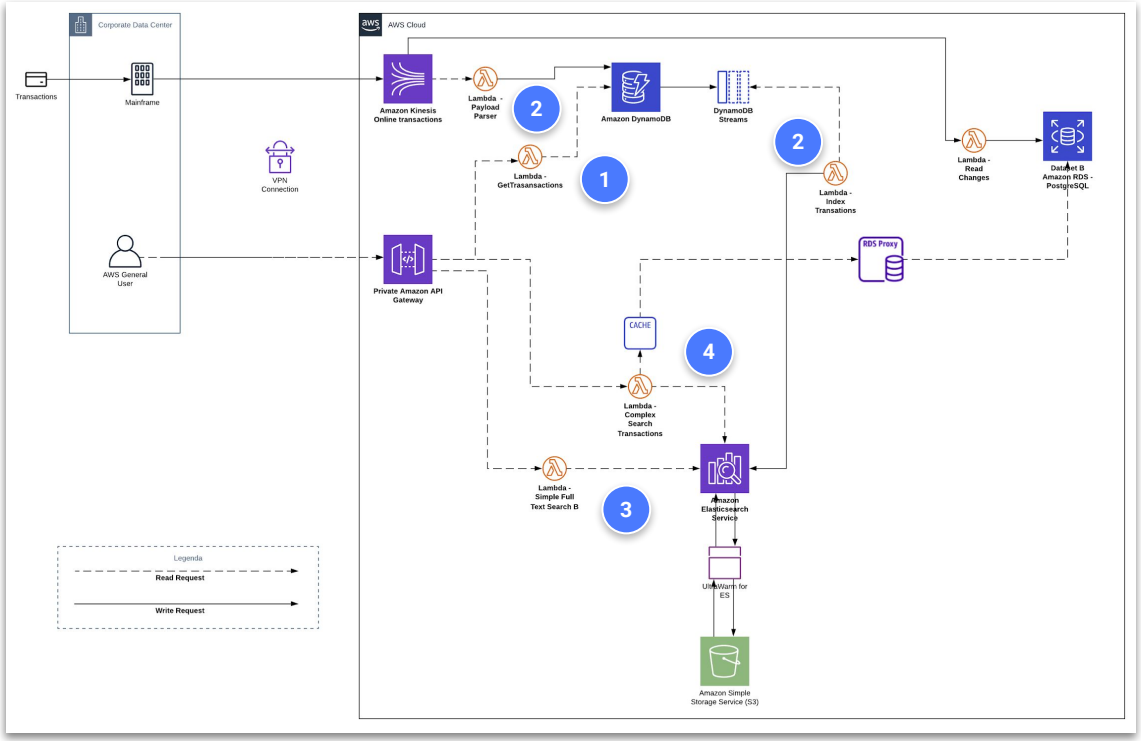


System Context



Container Model

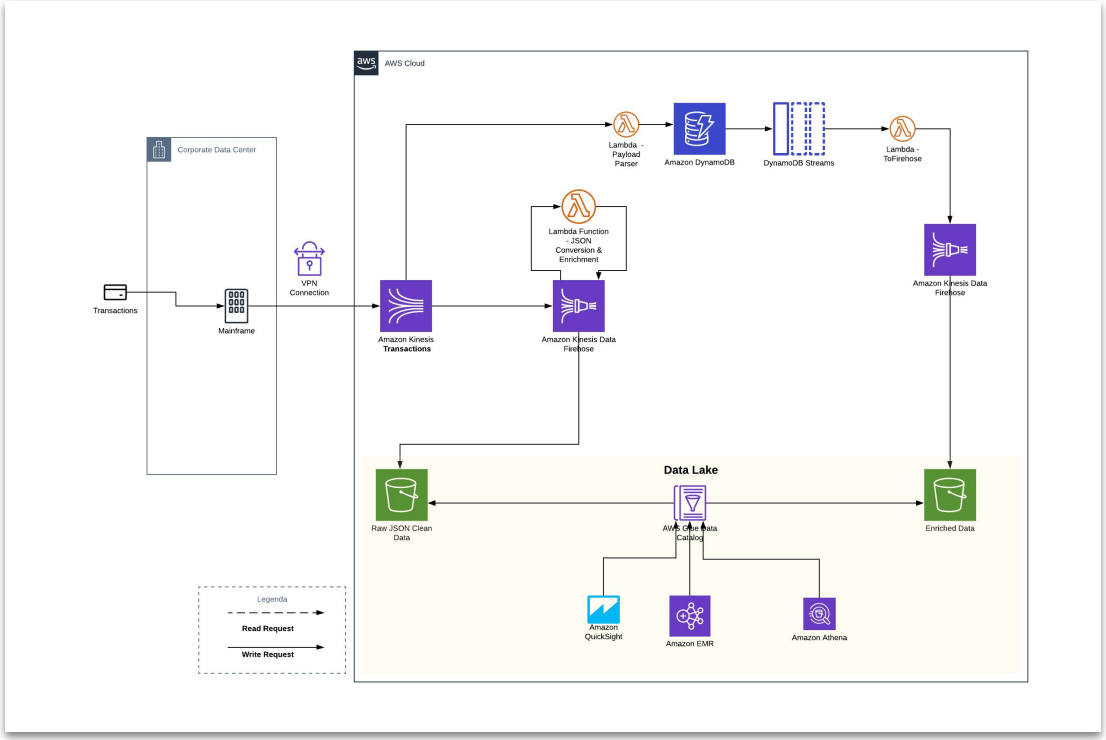
Full-Text and Advanced Search



- Enables users to search their transactions
- Multi-Database : DynamoDB, RDS, Amazon ElasticSearch
- Uses DynamoDB Streams to replicate data to ES
- Ultrawarm for low-frequency data
- With RDS Proxy and Simple Cache
- Explicitly disregards API Gateway Velocity Templates

