



Alan Duffin

X00159409

CERTIFICATE, CLOUD SOLUTIONS ARCHITECTURE 2019, PROJECT
FINAL SUBMISSION

TABLE OF CONTENTS

FINAL PROJECT SUBMISSION	3
PROJECT PROPOSAL	5
INITIAL RESEARCH AND PROJECT PLAN	7
FEASIBILITY AND REFINED RESEARCH PLAN	20
LITERATURE REVIEW	37
TECHNICAL ASSESSMENT	56
IMPLEMENTATION ITERATIONS.....	99
CONCLUSION.....	246

CERTIFICATE CLOUD SOLUTIONS FINAL SUBMISSION - INTRODUCTION

Workflow is a common pattern for solving use cases across a broad range of sectors. These use cases require maintaining state and coordinating operations across sometimes heterogeneous / distributed participants over extended time periods. Examples of common problem areas include loan/insurance origination, order management, billing systems and logistics fulfilment.

The use case selected for my project is Loan origination. This is the process whereby a potential borrower applies for a new loan, and a lender processes that application. Loan Origination includes all the steps from taking a loan application up to drawdown of funds or refusal of the application. The loan origination process has many long running steps, including human decision points.

The central objective of this project is to build a solution for long – running mortgage processing that requires human interaction / approval. The specific problem domain is the mortgage application / approval process. The implementation platform chosen after the Literature Review and Technical Evaluation stages was the AWS Serverless Application Model using AWS Step Functions and AWS Lambda.

The breadth of AWS Services that I required to implement this project was very wide. The full list of AWS Services used are:

- CloudFormation
- Serverless Application Model templates
- IAM
- S3
- DynamoDB
- Lambda
- SNS
- API Gateway Rest API
- API Gateway Websocket API
- Step Functions

In addition to these AWS services, it was necessary to learn a number of other open source frameworks that would allow me to create a useable development / testing environment. This are listed below:

- Git
- Javascript / NodeJS / NPM
- Angular / Typescript
- Docker
- Websockets NPM
- Visual Studio Code IDE

Some simple project metrics from my project are:

Metric	#
Lambda Functions	31
Lines of Code	6160
S3 Buckets	2
SNS Topics	2
DynamoDB Tables	6
API Gateway HTTP Endpoints	14
Step Function States	29

CERT.CLOUD.SOLUTIONS PROJECT PROPOSAL

STUDENT

Student Name: Alan Duffin

Student ID: X00159409

Module: Certificate in Cloud Solutions Architecture (Project)

OVERVIEW

This document is an initial proposal for the final project module of the Certificate in Solutions Architecture at TUD (Tallaght Campus).

Workflow is a common pattern for solving use cases across a broad range of sectors. These use cases require maintaining state and coordinating operations across sometimes heterogeneous / distributed participants over extended time periods. Examples of common problem areas include loan/insurance origination, order management, billing systems and logistics fulfilment.

These systems have a common problem set that must be solved. The workflow pattern must address requirements such as state transition, error handling, component / service integration and human interaction/decision making.

Many vendors have designed solution frameworks around the concept of the Finite State Machine such as BPEL (Business Process Execution Language). The growth of SaaS and PaaS platforms has allowed users to take advantage of fully managed, scalable services for many problem domains including workflow.

Amazon Web Services provides access to two managed services for workflow use cases. These are:

- 1 AWS Step Functions
- 2 AWS Simple Workflow Service

GOALS

The goal of the proposed project is to compare and contrast AWS Step Functions and AWS Simple Workflow for a problem domain such as Loan Origination under a number of categories including:

- Features
- Ease of use
- Latency
- Cost
- Maintainability / Troubleshooting
- State Maintenance / Error Handling

Dublin Technological University,
Certificate, Cloud Solutions Architecture (2020)

Initial Research and Project Plan

Student Name: Alan Duffin

Student ID: X00159409

CONTENTS

Overview	2
Use Case Outline	3
Initial Research	7
Initial Project Plan	8
Risks.....	9

OVERVIEW

Workflow is a common pattern for solving use cases across a broad range of sectors. These use cases require maintaining state and coordinating operations across sometimes heterogeneous / distributed participants over extended time periods. Examples of common problem areas include loan/insurance origination, order management, billing systems and logistics fulfilment.

These systems have a common problem set that must be solved. The workflow pattern must address requirements such as state transition, error handling, component / service integration and human interaction/decision making.

Many vendors have designed solution frameworks around the concept of the Finite State Machine such as BPEL (Business Process Execution Language). The growth of SaaS and PaaS platforms has allowed users to take advantage of fully managed, scalable services for many problem domains including workflow.

My project proposal is to research the available Workflow/Servlerless services that are available from the three largest SaaS Cloud providers: Google Cloud, Amazon Web Services, Microsoft Azure. I will subsequently create a comparison matrix for these SaaS platforms based on a set of required criteria for the Workflow use case that is to be implemented. This matrix will be used to select a Cloud vendor platform upon which to implement the use case. Some of these criteria/features are:

- Features
- Ease of use
- Latency
- Cost
- Maintainability / Troubleshooting
- State Maintenance / Error Handling.

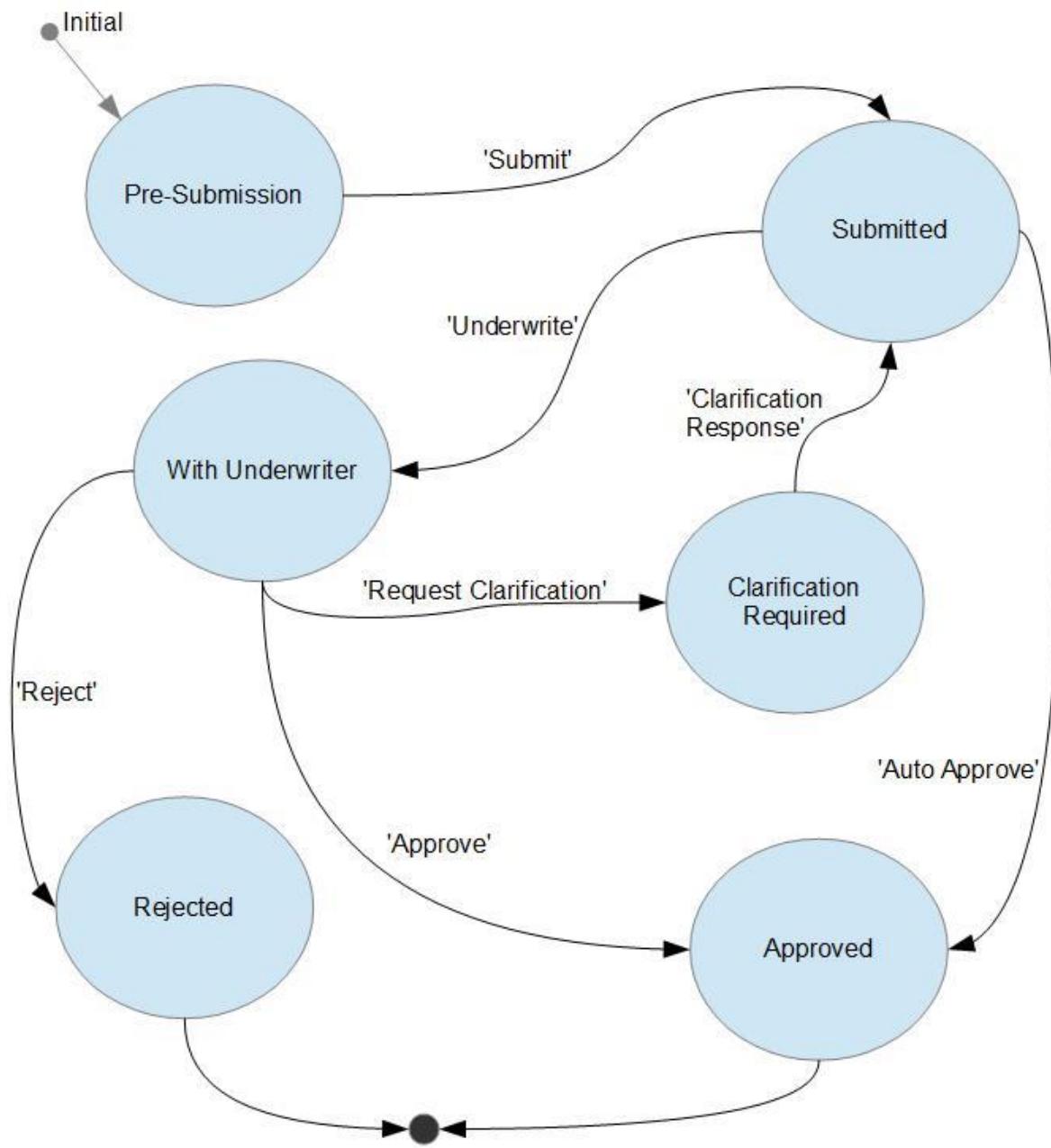
USE CASE OUTLINE

Loan origination is the process by which a borrower applies for a new loan, and a lender processes that application. Origination includes all the steps from taking a loan application up to drawdown of funds or refusal of the application.

The loan origination process has many steps, including:

- Loan application — The homebuyer fills out a loan application form.
- Documents — The buyer submits documentation for income, employment and financial status.
- Underwriting — The lender reviews the loan application, verifies the buyer's credit and determines if the buyer's income and financial status is such that the buyer qualifies for the loan.
- Finalise loan application — Once the terms are agreed on, the loan application is processed by the lender, going through several departments, such as underwriting and documentation processing.
- Loan approval — After final processing, the lender decides to approve or reject the loan application.

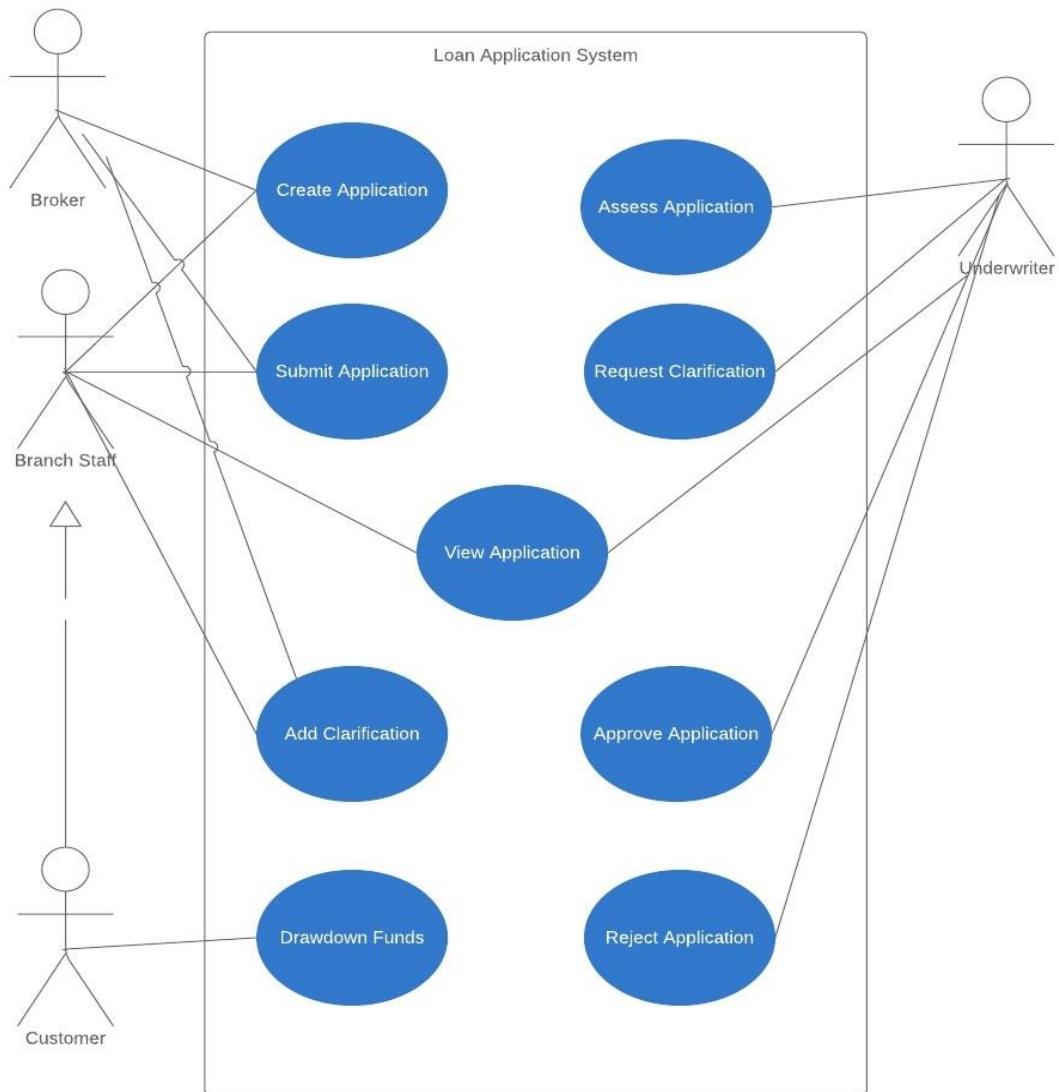
The Lifecycle of a Homeloan Application



Human Actors	
Branch Staff	Actor 1
Broker	Actor 2
Customer	Actor 3
Underwriter	Actor 4

Loan Origination Use Case Diagram

Alan Duffin | October 3, 2019



The Bank Branch Staff Member or Broker (Actor 1) via the user interface will view the list of active Loan Applications.

- Actor 1 can choose to view the details of any individual Loan Application.
- Actor 1 can create a new Loan Application on behalf of a customer.
- Actor 1 can input all required customer and loan details.
- After reviewing all information Actor 1 can submit the Loan application into the Approvals workflow. This should initiate a workflow execution.
- Possibly notifications should be sent to all human actors of loan application state changes. This could be via email, or a front end notification.
- The workflow execution should consist of the steps shown in the table below (not exhaustive).

Step Number	Workflow Component
1	Validate that all mandatory data has been input for the Approval process
2	Run the affordability calculator and validate that the case does not fall outside loan rules e.g. age of applicants, loan term, unsuitable employment, not in current employment for long enough etc.
3	Decision rule to either auto approve or route the application to an Underwriter (Actor 2) for human review.
4	The Auto Approval service will mark the application as approved and notify Actor 1. (Notification method unknown at the moment)
5	The Underwriters have discretion to approve the loan application based on extenuating circumstances such as Junior doctor with immediate prospects of salary increases and stable job prospects.
6	If manual Approval is required the workflow will notify Actor 4 that the Loan Application is awaiting review and pause execution.
7	Actor 4 must open the Loan Application and review details.
8	Actor 4 can then choose to Approve or Decline the application. This is done via the front end and calls a service that restarts the workflow execution.
9	The workflow will then notify Actor 1 and the Customer (Actor 3) using some mechanism.
10	The workflow will terminate.

INITIAL RESEARCH

My initial research of the Google, Microsoft and AWS PaaS services, show that the services listed below can be considered as candidate solution platforms for the workflow use-case of this project.

Each of the PaaS cloud vendors provide services that will be required for this use case. These provide solutions for the following problem domains:

- 1 Workflow Orchestration
- 2 Serverless hosting of business logic.
- 3 Publish/Subscribe for messaging/notification.
- 4 Queuing for asynchronous task processing.

SaaS Provider	Workflow related Services
Google Cloud	Cloud Composer. https://cloud.google.com/composer/ Apache Airflow. https://airflow.apache.org/ Cloud Functions. https://cloud.google.com/functions/ Cloud Pub/Sub. https://cloud.google.com/pubsub/docs/
Microsoft Azure	Logic Apps. https://azure.microsoft.com/en-us/services/logic-apps/ Microsoft Flow. https://flow.microsoft.com/en-us/ Azure Functions. https://azure.microsoft.com/en-us/services/functions/ Event Grid. https://azure.microsoft.com/en-us/services/event-grid/ Queue Storage. https://azure.microsoft.com/en-us/services/storage/queues/
Amazon Web Services	Step Functions. https://aws.amazon.com/step-functions/ Lambda. https://aws.amazon.com/lambda/ SNS. https://aws.amazon.com/sns/ SQS. https://aws.amazon.com/sqs/

INITIAL PROJECT PLAN

	Tasks	Dates
Literature Review	Amazon Web Services	Sun 13 th - 20 th Oct
	Microsoft Azure	
	Google Cloud Platform	
Technology Assessment	Amazon Web Services	Sun 20 th - 27 th Oct
	Microsoft Azure	
	Google Cloud Platform	
Implementation Iteration 1	Create Database Schema. Implement Database CRUD code. Implement / Test Individual Services using Serverless functions and REST API.	Sun 27 th Oct - 10 th Nov
Implementation Iteration 2	Create initial Workflow orchestration and tasks. Integration testing with services from Iteration 1.	Sun 10th - 24 th Nov
Implementation Iteration 3	Create / Test User Interface. Create / Test Notification mechanisms for Human actors. Final integration testing	Sun 24 th Nov - 8 th Dec
Prepare Final Report		Sun 8th th - 15 th Dec

RISKS

- The Moodle Project schedule lists 1 week for Literature Review and 1 week for R & D of all the required services across the 3 cloud vendors. I do not believe this to be achievable.
- To build a fully functional project implementation will require a combination of many services. For example, if implemented on AWS these would include SNS, SQS, Lambda, S3, API Gateway. A browser-based frontend will be required to allow user interaction. This would require significant software development using HTML, javascript, etc perhaps using AWS frameworks such as ASW Amplify, Cognito, IAM. There is a risk that the software development effort may not be achievable in the allotted timeframe.
- Design/Architecture lacks flexibility, is not fit for purpose or is infeasible.
- Estimates are inaccurate perhaps due to scope creep or required activities are missing from scope.
- Learning curves lead to delays because developer is inexperienced.
- Requirements are incomplete or fail to align with available systems/services. User interface doesn't allow users to complete tasks or is low quality.

VERSION 1.0

12/10/2019



ALAN DUFFIN

X00159409

CERTIFICATE, CLOUD SOLUTIONS ARCHITECTURE 2019, PROJECT

FEASIBILITY AND REFINED RESEARCH PLAN

TABLE OF CONTENTS

PROJECT BACKGROUND AND DESCRIPTION	3
PROJECT DESCRIPTION.....	4
PROJECT USECASE.....	6
PROJECT DELIVERABLES	9
TECHNICAL AREAS OF INVESTIGATION	10
TECHNICAL REQUIREMENTS AND COSTS.....	12
PROJECT TIMELINES	13
RISKS AND ISSUES MANAGEMENT.....	14
RISK MITIGATION	14
REFERENCE LIST AND BIBLIOGRAPHY	15

PROJECT BACKGROUND AND DESCRIPTION

Workflow is a common pattern for solving use cases across a broad range of sectors. These use cases require maintaining state and coordinating operations across sometimes heterogeneous / distributed participants over extended time periods. Examples of common problem areas include loan/insurance origination, order management, billing systems and logistics fulfilment.

These systems have a common problem set that must be solved. The workflow pattern must address requirements such as state transition, error handling, component / service integration and human interaction/decision making.

Many vendors have designed solution frameworks around the concept of the Finite State Machine such as BPEL (Business Process Execution Language). The growth of SaaS and PaaS platforms has allowed users to take advantage of fully managed, scalable services for many problem domains including workflow.

My project proposal is to research the available Workflow/Servlerless services that are available from the three largest SaaS Cloud providers: Google Cloud, Amazon Web Services, Microsoft Azure. I will subsequently create a comparison matrix for these SaaS platforms based on a set of required criteria for the Workflow use case that is to be implemented. This matrix will be used to select a Cloud vendor platform upon which to implement the use case. Some of these criteria/features are:

- Features
- Ease of use
- Latency
- Cost
- Maintainability / Troubleshooting
- State Maintenance / Error Handling.

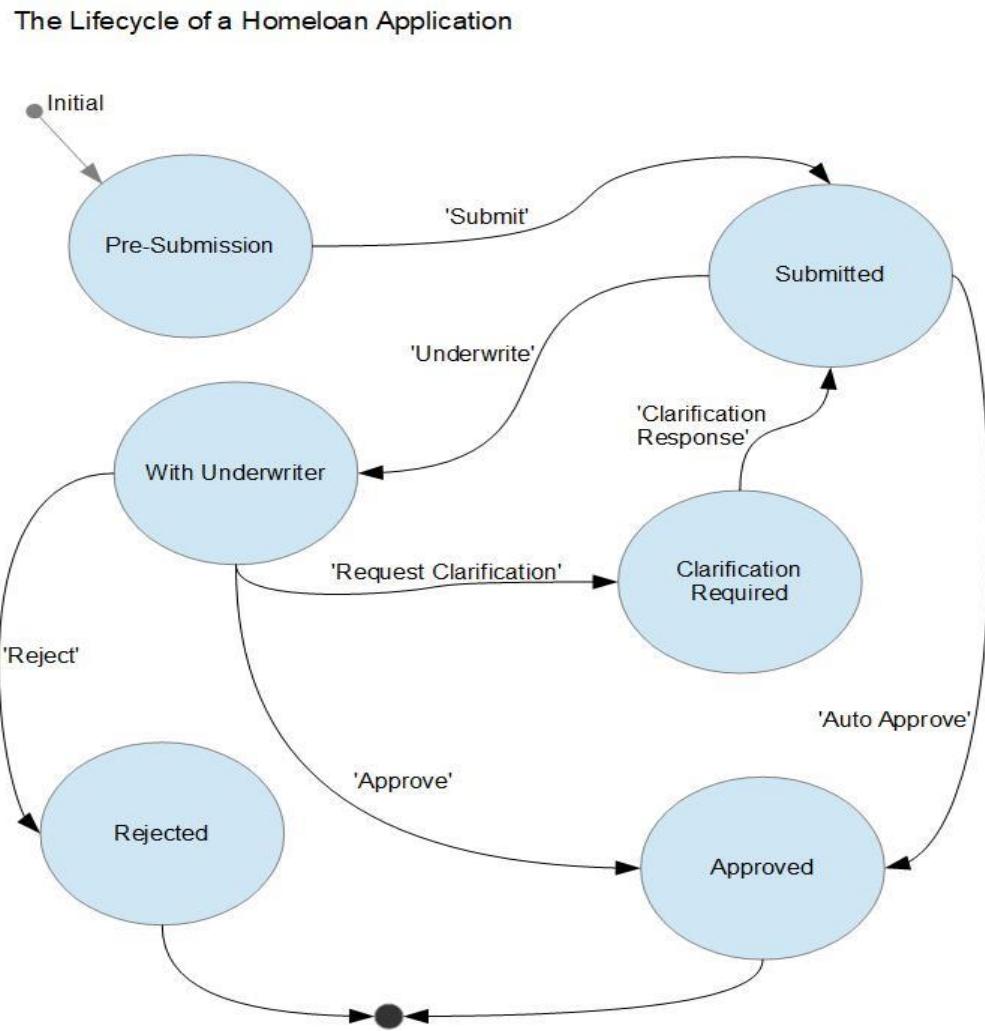
PROJECT DESCRIPTION

Loan origination is the process by which a borrower applies for a new loan, and a lender processes that application. Origination includes all the steps from taking a loan application up to drawdown of funds or refusal of the application.

The loan origination process has many steps, including:

- Loan application — The homebuyer fills out a loan application form.
- Documents — The buyer submits documentation for income, employment and financial status.
- Underwriting — The lender reviews the loan application, verifies the buyer's credit and determines if the buyer's income and financial status is such that the buyer qualifies for the loan.
- Finalise loan application — Once the terms are agreed on, the loan application is processed by the lender, going through several departments, such as underwriting and documentation processing.
- Loan approval — After final processing, the lender decides to approve or reject the loan application.

The Lifecycle of a Homeloan Application



PROJECT USECASE

Human Actors	
Branch Staff	Actor 1
Broker	Actor 2
Customer	Actor 3
Underwriter	Actor 4

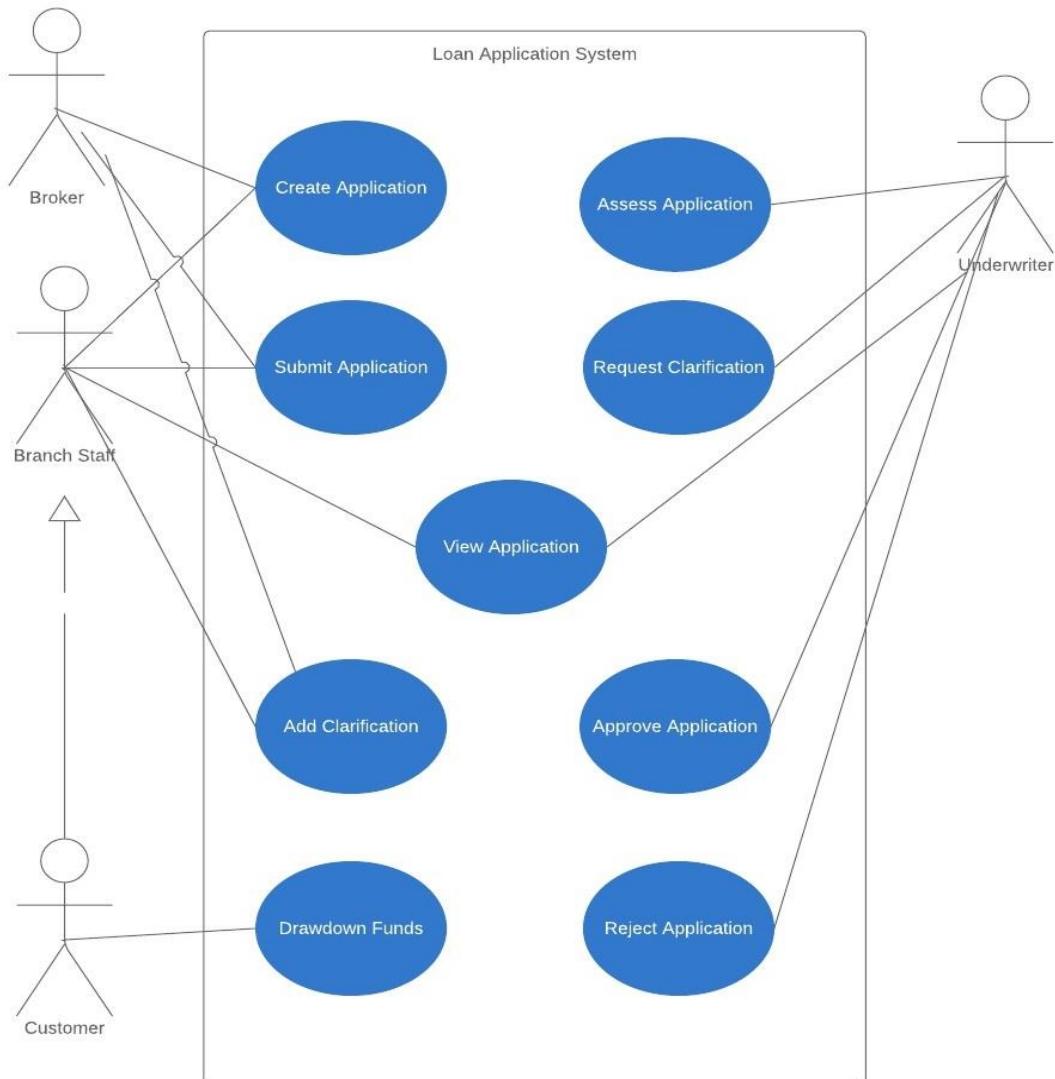
The Bank Branch Staff Member or Broker (Actor 1) via the user interface will view the list of active Loan Applications.

