jamk.fi

Implementing Amazon Web Services integration connector with IBM App Connect Enterprise

Noora Backlund

Bachelor's thesis
September 2020
Technology
Bachelor of Engineering in Information Technology, Software Development

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s)	Type of publication	Date
Backlund, Noora	Bachelor's thesis	September 2020
		Language of publication: English
	Number of pages: 73	Permission for web publication: x

Title of publication

Implementing Amazon Web Services integration connector with IBM App Connect Enterprise

Degree programme

Bachelor of Engineering in Information Technology, Software Development

Supervisor(s)

Salmikangas, Esa

Assigned by

Teemu Tasanto, ATR Soft Oy

Abstract

The emergence and growth of cloud environments has created new avenues for enterprise application integration and brought about new challenges on how to go about integrating the new with the old. A generic integration connector was created to allow for transfer of messages from Amazon Web Services -based applications to third party services by using IBM's App Connect Enterprise Enterprise Service Bus platform.

The integration was implemented through design-based research methods, where the researcher acts as both the implementor and the designer, creating new knowledge based on theoretical basis through iterative cycles until an ideal solution is reached.

The integration was designed according to Service Oriented Architecture principles and reliable messaging considerations. The resulting integration consisted of Simple Queue Service message queues and a message consumer Lambda function on the Amazon Web Services environment, and a connector application, intermediate message queues and recipient-specific connector applications on the Enterprise Service Bus platform. Components on both environments were built to be easily scalable and extendable to a dynamic category of services.

The resulting integration implementation fulfilled basic Service Oriented Architecture principles and provided methods to account for reliable messaging requirements. The proof of concept integration provided a basic functionality for message transfer, with expansion and adjustment possibilities for different environments.

Development of the integration with design-based research methods provided valuable insights into possible implementations and direction of future research.

Keywords/tags (subjects)

Integration, AWS, ESB, IBM, ACE, Amazon Web Services, SOA, Reliable Messaging

Miscellaneous

Contents

Figures

Tables

1 Introduction

The turn of the millennium saw ever more rapid changes in the information technology landscape, with adoption of Software-as-a-Service (SaaS) systems increasing over five-fold since 2011 (Burger 2014). While cloud-based solutions have further gained popularity in 2010s, the old and established on-premise enterprise solutions are by no means going away, with organizations increasingly often maintaining a combination of on-premise and cloud applications.

Integration of applications has long been a staple for larger enterprises looking to simplify and improve data administration, with multiple solution providers to cater for companies' integration needs for on-premise systems. With the adoption of cloud-based services, the demand for integrations has also shifted to target applications on the cloud. Cloud applications may consist of individual functionalities such as message queues, machine learning modules or data storage, for which data connections and integrations need to be separately configured. In a more traditional on-premise scenario, the integrated system has been a complete solution with existing interfaces. With no precedent or ready-made implementations on integrating a cloud platform with more traditional integration solutions, a proof-of-concept methodology is required to effectively integrate cloud services with existing integration solutions.

While traditional methods of integration are well-established and in widespread use, the challenge lies in designing and implementing a solution which extends the traditional integration architecture to function for a cloud-based environment. Simultaneously, the solution should utilize the unique features of the cloud to support the traditional methods of implementation. This research focuses on creating an integration to connect an Amazon Web Services environment with an on-premise IBM Enterprise Service Bus implementation to allow for message transfer from cloud-based applications to on-premise applications in a scalable and extendable manner.

2 Research design

2.1 Purpose of the research

The goal of the research is to introduce an integration to connect endpoints in modern cloud environments to systems running within an on-premise environment, allowing for data transfer between the endpoints. With adoption of cloud-based systems experiencing double-digit percentual growth rates (Gartner, 2018; Synergy, 2018; Forbes, 2018), the proposed integration solution would also have to scale in both data volume and number of endpoints in order to provide a viable long-term solution to integrating the cloud with the on-premises environment.

As the integration solution is intended to be implemented within an enterprise environment with production- and business-critical data, special attention must be paid to the quality and reliability of the data to be transferred through the integration. Enterprise advisor Gartner estimated the annual cost of poor-quality data to an enterprise to be an average of 15 million USD, with increasing challenges to maintain quality data as the complexity of information technology environments increases (Moore, 2018). In a similar trend, IBM estimated the annual cost of poor-quality data to the US economy to be in the trillions (*Extracting business value from the 4 V's of big data*, N.d.). For the purpose of this research, the quality and integrity of data within the scope of the integration will be accounted for by implementing quality-of-service targets. The targets chosen for this research are outlined by Allen Brown (2001) in his paper on reliable messaging, consisting of the following aspects: sender and recipient authentication, traceability of messages, only-once receipt of messages, preservation of message order and delivery failure notifications to both the sender and recipient of the message.

For the purpose of this research, the scope of the integration is strictly limited to the minimal functional implementation that fulfills the scalability and reliability requirements imposed in previous chapters. The direction of data flow will be limited to cloud services as the system of origin, and on-premise services as the destination

endpoint. Cloud infrastructure and service provider Amazon Web Services, and more specifically their message queue service Simple Queue Service, will be used as the starting point for the data. The middleware used to connect Amazon Web Services to the on-premise services is limited to IBM's App Connect Enterprise and MQ WebSphere systems. On-premise application interfaces are mimicked by implementing a File Transfer Protocol (FTP) server and a Hyper Text Transfer Protocol (HTTP) echo server. The combination of FTP and HTTP servers provide a method to verify a successful distribution of messages to separate destinations, in addition to verifying the functionality for two common interfaces.

While important in an enterprise or production environment, the target solution will not implement thorough logging, high availability or security features, other than the minimum required to satisfy the aspects of reliable messaging. A genuine production environment would likely benefit from implementation of features such as data warehousing, redundancy or clustering of each part of the integration, proxies, monitored access layers and gateways, backups, as well as alerts in case of errors or unexpected situations.

2.2 Research method

The chosen research methodology is design-based research, as this method allows for a flexible way of generating new understanding of how to solve practical issues by iterating and building on existing theory and research. Hoadley (2004) places design-based research on the opposite end of the spectrum from scientifically rigorous and structured experimental research, as design-based research is often highly dependent on the exact context in which it was performed.

Contrary to more traditional research methods which thrive in controlled laboratory settings, design-based research is by nature rooted in real-world settings with a multitude of variables affecting the research outcome. The research process itself is flexible and may be adjusted during research to better fit the required objectives, provided that the adjustments and the reasoning behind the adjustment is documented

through the process. Hoadley (2004) points out, that design-based research approach requires a sufficient amount of self-reflection and introspection from the researcher during the research process but is capable of producing powerful results when the design context is taken into consideration. Wang and Hannafin (2005) also point out, that results obtained through design-based research are considered to fit real-world scenarios better than results obtained in tightly controlled environments. (Hoadley, 2004; Wang, & Hannafin, 2005, 8.)

Additionally, design-based research stresses the importance of flexibility and iterative process, where the researcher and practitioner work in tandem to develop approaches to solve practical problems. In practice, the line between a researcher and a practitioner is often blurred in design-based research, where both sides influence each other to reach the best possible outcome for the challenge studied. The iterative nature of the research method signifies that the cycle of design, implementation, analysis and redesign may be completed multiple times before reaching the optimal solution. (Wang, & Hannafin, 2005, 8.)

Even though design-based research is flexible and dynamic by its nature, the initial designs and theories are still steadily grounded in existing theory and research. Existing literature regarding the subject matter forms a solid base for the research, with focus of the research targeting either problems identified by existing knowledge, or gaps of information. (Wang, & Hannafin, 2005, 8-9.)

Literature on design-based research indicates that the methodology has been heavily utilized in the context of researching education methods, where highly structured scientific experiments cannot account for the possible variables arising from the context of a classroom. In a similar manner, information technology systems and their different implementations in various corporate environments provide such a wide scale of variables, that design-based research is likely to provide a more practical answer to the research problem than a rigorous scientific experiment. Each information technology environment is different, and by accounting for the specific context in which the research has been conducted, the solution may be applied to different

types of environments by evaluating and adapting the context-specific actions taken in this research.

For this research, existing literature and documentation is used as a base for implementation of the solution. Data integration is by no means a new field within information technology, with multitude of recommendations and implementations on effective integration methods, discussed further in Chapter 4. However, as the emergence of cloud environments is a relatively recent phenomenon, the interplay of cloud and on-premise system integrations is a fresh field with little or no studies on the matter. Design-based research method is used to attempt to fill the gap in knowledge in the specific scope of this research by building upon existing theories and knowledge, and by adapting the knowledge to fit modern landscapes.

As the research concerns new knowledge, building the solution will be an iterative process, where challenges and issues encountered during the research will also shape the design to account for the specific context of the environment. Design choices and strategies are identified during the research in order to provide a pragmatic solution to the research problem, which can be applied to similar real-world situations.

The research mainly consists of qualitative research, where information is reviewed and generated to produce new knowledge – in this case, a functional generic integration to unite endpoints in cloud and on-premise environments. The result of the research will be assessed in a qualitative manner to review how well the implementation fits the available theoretical basis.

2.3 Research credibility

In order to ensure credibility and reliability of the research, current knowledge base upon which the new knowledge is built is sourced from peer-reviewed documents where possible. However, as specific information technology systems are rarely the target of scientific publications, documentation, knowledge base articles and best

practice recommendations by leading organizations within their field are considered as credible source of information.

Even though most recent articles are likely to accurately portray the current state of the field, finding recent publications on specific information technology topics can be a challenge. Some topics (such as protocols, data structures and fundamentals of data integration) have changed little in the past decades, and publications from early 2000s or even 1990s still provide valid information for modern implementations, as the principles governing these topics have remained unchanged. While modern publications by reliable sources are preferred, information has also been drawn from older publications where the underlying principles still hold true.

2.4 Research environment

The research environment consists of the bare minimum of services required to implement a generic integration from Amazon Web Services (AWS) to on-premise services. The information technology system for this research includes one AWS account used for cloud-based services, an internal network environment with internet connectivity and the capability to route traffic from its public IP address to internal network, as well as a server infrastructure environment within the internal network to host integration and target application services.

AWS cloud service contains the components responsible for creation of data, as well as the functionality to send data to the on-premise environment. The on-premise environment contains a network device capable of routing the incoming traffic to a dedicated integration server, as well as target servers to which the data from AWS needs to be transferred. A professional corporate network would also need to account for other components and practices not present in this proof-of-concept environment, such as firewalls, network segmentation and active directory components. The introduction of these components would likely result in minor configuration changes unique to the specific target environment, but the principles presented in this research remain the same.

3 Integration and messaging concepts

3.1 Enterprise Application Integration

Enterprise Application Integration (EAI) allows for different applications to exchange information between each other through message-based data transfer (What is integration? N.d.). Lee, Siau and Hong (2003) further extend the term EAI to concern integrating existing applications in an enterprise environment with new applications, allowing for reuse of existing resources in tandem with new features or data brought in by new applications. Lee et al. (2003) also describe EAI as having a middleware serving as a common interface to which all integrated applications can connect to, instead of applications connecting directly to each other in a point-to-point manner, reducing the need for integration programming.

The value of EAI market at a global scale was estimated to be just over 10 billion USD in 2017, with expectations of significant growth in the following years. Drivers for EAI market growth include, for example, growth in implementations of cloud-based ERP systems, increasing adoption of e-business and ever more common automation of business processes. IBM Corporation is among the most important EAI stakeholders, alongside organizations such as Microsoft Corporation, major enterprise resource planning software provider SAP SE and technology company Oracle Corporation. (More 2020.)

3.2 Message

In integration terms, a message is some form of a data structure containing header information and a message body. The data structure is not limited to any specific formats, but rather must be understood by the integration middleware. Message header information contains metadata used by the middleware to process the message, while the message body contains the actual information to be passed between the integrated systems. (Messaging Patterns N.d.)

3.3 Reliable Messaging

Reliable Messaging requires that no messages are lost during integration, even if the messages are not sent in a transactional manner – an especially important concept for high-reliability applications. In practice, this may consist of returning an error response to the source system in the case of a socket-based connection, or retaining the data contained within the message in the case of a failed transfer in order to allow for reprocessing of the message. (Reliability Patterns N.d.)

Brown (2001) further specifies quality-of-service objectives for implementing Reliable Messaging to include the following five principles, of which the first four are considered uncontroversial:

- Authentication of both sender and recipient of messages
- Traceability of messages through the integration
- Message delivered exactly once
- Order of messages should be maintained
- Both the sender and the recipient should be notified of a delivery failure

As the last principle imposes additional requirements on how to receive reports of failure upon the recipient system, delivery failure notification is here considered to be a principle that should be implemented at a best-effort level. Failure notifications will only be sent to systems capable of receiving them, while ensuring that every delivery failure will be reported to some extent.

3.4 Scalability

In the scope of this research, scalability concerns the ability to add new applications to the integration solution with minimal effort or required code changes. An easily scalable solution removes the need to go through the process of regression testing to verify the continued functionality of the integration. Likewise, when a deployment of

new codebase is not required, the time and costs required to introduce new applications to the integration are reduced significantly.

3.5 Service-Oriented Architecture

Service-Oriented Architecture (SOA) denotes a concept, where applications provide functionality in the form of services reusable across the application (Menge 2007).

Services adhering to SOA principles may be coupled together into larger constructs to execute more complex actions, and each service may be reused at will in different constructs. As the users of the service are not aware of the underlying implementation, the services can be changed or updated at will without changes to the service user's implementation. (Tyson 2020.)

3.6 Enterprise Service Bus

Enterprise Service Bus (ESB) is a message-based integration infrastructure that mediates between multiple distinct applications in a flexible manner (Schmidt, Hutchison, Lambros & Phippen 2005; Menge 2007).

ESB is as an enabler of SOA by acting as the flexible connector of services. ESB is not restricted to any specific protocols or standards, thus allowing for creation of connections between systems that would otherwise be incompatible. The implementation of protocols and connections is conducted within the ESB, and as a result the sending and receiving systems do not need to be aware of the internal workings of ESB for the connectivity to take place. (Schmidt et al. 2005; Menge 2007.)

