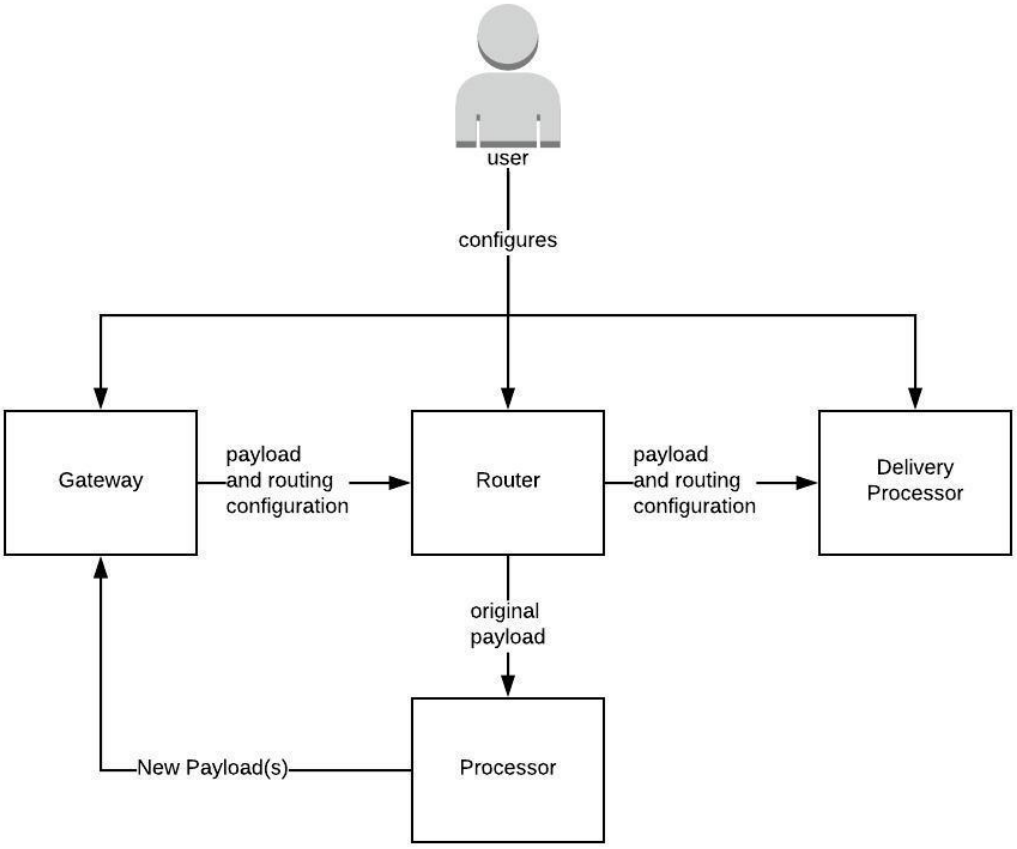
webMethods Future State Architecture

Data Fabric 2.0

Purpose

This document describes a future state architecture for WebMethods. WebMethods is a system that supports data integration between multiple systems. WebMethods currently runs on top of several EC2 instances.

Current Data Flow



Data enters WebMethods through a Gateway. A Gateway may support data being pushed to it, such as an HTTP Gateway, or it may support polling such as an FTP Gateway. A transaction is started each time data is sent to a Gateway or a Gateway finds new data during polling.

Gateways read the data for a transaction and extracts metadata used to route the payload. This metadata is composed of a Sender, a Receiver, and a Document Type. The Gateway then sends the payload and the routing metadata to Trading Networks. The process to read routing metadata from payloads can be configured on each Gateway.

Trading Networks sends transactions to Process Services. The routing metadata is used to look up which Process Service receives the transaction.

One type of Process Service is a Delivery Process. The routing metadata is used to look up which Delivery Process receives the transaction. In some cases, Trading Networks may do some translation of the payload. Trading Networks may also extract different routing metadata for the Delivery Process lookup. Assigning Delivery Processes to routing metadata, as well as additional routing metadata extraction and any optional translation are all configured on Trading Networks.

Transactions sent to a Delivery Process are delivered to the receiving system. Some Delivery Processes will batch up transactions to be delivered all at once. The details of how to send transactions to a receiving system are configured on each Delivery Process.

Requirements

Transactions must be able to be delivered to different protocols depending on the receiver and the data type.

The system must support transforming the original data. For instance, spaces may be replaced by commas. Whether this transformation should occur is based on the sender, receiver, and/or data type.