- Actor 1 can choose to view the details of any individual Loan Application.
- Actor 1 can create a new Loan Application on behalf of a customer.
- Actor 1 can input all required customer and loan details.
- After reviewing all information Actor 1 can submit the Loan application into the Approvals workflow. This should initiate a workflow execution.
- Possibly notifications should be sent to all human actors of loan application state changes. This could be via email, or a front end notification.
- The workflow execution should consist of the steps shown in the table below (not exhaustive).

Step Number	Workflow Component
1	Validate that all mandatory data has been input for the Approval process
2	Run the affordability calculator and validate that the case does not fall outside loan rules e.g. age of applicants, loan term, unsuitable employment, not in current employment for long enough etc.
3	Decision rule to either auto approve or route the application to an Underwriter (Actor 2) for human review.
4	The Auto Approval service will mark the application as approved and notify Actor 1. (Notification method unknown at the moment)
5	The Underwriters have discretion to approve the loan application based on extenuating circumstances such as Junior doctor with immediate prospects of salary increases and stable job prospects.
6	If manual Approval is required the workflow will notify Actor 4 that the Loan Application is awaiting review and pause execution.
7	Actor 4 must open the Loan Application and review details.
8	Actor 4 can then choose to Approve or Decline the application. This is done via the front end and calls a service that restarts the workflow execution.
9	The workflow will then notify Actor 1 and the Customer (Actor 3) using some mechanism.
10	The workflow will terminate.

Loan Origination Use Case Diagram

Alan Duffin | October 3, 2019



PROJECT DELIVERABLES

Key Deliverables	Interim Deliverables	Date Due
Initial Research Proposal		6 th October
Feasibility and Research Project Plan		13 th October
Literature Review/Research		20 th October
Technology Research and Development		27 th October
Project Implementation	Iteration 1	10 th November
	Iteration 2	24 th November
	Final Iteration	8 th December
Final Report and Presentation		15 th December

TECHNICAL AREAS OF INVESTIGATION

The following table lists the PaaS services that will be required to implement a solution for my project use case.

SaaS Provider	Services for Investigation
Google Cloud	Cloud Composer. Apache Airflow. Cloud Functions. Cloud Pub/Sub. Cloud Datastore.
Microsoft Azure	Logic Apps. Microsoft Flow. Azure Functions. Event Grid. Queue Storage. CosmosDB.
Amazon Web Services	Step Functions. Lambda. SNS. SQS. DynamoDB.

The most important services for research are obviously the workflow/orchestration PaaS services that are provided by Google Cloud Platform, AWS and Azure. These services are Google Orchestrator, AWS Step Functions and Azure Logic Apps.

Cloud Composer is a hosted solution for the open source Apache Airflow platform. It is based on a Directed Acyclic Graph which is slightly different than a State Machine that does allow cycles. It supports a hybrid cloud or multi vendor solution. Only the Python development language is supported.

AWS Step Functions provides a serverless orchestration solution. Step Functions provides pre build steps that implement basic primitives to speed development and improve reliability and reuse. Step Functions also integrate out of the box with other AWS services. It can also support a hybrid cloud model.

The Azure Logic Apps service was designed specifically by Microsoft to allow non-developers to design and build Workflows in the cloud. Logic Apps benefits from a library of hundreds of pre-built Connectors to integrate with existing Azure Services. Logic Apps also support the hybrid cloud model by allowing the connection of distributed systems across on-premises and in-cloud environments.

My research into these services will focus on criteria that will allow speed to market, ease of use, cost, ease of maintenance and diagnostic tools. Ideally the service should have out of the box support for integration with the cloud vendor NoSQL database, Notification service, Serverless components and Queue services. The initial research should estimate the usage statistics for the application in terms of throughput, storage, fixed costs. The cloud vendor Cost Calculators will then be used to estimate the cost of deployment and running the application for the remainder of the year. Should any of the Cloud vendors prove to be outliers in terms of costs that may prove to be a deciding factor in the ultimate selection decision.

TECHNICAL REQUIREMENTS AND COSTS

In order to build and test prototypes to investigate PaaS services associated with the project use case, I will require an account with each of the three Cloud vendors that are under investigation.

I will also require a local software development environment on my laptop. The three vendors do not all support a common set of programming languages for each of the services that I will be using. The only common language available seems to be Javascript. I will require a Javascript / NodeJS development environment.

Google Cloud Platform, AWS and Azure all have different pricing models even for equivalent services.

Google Cloud Platform and Azure both provide a credit voucher when customers initially subscribe. AWS provide a 12 month period where services are available for free but subject to usage limits.

Azure provide 170 euro free credit and Google Cloud Platform provide \$300 credit upon subscription.

The pricing structure for the services that are under investigation for this project are as follows:

	AWS	Azure	Google
Workflow	4,000 free state transitions p/m. First year	€0.000022 per execution.	\$280 p/m approx
Servlerless	1 M Requests free p/m	1 M Requests free p/m	2 M Requests free p/m
Queue Service	1 M Requests free p/m	100K operations free p/m	10GB free p/m
Database	25GB free p/m 200M requests free p/m	5-GB and 400 RUs provisioned throughput first year.	1 GB free p/d 50K Reads/20K writes p/d

PROJECT TIMELINES

	Tasks	Dates
Literature Review	Amazon Web Services	Sun 13 th - 20 th Oct
	Microsoft Azure	
	Google Cloud Platform	
Technology Assessment	Amazon Web Services	Sun 20 th - 27 th Oct
	Microsoft Azure	
	Google Cloud Platform	
Implementation Iteration 1	Create Database Schema. Implement Database CRUD code. Implement / Test Individual Services using Serverless functions and REST API.	Sun 27 th Oct - 10 th Nov
Implementation Iteration 2	Create initial Workflow orchestration and tasks. Integration testing with services from Iteration 1.	Sun 10th - 24 th Nov
Implementation Iteration 3	Create / Test User Interface. Create / Test Notification mechanisms for Human actors. Final integration testing	Sun 24 th Nov - 8 th Dec
Prepare Final Report		Sun 8th th - 15 th Dec

RISKS AND ISSUES MANAGEMENT

- The Moodle Project schedule prescribes one week for Literature Review and one week for R & D of all the required services across the 3 cloud vendors. I do not believe this to be achievable.
- To build a fully functional project implementation will require a combination of many services. For example, if implemented on AWS these would include SNS, SQS, Lambda, S3, API Gateway. A browser-based frontend will be required to allow user interaction. This would require significant software development using HTML, javascript, etc perhaps using AWS frameworks such as ASW Amplify, Cognito, IAM. There is a risk that the software development effort may not be achievable in the allotted timeframe.
- Design/Architecture lacks flexibility, is not fit for purpose or is infeasible.
- Estimates are inaccurate perhaps due to scope creep or required activities are missing from scope.
- Learning curves lead to delays because developer is inexperienced.
- Requirements are incomplete or fail to align with available systems/services. User interface doesn't allow users to complete tasks or is low quality.

RISK MITIGATION

The core functionality for this project is to implement a workflow orchestration for the mortgage origination use case. The individual services that need to be developed will be done on a Serverless platform. The application data will be stored on a NoSQL managed cloud database. The services will be exposed via HTTP endpoints. Ideally there would also be a web based front end to allow a user friendly interface coupled with appropriate Authentication and Authorisation controls. However, due to project deadlines this front end which is not part of the core Workflow research may be omitted.

REFERENCE LIST AND BIBLIOGRAPHY

Cloud Services FAQs

Google Cloud, Composer. <https://cloud.google.com/composer/> (10 Oct, 2019)

Apache Airflow. <https://airflow.apache.org/> (10 Oct, 2019)

Google Cloud, Functions. <https://cloud.google.com/functions/> (10 Oct, 2019)

Google Cloud, Pub/Sub. <https://cloud.google.com/pubsub/docs/> (10 Oct, 2019)

Google Cloud, Datastore. <https://cloud.google.com/datastore/> (10 Oct, 2019)

Azure, Logic Apps. <https://azure.microsoft.com/en-us/services/logic-apps/> (11 Oct, 2019)

Azure, Functions. <https://azure.microsoft.com/en-us/services/functions/> (11 Oct, 2019)

Azure, Event Grid. <https://azure.microsoft.com/en-us/services/event-grid/> (11 Oct, 2019)

Azure, Queue Storage. <https://azure.microsoft.com/en-us/services/storage/queues/> (11 Oct, 2019)

Azure, ComosDB. <https://docs.microsoft.com/en-us/azure/cosmos-db/introduction> (11 Oct, 2019)

AWS, Step Functions. <https://aws.amazon.com/step-functions/> (12 Oct, 2019)

AWS, Lambda. <https://aws.amazon.com/lambda/> (12 Oct, 2019)

AWS, SNS. <https://aws.amazon.com/sns/> (12 Oct, 2019)

AWS, SQS. <https://aws.amazon.com/sqs/> (12 Oct, 2019)

AWS, Dynamo DB. <https://aws.amazon.com/dynamodb/> (12 Oct, 2019)

Cloud Services Pricing

Google Cloud, Composer. <https://cloud.google.com/composer/pricing/> (10 Oct, 2019)

Google Cloud, Functions. <https://cloud.google.com/functions/pricing/> (10 Oct, 2019)

Google Cloud, Pub/Sub. <https://cloud.google.com/pubsub/pricing/> (10 Oct, 2019)

Google Cloud, Datastore. <https://cloud.google.com/datastore/pricing> (10 Oct, 2019)

Google Cloud, Free Limits. <https://cloud.google.com/free/> (10 Oct, 2019)

Azure, Logic Apps. <https://azure.microsoft.com/en-us/pricing/details/logic-apps/> (11 Oct, 2019)

Azure, Functions. <https://azure.microsoft.com/en-us/pricing/details/functions/> (11 Oct, 2019)

Azure, Event Grid. <https://azure.microsoft.com/en-us/pricing/details/event-grid/> (11 Oct, 2019)

Azure, Queue Storage. <https://azure.microsoft.com/en-in/pricing/details/storage/queues/> (11 Oct, 19)

Azure, ComosDB. <https://azure.microsoft.com/en-us/pricing/details/cosmos-db/> (11 Oct, 2019)

Azure, Free Limits. <https://azure.microsoft.com/en-us/free/#new-products>

AWS, Step Functions. <https://aws.amazon.com/step-functions/pricing/> (12 Oct, 2019)

AWS, Lambda. <https://aws.amazon.com/lambda/pricing/> (12 Oct, 2019)

AWS, SNS. <https://aws.amazon.com/sns/pricing/> (12 Oct, 2019)

AWS, SQS. <https://aws.amazon.com/sqs/pricing/> (12 Oct, 2019)

AWS, Dynamo DB. <https://aws.amazon.com/dynamodb/pricing/provisioned/> (12 Oct, 2019)

AWS, Free Limits. <https://aws.amazon.com/free/>

VERSION 1.0

19/10/2019



ALAN DUFFIN

X00159409

CERTIFICATE, CLOUD SOLUTIONS ARCHITECTURE 2019, PROJECT

LITERATURE REVIEW

TABLE OF CONTENTS

AREAS OF RESEARCH.....	3
WHAT IS A FINITE STATE MACHINE	4
WHAT IS SERVERLESS	5
WORKFLOW SERVICES IN THE CLOUD	10
THE FUTURE OF SERVERLESS	12
REFERENCE LIST AND BIBLIOGRAPHY	16

AREAS OF RESEARCH

The project that I have chosen to undertake will attempt to develop an implementation of a use case that relies on the following three concepts:

- 1 Serverless Computing
- 2 Workflow
- 3 Finite State Machines

In order to understand the theory behind each of these concepts and the practical benefits / disadvantages with regard to my specific use-case I will undertake to review a range of academic / peer reviewed literature on each subject.

WHAT IS A FINITE STATE MACHINE

A FSM is defined on Wikipedia as:

"A finite-state machine (FSM) or finite-state automaton (FSA, plural: automata), finite automaton, or simply a state machine, is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. " [5]

"A state is a description of the status of a system that is waiting to execute a transition. A transition is a set of actions to be executed when a condition is fulfilled or when an event is received. For example, when using an audio system to listen to the radio (the system is in the "radio" state), receiving a "next" stimulus results in moving to the next station. When the system is in the "CD" state, the "next" stimulus results in moving to the next track. Identical stimuli trigger different actions depending on the current state. [5]

In some finite-state machine representations, it is also possible to associate actions with a state:

- 1 *an entry action: performed when entering the state, and*
- 2 *an exit action: performed when exiting the state." [5]*

A Finite State Machine is a generalisation of a Directed Graph. Graphs of this type are used across the field of computer science to solve problems such as web crawling, network routing, software dependencies etc. FSMs are used to understand the behaviour of a system either a computer system or physical external system. The nodes represent the unique system states. The vertices represent the transitions move the system from one state to another. Vertices can be initiated by events and are associated with 'inputs'. It is usual that the transition will initiate some action that must be executed as the system transitions between states. Actions can also be executed during the entrance or exit from a state.

The most famous implementation of a Finite State Machine is the *Turing Machine* [7], which is an FSM that can read and write data to tape. The input that is read from the tape instructs the Turing Machine to transition into another state. A Turing Machine has one interesting difference from a State Machine - it can count because it has storage in the form of its tape.

It is this underlying concept of a set of finite states connected by transitions/actions that allows us to model complex systems and implement workflow solutions.

WHAT IS SERVERLESS

Martin Fowler, a leading author and expert on software analysis and design defines Serverless Computing as encompassing two different but overlapping areas:

- 1 "Serverless was first used to describe applications that significantly or fully incorporate third-party, cloud-hosted applications and services, to manage server-side logic and state."
- 2 "These are typically "rich client" applications that use the vast ecosystem of cloud-accessible storage, authentication services, and so on. These types of services have been previously described as "Backend as a Service" or BaaS." [1]

The term "Serverless" has become synonymous with the FaaS concept and that is what this Literature Review will cover.

Fowler also makes a significant observation about Serverless architectures which is particularly applicable to the project use-case that I have chosen to investigate. He states that whereas for traditional 3-tier applications "*all flow, control, and security was managed by the central server application. In the Serverless version there is no central arbiter of these concerns. Instead we see a preference for choreography over orchestration, with each component playing a more architecturally aware role*"[1].

While the predominant pattern for microservice/serverless applications is choreography, many applications build on Microservice/Serverless platforms will require orchestration of tasks. Services like AWS Step Functions are implemented using the concept of a State Machines which are designed to coordinate tasks and will ensure application correctness by using retries, wait timers, and rollbacks. Workflows are by their nature asynchronous and therefore should not be used to process synchronous requests.

Fowler notes the well known benefits of this approach (in common with Microservices patterns) result in systems that "are often more flexible and amenable to change, both as a whole and through independent updates to components; there is better division of concerns; and there are also some fascinating cost benefits[1]". He notes that "whether the benefits of flexibility and cost are worth the added complexity of multiple backend components is very context dependent.[1]"

Fowler also decomposes "Functions as a Service" into a set of practical characteristics [1].

- "Running backend code without managing your own server systems or your own long-lived server applications".
- Essentially FaaS replaces a long running process possibly on a physical machine with something "that doesn't need a provisioned server, nor an application that is running all the time".
- "FaaS offerings do not require coding to a specific framework or library. FaaS functions are regular applications when it comes to language and environment."
- "In a FaaS environment we upload the code for our function to the FaaS provider, and the provider does everything else necessary for provisioning resources, instantiating VMs, managing processes, etc."
- "Horizontal scaling is completely automatic, elastic, and managed by the provider."
- The "compute containers" executing your functions are ephemeral, with the FaaS provider creating and destroying them purely driven by runtime need.
- "Functions in FaaS are typically triggered by event types defined by the provider."
- "Most providers also allow functions to be triggered as a response to inbound HTTP requests"

Fowler also details FaaS restrictions under the following headings:

- 1 State: FaaS functions as transient and "any state of a FaaS function that is required to be persistent needs to be externalized"[1].
- 2 Execution duration: Each FaaS function execution is time limited. "This means that certain classes of long-lived tasks are not suited to FaaS functions without re-architecture"[1].
3. Cold Starts: Every FaaS platform must start and "initialize an instance of a function before each event" [1].

Fowler asks the question whether Serverless FaaS functions are a specialisation of PaaS - "If your PaaS can efficiently start instances in 20ms that run for half a second, then call it serverless." [1]

"The key operational difference between FaaS and PaaS is scaling" [1]. When using PaaS consideration must be given to how you scale, e.g. how many VMs are required. These considerations are completely transparent with a FaaS application. Even PaaS with some form of autoscaling will not achieve scaling at the level of individual requests.

"A true DevOps culture becomes even more important in a Serverless world since those other non-sysadmin activities still need to get done, and often it's developers who are now responsible for them. These activities may not come naturally to many developers and technical leads, so education and close collaboration with operations folk is of utmost importance." [1]

According to the Berkeley View on Serverless Computing [3], forms of Serverless first appeared in 2009. At the time there were two competing approaches to virtualization in the cloud :

“Amazon EC2 is at one end of the spectrum. An EC2 instance looks much like physical hardware, and users can control nearly the entire software stack, from the kernel upward. ... At the other extreme of the spectrum are application domain-specific platforms such as Google App Engine ... enforcing an application structure of clean separation between a stateless computation tier and a stateful storage tier. App Engine’s impressive automatic scaling and high-availability mechanisms ... rely on these constraints.” [3]

“The marketplace eventually embraced Amazon’s low-level virtual machine approach to cloud computing” [3] they posit because in the “early days of cloud computing users wanted to recreate the same computing environment in the cloud that they had on their local computers to simplify porting their workloads.” “The long list of low-level virtual machine management responsibilities inspired customers with simpler applications to ask for an easier path to the cloud for new applications.” Recognition of these needs led to a new option from Amazon in 2015 called the AWS Lambda service. [3]

The Berkeley authors agree with Fowler that Serverless computing encompasses both FaaS and BaaS offerings.

“While cloud functions—packaged as FaaS (Function as a Service) offerings — represent the core of serverless computing, cloud platforms also provide specialized serverless frameworks that cater to specific application requirements as BaaS (Backend as a Service) offerings [7]. Put simply, serverless computing = FaaS + BaaS. In our definition, for a service to be considered serverless, it must scale automatically with no need for explicit provisioning, and be billed based on usage.” [3]

Cloud functions (i.e., FaaS) provide general compute and are complemented by an ecosystem of specialized Backend as a Service (BaaS) offerings such as object storage, databases, or messaging. Specifically, a serverless application on AWS might use Lambda with S3 and DynamoDB, while an application on Google’s cloud might use Cloud Functions with Cloud Pub/Sub. [3]

The Berkeley authors provide a table of the contrasting characteristics of Serverless vs Serverless Cloud, shown below in Figure 1/Table 2 [3].

	<i>Characteristic</i>	<i>AWS Serverless Cloud</i>	<i>AWS Serverful Cloud</i>
PROGRAMMER	When the program is run	On event selected by Cloud user	Continuously until explicitly stopped
	Programming Language	JavaScript, Python, Java, Go, C#, etc. ⁴	Any
	Program State	Kept in storage (stateless)	Anywhere (stateful or stateless)
	Maximum Memory Size	0.125 - 3 GiB (Cloud user selects)	0.5 - 1952 GiB (Cloud user selects)
	Maximum Local Storage	0.5 GiB	0 - 3600 GiB (Cloud user selects)
	Maximum Run Time	900 seconds	None
	Minimum Accounting Unit	0.1 seconds	60 seconds
	Price per Accounting Unit	\$0.0000002 (assuming 0.125 GiB)	\$0.0000867 - \$0.4080000
	Operating System & Libraries	Cloud provider selects ⁵	Cloud user selects
SYSADMIN	Server Instance	Cloud provider selects	Cloud user selects
	Scaling ⁶	Cloud provider responsible	Cloud user responsible
	Deployment	Cloud provider responsible	Cloud user responsible
	Fault Tolerance	Cloud provider responsible	Cloud user responsible
	Monitoring	Cloud provider responsible	Cloud user responsible
	Logging	Cloud provider responsible	Cloud user responsible

Table 2: Characteristics of serverless cloud functions vs. serverful cloud VMs divided into programming and system administration categories. Specifications and prices correspond to AWS Lambda and to on-demand AWS EC2 instances.

The Berkeley authors define precisely three critical distinctions between Serverless and Serverful computing [2]:

1. *Decoupled computation and storage. The storage and computation scale separately and are provisioned and priced independently.*
2. *Executing code without managing resource allocation. Instead of requesting resources, the user provides a piece of code and the cloud automatically provisions resources to execute that code.*
3. *Paying in proportion to resources used instead of for resources allocated.*

“Google’s original App Engine, largely rebuffed by the market just a few years before serverless computing gained in popularity, also allowed developers to deploy code while leaving most aspects of operations to the cloud provider. We believe serverless computing represents significant innovation over PaaS and other previous models .” [3] However, modern Serverless offerings are far more advanced in several important factors: better autoscaling, strong isolation, and ecosystem support. [3]

For example they mention how the Autoscaling provided by AWS Lambda tracks the load with much more accuracy than EC2 autoscaling, so customers are billed in a more fine-grained manner of per 100 ms rather than minutes or hours. Customers are charged for execution time rather than per resource provisioned.

The Berkeley authors note that this “ensured the cloud provider had skin in the game on autoscaling, and consequently provided incentives to ensure efficient resource allocation”. [3]

Martin Fowler outlines a list of more prosaic, less thoughtful concerns with the current serverless platforms. He is concerned with the lack of standard API resulting in Vendor control / Lock In. The repetition of logic across client platforms that results for the use a only BaaS services on the server side. Multiple client side platforms will need to duplicate logic, e.g. iOS, Android, JavaScript, etc. Also Fowler shares the concern of both sets of Berkeley authors for the lack of in-server state for Serverless FaaS. [1]

WORKFLOW SERVICES IN THE CLOUD

"A workflow consists of an orchestrated and repeatable pattern of activity, enabled by the systematic organization of resources into processes that transform materials, provide services, or process information. It can be depicted as a sequence of operations, the work of a person or group, the work of an organization of staff, or one or more simple or complex mechanisms."

"From a more abstract or higher-level perspective, workflow may be considered a view or representation of real work.[3] The flow being described may refer to a document, service, or product that is being transferred from one step to another." [8]

The principle of process improvement methodologies such as Sigma 6 [9] and Lean [10] are modeled using workflows.

The American Scientific Research Journal paper *Serverless Computing and Scheduling Tasks* states "With the massive growth of data, scientists in different arenas of science, astronomy, engineering and physics are developing broad applications. These applications normally consist of dependent tasks that form workflows". [6]

"Such workflows are typically complex and extensive in nature, since they consist of large number of processes with extended number of tasks that have dependency constraints. This kind of complex applications require a high performance environment so that they can be performed within a given time constraint" [6]

"Applications workflows can be represented as a Directed Acyclic Graph (DAG), in which each computational task is denoted by a node, and the relation between the precedence-constrained tasks is denoted by directed edges"[6]

Workflow services have many advantages for certain problem domains that require task coordination.

- 1 The underlying execution is easily to model and reason as a Directed Graph of tasks.
- 2 The state of individual workflow execution processes is measurable and observable.
- 3 Workflow platforms provide logging, error handling, resilience etc
- 4 Workflow platforms manage data state and transmission between independent, individual tasks.

Workflow platforms such as Microsoft Logic Apps also allow non technical users to build (via user-friendly interfaces) moderately complex applications by organising predefined Microsoft services and resources such as Email, Authentication, Office, SQL etc into a sequence of dependent tasks.

Serverless Architectures are designed to be event driven in common with Microservices best practice. However, for many situations the solution required coordinating a series of API calls while maintaining state and this is the problem domain for the workflow pattern.

LogicApps was one of the first successful cloud workflow service, targeting developers and everyone else with a graphical workflow editor and a simple JSON definition language. It provided over 100 ready-to-use connectors to integrate with APIs, Microsoft products, databases, etc.

AWS then introduced StepFunctions, which is gaining market share but suffered initially from well documented operational problems: the execution history is lost if the workflow is amended in any way, it also has no support for workflow patterns like branching and joins. Its definition language is JSON based, simple, and concise, providing data referencing, error-handling with retries and catches.

Google Cloud Platform basically offers a managed environment wrapped around the open-source AirFlow platform called Cloud Composer. AirFlow has a good reputation and uses a Python domain specific language (DSL) for workflow definition. It has good extensibility hooks and provides good cross task communication.

AWS StepFunctions and Azure LogicApps are both charge based on state changes/function calls. Google Composer charges based on the provisioning of VM/Storage resources required to run an underlying AirFlow platform. This seems to disqualify it from the Serverless category.

THE FUTURE OF SERVERLESS

The following two academic papers from UC Berkeley reflect very different viewpoints on the current state and future of Serverless computing:

- 1 Serverless Computing: One Step Forward, Two Steps Back. J. Hellerstein. 2019
- 2 A Berkeley View on Serverless Computing. E. Jonas. 2019

Hellerstein is quite critical of the gaps in the current generation of serverless platforms and states that their design is at odds with dominant trends such as data centric and distributed computing. He believes that the current offerings will retard cloud innovation for cloud and data systems.

Hellerstein believes that the cloud today is largely used as an “outsourcing platform for standard enterprise data services, the majority of cloud services are simply multi-tenant, easier-to-administer clones of legacy enterprise data services like object storage, databases, queueing systems, and web/app servers”. [4]

“A FaaS offering by itself is of little value, since each function execution is isolated and ephemeral. Building applications on FaaS requires data management in both persistent and temporary storage, in addition to mechanisms to trigger and scale function execution.” [4]

The two steps backwards that are referred to by Hellerstein et al, are that the FaaS offerings “ignore the importance of efficient data processing and retard the development of distributed systems which at the core of most current innovations in modern computing.” [4]

Hellerstein outlines three categories of applications which he believes suit the current suite of FaaS platforms [4].

Embarrassingly parallel functions: These exploit auto-scaling features because independent requests never communicate with each other and require only small amounts of computation.

Orchestration functions: He mentions that Lambda functions are mostly used to preprocess event streams before channelling them to storage or data analytics where the “heavy lifting” of the computation is done, not by Lambda functions.

Function Composition: These are collections of functions that are composed to build applications including workflows of functions chained together via data dependencies.

Thus, he proposes that the current generation of FaaS solutions are suitable only for simple workloads of independent tasks.

The researchers criticize the FaaS offerings using these observations. [4]

FaaS is a Data-Shipping Architecture: “shipping data to code” is an architectural antipattern among system designers.

FaaS Stymies Distributed Computing: FaaS has no network addressability, so functions cooperate only by passing data through slow and expensive storage. There is currently no mechanism for thousands of cores to work together efficiently.

Discourages Open Source innovation: Most popular open source software cannot run at scale in current serverless offerings. Open-source data systems are impossible to build on current FaaS offerings which locks users into either using proprietary provider services or maintaining their own server.

Hellerstein et al, then outline three problem domains to illustrate the limitations and high costs associated with FaaS [4]:

1 **Model Training Algorithms:** Lambda is 21x slower and 7.3x more expensive than running on EC2.