NOTE: Simplified diagram

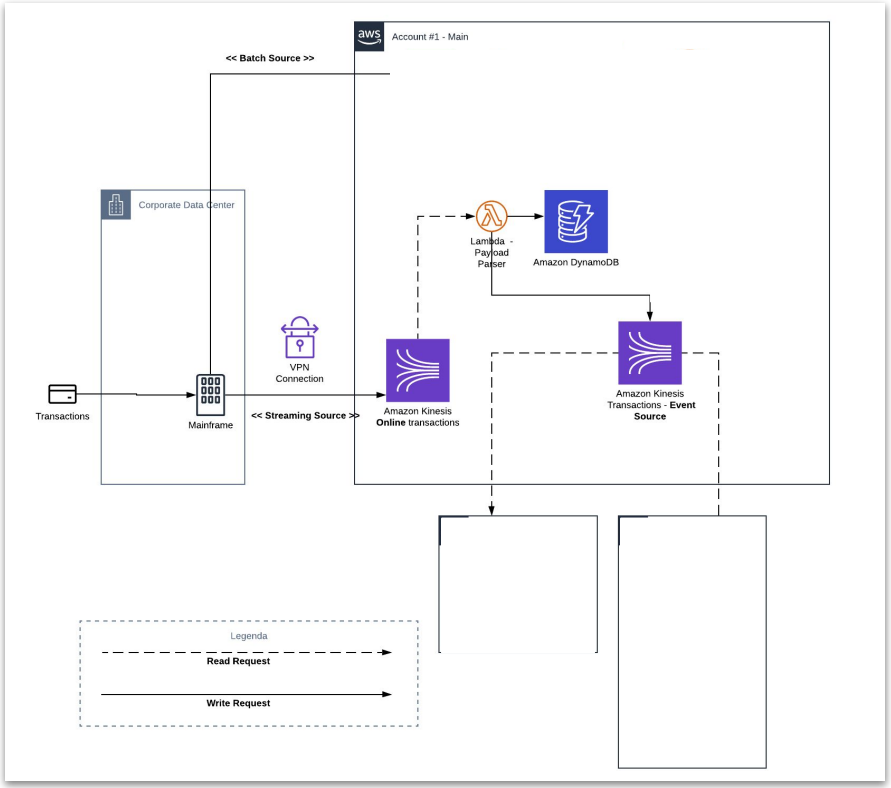
Data Lake Integration



- Where value lies: liberating the data for wider consumption
- Easy “in team” data exploration
- Used for analytics
- Standardized out-of-team consumption
- (data stewardship and governance external)
- Raw and Enriched (clean) data available

NOTE: Simplified diagram

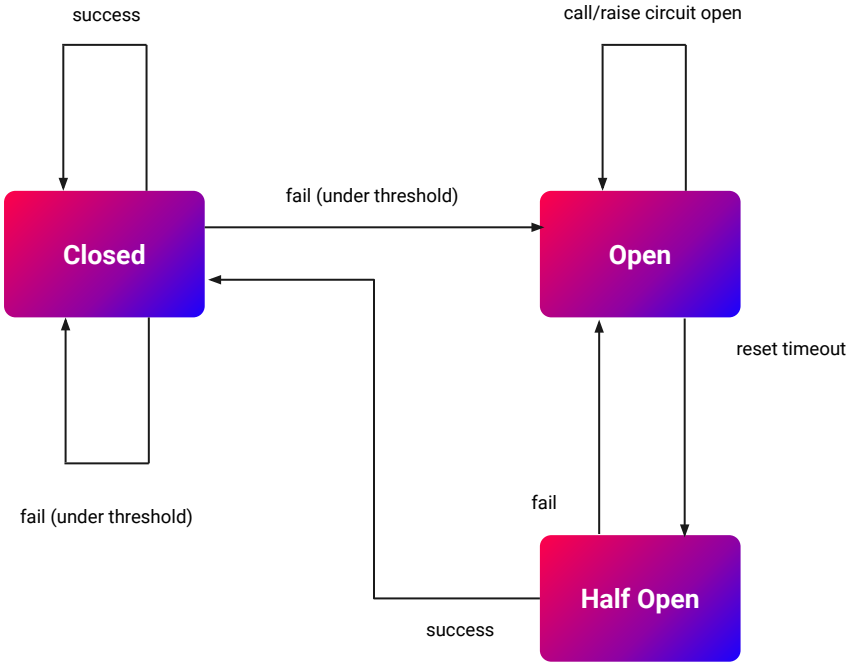
Event Source



- Kinesis-based event source
- Follows financial transaction lifecycle
- Strong decoupling
- Useful to other live systems
- With idempotency tag
- Does not allow to replay history
- Contract testing with Pact