The system must support sending batches of documents on a scheduled basis. Some batches must be a single concatenated payload. Whether this batching should occur is based on the sender, receiver, and/or data type.

Authentication should occur using Azure AD. Access to UI functionality must be roles based.

Roles can be configured to allow the following actions:

- See transaction history for a specific data system or for all transactions
- View the payload for transactions for a specific data system or for all transactions
- Retry failed transactions for a specific data system or for all transactions
- Download transaction payloads for a specific data system or for all transactions

Admin users should be able to perform the following actions:

- Configure new integrations
- Configure gateways, process services, and delivery services
- Create roles and assign users to roles
- Create data types

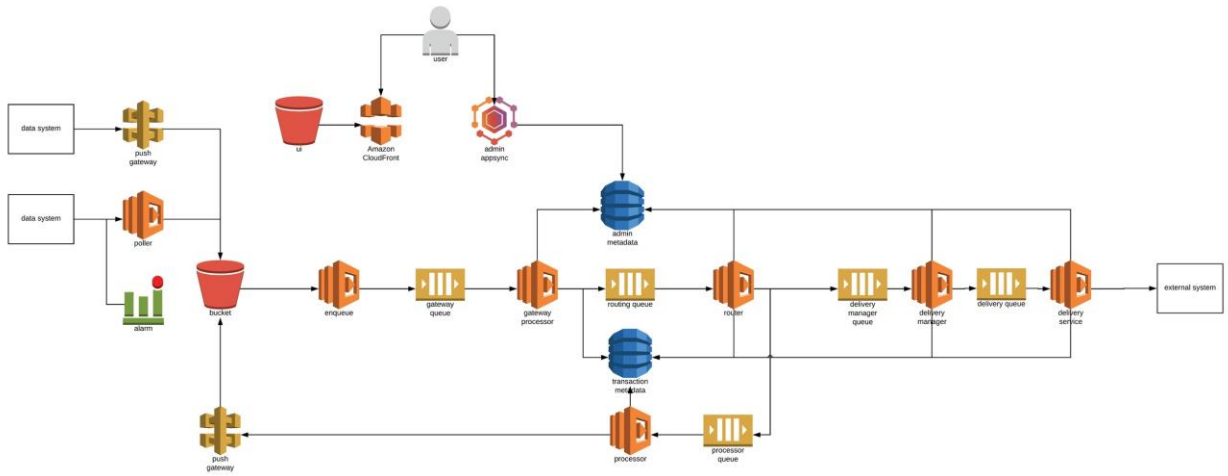
Data Fabric Principles

Data Fabric divides data movement into three general phases. Ingestion, Processing, and Delivery. The current version of Data Fabric handles ingestion and is able to deliver data by

allowing data systems to query the data, or by publishing the data to end systems. Custom processing is managed in a subsystem called Data Factory that can pull in related data types and perform transformations before sending the data to an end system.

The following architecture is a high-level view of an expanded Data Fabric with greater capabilities that allows for more data movement use cases, including those required by a webMethods rearchitecture.

High Level



A transaction is started when a source Data System sends a payload to a Gateway or a polling Gateway pulls data from a source Data System. This data is immediately stored in S3.

S3 triggers a routing metadata extraction process in a Gateway Lambda. The Gateway Lambda sends the routing metadata to a Router Lambda.

The Router Lambda will use the routing metadata to decide what Process to send the payload to. Some processes may transform the data and send it back through a gateway to be ingested again. Other processes may store the data for later concatenated delivery. Processes may have multiple Process Queues that allow for multiple independent work streams.

Delivery Processes are a special type of Process. Delivery Processes send data to a consuming Data System.

All Lambdas use a similar computational pattern for work intake, throttling, failure handling, and logging. Every Lambda which handles a transaction stores execution and result metadata in a DynamoDB table.

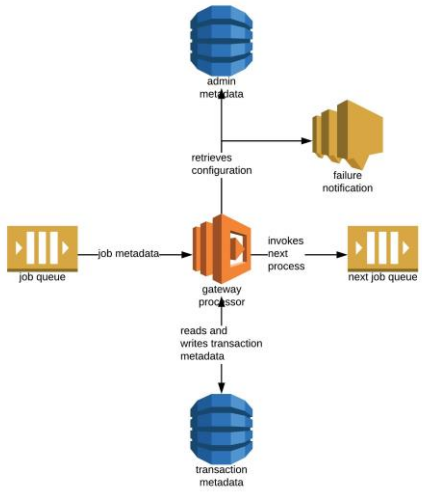
Gateways, Routing, and Processes are all configurable using data from an admin DynamoDB table.

