AWS for Industries

Low latency cloud-native exchanges

by John Kamel, Gianluca Bertelli, Kandha Sankarapandian, Mike Perna, and Thomaz Silva | on 15 NOV 2023 | in Amazon CloudWatch, Amazon EC2, Amazon ElastiCache, Amazon Kinesis, AWS Transit Gateway, Financial Services, Industries | Permalink | Comments | Share

Large capital market infrastructure providers (specifically exchanges) around the world use the cloud to accelerate their innovation and modernization journey. Trading workloads present unique challenges, including low latency responses, low jitter network performance, and ensuring fairness. With different services and features, Amazon Web Services (AWS) enable exchanges to develop use cases built with cloud-native services to meet an array of required performance characteristics.

This blog explores architecture patterns and key AWS services used to build a prototype cloud-native exchange with comparable latency profiles to on-premises solutions.

Key challenges of building a low latency solution on-premises:

- Co-location or proximity space is limited and expensive, often requiring long lease terms. Most importantly this equipment is not scalable and needs to be provisioned at maximum estimated capacity requirements.
- On-premises infrastructure requires specialized equipment and configurations, and are therefore expensive to own and operate.
- Data portability and global market access is challenging with on-premises technology. Exchanges looking to expand into new revenue streams or pools globally are often limited by the physical aspects of on-premises infrastructure and location of their data centers.

AWS Prototype Architecture

Design considerations were tailored around latency requirements for a Central Limit Order Book matching engine supporting traditional and regulated markets. Exchanges using a Micro Auction model are also well within the scope of what this protype has shown.

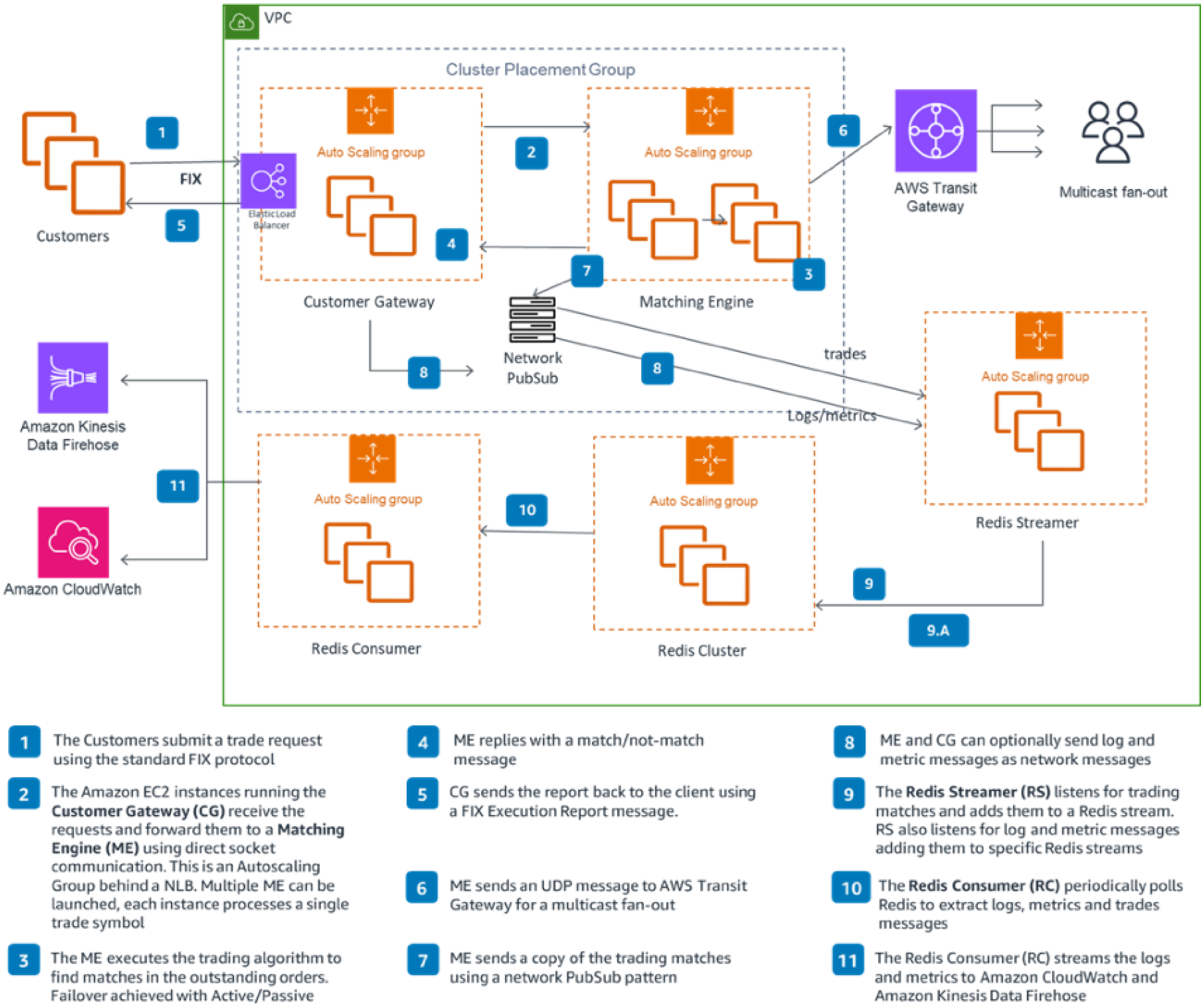
A critical requirement for this solution is that the communication between components (like the client gateway and matching engine) occurs as directly as possible to minimize any additional network latency and jitter. Physical proximity of the components is the first critical point.