Additionally, ESB may operate on the messages transferred from one application to another in order to satisfy the requirements imposed by the recipient system. This may include actions such as data transformation and mapping between the systems, routing choices, encryption actions or message format changes. ESB may also hold the role of a mediator or a validator, tracking and monitoring the flow of the

messages from one system to another to provide insight into integration reliability and troubleshooting scenarios. (Schmidt et al. 2005; Menge 2007.)

3.7 Amazon Web Services

Amazon Web Services (AWS) is a cloud computing platform providing a variety of services to individuals and enterprises at a global level. These services include, for example, storage space, machine learning capabilities and databases optimized for specific applications. As a cloud computing platform, the underlying infrastructure is managed by Amazon, while the customer manages the service itself. (Cloud computing with AWS N.d.)

3.8 Message Queue

Message Queue (MQ) is a temporary storage location to hold messages in a buffer between a sending and a receiving application, allowing for an asynchronous manner of communication between the two (Sharma 2019; Message Queues N.d.; Johansson 2019).

In addition to asynchronous communication, message queues provide a way to decouple applications from one another, allowing for communication between the two systems without the applications being aware of each other. This provides benefits in easier maintenance of the applications, ability to control communication between the systems in a more specific manner and allows for independent development of each application without interdependencies. (Johansson 2019.)

The specific message queue providers to be used in the scope of this research are IBM Corporation's WebSphere MQ and AWS's Simple Queue Services (SQS). IBM WebSphere MQ service is used alongside ESB functionality, while SQS queues are utilized as data sources on AWS platform.

3.9 AWS Lambda

AWS's Lambda is a service which allows for execution of custom code, which may be triggered from other AWS services, HTTP endpoints or other activity (AWS Lambda N.d.). For the scope of this research, the use cases for Lambda functions consist of manipulating data in SQS queues, namely by populating SQS queues with test data and delivering data from queues to ESB.

3.10 CloudWatch

Amazon CloudWatch is a monitoring and logging service, which provides logs and statistics on health and execution of applications within AWS environment (Amazon CloudWatch N.d.). For the scope of this research, CloudWatch is used to monitor the health and actions of Lambda functions used to populate test data and deliver data to ESB.

3.11 Protocols

3.11.1 HTTP and HTTPS

Hypertext Transfer Protocol (HTTP) is a client-server protocol used to exchange data over the internet. A typical HTTP data exchange starts by the client sending a request to a web server and ends by the server responding with a HTTP response message, containing metadata regarding the request as well as information returned from the server. (An Overview of HTTP N.d.)

HTTP allows for authentication within the request made by the client. A common form of authentication, "Basic", can be implemented by adding the Authorization -header to the HTTP request. An Authorization-header should contain the text "Basic" followed by a space, to which base64-encoded colon-joined username and password are appended. This form of authentication is not secure unless performed over a HTTPS connection. (HTTP Authentication N.d.)

Hypertext Transfer Protocol Secure (HTTPS) uses Transport Layer Security (TLS) protocol to encrypt the underlying HTTP protocol, allowing for transmission of sensitive data over the HTTP protocol securely. TLS utilizes a public-private-key encryption method to encrypt the data and relies on domain-specific certificates signed by external parties to deliver the public key required for data encryption, and simultaneously act as server authentication. (What is HTTPS? N.d.)

3.11.2 FTP and SFTP

File Transfer Protocol (FTP) can be used to efficiently transfer files across the internet in a client-server-based manner. FTP adds reliability to file transfers to and from a remote computer and circumvents potential file system incompatibilities between the source and destination machines. FTP connection is initialized by creating a control connection from the client to the server, in which commands and responses between the client and the server are transferred. Executing a command to transfer a file results in opening of a data connection, responsible for the process of transferring the file data, after which the data connection is closed. (FTP N.d.)

SSH File Transfer Protocol (SFTP) implements the traditional FTP functionality over Secure Socket Shell (SSH) protocol, which has a more robust set of security and authentication features. Essentially, SFTP is a secure version of FTP. (SFTP – SSH Secure File Transfer Protocol N.d.)

3.12 Data structures

3.12.1 XML

eXtensible Markup Language (XML) is a structure for data storage and transportation. An XML document consists of tags in a tree-like format, where only one tag may be at the highest (root) level of the document, and all remaining tags must be under another tag. A tag consists of tag name, attributes and a value – for example, <date type=”creation”>2020-08-20</date> would have a name of “date”, an attribute named “type” with a value of “creation”, and the value of “date” tag would be

“2020-08-20”. Definition of tag names and XML structure is left to the author of the document. (Introduction to XML N.d.)

As XML provides a structured syntax to store data, an XML document can be manipulated programmatically. As a result, XML is a popular choice for message format in integrations, as XML can be easily parsed and manipulated by integrations.

3.12.2 SOAP

Simple Object Access Protocol (SOAP) is an application communication protocol based on XML, used in sending and receiving messages with a more strictly defined structure than in XML. While structured like a regular XML message, SOAP has a stricter requirement on required elements and structure. A SOAP message is required to use SOAP Envelope namespace within the XML message and must contain the Envelope -element at the root of the message. A SOAP message must also contain a Body -element under the Envelope-element and may optionally contain Header and Fault elements under the Envelope-element. (XML Soap N.d.)

3.12.3 JSON

JavaScript Object Notation (JSON) is, like XML, a syntax for storing and transferring data. JSON syntax is based on JavaScript and operates on key-value pairs of information in a tree-like structure. (JSON Introduction N.d.)

Like XML, JSON datasets can be manipulated and transformed programmatically, and are often used to transform information in web-based applications.

3.13 Programming languages

3.13.1 Node.js

Node.js is a JavaScript runtime designed for building network applications, allowing for asynchronous and concurrent execution of code (About Node.js N.d.). Node.js is

used to run JavaScript code, and is among the runtime environment choices for AWS Lambda functions (Building Lambda functions with Node.js, N.d.).

3.13.2 ESQL

Extended Structured Query Language (ESQL) is a programming language specific to IBM's Integration Bus ESB. ESQL is based on the traditional Structured Query Language (SQL), but also includes additional features and properties useful in accessing and manipulating IBM's product-specific structures within messages in message flows. (ESQL Overview, N.d.)

4 Theoretical framework

4.1 Integration design

Integrating data and applications is by no means a new business. 1960s saw the first emergence of transaction processing systems, with database management systems following in 1970s and various decision support systems to support business processes emerging in 1980s. 1990s saw an ever increasing adoption of the Internet and globalization, driving the need to integrate multiple distinct systems. Enterprise Application Integration (EAI) as a widespread concept emerged in late 1990s to reduce programming complexity and costs of integration between systems and has been the target of extensive research and studies since its birth. (Lee et. al. 2003.)

The early years of EAI brought about two main design paradigms for integrations: point-to-point integration and hub-and-spoke integration. True to the name, point-to-point integrations consisted of applications communicating directly with each other, either synchronously or asynchronously through Message-Oriented Middleware (MOM) such as message queue brokers. An example architecture of point-to-point integrations is displayed below in Figure 1. (Gulledge 2006, 9-10; *What is integration?* N.d.)

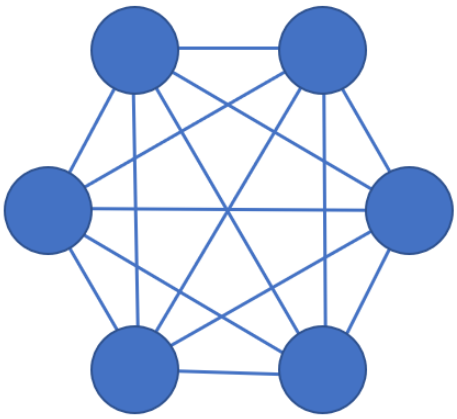

Figure 1: Point-to-point integration architecture

While point-to-point integration is maintainable in smaller environments, increasingly large number of applications would quickly overwhelm the number of required integrations. Any updates to infrastructure of member applications would also cause significant maintenance efforts. Hub-and-spoke design solves the complexity problem by introducing a central broker (the “hub”), which creates separate connections (“spokes”) to each application. A simple hub-and-spoke integration design is portrayed in Figure 2. (Gulledge 2006, 14; *What is integration?* N.d.)

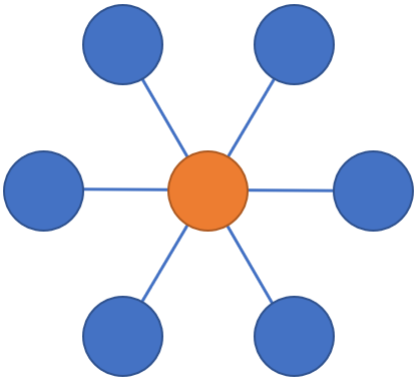

Figure 2: Hub-and-spoke integration design

On the other hand, introducing a central hub through which all connections are formed creates a single point of failure; should the hub be unavailable, none of the connections between applications would function. While the maintenance efforts are smaller in hub-and-spoke designs, the hub point still acts as a potential point of failure and a potential bottleneck in terms of resources and scalability. (*What is integration?* N.d.)

While ESB resembles EAI's hub-and-spoke design, SOA principles are utilized to create standardized services reusable for multiple integrations, further decreasing the complexity of hub-and-spoke based integration designs (*What is integration?* N.d.). Where each application in a hub-and-spoke design may contact the central broker in a unique format, ESB enforces specific standards to be used for connections, regardless of the connecting application. Enforcing specific standards allows ESB to use a limited number of services to connect with a larger number of applications, while utilizing additional containers for other functionalities, such as message routing or transformation. (Menge 2007.)

Menge (2007) illustrated a simple ESB overview with an unspecified number of containers, shown in Figure 3. Each ESB service container may contain multiple functionalities, though only a limited set are shown in the overview. Functionalities within the containers are modular, and therefore reusable by multiple applications; for example, two different applications connecting to the application in Menge's ESB overview could likely both utilize the same Application Adapter, even if both connecting applications had undergone different logic through ESB. In a similar manner, a common application could be used to transform messages between multiple different systems, if the logic for message transformation is common between all messages.

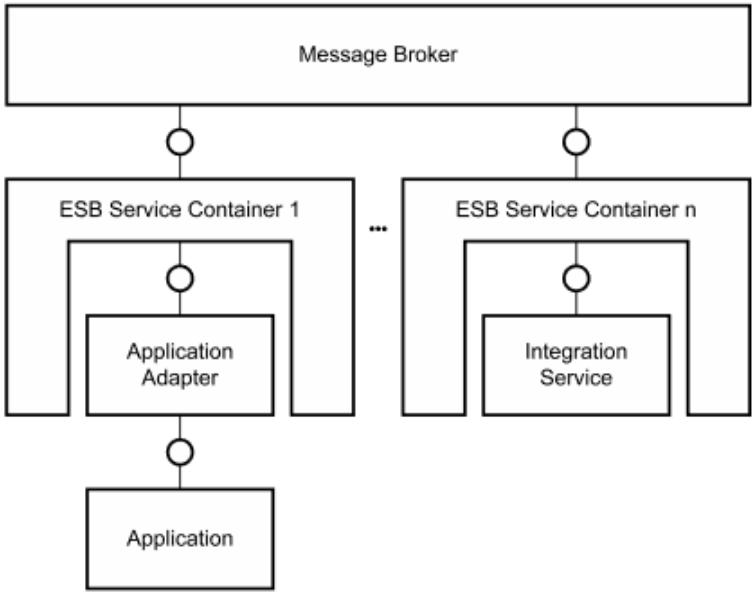

Figure 3: Simplistic ESB overview

The focus of this research is to provide SOA based approach to integrate systems in AWS environment to third party applications. While the theoretical basis for ESB implementation architectures is broad, AWS is such a new environment that best practice implementations and theories have not yet been crafted for integration purposes. In this research, lessons from ESB architecture and design in traditional environments are transferred to the implementation within AWS environment where possible.

4.2 Reliable messaging

4.2.1 Using reliable messaging to guide implementation

The concept of reliable messaging was introduced in Chapter 3.4, defined by Brown (2001) to include requirements of authentication of sender and receiver, traceability of messages, exactly once transfer of messages, maintained message order and failure notification when unable to deliver a message. The specific method of fulfilling each requirement is left to the implementer and will likely vary across different

environments. Theoretical basis for implementation of each requirement is examined based on requirements set for this research and may require significant re-examination when applied to a different setting.

4.2.2 Sender and recipient authentication

Authentication entails verification of identity to ensure that the user or service being authenticated is who they claim to be (*Authentication and Authorization* N.d.). In order to fulfill the authentication requirement of reliable messaging principles, both the client and the server should authenticate each other in order to ensure that messages are transferred between the intended parties. Available authentication methods vary across different services, but in general, clients are often authenticated using credential checks such as password challenges or security tokens, while servers are usually authenticated through certificates.

Authentication and authorization are implied to be synonymous in the requirement of sender and recipient authentication. Authentication concerns the identity of the service, while authorization focuses on the permissions which the authenticated service has access after a successful authentication (*Authentication and Authorization* N.d.). In order to keep the scope of this thesis limited, authenticated services are also assumed to be authorized, and no separate permission schemes are implemented. In practice, especially on production environments, authorization schemes should be implemented to ensure system security. Preferably, principles of Least Privilege should be followed in order to provide as little access as required to perform the required tasks, in order to limit potential damage from accidents, errors and abuse (Saltzer & Schroeder 1975, 1282).

Web interfaces support a multitude of authentication methods, including username-password combinations, cookie-based authentication, tokens and signatures. Of these, the simplest method to implement is HTTP Basic Authentication. Basic Authentication involves providing a base64-encoded username and a password within the HTTP request headers, and is used for the purpose of this research where applicable. HTTP Basic Authentication itself does not provide any encryption, but instead

relies on sufficiently well encrypted connections to keep the credentials secure. (Nemeth 2015.)

As the focus of the research is the overall architecture of the implemented integration solution, the authentication methods utilized in this research have been chosen based on ease of implementation, rather than security. If HTTP Basic Authentication is used in production environments, the surrounding environment should be properly configured to use HTTPS connections, or another authentication method should be considered instead. Some environments may also benefit from using Lightweight Directory Access Protocol (LDAP) queries for centralized authentication and authorization checks.