Users can see the status of a transaction, see transaction logs, view transaction payloads, or retry a transaction through an admin portal delivered from S3 through Cloudfront. The admin portal integrates with an AppSync GraphQL API. Authentication into the GraphQL interface is done by Cognito which uses Coke's Azure AD as an IDP.

Each transaction goes through one Gateway, and either one delivery service or one processing service. However, there are multiple gateway, delivery, and processing services with different functionalities in order to accommodate different types of files coming from and to different sources.

Computational Pattern

All Lambdas are built using a common computation pattern. This pattern allows for scalability, managed failures, configurability, and traceability.



Work is put onto an SQS queue which triggers a Lambda. The SQS queue can be encrypted but cannot be FIFO. Throttling can be done by limiting the Lambda concurrency.

Execution is configured using an admin metadata DynamoDB table. The result and any logs from the execution are saved to a transaction metadata DynamoDB table.

Failures can be handled in two ways. Retriable failures should fail the Lambda, which will retry the computation again after the SQS timeout period. Non-Retriable failures can either go onto a failure dead letter SQS queue, or directly to a dead letter SNS topic. Dead letter queues are not implemented using the built in Lambda mechanism when triggered by SQS. This is because SQS is a stream-based invocation, and the built in DLQ mechanism requires an asynchronous non-stream-based trigger. Instead, DLQ capabilities are written into the Lambda code. The failure is written to the SQS queue or SNS topic, and the Lambda returns a success to the triggering SQS queue. Failures to write to the dead letter queue or topic are treated as retriable failures.

Each invocation of a Lambda should write a single log message. This log messages should be in a structured JSON format. Each log message should include as much data as possible, while taking into consideration the cost for the total volume of log messages. Structured JSON logs are useful for querying logs during incidents, extracting metrics, and generating alerts.

SQS triggered Lambdas can have a batch size greater than 1. Batch size must be taken into account for failure handling and logging. A larger batch size means less overhead per message, but retriable failures for a single message will require retrying all messages. Additionally, when the batch size is greater than 1, each individual record handled by the Lambda invocation should write a log message and not the entire invocation.

SQS guarantees at-least-once delivery. What this means is that a message will be sent to at least once Lambda. However, there is the possibility that SQS will deliver a duplicate message to a second Lambda. In order to prevent reprocessing or redelivery of data, all messages from SQS need to be deduplicated.

A series of workers built with this pattern can be used to divide a long running job into multiple parts. The failure of each part is isolated so costly computation can be separated from computations that have a higher chance of failure

Data Ingestion

Implementation

All data ingestion lands in an S3 bucket. Success should not be indicated to source system until the S3 write has been confirmed. Each Gateway type writes to a specific subfolder in the S3 bucket.

S3 writes trigger an Enqueueing Lambda which forwards the message into the SQS queue for a specific Gateway Lambda. The SQS queue that receives the job will depend on the path of the S3 bucket.

Each type of Gateway would have additional infrastructure to support pulling data into S3 or receiving pushed data and landing it in S3. The architecture for specific Gateway ingestions is outside of the scope of this document.

Pulling data can be achieved with a Lambda triggered by a Cloudwatch event.

Failure Modes

Failures can occur in two places during ingestion: prior to the drop into S3 and during execution of the Enqueuing Lambda.

In the case of a failure prior to the data being written to S3, the specific gateway ingestion process should send a failure response back to the system. It is then on the source system to

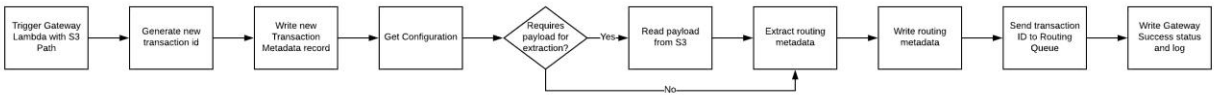
retry the push. In the case of a polling ingestion process failing to write to S3, the process should have a method for pulling that data on the next scheduled execution. All specific gateway architectures should include alerts for failed ingestion to S3.

The Enqueuing Lambda is a variation of the normal computational pattern described above. Since S3 triggers are asynchronous and non-stream based, there is no unlimited retry capability but there the built in DLQ mechanism is available. This means that the Enqueuing Lambda needs to do as little as possible since its failure mechanism is not as strong. Any failures that occur during the Enqueuing Lambda will be automatically retried twice and then sent to a DLQ. Failures of the Enqueuing Lambda will be configured to trigger alerts in CloudWatch to notify Operations teams of an issue. Failure of the Enqueuing Lambda could be due to a failure to write to the Gateway SQS queue or during processing.