2 **Low-Latency Prediction Serving via Batching:** prediction serving relies on access to specialized hardware like GPUs, which are not available through AWS Lambda. A “serverful” implementation had a per batch latency of 2.8ms—127x faster than the optimized Lambda implementation. To scale this application to 1 million messages a second, the SQS request rate alone would cost \$1,584 per hour.

3 **Distributed Computing:** Lambda forbids direct network connectivity between functions, so we are forced to try alternative solutions to achieve distributed computation.

Hellerstein believes that old enterprise products will migrate to new, more efficient cloud-based vendors but this type of application problem domain will not drive innovation in the cloud. It will discourage open source development and allow the cloud vendors increase market dominance for their proprietary solutions.

Hellerstein's colleagues at UC Berkeley Jonas et al, in their paper “A Berkeley View of Serverless Computing” share some of his observations but have a more optimistic attitude. Their research

corroborates Hellerstein's assertion that the vast majority of applications currently using Serverless platforms are the trivial web/api, data processing use cases. See Figure 2/Table 4.

Percent	Use Case
32.00%	Web and API serving
21.00%	Data Processing, e.g., batch ETL (database Extract, Transform, and Load)
17.00%	Integrating 3rd Party Services
16.00%	Internal tooling
8.00%	Chat bots e.g., Alexa Skills (SDK for Alexa AI Assistant)
6.00%	IOT

Table 4: Popularity of serverless computing use cases according to a 2018 survey [3].

Jonas et al, state that “*Customers benefit from increased programming productivity, and in many scenarios can enjoy cost savings as well, a consequence of the higher utilization of underlying servers. Even if serverless computing lets customers be more efficient, the Jevons paradox suggests that they will increase their use of the cloud rather than cut back as the greater efficiency will increase the demand by adding users.*” [3]

They identify similar use cases to Hellerstein, such as high performance databases that rely on shared memory. FaaS components run in isolation so cannot be used. Distributed databases do not share memory but rely on the assumption that nodes remain online and are directly addressable. Neither of those premises hold for Serverless components. They thus rule out implementing databases in a FaaS platform and expect it to remain BaaS.

Jonas identifies four limitations of the current generation of serverless offerings [3].

Inadequate storage for fine-grained operations. This is due to the limitations of cloud storage services. Object storage services (AWS S3, Azure Blob Storage) are scalable with cheap long-term object storage, but have high access costs and latencies. Key-value databases (DynamoDB, Google Cloud Datastore) provide high IOPS, but are expensive and can take a long time to scale up. In-memory storage instances such as Memcached or Redis are not fault tolerant and cannot autoscale.

Lack of fine-grained coordination. To support stateful applications, serverless frameworks must provide a coordination mechanism.

Poor support for Communication patterns. With VM solutions such as EC2, all tasks on the instance share the copy of the data, the communication complexity of the broadcast and aggregation is $O(N)$, where N is the number of VMs. However, with FaaS this complexity is $O(N \times K)$, where K is the number of functions per VM.

Predictable Performance. The variability of hardware resources that results from giving the cloud provider flexibility to choose the underlying server.

With regard to the second limitation mentioned above, the authors seem to have overlooked the Workflow/Orchestration services that have been implemented by Cloud vendors (such as Overlooked Step Functions, Cloud Orchestrator) to address the coordination of stateless services to build stateful applications.

Jonas et al, “predict that Serverless use will skyrocket” and that the current offerings are an “important maturation” in the evolution of cloud computing [3]. This is a divergent opinion to their colleagues Hellerstein et al, who also believe that the use of Serverless will increase but that the underlying architecture is not a maturation but a diversion.

They conclude with the following more specific predictions [3]:

- Next generation BaaS storage to allow more types of applications to adopt serverless computing. Such storage will match block storage performance and allow ephemeral and durable storage.
- Secure Serverless computing to become simpler to program.
- Billing models change to allow any application, at any scale, to cost less with serverless computing.
- Applications that cannot be developed on serverless, such as OLTP databases will be offered as services from all cloud providers.
- Serverless computing will become the default computing paradigm of the Cloud Era.[3]

REFERENCE LIST AND BIBLIOGRAPHY

- [1] Serverless Architectures. <https://martinfowler.com/articles/serverless.html>
- [2] A Berkeley View on Serverless Computing.
<https://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf>
- [3] Cloud Programming Simplified: A Berkeley View on Serverless Computing.
<https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf>
- [4] Serverless Computing: One Step Forward, Two Steps Back. UC Berkely.
<http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf>
- [5] Finite State Machine https://en.wikipedia.org/wiki/Finite-state_machine
- [6] American Scientific Journal. Serverless Computing and Scheduling Tasks in the Cloud.
https://asrjetsjournal.org/index.php/American_Scientific_Journal/article/view/3913/1403
- [7] Turing Machine. https://en.wikipedia.org/wiki/Turing_machine (15 Oct, 2019)
- [8] Workflow. <https://en.wikipedia.org/wiki/Workflow> (16 Oct, 2019)
- [9] Sigma Six. https://en.wikipedia.org/wiki/Six_Sigma (15 Oct, 2019)
- [10] Lean. https://en.wikipedia.org/wiki/Lean_thinking (16 Oct, 2019)

Cloud Services FAQs

- Google Cloud, Composer. <https://cloud.google.com/composer/> (10 Oct, 2019)
- Apache Airflow. <https://airflow.apache.org/> (10 Oct, 2019)
- Google Cloud, Functions. <https://cloud.google.com/functions/> (10 Oct, 2019)
- Google Cloud, Pub/Sub. <https://cloud.google.com/pubsub/docs/> (10 Oct, 2019)
- Google Cloud, Datastore. <https://cloud.google.com/datastore/> (10 Oct, 2019)
- Azure, Logic Apps. <https://azure.microsoft.com/en-us/services/logic-apps/> (11 Oct, 2019)
- Azure, Functions. <https://azure.microsoft.com/en-us/services/functions/> (11 Oct, 2019)
- Azure, Event Grid. <https://azure.microsoft.com/en-us/services/event-grid/> (11 Oct, 2019)
- Azure, Queue Storage. <https://azure.microsoft.com/en-us/services/storage/queues/> (11 Oct, 2019)
- Azure, ComosDB. <https://docs.microsoft.com/en-us/azure/cosmos-db/introduction> (11 Oct, 2019)

AWS, Step Functions. <https://aws.amazon.com/step-functions/> (12 Oct, 2019)

AWS, Lambda. <https://aws.amazon.com/lambda/> (12 Oct, 2019)

AWS, SNS. <https://aws.amazon.com/sns/> (12 Oct, 2019)

AWS, SQS. <https://aws.amazon.com/sqs/> (12 Oct, 2019)

AWS, Dynamo DB. <https://aws.amazon.com/dynamodb/> (12 Oct, 2019)

Cloud Services Pricing

Google Cloud, Composer. <https://cloud.google.com/composer/pricing/> (10 Oct, 2019)

Google Cloud, Functions. <https://cloud.google.com/functions/pricing/> (10 Oct, 2019)

Google Cloud, Pub/Sub. <https://cloud.google.com/pubsub/pricing/> (10 Oct, 2019)

Google Cloud, Datastore. <https://cloud.google.com/datastore/pricing> (10 Oct, 2019)

Google Cloud, Free Limits. <https://cloud.google.com/free/> (10 Oct, 2019)

Azure, Logic Apps. <https://azure.microsoft.com/en-us/pricing/details/logic-apps/> (11 Oct, 2019)

Azure, Functions. <https://azure.microsoft.com/en-us/pricing/details/functions/> (11 Oct, 2019)

Azure, Event Grid. <https://azure.microsoft.com/en-us/pricing/details/event-grid/> (11 Oct, 2019)

Azure, Queue Storage. <https://azure.microsoft.com/en-in/pricing/details/storage/queues/> (11 Oct, 19)

Azure, ComosDB. <https://azure.microsoft.com/en-us/pricing/details/cosmos-db/> (11 Oct, 2019)

Azure, Free Limits. <https://azure.microsoft.com/en-us/free/#new-products>

AWS, Step Functions. <https://aws.amazon.com/step-functions/pricing/> (12 Oct, 2019)

AWS, Lambda. <https://aws.amazon.com/lambda/pricing/> (12 Oct, 2019)

AWS, SNS. <https://aws.amazon.com/sns/pricing/> (12 Oct, 2019)

AWS, SQS. <https://aws.amazon.com/sqs/pricing/> (12 Oct, 2019)

AWS, Dynamo DB. <https://aws.amazon.com/dynamodb/pricing/provisioned/> (12 Oct, 2019)

AWS, Free Limits. <https://aws.amazon.com/free/>

VERSION 1.0

24/10/2019



ALAN DUFFIN

X00159409

CERTIFICATE, CLOUD SOLUTIONS ARCHITECTURE 2019, PROJECT

TECHNICAL ASSESSMENT

TABLE OF CONTENTS

AREAS OF INVESTIGATION	3
TECHNICAL ARCHITECTURE.....	4
AZURE SERVERLESS (FUNCTIONS AS A SERVICE)	6
AZURE NOSQL	13
AWS IAM	21
AWS SERVERLESS (FUNCTIONS AS A SERVICE).....	25
AWS NOSQL	29
AWS WORKFLOW AND ORCHESTRATION	33
REFERENCE LIST AND BIBLIOGRAPHY	42

AREAS OF INVESTIGATION

In order to begin the Implementation Phase of the project it will be necessary to undertake a preliminary technical assessment of the available Functions as a Service (FaaS), Serverless and Workflow / Orchestration services that are provided by the main Cloud vendors.

To this end, this document will outline the specific services that are available from Microsoft Azure and Amazon Web Services. To build a working system other services such as NoSQL and Authorisation/Authentication will be required and they will also be covered although in less depth as they are not the focus of the problem domain that is under investigation.

The list of Services are outlined next.

Amazon Web Services provides the following services that would be required:

- AWS Identity and Access Management
- AWS DynamoDB (NoSQL)
- AWS Lambda (Serverless)
- AWS Step Functions (Workflow)

Microsoft Azure provides the following equivalent services:

- Azure Role Based Access Control (RBAC)
- Azure Cosmos DB (NoSQL)
- Azure Functions (Serverless)
- Azure Logic Apps (Workflow)

TECHNICAL ARCHITECTURE

As described above the basic architecture for the project if implemented on the AWS platform would involve the servers shown in the architecture diagram below. (Figure AWS.ARCH.1)

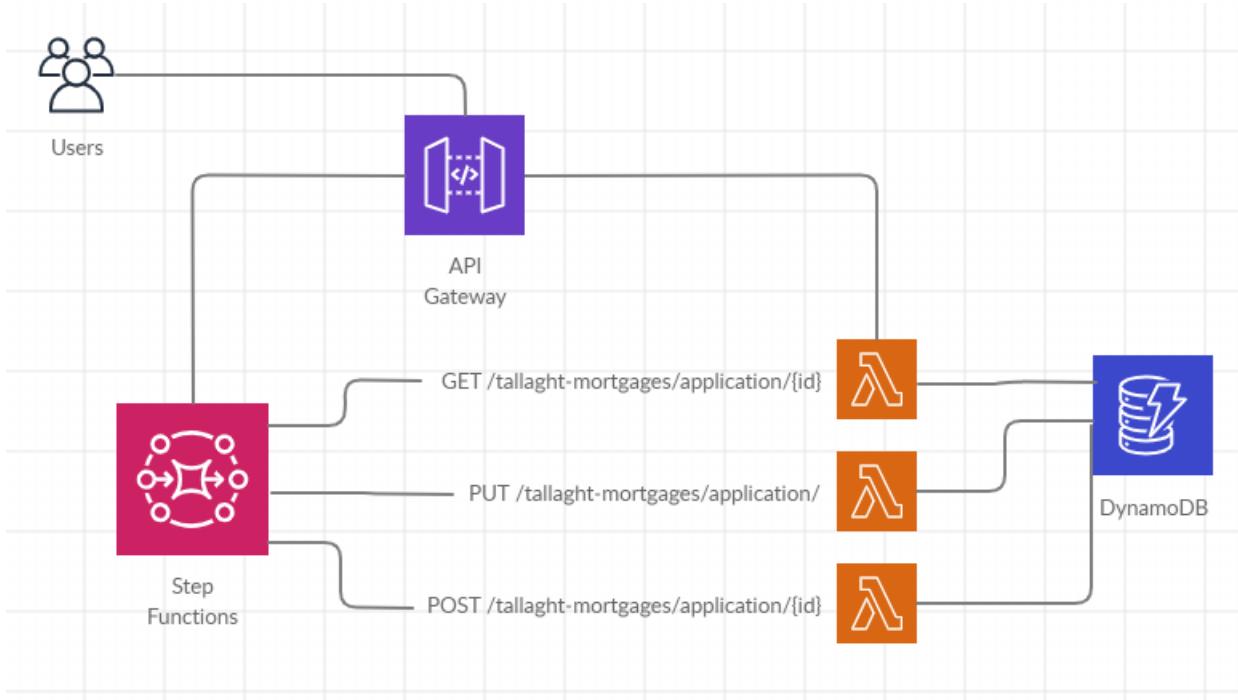


Figure AWS.ARCH.1

AZURE SERVERLESS (FUNCTIONS AS A SERVICE)

The Azure Serverless offering is called Azure Function Apps. In this section I will outline the steps that I have taken to build and deploy a simple component.

Step 1: Navigate to the Microsoft Azure Function App console and select Create. (Figure AZ.FUNC.1)

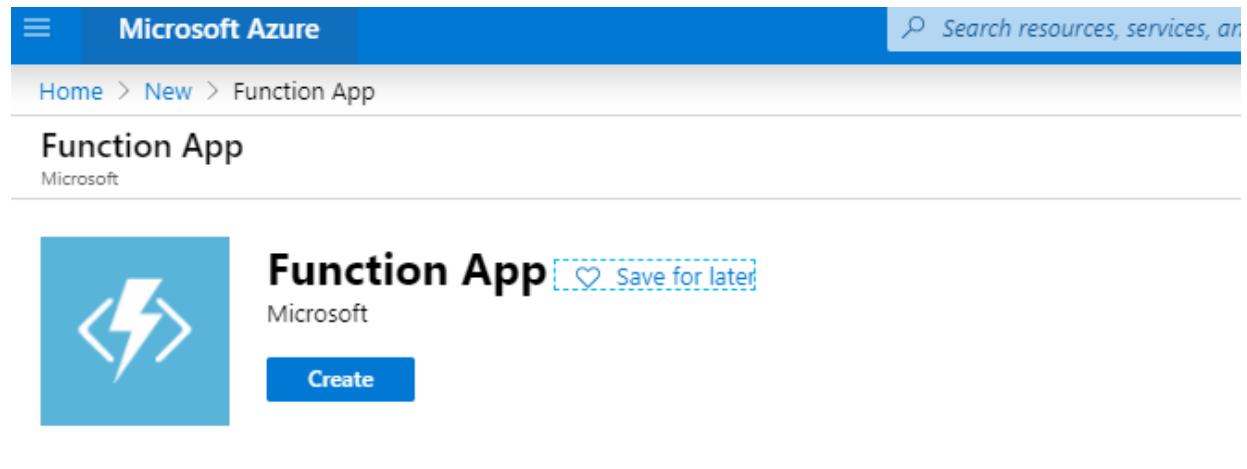


Figure AZ.FUNC.1

Step 2: Create a Resource Group and select a Function App name. (Figure AZ.FUNC.2)

The screenshot shows the Microsoft Azure portal interface for creating a new Function App. The top navigation bar includes the Microsoft Azure logo, a search bar, and a 'Search resources, services, and docs (G+)' field. Below the navigation, the breadcrumb trail shows 'Home > New > Function App > Function App'. The main title is 'Function App'. A blue info bar at the top left says 'Looking for the classic Function App create experience? →'.

Subscription * ⓘ: Free Trial

Resource Group * ⓘ: TestMortgagesResourceGroup (dropdown menu with 'Create new' option)

Instance Details

Function App name *: ReadCosmoFunction (with '.azurewebsites.net' suffix)

Publish *: Code (selected) Docker Container

Runtime stack *: Node.js

Region *: West Europe

At the bottom, there are three buttons: 'Review + create' (blue), '< Previous' (disabled), and 'Next : Hosting >'.

Figure AZ.FUNC.2

Step 3: Select the Create option. (Figure AZ.FUNC.3)

Microsoft Azure

Search resources, services, ...

Home > New > Function App > Function App

Function App

Subscription	71812486-e409-427b-8445-dd428a8f5946
Resource Group	TestMortgagesResourceGroup
Name	ReadCosmoFunction
Runtime stack	Node.js

Hosting

Storage (New)

Storage account	storageaccounttestmb407
-----------------	-------------------------

Plan (New)

Plan type	Consumption
Name	ASP-TestMortgagesResourceGroup-aac7
Operating System	Windows
Region	West Europe
SKU	Dynamic

Monitoring (New)

Application Insights	Enabled
----------------------	---------

Create < Previous Next > Download a template for automation

Figure AZ.FUNC.3

Step 4: The Function App should take a few minutes to create. (Figure AZ.FUNC.4)

The screenshot shows the Microsoft Azure portal interface. At the top, there's a blue header bar with the Microsoft Azure logo and a search bar that says "Search resources, services, and docs (G+)". Below the header, the URL "Home > Microsoft.Web-FunctionApp-Portal-94180632-90aa - Overview" is visible. The main title is "Microsoft.Web-FunctionApp-Portal-94180632-90aa - Overview". On the left, there's a sidebar with icons for Overview, Inputs, Outputs, and Template. The main content area has a "Deployment" section with a green checkmark icon and the message "Your deployment is complete". It lists the deployment name, subscription, and resource group. Below this, there's a "Deployment details" section with a "Download" link and a "Next steps" section with a "Go to resource" button.

Figure AZ.FUNC.4

Step 5: Select Go to Resource. (Figure AZ.FUNC.5)

The screenshot shows the Microsoft Azure portal interface, specifically the "TestMortgagesResourceGroup" resource group overview. At the top, there's a blue header bar with the Microsoft Azure logo and a search bar that says "Search (Ctrl+J)". Below the header, the URL "Home > Microsoft.Web-FunctionApp-Portal-94180632-90aa - Overview > ReadCosmoFunction > TestMortgagesResourceGroup" is visible. The main title is "TestMortgagesResourceGroup". On the left, there's a sidebar with icons for Overview, Activity log, Access control (IAM), Tags, Events, Settings, Quickstart, Deployments, Policies, Properties, Locks, and Export template. The main content area shows deployment details: Subscription (Free Trial), Subscription ID (71812486-e409-427b-8445-dd428a8f5946), and Tags (Click here to add tags). It also shows a table of resources with columns for Name, Type, and Status. The table includes entries for an App Service plan, App Service, Application Insights, Storage account, and Azure Cosmos DB account.

Name	Type	Status
ASP-TestMortgagesResourceGroup-aac7	App Service plan	Active
ReadCosmoFunction	App Service	Active
ReadCosmoFunction	Application Insights	Active
storageaccounttestmb407	Storage account	Active
test-mortgage-account	Azure Cosmos DB account	Active

Figure AZ.FUNC.5

Step 6: Add the Function Code. (Figure AZ.FUNC.6)

The screenshot shows the Azure Functions portal interface. On the left, there is a search bar labeled "Search by trigger, language, or description" and a dropdown menu set to "All". Below the search bar are two function templates: "HTTP trigger" and "Timer trigger". The "HTTP trigger" template is selected. On the right, a modal window titled "New Function" is open. It has fields for "Name" (set to "HttpMortgageTrigger"), "Authorization level" (set to "Function"), and "Create" and "Cancel" buttons.

Figure AZ.FUNC.6

Step 7: In order to use NodeJS modules it is necessary to login to the Kudu Configuration terminal for the Function App. (Figure AZ.FUNC.7)

The screenshot shows the Kudu Configuration terminal for a Function App. The top navigation bar includes links for "Environment", "Debug console", "Process explorer", "Tools", and "Site extensions". The main content area is divided into sections: "Environment" and "REST API". The "Environment" section displays various configuration details such as Build, Azure App Service version, Site up time, Site folder, and Temp folder. The "REST API" section lists several endpoints and their descriptions, including "App Settings", "Deployments", "Source control info", "Files", "Log streaming", "Processes and mini-dumps", "Runtime versions", "Site Extensions", "Web hooks", "WebJobs", and "Functions".

More information about Kudu can be found on the [wiki](#).

Figure AZ.FUNC.7

Step 8: Create a package.json file in the folder wwwroot/. (Figure AZ.FUNC.8)

The screenshot shows the Kudu interface with two main sections. The top section is a file browser titled "... / wwwroot" showing three items: "HttpMortgageTrigger", "host.json", and "package.json". The "package.json" file was modified on 10/26/2019, 1:08:50 PM. The bottom section is a "Kudu Remote Execution Console" window with a dark blue background. It displays a PowerShell session with the following commands:

```
Kudu Remote Execution Console
Type 'exit' then hit 'enter' to get a new powershell process.
Type 'cls' to clear the console

PS D:\home> cd site\wwwroot
cd site\wwwroot
PS D:\home\site\wwwroot>
```

Figure AZ.FUNC.8.

Step 9: Edit the code to connect to the Database server. (Figure AZ.FUNC.9)

The screenshot shows the Azure Functions portal interface. On the left, the sidebar shows the project structure: "TestCosmosReadFunction" under "Function Apps", with "HttpTrigger1" selected. The main area displays the function code:

```

var response = {
  result : true
};
mongoClient.connect("mongodb://test-mortgage-account:yeltIYtpy74R72tpQTIIcHtI7T3gOCYlHPlAvIdXOOlxmHX5XX");

if (!err) {
  context.res.status(200).json(err);
  client.close();
  return context.done();
} else {
  context.res.status(500).json({error : err});
  client.close();
  return context.done();
}
};

console.log("connected");

} catch (e) {
result = e.message;
}

```

The logs pane at the bottom shows a single log entry:

```
Console
```

The output pane shows the JSON response from the function:

```
{
  "error": {
    "name": "MongoNetworkError",
    "errorLabels": [
      "TransientTransactionError"
    ]
  }
}
```

Figure AZ.FUNC.9.

I have been unable to get the code to work successfully before the project deadline. There have been various errors such as:

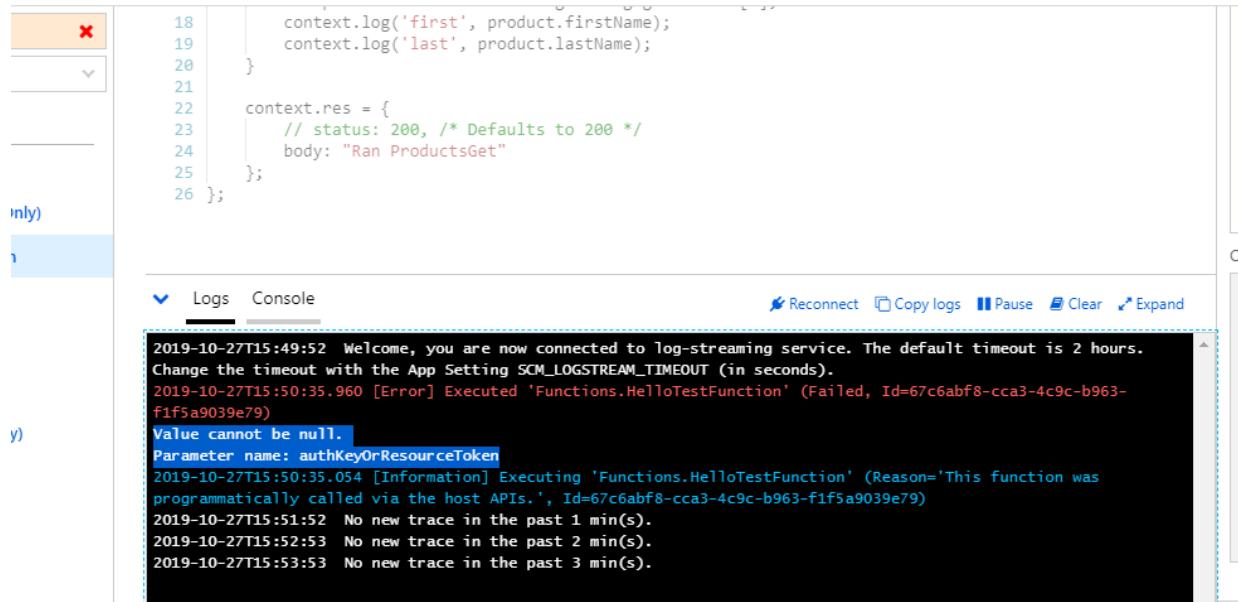
```
{
  "error": {
    "name": "MongoNetworkError",
    "errorLabels": [
      "TransientTransactionError"
    ]
  }
}
```

Other errors have include “[MongoError – Authentication Error](#)”.

I have found the Azure services, user console, development guides, logging and other features to inferior to the AWS development environment and literature.

My attempt to use the Visual Studio Code IDE and deployment tools did not succeed either. The error produced is seen in Figure AZ.FUNC.9. Apparently Azure Function bindings can only be used with SQL libraries.

n - HelloTestFunction



The screenshot shows the Azure Functions log stream interface. At the top, there is a code editor window with a file named 'n - HelloTestFunction.cs' containing C# code. Below the code editor is a log stream window titled 'Logs'. The log stream shows the following entries:

```
2019-10-27T15:49:52 Welcome, you are now connected to log-streaming service. The default timeout is 2 hours.
Change the timeout with the App Setting SCM_LOGSTREAM_TIMEOUT (in seconds).
2019-10-27T15:50:35.960 [Error] Executed 'Functions.HelloTestFunction' (Failed, Id=67c6abf8-cca3-4c9c-b963-f1f5a9039e79)
Value cannot be null.
Parameter name: authKeyOrResourceToken
2019-10-27T15:50:35.054 [Information] Executing 'Functions.HelloTestFunction' (Reason='This function was
programmatically called via the host APIs.', Id=67c6abf8-cca3-4c9c-b963-f1f5a9039e79)
2019-10-27T15:51:52 No new trace in the past 1 min(s).
2019-10-27T15:52:53 No new trace in the past 2 min(s).
2019-10-27T15:53:53 No new trace in the past 3 min(s).
```

Figure AZ.FUNC.9.

AZURE NOSQL

Step 1: Create a Cosmos DB Account. (Figure AZ.COSMOS.1)

The screenshot shows the 'Create Azure Cosmos DB Account' wizard on the 'Project Details' step. The top navigation bar includes the Microsoft Azure logo, a search bar, and a breadcrumb trail: Home > Quickstart Center > Set up a database > Create Azure Cosmos DB Account.

Project Details

Instructions: Create a new Azure Cosmos DB account with multi-region writes in North Europe, West Europe, South Central US, or North Central US by Nov 15.

Subscription *: Free Trial

Resource Group *: (New) TestMortgagesResourceGroup | [Create new](#)

Instance Details

Account Name *: test-mortgage-account

API * ⓘ: Azure Cosmos DB for MongoDB API

Apache Spark ⓘ: Notebooks Notebooks with Apache Spark [None](#) | [Sign up for Apache Spark preview](#)

Location *: (Europe) UK South

Buttons at the bottom: [Review + create](#) | [Previous](#) | [Next: Network](#)

Figure AZ.COSMOS.1

Step 2: Configure the Virtual Network and Firewall. (Figure AZ.COSMOS.2)

The screenshot shows the Microsoft Azure portal interface for creating an Azure Cosmos DB account. The top navigation bar includes the Microsoft Azure logo, a search bar, and a breadcrumb trail: Home > Quickstart Center > Set up a database > Create Azure Cosmos DB Account.