Serverless Patterns

Circuit Breaker Pattern

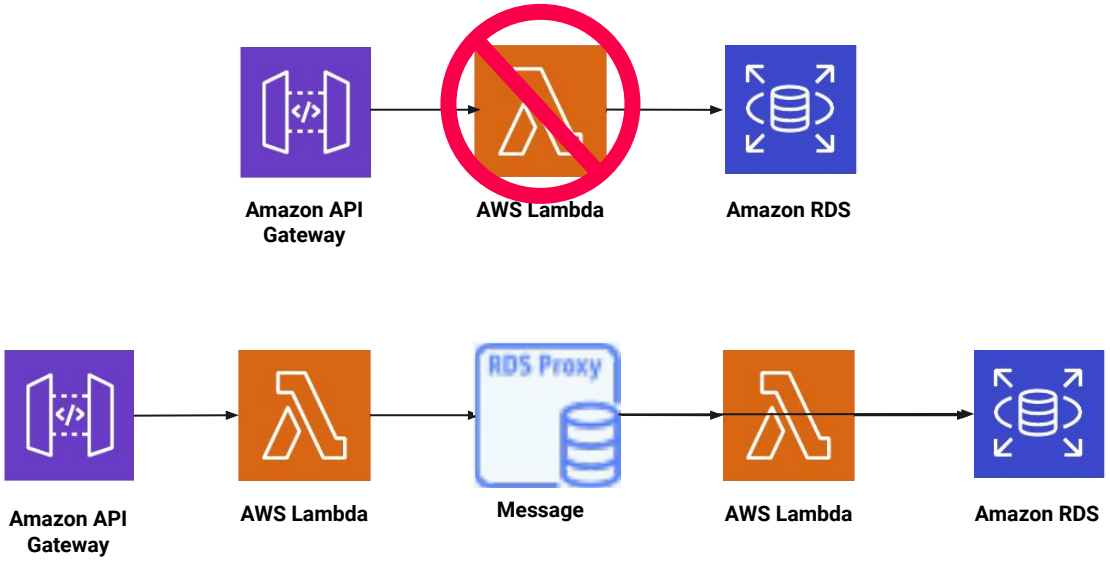


- Strong requirements for transaction consistency and availability
- Automated switch-off on high number of errors
- Mainframe system as fallback
- Prevents downstream service flooding
- Reduces resource consumption

Controlling Function Scaling and Throttling

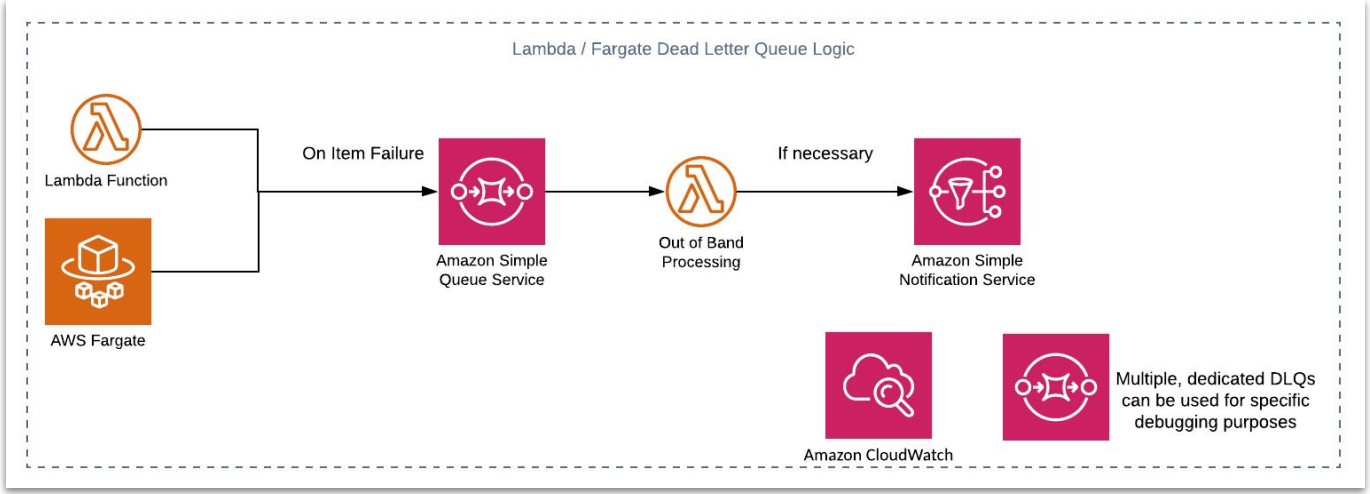
- Make sure to monitor account quotas
- Consider Provisioned Concurrency
- Set-up extensive monitoring in Cloudwatch (*Throttles*, *Invocations*, *ExecutionTime*)
- Look for tail-latency events

Impact on Downstream Systems



Set Reserved Concurrent Executions

Dead-Letter-Queue Pattern on Lambda

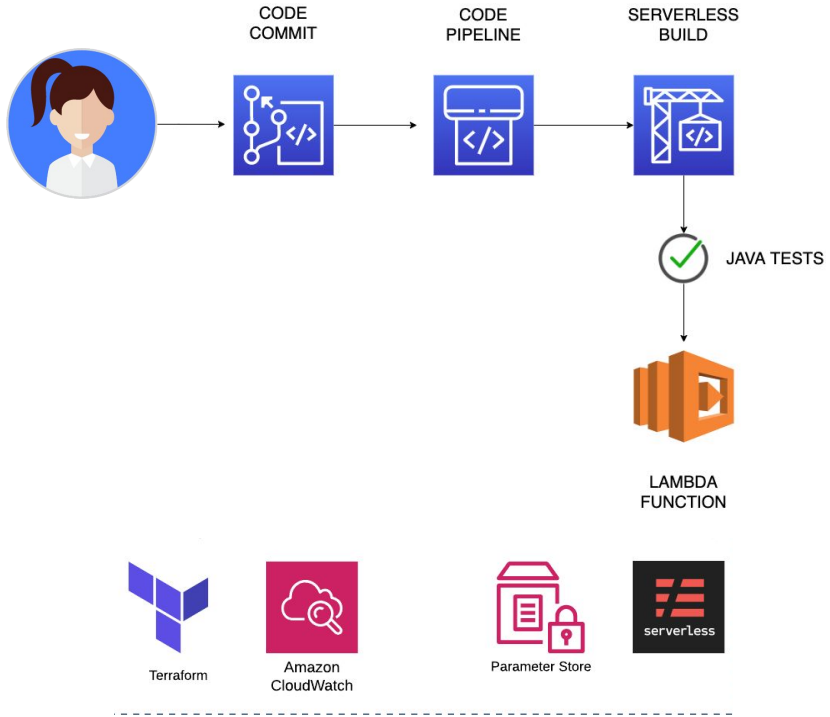


- Kinesis specific logic: Bisect-on-failure and multiple retries
- Aware of Lambda context
- Can have performance impact on the overall system
- Failures trigger notifications
- Lambda Destinations unsuitable