FTP Polling Example

Routing Metadata Extraction

Implementation



Extraction of routing metadata from a payload is done by a Gateway Lambda. Each specific type of Gateway has its own Gateway Lambda and Gateway SQS queue. S3 events are put onto the Gateway SQS queue by the Enqueuing Lambda.

Gateway Lambdas start by trying to read a record representing the transaction from the transaction metadata DynamoDB table. This record will only exist if the Gateway Lambda invocation is handling a retry from a previously failed attempt to extract the routing metadata. If a transaction metadata record exists and already has routing metadata, then the Gateway Lambda writes the record to the Routing Queue. If a transaction metadata record exists but there is no routing metadata, then the Gateway Lambda proceeds to extract the routing metadata. If a transaction metadata record does not exist, then the Gateway Lambda writes one to the transaction metadata DynamoDB table.

To extract routing metadata, the Gateway Lambda reads configuration data from the admin DynamoDB table. The Gateway Lambda may also need to read the payload from S3. The format of the configuration data, as well as the process to combine configuration data and a payload to extract routing metadata is specific to each particular Gateway and is outside the scope of this document. After the routing metadata is extracted, it is written to the transaction metadata DynamoDB table. Once the routing metadata has been saved, a message is put on the Routing Queue. Finally, the transaction metadata record is updated with a successful status for the gateway processing.

Failure Modes

Routing metadata extraction failures can occur during execution of the Gateway Lambda. The Gateway Lambda implements the computational pattern described above and performs error handling as described in that section.

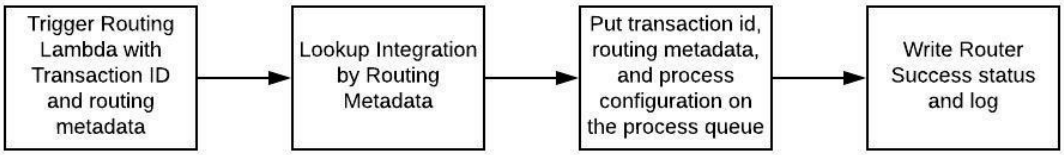
Failure of the Gateway Lambda could be due to the following reasons:

- failure to write data to the transaction metadata table
- failure to read configuration from the admin metadata table
- failure to read the payload from S3 due to an S3 error
- failure to read the payload from S3 due to the data being missing
- failure to extract routing metadata from the payload
- failure to write to the Router Queue

Failure to extract routing metadata is a non-retriable failure. Failure to read the payload from S3 due to the data being missing is a non-retriable failure. All other failures described are retriable. Any unexpected error should be treated as a non-retriable failure.

Routing

Implementation



Routing is done by a Routing Lambda. The Routing Lambda is triggered by a Routing SQS queue. Gateway Lambdas write data to the Routing SQS queue to begin routing transactions.

The Routing Lambda uses the routing metadata passed from the Gateway Lambda to lookup an integration configuration in the admin metadata table. The integration configuration will have a process type and process configuration. The router uses the process type to decide which process queue to send the transaction to. The transaction is sent to the processor on its queue along with the process configuration. Finally, the transaction metadata record is updated with a successful status for the router processing.

Failure Modes

Routing metadata extraction failures can occur during execution of the Gateway Lambda. The Gateway Lambda implements the computational pattern described above and performs error handling as described in that section.

Failure of the Routing Lambda could be due to the following reasons:

- failure to read the transaction metadata from the transaction metadata DynamoDB table

- failure to read configuration from the admin metadata DynamoDB table due to a DynamoDB failure
- failure to read configuration from the admin metadata DynamoDB table, due to the integration not being configured
- failure to extract new routing metadata
- failure to write to the transaction metadata DynamoDB table after extracting new routing metadata
- failure to send a message to the Translation SQS queue
- failure to send a message to the Delivery SQS queue

Failure to read configuration due to the integration not being configured is a non-retriable failure. Failure to extract new routing metadata is a non-retriable failure. All other failures described are retriable. Any unexpected error should be treated as a non-retriable failure.

Processing

Implementation

Implementation details of specific processors is outside the scope of this document. However, there are some common patterns.

