ADR-021: Moneta Core Architecture

Date: 2025-06-15

Status: Proposed

1. Context

The Moneta Network is a platform designed to facilitate transactions and interactions between Membership Organizations (MOs) and Publishers. The central hub, **Moneta Core**, acts as a trusted intermediary, ensuring all business flows are securely processed, verified, and recorded. It is responsible for managing the lifecycle of MOs, publishers, and the digital mPass, as well as handling transactions, disputes, reconciliation, and settlement processes.

A core tenet of the platform is that all business processes are represented as versioned, multi-signature **"Contracts"**. These contracts are initiated by an MO or publisher, sent to Moneta Core for validation and its own signature, and then forwarded to the counterparty. This creates an auditable, non-repudiable trail for every significant event on the network.

The system must be built on AWS using serverless technologies to handle high throughput, specifically for the event ingestion endpoint which is expected to handle 10 million requests per month, with peaks of 1,000 requests per second. The architecture needs to be secure, scalable, highly available, and cost-effective.

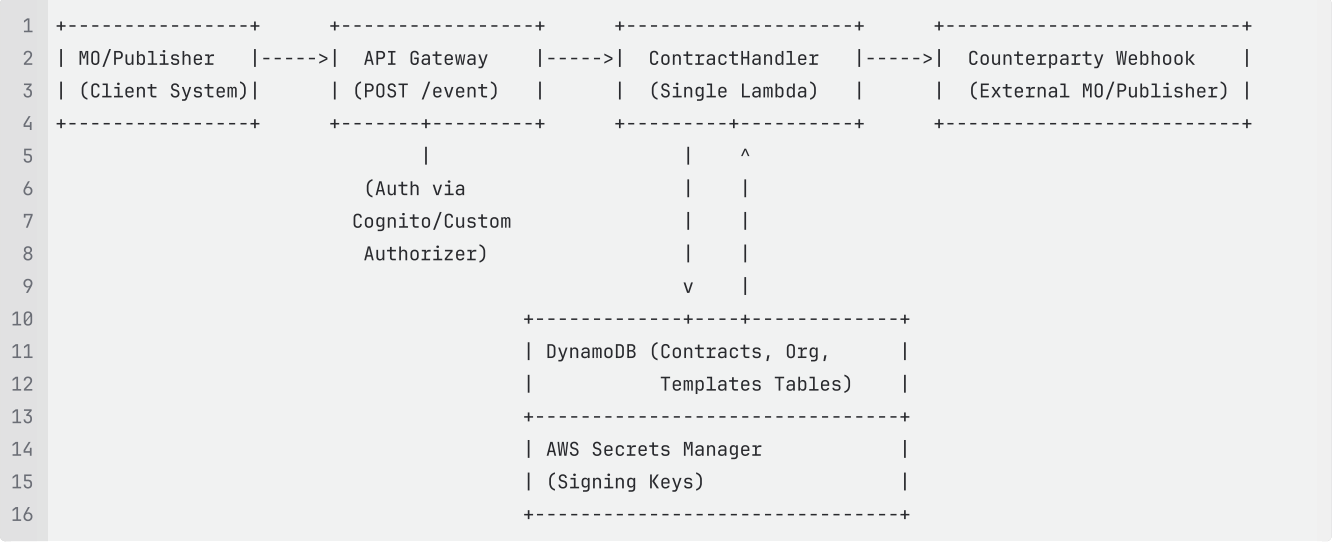
This ADR proposes the foundational architecture for the **Moneta Core**, focusing on the critical **Core Module** (contract and event management) and its interaction with other business modules like Organization and mPass management.