Developer Experience

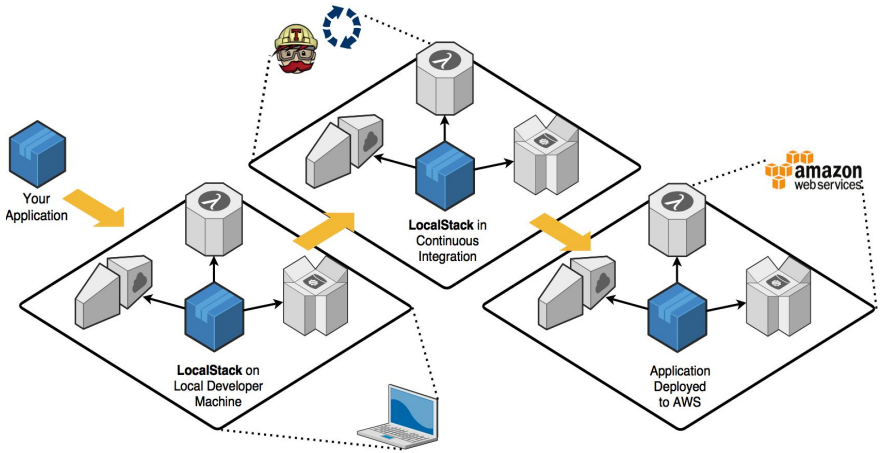
CI/CD

- Using AWS Developer Tools
- Monorepo for functions, continuous deployment multi-branch pipeline
- Backed by build automation tools and integrated with existing on-prem solutions
- **Serverless Framework** core tool for Lambda Functions
- Terraform for infra resources
- Unit & Integrations Tests running in the pipeline with artefact promotion
- Canary Deployment with gradual traffic rollout
- Supports manual approvals for compliance, where needed



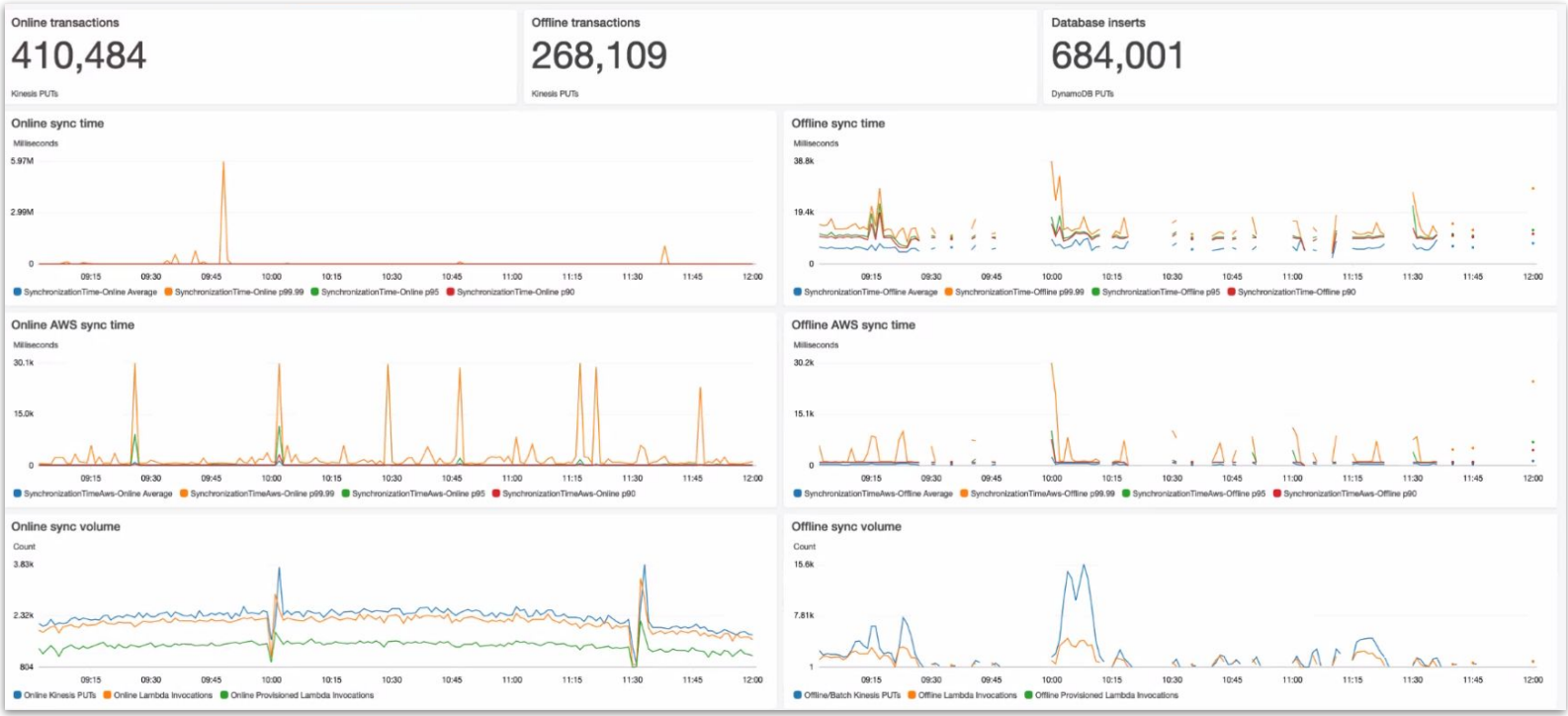
Testing & Debugging Serverless

- Used Localstack,
- Integrated with IntelliJ using AWS toolkit to test/debug Lambda's
- Make ample use of AWS X-Ray & AWS Cloudwatch
- Testing in production with feature flags and Canaries (Serverless framework)