Recipient authentication can be implemented by creating SSL certificates for the recipient services and using HTTPS connections to securely connect to the target recipients. SSL certificate contains the recipient's domain name, validity time, issuer information and public key to allow for connection encryption between the recipient and the sender. The sending party can verify the authenticity of the recipient's SSL certificate with the certificate issuer, and if the check fails or cannot be performed, the recipient can be deemed untrusted. (*What is an SSL Certificate?* N.d.)

Recipient authentication has not been implemented for this research due to the additional complexity of setup, requiring introduction of signed certificates for server authentication. In production settings, recipient authentication should be performed with SSL certificates or other possible authentication schemes in order to ensure secure transfer of data between correct endpoints.

4.2.3 Message traceability

Message traceability implies that the movements and actions of each message received by the ESB can be followed through the integration, from the moment of arrival to the integration to the moment of delivery to a target application. Tracing, as a process, is usually used for debug and performance analysis purposes, and usually

entails the recording and storage of runtime events for later analysis (Kraft, Wall & Kienle 2010).

The frameworks and methodologies for implementing traces vary across applications and need to be implemented on a case-by-case basis. As this research focuses on using AWS and IBM's App Connect Enterprise, message tracing is implemented for methodologies available within AWS and IBM's platforms. The central ideas for tracing follow some of Kraft and colleagues' (2010) lessons from their documented implementation, regardless of the framework used.

The main concepts to be traced by Kraft and colleagues (2010) included the identity of the process being traced, timestamp for the trace as well as the reason for the trace. The main considerations revolved around overhead, system resources and efficient tracing, likely due to the highly resource-restricted environment in which the experiments were performed. Despite resource overhead concerns, permanent tracing was recommended. Integrated tracing within software would be a permanent and tested fixture of the program, more likely to be maintained even after updates to the software. Permanent tracing would also produce on-demand diagnostics and logs, allowing for developers to rely on these logs as a source of diagnostic data. Additionally, tracing overhead costs were minimal when compared to benefits reaped from traces. Storage of traces was recommended to be on fixed-size ring-buffer on a pre-initialized data structure. Traces should also be simple and devoid of logic where possible. (Kraft, Wall & Kienle 2010.)

In contrast to Kraft and colleagues' environment, the infrastructure used for this research is not particularly resource-limited, even though higher efficiency and less overhead can be argued to be preferable. As resources are not considered limited, implementation of tracing will focus on the functional lessons from Kraft and colleagues and overlook the resource-optimization side of implementation. Trace logs will include the "what", "when" and "why" of the integration: integration and message identity, timestamp and status (or error) information. An ideal implementation of trace logs would follow Kraft and colleagues' suggestion of fixed-size log files but tap into the larger amount of resources available and implement a set of rotating log

files. New log files would be created after a set time period, or when the old log files reach a certain size, providing a longer history of diagnostic data. Logs are by default split into multiple files on AWS platform with CloudWatch logs, while file-level logs for IBM's ESB would require additional configuration. Configuration of split logs for ESB has been left out of scope for this research. An ideal solution would also unify all log files to be centrally available from a single source, with unified or highly similar structure to allow for easier diagnosis. Utilizing logging solutions such as Splunk or ELK to collectively store traces from multiple locations would provide a single platform from which tracing information could be examined for debugging or analysis purposes. For the purpose of this research, the availability of logs in two separate locations (AWS and ESB) is deemed adequate for the purpose.

4.2.4 Exactly once receipt of messages

The concept of exactly once receipt of messages strongly relates to integrity of data in the receiving system, as any missing messages or duplicate messages may lead to a different set of data present in the recipient system compared to the sender system. The exact consequences of duplicates or missing messages vary based on the recipient system type. Some systems, such as applications hosting financial transactions or orders, require exactly once delivery by their nature in order to avoid compromising data integrity. More informative systems, such as applications handling the most recent stock price values, would not be completely compromised by an undelivered message. (*Messaging Concepts* N.d.)

Additional safeguards for exactly once receipt of messages include durability of messages and transactions. Durability of messages implies that the messages will persist in a permanent storage through system shutdown, even if the shutdown occurred during message processing (ibid.). In practice, durability requires messages to be stored within persistent data structures such as message queues, in contrast to non-durable Remote Procedure Calls (RPCs), where the message is processed synchronously without intermediate storage of the message contents.

Transactional processing means that the processing cannot be partially successful; if a single part of the processing fails, the successfully processed parts within the transaction will be rolled back as well, with changes to all systems committed only if all actions were successful (*Transaction Processing* 2017). In order to ensure exactly once receipt of messages, the integration solution must be implemented by accounting for the principles of transactional processing.

4.2.5 Maintained message delivery order

Maintained message delivery order means that the messages sent directly between a specific sender and recipient will always be received in the same order in which they were sent (*Message Delivery Reliability* N.d.). This is especially critical for systems relying on the correct order of events, as a wrong order of events may result in a completely different outcome compared to the correct order of events. A practical way for maintaining message delivery order is to process messages in a First-In-First-Out (FIFO) basis, where the first messages to arrive on the message queue are the first ones to be processed (*Priority* N.d.; *Amazon SQS FIFO (First-In-First-Out) queues* N.d.). Both AWS and IBM provide message queues with FIFO-capability and are used for the purpose of this research to ensure maintained order of messages.

4.2.6 Delivery failure notification to all parties

Delivery failure notification, especially to the sending party, is important in maintaining the exactly once delivery principle in the event of a processing failure of the message. Without a success or failure acknowledgment, the sending party has no way of knowing whether the message has been processed successfully. Failure notification is crucial for determining if the message needs to be resent due to temporary or permanent delivery failures. Temporary delivery failures include issues which resolve without changes to the message, such as network connection issues. Permanent delivery failures refer to problems within the message, which make the message undeliverable, such as syntax errors. The sending party needs to be informed of a temporary failure in order to retry sending the message or transfer the message to a Dead Letter Queue in the case of a permanent failure. (Mitchell 2017.)

The purpose for a delivery notification failure for the recipient is debatable, as the receiving system is usually either technically unable to receive failure notifications, or unable to act based on the notifications. As an example, an FTP server would not have a method to receive information that the processing of a message targeted towards the FTP server failed, nor could the FTP server affect the processing of the message in any manner. Similarly, should the delivery failure occur due to inability to contact the FTP server, notifying the server of the failure would also likely end up in a failure. Therefore, for the purpose of this research, failure notifications for the recipient are left out of scope. Some scenarios may include valid reasons for additional notifications to the recipient in the event of processing failures, in which case the methodology for handling delivery notifications should be planned based on the exact circumstances within the environment.

5 Implementation

5.1 Implementation overview

While design-based research allows for iteration throughout the research process, the overall architecture of the implementation remained nearly unchanged throughout the implementation phase. The original high-level architectural design proved suitable for the solution implementation, and the main changes and tweaks to the implementation occurred within individual components. The final architecture consists of two main environments: AWS and the internal network of the research target, separated by public internet. Figure 4 displays the high level architecture of the implemented solution.

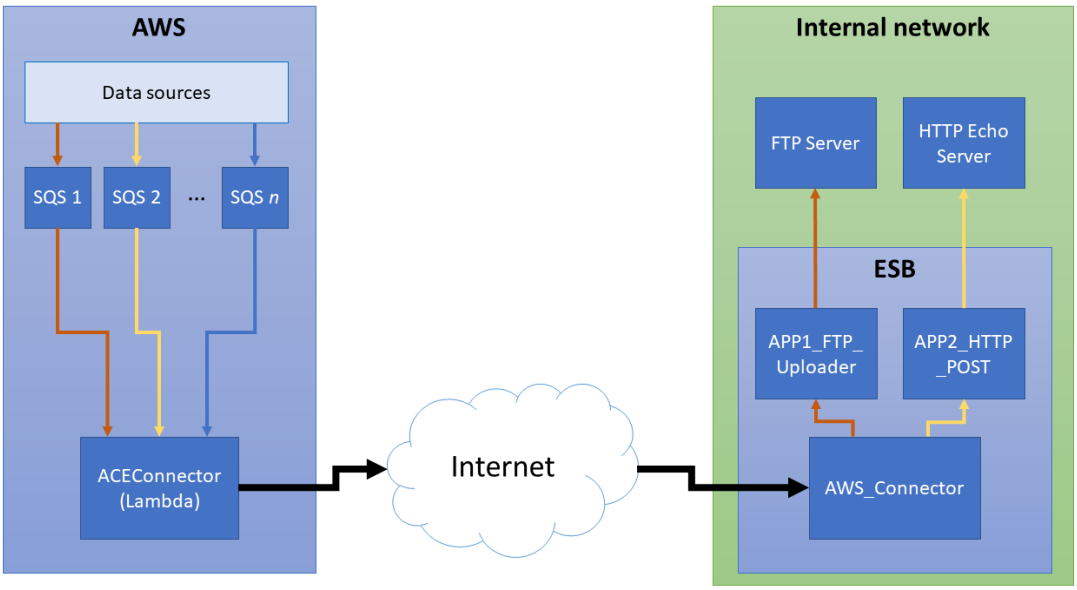

Figure 4: Implementation architecture

The AWS environment contains two SQS queues to serve as data pickup points. In a real-world scenario, applications would deliver messages to these SQS queues. The number of SQS queues to be used as source points can be extended at will with no changes to code. The SQS queues are set as triggers for a single Lambda function, ACEConnector. ACEConnector is responsible for reading the messages from the source queues, transmitting the messages across the internet to ESB and deleting the messages from the source queues if the transfer was successful.

The internal network of the research target contains an ESB implementation consisting of IBM's WebSphere MQ message queuing platform and IBM's App Connect Enterprise integration broker, as well as target applications simulating receiving applications. AWS_Connector is an application within the ESB, responsible for receiving data from AWS. AWS_Connector exposes a HTTP interface to the public internet, capable of receiving HTTP POST requests. AWS_Connector application receives and distributes the messages sent by AWS's Lambda internally to other ESB applications by determining the correct route from AWS source queue names. After distribution, each intermediate ESB application handles the messages based on internal logic of

the application, before transmitting the messages to receiving systems. For the purpose of this research, receiving services were simulated with one fully functional FTP-server and a HTTP echo server, but could in production scenarios include specialized systems such as ERPs, data warehouses or Customer Relationship Management (CRM) systems.

5.2 Amazon Web Services service implementation

5.2.1 Message queues as message sources

As message queues are available as a service in AWS platform, and are purpose-built for transmitting messages, message queues in the form of SQS queues were chosen as an integral part of the integration implementation. At the time of the implementation, AWS provided two alternatives for SQS queues: Standard Queues and FIFO queues. Standard queues produce a higher throughput of messages but may deliver messages out of order or more than once (*Amazon SQS Standard queues* N.d.). FIFO-queues, on the other hand, ensure correct message delivery order and exactly once delivery of messages (*Amazon SQS FIFO (First-In-First-Out) queues* N.d.). In cases where throughput is more important than exact delivery order or redundancy in deliveries, standard queues are a reasonable choice, but do not provide the properties required by reliable messaging principles. For this reason, FIFO queues were chosen as the queue type for the implementation.

During the first iteration of the implementation, only one queue was created to host data to be received from multiple applications in order to keep the design simple. The initial solution to integrating AWS-based services would have relied on the sending applications to provide identifying information, based on which the messages would be distributed by ESB to different target applications. During further iterations of the integration solution, the responsibility of message routing was transferred from the sending application to the integration, with the target system determined based on which message queue originally received the message. As a result, the number of queues increased from one input queue common for all applications to one input queue for each application. Splitting the data input to individual queues

also provided the additional benefit of being able to easily route data from individual applications to a different handler for development or debug purposes.

When a message is read from SQS queue, the message is not deleted from the queue, but is rather given a visibility timeout, during which the same message cannot be read from the queue again. Visibility timeout prevents the message reader (also known as the message consumer) from accidentally processing the message twice and allows a message to be returned to the queue successfully in the case of a failure during message processing. The message is removed from the queue only after being specifically deleted by the consumer or after a successful processing confirmation. (*Amazon SQS visibility timeout* N.d.)

The consumer of a message may also return an error, in which case configured queue parameters can determine the course of action for the failed messages. Some messages may be undeliverable due to unresolvable issues, such as syntax errors, in which case removal from the queue is justified. For SQS queues, a dead-letter queue of the matching type can be configured, to which undelivered messages can be pushed after an adjustable number of retries. (*Amazon SQS dead-letter queues* N.d.)

Dead-letter queues are utilized in the implementation to ensure persistency and prevent loss of data in case of failures by storing permanently failing messages separately from the actual input message queue. Messages in dead letter queues can be later examined by developers to determine causes of failure or moved back to the application queue to attempt resending of data. For each application queue in AWS, a matching dead-letter queue was created, and the application queue was configured to transfer the message to the dead-letter queue after ten failed processing attempts. The specific retry numbers are configurable per application basis and can be adjusted as necessary.

Configuration details utilized in the final version of the implementation are described in Table 1. Queue names are unique for each application, named in a manner descriptive of the sending application. Dead-letter queue names combined the application queue name with a “_Fail” suffix. A sufficiently informative naming scenario

should be used in a production scenario. Visibility timeout should be set to a long enough value to ensure a response has successfully been received from the message consumer. If the message can be expected to traverse through multiple calls over the network, the visibility timeout on the SQS queue may need to be set to multiple minutes to ensure that the message consumer does not return a response message after the visibility timeout of the original message has expired. Queue permissions need to be set either during queue creation or message consumer configuration to allow consumer to read messages from the queue. In this research, the configuration was conducted during message consumer implementation, removing the need to specify permissions with each queue creation. Dead-letter queue settings were configured only for the main queues responsible for providing messages for transfer. Dead-letter queues themselves were created in the same manner as main application queues, but without any dead-letter queue configurations.

Table 1: SQS queue configurations

Property name	**Property value**
Type	FIFO
Name	MockDataSource1.fifo
Visibility timeout	200 Seconds
Message retention period	4 Days
Delivery delay	0 Seconds
Maximum message size	256 KB
Receive message wait time	0 Seconds
Content-based deduplication	Not selected
Access policy	Basic
Define who can send messages to the queue	Only the queue owner
Define who can receive messages from the queue	Only the queue owner
Dead-letter queue options (only for application queues):	
Set this queue to receive undeliverable messages	Enabled
Choose queue	Choose a dead-letter queue
Queue ARN	<redacted>:MockDataSource1_Fail.fifo
Maximum receives	10

A total of four SQS queues were created for the final version of the implementation: two queues for each application, of which one was the main queue responsible for holding messages to be transferred to ESB, and the other was a dead-letter queue for storing messages that repeatedly failed to be processed. The complete set of SQS queues for the implementation is listed below in Table 2.