A number of AWS services were used to design and test a scalable cloud-native exchange prototype meant to optimize latency. Diving deeper into the specific architecture (shown in Figure 1), the solution leverages Amazon Elastic Compute Cloud (Amazon EC2) cluster placement groups and in-memory caches to enable low latency. Amazon EC2 offers constructs such as Cluster Placement Groups (CPG) to place interdependent Amazon EC2 instances in close proximity inside the same data center within an Availability Zone.

This strategy reduces the number of network hops and enables the low latency node-to-node network communication needed for high performance workloads. Amazon EC2 cluster placement groups are being used to minimize latency between the customer gateways, matching engine nodes, and market feed nodes.

The order generator simulates hundreds of clients submitting orders by sending 10,000 FIX new order messages per second to the customer gateway. It sends orders sequentially to the matching engine emulator using a library that enables straightforward implementation of direct socket communication (ZeroMQ) to achieve low latency.

The matching engine then publishes trade data (over ZeroMQ) to be consumed by a Redis stream, which publishes transaction data to Amazon Kinesis Data Stream and Amazon CloudWatch for post-trade analysis and telemetry. This also includes multicast through the AWS Transit Gateway for distribution of tick data from the match. After the matching engine completes these processes, it returns a message to the customer gateway through ZeroMQ and uses the Transit Gateway to publish trade's market data through the multicast. The customer gateway then returns a FIX message back to the order generator (the simulated customer).



1 The Customers submit a trade request using the standard FIX protocol

2 The Amazon EC2 instances running the **Customer Gateway (CG)** receive the requests and forward them to a **Matching Engine (ME)** using direct socket communication. This is an Autoscaling Group behind a NLB. Multiple ME can be launched, each instance processes a single trade symbol

3 The ME executes the trading algorithm to find matches in the outstanding orders. Failover achieved with Active/Passive

4 ME replies with a match/not-match message

5 CG sends the report back to the client using a FIX Execution Report message.

6 ME sends an UDP message to AWS Transit Gateway for a multicast fan-out

7 ME sends a copy of the trading matches using a network PubSub pattern

8 ME and CG can optionally send log and metric messages as network messages

9 The **Redis Streamer (RS)** listens for trading matches and adds them to a Redis stream. RS also listens for log and metric messages adding them to specific Redis streams

10 The **Redis Consumer (RC)** periodically polls Redis to extract logs, metrics and trades messages

11 The Redis Consumer (RC) streams the logs and metrics to Amazon CloudWatch and Amazon Kinesis Data Firehose

Figure 1: Reference architecture

Experimental Results

From an architecture standpoint, experiments were initially executed using Redis, which did not produce the targeted latency characteristics (see Figure 2). Further testing was done using ZeroMQ. It enables socket-to-socket communication without an intermediate broker, enabling direct communication between the client gateway and the matching engine. This approach was chosen since it significantly reduced P99 round-trip latency.