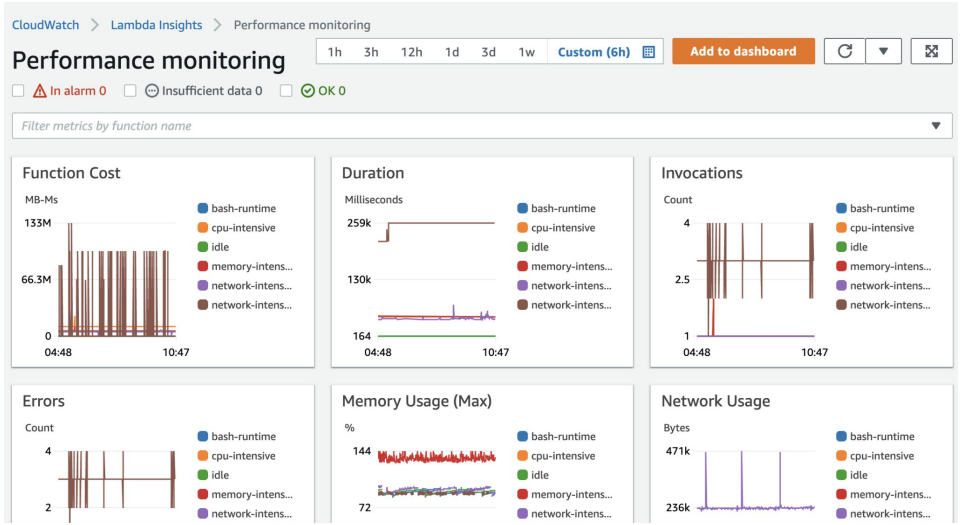


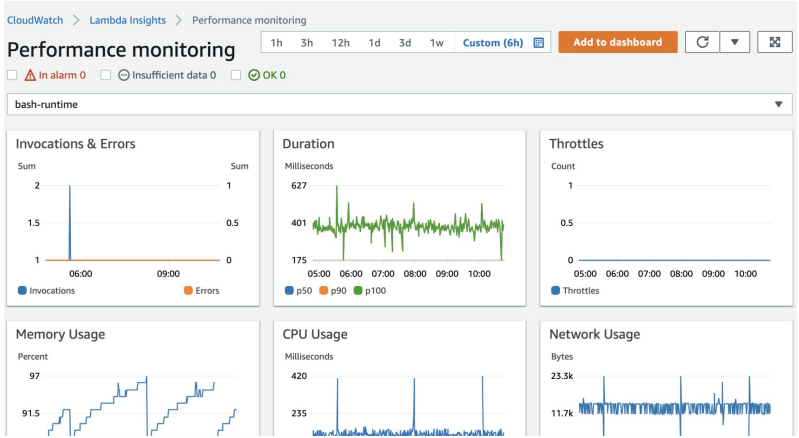
Performances

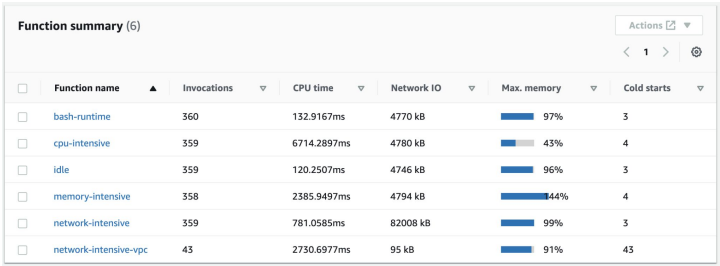
Results


Online transactions
410,484
Kinesis PUTs
Offline transactions
268,109
Kinesis PUTs
Database inserts
684,001
DynamoDB PUTs
Online sync time
Milliseconds
5.97M
2.99M
0
09:15 09:30 09:45 10:00 10:15 10:30 10:45 11:00 11:15 11:30 11:45 12:00
SynchronizationTime-Online Average
SynchronizationTime-Online p99.99
SynchronizationTime-Online p95
SynchronizationTime-Online p90
Offline sync time
Milliseconds
38.8k
19.4k
0
09:15 09:30 09:45 10:00 10:15 10:30 10:45 11:00 11:15 11:30 11:45 12:00
SynchronizationTime-Offline Average
SynchronizationTime-Offline p99.99
SynchronizationTime-Offline p95
SynchronizationTime-Offline p90
Online AWS sync time
Milliseconds
30.1k
15.0k
0
09:15 09:30 09:45 10:00 10:15 10:30 10:45 11:00 11:15 11:30 11:45 12:00
SynchronizationTimeAws-Online Average
SynchronizationTimeAws-Online p99.99
SynchronizationTimeAws-Online p95
SynchronizationTimeAws-Online p90
Offline AWS sync time
Milliseconds
30.2k
15.1k
0
09:15 09:30 09:45 10:00 10:15 10:30 10:45 11:00 11:15 11:30 11:45 12:00
SynchronizationTimeAws-Offline Average
SynchronizationTimeAws-Offline p99.99
SynchronizationTimeAws-Offline p95
SynchronizationTimeAws-Offline p90
Online sync volume
Count
3.83k
2.32k
804
09:15 09:30 09:45 10:00 10:15 10:30 10:45 11:00 11:15 11:30 11:45 12:00
Online Kinesis PUTs
Online Lambda Invocations
Online Provisioned Lambda Invocations
Offline sync volume
Count
15.6k
7.81k
1
09:15 09:30 09:45 10:00 10:15 10:30 10:45 11:00 11:15 11:30 11:45 12:00
Offline/Batch Kinesis PUTs
Offline Lambda Invocations
Offline Provisioned Lambda Invocations