2. Decision

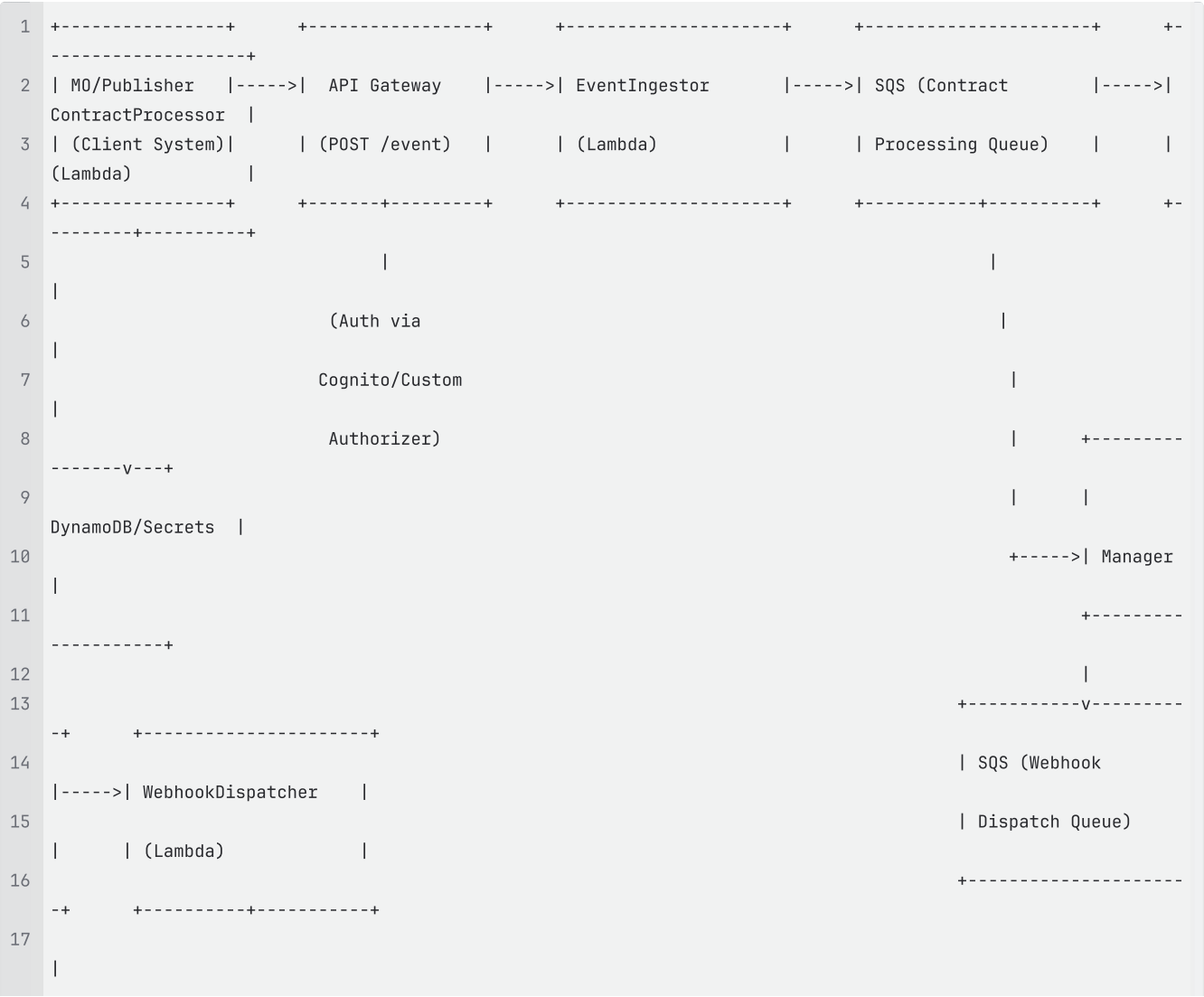
We will adopt a **decoupled, asynchronous architecture (Option 2)** for event processing. While a synchronous model (Option 1) offers simplicity, it fails to meet the system's high-throughput and reliability requirements. The asynchronous design separates the initial, fast ingestion of events from the more complex, stateful contract processing and dispatching logic, ensuring the system is scalable, resilient, and performant enough to handle the expected load.