Table 2: Implemented SQS queues

Queue name	**Description**
MockDataSource1.fifo	Main queue for Application 1
MockDataSource1_Fail.fifo	Dead-letter queue for Application 1
MockDataSource2.fifo	Main queue for Application 2
MockDataSource2_Fail.fifo	Dead-letter queue for Application 2

5.2.2 Lambda function as the message consumer

While technically a large variety of services could be utilized to act as message consumers to transfer the messages from SQS queues, the simplest method of consuming messages was to utilize services offered within AWS environment. Being able to consume and delete messages on SQS queues requires AWS credentials and the SQS queues need to be configured to permit modification by the AWS identity used by the consumer (*Identity and access management in Amazon SQS* N.d.). In practice, using a third-party component as a consumer of messages would require maintaining a set of up-to-date AWS credentials on the consumer component, whereas relying on AWS-native services to eliminates the need for identity management other than permission delegations.

Out of the services available in AWS, Lambda provided the simplest and the most flexible way to consume and handle SQS messages. Lambda functions allow for execution of custom code in a serverless environment, providing a lightweight maintenance-free platform to create the required functionality to send messages from AWS to ESB. Additionally, Lambda functions can use SQS queues as trigger sources, allowing for the code within a Lambda function to be executed as soon as a message is

detected within the SQS queue. Lambda functions provide native support for Java, Go, PowerShell, Node.js, C#, Python and Ruby, with additional support for other programming languages through a Runtime API (*AWS Lambda FAQs* N.d.). Node.js was chosen for the implementation due to performance reasons and native HTTP request capabilities.

The final implementation of the integration contains one Lambda function: ACEConnector. ACEConnector utilizes the main SQS queues of applications, MockDataSource1.fifo and MockDataSource2.fifo, as trigger sources for code execution. Table 3 contains the parameters used to initialize the Lambda function.

Table 3: Lambda function configuration values

Property name	**Property value**
Choice of function base	Author from scratch
Function name	ACEConnector
Runtime	Node.js 1.2.x

Creation of the Lambda function simultaneously created a new identity role within AWS, to which a permission policy could be created to allow SQS access and modification rights. SQS access permissions were configured by locating ACEConnector's execution role in AWS Identity and Access Management portal and attaching a new policy to the role with configurations specified in Table 4. The configured policy allows the Lambda function to read SQS metadata information, read messages from SQS queues and delete messages from SQS queues, if the SQS queue is within the same AWS account as the Lambda function. Lambda functions are also technically capable of accessing SQS queues on other AWS accounts within the same geographic region. External account queue accesses would require the use of Simple Token Service within Lambda code and additional permission configurations on the SQS queue. In order to avoid complexity, only SQS queues in the same account were configured to be used as data sources.

Table 4: SQS access policy for ACEConnector

Property name	**Property value**
Service	SQS
Actions	Read: GetQueueAttributes
	Read: ReceiveMessage
	Write: DeleteMessage
Resources	arn:aws:sqs:*:*:*
Name	ACEConnectorSQSAccess

After permission configurations, SQS queues were added as triggers through the Lambda interface for ACEConnector. As the queues had already been created and permission policies attached to ACEConnector's execution role, the required configurations were minimal, shown in Table 5.

Table 5: Lambda trigger configurations

Property name	**Property value**
Trigger source selection	SQS
SQS queue	<redacted>:MockDataSource1.fifo
Batch size	10
Enable trigger	Selected

The source code for ACEConnector was the single portion of AWS side implementation which experienced the most iterations during the process of implementation, both expanding and simplifying the functionalities of the code. The initial versions of the code simply aimed at transferring the message to ESB through a HTTP POST request, interpreting the operation as successful if ESB confirmed that the messages had been received, regardless of processing outcome. The simultaneous development of the receiving application at ESB side also provided more conditions to account for within the Lambda code, such as specific HTTP response codes based on the processing result within the connector at ESB side. The later versions of the code implemented reliable messaging features to ensure delivery and failure notification,

allowing for processing retries and dead-letter queue handling procedures on AWS side in case of permanent failures.

The final version of the implemented code in ACEConnector is available in Appendix 1 and is described in this paragraph briefly. The code implementation is based on the work of Hamza Sabljakovic (2019), utilizing parts of his code while modifying and adding on to others. ACEConnector takes in event data from SQS, containing metadata on the SQS queue itself, as well as the messages retrieved from the source SQS queue. The code in ACEConnector starts by defining the configurations required to perform a HTTP call: target host IP address, port number, URL path, HTTP method and Authorization header for authentication. A request object is created with the previously set up configurations and a callback function to verify the HTTP status code received from ESB once the call has been conducted. The event data retrieved from the SQS queue is added as the HTTP request body, and the HTTP POST request is performed. For the purpose of this implementation, the request is considered failed in situations where the request outright returns an error, or the returned HTTP status code is anything other than 200. If the messages were delivered successfully, the status code is logged to console and execution finished, triggering the deletion of the messages from the SQS queue. If errors were encountered during processing, an exception is thrown from ACEConnector with the encountered error details. The exception provides the SQS queue with information that the messages were processed unsuccessfully, either prompting the SQS queue to move the messages to dead-letter queue if the maximum retry count was reached, or allowing the messages to remain in the queue to be reprocessed once the visibility timeout has expired. Failure feedback to the originating SQS queue enforces reliable messaging principles, as the source of the message will receive notification of delivery failure and no messages are lost even in cases of delivery or processing failures.

5.2.3 Logging with CloudWatch

Fulfilling the traceability requirement of reliable messaging principles required the implementation of logging features on AWS. Fortunately, most of the work was already completed by AWS under the hood of the services. AWS automatically creates

logs from available services, including SQS queues and Lambda functions. CloudWatch logs include both visual dashboards and text-based logs for various indicators. Figure 5 shows an example view from CloudWatch on SQS sent message statistics.

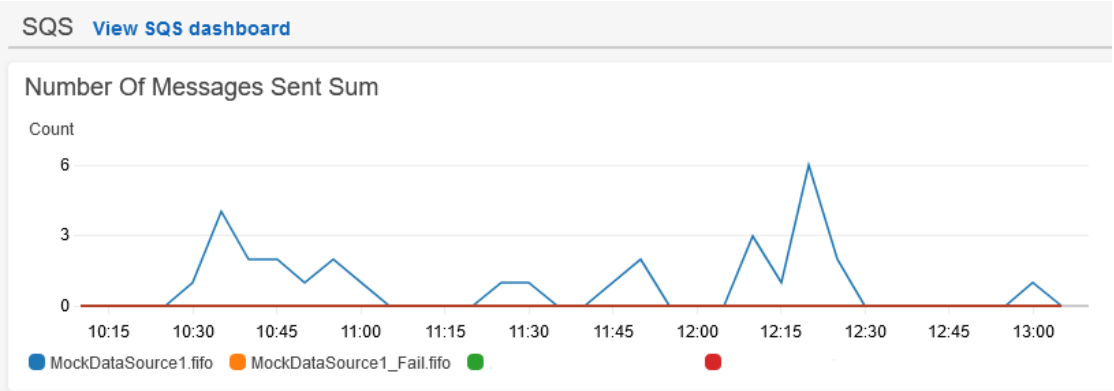

Figure 5: SQS sent messages graph on CloudWatch dashboard

Text-based logs of CloudWatch include the main aspects defined by Kraft and colleagues (2010) in their lessons on implementation of tracing and logs; log entries include the timestamp for the event, message identifier, information type and specifying information, shown in Figure 6. Any information written to console during Lambda execution is logged to CloudWatch logs, allowing for custom logging events in code, such as the status code logs in Figure 6.

Figure 6: Text-based logs on CloudWatch

5.2.4 Test data population with helper Lambda

A separate Lambda function, SQSMessageGenerator, was used to populate data to SQS queues and verify the functionality of the integration. A predetermined number of messages were generated and filled to each queue for integration processing. After the processing was complete, the receiving system logs and integration traces were reviewed to verify that the messages were distributed correctly by the integration. In order to keep the scope of the research within reasonable limits, the specific testing procedures and results are not reported in this research.

SQSMessageGenerator is a Lambda function based on Node.js runtime. SQSMessageGenerator aims to emulate sender applications by creating a set of messages and sending the messages to different SQS queues, from which the messages would be picked up by ACEConnector. The specifics of the code can be tweaked to change the total number of messages created, target queues and the relative distribution of messages for each queue. As SQSMessageGenerator is not a direct part of the integration solution, but rather a utility used to test the implementation, the specific functionality of the code is not discussed here. The source code for SQSMessageGenerator is provided in Appendix 2 and is adaptable to various testing scenarios with minor tweaks to the code. The code is adapted from the example provided by AWS in their developer guide (*Sending and Receiving Messages in Amazon SQS* N.d.).

5.2.5 Final AWS architecture

Even though a rough plan for AWS environment implementation existed prior to configuring the first functionalities within AWS, the detailed architectural implementation took shape only after the entire integration implementation had reached the final form. Figure 7 shows the architecture of AWS components in the final implementation, including the helper lambda function for test data population, which in production use case would be replaced by separate sender applications.

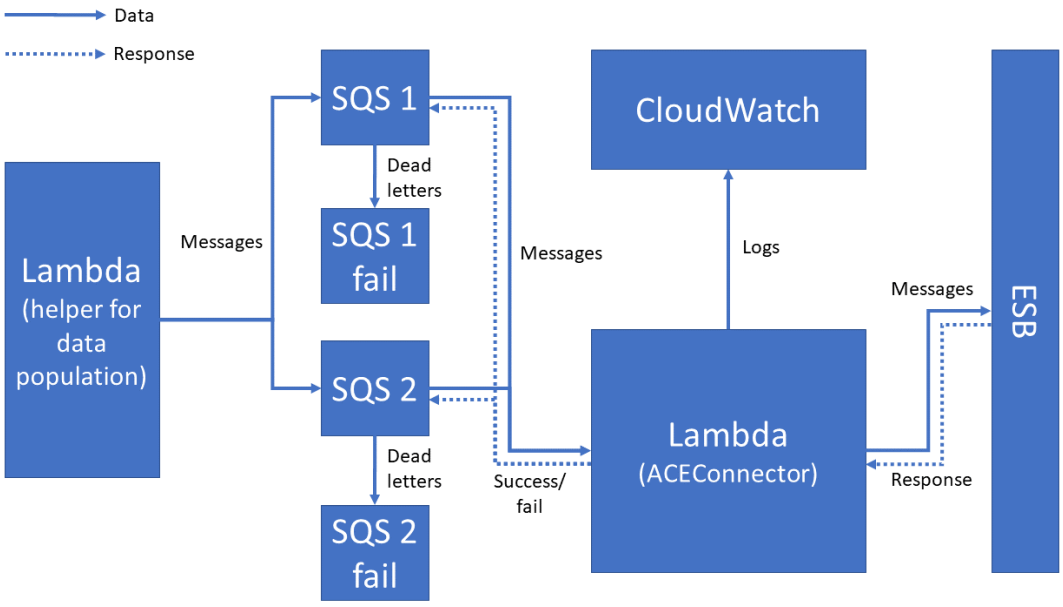

Figure 7: AWS environment integration architecture

The solid arrows on Figure 7 show the flow of messages from the data source to ESB and CloudWatch logs. The dotted arrows indicate the flow of response information from ESB back to ACEConnector, and from ACEConnector to individual SQS queues acting as data sources. On the event of multiple failures and maximum retry caps reached, message data will be transported to dead-letter queues, at which point a developer or a support personnel should begin investigations on the failure mechanism and required remedying actions.

5.3 Notes on transit across the public internet

As the implemented integration is a proof-of-concept solution in a development environment and not intended to be transferred to production as is, the architectural design on transit of messages across the internet was kept simple. ACEConnector performs a HTTP POST request to a public IP address, owned by a router within the target network. Port forward rules were configured in the router to transfer the request directly to the integration server's port 8081, the HTTP listener for the AWS connector. SSL certificates and HTTPS connectivity were not implemented for the

development environment but should be considered a bare minimum standard for production environments. Proper access routing through demilitarized network zones should also be considered for production use cases.

5.4 Enterprise Service Bus implementation

5.4.1 ESB implementation overview

The implementation of integration on ESB was conducted by delegating responsibilities for individual functionalities to separate applications, in accordance with SOA principles. The division of responsibilities resulted in one AWS Connector application responsible for distributing incoming messages sent from AWS, and individual applications responsible for subsequently processing and delivering messages to target systems. The system-specific modules could be further divided into transformation and connector modules to better adhere to SOA principles, which would provide more reusability especially in larger and more complex environments. For the purpose of this research, the message-processing applications were left as a single module to avoid complexity.

5.4.2 AWS_Connector application

Like with ACEConnector implementation, the final form of AWS_Connector is the product of multiple stages of evolution. The development started with the most basic required functionalities, consisting of successful receipt of a message, delivery to target queue and sending of a HTTP response to the caller. Later implementations saw the addition of batch-processing of messages and conversion of routing information from hardcoded lists within code to configuration files. Error-handling and logging provided the necessary functionality to ensure message traceability and delivery failure notifications, as outlined by reliable messaging guidelines.

AWS_Connector application contains all functionality related to message transfers to and from AWS environment. While an application may contain multiple message flows, only HTTP_Inbound message flow has been implemented in this research. If

messages were to be transferred from ESB towards AWS, an additional HTTP_Outbound message flow would be implemented within AWS_Connector application.

HTTP_Inbound message flow is responsible for providing a public HTTP interface capable of receiving messages through HTTP POST requests, routing the message to the correct application within ESB for further processing and sending a reply to the creator of the HTTP POST request. Figure 8 shows the message flow order and nodes for HTTP_Inbound, with yellow paths representing actions for unsuccessful processing attempts and the green path representing the actions in a successful processing case. Each node along the paths in Figure 8 represent a functionality or an action within the message flow, such as output of data to a trace file, addition of a HTTP-header to a message or code execution.