The main title is "Create Azure Cosmos DB Account". A purple callout box at the top right provides instructions: "Create a new Azure Cosmos DB account with multi-region writes in North Europe, West Europe, South Central US, or North Central US by November 1st, 2019. Get started now!"

The "Network" tab is selected in the navigation bar. The configuration section is titled "Configure Virtual Networks". It shows a "Virtual Network" dropdown set to "(new) test-mortgages-network" with a "Create a new virtual network" link below it. Under "Subnet", a dropdown is set to "(new) default (10.0.0.0/24)".

The "Configure Firewall" section contains two entries. The first entry, "Allow access from Azure Portal", has an "Allow" button highlighted in blue. The second entry, "Allow access from my IP (89.101.52.129)", also has an "Allow" button highlighted in blue.

At the bottom, there are three buttons: "Review + create" (highlighted in blue), "Previous", and "Next: Tags".

Figure AZ.COSMOS.2

Step 3: Review the Cosmos DB Account. (Figure AZ.COSMOS.3)

The screenshot shows the Microsoft Azure portal interface for creating a new Cosmos DB account. At the top, there's a blue header bar with the Microsoft Azure logo and a search bar. Below the header, a breadcrumb navigation path shows: Home > Quickstart Center > Set up a database > Create Azure Cosmos DB Account. The main title is "Create Azure Cosmos DB Account". A green validation message "Validation Success" is displayed. Below the message, there are tabs for Basics, Network, Tags, and Review + create, with "Review + create" being the active tab. The "Basics" section contains the following configuration details:

Subscription	Free Trial
Resource Group	(new) TestMortgagesResourceGroup
Location	(Europe) UK South
Account Name	(new) test-mortgage-account
API	Azure Cosmos DB for MongoDB API
Geo-Redundancy	Disable
Multi-region Writes	Disable

The "Virtual Network" section includes:

Virtual Network	(new) test-mortgages-network
Subnet	(new) default (10.0.0.0/24)

The "Firewall" section is present but currently empty.

Figure AZ.COSMOS.3.

Step 4: The deployment will take a number of minutes to complete. (Figure AZ.COSMO.4)

The screenshot shows the Microsoft Azure Deployment Overview page for a deployment named "Microsoft.Azure.CosmosDB-20191026112334". The main message is "Your deployment is underway". Deployment details include: Deployment name: Microsoft.Azure.CosmosDB-20191026112334, Subscription: Free Trial, Resource group: TestMortgagesResourceGroup. The status bar shows: Start time: 10/26/2019, 11:31:22 AM, Correlation ID: bd83dd64-17fa-482b-be35-cbbdf286c8db. A table titled "Deployment details" shows no results. A "Next steps" section is present.

Figure AZ.COSMO.4

Step 5: Once the deployment is complete, select Go to resource. (Figure AZ.COSMOS.5)

The screenshot shows the Microsoft Azure Deployment Overview page for the same deployment. The main message is "Your deployment is complete". Deployment details are identical to Figure AZ.COSMO.4. A "Go to resource" button is visible at the bottom of the page.

Figure AZ.COSMOS.5

Step 6: Select the NodeJS platform in order to obtain starter code that will allow us to connect to the database. (Figure AZ.COSMOS.6)

Congratulations! Your Azure Cosmos DB for MongoDB API account is ready.

Now, let's connect your existing MongoDB app to it:

Choose a platform

- [.NET](#)
- Node.js**
- [MongoDB Shell](#)
- [Java](#)
- [Python](#)
- [Others](#)

1 Connect your existing MongoDB .NET app

You can use your existing MongoDB .NET driver to work with Azure Cosmos DB. Make sure to enable SSL. Here is an example:

```
string connectionString =
    @"mongodb://test-mortgage-account:yelIYtpy74R72tpQTIIcHtI7T3gOCYlHP1AvIdXOOxmHXSKYLzj7Q0xMXvSKMZLAoqEgJuIKxzZhMBcpujg=@test-mortgage-account";
MongoClientSettings settings = MongoClientSettings.FromUrl(
    new MongoUrl(connectionString));
settings.SslSettings =
    new SslSettings() { EnabledSslProtocols = SslProtocols.Tls12 };
var mongoClient = new MongoClient(settings);
```

PRIMARY CONNECTION STRING

```
mongodb://test-mortgage-account:yelIYtpy74R72tpQTIIcHtI7T3gOCYlHP1AvIdXOOxmHXSKYLzj7Q0xMXvSKMZLAoqEgJuIKxzZhMBcpujg=@test-mortgage-account.doc..
```

For more details on configuring .NET driver to use SSL, follow [this article](#).

Questions? [Contact us](#)

Figure AZ.COSMOS.6

Step 7: Add a Collection to the new NoSQL database. (Figure AZ.COSMOS.7)

Add Collection

i Start at \$24/mo per database, multiple containers included
[More details](#)

- +

Estimated spend (USD): **\$0.032 hourly / \$0.77 daily** (1 region,
400RU/s, \$0.00008/RU)

* Collection id ⓘ

* Shard key ⓘ

My shard key is larger than 100 bytes

OK

Figure AZ.COSMOS.7

Step 8: Add the local IP address to the Firewall to allow access from the home device via the Web Console. (Figure AZ.COSMOS.8)

The screenshot shows the 'Firewall' settings for an Azure Cosmos DB account. At the top, there is a warning message: 'The way Data Explorer accesses your databases and containers has changed and you need to update your Firewall settings to add your current IP address to the firewall rules. For your convenience we prepared the change already, just click 'Save' to save the changes. Note that firewall changes may take up to 15min to propagate. If you need access urgently or for some reason you are unable to update firewall settings, please contact support.' Below this, there is a section titled 'Allow access from' with a radio button for 'Selected networks' which is selected. A note below says 'Configure network security for your Azure Cosmos DB account. [Learn more.](#)' Under 'Virtual networks', it shows a table with one entry:

Virtual Ne...	Subnet	Address r...	Endpoint ...	Resource ...	Subscripti...
> test-mor...	1	10.0.0.0/16	testmortg...	Free Trial	***

Below this is a 'Firewall' section with the instruction 'Add IP ranges to allow access from the internet or your on-premises networks.' and a link '+ Add my current IP (89.101.52.129)'. At the bottom are 'Save' and 'Discard' buttons.

Figure AZ.COSMOS.8

Step 9: Add a document/Item to the Collection that we can use during testing of the Azure Function.

The screenshot shows the Microsoft Azure Cosmos DB portal interface. At the top, there's a navigation bar with 'Microsoft Azure' and 'Cosmos DB > test-mortgage-account'. Below the navigation bar is a toolbar with icons for New Document, Save, Discard, and New Shell. On the left, there's a 'COLLECTIONS' sidebar with a tree view. Under 'test-mortgage-apps-db', there's a 'test-mortgage-apps-table' node with three children: 'Documents' (which is selected and highlighted in blue), 'Settings', and 'Scale'. In the main content area, there's a 'Documents' tab and a search bar that says 'Type a query predicate (e.g., {`a`:'foo'}), or choose one from the drop down list, or leave empty to query all documents.' Below the search bar is a table with two columns: '_id' and '/Applic...'. There's a single row visible with the '_id' value '10001'. The JSON content of the document is displayed on the right, numbered 1 through 7. The JSON is:

```
1 {  
2   "id" : "10001",  
3   "ApplicationID": "33218896",  
4   "firstName": "Alan",  
5   "lastName": "Duffin",  
6   "loanAmount": 250000  
7 }
```

Figure AZ.COSMOS.9.

AWS IAM

To allow another AWS service or custom code to access the datastore, in this case DynamoDB, an IAM Role needs to be created that contains the appropriate permissions. For this investigation phase we can include just Scan, Query and GetItem permissions.

Step 1: Navigate to the IAM Console.

Step 2: Select Create Role. (Figure AWS.IAM.1)

The screenshot shows the 'Create role' wizard in the AWS IAM console. Step 1 is selected (indicated by a blue circle with '1'). The page title is 'Create role'. Below it, a sub-section titled 'Select type of trusted entity' is shown. There are four options: 'AWS service' (selected), 'Another AWS account', 'Web identity', and 'SAML 2.0 federation'. Each option has a description and a 'Learn more' link. Below this, a section titled 'Choose the service that will use this role' lists services categorized by provider. The categories are EC2, Lambda, and others. Under EC2, services like API Gateway, AWS Backup, AWS Chatbot, AWS Support, Amplify, and AppStream 2.0 are listed. Under Lambda, services like CodeDeploy, Comprehend, Config, Connect, DMS, Data Lifecycle Manager, ElastiCache, Elastic Beanstalk, Elastic Container Service, Elastic Transcoder, Forecast, Lex, License Manager, Machine Learning, Macie, MediaConvert, S3, SMS, SNS, SWF, SageMaker, and Security Hub are listed.

Category	Services
EC2	API Gateway, AWS Backup, AWS Chatbot, AWS Support, Amplify, AppStream 2.0
Lambda	CodeDeploy, Comprehend, Config, Connect, DMS, Data Lifecycle Manager, ElastiCache, Elastic Beanstalk, Elastic Container Service, Elastic Transcoder, Forecast, Lex, License Manager, Machine Learning, Macie, MediaConvert, S3, SMS, SNS, SWF, SageMaker, Security Hub
Others	

Figure AWS.IAM.1

Step 3: Select the required Access Permissions. (Figure AWS.IAM.2)

Actions Specify the actions allowed in DynamoDB ⓘ

[close](#) [Filter actions](#)

[Switch to deny permissions ⓘ](#)

Manual actions (add actions)

All DynamoDB actions (dynamodb:*)

Access level

List [Expand all](#) | [Collapse all](#)

- ListBackups ⓘ
- ListGlobalTables ⓘ
- ListTables ⓘ

Read (3 selected) [Edit](#)

- BatchGetItem ⓘ
- ConditionCheckItem ⓘ
- DescribeBackup ⓘ
- DescribeContinuousBackups ⓘ
- DescribeGlobalTable ⓘ
- DescribeGlobalTableSettings ⓘ
- DescribeLimits ⓘ
- DescribeReservedCapacity ⓘ
- DescribeReservedCapacityOfferings ⓘ
- DescribeStream ⓘ
- DescribeTable ⓘ
- DescribeTimeToLive ⓘ
- GetItem ⓘ
- GetRecords ⓘ
- GetShardIterator ⓘ
- ListStreams ⓘ
- ListTagsOfResource ⓘ
- Query ⓘ
- Scan ⓘ

Tagging

Write

Resources Specify table resource ARN for the **Query** and 2 more actions. ⓘ

[Create policy](#) [Preview](#) [Edit](#) [Delete](#) [Version history](#) [AWS Lambda integration](#) [AWS Step Functions integration](#)

Figure AWS.IAM.2

Step 4: Name and Review the Policy. (Figures AWS.IAM.3, AWS.IAM.4)

Create policy

1 2

Review policy

Name*	TestDynamoAccessPolicy												
Use alphanumeric and '+, @-' characters. Maximum 128 characters.													
Description	Policy to allow Read,Scan,Query to the <u>TestTable</u>												
Maximum 1000 characters. Use alphanumeric and '+, @-' characters.													
Summary Filter													
<table border="1"> <thead> <tr> <th>Service</th> <th>Access level</th> <th>Resource</th> <th>Request condition</th> </tr> </thead> <tbody> <tr> <td>Allow (1 of 200 services) Show remaining 199</td> <td>DynamoDB</td> <td>Limited: Read</td> <td>TableName string like TestTable</td> </tr> <tr> <td></td> <td></td> <td></td> <td>None</td> </tr> </tbody> </table>		Service	Access level	Resource	Request condition	Allow (1 of 200 services) Show remaining 199	DynamoDB	Limited: Read	TableName string like TestTable				None
Service	Access level	Resource	Request condition										
Allow (1 of 200 services) Show remaining 199	DynamoDB	Limited: Read	TableName string like TestTable										
			None										

Figure AWS.IAM.3

TestDynamoAccessPolicy has been created.				
Create policy Policy actions				
Filter policies <input type="text" value="Test"/>				
	Policy name	Type	Used as	Description
<input type="radio"/>	AWSLambdaTestHarnessExecutionRole-288d9f42-...	Customer managed	Permissions policy (1)	
<input checked="" type="radio"/>	TestDynamoAccessPolicy	Customer managed	None	Policy to allow Read,Scan,Query to the TestTable

Figure AWS.IAM.4

Step 5: Add the Policy to the new Role. (Figure AWS.IAM.5)

Add permissions to AccessDynamoRole

Attach Permissions

Create policy

Filter policies ▾ test

	Policy name ▾
<input checked="" type="checkbox"/>	TestDynamoAccessPolicy



The screenshot shows the 'Attach Permissions' section of the AWS IAM console. It includes a 'Create policy' button, a 'Filter policies' dropdown set to 'test', and a table listing a single policy named 'TestDynamoAccessPolicy'. A checkbox next to the policy name is checked, indicating it is selected for attachment.

Figure AWS.IAM.5

AWS SERVERLESS (FUNCTIONS AS A SERVICE)

The AWS FaaS/Serverless offering is called AWS Lambda. In order to implement the project I will need to create a small suite of Lambda functions that will perform tasks such as reading, writing to a persistent data store, sending notifications of Mortgage Application state changes etc.

The following are the basic steps that are required in order to create a Lambda function.

Step 1: Navigate to the AWS Lambda Console. (Figure AWS.LAMBDA.1)

The screenshot shows the AWS Lambda Functions console. At the top, there is a breadcrumb navigation: Lambda > Functions. Below this, a header bar includes a search icon, an 'Actions' dropdown, and a prominent orange 'Create function' button. The main area is titled 'Functions (0)' and contains a search bar with the placeholder 'Filter by tags and attributes or search by keyword'. A table header is visible with columns: Function name, Description, Runtime, Code size, and Last modified. A message at the bottom states 'There is no data to display.'

Figure AWS.LAMBDA.1

Step 2: Select Create Function. (Figure AWS.LAMDBA.2)

The screenshot shows the 'Basic information' step of the 'Create Function' wizard. It has two main sections: 'Function name' and 'Runtime'. The 'Function name' section contains a text input field with 'TestReadFromDynamo' and a note below it: 'Enter a name that describes the purpose of your function.' The 'Runtime' section contains a dropdown menu set to 'Node.js 10.x' and a note: 'Choose the language to use to write your function.' Below these sections, there is a 'Permissions' section with a note: 'Lambda will create an execution role with permission to upload logs to Amazon CloudWatch Logs. You can configure and modify permissions further when you add triggers.' At the bottom, there is a 'Choose or create an execution role' section with three options: 'Create a new role with basic Lambda permissions', 'Use an existing role', and 'Create a new role from AWS policy templates'. The third option is selected, indicated by a blue circle.

Figure AWS.LAMBDA.2

Step 3: Use an existing IAM Role and Choose the Role from previous section. (Figure AWS.LAMBDA.3)

▼ Choose or create an execution role

Execution role
Choose a role that defines the permissions of your function. To create a custom role, go to the [IAM console](#).

Create a new role with basic Lambda permissions

Use an existing role

Create a new role from AWS policy templates

Info Role creation might take a few minutes. The new role will be scoped to the current function. To use it with other functions, you can modify it in the IAM console.

Role name
Enter a name for your new role.

Use only letters, numbers, hyphens, or underscores with no spaces.

Policy templates - optional [Info](#)
Choose one or more policy templates.

DynamoDB Lambda

Figure AWS.LAMBDA.3

Step 4: Confirmation of new Function. (Figure AWS.LAMBDA.4)

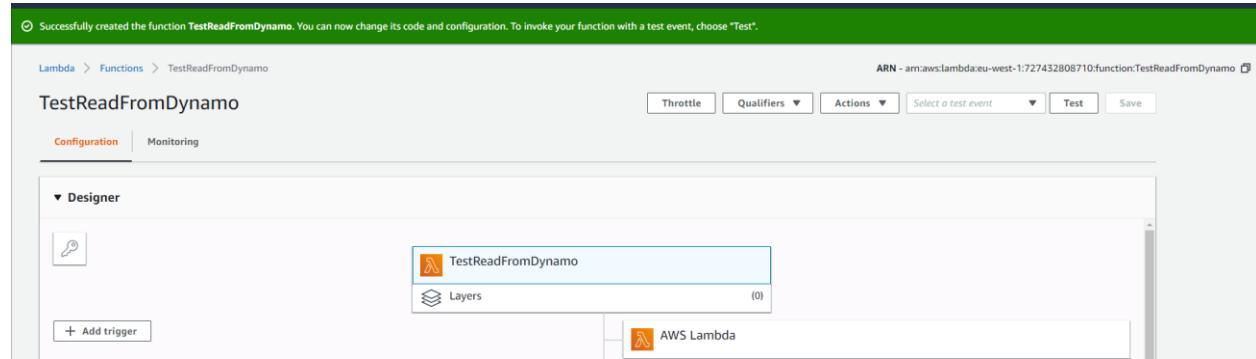


Figure AWS.LAMBDA.4

Step 4: Begin to input the source code. (Figure AWS.LAMBDA.5)

This code makes a simple call to read an item from the NoSQL data store.

The screenshot shows the AWS Lambda Test Environment interface. At the top, there are dropdown menus for 'Code entry type' (set to 'Edit code inline'), 'Runtime' (set to 'Node.js 10.x'), and 'Handler' (set to 'index.handler'). Below the header is a toolbar with standard file operations: File, Edit, Find, View, Go, Tools, Window. A gear icon is also present. The main area contains a code editor window titled 'index.js' which contains the following JavaScript code:

```
const AWS = require('aws-sdk'); // eslint-disable-line import/no-extraneous-dependencies
const dynamoDb = new AWS.DynamoDB.DocumentClient();
exports.handler = function (event, context, callback) {
  const params = {
    TableName: 'testtable',
    Key: {
      'TestID': '10001',
      'TestSortKey': '5000'
    }
  };
  dynamoDb.get(params, (error, result) => {
    if (error) {
      callback(null, {
        statusCode: error.statusCode || 501,
        headers: { 'Content-Type': 'text/plain' },
        body: 'Couldn\'t fetch the item.'
      });
    } else {
      callback(null, {
        statusCode: 200,
        body: JSON.stringify(result.Item)
      });
    }
  });
}
```

Below the code editor is an 'Execution Result' panel. It shows the 'Response' object returned by the Lambda function:

```
{ "statusCode": 200, "body": "{\"TestID\":\"10001\",\"TestSortKey\":\"5000\",\"Name\":\"Alan\"}" }
```

It also displays the 'Request ID' and 'Function Logs'. The status bar at the bottom right indicates 'Status: Succeeded | Max Memory Used: 95 MB | Time: 639.26 ms'.

Figure AWS.LAMBDA.5

Step 5: Create a Test Event. (Figure AWS.LAMBDA.6)

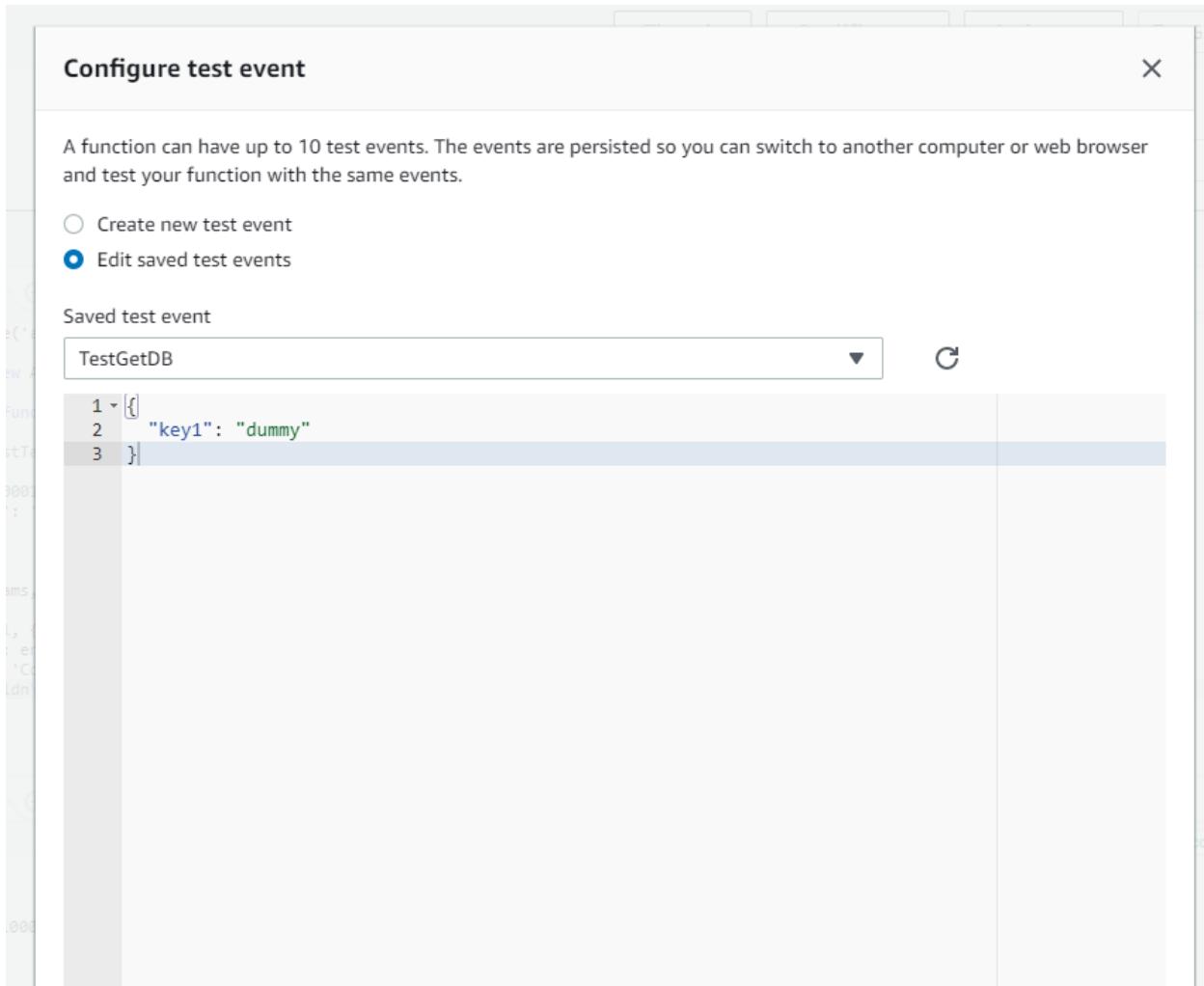


Figure AWS.LAMBDA.6

Step 6: Fire the Test Event. (Figure AWS.LAMBDA.7)

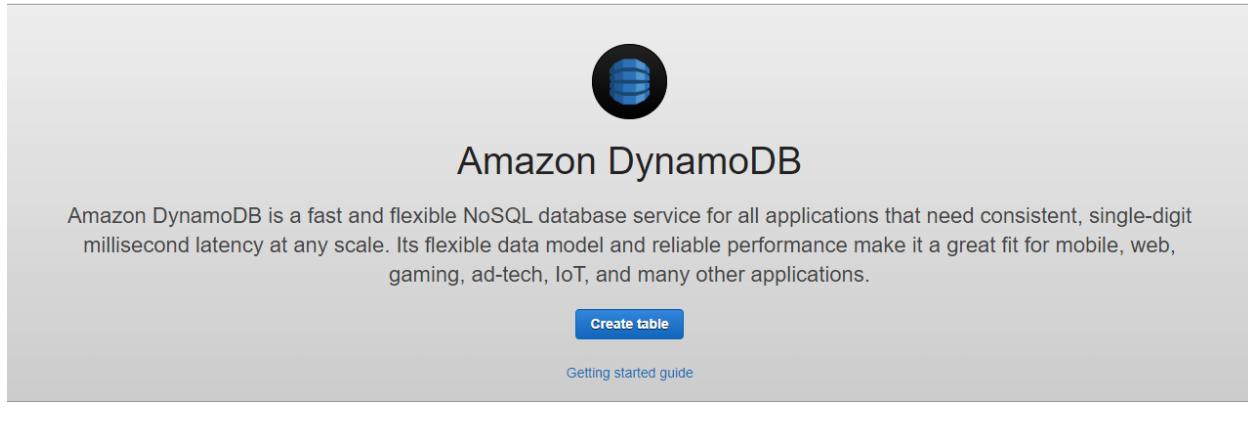


Figure AWS.LAMBDA.7

AWS NOSQL

It will be necessary to store individual Mortage Applications, therefore I require a Cloud provisioned Database. For expediency, I will investigate the most popular NoSQL services that are available from Azure and AWS. The AWS NoSQL offering is called DynamoDB. The following steps outline the simple tasks that are required to create a DynamoDB table.

Step 1: Navigate to the DynamoDB Console. (Figure AWS.DB.1)



Create tables



Add and query items



Monitor and manage

Just specify the desired read and write throughput for

Once you have created a DynamoDB table, use the AWS SDKs to write, read,

Using the AWS Management Console, you can monitor p

Figure AWS.DB.1

Step 2: Select Create Table. Figure (AWS.DB.2)

Step 3: Input a Partition Key Name and Type. The Partition Key must be an efficient Hash value that will result in effective distribution of table Items across logical partitions. Figure (AWS.DB.2)

Step 4: If required, Create a Composite Primary Key by specifying a Sort Key. Figure (AWS.DB.2)

Create DynamoDB table

DynamoDB is a schema-less database that only requires a table name and primary key. The table's primary key is made up of one or two attributes that uniquely identify items, partition the data, and sort data within each partition.

Table name* ⓘ

Primary key* Partition key

String ⓘ

Add sort key

String ⓘ

Table settings

Default settings provide the fastest way to get started with your table. You can modify these default settings now or after your table has been created.

Use default settings

- No secondary indexes.
- Provisioned capacity set to 5 reads and 5 writes.
- Basic alarms with 80% upper threshold using SNS topic "dynamodb".
- Encryption at Rest with DEFAULT encryption type.

Figure (AWS.DB.2)

Step 5: Create the Table. Wait for the Table to be Provisioned. (Figure AWS.DB.3)

The screenshot shows the AWS DynamoDB console interface. On the left, there's a sidebar with a 'Create table' button and a search/filter bar. The main area has tabs for 'Overview', 'Items', 'Metrics', etc., with 'Overview' currently selected. A message box at the top says 'Table is being created'. Below it, under 'Recent alerts', it states 'No CloudWatch alarms have been triggered for this table.' Under 'Stream details', there are fields for 'Stream enabled' (No), 'View type' (-), and 'Latest stream ARN' (-). A 'Manage Stream' button is present. The 'Table details' section contains a table of configuration parameters:

Table name	TestTable
Primary partition key	TestID (String)
Primary sort key	TestSortKey (String)
Point-in-time recovery	-
Encryption Type	DEFAULT
KMS Master Key ARN	Not Applicable
Time to live attribute	-
Table status	Creating
Creation date	October 25, 2019 at 9:15:10 AM UTC+1
Read/write capacity mode	Provisioned
Last change to on-demand mode	-
Provisioned read capacity units	5 (Auto Scaling Disabled)
Provisioned write capacity units	5 (Auto Scaling Disabled)
Last decrease time	-
Last increase time	-
Storage size (in bytes)	0 bytes
Item count	0

Figure AWS.DB.3

Step 6: Select Create Item. (Figure AWS.DB.4)

The screenshot shows the AWS DynamoDB console interface. The left sidebar shows the 'Items' tab is selected. The main area has tabs for 'Overview', 'Items', 'Metrics', etc., with 'Items' currently selected. A 'Create item' button is located at the top left of the main content area. Below it, a search bar displays 'Scan: [Table] TestTable: TestID, TestSortKey'. There are also buttons for 'Add filter' and 'Start search'. At the bottom, there are dropdown menus for 'TestID' and 'TestSortKey'.