2.1. High-Level Architecture Diagram

Option 1: Synchronous Monolithic Handler



Option 2: Decoupled Asynchronous Handler



```
+------------v-------------+
| Counterparty Webhook     |
| (External MO/Publisher)  |
+--------------------------+
```

Comparison of Options

Criterion	Option 1: Synchronous	Option 2: Asynchronous (Decoupled)
Performance/Latency	**Poor**. High latency for the client, as it must wait for all processing, including external webhook calls.	**Excellent**. Very low latency for the client (202 Accepted response), as heavy processing is offloaded.
Reliability/Availability	**Low**. A single failure (e.g., webhook timeout) causes the entire request to fail. No built-in retries.	**High**. SQS queues provide durability and automatic retries. Failures are isolated and don't impact ingestion.
Scalability	**Poor**. Throughput is limited by the slowest step (the webhook). Cannot easily scale to 1000 req/sec.	**Excellent**. Can easily absorb traffic spikes. Processing scales independently by adjusting Lambda concurrency.
Simplicity	**High**. The logic is in one place, making it easy to understand initially.	**Medium**. More moving parts (queues, multiple Lambdas) require more complex debugging and end-to-end tracing.
Cost	Slightly lower due to fewer components, but potentially higher compute costs from longer Lambda durations.	Slightly higher due to SQS costs, but overall more cost-effective at scale due to optimized, shorter Lambda runs.

2.2. Core Module: Event and Contract Flow

The critical POST /event flow will be implemented as a single, synchronous process:

1. **API Gateway**: A REST API will serve as the entry point. The /event endpoint will be configured with a Lambda proxy integration. A Custom Authorizer Lambda will handle authentication and authorization.
2. **Contract Handling (ContractHandler Lambda)**: This single Lambda function executes the entire workflow:

- **Initial Validation**: It performs validation on the incoming event payload.
- **Template Retrieval**: It fetches the relevant contract template from a ContractTemplates DynamoDB table. Templates will be cached in memory to reduce latency.

- **Full Contract Validation**: It validates the contract payload and signatures against the rules defined in the template.
- **Database Interaction**:
  - It queries the Contracts table by contract_id.
  - If the contract is new, it's created. If it exists, it's updated with the new signature and event log entry. Optimistic locking will be used to handle concurrent updates.
- **Signing**: It retrieves Moneta's private key to signs the contract.
- **Dispatching**: It constructs the outgoing event and directly calls the counterparty's registered webhook via an HTTP request.
- **Response**: The Lambda waits for the webhook call to complete and then returns a final status (200 OK or an error code) to the original caller.

Logging: All events and key processing steps will be logged to CloudWatch for auditing. AWS X-Ray will be enabled for tracing.

2.3. Database and Storage Design

- **DynamoDB - Contracts Table**:
  - **Partition Key**: contract_id (String).
  - **Attributes**:
    - description: (String) A human-readable summary of the contract's purpose.
    - template_id: (String) Links to the ContractTemplates table.
    - mo_id: (String) FK to MembershipOrganizations table.
    - publisher_id: (String) FK to Publishers table.
    - mPass_id: (String) FK to MPasses table.
    - version: (Number) The version of the contract template used.
    - payload: (Map) The data payload of the contract.
    - signatures_map: (Map) Contains signatures from all parties.
    - event_logs: (List) A history of events for this contract.
    - status: (String) The current status of the contract.
    - created_at: (Number) Unix timestamp.
    - finalized_at: (Number) Unix timestamp.
- **DynamoDB - Organizations Table**:
  - **Partition Key**: org_id (String, UUID v4).
  - **Attributes**:
    - name: (String) The legal name of the organization.
    - description: (String) A brief description of the organization.
    - created_at: (Number) The Unix timestamp of when the organization was registered.
    - created_by: (String) The identifier of the user who created the record.
    - status: (String) The current status of the organization (e.g., PENDING, ACTIVE, SUSPENDED).
- **DynamoDB - MembershipOrganizations Table**:
  - **Partition Key**: id (String, UUID v4).
  - **Global Secondary Index (GSI)**:
    - **Partition Key**: mic (String) - Moneta Issuer Code for human-friendly lookups.
  - **Attributes**:
    - name: (String) The official name of the Membership Organization.
    - status: (String) The current status (e.g., Pending, Active, Suspended, Destroyed).
    - createdAt: (Number) The Unix timestamp of when the record was created.
    - updatedAt: (Number) The Unix timestamp of the last update.
    - metadata: (Map) A JSON blob for MO-specific details like contact info, webhook endpoints, and other configuration.