From numbers to percentages



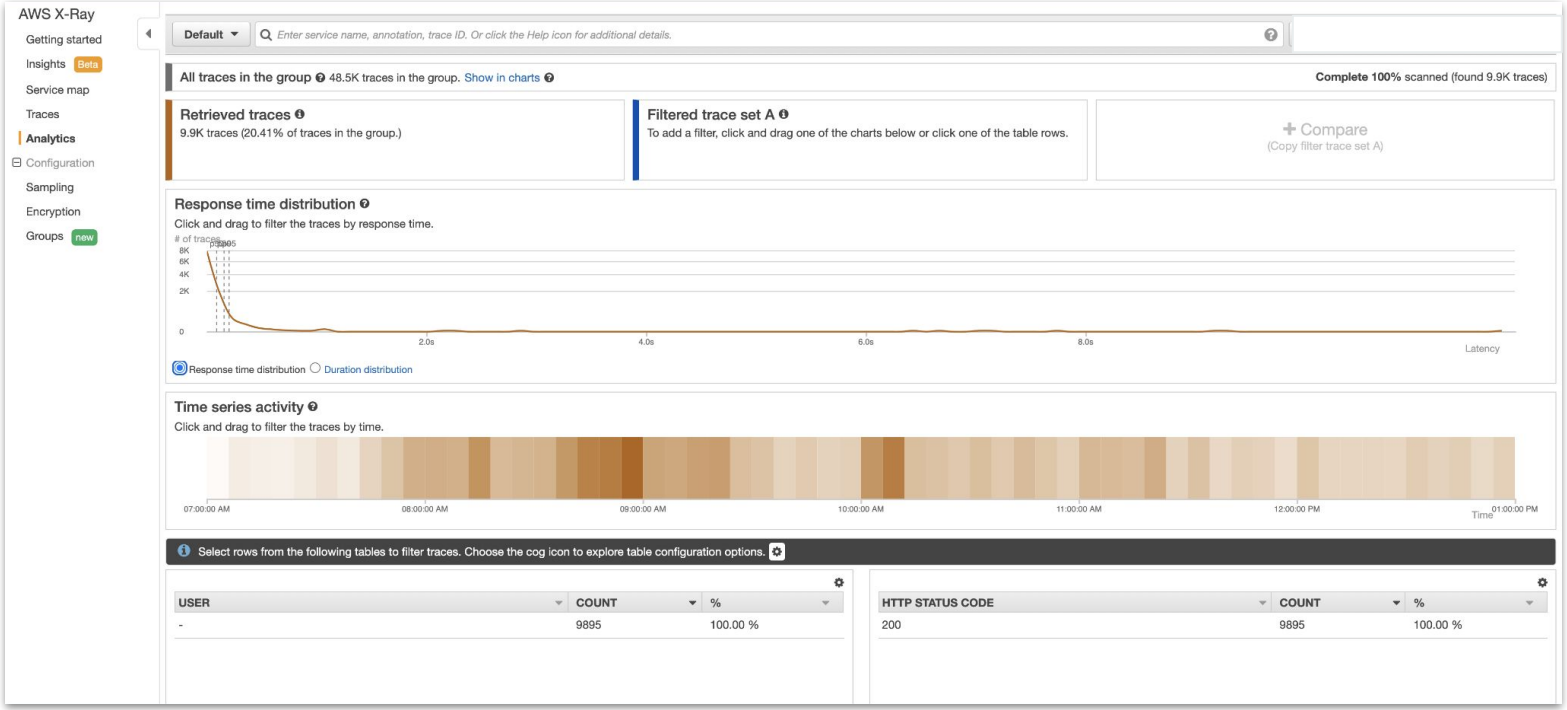




Measuring for real

- Distributed systems hard to tackle issue
- Lambda's NTP servers cannot be configured
- Lambda **should run on** Stratum 3 NTP servers
- Run a small test using Lambda & Python
- With DC<->AWS RTT at 40 ms, clock offset < 10ms
- No guarantees are made

Response Time



Response time for the system p95 < 100ms , p50 ~60 ms

Example Java Optimizations on 

Cold-Start

- **Optimize** libraries import
- Use **Provisioned Concurrency** (> cost, >complexity, >deployment time)
- X-Ray **adds** non-negligible **latency** (Cloudwatch Embedded Metrics)
- **Reduce** artifact Size
- Monitor the **p99**

Execution

- Tune to the **right** memory allocation
- Move as much as possible in the init phase (DynamoDB or 3rd party client)
- Fetch secrets at **init** time and handle failures
- Watch your **framework**
- Monitor the **average** and tail latency

Performance samples...

Below are the results from our load testing activity, along with the adjustments we made in order to keep response times as low as possible.

33ms
avg
39.18ms
p95
387ms
max

Mainframe to Kinesis

- Multiple Kinesis Shards
- Enhanced fan-out/stream consumer enabled
- On-demand DynamoDB