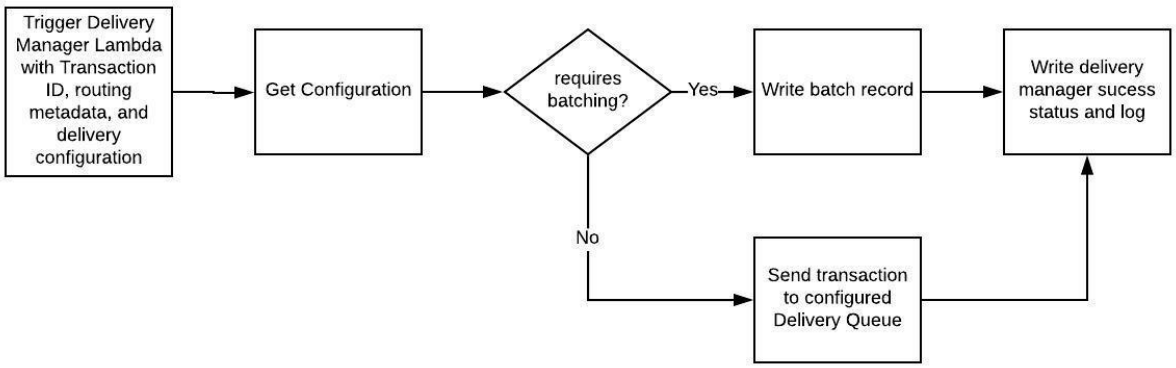
All processors should be triggered by an SQS queue using the computation pattern described above. Messages from SQS may then trigger a simple Lambda or may kick off a Step Function workflow.

Processors generally send messages back through to the Router. In order to do this, they will send messages to a Reprocess Gateway. Processors can send new routing metadata to the reprocess gateway, and optionally a new payload. Processors can send 0, 1, or many messages to the Reprocess Gateway for every incoming invocation.

Transaction Delivery

Implementation

Transaction delivery is divided into two phases. First is delivery management which handles common delivery requirements such as delays or throttling. Next, is the actual delivering of the payload to a downstream system.

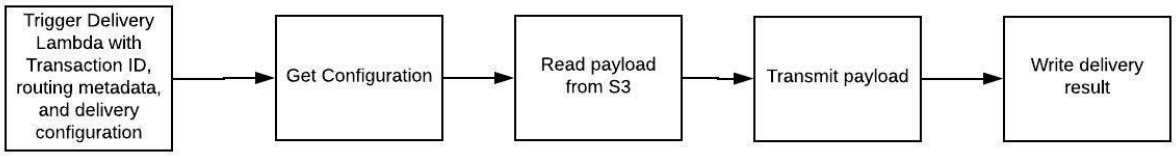


Delivery management is done by a Delivery Management Lambda. The Delivery Management Lambda is triggered by a Delivery Management SQS queue. Messages are put onto the Delivery Management SQS queue by the Routing Lambda.

The Delivery Management Lambda begin by reading configuration from the admin metadata DynamoDB table. The Delivery Management Lambda then decides whether the payload can be sent directly to the consuming Data System, or if some form of delivery management such as throttling or delayed delivery is required.

If no delivery management is required, the Delivery Management Lambda uses the delivery configuration sent from the Routing Lambda to choose a Delivery Queue to send the transaction to. The Delivery Management Lambda sends the transaction to the configured SQS queue, and then writes a successful delivery management status to the transaction metadata record.

If delayed delivery or throttling is required, a record is written to a delivery management DynamoDB table. At some future point, a Cloudwatch Event will trigger a process to pick up delayed or throttled writes and send them to the appropriate delivery queue.



Delivery of the payload is handled by a Delivery Lambda. Each protocol for delivering payloads is a separate Lambda. Each Delivery Queue has its own dedicated Delivery Lambda that it triggers.

The details of how payloads are transmitted are specific to each delivery process and are outside the scope of this document. The general pattern is for the Lambda to get any necessary configuration from the admin metadata database, read the payload from S3, transmit the payload, and then write a delivery result to the transaction metadata table.

Failure Modes

Delivery failures can occur during execution of the Delivery Lambda or the Batch Delivery Lambda. The Delivery Lambda and the Batch Delivery Lambda implement the computational pattern described above and performs error handling as described in that section.