        - certificates: (List) A list of identifiers for associated certificates.
        - services: (List) A list of supported service codes.
        - industries: (List) A list of applicable industry codes.
- **DynamoDB - Publishers Table**:
    - **Partition Key**: publisherID (String, UUIDv4).
    - **Global Secondary Index (GSI)**:
        - **Partition Key**: mpc (String) - Moneta Partner Code for human-friendly lookups.
    - **Attributes**:
        - name: (String) The official name of the Publisher.
        - status: (String) The current status (e.g., Pending, Active, Suspended, Destroyed).
        - ssoGroupID: (String, optional) Identifier for SSO grouping if applicable.
        - createdAt: (Number) The Unix timestamp of when the record was created.
        - updatedAt: (Number) The Unix timestamp of the last update.
        - createdBy: (String) The identifier of the user who created the record.
        - metadata: (Map) A JSON blob for Publisher-specific details like service endpoints, categories, and logo URL.
        - certificates: (List) A list of identifiers for associated certificates.
        - services: (List) A list of supported service codes.
        - industries: (List) A list of applicable industry codes.
- **DynamoDB - MODevelopers Table**:
    - **Partition Key**: id (String, UUID v4 from Cognito User Pool).
    - **Global Secondary Index (GSI)**:
        - **Partition Key**: mo_id (String) - To query all developers associated with a specific MembershipOrganizations record.
    - **Attributes**:
        - name: (String) The developer's name.
        - email: (String) The developer's email address.
        - created_at: (Number) The Unix timestamp of when the developer was added.
        - created_by: (String) The identifier of the user who added this developer.
        - status: (String) The developer's status (e.g., ACTIVE, INACTIVE).
- **DynamoDB - PublisherDevelopers Table**:
    - **Partition Key**: id (String, UUID v4 from Cognito User Pool).
    - **Global Secondary Index (GSI)**:
        - **Partition Key**: publisher_id (String) - To query all developers associated with a specific Publishers record.
    - **Attributes**:
        - name: (String) The developer's name.
        - email: (String) The developer's email address.
        - created_at: (Number) The Unix timestamp of when the developer was added.
        - created_by: (String) The identifier of the user who added this developer.
        - status: (String) The developer's status (e.g., ACTIVE, INACTIVE).
- **DynamoDB - MPasses Table**:
    - **Partition Key**: mPassID (String, UUIDv4).
    - **Global Secondary Indexes (GSIs)**:
        - **GSI 1 (mPassNumber-index)**: PK: mPassNumber - For direct lookup via human-friendly number.
        - **GSI 2 (moID-mPassNumber-index)**: PK: moID, SK: mPassNumber - To find all passes for an MO.
        - **GSI 3 (billingAccountID-mPassID-index)**: PK: billingAccountID, SK: mPassID - To find all passes for a billing account.
    - **Attributes**:

    - userID_internal: (String) The issuing MO's internal user identifier.
    - status: (String) e.g., Pending, Active, Suspended_UserLock, Suspended_AdminLock, Expired, Destroyed.
    - tier: (String) e.g., "Standard", "Platinum", "Gold".
    - activePublicKeyID: (String) UUIDv4, FK to a MPassPublicKeysTable.
    - createdAt: (Number) Unix timestamp.
    - updatedAt: (Number) Unix timestamp.
    - expiresAt: (Number) Unix timestamp.
    - metadata: (Map) A JSON blob for any other mPass-specific details.
- **DynamoDB - BillingAccounts Table**:
  - **Partition Key**: billingAccountID (String, UUIDv4).
  - **Global Secondary Index (GSI)**:
    - **GSI 1 (moID-accountType-index)**: PK: moID (String, FK to MembershipOrganizations.id), SK: accountType (String).
  - **Attributes**:
    - ownerEntityID: (String, UUIDv4) Identifier for the primary owner (e.g., mPassID or publisherID).
    - status: (String) e.g., "Active", "Suspended", "Closed".
    - paymentResponsibilityInfo: (Map) JSON blob detailing payment responsibilities, especially for Company/Family types.
    - policySettings: (Map) JSON blob for defining policies like spending limits or category restrictions, primarily for Family accounts.
    - createdAt: (Number) Unix timestamp.
    - updatedAt: (Number) Unix timestamp.
- **DynamoDB - ContractTemplates Table**:
  - **Partition Key**: template_name (String, e.g., "transaction-contract").
  - **Sort Key**: version (Number).
  - **Attributes**:
    - description: (String) A human-readable description of the template.
    - created_at: (Number) The Unix timestamp of when the version was created.
    - is_latest: (Boolean) A flag to easily identify the current, active version of a template.
    - template_payload: (String) The full YAML content defining the template's attributes, schema rules, and signature rules.