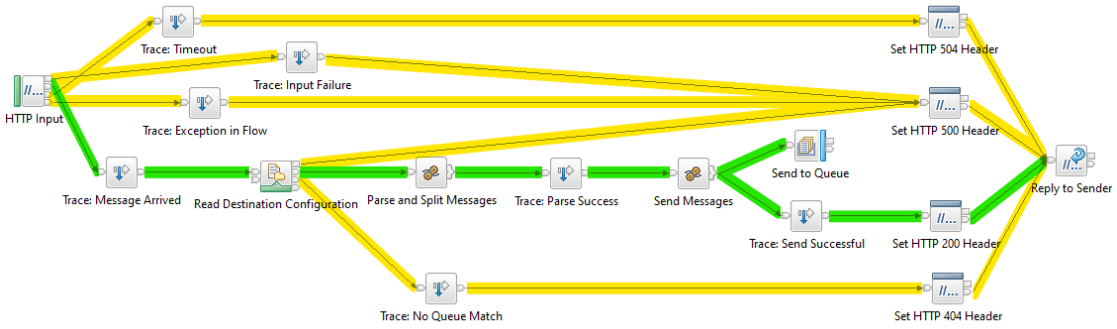

Figure 8: HTTP_Inbound flow order and success/failure paths

HTTP Input node was used to create an interface capable of receiving HTTP POST requests. HTTP Input is a standard App Connect Enterprise (ACE) HTTPInput node, configured as outlined in Table 6. Path suffix for URL set to “/aws/in” specifies the exact address at which the server should listen for the connections, such as “http://server:7800/aws/in”. As the event data received by ACEConnector Lambda is by default JSON and sent to the HTTP Input node without any modifications, message domain was set to JSON to allow ACE to parse and process the incoming message.

Table 6: HTTP Input configurations

Property name	**Property value**
Path suffix for URL	/aws/in
Message domain	JSON: For JavaScript Object Notation messages
Maximum client wait time (sec)	180
Security profile	Default Propagation
Identity token type	Transport Default

The security profile and identity token type configurations relate to enabling authentication capabilities for the HTTP Input node, fulfilling the sender authentication requirement of reliable messaging principles. Enabling basic authentication functionalities in ACE requires a policy to be created and deployed on the same integration server as the application. For this purpose, HTTPInputSecurityPolicy was created within a policy group, AWSConnectorPolicies, specific configurations in Table 7.

Table 7: Security policy configuration

Property name	**Property value**
Name	HTTPInputSecurityPolicy
Type	Security Profiles
Templates	Security Profiles
Authentication	Local
Authentication configuration	AWSConnectorInbound

The final addition to enable basic authentication on the HTTP input node involved creating a vault entry for credential named “AWSConnectorInbound” with the integration broker’s command line interface. The broker’s key vault is a central place to securely store a complete list of authentication parameters used by the integration nodes, such as FTP or HTTP functionalities. Key vault and vault entry were created through IBM App Connect Enterprise Console with the following commands:

```
mqsivault --work-dir D:\Users\ESB\IBM\ACET11\workspace\INTSRV01 --create --
vault-key <secret_key>
```

```
mqsicredentials --work-dir D:\Users\ESB\IBM\ACET11\workspace\INTSRV01 --
create --vault-key <secret_key> --credential-type local --credential-name
AWSConnectorInbound --username AWS --password <password>
```

The commands and configurations above tied the authentication of the HTTP interface to AWSConnectorInbound credentials, accessible by providing the username "AWS" and the password set during the credential creation. The same username and password were used in ACEConnector Lambda to create the Authorization header for the HTTP POST request. During an inbound message to the HTTP Input node, the Authorization header is read from the HTTP POST request. The authorization method and credentials are parsed and compared to the identity within the key vault, and if the credentials match, the message is read into the flow. In case of wrong credentials, a "401 Unauthorized" response code is sent back to the calling party, prompting ACEConnector to throw an exception within the Lambda code, returning the messages back to the SQS queues.

HTTP Input node offers four distinct outcomes for processing: timeout, input failure, successful receipt and a catch-terminal for handling exceptions during processing. Terminals related to failure (timeout, input failure and exception) are connected to Trace nodes responsible for updating log files with information on message handling progress. After logging, a X-Original-HTTP-Status-Code header with relevant status code is added to HTTPReply properties of the message by a HTTP Header node. Finally, the message is transferred to a HTTP Reply node in order to trigger a HTTP response message for the sender of the HTTP request.

For successful message reads, additional actions related to message routing and propagation are performed before the final HTTP Reply node. After the successful receipt of the message has been logged by a Trace Node, a configuration file containing queue routing information is read and analyzed by using a File Read node. The specific configurations for File Read, named Read Destination Configuration on Figure 8, are listed in Table 8.

Table 8: Read Destination Configuration properties

Property name	**Property value**
Input directory	D:\Users\ESB\IBM\ACET11\workspace\FlowConfig
File name or pattern	AWS_Connector_destinations.json
Result data location	$ResultRoot
Output data location	$Environment/Variables/filecontent
Copy local environment	Selected
Record selection expression	$InputRoot/JSON/Data/Records/Item/eventSourceARN = $Environment/Variables/filecontent/JSON/Data/QueueList/Item/OriginQueue
Message domain	JSON: For JavaScript Object Notation messages

The configuration file read process reads the contents of AWS_Connector_destinations.json to memory, and attempts to match the value of OriginQueue in AWS_Connector_destinations.json to eventSourceARN in the incoming message structure. The implementation produced during the research contained only two routing entries, indicating the origin queue and the destination message queue within ESB to which the message should be routed:

```
{
          "QueueList": [
          {
                    "OriginQueue" : "arn:aws:sqs:eu-north-
1:737528452624:MockDataSource1.fifo",
                    "Destination" : "FTP.UPLOADER.IN"
          },
          {
                    "OriginQueue" : "arn:aws:sqs:eu-north-
1:737528452624:MockDataSource2.fifo",
                    "Destination" : "HTTP.UPLOADER.IN"
          }]
}
```

If no match for the eventSourceARN contained within the incoming message is found, the message cannot be routed to an application within the ESB. In these scenarios, the routing failure is logged, a HTTP header is added to the message and a reply of an unsuccessful processing event is sent back to caller.

A successful routing match is followed by extracting the actual messages from their metadata-wrapper with an ESQL Compute node, source code available in Appendix 3. As Figure 9 shows, the actual data to be transferred within the message is embedded within the message body as a string in JSON format.

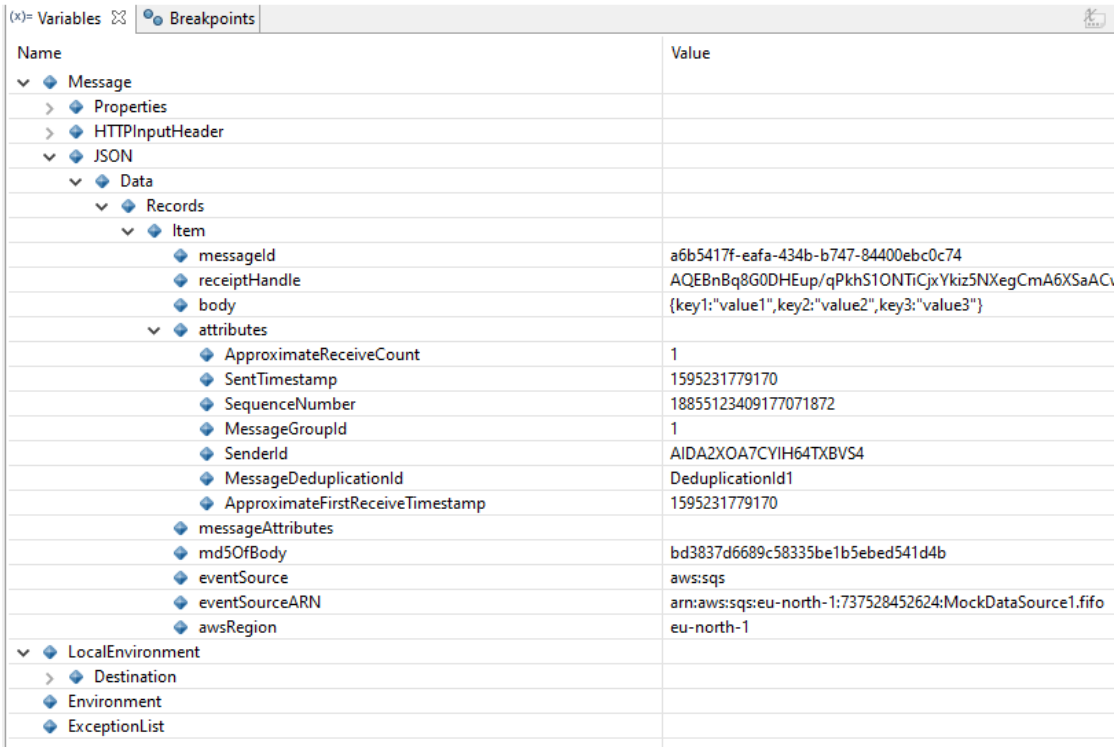

Figure 9: Incoming message structure in ACE debugger

Parse and Split Messages compute node creates a staging area in memory and processes all Item tags in the incoming message separately. Each Item tag is a separate message originating from the SQS queues, and for each Item, a destination queue name is determined by matching the eventSourceARN to the OriginQueue in the configuration file. After the destination queue name has been determined, the JSON-string from the message body is cast as a binary string to a temporary storage variable, from where the string is parsed from JSON to XMLNSC to create a structured set of data within the staging area in memory. After all messages have been handled, the count of parsed messages and source queue information are stored for use by the following Trace nodes and the message flow proceeds to send messages.

Send Messages node sets the correct target WebSphere MQ message queue name for each message staged in the memory and sends the messages to the applications responsible for further processing. The source code for Send Messages compute node is available in Appendix 4. The code loops through all messages stored in the staging area in memory. For each message, the code sets the destination queue to the queue obtained from the configuration file. A unique message identifier is then created for internal logging purposes. MQRFH2-headers are populated with message-related metadata and the message is sent to the secondary terminal of the flow, leading to a MQ Output node. MQ Output node then delivers the message to the previously determined message queue. After all staged messages have been processed, a final message is sent to the primary output terminal of the compute node for logging, HTTP response code addition and a HTTP response creation for the caller. The compute node compute mode property was set to "LocalEnvironment and Message" to allow passing of the destination queue name to the MQ Output node.

The use of WebSphere MQ message queues required the configuration of a MQ Endpoint policy. The policy was created within AWSConnectorPolicies policy group, alongside previously covered HTTPInputSecurityPolicy. The specific configurations for the policy are listed in Table 9. The values for queue manager name, host name port number and channel name will vary based on implementation and setup of WebSphere MQ server.

Table 9: MQEndpointPolicy configuration

Property name	**Property value**
Name	MQEndpointPolicy
Type	MQEndpoint
Template	MQEndpoint
Connection	CLIENT
Queue manager name	QM1
Queue manager host name	127.0.0.1
Listener port number	1414
Channel name	qm1
Use SSL	false

Send to Queue node requires a destination list configuration to allow dynamic selection of the target message queue. Specific configurations for the MQ Output node are listed in Table 10 below. As the queue name is not specified and destination mode is set to destination list, the queue name is read in from the memory instead of a statically set value in the node configuration.

Table 10: Send to Queue node configurations

Property name	**Property value**
Queue name	Empty
Connection	Local queue manager
Destination queue manager name	QM1
Destination mode	Destination List
Policy	Empty

Trace nodes were configured to append updates to a trace file specific to the HTTP_Inbound application on the integration broker. The configuration of the trace file was simplified by promoting the destination and file path properties of each trace node to the message flow level, allowing for configuration of all trace nodes at once. The exact log message varied across trace files based on the action to be traced and the information available. While Kraft and colleagues (2010) recommended avoiding using dynamic logic in trace files, dynamic contents in logs were found to provide more benefits than drawbacks during the implementation. Trace node message configurations and sample logs are available in Appendix 5.

5.4.3 APP1_FTP_Uploader implementation

APP1_FTP_Uploader was implemented as an application responsible for transforming the message received from AWS_Connector and delivering the transformed message to the target FTP server. Only minimal functionality was implemented to keep the scope limited. Figure 10 shows the ideal path of the message in green, with yellow paths indicating actions during processing or delivery failure.

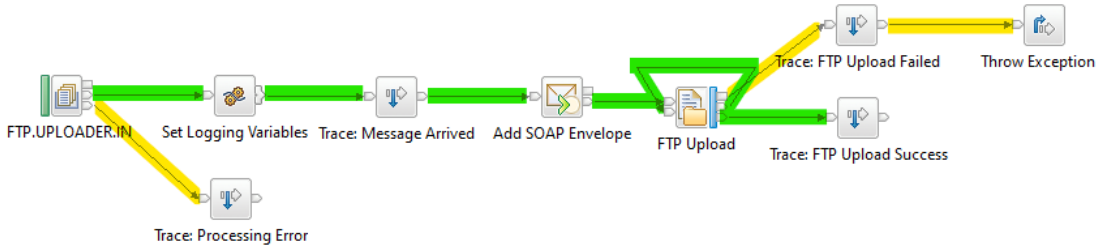

Figure 10: FTP Uploader message flow

The application reads in messages from queue FTP.UPLOADER.IN on WebSphere MQ queue manager QM1. On successful reads, Set Logging Variables, an ESQL Compute node, retrieves metadata for logging from message headers and sets a wildcard variable later used in file naming conventions during FTP upload. Source code for Set Logging Variables node is available in Appendix 6. After message arrival has been written to trace file, a SOAP envelope is added to the incoming message just to emulate a simple processing functionality for the incoming message. Once the SOAP envelope has been added, the message is stored on a remote FTP server using File Output node. The File Output node is configured to utilize FTP for file transfer, alongside with authentication with an identity stored in the integration broker's key vault. Specific configurations for the File Output node are listed below in Table 11.