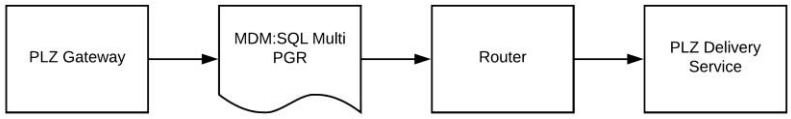
Failure of the Delivery Lambda could be due to the following reasons:

- failure to retrieve configuration
- failure to read payload from s3 due to an s3 failure
- failure to read payload from s3 due to the data being missing
- failure to transmit the payload
- failure to write the delivery result

Failure to read the payload from s3 due to the data being missing is a non-retriable failure. Failure to write the delivery result is also a non-retriable failure, because we do not want to transmit the data gain, we just want to alert on the failure. All other failures described are retriable. Any unexpected error should be treated as a non-retriable failure.

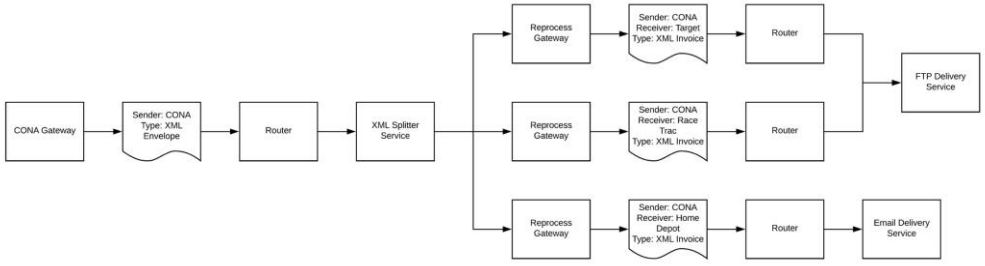
Data Examples

Simple Passthrough



The simplest use case is a simple pass through of data. A payload is sent or retrieved from some source system. Its routing metadata is extracted, and the router is configured to send it directly to a consuming system.

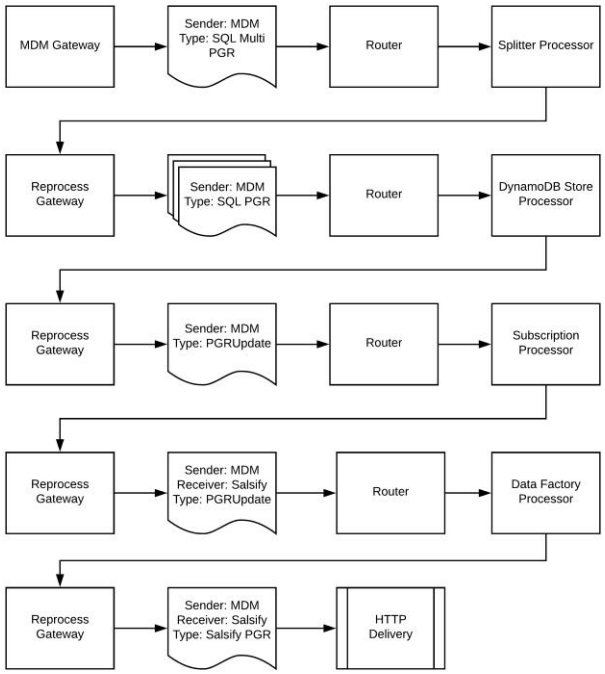
XML Split



An example of a more complex workflow splitting an XML payload into multiple documents. A gateway may send an XML payload that contains multiple subdocuments that each need to get sent to their own consuming systems. The router is configured to send the original XML envelope to an XML Splitter Service. The XML Splitter Service sends each of the subdocuments to the Reprocessing Gateway, each with a different receiver. The XML Splitter Service has been preconfigured to know where the subdocuments are in the XML structure, and how to retrieve

the receiver information based on the data type. The router is configured to send each subdocument from the Reprocessing Gateway out for delivery. Some subdocuments may be configured to be sent through different protocols.

Master Data Syndication



Master Data Syndication is also possible. An example would be an update to a PGR being saved into DynamoDB for querying, and then sent to Salsify.

An MDM Gateway would query the MDM SQL servers and send the full response to the router. This response would be the full result set of the query and would contain multiple records. The router would be configured to send this result set to a JSON Splitter Service which would split the result set into individual records and send each one to the Reprocessing Gateway with a new data type.

The Router would be configured to send each individual, raw record coming through the Reprocessing Gateway to a DynamoDB Storage Service. This service would write the records into DynamoDB, and then send a message back through the Reprocessing Gateway with a new data type representing the updated record, and with a payload that included the previous and new values.

The Router would be configured to send the PGR update record to a subscription service. The subscription service would be configured to send the PGR update record back to the Reprocessing Gateway with multiple times with different receivers, including Salsify.

The router would be configured to send PGR update records for Salsify through the Data Factory Service. The Data Factory service would query for additional data related to the PGR and build a custom Salsify PGR record. This Salsify PGR record would be sent to the Reprocessing Gateway with a new data type.