2.4. Other Modules

Each module will be implemented as a distinct microservice, with its own API endpoints and Lambda functions, interacting with the DynamoDB tables defined above.

- **Organization Management**:
  - **Functionalities**:
    - Manage MO lifecycle (onboarding, status changes, offboarding).
    - Manage Publisher lifecycle (onboarding, status changes, offboarding).
    - Manage MO and Publisher certificates, services, and industries.
    - Manage MO and Publisher specific policies.

Design ADR-024: Moneta Network Organization Management

- **mPass Management**:
  - **Functionalities**:
    - Manage mPass lifecycle (issuance, status changes, locking, expiration, destruction).
    - Manage mPass-related certificates and policies.

Detail design ADR-023: mPass Management

- **Transaction Management**: This module binds with the Core module to process transaction contracts. It contains the specific business logic for validating, processing, and recording financial transactions.

This module technical document will be detailed at another document

- **Reconciliation Management**: This module binds with the Core module to process reconciliation contracts, comparing transaction records between MOs and Publishers to ensure consistency.

This module technical document will be detailed at another document

- **Settlement Management**: This module binds with the Core module to process settlement contracts, calculating and recording the final revenue splits between MOs and Publishers.

This module technical document will be detailed at another document

- **Network Policy Management**: This module binds with the Core module to process policy-related contracts, enabling the creation and enforcement of network-wide rules and configurations.

This module technical document will be detailed at another document

3. Consequences

3.1. Pros

- **Architectural Simplicity**: The flow is straightforward, with fewer moving parts (no SQS queues, fewer Lambdas), making it easier to develop, debug, and reason about.
- **Immediate Feedback**: The client receives an immediate, definitive 200 OK or error response, as the entire process is synchronous. This eliminates the complexity of handling eventual consistency on the client side.
- **Lower Operational Overhead**: Managing a single Lambda function and its configuration is simpler than managing a multi-stage, event-driven pipeline.

3.2. Cons

- **Reduced Performance & Scalability**: API latency will be significantly higher because the client must wait for all downstream processing to complete, including database operations and a call to an external webhook. This tight coupling makes the endpoint's performance dependent on the slowest part of the chain (often the external webhook).
- **Lower Availability & Resilience**: The system is more brittle. A failure at any step, especially a timeout or error when calling the counterparty's webhook, will cause the entire API request to fail. The architecture lacks the built-in retry and dead-lettering capabilities that SQS provides, making it less resilient to transient network issues.
- **Risk of Lambda Timeouts**: The combined execution time of the entire synchronous flow could exceed the maximum API Gateway (29s) or Lambda (15min) timeouts, particularly if a counterparty webhook is slow to respond. This poses a significant operational risk at scale.
- **High Throughput Challenges**: Meeting the peak requirement of 1000 requests per second will be very challenging with this synchronous model due to the increased latency and blocking nature of the API call.

4. Design Assessment

4.1. Security

- **Authentication & Authorization**: API Gateway will use a Custom Lambda Authorizer to validate signed requests from MOs and Publishers, ensuring that only authenticated and authorized entities can submit events.