Table 11: FTP Upload node configurations

Property name	**Property value**
File name or pattern	*.xml
File action	Write directly to the output file (append if file exists)
Data location	$Body
Remote Transfer	Selected
Transfer protocol	FTP
Server and port	192.168.1.96:21
Security identity	ftpIdentity
Server directory	/home/integration/integrations/in
Transfer mode	ASCII
Action if remote file exists	Replace Existing File (PUT)

The File Output node utilizes the wildcard variable created earlier to name the outgoing file, with message body to be used as file contents. The target folder was configured as a folder accessible through FTP on a remote Linux-based FTP server. An identity for ftpIdentity containing the remote server FTP username and password combination was added to the broker's key vault with IBM App Connect Enterprise Console with the following command:

```
mqsisetdbparms -w D:\Users\ESB\IBM\ACET11\workspace\INTSRV01 -n
ftp::ftpIdentity -u integration -p <password>
```

Once the File Output node has processed the message, the results are logged to a trace file. Traces are formed in the same manner as in AWS_Connector to an FTP_Upload message flow specific trace file, specific Pattern configurations and example traces are available in Appendix 7.

In case of FTP upload failure, the message flow throws an exception, triggering a reversal of the message flow direction to the MQ Input node FTP.UPLOADER.IN. Upon reaching the MQ Input node, the message is re-sent through Catch-terminal, resulting in Trace: Processing Error node logging the failure, and transfer of the message to the Backout Requeue Queue for FTP.UPLOADER.IN: FTP.UPLOADER.IN.FAIL. Throwing an exception ensures that the broker treats the processing as failed and stores the message in a dead-letter queue for further investigations and processing, ensuring data persistence.

5.4.4 APP2_HTTP_POST implementation

The second message processing flow implemented on ESB was a message flow with the responsibility of forwarding the incoming message to a HTTP server through a HTTP POST request. The implementation was kept as simple as possible to limit the scope of the work, and only the necessary functionality to create a HTTP POST request and enable message tracing was included. Figure 11 shows the final layout of

the message flow, as well as the ideal path in green, with processing failure paths in yellow.

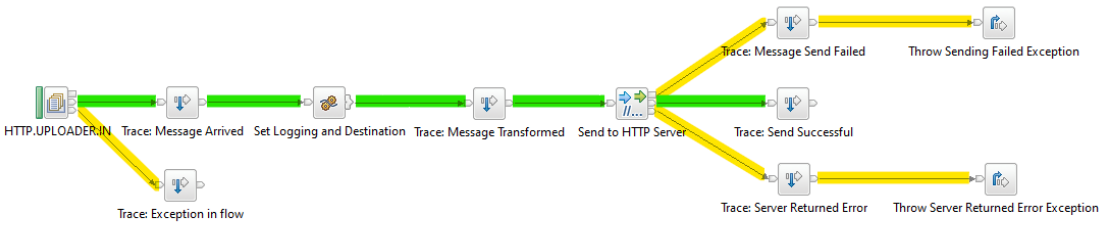

Figure 11: HTTP_POST message flow implementation

The message flow monitors a WebSphere MQ message queue, HTTP.UPLOADER.IN. Once a message is read from the queue, the read event is logged in a trace file and the message is transferred to Set Logging and Destination, an ESQL Compute node, for processing. The source code for Set Logging and Destination is available at Appendix 8. The compute node extracts metadata from incoming message headers and sets the target HTTP address. As the implementation mimics a web application interface with a HTTP echo server, the target address was configured to contain the message identifier to allow monitoring of received messages through the echo server's output information. A HTTP echo server does not store any of the received data and does not provide any handling for the incoming HTTP requests, only sends a reply to the caller indicating that the HTTP request was successful. Using the message identifier as the HTTP address suffix resulted in the HTTP echo server displaying the identifier of each message for which a HTTP POST request was received, allowing for easy verification that correct messages were received.

Once the necessary values to create a HTTP request have been set, the event is logged to trace files and the message passed on to HTTP Request node for HTTP POST request to the target server. The configurations used in HTTP Request node are listed in Table 12.

Table 12: HTTP Request configurations

Property name	**Property value**
Web service URL	http://192.168.1.97:8080
Request timeout (sec)	120
HTTP method	POST
Response Message Parsing	MIME: For MIME wrapped data including multipart
Replace input with error	Selected
Use whole input message as request	Selected
Replace input message with web-service response	Selected
Generate default HTTP headers from input	Selected

With the configurations set as in Table 12, the HTTP Request node returns the success or failure information from the HTTP POST request. The message flow logs the result of the HTTP request in trace files. In the case of unsuccessful requests, the message flows throws an exception with a Throw node to ensure that the message gets backed out into the original MQ queue, and from there to a dead-letter queue for later analysis.

Trace node configurations follow a similar pattern as for FTP Uploader, specifications and example logs available in Appendix 9.

5.4.5 Final ESB architecture

The final implementation of modules on ESB consisted of the applications listed in previous chapters working in tandem. Figure 12 displays the relationship between the individual components within the ESB, including the flow of data between components.

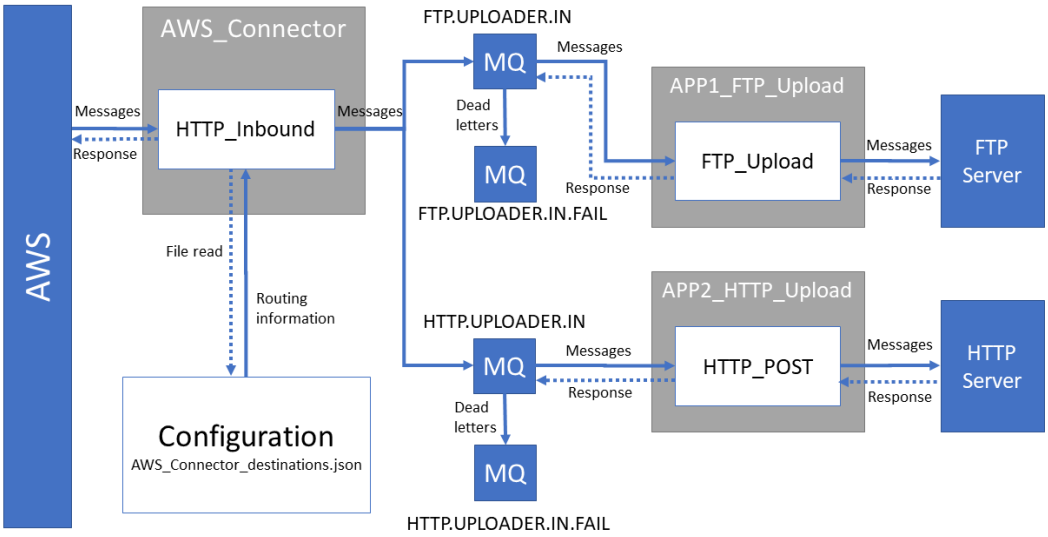

Figure 12: ESB architecture diagram

The implemented solution consisted of a connector application for communications with AWS, a configuration file to store message routing information, message queues to serve as persistent storage locations for messages awaiting processing and two recipient-specific applications to process and transfer messages. The scope of the research was limited to only providing a functional proof-of-concept solution for the integration, and as a result, security measures and other architectural considerations were not implemented.

5.5 Mock recipient applications

5.5.1 Overview of recipient applications

As the only purpose for mock applications in the documented implementation was to emulate the functionality of the application-specific connectors in the integration, the mock applications are described here only briefly. In practice, the mock applications described in the next chapters are fully replaceable with any application interfaces or connectors serving as the recipient side of the integration. The specific

configurations of the connector application output nodes may need to be adjusted based on the exact requirements set by the receiving applications.

5.5.2 FTP server

The FTP server utilized in the implementation was a default installation of vsftpd FTP server on an Ubuntu server. The integration broker was provided access to the FTP server with local server credentials, no additional configurations other than providing a place to store files were implemented for the user. FTP server logs on the Ubuntu server were utilized to verify and test the functionality of the integration.

5.5.3 HTTP echo server

A HTTP server interface was simulated by utilizing a containerized HTTP Echo Server in an Ubuntu server to provide a response to any requests received by the Echo Server. The Echo Server is a freely available HTTP service in a docker image, which echoes back any requests sent to the server (*Echo Server* N.d.). Echo Server output information was used to verify that correct messages were received by the Echo Server.

6 Results

The goal of the research was to implement an integration capable of transferring data from AWS to external environments via IBM ESB environment, while adhering to SOA and reliable messaging principles. This was achieved by implementing each component of the integration as an entity limited to performing a single functionality. Simultaneously, each component of the integration was designed to be reusable in the environment. Measures required by reliable messaging principles were implemented throughout the integration where suitable. The final implementation architecture is shown in Figure 13.

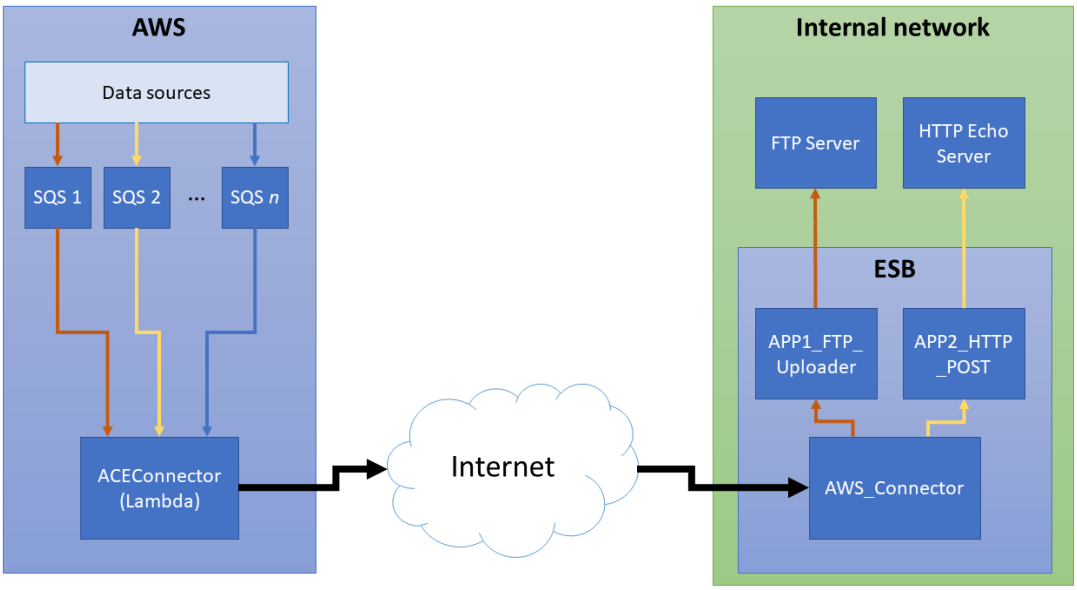

Figure 13: Implementation architecture

Implementing the integration based on SOA principles was closely equivalent to taking the path of least resistance in AWS environment, as AWS itself provides services in a SOA manner. The integration could be pieced together from various reusable services, and implementation in a non-SOA manner would have been difficult. AWS identity management features ensured authentication within the AWS environment, while SQS FIFO queues provided the means to ensure exactly-once and in-order delivery. CloudWatch logs ensure the traceability of actions within AWS, and the Lambda code provides delivery failure notification possibilities.

AWS side implementation still holds room for improvement and expansion. Currently the receipt of messages is limited to components within the same AWS environment, but the message consumption process could also be extended to other AWS environments within the same geographic region using Simple Token Service (STS) based execution role assumption within the message consumer code. This would require significant changes to the Lambda code, as SQS queues on external accounts cannot be set as triggers for the Lambda function. Instead, CloudWatch alerts could be used to

send timed triggers to the Lambda to poll for new messages on the external SQS queue.

Logging with CloudWatch required minimal effort due to the extensive work performed by AWS behind the scenes. Even an unrefined version of logging capabilities provided enough logs to support the traceability requirements of reliable messaging principles. CloudWatch holds further functionalities regarding alerts and notifications that should be explored in the case of a production environment deployment, but which were left out of scope for this implementation. CloudWatch includes methods to, for example, send alerts through SMS messages or email on the event of errors above a configurable threshold, which could be an invaluable aid in production environments looking to fully implement reliable messaging principles and actively monitor integrations in production.

The ESB side implementation loosely followed SOA principles by implementing a generic connector between AWS and ESB but could have been further refined by splitting the FTP Uploader and HTTP POST message flows into separate components. The first half would have provided transformation and mapping services for the specific message type under processing, while the latter half would have functioned as a generic HTTP or FTP uploader, capable of processing any message types provided. The current implementation also only supports incoming messages in JSON format, but with some work, could be extended to support a dynamic set of data structures.

Reliable messaging principles were mostly fulfilled throughout the implementation, with lack of implementation more often due to scope limitations or avoiding complex solutions, rather than actual technical obstacles. Sender authentication functionalities were implemented in the simplest available form, usually in the form of local credentials or basic authentication. A production environment might benefit from Lightweight Directory Access Protocol (LDAP) or Certificate-based authentication forms. Implementing more complex authentication formats would have required a more sophisticated environment and would likely have pushed the scope of the research too far from a reasonable size. Likewise, recipient authentication was not implemented in any form, as recipient authentication would most often be conducted through SSL

certificates and would have required domain registrations and a more complex environment setup. A production environment would be well advised to utilize recipient authentication for security reasons.

Traceability of messages was ensured through logging at each stage of the integration, with an identifier for a message traceable throughout the integration. The log files provide basic information regarding the flow of the message but could be improved upon by utilizing a data warehousing system to store further information, such as contents of messages at different stages of the integration.

Exactly-once and in-order requirements are maintained throughout the integration by the nature of the used platform. IBM's App Connect Enterprise solution processes messages in order, and when transit message queues are used to store messages between modules, the queues ensure First-In-First-Out delivery of messages. Error handling built within the message flow processes ensure that undeliverable messages are returned to dead-letter queues at both AWS and IBM platforms, so that no messages are lost during integration even in unexpected situations. An ideal solution would ensure a single collection point for all dead-letter messages, but would have required the implementation of a more sophisticated manner of delivery failure receipt towards AWS.

Failure notification features were built to some extent but were not extended to reach all the way to the initial sender application. ESB will notify AWS about an unrouteable message but will not pass on information of a message cannot be processed by a subsequent application within ESB. For a proof-of-concept solution with no real sender applications, the exact manner of implementing a delivery failure notification for the original sender application is unclear. In practice, a notification method for responding to the sender application could be built as a separate application within ESB, responsible for transferring success or failure notifications to the sending application through either HTTP responses or asynchronous messages.