Finally, the router would be configured to send Salsify PGR data for Salsify to an HTTP delivery service which would send the data to Salsify.

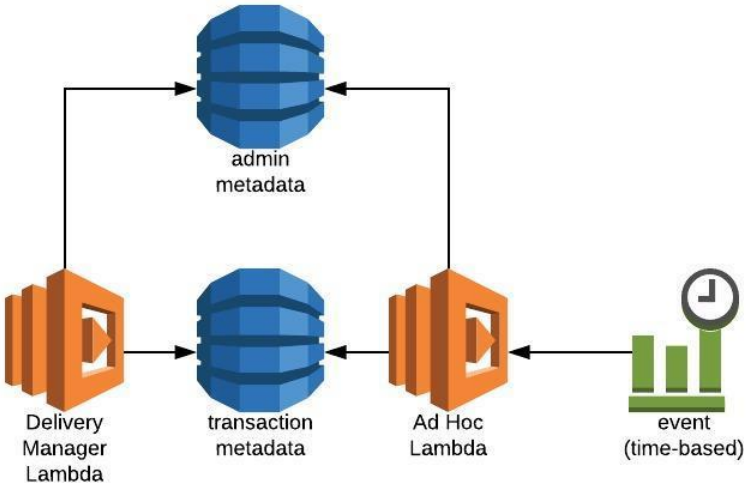
Admin Interface

The admin interface is made up of a Web UI connected to a GraphQL admin API. The Web UI is hosted in S3 and uses Cloudfront for CDN and Route53 for DNS. Authentication is performed through Cognito, which uses the Coca-Cola Azure AD as an IDP. The GraphQL admin API is built on AppSync, which integrates directly with Cognito for authentication and authorization. The admin API is able to read and write data from the admin metadata DynamoDB table, read data from the transaction metadata table, send messages to any Lambda Queues, and read data from S3.

Requirements for the UI are documented in the WebMethods UI Requirements document.

Operations

Ad Hoc Holds

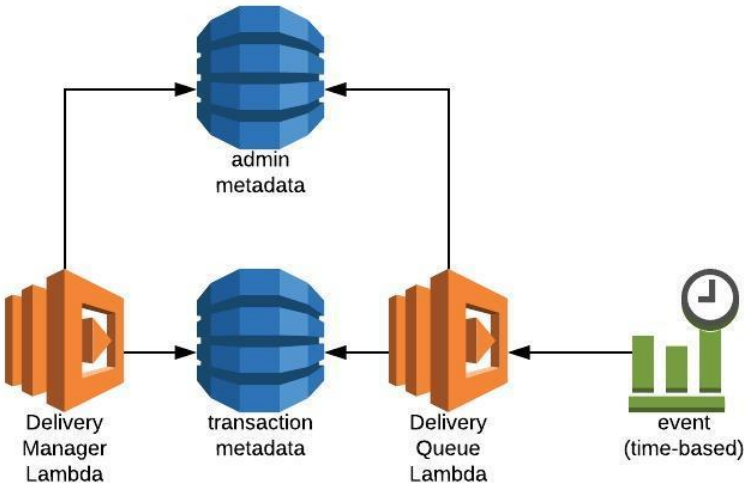


Ad-Hoc holds can be placed on Data Systems. An ad-hoc hold prevents delivery from going to a Data System between two time periods. Creating an ad-hoc hold is done through the operational UI. When an ad-hoc hold is placed, a record is written in DynamoDB with an active status, and the start and end times. When the Data Manager receives a transaction, it checks the transaction's receiver against the list of active ad-hoc. If an ad-hoc hold is found, and the current time is within the time bounds for the ad-hoc hold, then a hold record is written to Dynamo representing the transaction.

A Cloudwatch Alarm exists which triggers an Ad-Hoc Lambda on some preconfigured, recurring schedule. The Ad-Hoc Lambda searches for active ad-hoc holds whose end time has passed. The Ad-Hoc Lambda find all hold records associated with the ad-hoc hold and resubmits those transactions to the Data Manager. The Ad-Hoc Lambda then updates the ad-hoc record to inactive.

Sender Data Systems are not impacted when an ad-hoc hold is placed on a Receiver Data System.

Delivery Windows



Data Systems can be configured to have delivery windows. When the Data Manager receives a transaction, it checks the transaction's receiver to determine if there is a delivery window. If the receiver Data System is configured to have a delivery window and the current time is not within the window, then a hold record is written to Dynamo representing the transaction.