Figure AWS.DB.4

Step 7: Input the Item values. (Figure AWS.DB.5)

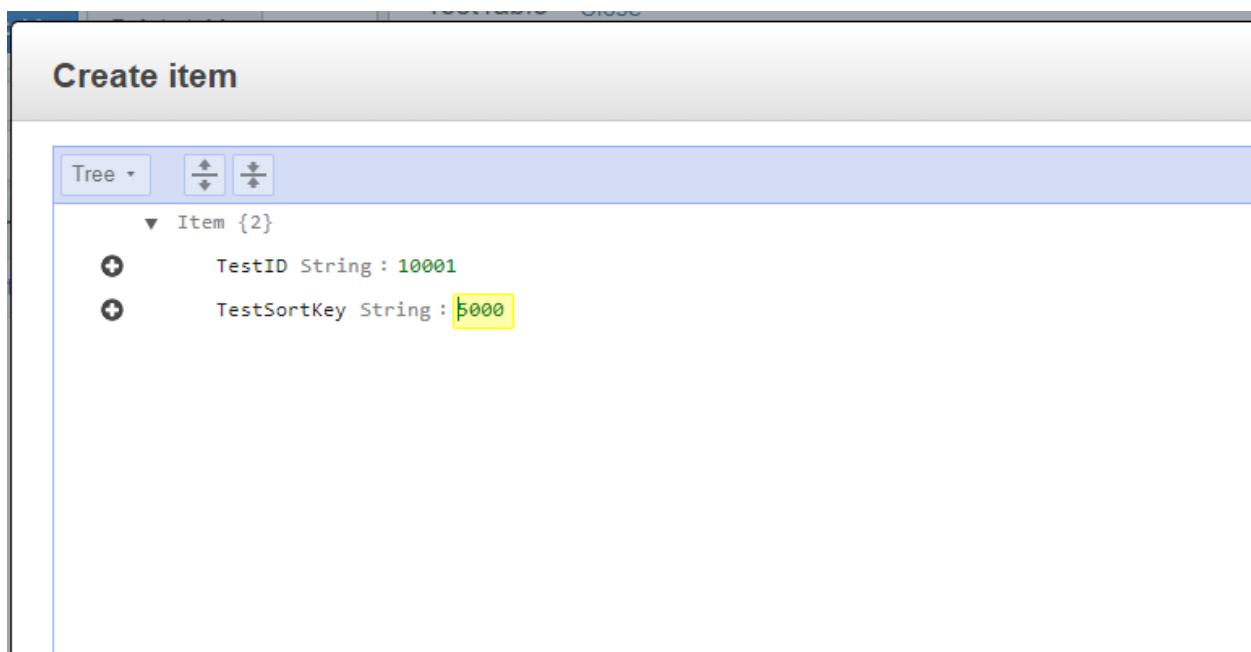


Figure AWS.DB.5

Step 8: View the Table Items. Figure AWS.DB.6

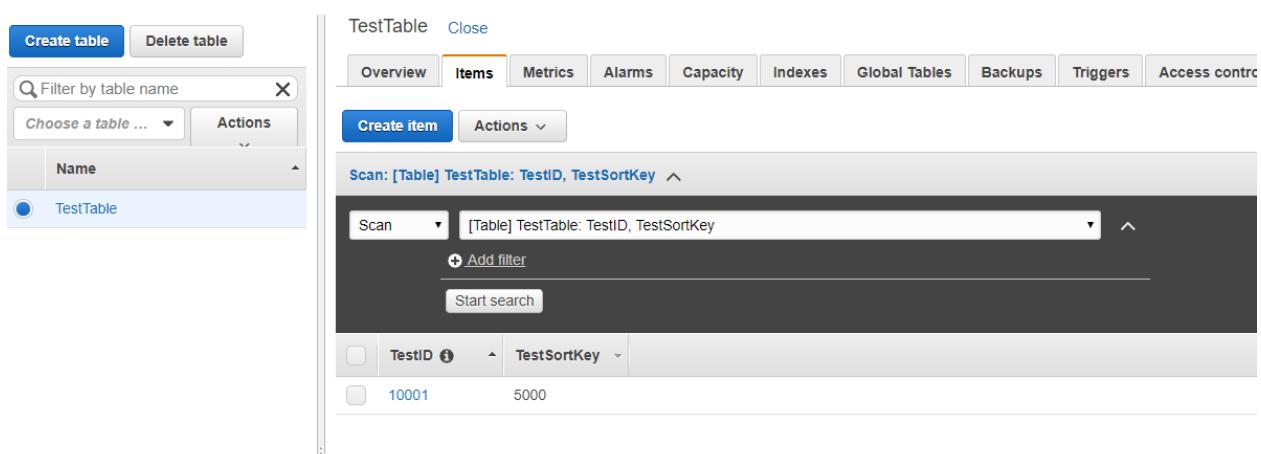


Figure AWS.DB.6

AWS WORKFLOW AND ORCHESTRATION

The AWS offering for Workflow and Orchestration is called the Step Functions service. In order to gain an understanding of the basic actions that are required to create a Workflow process using AWS Step Functions I will follow the Getting Started tutorials that are provided on the AWS website.

This small example will allow us to create a Step Function State Machine that contains a Task that will invoke the Lambda function that we created in the previous section.

Step 1: Create an IAM Role that will be used to allow the Step Function State Machine to invoke a Lambda function. (Figure AWS.STEP.1)

Create role

1 2 3 4

Select type of trusted entity

AWS service EC2, Lambda and others **Another AWS account** Belonging to you or 3rd party **Web identity** Cognito or any OpenID provider **SAML 2.0 federation** Your corporate directory

Allows AWS services to perform actions on your behalf. [Learn more](#)

Choose the service that will use this role

EC2
Allows EC2 instances to call AWS services on your behalf.

Lambda
Allows Lambda functions to call AWS services on your behalf.

API Gateway	CodeDeploy	ElastiCache	Lambda	S3
AWS Backup	Comprehend	Elastic Beanstalk	Lex	SMS
AWS Chatbot	Config	Elastic Container Service	License Manager	SNS
AWS Support	Connect	Elastic Transcoder	Machine Learning	SWF
Amplify	DMS	Elastic Load Balancing	Macie	SageMaker
AppStream 2.0	Data Lifecycle Manager	Forecast	MediaConvert	Security Hub

* Required Cancel **Next: Permissions**

Figure: AWS.STEP.1

Step 2: Attach the AWSLambdaRole Policy to the new Role. (Figure AWS.STEP.2)**▼ Attach permissions policies**

Choose one or more policies to attach to your new role.

[Create policy](#) 

Showing 20 results

	Policy name	Used as
<input type="checkbox"/>	AWSLambdaInvocation-DynamoDB	None
<input type="checkbox"/>	AWSLambdaKinesisExecutionRole	None
<input type="checkbox"/>	AWSLambdaMicroserviceExecutionRole-b78ec99a-cd0e-4729-bbff-1a430128f29d	Permissions policy (1)
<input type="checkbox"/>	AWSLambdaReadOnlyAccess	None
<input type="checkbox"/>	AWSLambdaReplicator	None
<input checked="" type="checkbox"/>	AWSLambdaRole	None

AWSLambdaRole

▶ Set permissions boundary

Figure AWS.STEP.2

Step 3: Name and Complete the Role creation. (Figure AWS.STEP.3)

Create role

Review

Provide the required information below and review this role before you create it.

Role name* Use alphanumeric and '+=.,@-' characters. Maximum 64 characters.

Role description Maximum 1000 characters. Use alphanumeric and '+=.,@-' characters.

Trusted entities AWS service: lambda.amazonaws.com

Policies AWSLambdaRole

Permissions boundary Permissions boundary is not set

No tags were added.

* Required

Cancel Previous **Create role**

Figure AWS.STEP.3

Step 4: Create a new Step Function State Machine. (Figure AWS.STEP.4)

Step Functions > State machines

State machines (0)								View details	Edit	Copy to new	Delete	Create state machine				
<input type="text"/> Search for state machines												< 1 >				
Name	Creation date	Status	Running	Succeeded	Failed	Timed out	Aborted									
No state machines																
Create state machine																

Figure AWS.STEP.4

Step 5: Select 'Author with code snippets'. (Figure AWS.STEP.5)

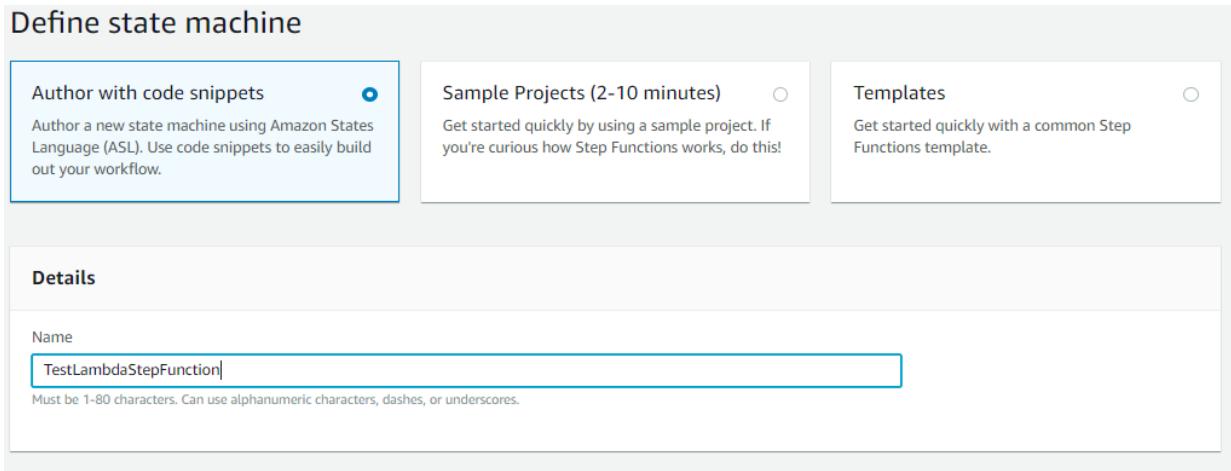


Figure AWS.STEP.5

Step 6: Define a Task state and specify the ARN for the existing Lambda. (Figure AWS.STEP.6)

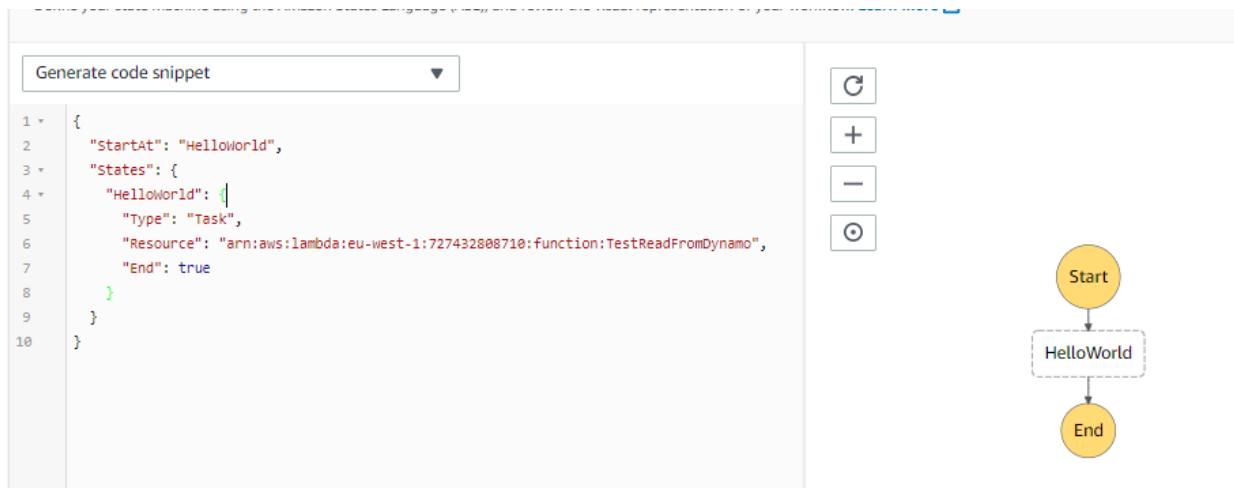


Figure AWS.STEP.6

Step 7: Choose the existing IAM Role. (Figure AWS.STEP.7)

Configure settings

IAM role for executions
The IAM role defines which AWS resources an execution is allowed to access

Create an IAM role for me
Create an IAM role based on your state machine definition

Choose an existing IAM role
Select from the list or provide a role ARN

I will use an existing role

I will provide an IAM role ARN

IAM role ARN
`arn:aws:iam::727432808710:role/TestLambdaExecutionRole`

▶ Tags - optional
A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to search and filter your resources or track your AWS costs.

Cancel Previous **Create state machine**

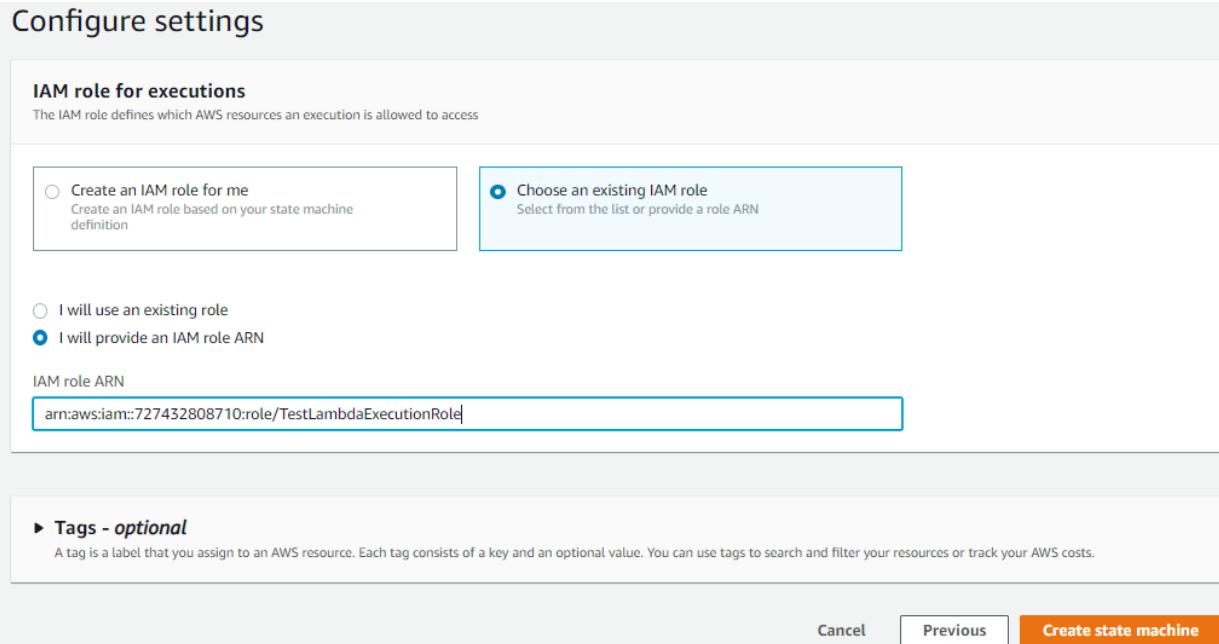


Figure AWS.STEP.7

Step 8: Create the State Machine and Save the ARN. (Figure AWS.STEP.8)

⌚ State machine successfully created X

Step Functions > State machines > TestLambdaStepFunction

TestLambdaStepFunction Edit Delete Actions ▾

Details

ARN
`arn:aws:states:eu-west-1:727432808710:stateMachine:TestLambdaStepFunction`

IAM role ARN
`arn:aws:iam::727432808710:role/TestLambdaExecutionRole`

Creation date
Oct 25, 2019 06:52:13.082 PM

Executions Definition Tags

Executions (0) C View details Stop execution Start execution

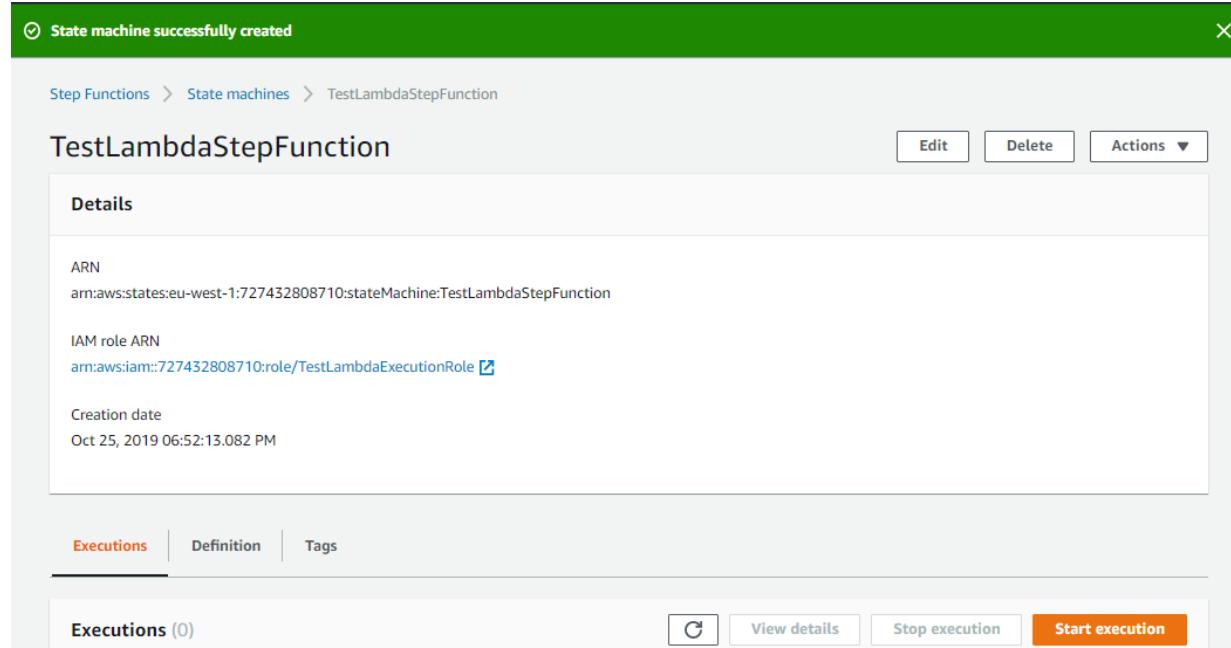


Figure AWS.STEP.8

Step 9: Test the new Step Function by creating a new Execution. (Figure AWS.STEP.9)

New execution

Start an execution using the latest definition of the state machine. [Learn more](#)

Enter an execution name - optional
Enter your execution id here
0c69f086-c832-6e5d-b7a7-9fb9f1cbd609

Input - optional
Enter input values for this execution in JSON format

```

1  {
2      "Comment": "Dummy Data"
3  }

```

Open in a new browser tab

Figure AWS.STEP.9

Step 10: The Execution has failed. (Figure AWS.STEP.10, AWS.STEP.11)

0c69f086-c832-6e5d-b7a7-9fb9f1cbd609 Edit state machine New execution

Execution details	
Execution Status	Started ✖ Failed
Execution ARN	arn:aws:states:eu-west-1:727432808710:execution:TestLambdaStepFunction:0c69f086-c832-6e5d-b7a7-9fb9f1cbd609
Input	<pre>{ "Comment": "Dummy Data" }</pre>
Output	
End Time	Oct 25, 2019 06:53:55.698 PM

Figure AWS.STEP.10

Execution event history					
ID	Type	Step	Resource	Elapsed Time (ms)	Timestamp
► 1	ExecutionStarted	-	-	0	Oct 25, 2019 06:53:55.604 PM
► 2	TaskStateEntered	HelloWorld	-	27	Oct 25, 2019 06:53:55.631 PM
► 3	LambdaFunctionScheduled	HelloWorld	Lambda CloudWatch logs	27	Oct 25, 2019 06:53:55.631 PM
► 4	LambdaFunctionStartFailed	HelloWorld	Lambda CloudWatch logs	94	Oct 25, 2019 06:53:55.698 PM
▼ 5	ExecutionFailed	-	-	94	Oct 25, 2019 06:53:55.698 PM
<pre>{ "error": "States.TaskFailed", "cause": "Neither the global service principal states.amazonaws.com, nor the regional one is authorized to assume the provided role." }</pre>					

Figure AWS.STEP.11

After investigation, the error seems to be caused due to the Policy that was specified. It does not seem to be allowed to specify an unrestricted Resource section “*”. The error was overcome by specifying the Resource to be Lambda functions scoped to the specific AWS account.

```

"Action": [
    "lambda:InvokeFunction"
],
"Resource": [
    "*"
]

"Action": [
    "lambda:InvokeFunction"
],
"Resource": [
    "arn:aws:lambda:eu-west-1:727432808710:function:*
```

Step 11: Amend the Policy Permission as shown below. (Figure AWS.STEP.12)

Policies > LambdaInvokeScopedAccessPolicy-733a2430-f6ad-4ec8-b5c3-a7bf39e88bbf

Summary

Policy ARN: arn:aws:iam::727432808710:policy/service-role/LambdaInvokeScopedAccessPolicy-733a2430-f6ad-4ec8-b5c3-a7bf39e88bbf

Description: Allow AWS Step Functions to invoke Lambda functions on your behalf

Permissions Policy usage Policy versions Access Advisor

Policy summary { } JSON Edit policy

```

1  {
2   "Version": "2012-10-17",
3   "Statement": [
4     {
5       "Effect": "Allow",
6       "Action": [
7         "lambda:InvokeFunction"
8       ],
9       "Resource": [
10      "arn:aws:lambda:eu-west-1:727432808710:function:TestReadFromDynamo"
11    ]
12  }
13 }
14 }
```

Figure AWS.STEP.12

Step 12: Retest the Step Function. (Figure AWS.STEP.13)

Step Functions > State machines > TestLambdaStepFunction > f166c3d0-6107-55ff-5beb-ee26e8d29c1f

f166c3d0-6107-55ff-5beb-ee26e8d29c1f

Edit state machine New execution Stop execution

Execution details	
Execution Status	Started Running
Execution ARN	arn:aws:states:eu-west-1:727432808710:execution:TestLambdaStepFunction:f166c3d0-6107-55ff-5beb-ee26e8d29c1f
Input	Output
End Time	-

Visual workflow

Code Step details

Figure AWS.STEP.13

Step 13: Note the Step Function Successful Output. (Figure AWS.STEP.14)

The screenshot shows the AWS Step Functions console with the following details:

- Path:** Step Functions > State machines > TestLambdaStepFunction > f166c3d0-6107-55ff-5beb-ee26e8d29c1f
- Execution ID:** f166c3d0-6107-55ff-5beb-ee26e8d29c1f
- Status:** Succeeded
- Execution ARN:** arn:aws:states:eu-west-1:727432808710:execution:TestLambdaStepFunction:f166c3d0-6107-55ff-5beb-ee26e8d29c1f
- Input:**

```
{ "Comment": "Insert your JSON here!" }
```
- Output:**

```
{
  "statusCode": 200,
  "body": "
{\\"TestID\\":\\"10001\\",\\"TestSortKey\\":\\"5000\\",\\"Name\\":\\"Alan\\"}
}"
}
```
- Start Time:** Oct 25, 2019 07:06:01.554 PM
- End Time:** Oct 25, 2019 07:06:02.961 PM

The output demonstrates that the Step Function invoked the Lambda function which responded with the DynamoDB table item.

TECHNICAL DECISIONS

My investigations of the Azure and AWS Serverless platforms as outlined in previous sections has allowed me to confirm that the AWS platform will be the cloud platform upon which I will develop this project.

In the two weeks of investigation I have formed the opinion that the developer experience and speed to market is superior on the AWS platform. In terms of ease of development and deployment, documentation, debugging, tooling I have found the AWS platform to be much more user friendly.

A percentage of this may be attributable to existing familiarity, but I have found the Azure documentation, support sites, technology stack to be below the standard I expected.

Therefore the technology resources the I will use are AWS services and open source development tools. The list is provided below.

- 1 NodeJS v13
- 2 Visual Studio Code
- 3 AWS DynamoDB
- 4 AWS Lambda
- 5 AWS Step Functions

PROJECT MILESTONES

The follow table lists the basic constituent components that are required to build the project implementation. This also give an indication of the milestone dates that should be met in order to deliver the project by the required date.