All in all, the integration implementation solved the original goal of implementing a way to transfer information from AWS applications to services within an internal

network. While the implementation is functional and works as intended, many features could still be added to provide more value, especially for enterprise stakeholders.

7 Conclusions

The aim of the research was to create a proof-of-concept integration to transfer messages from Amazon Web Services to external services by utilizing IBM's App Connect Enterprise ESB. The goal was to design the integration to adhere to SOA principles and fulfill requirements for reliable messaging.

While not all outlined features were implemented in each stage of the integration, the development process demonstrated that the implementation would be possible even for more complex situations. The integration successfully gathered messages from multiple locations and transferred the messages to their target destinations, while maintaining a modular architecture and providing reliability measures. The integration did not fully adhere to SOA principles but could be modified to further separate functionalities within each stage with some effort. Additionally, some of the reliable messaging principles, especially recipient authentication and delivery failure messages to recipient, were difficult to implement in a reasonable manner in the development environment.

The chosen research method focuses on building new information upon existing knowledge through an ongoing process of implementation, examination and adjustment. Design-based research method worked well for the implementation, allowing for creation of practical knowledge to fit real-world use cases. Design-based research works in a qualitative manner and the direct evaluation of success or failure is not clear-cut, leaving much of the evaluation of success to individual readers. The resulting knowledge should therefore be critically examined and applied in real world scenarios where relevant, while adjusting practices to better fit the target environment.

The results of this research can be applied to implement integration connectors from AWS to third party systems with IBM's ESB platform, without having to create an architectural design from the scratch. The implementation provides a rough base for a basic connector setup with some example use case implementations, on which new implementations can build additional functionality and features or adapt to better fit the specific scenario at hand. The research included recommendations on modifications and decisions worth considering at each stage of the implementation.

This research provides a base to be used as a stage for further research. Future researchers could examine an implementation for an integration in the reverse direction; sending data from third-party applications to AWS. Additionally, further research could be directed towards performance- and stress-testing the integration implementation, focused on measuring the scalability of the integration under heavy loads. Further research could be conducted into creating a similar connector to collect messages from SQS queues in external AWS accounts for processing within an internal ESB. Additionally, further research is required into methods to reliably relay failure notifications from applications within ESB to the originating applications in AWS.

References

About Node.js. N.d. Article on Node.js website. Accessed 23 August 2020. Retrieved from https://nodejs.org/en/about/.

Amazon CloudWatch. N.d. Article on AWS website. Accessed 23 August 2020. Retrieved from https://aws.amazon.com/cloudwatch/.

Amazon SQS dead-letter queues. Article on AWS website. Accessed 09 September 2020. Retrieved from https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-dead-letter-queues.html.

Amazon SQS FIFO (First-In-First-Out) queues. Article on AWS website. Accessed 06 September 2020. Retrieved from https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/FIFO-queues.html.

Amazon SQS Standard queues. Article on AWS website. Accessed 09 September 2020. Retrieved from https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/standard-queues.html.

Amazon SQS visibility timeout. Article on AWS website. Accessed 09 September 2020. Retrieved from https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-visibility-timeout.html.

An Overview of HTTP. N.d. Article on MDN website. Accessed 23 August 2020. Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.

Authentication and Authorization. N.d. Article on Auth0 website. Accessed 05 September 2020. Retrieved from https://auth0.com/docs/authorization/authentication-and-authorization.

AWS Lambda. N.d. Article on AWS website. Accessed 23 August 2020. Retrieved from https://aws.amazon.com/lambda/.

AWS Lambda FAQs. N.d. Article on AWS website. Accessed 10 September 2020. Retrieved from https://aws.amazon.com/lambda/faqs/.

Brown, A. 2001. *Reliable Messaging*. Article on World Wide Web Consortium (W3C) website. Accessed 29 July 2020. Retrieved from https://www.w3.org/2001/03/WSWS-popa/paper40.

Building Lambda functions with Node.js. N.d. Article on AWS website. Accessed 23 August 2020. Retrieved from https://docs.aws.amazon.com/lambda/latest/dg/lambda-nodejs.html.

Burger, A. 2014. *Report: SaaS Usage up 500% Since 2010; Second Cloud Front on Its Way*. Article on TeleCompetitor website. Accessed 14 September 2020. Retrieved

from https://www.telecompetitor.com/report-saas-usage-up-500-since-2010-second-cloud-front-on-its-way/.

Cloud computing with AWS. N.d. Article on AWS website. Accessed 23 August 2020. Retrieved from https://aws.amazon.com/what-is-aws/?nc1=f_cc.

Columbus, L. 2018. *85% of Enterprise Workloads Will Be In The Cloud By 2020.* Article on Forbes' website. Accessed 29 July 2020. Retrieved from https://www.forbes.com/sites/louiscolumbus/2018/01/07/83-of-enterprise-workloads-will-be-in-the-cloud-by-2020/#2e513c5b6261.

Doan, A., Halevy, A. & Ives, Z. 2012. *Principles of Data Integration*. San Fransisco: Elsevier.

Echo Server. N.d. jmalloc Echo Server container download page on DockerHub website. Accessed 11 September 2020. Retrieved from https://hub.docker.com/r/jmalloc/echo-server/.

ESQL Overview. N.d. Knowledge base article on IBM Knowledge Center website. Accessed 23 August 2020. Retrieved from https://www.ibm.com/support/knowledgecenter/en/SSMKHH_10.0.0/com.ibm.etools.mft.doc/ak00990_.htm.

Extracting business value from the 4 V's of big data. N.d. Infographic on IBM's Big Data & Analytics Hub website. Accessed 29 July 2020. Retrieved from https://www.ibmbigdatahub.com/infographic/extracting-business-value-4-vs-big-data.

FTP. N.d. Article on JavaTPoint website. Accessed 23 August 2020. Retrieved from https://www.javatpoint.com/computer-network-ftp.

Gartner Says 20 Percent of Spending in Key IT Segments Will Shift to the Cloud by 2022. 2018. Press release on Gartner's website. Accessed 29 July 2020. Retrieved from https://www.gartner.com/en/newsroom/press-releases/2018-09-18-gartner-says-28-percent-of-spending-in-key-IT-segments-will-shift-to-the-cloud-by-2022.

Gulledge, T. 2006. What is integration? *Industrial Management & Data Systems*, 106(1), 5-20.

Hoadley, C. 2004. Methodological Alignment in Design-Based Research. *Educational Psychologist, 39*, 203-212.

HTTP Authentication. N.d. Article on MDN website. Accessed 23 August 2020. Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication.

Identity and access management in Amazon SQS. N.d. Article on AWS website. Accessed 10 September 2020. Retrieved from https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-authentication-and-access-control.html.

Introduction to XML. N.d. Article on W3Schools website. Accessed 23 August 2020. Retrieved from https://www.w3schools.com/xml/xml_whatis.asp

Johansson, L. 2019. *What is message queuing?* Article on CloudAMQP Website. Accessed 23 August 2020. Retrieved from https://www.cloudamqp.com/blog/2014-12-03-what-is-message-queuing.html.

JSON Introduction. N.d. Article on W3Schools website. Accessed 23 August 2020. Retrieved from https://www.w3schools.com/js/js_json_intro.asp.

Kraft, J., Wall, A., & Kienle, H. 2010. Trace recording for embedded systems: Lessons leared from five industrial projects. *International Conference on Runtime Verification*, 315-329.

Lee, J., Siau, K. & Hong, S. 2003. Enterprise Integration with ERP and EAI. *Communications of the ACM, 46.2*, 54-60.

Lenzerini, M. 2002. Data integration: A theoretical perspective. *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 233-246.

Menge, F. 2007. Enterprise service bus. *Free and open source software conference, 2*, 1-6.

Message Delivery Reliability. N.d. Documentation on Akka website. Accessed 06 September 2020. Retrieved from https://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html.

Message Queues. N.d. Article on AWS website. Accessed 23 August 2020. Retrieved from https://aws.amazon.com/message-queue/.

Messaging Concepts. N.d. Article on Apache website. Accessed 06 September 2020. Retrieved from https://activemq.apache.org/components/artemis/documentation/1.1.0/messaging-concepts.html.

Messaging Patterns. N.d. Article on Enterprise Integration Patterns website. Accessed 23 August 2020. Retrieved from https://www.enterpriseintegrationpatterns.com/patterns/messaging/Introduction.html.

Mitchell, L. 2017. *Handling Failure Successfully in RabbitMQ*. Article on Medium website. Accessed 06 September 2020. Retrieved from https://medium.com/codait/handling-failure-successfully-in-rabbitmq-22ffa982b60f.

Moore, S. 2018. *How to Create a Business Case for Data Quality Improvement*. Publication on Gartner's website. Accessed 29 July 2020. Retrieved from https://www.gartner.com/smarterwithgartner/how-to-create-a-business-case-for-data-quality-improvement/.

More, A. 2020. *Enterprise Application Integration Market 2020 Industry Growth Analysis, Segmentation, Size, Share, Trend, Future Demand and Leading Players Updates by Forecast By 360 Market Updates.* Article on MarketWatch website. Accessed 23 August 2020. Retrieved from https://www.marketwatch.com/press-release/enterprise-application-integration-market-2020-industry-growth-analysis-segmentation-size-share-trend-future-demand-and-leading-players-updates-by-forecast-by-360-market-updates-2020-06-25.

Nemeth, G. 2015. *Web Authentication Methods Explained*. Blog post on RisingStack website. Accessed 05 September 2020. Retrieved from https://blog.risingstack.com/web-authentication-methods-explained/.

Priority. N.d. Article on IBM website. Accessed 06 September 2020. Retrieved from https://www.ibm.com/support/knowledgecenter/SSFKSJ_9.1.0/com.ibm.mq.dev.doc/q026260_.htm.

Quarterly SaaS Spending Reaches $20 billion as Microsoft Extends its Market Leadership. 2018. Article on Synergy's website. Accessed 29 July 2020. Retrieved from https://www.srgresearch.com/articles/quarterly-saas-spending-reaches-20-billion-microsoft-extends-its-market-leadership.

Reliability Patterns. N.d. Article on MuleSoft website. Accessed 23 August 2020. Retrieved from https://docs.mulesoft.com/mule-runtime/4.3/reliability-patterns.

Sabljakovic, H. 2019. *Make a http post from aws lambda*. Article on Medium website. Accessed 11 September 2020. Retrieved from https://medium.com/@sabljakovich/http-post-request-from-node-js-in-aws-lambda-826d57f0680.

Saltzer, J. H., & Schroeder, M. D. 1975. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1278-1308.

Schmidt, M. T., Hutchison, B., Lambros, P. & Phippen, R. 2005. The enterprise service bus: making service-oriented architecture real. *IBM Systems Journal, 44.4*, 781-797.

Sending and Receiving Messages in Amazon SQS. N.d. Developer guide on AWS website. Accessed 11 September 2020. Retrieved from https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/sqs-examples-send-receive-messages.html.

SFTP – SSH Secure File Transfer Protocol. N.d. Article on SSH.com website. Accessed 23 August 2020. Retrieved from https://www.ssh.com/ssh/sftp/.

Sharma, M. 2019. *What is Message Queue.* Article on Medium website. Accessed 23 August 2020. Retrieved from https://medium.com/@Mohitdtumce/what-is-message-queue-b5468ff6db50.

Sherman, R. 2015. *Business Intelligence Guidebook: From Data Integration to Analytics*. San Fransisco: Elsevier.

Transaction Processing. 2017. Article on Technopedia website. Accessed 06 September 2020. Retrieved from https://www.techopedia.com/definition/464/transaction-processing.

Tyson, M. 2020. *What is service-oriented architecture?* Article on InfoWorld website. Accessed 02 September 2020. Retrieved from https://www.infoworld.com/article/2071889/what-is-service-oriented-architecture.html.

Wang, F., & Hannafin, M. 2005. Design-Based Research and Technology-Enhanced Learning Environments. *Educational Technology Research and Development*, *53*, 5-23.

What is an SSL Certificate? N.d. Article on CloudFlare website. Accessed 05 September 2020. Retrieved from https://www.cloudflare.com/learning/ssl/what-is-an-ssl-certificate/.

What Is HTTPS? N.d. Article on CloudFlare website. Accessed 23 August 2020. Retrieved from https://www.cloudflare.com/learning/ssl/what-is-https/.

What is integration? N.d. Article on RedHat website. Accessed 23 August 2020. Retrieved from https://www.redhat.com/en/topics/integration/what-is-integration.

XML Soap. N.d. Article on W3Schools website. Accessed 23 August 2020. Retrieved from https://www.w3schools.com/xml/xml_soap.asp.