28.59ms
avg
37.2ms
p95
2s
max

User to API Gateway

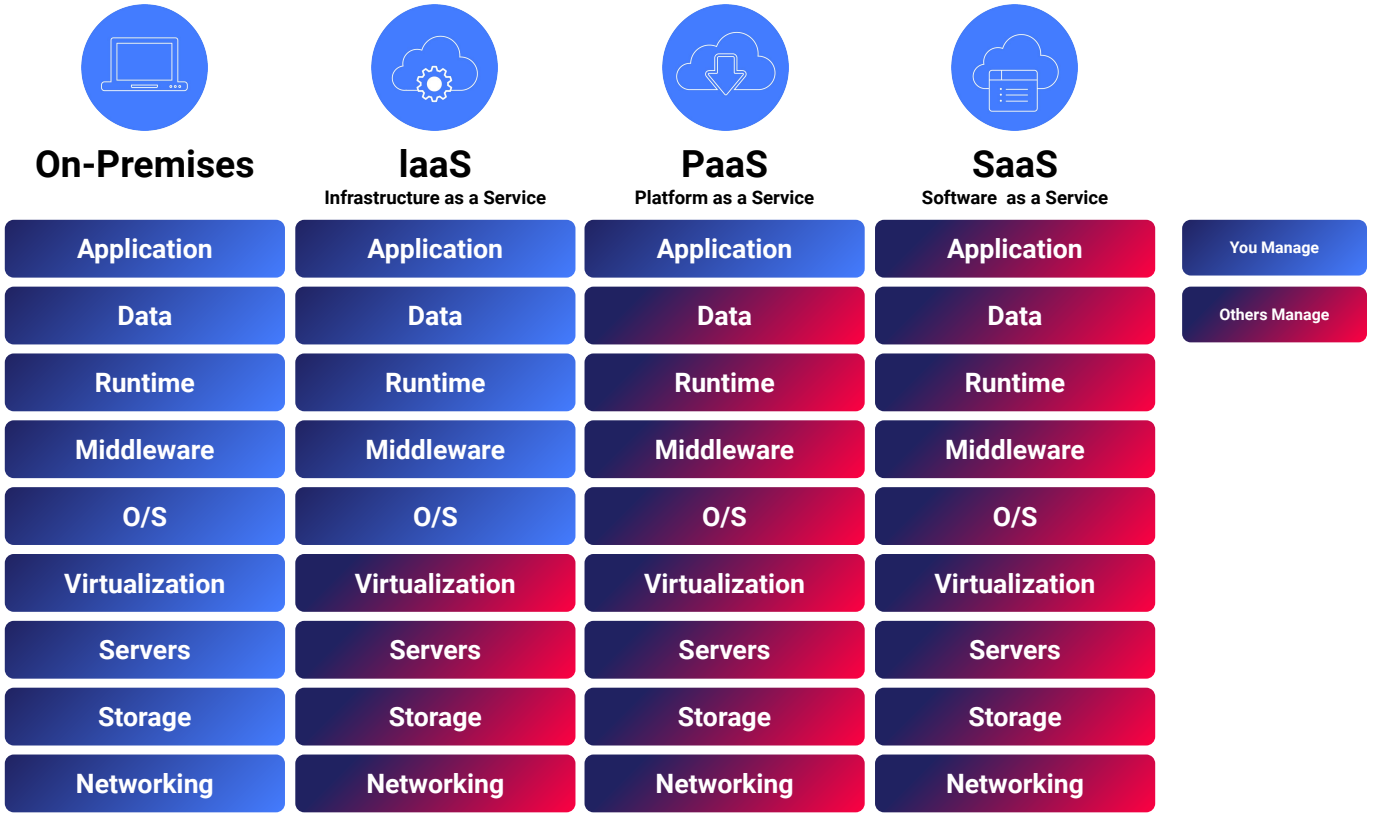
- Lambda with Provisioned Concurrency
- On-demand DynamoDB

Takeaway System Optimizations

1. Java **can** be optimized for low latency application on Lambda
2. **Enhanced Fan-Out** increases throughput and **reduces latency**
3. Use **Lambda Destinations** (Bisect on Function Error)
4. **Provisioned Concurrency** is a bit expensive but significantly helps controlling the cold-start problem
5. Kinesis batch size tuning **had more impact on avg latency** than JVM optimizations

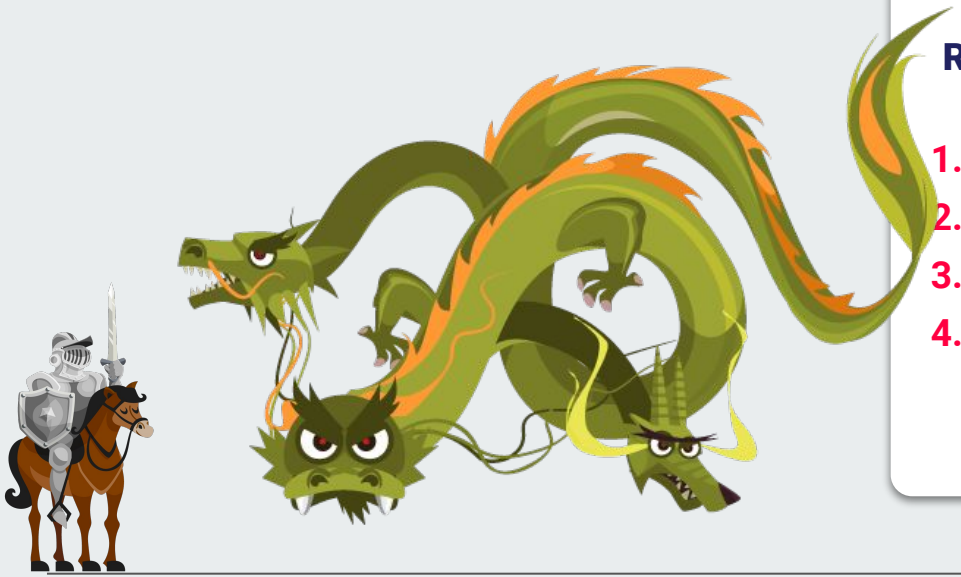
Cloud Operating Model

Serverless


On-Premises
IaaS
Infrastructure as a Service
PaaS
Platform as a Service
SaaS
Software as a Service
Application
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
You Manage
Others Manage

Fully Serverless

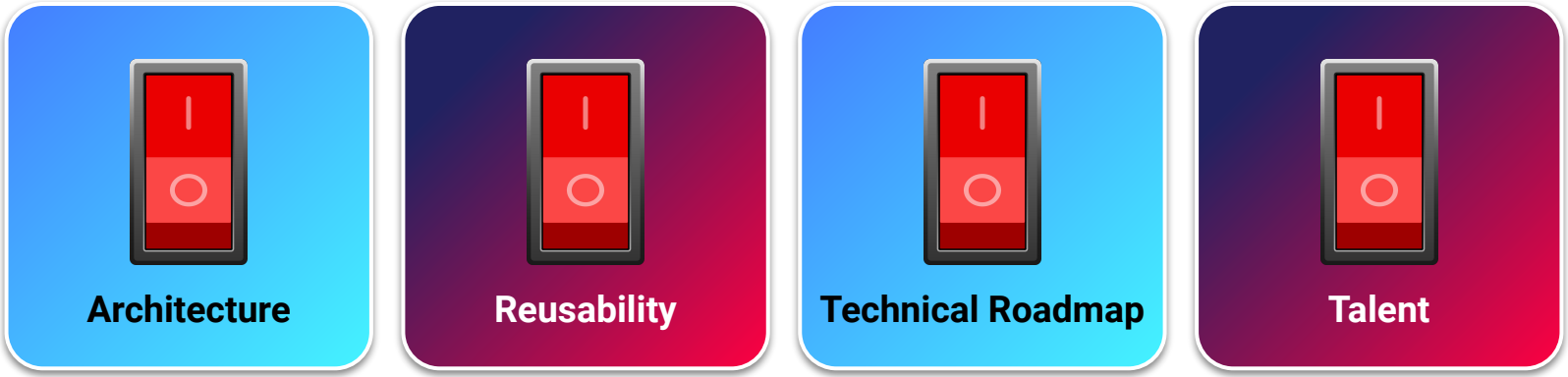
Leapfrogging cloud transformation, neglecting the cultural and organizational changes results in **higher risk** and **more complexity**.