- **Data Protection**: All data will be encrypted in transit using TLS and at rest using DynamoDB's default server-side encryption. Sensitive signing keys are never exposed; they are stored and used within AWS Secrets Manager.
- **Least Privilege**: The ContractHandler Lambda function will be granted a narrowly scoped IAM role with the minimum permissions required to access specific DynamoDB tables and secrets.
- **Input Validation**: Strict validation of the contract payload against the template schema is critical to prevent injection attacks and ensure data integrity.

4.2. Performance

- **Write-Intensive Operations**: The requirement for fast create/update actions is severely hampered by this synchronous design. The total latency is the sum of all serial operations, including a network call to an external webhook, which is unpredictable and out of our control.
- **Database Queries**:
  - **By contract_id**: Queries using the Partition Key are extremely fast (single-digit ms) and scale well.
  - **By other attributes**: The requirement to search by mo_id, publisher_id, mPass_id, type, and created_at makes DynamoDB complex. To avoid slow, expensive table scans, **multiple Global Secondary Indexes (GSIs)** must be created on the Contracts table. For example: GSI1: PK=mo_id, GSI2: PK=publisher_id, etc. Searching by a date range on created_at alongside other filters may require exporting data to a dedicated search service like Amazon OpenSearch.
- **Conclusion**: The current synchronous architecture is **not suitable** for a write-intensive system with high throughput and low latency requirements. It introduces significant bottlenecks and reliability risks.

4.3. Cost

- **Pros of the Solution**: The serverless model is highly cost-effective as you only pay for what you use, avoiding idle provisioned infrastructure.
- **Cons of the Solution**: The primary cost driver will be DynamoDB writes, especially with the necessary GSIs. Each write to the main table will be multiplied across all indexes, significantly increasing cost.
- Estimated Monthly Cost (On-Demand Pricing)
  Based on 10,000,000 contracts per month, with an average item size of 5KB, 5 writes and 4 reads per contract.

Service	Component	Calculation	Estimated Cost/Month
DynamoDB	Writes (Main Table + 4 GSIs)	50M writes * (5 WCU + 4 WCU) = 450M WCUs	~$562.50
	Reads	40M reads * 2 RCUs = 80M RCUs	~$20.00
	Storage (Main + GSIs)	~100 GB	~$25.00
AWS Lambda	Compute & Requests	10M requests @ 500ms/512MB	~$38.00
API Gateway	HTTP API Requests	10M requests	~$10.00
Other	CloudWatch, S3, Secrets Manager	Logging, Archiving, Key Storage	~$61.00
Total			**~$716.50**

Note: This estimate is approximate. Actual costs may vary based on traffic patterns, data size, and the number of GSIs implemented.

4.4. Multi-Region & Global Scalability

- **Latency**: The current single-region, synchronous design is **unsuitable for a global user base**. A request from an MO in Europe to the API in us-east-1 will incur high round-trip network latency. The problem is amplified by the synchronous webhook call, which forces the Lambda to wait for another transatlantic round trip, leading to poor user experience and a high risk of timeouts.
- **Data Residency**: Storing all data in a single AWS region may violate data sovereignty regulations like GDPR, which require certain user data to remain within specific geographic boundaries.
- **Availability**: A single-region deployment creates a single point of failure. A regional service outage would bring down the entire Moneta Core platform.
- **Path to Globalization (Requires Architecture Change)**:

1. **Multi-Region Deployment**: The application stack (API Gateway, Lambda) would need to be deployed to multiple regions (e.g., us-east-1, eu-west-1, ap-southeast-1). Amazon Route 53 can then use latency-based routing to direct clients to the nearest regional endpoint.
2. **Global Data Plane**: **DynamoDB Global Tables** must be used to replicate data across all active regions. This provides low-latency read and write access to a local copy of the data, but it significantly increases cost, as every write is replicated (and paid for) in each region.
3. **Decoupling is Essential**: The synchronous webhook call must be replaced with an asynchronous, event-driven pattern (e.g., using SQS or EventBridge). This would allow the regional API endpoint to respond quickly and reliably, while a separate process handles the dispatch to the external partner system. This change is fundamental and moves the design away from the simple, monolithic model proposed in Section 2.