Despite realizing that Redis can be used as an alternative and to guarantee order sequencing, this approach introduced significant latency. As a comparison, where Redis was used as an intermediate node between the producer (client gateway) and the consumer (matching engine) it was tested in two configurations (LMOVE and LPOP) where it acted as a Pub/Sub client.

While using Redis LMOVE, message order is preserved in a queue and provides reliability and fault tolerance in the event of matching engine node failure. However, it only allows processing of one message at a time, causing significant tail latency (P99).

When using just Redis LPOP, it was possible to enable a pipeline approach (retrieve multiple messages at the same time) to increase performance. However, the reliability offered by the Redis LMOVE model was lost. Figure 2 shows the latency impact of using Redis instead of ZeroMQ.

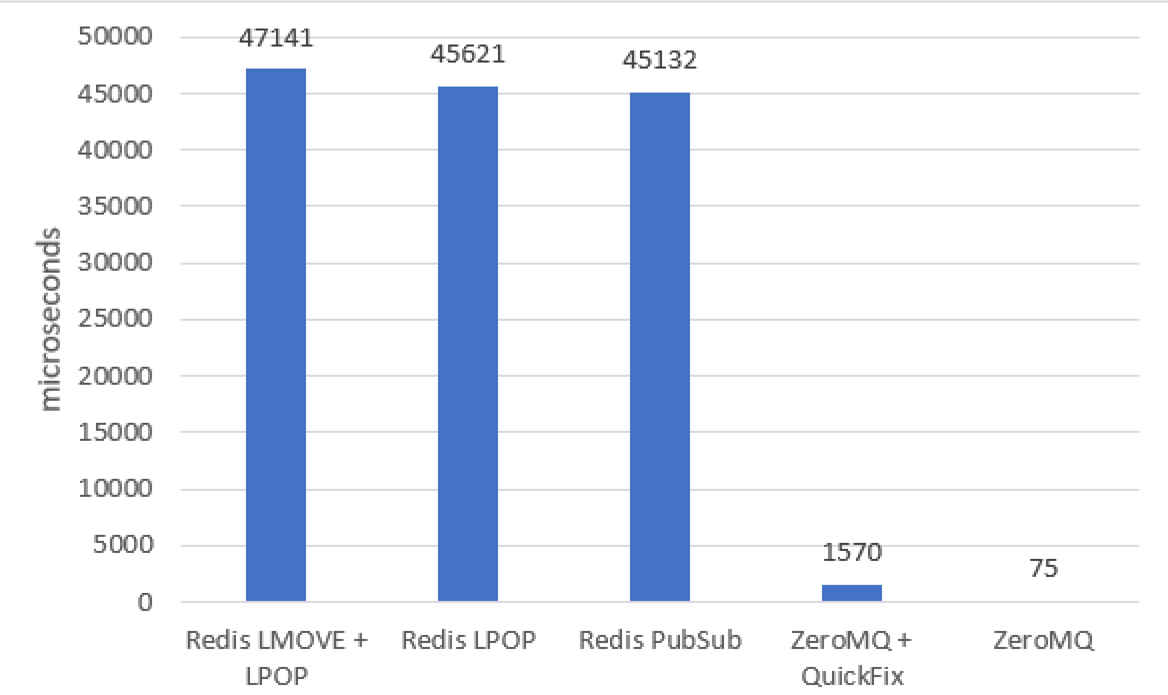


Figure 2: Comparison of round-trip latency (P99) results for the Broker and without Broker approaches

Incorporating the previous implementation approach, the prototype demonstrated a round-trip latency of 55-124 microseconds (P50) and 75-157 microseconds (P99). This includes an intentional delay inside the matching engine to mimic the full matching logic and the time for the order gateways to send a client acknowledgement.

The variability in round-trip latency figures were due to the type of Amazon EC2 instances used. When running the tests with different Amazon EC2 instance types, it was noted that for the implementation of a single threaded customer gateway, and matching engine, the CPU clock frequency has the biggest impact on the overall latency.

For the purposes of this experiment, and at the time of testing, it was found that using an m5zn.6xLarge achieved the lowest P50 and P99 latency.

Design and Benefits Considerations

Running high-frequency trading systems in a cloud-native environment brings several benefits for a financial institution. A cloud-native approach enables the development of systems that are resilient, highly available, and optimized for elastic capacity.