Due Date	Deliverables	
3 rd November	Workflow Design	
	CRUD Design	
	Database Schema	
10 th November	Database Build CRUD API Build	Unit Testing
17 th November	Component Services Design	
24 th November	Component Services Build	Unit + Integration Testing
1 st December	Workflow Implementation	Integration Testing
8 th December	Workflow Implementation II	Integration Testing

REFERENCE LIST AND BIBLIOGRAPHY

Cloud Services FAQs

Google Cloud, Composer. <https://cloud.google.com/composer/> (10 Oct, 2019)

Apache Airflow. <https://airflow.apache.org/> (10 Oct, 2019)

Google Cloud, Functions. <https://cloud.google.com/functions/> (10 Oct, 2019)

Google Cloud, Pub/Sub. <https://cloud.google.com/pubsub/docs/> (10 Oct, 2019)

Google Cloud, Datastore. <https://cloud.google.com/datastore/> (10 Oct, 2019)

Azure, Logic Apps. <https://azure.microsoft.com/en-us/services/logic-apps/> (11 Oct, 2019)

Azure, Functions. <https://azure.microsoft.com/en-us/services/functions/> (11 Oct, 2019)

Azure, Event Grid. <https://azure.microsoft.com/en-us/services/event-grid/> (11 Oct, 2019)

Azure, Queue Storage. <https://azure.microsoft.com/en-us/services/storage/queues/> (11 Oct, 2019)

Azure, ComosDB. <https://docs.microsoft.com/en-us/azure/cosmos-db/introduction> (11 Oct, 2019)

AWS, Step Functions. <https://aws.amazon.com/step-functions/> (12 Oct, 2019)

AWS, Lambda. <https://aws.amazon.com/lambda/> (12 Oct, 2019)

AWS, SNS. <https://aws.amazon.com/sns/> (12 Oct, 2019)

AWS, SQS. <https://aws.amazon.com/sqs/> (12 Oct, 2019)

AWS, Dynamo DB. <https://aws.amazon.com/dynamodb/> (12 Oct, 2019)

Cloud Services Pricing

Google Cloud, Composer. <https://cloud.google.com/composer/pricing/> (10 Oct, 2019)

Google Cloud, Functions. <https://cloud.google.com/functions/pricing/> (10 Oct, 2019)

Google Cloud, Pub/Sub. <https://cloud.google.com/pubsub/pricing/> (10 Oct, 2019)

Google Cloud, Datastore. <https://cloud.google.com/datastore/pricing> (10 Oct, 2019)

Google Cloud, Free Limits. <https://cloud.google.com/free/> (10 Oct, 2019)

Azure, Logic Apps. <https://azure.microsoft.com/en-us/pricing/details/logic-apps/> (11 Oct, 2019)

Azure, Functions. <https://azure.microsoft.com/en-us/pricing/details/functions/> (11 Oct, 2019)

Azure, Event Grid. <https://azure.microsoft.com/en-us/pricing/details/event-grid/> (11 Oct, 2019)

Azure, Queue Storage. <https://azure.microsoft.com/en-in/pricing/details/storage/queues/> (11 Oct, 19)

Azure, ComosDB. <https://azure.microsoft.com/en-us/pricing/details/cosmos-db/> (11 Oct, 2019)

Azure, Free Limits. <https://azure.microsoft.com/en-us/free/#new-products>

AWS, Step Functions. <https://aws.amazon.com/step-functions/pricing/> (12 Oct, 2019)

AWS, Lambda. <https://aws.amazon.com/lambda/pricing/> (12 Oct, 2019)

AWS, SNS. <https://aws.amazon.com/sns/pricing/> (12 Oct, 2019)

AWS, SQS. <https://aws.amazon.com/sqs/pricing/> (12 Oct, 2019)

AWS, Dynamo DB. <https://aws.amazon.com/dynamodb/pricing/provisioned/> (12 Oct, 2019)

AWS, Free Limits. <https://aws.amazon.com/free/>

Dublin Technological University,
Certificate, Cloud Solutions Architecture (2020)

Implementation Iteration 1

Student Name: Alan Duffin

Student ID: X00159409

ITERATION 1 TASKS

Project Phase	Service / Area	Implementation Task
Design / Planning	AWS Step Functions	Design Initial Workflow / State Machine
	AWS Lambda	Design API/Services required for Workflow / Application tasks
	AWS DynamoDB	Design Database Schema
Build / Development	Development Environment Toolkit /	Create AWS Free Tier Account Install Developer IDE / SDK / CLI
Build / Development	AWS DynamoDB	Build CloudFormation template. Database Tables / Secondary Indexes
Build / Development	AWS Lambda	Build Serverless (SAM) template for customer / mortgage services.
Test		Test CRUD / Component Services + Database Interaction
Build / Development	AWS Step Functions	Build CloudFormation template for initial Workflow.
Test		Test State Transitions / Events
Build / Development	AWS SNS	Build SNS Topics and Subscriptions.
Test		Test Notification Events and Subscriptions

WORKFLOW DESIGN

The central objective of this project is to build a solution for long – running applications that require human interaction / approval. The specific problem domain is the mortgage application / approval process. The implementation platform chosen as suitable for this type of application is the Serverless Application Model. This will involve development of a suite of AWS Lambda services that will act as tasks within the Workflow. Figure WORK.1

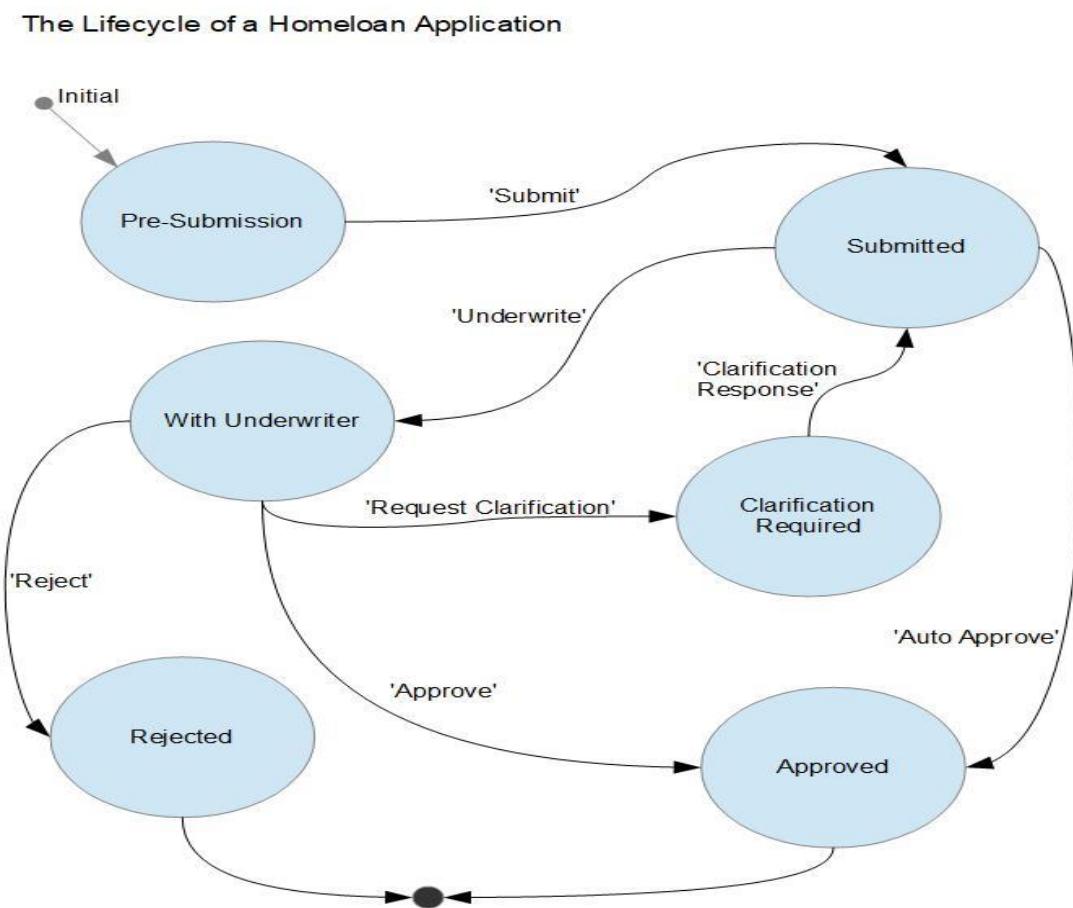


Figure WORK.1

SERVICES DESIGN

In order to build the workflow it is first necessary to put in place the substrate infrastructure / components that will be used as tasks within the workflow orchestration. The services are outlined below.

Service	URI	ACTION
Create Customer	/customer/	PUT
Get Customer	/customer/{id}	GET
Update Customer	/customer/{id}	POST
Create Mortgage	/mortgage/	PUT
Get Mortgage	/mortgage/{id}	GET
List Mortgages by State	/mortgage/status={statusText}	GET
Update Mortgage	/mortgage/{id}	POST
Mortgage Underwriter Notification	/mortgage/{morgageld}/underwriter/	POST
Mortgage Branch Notification	/mortgage/{morgageld}/branch/	POST
Mortgage Affordability Calculation	/mortgage/{morgageld}/afforability	POST
Mortgage Approval	/mortgage/{morgageld}/approve	POST
Submit Mortgage	/mortgage/{morgageld}/submit	POST

DATABASE SCHEMA DESIGN

The data model for this project consists of a Customer and Mortgage. There is a Many to One relationship between customers and mortgage application objects. In order to allow users to search mortgages that require action, it is necessary to allow query the mortgage table by status. This requires a Global Secondary Index on the Mortgage table that is indexed by the *mortgageStatus* attribute. See Figure DB.1

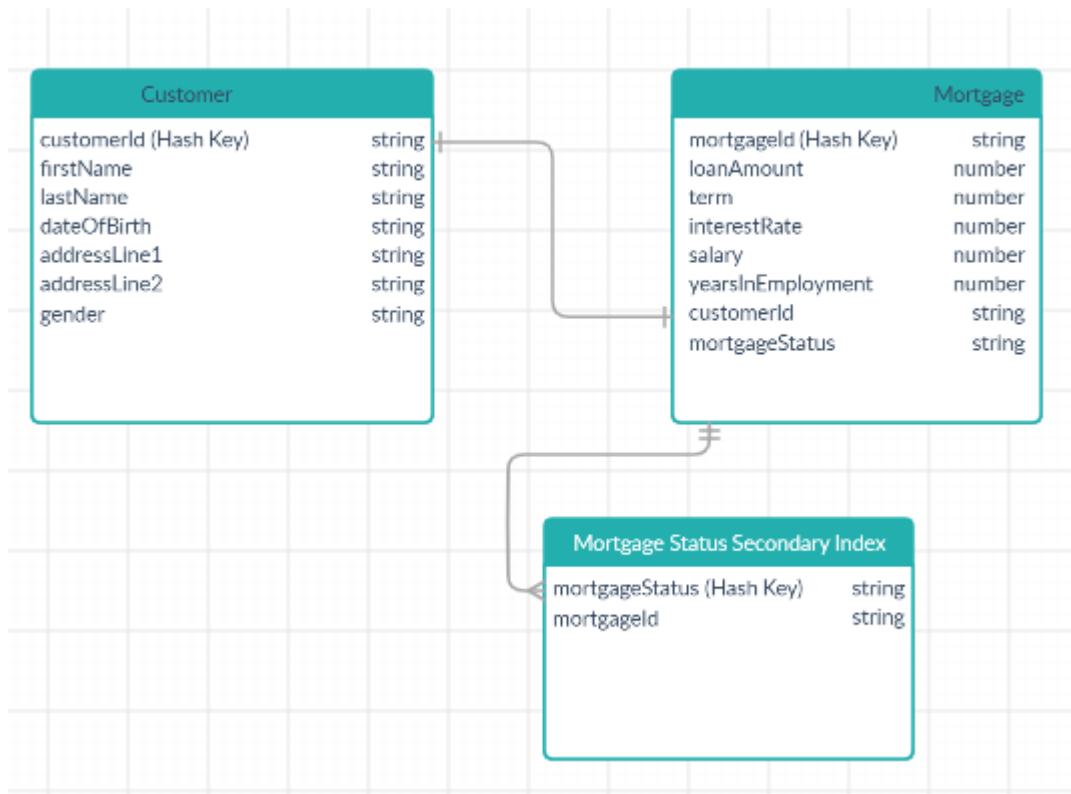


Figure DB.1

DEVELOPMENT ENVIRONMENT – AWS ACCOUNT

Navigate to the URL listed [1]. Follow the AWS instructions.

DEVELOPMENT ENVIRONMENT – AWS CLI

Navigate to the URL listed [2]. Follow the AWS instructions.

DEVELOPMENT ENVIRONMENT – AWS CLI SAM

Navigate to the URL listed [3]. Follow the AWS instructions.

DEVELOPMENT ENVIRONMENT – NODE JS / NPM

Navigate to the URL listed [4]. Follow the installation instructions.

DEVELOPMENT ENVIRONMENT – CONFIGURE AWC CLI

Create an AWS IAM Administrator Role. Figure IAM.1

The screenshot shows the AWS IAM Roles page. At the top left, there is a breadcrumb navigation: Roles > appd.aws.temporal.cli.role. Below it, a summary table provides details about the role:

Role ARN	arn:aws:iam::727432808710:role/appd.aws.temporal.cli.role
Role description	Allow CLI to assume temp Administrator role Edit
Instance Profile ARNs	[Edit]
Path	/
Creation time	2019-05-28 18:35 UTC
Maximum CLI/API session duration	1 hour Edit
Give this link to users who can switch roles in the console	https://signin.aws.amazon.com/switchrole?roleName=appd.aws.temporal.cli.role&account=appd-aws2 [Edit]

Below the summary, there are tabs for Permissions, Trust relationships, Tags (1), Access Advisor, and Revoke sessions. The Permissions tab is selected. Under the Permissions section, there is a heading "Permissions policies (1 policy applied)" followed by a button labeled "Attach policies".

Policy name	Policy type
AdministratorAccess	AWS managed policy

Figure IAM.1

Create an AWS IAM Group for Developers. Assign Policy to allow this Group to perform temporary assume-role of the Administrator Role. Figure IAM.2

The screenshot shows the AWS IAM Groups page. At the top, there's a navigation bar with 'Resource Groups' and a bell icon. Below it, the path 'IAM > Groups > AD.AWS.DEVELOPERS' is shown. A 'Summary' section displays the following details:

Group ARN:	arn:aws:iam::727432808710:group/AD.AWS.DEVELOPERS
Users (in this group):	2
Path:	/
Creation Time:	2019-05-28 18:27 UTC

Below the summary, there are three tabs: 'Users' (disabled), 'Permissions' (selected), and 'Access Advisor'. Under the 'Permissions' tab, the 'Managed Policies' section contains a table:

Policy Name	Actions
ad.aws.developer.assume.admin	Show Policy Detach Policy Simulate Policy

Figure IAM.2

Create an IAM Developer User and Assign to the Developer Group. Figure IAM3.

Users > ad.aws.developer1

Summary

The screenshot shows the 'Summary' tab for the user 'ad.aws.developer1'. It displays the User ARN (arn:aws:iam::727432808710:user/ad.aws.developer1), Path (/), and Creation time (2019-05-28 18:25 UTC). Below this, there are tabs for 'Permissions', 'Groups (1)', 'Tags (1)', 'Security credentials', and 'Access Advisor'. The 'Groups (1)' tab is selected, showing a list with one item: 'AD.AWS.DEVELOPERS' which has the attached permission 'ad.aws.developer_ASSUME_ADMIN'. A blue button labeled 'Add user to groups' is visible.

Figure IAM3.

DEVELOPMENT ENVIRONMENT – CONFIGURE AWS CLI

See instructions [5]. Using the Access Key ID and Secret Access Key downloaded when creating the Developer user we now must configure the CLI.

DEVELOPMENT ENVIRONMENT – INSTALL VISUAL STUDIO CODE

Navigate to the URL listed [6]. Follow the installation instructions.

DEVELOPMENT ENVIRONMENT – CONFIGURE VISUAL STUDIO CODE

Navigate to the URL listed [7]. Follow the installation instructions. Figure VS.1

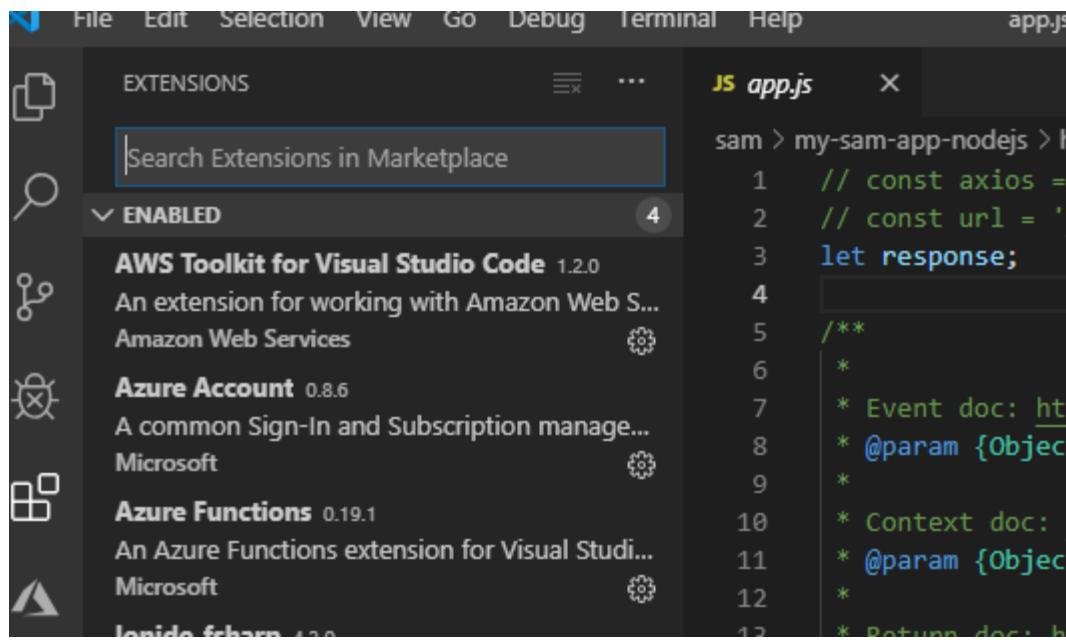


Figure VS.1

DEVELOPMENT ENVIRONMENT – DEPLOY SAMPLE SAM – COMMAND LINE

In order to test the toolchain I tried to deploy a sample application from the Command Line. Figure SAM.CMD.1

```

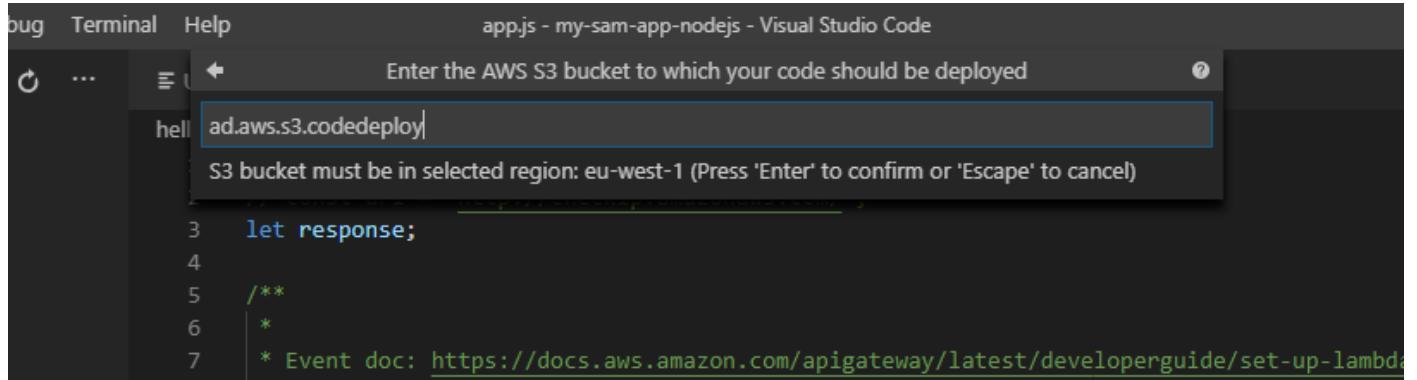
C:\Users\User\aws\sam\git\tallaght.mortgages\tud.mortgages\Tallaght-Mortgages-App>sam package --s3-bucket ad.avs.s3.codedeploy --output-template-file out.template.yaml
Successfully packaged artifacts and wrote output template to file out.template.yaml.
Execute the following command to deploy the packaged template
aws cloudformation deploy --template-file C:\Users\User\aws\git\tallaght.mortgages\tud.mortgages\Tallaght-Mortgages-App\out.template.yaml --stack-name <YOUR STACK NAME>
C:\Users\User\aws\sam\git\tallaght.mortgages\tud.mortgages\Tallaght-Mortgages-App>aws cloudformation deploy --template-file out.template.yaml --stack-name Mortgages-Stack --capabilities CAPABILITY_IAM
Waiting for changeset to be created..
No changes to deploy. Stack Mortgages-Stack is up to date
C:\Users\User\aws\sam\git\tallaght.mortgages\tud.mortgages\Tallaght-Mortgages-App>

```

Figure SAM.CMD.1

DEVELOPMENT ENVIRONMENT – DEPLOY SAMPLE SAM – VS CODE

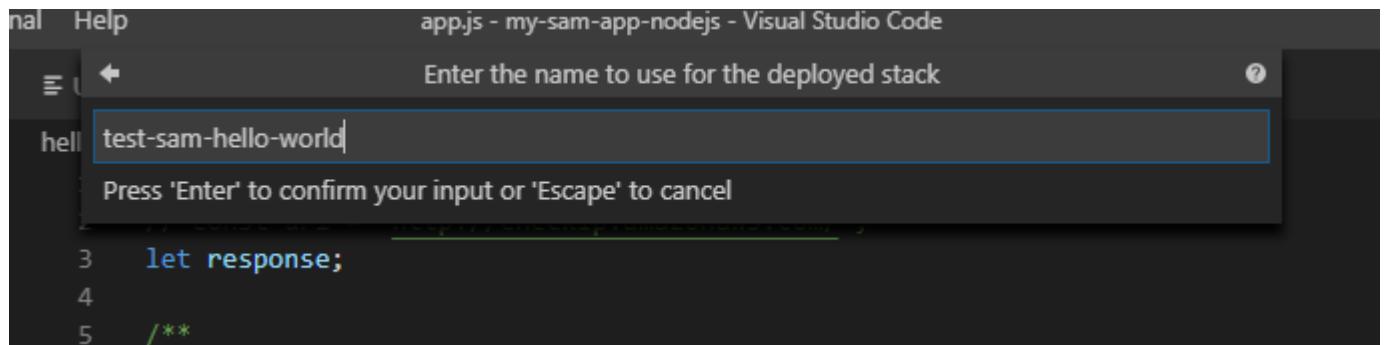
In order to test the toolchain and IDE I tried to deploy a sample application from the Visual Studio Code IDE. Figure SAM.VSC.1, Figure SAM.VSC.2.



The screenshot shows the Visual Studio Code interface with a dark theme. A modal dialog box is open in the center. The title bar of the dialog says "Enter the AWS S3 bucket to which your code should be deployed". Inside the dialog, the text "ad.aws.s3.codedeploy" is typed into a input field. Below the input field, a message says "S3 bucket must be in selected region: eu-west-1 (Press 'Enter' to confirm or 'Escape' to cancel)". The background of the code editor shows some JavaScript code:

```
3 let response;
4
5 /**
6 *
7 * Event doc: https://docs.aws.amazon.com/apigateway/latest/developerguide/set-up-lambda.html
```

Figure SAM.VSC.1

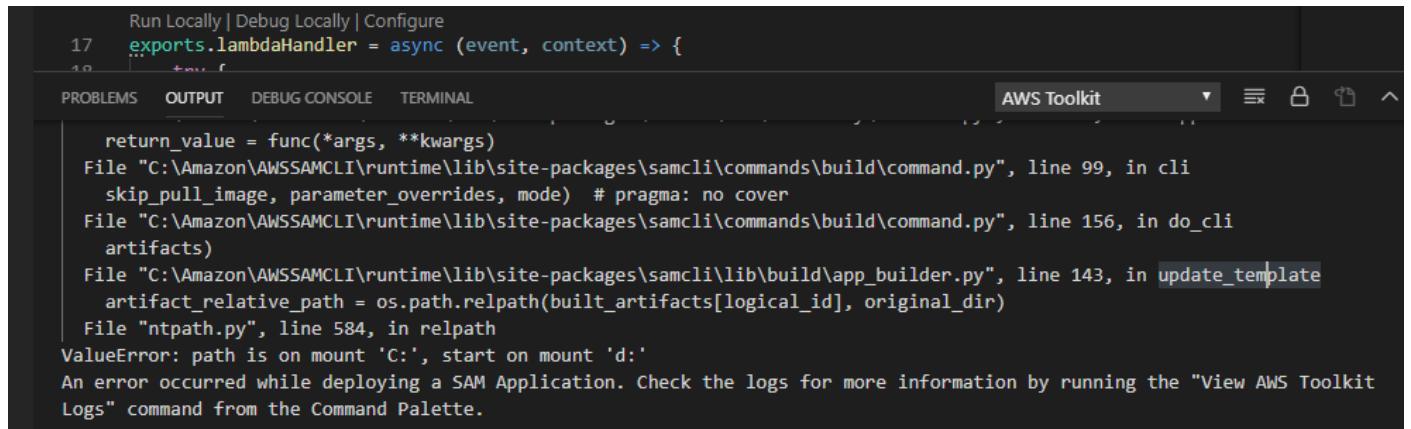


The screenshot shows the Visual Studio Code interface with a dark theme. A modal dialog box is open in the center. The title bar of the dialog says "Enter the name to use for the deployed stack". Inside the dialog, the text "test-sam-hello-world" is typed into a input field. Below the input field, a message says "Press 'Enter' to confirm your input or 'Escape' to cancel". The background of the code editor shows some JavaScript code:

```
3 let response;
4
5 /**
```

Figure SAM.VSC.2

My first attempt to deploy the code from the IDE failed. Figure SAM.VSC.3



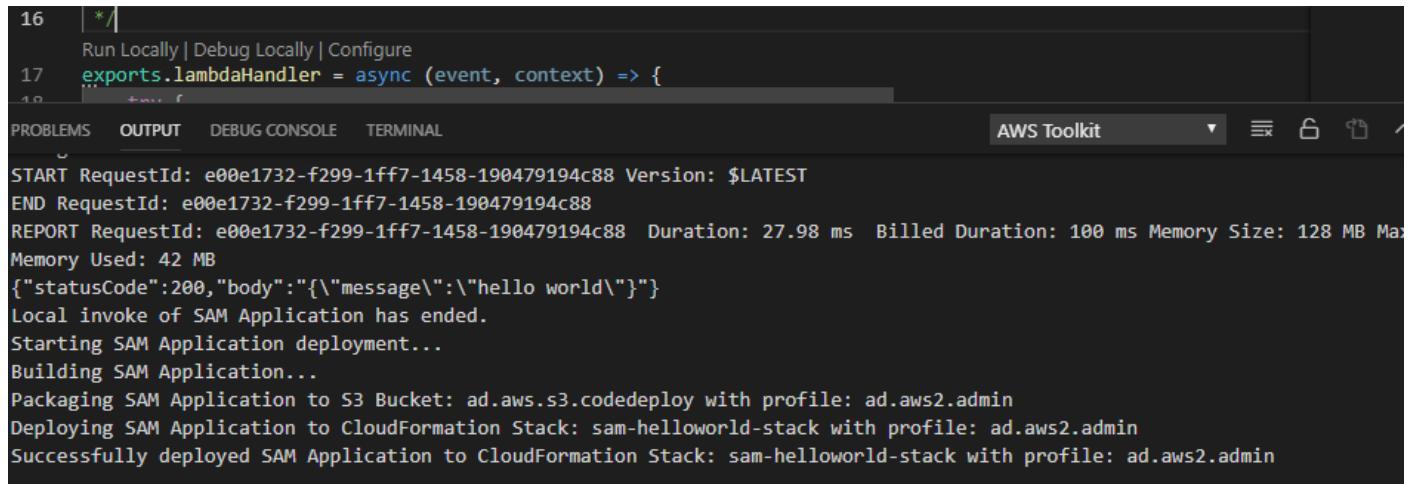
A screenshot of the VS Code interface showing a deployment error. The top bar includes 'Run Locally | Debug Locally | Configure' and the AWS Toolkit extension icon. The code editor shows a partial lambda function definition:

```
17 exports.lambdaHandler = async (event, context) => {  
18     ...  
19     return_value = func(*args, **kwargs)  
File "C:\Amazon\AWSSAMCLI\runtime\lib\site-packages\samcli\commands\build\command.py", line 99, in cli  
    skip_pull_image, parameter_overrides, mode) # pragma: no cover  
File "C:\Amazon\AWSSAMCLI\runtime\lib\site-packages\samcli\commands\build\command.py", line 156, in do_cli  
    artifacts)  
File "C:\Amazon\AWSSAMCLI\runtime\lib\site-packages\samcli\lib\build\app_builder.py", line 143, in update_template  
    artifact_relative_path = os.path.relpath(built_artifacts[logical_id], original_dir)  
File "ntpath.py", line 584, in relpath  
ValueError: path is on mount 'C:', start on mount 'd:'  
An error occurred while deploying a SAM Application. Check the logs for more information by running the "View AWS Toolkit Logs" command from the Command Palette.
```

The output panel shows the error message: 'ValueError: path is on mount 'C:', start on mount 'd:'' followed by 'An error occurred while deploying a SAM Application. Check the logs for more information by running the "View AWS Toolkit Logs" command from the Command Palette.'