When a Data System is configured to have a delivery window, a Cloudwatch alarm is created for the Data System. The Cloudwatch alarm is configured to trigger a Delivery Queue Lambda at the beginning of the delivery window. The Delivery Queue Lambda queries for all transactions that are on hold for the Data System and resubmits them to the Delivery Manager.

Sender Data Systems are not impacted when a Receiver Data System has a delivery window.

Manual Edits

Manual edits are supported through the operational UI. Operations users can craft a payload and send it through a specific Gateway Service. The payload will be saved to S3, and a message will be placed on a Gateway Queue. Operational users can also submit the payload to the router by specifying routing metadata. In this case, the payload will be saved to S3, a transaction record will be created in DynamoDB, and a message will be placed on the Router Queue.

Manual edits can be created from an existing transaction. This is useful when a transaction fails and needs to be manually edited to be reprocessed. When a manual edit is created based on an existing transaction, its transaction record refers to the original transaction.

Data Retention

Data is retained in S3 for 45 days. A data archival policy exists to archive data to Glacier after 45 days.

There are multiple transaction metadata DynamoDB tables at any given point in time. New tables are automatically created on a schedule. Only a single transaction metadata table is written to by Gateway Lambdas at any given time. The messages sent between Lambdas through SQS queues contains the DynamoDB table with the transaction metadata, as well as the transaction metadata identifier. DynamoDB tables are backed up and destroyed after 45 days.

Security

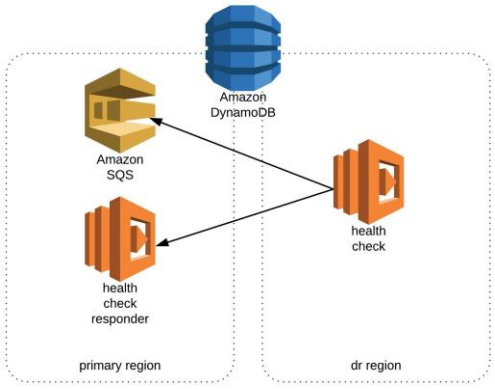
Encryption At Rest

Data can be encrypted in S3, DynamoDB, and SQS using KMS encryption. KMS keys can be rotated using Secrets Manager and a key rotation Lambda.

Encrypted File Processing

For payloads that are encrypted as they are sent through a gateway, but need to be decrypted before processing or delivery, the payload can be routed to the Decryption Processing Service. The Decryption Processing Service will decrypt the payload, and resubmit it to the Reprocessing Gateway.

Disaster Recovery



All DynamoDB tables are created as global tables and cross-region S3 bucket replication is turned on to provide for data redundancy.

In the event of a failure of Lambda or SQS in the primary region, a hot-standby DR instance is available in the DR region that can be failed over to. All Lambdas in hot-standby mode are throttled to 0 concurrent instances to prevent execution. Failure is detected using a Health

Check Lambda in the DR region that attempts to use SQS and Lambda in the primary region. A Health Check SQS queue and Health Check Responder Lambda exist in the Primary region. The DR Health Check Lambda puts a message on the Health Check SQS queue and reads the message out. It also invokes the Health Check Responder and compares the result to a known response. If either of these health checks fail, the system begins a failover.

Failover is achieved by writing a record to the global DynamoDB instance marking the DR region as the active region. All Lambdas are required to search for this value to verify that their region is active. After the active region marker has been changed, all Lambdas in the DR environment have their concurrency limits removed. A search is done in DynamoDB for all transactions that are currently in process and are requeued.

Account Structure

Development, testing, and live system integrations happen within three accounts: Dev, NonProd, Prod. The Prod account is where the live system integrations exist. Developers have no access to this account, through the CLI or through the console. NonProd contains a non-live version of the system where configuration and testing take place. Developers have read only access to the NonProd account. An instance of GoCD runs on an EC2 instance inside of the NonProd account, which deploys infrastructure to all three Accounts. The Dev account is for each developer to create their own instance of the platform for development and testing. Developers have read and write access to a limited number of services in the Dev account.

Testing

Component Failure Testing

The blast radius due to the failure of an individual component should be minimized due to the computational pattern described above. Automated testing is performed to verify the blast radius. Test transactions are run through a testing environment, and components are either configured to fail in predetermined ways, or components are destroyed. The automated tests verify that the system was able to handle the failure as described in this document.

End to End Performance Testing

Performance of the system is automatically tested by landing test documents into S3 and delivering the results to a mock REST endpoint.