One existing perception about deploying high-frequency low latency systems with a cloud-native approach is the assumption of latency penalties introduced by virtualized environments and their distributed nature. While this is a cloud-native exchange prototype that excludes full implementation of a resilient multi-Availability Zone (AZ) deployment, it does mimic full matching engine logic delays. It also demonstrates key building blocks of a scalable architecture, including market data consideration and full transaction logging.

At the session layer (Figure 1, Step 7) we used ZeroMQ for high performance network communication. However, it is possible to use other high performance messaging solutions (for example, NNG or Aeron), and even further performance gains can be achieved with network stack tuning with tools like DPDK.

In a full deployment of a production-ready solution, multiple client gateway instances would scale-out to support the load of multiple incoming customer orders. Each client gateway communicates with the matching engine(s), which handles a single transaction at a time to ensure sequencing. This may require multiple customer gateway instances fronted by a low latency load balancer, and matching engines in active/passive mode. In the event of a failure, recovery is achieved with minimal recovery time objective (RTO).

This prototype also uses auto scaling groups to provision Amazon EC2 instances for each of the components, which enables auto-provisioning. Another optimization approach is using containers in clusters organized by an orchestrator like Amazon Elastic Kubernetes Service (Amazon EKS).

By relying on Amazon EKS, the client gateway and matching engine logical components could be organized in containers deployed inside the same compute instance. This provided the potential to reduce even more latency between the individual components of the system. An additional benefit is the reduced overhead in spinning up new nodes required for horizontal scalability or for fault tolerance.

The architecture presented (Figure 1) also used multicast over AWS Transit Gateway to return the latest price quotes after every order update. AWS Transit Gateway supports multicast traffic between subnets of attached VPCs within a region, and it serves as a multicast router for instances sending traffic destined for multiple receiving instances. In addition, using AWS Resource Access Manager (AWS RAM), multicast data can be shared to subscribers in other AWS accounts—making this an option for regional exchange data delivery.

Trading systems require strong durability of transaction logs, which is imperative to supporting failover processes, generating application logs and telemetry metrics and supporting order and trade queries. This needs to be performed without an impact to latency.

To minimize the performance impact, a Distributed Logging pattern was used to extract data, log metrics from the nodes, and send them to Redis. Each node in the system publishes log messages and metrics to a ZeroMQ Pub/Sub socket. A subscriber node listens for these messages, and adds them to a Redis Stream. Messages added to the Redis

Stream are processed by a consumer node, which then publishes the stream messages as Amazon CloudWatch log entries and Amazon CloudWatch metrics.

Other downstream pipelines can be integrated at this stage, such as sending messages to an Amazon Kinesis data stream for analytics and feeding data lake, risk, and compliance systems. While Amazon ElastiCache is a potential alternative for this stage, it was not used since it introduced latency. In the architecture we presented, some Redis Cluster nodes were present in the Cluster Placement Group, reducing the network latency required to write transaction logs into the Redis cluster.

Another consideration of building exchange solutions is collocation. Historically, it has not been possible to achieve collocation in the cloud. However, with the recent introduction of Amazon EC2 Shared Cluster Placement Groups, it is now possible to achieve collocation and proximity solutions on AWS.

Conclusion

In this blog we demonstrated the ability to run a cloud-native exchange in an AWS Region and obtain high performance, round-trip latency of 55-124 microseconds (P50) and 75-157 microseconds (P99). In this case, design principles were focused around latency while meeting other basic exchange requirements around logging and market data distribution.

In any solution, design considerations should be accounted for a full before conducting a solution buildout. Fully consider ensuring data persistence, minimizing cluster management, achieving specific resiliency requirements, or offering co-location in the cloud solutions. AWS has many services and design patterns to support businesses and their technology needs as they continue their innovation journey.

Check out more AWS Partners or contact an AWS Representative to know how we can help accelerate your business.

We would like to acknowledge Ganesh Raam Ramadurai, and Cameron Worrell for their contributions to this blog. It couldn't have been done without them.

Further Reading

- Rethinking the low latency trade value proposition using AWS Local Zones
- Increasing flexibility: Capital Markets firms in the cloud adapt more quickly
- Introducing Amazon FinSpace with Managed kdb Insights, a fully managed analytics engine, commonly used by capital markets customers for analysis of real-time and historical time series data

TAGS: Capital Markets, High Frequency Trading

Comments