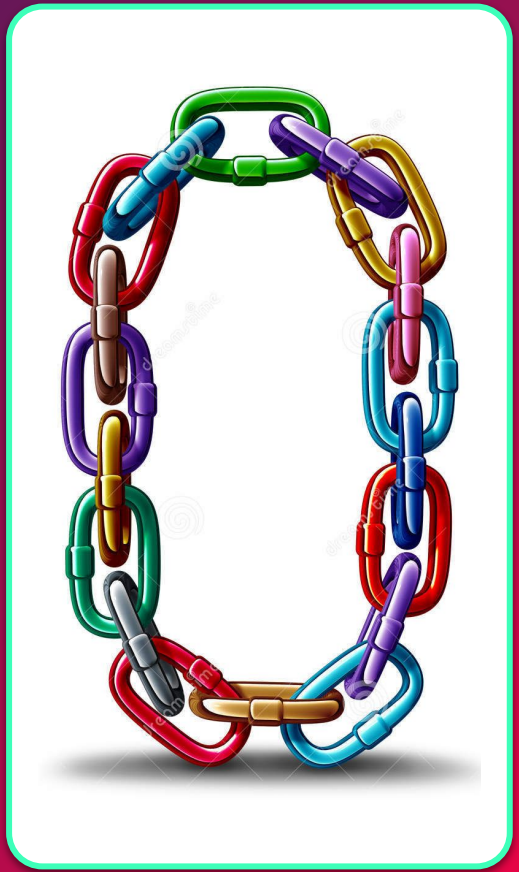
Running before walking

1. Completely **different** operating model
2. **Low levels** of engineering maturity
3. **Complex** technical solution
4. **No man's land** of compliance and internal processes

Switches



At low levels of engineering maturity we can’t break the link between cloud architecture and operating model

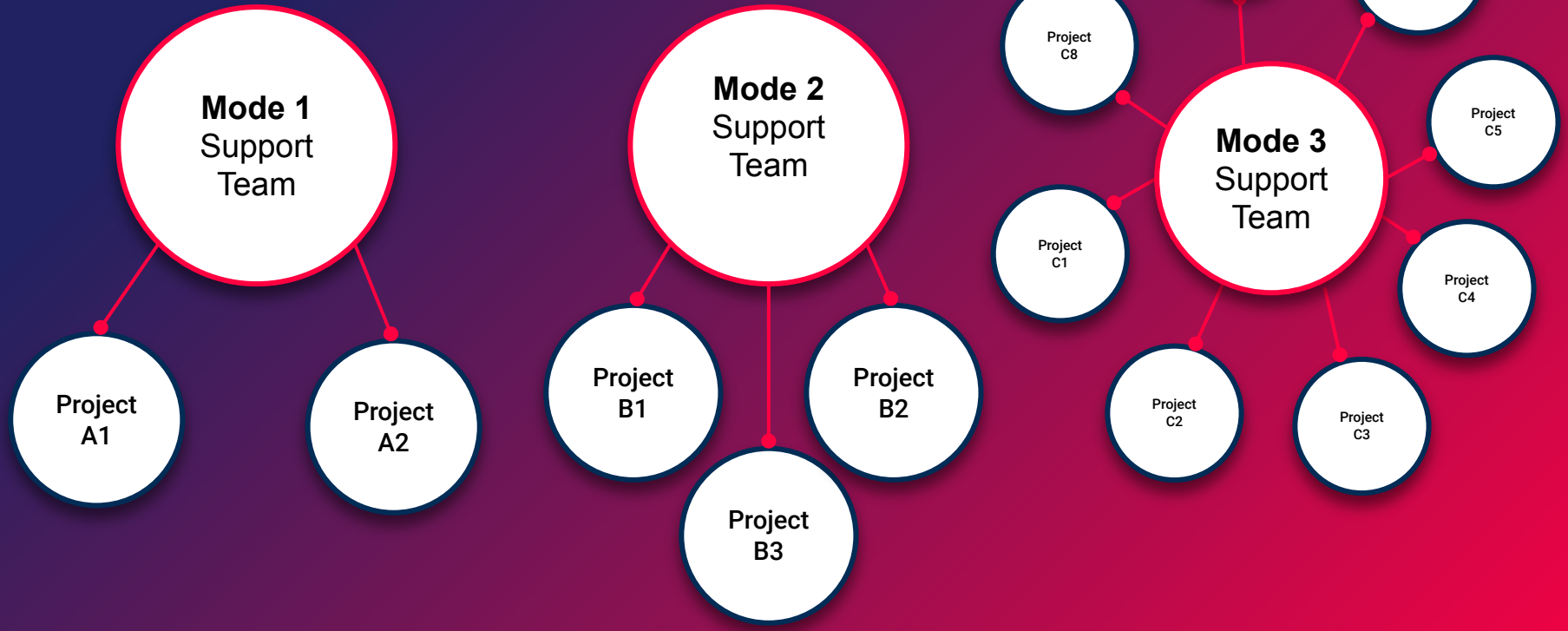


Two Key Questions

“What is our level of engineering maturity ?”

“What operating model are we going to use?”

Limiting the Support ratio...



CONTINO

Thank You

London
london@contino.io

New York
newyork@contino.io

Atlanta
atlanta@contino.io

Melbourne
melbourne@contino.io

Sydney
sydney@contino.io

Brisbane
brisbane@contino.io

www **contino.io** **continohq** in **contino**