Figure SAM.VSC.3

It appears that the VSCode could not create a relative path between the IDE workspace installed on C: and the project code location on D:. I was not able to find a configuration setting to resolve this error. However the workaround was to move the project code to the same drive.



A screenshot of the VS Code interface showing a successful deployment log. The top bar includes 'Run Locally | Debug Locally | Configure' and the AWS Toolkit extension icon. The code editor shows a partial lambda function definition:

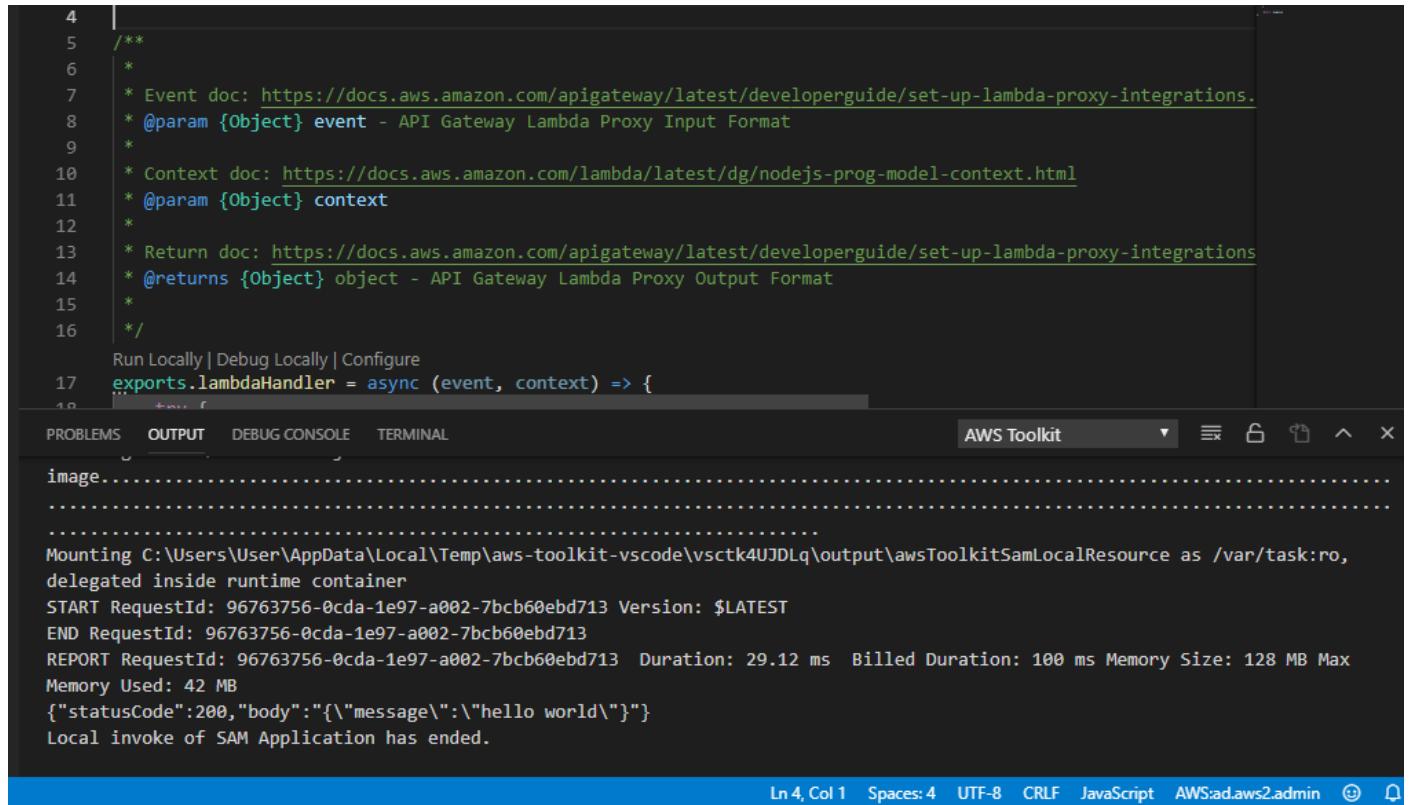
```
16 /*/  
17 Run Locally | Debug Locally | Configure  
18 exports.lambdaHandler = async (event, context) => {  
19     ...  
20     START RequestId: e00e1732-f299-1ff7-1458-190479194c88 Version: $LATEST  
21     END RequestId: e00e1732-f299-1ff7-1458-190479194c88  
22     REPORT RequestId: e00e1732-f299-1ff7-1458-190479194c88 Duration: 27.98 ms Billed Duration: 100 ms Memory Size: 128 MB Max  
Memory Used: 42 MB  
{"statusCode":200,"body":"{\"message\":\"hello world\"}"}  
Local invoke of SAM Application has ended.  
Starting SAM Application deployment...  
Building SAM Application...  
Packaging SAM Application to S3 Bucket: ad.aws.s3.codedeploy with profile: ad.aws2.admin  
Deploying SAM Application to CloudFormation Stack: sam-helloworld-stack with profile: ad.aws2.admin  
Successfully deployed SAM Application to CloudFormation Stack: sam-helloworld-stack with profile: ad.aws2.admin
```

Figure SAM.VSC.3

DEVELOPMENT ENVIRONMENT – TEST SAM LOCALLY

In order to aid and speed up development it is possible to configure a Serverless Application Model application to run locally for testing purposes. However to do that it is required to install Docker.

Initially I installed Docker Desktop. But this would not run unless a commercial licence for Windows Server exists. I had to uninstall Docker Desktop. I then installed Docker Toolbox [8]. I had numerous errors trying to deploy the SAM application locally on Docker. There were permissions issues, path problems, IDE configuration issues. It took 2 evenings to resolve all of the errors. However it is now possible for me to deploy and debug locally. Figure SAM.VSC.4



The screenshot shows a Visual Studio Code (VS Code) interface with the AWS Toolkit extension installed. The code editor displays a file named `index.js` containing a Lambda handler function:

```

4  /**
5   *
6   * Event doc: https://docs.aws.amazon.com/apigateway/latest/developerguide/set-up-lambda-proxy-integrations.html
7   * @param {Object} event - API Gateway Lambda Proxy Input Format
8   *
9   * Context doc: https://docs.aws.amazon.com/lambda/latest/dg/nodejs-prog-model-context.html
10  * @param {Object} context
11  *
12  * Return doc: https://docs.aws.amazon.com/apigateway/latest/developerguide/set-up-lambda-proxy-integrations.html
13  * @returns {Object} object - API Gateway Lambda Proxy Output Format
14  *
15  */
16 */
Run Locally | Debug Locally | Configure
17 exports.lambdaHandler = async (event, context) => {
18   ...

```

The bottom pane shows the terminal output of the Lambda function execution:

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL AWS Toolkit
image.....
.....
Mounting C:\Users\User\AppData\Local\Temp\aws-toolkit-vscode\vsctk4UJDlq\output\awsToolkitSamLocalResource as /var/task:ro,
delegated inside runtime container
START RequestId: 96763756-0cda-1e97-a002-7bcb60ebd713 Version: $LATEST
END RequestId: 96763756-0cda-1e97-a002-7bcb60ebd713
REPORT RequestId: 96763756-0cda-1e97-a002-7bcb60ebd713 Duration: 29.12 ms Billed Duration: 100 ms Memory Size: 128 MB Max
Memory Used: 42 MB
{"statusCode":200,"body":"{\\"message\\":\\"hello world\\\"}"}
Local invoke of SAM Application has ended.

```

At the bottom of the terminal window, status indicators show: Ln 4, Col 1, Spaces: 4, UTF-8, CRLF, JavaScript, AWS:ad.aws2.admin, and two small icons.

Figure SAM.VSC.4

JS app.js ⚡ Invoked sam-helloworld-stack-HelloWorldFunction-8O0IQOAZIAO2 X

Invoke function sam-helloworld-stack-HelloWorldFunction-8O0IQOAZIAO2

Select a file to use as payload:

Choose File No file chosen

Or, use a sample request payload from a template:

"Alan Duffin"

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL AWS Lambda

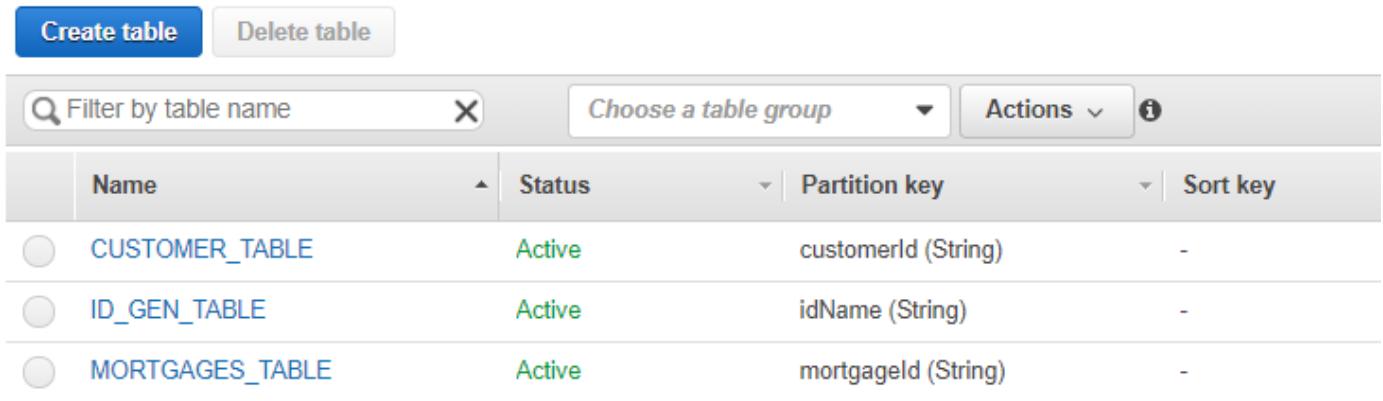
```
Invocation result for arn:aws:lambda:eu-west-1:727432808710:function:sam-helloworld-stack-HelloWorldFunction
Logs:
START RequestId: fe6a1e57-1136-4648-b73d-e40ba05097c0 Version: $LATEST
END RequestId: fe6a1e57-1136-4648-b73d-e40ba05097c0
REPORT RequestId: fe6a1e57-1136-4648-b73d-e40ba05097c0 Duration: 114.56 ms Billed Duration: 200 ms Memory Size: 128 MB
Memory Used: 75 MB Init Duration: 163.66 ms

Payload:
{"statusCode":200,"body":"{\\"message\\":\\"hello world\\\"}"}
```

Figure SAM.VSC.5

DEVELOPMENT – BUILD DYNAMODB STACK

Cloudformation Stack link [9]. This CloudFormation template will create the tables listed in Figure DB.1.



The screenshot shows the AWS DynamoDB management console. At the top, there are buttons for "Create table" and "Delete table". Below that is a search bar labeled "Filter by table name" and a dropdown menu "Choose a table group". On the right, there's an "Actions" button with a help icon. The main area is a table with four columns: "Name", "Status", "Partition key", and "Sort key". The "Name" column is sorted in ascending order. There are three rows in the table:

Name	Status	Partition key	Sort key
CUSTOMER_TABLE	Active	customerId (String)	-
ID_GEN_TABLE	Active	idName (String)	-
MORTGAGES_TABLE	Active	mortgageId (String)	-

Figure DB.1

In order to create unique Identifiers for Customers and Mortgage applications I needed to create a table that would store a counter for the ids and allow atomic updates when new customers and mortgage applications are created. UUIDs would not be suitable for customer id or mortgage application numbers.

The ID_GEN_TABLE structure is shown in Figure DB.2

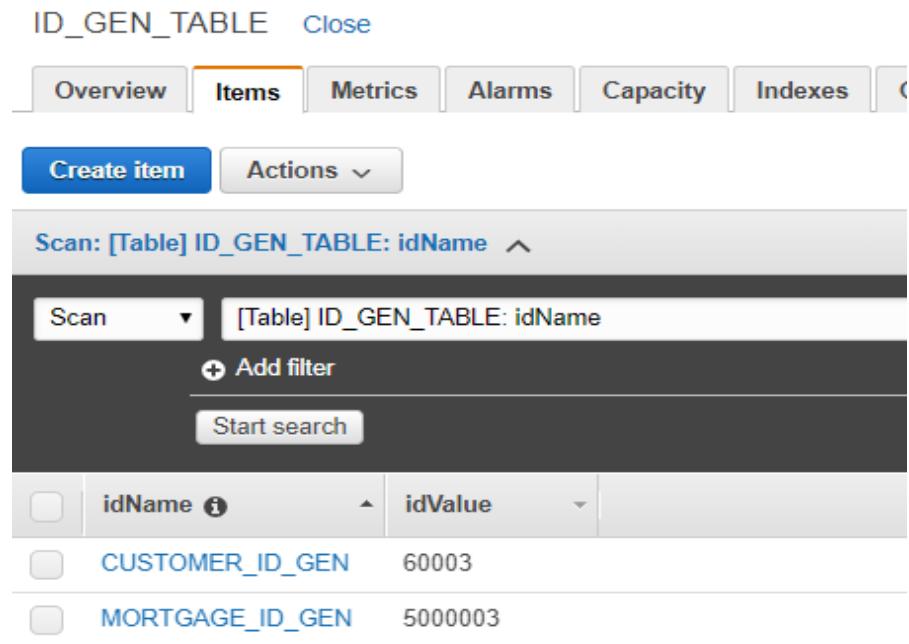


Figure DB.2

With regard to the Mortgages table, there is a requirement to allow application users to search all mortgage applications by the status code, e.g. 'PreSubmission' or 'In Assessment'. I tried a number of attempts to structure the table to achieve this but failed in the first 2 or 3 attempts. I had thought originally that the statusCode should be a Range Key and make up part of a Composite primary key. However this does not provide the required access. When a table has a composite primary key then all getItem operations MUST contain both the Hash and Range key. However in this scenario we do not know both the Hash and Range key. So the solution was to create Secondary Global Index which allows a different Hash Key than the main table. In this way we can use the getItem on the table when a single mortgage is required and we know the id. Then when searching for all mortgages that currently have a particular status we can perform a query operation on the Index. Figure DB.3

```
KeySchema:  
  - AttributeName: mortgageId  
    KeyType: HASH  
  BillingMode: PAY_PER_REQUEST  
GlobalSecondaryIndexes:  
  - IndexName: MortgateStatusIndex  
    KeySchema:  
      - AttributeName: mortgageStatus  
        KeyType: HASH  
    Projection:  
      ProjectionType: ALL
```

Figure DB.3

MORTGAGES_TABLE Close

Overview Items Metrics Alarms Capacity Indexes Global Tables Backups Triggers Access control Tags

Create item Actions ▾ ⚙️ ⟳

Query: [Index] MortgateStatusIndex: status ▾ Viewing 1 to 2 items

Query ▾ [Index] MortgateStatusIndex: status

Partition key status String = PreSubmission + Add filter

Sort Ascending Descending

Start search

	mortgageld	customerId	employerName	loanAmount	salary	term	yearsInEmployment	status
<input type="checkbox"/>	500013	6000002	Microsoft	200001	100001	21	11	PreSubmission
<input type="checkbox"/>	500014	6000003	Smurfit	300000	50000	20	20	PreSubmission

Figure DB.4

DEVELOPMENT – BUILD SERVERLESS SERVICES – LAMBDA.

Cloudformation Stack link [10].

It has taken between 16-18 hours this week alone to get a small suite of HTTP services built, deployed and tested. There were many issues and problems to be worked through. There was the learning curve on the following areas:

- 1 How to properly structure Restful HTTP operations.
- 2 Learning how to work with NodeJS.
- 3 Learning the AWS Serverless Application Model framework concepts and syntax.
- 4 Once the basic service APIs were deployed, I then had to learn how to used the AWS SDK to integrate with DynamoDB. This took quite a long time to get even the simplest operations working. The DynamoDB operations have a complex set of required parameters and difficult syntax. See Figure LAMBDA.1
- 6 Understanding the Promise and asyn / await mechanisms for calling asynchronous AWS SDK operations also took time.

```

try {
  const genIdResponse = await dynamoDb.updateItem({
    "TableName": "ID_GEN_TABLE",
    "ReturnValues": "UPDATED_NEW",
    "ExpressionAttributeValues": {
      ":incr": {"N": "1"}
    },
    "UpdateExpression": "SET idValue = idValue + :incr",
    "Key": {
      "idName": {
        "S": "MORTGAGE_ID_GEN"
      }
    }
  }).promise();
}

```

Figure LAMBDA.1

This figure illustrates the required query that is necessary to atomically read and increment a DynamoDB attribute.

Figure LAMBDA.2 illustrates the required operation structure for a simple update on the mortgage items.

```
var newMortgage = {
  TableName: "MORTGAGES_TABLE",
  Key: {
    "mortgageId": {
      S: mortgageId
    }
  },
  ExpressionAttributeNames: {
    "#A": "loanAmount",
    "#B": "yearsInEmployment",
    "#C": "salary",
    "#D": "employerName",
    "#E": "term",
    "#F": "mortgageStatus"
  },
  UpdateExpression: "set #A = :a, #B = :b, #C = :c, #D = :d, #E = :e, #F = :f",
  ExpressionAttributeValues: {
    ":a": { "N": mortgageRequest.loanAmount },
    ":b": { "N": mortgageRequest.yearsInEmployment },
    ":c": { "N": mortgageRequest.salary },
    ":d": { "S": mortgageRequest.employerName },
    ":e": { "N": mortgageRequest.term },
    ":f": { "S": mortgageRequest.mortgageStatus }
  },
  ReturnValues: "ALL_NEW"
};
```

Figure LAMBDA.2

There were many errors encountered. A sample can be found illustrated in Figure LAMBDA.3 and LAMBDA.4

```
sat Nov 02 14:26:06 UTC 2019 : Endpoint response headers: {Date=Sat, 
12 Nov 2019 14:26:06 GMT, Content-Type=application/json, Content-Length=1000, Connection=keep-alive, x-amzn-RequestId=4d7035f5-d4bb-4c42
.b538-4411600e3aad, x-amzn-Remapped-Content-Length=0, X-Amz-Executed
.Version=$LATEST, X-Amzn-Trace-Id=root=1-5dbd91fd-93d216a362b27c9c1c
:b9a96;sampled=0}
sat Nov 02 14:26:06 UTC 2019 : Endpoint response body before transformations: {"statusCode":400,"error":"Could not obtain new Mortgage I
": ResourceNotFoundException: Requested resource not found\n    at R
equest.extractError (/var/runtime/node_modules/aws-sdk/lib/protocol/
json.js:51:27)\n    at Request.callListeners (/var/runtime/node_
modules/aws-sdk/lib/sequential_executor.js:106:20)\n    at Request.emit
(/var/runtime/node_modules/aws-sdk/lib/sequential_executor.js:78:10)
\n    at Request.emit (/var/runtime/node_modules/aws-sdk/lib/reques
t.js:683:14)\n    at Request.transition (/var/runtime/node_modules/a
ws-sdk/lib/request.js:22:10)\n    at AcceptorStateMachine.runTo (/va
r/runtime/node_modules/aws-sdk/lib/state_machine.js:14:12)\n    at /
var/runtime/node_modules/aws-sdk/lib/state_machine.js:26:10\n    at
request.<anonymous> (/var/runtime/node_modules/aws-sdk/lib/request.j
s:38:9)\n    at Request.<anonymous> (/var/runtime/node_modules/aws-s
dk/lib/request.js:685:12)\n    at Request.callListeners (/var/runtim
e/node_modules/aws-sdk/lib/sequential [TRUNCATED]
sat Nov 02 14:26:06 UTC 2019 : Execution failed due to configuration
error: Malformed Lambda proxy response
sat Nov 02 14:26:06 UTC 2019 : Method completed with status: 502
```

Figure LAMBDA.3