Appendices

Appendix 1. Lambda source code: ACEConnector.js

```
const http = require('http');

const sendData = (eventData) => {
    return new Promise((resolve, reject) => {
        var postOptions = {
            host: process.env.HOST_IP,
            port: process.env.HOST_PORT,
            path: process.env.HOST_PATH,
            method: 'POST',
            headers: { Authorization: 'Basic ' + new
Buffer.from(process.env.USERNAME + ':' + process.env.PASSWORD, 'utf-
8').toString('base64') }
        };

        //Request object creation, return status code from integra-
tion server
        const req = http.request(postOptions, (result) => {
            var statusCode = JSON.stringify(result.statusCode);
            //Reject if statusCode is not 200 (i.e. message NOT de-
livered)
            statusCode != 200 ? reject(result.statusMessage) : re-
solve(statusCode);
        });

        //Reject promise with received error
        req.on('error', e => reject(e.message));

        //Send data to integration server and complete request
        req.write(JSON.stringify(eventData));
        req.end();
    });
};

exports.handler = async (event) => {
    //Send data to integration server
    await sendData(event)
        .then(result => console.log('Status code: ' + result))
        .catch(err => {
            console.error('Encountered error: ' + err);
            //Throw an exception to ensure message is returned to
queue
            throw new Error('Encountered error: ' + err);
        });
};
```

Appendix 2. Lambda source code: SQSMessageGenerator

```
// Dependencies and queue object setup
var AWS = require('aws-sdk');
var sqs = new AWS.SQS({region: 'eu-north-1'});

// Function for sending message to queue
const sendData = (targetQueue, msgBody, counter) => {
    return new Promise((resolve, reject) => {
        // Queue attribute and message body setup
        var params = {
            MessageBody: msgBody,
            MessageDeduplicationId: counter.toString(),
            MessageGroupId: counter.toString(),
            QueueUrl: targetQueue
        };

        // Send to queue
        sqs.sendMessage(params, function(err, data){
            if (err){
                console.log("Error: ", err);
                reject(err);
            } else {
                resolve(data.MessageId);
            }
        });
    });
}

exports.handler = async (event) => {
    var queues = ["https://sqs.eu-north-1.amazo-
naws.com/737528452624/MockDataSource1.fifo", "https://sqs.eu-
north-1.amazonaws.com/737528452624/MockDataSource2.fifo"]

    for (var i = 0; i < 10; i++) {
        // Set up target queue and message body
        var target = queues[i % 2];
        var body = "<message><targetQueue>" + target + "</tar-
getQueue><i>" + i + "</i></message>";
        //Send message to queue
        await sendData(target, body, i)
            .then(result => console.log("Message Id: " + re-
sult))
            .catch(err => console.log(err));
    }
};
```

Appendix 3. HTTP_Input message parse and split code

```
CREATE COMPUTE MODULE HTTP_Inbound_SetDestinationQueue
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
-- Reference declarations
DECLARE rMsgIn REFERENCE TO InputRoot.JSON.Data.Records;
CREATE FIELD Environment.Variables.Staging;
DECLARE rMsgOutStaging REFERENCE TO Environment.Variables.Staging;
DECLARE rItem REFERENCE TO rItem;
DECLARE rMsgOut REFERENCE TO rMsgOut;
DECLARE MsgParseCount INTEGER 0;

-- One incoming message batch may contain multiple messages,
-- messages parsed and forwarded to application flows separately
FOR rItem AS rMsgIn.Item[] DO
           -- Create a copy of the extracted message to environment
           CREATE LASTCHILD OF rMsgOutStaging AS rMsgOut NAME 'Item';

           -- Extract destination queue name
           SET rMsgOut.DestinationQueue = THE(
                      SELECT R.Destination
                      FROM Environment.Variables.filecon-
                      tent.JSON.Data.QueueList.Item[] AS R
                      WHERE R.OriginQueue = rItem.eventSourceARN
           );

           -- Extract message and parse as XMLNSC
           DECLARE Data BLOB;
           SET Data = CAST(rItem.body AS BLOB CCSID 1208);
           CREATE LASTCHILD OF rMsgOut DOMAIN('XMLNSC') PARSE(Data);

           -- Increase message counter
           SET MsgParseCount = MsgParseCount + 1;
END FOR;

-- Store message counter and source queue in environment for trace
nodes
SET Environment.Variables.MsgParseCount = MsgParseCount;
SET Environment.Variables.MsgSource = rMsgIn.Item.eventSourceARN;

-- All messages parsed to environment, continue flow
RETURN TRUE;

END;
END MODULE;
```

Appendix 4. HTTP_Inbound message sending code

```
CREATE COMPUTE MODULE HTTP_Inbound_SendMessages
           CREATE FUNCTION Main() RETURNS BOOLEAN
           BEGIN
-- Reference declarations
DECLARE rStagedMsgs REFERENCE TO Environment.Variables.Staging;
DECLARE rItem REFERENCE TO rItem;
DECLARE MsgSendCount INTEGER 0;

-- Send each message in staging area
FOR rItem AS rStagedMsgs.Item[] DO
-- Clear output queue destination list and populate it with
-- information obtained from configuration files
           SET OutputLocalEnvironment.Destination.MQ.DestinationData
           = NULL;
           CREATE FIELD OutputLocalEnvironment.Destina-
           tion.MQ.DestinationData.queueName VALUE rItem.Destination-
           Queue.Destination;

-- Create a unique message ID and add to list of processed messages
           DECLARE MsgId CHARACTER uuidaschar;
           SET Environment.Variables.MsgList = COALESCE( (Environ-
           ment.Variables.MsgList || ', ' || MsgId), MsgId);

           -- Add metadata to RFH2 headers
           SET OutputRoot.MQRFH2.(MQRFH2.Field)Version = 2;
           SET OutputRoot.MQRFH2.(MQRFH2.Field)Format = 'MQSTR';
           SET OutputRoot.MQRFH2.(MQRFH2.Field)NameValueCCSID = 1208;
           SET OutputRoot.MQRFH2.MsgInfo.OriginApplication =
           'AWS_Connector';
           SET OutputRoot.MQRFH2.MsgInfo.OriginFlow = 'HTTP_Inbound';
           SET OutputRoot.MQRFH2.MsgInfo.MsgId = MsgId;

           -- Move stored message to OutputRoot
           SET OutputRoot.XMLNSC = rItem.XMLNSC;

           -- Send message
           PROPAGATE TO TERMINAL 'out1';

           -- Increase sent message counter
           SET MsgSendCount = MsgSendCount + 1;
END FOR;

-- Store sent message counter for trace node
SET Environment.Variables.MsgSendCount = MsgSendCount;

-- All messages sent, send signal to logging and HTTPReply
RETURN TRUE;

END;
END MODULE;
```

Appendix 5. AWS_Connector trace node configurations

Trace node name	Trace pattern
Trace: Timeout	${CURRENT_TIMESTAMP} Timeout occurred when reading messages into integration.
Trace: Input Failure	${CURRENT_TIMESTAMP} Input was sent to integration, but could not be read into flow.
Trace: Exception in Flow	${CURRENT_TIMESTAMP} - Source: ${Root.JSON.Data.Records.Item.eventSourceARN} - Message processing failed, an error was encountered
Trace: Message Arrived	${CURRENT_TIMESTAMP} - Source: ${Root.JSON.Data.Records.Item.eventSourceARN} - Message batch arrived
Trace: No Queue Match	${CURRENT_TIMESTAMP} - Source: ${Root.JSON.Data.Records.Item.eventSourceARN} - No destination queue match found for input queue
Trace: Parse Success	${CURRENT_TIMESTAMP} - Source: ${Environment.Variables.MsgSource} - Messages parsed successfully - ${Environment.Variables.MsgParseCount} messages parsed.
Trace: Send Successful	${CURRENT_TIMESTAMP} - Source: ${Environment.Variables.MsgSource} - Message send process successful - ${Environment.Variables.MsgSendCount} messages sent, identifiers: ${Environment.Variables.MsgList}

2020-07-28 12:16:49.899623 - Source: 'arn:aws:sqs:eu-north-1:737528452624:MockDataSource2.fifo' - Message batch arrived

2020-07-28 12:16:49.991457 - Source: 'arn:aws:sqs:eu-north-1:737528452624:MockDataSource2.fifo' - Messages parsed successfully - 10 messages parsed.

2020-07-28 12:16:50.322409 - Source: 'arn:aws:sqs:eu-north-1:737528452624:MockDataSource2.fifo' - Message send process successful - 10 messages sent, identifiers: '65efc648-e8c6-4a61-8d26-5d9c91aa0775, d9f194b1-3fbd-4eee-a05b-974d94312131, f264b1ab-492e-4311-bc3e-220db11d0c00, 777ef3ce-3036-42e3-969c-b7d35aaa9178, 24a6ad81-f7d0-4e8f-be6a-d640964b4aa0, aac89fef-913a-4256-9d03-d826f1a8be57, 8ad4642e-bacf-41ce-b2b1-1d41500c7e30, 80c2a569-7014-4d2d-a67a-186e5502b2e9, 169d73af-8f11-41fe-82b0-8385b5990454, 85258494-b526-4af4-b9ce-b45f2ffb0a8b'

Appendix 6. FTP_Upload Set Logging Variables source code

```
CREATE COMPUTE MODULE FTP_Upload_Set_Logging_Variables

CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN

-- Copy entire message contents to be passed forward
CALL CopyEntireMessage();

-- Store information required by trace nodes and file output node
SET Environment.Variables.OriginApplication =
          InputRoot.MQRFH2.MsgInfo.OriginApplication;
SET Environment.Variables.OriginFlow =
          InputRoot.MQRFH2.MsgInfo.OriginFlow;
SET Environment.Variables.MsgId = InputRoot.MQRFH2.MsgInfo.MsgId;
SET OutputLocalEnvironment.Wildcard.WildcardMatch
          = COALESCE(Environment.Variables.MsgId,
          CAST(CURRENT_TIMESTAMP AS CHARACTER FORMAT
          'yyyyMMddHHmmssSSSS'));

-- Send forward in flow
RETURN TRUE;
END;

-------------------------------------------------------------

CREATE PROCEDURE CopyEntireMessage() BEGIN
          SET OutputRoot = InputRoot;
END;

END MODULE;
```

Appendix 7. APP1_FTP_Uploader trace node

Trace node name	Trace pattern
Trace: Processing Error	${CURRENT_TIMESTAMP} - Source: ${Environment.Variables.OriginApplication}: ${Environment.Variables.OriginFlow} - Id: ${Environment.Variables.MsgId} - Message processing encountered an error
Trace: Message Arrived	${CURRENT_TIMESTAMP} - Source: ${Environment.Variables.OriginApplication}: ${Environment.Variables.OriginFlow} - Id: ${Environment.Variables.MsgId} - Message arrived to flow
Trace: FTP Upload Failed	${CURRENT_TIMESTAMP} - Source: ${Environment.Variables.OriginApplication}: ${Environment.Variables.OriginFlow} - Id: ${Environment.Variables.MsgId} - Message upload to FTP service failed.
Trace: FTP Upload Success	${CURRENT_TIMESTAMP} - Source: ${Environment.Variables.OriginApplication}: ${Environment.Variables.OriginFlow} - Id: ${Environment.Variables.MsgId} - Message uploaded to FTP service.

2020-07-21 19:01:41.572168 - Source: 'AWS_Connector': 'HTTP_Inbound' - Id: '0b81f528-bf98-4597-a177-ed4102f5515c' - Message uploaded to FTP service.

2020-07-21 19:02:19.420471 - Source: 'AWS_Connector': 'HTTP_Inbound' - Id: 'b45895fd-c143-46c8-845f-7c05f3437d56' - Message arrived to flow

2020-07-21 19:02:28.191047 - Source: 'AWS_Connector': 'HTTP_Inbound' - Id: 'b45895fd-c143-46c8-845f-7c05f3437d56' - Message upload to FTP service failed.

2020-07-21 19:07:10.710610 - Source: 'AWS_Connector': 'HTTP_Inbound' - Id: 'b4a7bc25-0c68-4334-b086-eace0937ae07' - Message arrived to flow

2020-07-21 19:07:13.117682 - Source: 'AWS_Connector': 'HTTP_Inbound' - Id: 'b4a7bc25-0c68-4334-b086-eace0937ae07' - Message uploaded to FTP service.

Appendix 8. HTTP_POST Set Logging and Destination source code

```
CREATE COMPUTE MODULE HTTP_POST_Set_Logging_and_Destination
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
-- Copy message for sending
CALL CopyEntireMessage();

-- Set variables for logging and target url
SET Environment.Variables.OriginApplication =
          InputRoot.MQRFH2.MsgInfo.OriginApplication;
SET Environment.Variables.OriginFlow =
          InputRoot.MQRFH2.MsgInfo.OriginFlow;
SET Environment.Variables.MsgId = InputRoot.MQRFH2.MsgInfo.MsgId;
          SET OutputLocalEnvironment.Destination.HTTP.RequestURL =
          'http://192.168.1.97:8080/' ||
          Environment.Variables.MsgId;

-- Send message forward in flow
RETURN TRUE;
END;

CREATE PROCEDURE CopyMessageHeaders() BEGIN
          DECLARE I INTEGER 1;
          DECLARE J INTEGER;
          SET J = CARDINALITY(InputRoot.*[]);
          WHILE I < J DO
                    SET OutputRoot.*[I] = InputRoot.*[I];
                    SET I = I + 1;
          END WHILE;
END;

CREATE PROCEDURE CopyEntireMessage() BEGIN
          SET OutputRoot = InputRoot;
END;

END MODULE;
```

Appendix 9. APP2_HTTP_POST trace node configurations

Trace node name	**Trace pattern**
Trace: Message Arrived	${CURRENT_TIMESTAMP} - Source: ${Root.MQRFH2.MsgInfo.OriginApplication}: ${Root.MQRFH2.MsgInfo.OriginFlow} - Id: ${Root.MQRFH2.MsgInfo.MsgId} - Message arrived to flow
Trace: Exception in flow	${CURRENT_TIMESTAMP} - Source: ${Environment.Variables.OriginApplication}: ${Environment.Variables.OriginFlow} - Id: ${Environment.Variables.MsgId} - Exception encountered when processing message
Trace: Message Transformed	${CURRENT_TIMESTAMP} - Source: ${Environment.Variables.OriginApplication}: ${Environment.Variables.OriginFlow} - Id: ${Environment.Variables.MsgId} - Message transformed
Trace: Message Send Failed	${CURRENT_TIMESTAMP} - Source: ${Environment.Variables.OriginApplication}: ${Environment.Variables.OriginFlow} - Id: ${Environment.Variables.MsgId} - Unable to send message
Trace: Send Successful	${CURRENT_TIMESTAMP} - Source: ${Environment.Variables.OriginApplication}: ${Environment.Variables.OriginFlow} - Id: ${Environment.Variables.MsgId} - Message sent successfully
Trace: Server Returned Error	${CURRENT_TIMESTAMP} - Source: ${Environment.Variables.OriginApplication}: ${Environment.Variables.OriginFlow} - Id: ${Environment.Variables.MsgId} - Sending unsuccessful, server returned an error

2020-07-28 12:20:41.560537 - Source: 'AWS_Connector': 'HTTP_Inbound' - Id: 'b10b601d-e1a0-4e50-989f-57a4f581ee77' - Message arrived to flow

2020-07-28 12:20:41.560700 - Source: 'AWS_Connector': 'HTTP_Inbound' - Id: 'b10b601d-e1a0-4e50-989f-57a4f581ee77' - Message transformed

2020-07-28 12:20:41.561392 - Source: 'AWS_Connector': 'HTTP_Inbound' - Id: 'b10b601d-e1a0-4e50-989f-57a4f581ee77' - Message sent successfully