```
ration latency: 1527 ms
Sat Nov 02 14:39:27 UTC 2019 : Endpoint response headers: {Date=Sat, 02 Nov 2019 14:39:27 GMT, Content-Type=application/json, Content-Length=1288, Connection=keep-alive, x-amzn-RequestId=e984d21b-7bbb-48ff -bd66-c907c96d23f1, x-amzn-Remapped-Content-Length=0, X-Amz-Executed -Version=$LATEST, X-Amzn-Trace-Id=root=1-5dbd951e-5462164ee1e75ae8fd 5659f5;sampled=0}
Sat Nov 02 14:39:27 UTC 2019 : Endpoint response body before transformations: {"statusCode":400,"error":"Could not obtain new Mortgage ID: MultipleValidationErrors: There were 4 validation errors:\n* InvalidParameterType: Expected params.Item['term'].N to be a string\n* InvalidParameterType: Expected params.Item['salary'].N to be a string\n* InvalidParameterType: Expected params.Item['yearsInEmployment'].N to be a string\n* InvalidParameterType: Expected params.Item['loanAmount'].N to be a string\n    at ParamValidator.validate (/var/runtime/node_modules/aws-sdk/lib/param_validator.js:40:28)\n    at Request.VALIDATE_PARAMETERS (/var/runtime/node_modules/aws-sdk/lib/event_listeners.js:126:42)\n    at Request.callListeners (/var/runtime/node_modules/aws-sdk/lib/sequential_executor.js:96:12)\n    at /var/runtime/node_modules/aws-sdk/lib/sequential_executor.js:86:9\n    at finish (/var/runtime/node_modules/aws-sdk/lib/config.js:349:7)\n    at /var/runtime [TRUNCATED]
Sat Nov 02 14:39:27 UTC 2019 : Execution failed due to configuration error: Malformed Lambda proxy response
Sat Nov 02 14:39:27 UTC 2019 : Method completed with status: 502
```

LAMBDA.4

The Lambda functions created so far are shown in Figure LAMBDA.5

Functions (9)			
	Function name	Description	Runtime
<input type="radio"/>	TestReadFromDynamo		Node.js 10.x
<input type="radio"/>	Mortgages-Stack-CreateCustomerFunction-NC15CDP9GDUI		Node.js 10.x
<input type="radio"/>	Mortgages-Stack-ListByStatusMortgageFunction-SFRBQ8A8ULIO		Node.js 10.x
<input type="radio"/>	Mortgages-Stack-GetCustomerFunction-P1ADMOV52KA0		Node.js 10.x
<input type="radio"/>	Mortgages-Stack-UpdateMortgageFunction-15LUWGKY2ZX2Q		Node.js 10.x
<input type="radio"/>	Mortgages-Stack-DeleteMortgageFunction-L0XFGM8ONTYQ		Node.js 10.x
<input type="radio"/>	Mortgages-Stack-UpdateCustomerFunction-QXUSAOYPACZE		Node.js 10.x
<input type="radio"/>	Mortgages-Stack-CreateMortgageFunction-ZC5021VBH5GA		Node.js 10.x
<input type="radio"/>	Mortgages-Stack-GetMortgageFunction-14BLETUMUYZEJ		Node.js 10.x

Figure LAMBDA.5

The API Gateway resource that have been create are listed in Figure LAMBDA.6

The screenshot shows the AWS API Gateway console. The navigation bar at the top includes the AWS logo, Services dropdown, Resource Groups dropdown, and a user account indicator (ad.aws.admin @ ap). The main area displays the API structure for a stack named "Mortgages-Stack". On the left, a sidebar lists "APIs", "Stages", "Authorizers", "Gateway Responses", "Models", "Resource Policy", and "Documentation", with "Resources" currently selected. The main content area shows a tree view of resources under the root "/":

- /customer/{id} (highlighted)
- /customer
- /mortgage

Under the "/customer/{id}" resource, there are two methods listed:

- GET**: Authorization: None, API Key: Not required
- POST**: Authorization: None, API Key: Not required

Figure LAMBDA.6

DEVELOPMENT – WORKFLOW IMPLEMENTATION – STEP FUNCTION.

The template for the initial implementation can be found at the link [11]. An illustration of the Step Function implementation is shown in Figure STEP.1.

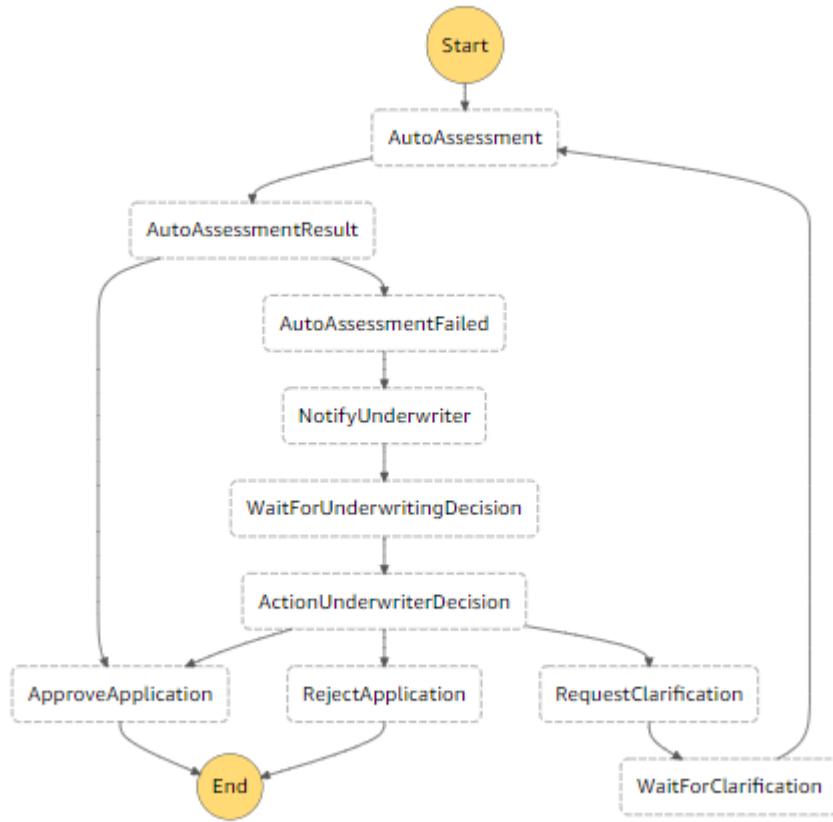


Figure STEP.1

REFERENCE LIST AND BIBLIOGRAPHY

- [1] AWS Free Tier. <https://aws.amazon.com/free/>
- [2] AWS CLI. <https://docs.aws.amazon.com/cli/latest/userguide/install-windows.html>
- [3] AWS CLI SAM. <https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-install-windows.html>
- [4] Node JS. <https://nodejs.org/en/download/>
- [5] Configure AWS CLI. <https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html>
- [6] Visual Studio Code. <https://code.visualstudio.com/download>
- [7] Configure AWS on Visual Studio Code. <https://docs.aws.amazon.com/toolkit-for-vscode/latest/userguide/setup-toolkit.html>
- [8] Install Docker Toolbox. https://docs.docker.com/toolbox/toolbox_install_windows/
- [9] DynamoDB Cloudformation Template.
<https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.dynamo/dynamo-create.yml>
- [10] Lambda Cloudformation Template.
<https://github.com/GeneralYeager/tallaght.mortgages/blob/master/tud.mortgages/Tallaght-Mortgages-App/template.yaml>
- [11] Step Function State Machine Template.
<https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.workflow/StateMachineV1.json>

Dublin Technological University,
Certificate, Cloud Solutions Architecture (2020)

Implementation Iteration 2.1

Student Name: Alan Duffin

Student ID: X00159409

SUBMISSION MATERIAL

Source Code NodeJS / Lambda	https://github.com/GeneralYeager/tallaght.mortgages/tree/master/tud.mortgages/Tallaght-Mortgages-App
Workflow Cloudformation	https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.workflow/StateMachineV3.txt
DynamoDB Cloudformation	https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.dynamo/dynamo-create.yml
Lambda / API Gateway Cloudformation	https://github.com/GeneralYeager/tallaght.mortgages/blob/master/tud.mortgages/Tallaght-Mortgages-App/template.yaml
Angular SPA Source Code	https://github.com/GeneralYeager/tallaght.mortgages/tree/master/mort-app-web

Table A.

ITERATION 2.1 TASKS

Building upon last weeks progress report, this document outlines the additional project tasks that have been completed for implementation phase 2. The follow tasks were completed this week.

Project Phase	Service / Area	Implementation Task
Build / Development	AWS Websockets	Human interaction requires notification with User Interface.
Build / Development	AWS Lambda	Build Lambda functions to manage API Gateway Websockets and notifications.
Build / Development	DynamoDB	Build table to manage API Websocket connection tokens.
Build / Development	Angular SPA	Build Browser based Front End to allow users to create/modify/submit Mortgage Applications for workflow assessment.
Test		Test Websocket Notification Events and subscriptions
Build / Development	AWS Lambda	Additiona Lambda functions to support Step Function execution.
Build / Development	AWS Step Functions	Extend Step Function state machine to support Human decision making.
Design	AWS Step Functions	Integrate Step Function generation into Cloudformation template.
Test	AWS Step Functions	Test State Transitions / Events

Table B.

I have extended the suite of Lambda services that will allow a the Step function state machine to transition Mortgage application states and allow notification to different user categories (Brokers and Underwriters). In order to support this I have had to build an intial Browser based Single Page Application using the Javascript Angular framework. At this point Brokers can create and amend mortgage applications. I have built Angular services that connect to the AWS API Gateway Websocket API that allows traffic to be initiated either by the client or the server. This is necessary so that uses can be notified of Mortgage application state changes. So users will be notified of State changes and required tasks via two channels SNS (email) and real time browser based websockets.

It has been necessary to build additional Lambda functions to support the integration between the Step Function State Machine and the API Gateway Websocket layer. Also some additional Lambda functions were developed specifically to transition the Mortgage Applications between states.

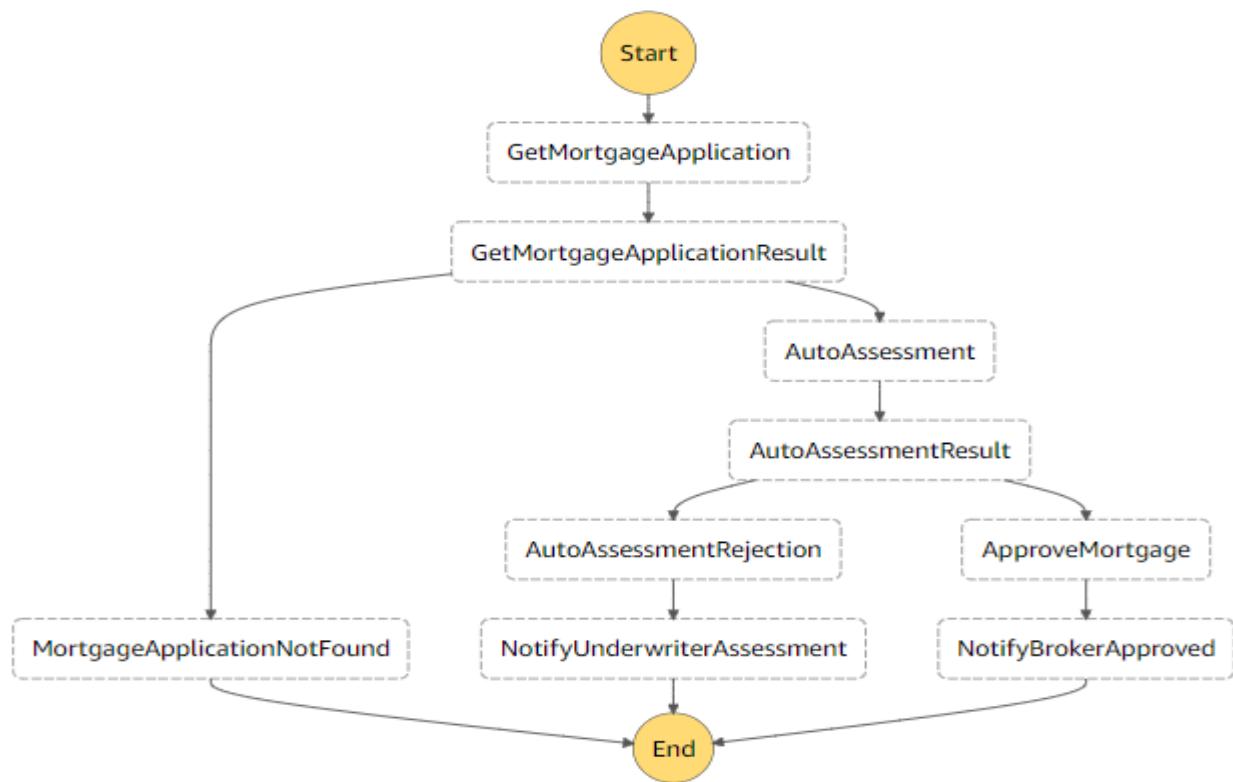
PLAN FOR ITERATION 2.2

Project Phase	Service / Area	Implementation Task
Build / Development	AWS Step Functions	Complete the Step Function Workflow state machine.
Build / Development	AWS Cloudformation	Integrate the deployment of the Workflow state machine into the project Cloud formation template.
Build / Development	AWS Lamda	Build the Lambda Functions to support pausing the workflow to await human decisions. Also manage the Step Function tokens.
Build / Development	DynamoDB	Implement new DB schema to map mortgage application to Step Function execution id. This is required to allow users approve or reject applications without reference to the step function execution id.
Build / Development	Angular	Extends the Browser application to allow submission of mortgages for assessment. Allow for clarification requests and Underwriter decision making.
Test	AWS Step Functions	Test State Transitions / Events / Human interaction

Table C.

The workflow state machine currently exists as seen in Figure AWS.STEP.1

Figure AWS.STEP.1



SERVLERLESS / LAMBDA FUNCTIONS

The following Lambda functions have now been completed as of this progress report. Figure AWS.LAMBDA.1

Mortgages-Stack-AutoAssessmentFunction-S2305392YHS1
Mortgages-Stack-ChangeMortgageStatusFunction-2NK931AA2H2N
Mortgages-Stack>CreateCustomerFunction-NC15CDP9GDUI
Mortgages-Stack>CreateMortgageFunction-ZC5021VBH5GA
Mortgages-Stack>DeleteMortgageFunction-L0XFGM8ONTYQ
Mortgages-Stack-GetCustomerFunction-P1ADM0V52KA0
Mortgages-Stack-GetMortgageFunction-14BLETUMUYZEJ
Mortgages-Stack-ListByStatusMortgageFunction-SFRBQ8ABULIO
Mortgages-Stack-MortgageSNSFunction-1TAXBKNV0FSUX
Mortgages-Stack-MortgageStatusSNSFunction-5UVPVE4OU9PF
Mortgages-Stack-NotifyUsersSNSFunction-216EXFJBHO0V
Mortgages-Stack-StartWorkflowFunction-4H7HZIPCE3XX
Mortgages-Stack-UpdateCustomerFunction-QXUSA0YPACZE
Mortgages-Stack-UpdateMortgageFunction-15LUWGKY2ZX2Q
Mortgages-Stack-WebSocketConnectFunction-1GH8JPQCLA8YJ
Mortgages-Stack-WebSocketDisconnectFunction-1PPRHIDN8QRG3
Mortgages-Stack-WebSocketSendFunction-H1E7UBOP0KFV
Mortgages-Stack-WorkflowGetCustomerFunction-1JZFBVZ6QABZG
Mortgages-Stack-WorkflowGetMortgageFunction-16L3UM67HNU4N
Mortgages-Stack-WorkflowUpdateMortgageFunction-1EKLNK5DXIDZG

Figure AWS.LAMBDA.1

This table lists the purpose of each of the completed Lambda Functions. Table AWS.LAMBDA.1

Lambda Function	Purpose
Mortgages-Stack-AutoAssessmentFunction	Perform Mortgage Rules check
Mortgages-Stack-ChangeMortgageStatusFunction	Modify the Mortgage Application Status and store.
Mortgages-Stack-CreateCustomerFunction	HTTP API Create Customer
Mortgages-Stack-CreateMortgageFunction	HTTP API Create Mortgage Application
Mortgages-Stack-DeleteMortgageFunction	HTTP API Delete Mortgage Application
Mortgages-Stack-GetCustomerFunction	HTTP API Find By PK Customer
Mortgages-Stack-GetMortgageFunction	HTTP API Find By PK Mortgage Application
Mortgages-Stack-ListByStatusMortgageFunction	HTTP API Find By Status Mortgage Application
Mortgages-Stack-MortgageStatusSNSFunction	Step Function Update Mortgage Status
Mortgages-Stack-NotifyUsersSNSFunction	Step Function Notify State Change
Mortgages-Stack-StartWorkflowFunction	HTTP API Execute Mortgage Workflow
Mortgages-Stack-UpdateCustomerFunction	HTTP API Update Customer
Mortgages-Stack-UpdateMortgageFunction	HTTP API Update Mortgage
Mortgages-Stack-WebSocketConnectFunction	Manage Websocket connection and store token.
Mortgages-Stack-WebSocketDisconnectFunction	Manage Websocket disconnect and delete token.
Mortgages-Stack-WebSocketSendFunction	Step Function send message to API Gateway Websocket for Browser broadcast.
Mortgages-Stack-WorkflowGetCustomerFunction	Step Function Find By PK Customer
Mortgages-Stack-WorkflowGetMortgageFunction	Step Function Find By PK Mortgage
Mortgages-Stack-WorkflowUpdateMortgageFunction	Step Function Update Mortgage

Table AWS.LAMBDA.1

The source code and deployment template for these functions are available at the GitHub repository listed in Table A or in the Appendix.

I have had to implement new Lambda functions to manage API Gateway Websocket functionality.

API GATEWAY WEBSOCKETS

Last year AWS announced API Gateway support for the Websocket protocol [12]. This allows for the creation of ...

“bidirectional communication applications using WebSocket APIs in Amazon API Gateway without having to provision and manage any servers.”

“HTTP-based APIs use a request/response model with a client sending a request to a service and the service responding synchronously back to the client. WebSocket-based APIs are bidirectional in nature. This means that a client can send messages to a service and services can independently send messages to its clients.”

“This bidirectional behavior allows for richer types of client/service interactions because services can push data to clients without a client needing to make an explicit request. WebSocket APIs are often used in real-time applications such as chat applications, collaboration platforms, multiplayer games, and financial trading platforms.”

The following API Gateway Websocket API has been created. Figure AWS.API.1

The screenshot shows the AWS Lambda interface for managing API routes. The top navigation bar includes the Amazon API Gateway logo, the path 'APIs > MortgageWebSocket (h9uk1z65s6) > Routes', and a 'Actions' dropdown. On the left, a sidebar lists 'APIs', 'Custom Domain Names', and the selected 'API: MortgageWebSoc...'. Under the API details, the 'Routes' tab is active, showing a list of predefined routes: '\$connect', '\$disconnect', 'sendmessage', and '\$default'. A 'Route Selection Expression' field contains '\$request.body.message' with an edit icon. A 'New Route Key' input field is also present.

Figure AWS.API.1

The screenshot shows the AWS API Gateway interface for a stage named 'Prod'. At the top, there are navigation links for APIs, Stages, and Prod. On the right, there are buttons for 'Show all hints' and a help icon. Below the navigation, there are tabs for 'Stages' and 'Create'. A 'Delete Stage' button is visible on the right. The main area is titled 'Prod Stage Editor' and contains a section for 'WebSocket URL' and 'Connection URL'. Below this, there are tabs for 'Settings' (which is selected), 'Logs/Tracing', 'Stage Variables', and 'Deployment History'. Under the 'Settings' tab, there is a section for 'Default Route Throttling' with a note about account-level throttling rates. An 'Enable throttling' checkbox is present.

Figure AWS.API.2

Because API Gateway support for Websockets is very recent, the online help is minimal. It took many hours to get the deployment and development correct.

There were many errors encountered such as incorrect endpoint format and futile attempts to get Lambda functions to establish wss protocol connections.

This error was caused by an invalid endpoint format.

The screenshot shows the AWS Lambda function execution results for a function named 'Mortgages-Stack-WebSocketSend...'. The results indicate a successful execution ('Execution result: succeeded'). The 'Logs' section is expanded, showing the error message: 'UnknownEndpoint: Inaccessible host: `wss'. This service may not be available in the `eu-west-1` region.' The 'Details' section shows the error stack trace, which includes multiple lines of Node.js and AWS SDK error logs. The stack trace starts with 'Request.ENTFOUND_ERROR' and ends with 'ClientRequest.EventEmitter.emit'.

Figure AWS.SOCKET.1

It is not possible for Lambda functions to connect to an external WSS endpoint. They must directly use the 'AWS.ApiGatewayManagementApi' sdk and connect via the API Gateway instance.

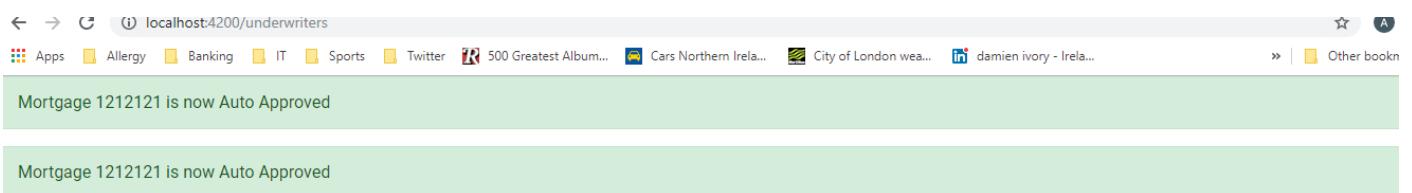
```

395     Runtime: nodejs10.x
396     Environment:
397       Variables:
398         TABLE_NAME: !Ref WebSocketConnectionTableName
399         # WEBSOCKET_ENDPOINT: !Join [ '', [ 'wss://', !Ref MortgageWebSocket, '.execute-api.', !Ref 'A
400         WEBSOCKET_ENDPOINT: !Join [ '', [ !Ref MortgageWebSocket, '.execute-api.', !Ref 'AWS::Region
401       Policies:
402         !DynamicDPCloudPolicy

```

Figure AWS.SOCKET.2

Websocket Success. Figure AWS.SOCKET.3



Tallaght Mortgages Underwriters Page

[Home](#)
underwriter-component works!

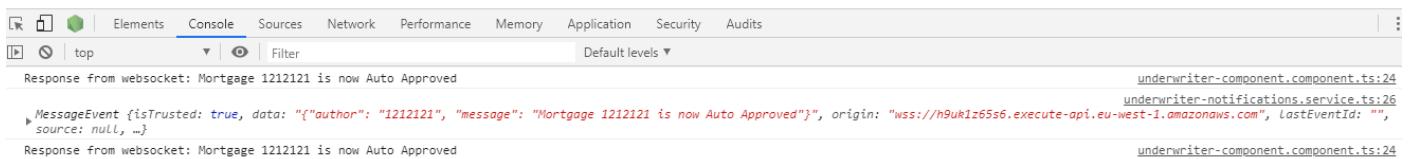


Figure AWS.SOCKET.3

Socket Broadcast Source Code:

```
exports.handler = async (event, context) => {
  let connectionData;
  console.log(event);
  try {
    connectionData = await ddb.scan({ TableName: TABLE_NAME, ProjectionExpression: 'connectionId' }).promise();
  } catch (e) {
    return { statusCode: 500, body: e.stack };
  }
  const apigwManagementApi = new AWS.ApiGatewayManagementApi({
    apiVersion: '2018-11-29',
    endpoint: WEBSOCKET_ENDPOINT //event.requestContext.domainName + '/' + event.requestContext.stage
  });
  //const postData = JSON.parse(event.body).data;
  const postData = '{"author": "1212121", "message": "Mortgage 1212121 is now Auto Approved"}';
  const postCalls = connectionData.Items.map(async ({ connectionId }) => {
    try {
      await apigwManagementApi.postToConnection({ ConnectionId: connectionId, Data: postData }).promise();
    } catch (e) {
      if (e.statusCode === 410) {
        console.log(`Found stale connection, deleting ${connectionId}`);
        await ddb.delete({ TableName: TABLE_NAME, Key: { connectionId } }).promise();
      } else {
        throw e;
      }
    }
  });
  try {
    await Promise.all(postCalls);
  } catch (e) {
    return { statusCode: 500, body: e.stack };
  }
}
```

```

    return { statusCode: 200, body: 'Data sent.' };
}

```

Web Browser Websocket Connection Source Code:

```

import { Injectable } from '@angular/core';

import { Observable, Observer, Subject } from 'rxjs'

@Injectable({
  providedIn: 'root'
})
export class WebsocketService {
  constructor() {}

  private subject: Subject<MessageEvent>;

  public connect(url): Subject<MessageEvent> {
    console.log("connect url");
    if (!this.subject) {
      this.subject = this.create(url);
      console.log("Successfully connected: " + url);
    }
    return this.subject;
  }

  private create(url): Subject<MessageEvent> {
    let ws = new WebSocket(url);
    console.log("create ws");
    let observable = Observable.create((obs: Observer<MessageEvent>) => {
      console.log("in obs");
      //ws.onmessage = obs.next.bind(obs);
      ws.onmessage = function(event) {
        console.log("WebSocket message received:", event);
        obs.next(event);
      };
      ws.onerror = obs.error.bind(obs);
      ws.onclose = obs.complete.bind(obs);
      return ws.close.bind(ws);
    });
    let observer = {
      next: (data: Object) => {
        console.log("in next");
        if (ws.readyState === WebSocket.OPEN) {

```

```
        ws.send(JSON.stringify(data));
    }
}
};

return Subject.create(observer, observable);
}
}
```

ANGULAR SINGLE PAGE APPLICATION

In order to build a Javascript based Bowser application to demonstrate and interact with the Step Function Workflow I have built a first iteration of an Angular SPA application.

Steps to set up the development environment are:

- 1 Install npm. <https://nodejs.org/en/>
- 2 Install Angular CLI. <https://angular.io/guide/setup-local>
- 3 Install Angular Material Design. <https://material.angular.io/guide/getting-started>
- 4 Install Websocket NPM Package. <https://www.npmjs.com/package/websocket>
- 5 The source code completed so far is stored in Git Hub.

At the moment the SPA allows users to create / modify / list mortgage application. View Websocket notifications.

The look and feel is very basic but will be sufficient to provide a working system.

Run the SPA locally – shown below.

```
C:\Users\User\aws\tallaght.mortgages\mort-app-web>ng serve
Your global Angular CLI version (8.3.18) is greater than your local
version (8.3.17). The local Angular CLI version is used.

To disable this warning use "ng config -g cli.warnings.versionMismatch false".
10% building 3/3 modules 0 active i ?wds?: Project is running at http://localhost:4200/webpack-dev-server/
i ?wds?: webpack output is served from /
i ?wds?: 404s will fallback to //index.html

chunk {main} main.js, main.js.map <main> 77.4 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map <polyfills> 264 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map <runtime> 6.15 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map <styles> 338 kB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map <vendor> 6.97 MB [initial] [rendered]
Date: 2019-11-17T14:58:39.235Z - Hash: 1c4a25c725afce1cf961 - Time: 55294ms
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **
i ?wdm?: Compiled successfully.
```

Figure AWS.ANGULAR.1

Home Page. Figure AWS.ANGULAR.2



Tallaght Mortgages Home Page

Brokers Underwriters

Tallaght Mortgages Brokers Page

Home Create New Mortgage

mortgageId	customerId	employerName	loanAmount	mortgageStatus	salary	term	yearsInEmployment
5000005	60002	Ryanair	222000	PreSubmission	22001	22	22
5000004	60003	AIB Ltd	999999	PreSubmission	100001	91	99
5000002	60002	Smurfit	300009	WithUnderwriter	50009	39	29
5000003	60003	Microsoft	200000	PreSubmission	100000	20	11

Broker Page. Figure AWS.ANGULAR.3

Modify Mortgages. Figure AWS.ANGULAR.4

mortgageId	customerId	employerName	loanAmount	mortgageStatus	salary	term	yearsInEmployment
5000005	60002	Ryanair	222000	PreSubmission	22001	22	23
		employerName Ryanair	Years in Employments 23		Salary 22001		
		Loan Amount 222000	Term (Years) 22				
Save							
5000004	60003	AIB Ltd	999999	PreSubmission	100001	91	99
5000002	60002	Smurfit	300009	WithUnderwriter	50009	39	29
5000003	60003	Microsoft	200000	PreSubmission	100000	20	11

Recieve Mortgage Status Notifications. Figure AWS.LAMBDA.5

Mortgage 1212121 is now Auto Approved

Tallaght Mortgages Underwriters Page

[Home](#)

Iunderwriter-component works!

Cross-Origin Resource Sharing ([CORS](#)) is a mechanism that uses additional [HTTP](#) headers to tell browsers to give a web application running at one [origin](#) access to selected resources from a different origin. A web application executes a **cross-origin HTTP request** when it requests a resource that has a different origin (domain, protocol, or port) from its own.

<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

The AWS documentation enabling CORS on API Gateway via Cloudformation is lacking. It took a couple of days for me to get the HTTP API calls working from my local development environment.

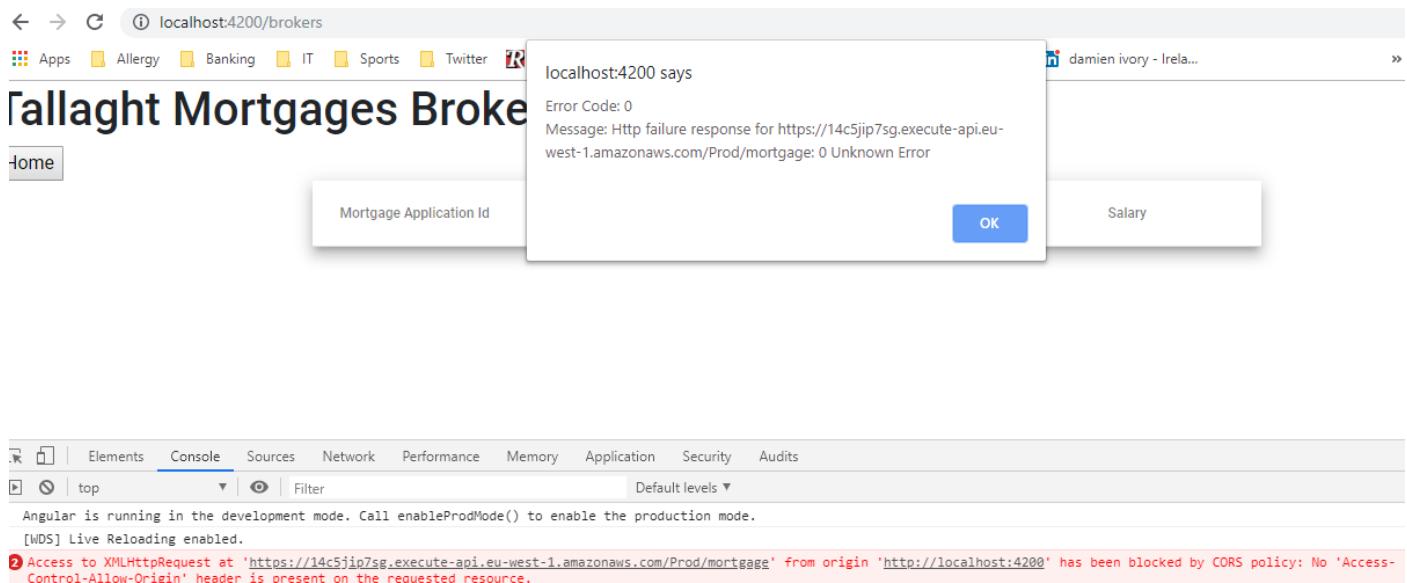


Figure AWS.CORS.1

Attempt to enable CORS post deployment.

The screenshot shows the AWS API Gateway console interface. The top navigation bar indicates the path: APIs > Mortgages-Stack (14c5jip7sg) > Resources > /mortgag... . The left sidebar contains links for APIs, Stages, Authorizers, Gateway Responses, Models, Resource Policy, Documentation, Dashboard, Settings, and MortgageWebSocket. The 'Resources' link is currently selected. In the main content area, a tree view shows resources under paths like /, /customers, /{id}, and /mortgage. A context menu is open over the /mortgage resource, specifically over the GET method. The menu is titled 'Actions' and includes three sections: 'METHOD ACTIONS' (Edit Method Documentation, Delete Method), 'RESOURCE ACTIONS' (Create Method, Create Resource, Enable CORS, Edit Resource Documentation, Delete Resource), and 'API ACTIONS' (Deploy API, Import API, Edit API Documentation, Delete API). To the right of the menu, there are two boxes: 'Method I' containing 'Auth: NOI' and 'ARN: arn:aws:apigateway:us-east-1::/restapis/14c5jip7sg/resources/.../methods/...', and 'Method F' containing 'Select an i...'. The URL in the browser's address bar is /mortgage - GET - Method I.

Figure AWS.CORS.2

The screenshot shows the AWS Lambda API Gateway CORS configuration page. On the left, a sidebar lists resources: /customer, /{id}, and /mortgage. The /mortgage resource is selected and expanded, showing methods: GET, OPTIONS, PUT, and a nested /{id} section with DELETE and GET methods. On the right, the 'Enable CORS' configuration panel is displayed. It includes fields for 'Gateway Responses for Mortgages-Stack API': 'DEFAULT 4XX' and 'DEFAULT 5XX'. Under 'Methods', checkboxes are checked for GET, OPTIONS, and PUT. Below this, 'Access-Control-Allow-Methods' is set to 'GET, OPTIONS, PUT'. The 'Access-Control-Allow-Headers' field contains 'Content-Type,X-Amz-Date,Authorization'. The 'Access-Control-Allow-Origin*' field is empty. A blue button at the bottom right says 'Enable CORS and replace existing CORS headers'.

Figure AWS.CORS.3

This screenshot shows the same CORS configuration page as Figure AWS.CORS.3, but with a different outcome. The 'Access-Control-Allow-Origin*' field now contains 'http://example.com'. The 'Enable CORS and replace existing CORS headers' button has been clicked, and the page displays a summary of successful steps: '✓ Add Access-Control-Allow-Headers, Access-Control-Allow-Methods, Access-Control-Allow-Origin Method Response Headers to OPTIONS method' and '✓ Add Access-Control-Allow-Headers, Access-Control-Allow-Methods, Access-Control-Allow-Origin Integration Response Header Mappings to OPTIONS method'. It also lists failed steps: '✗ Add Access-Control-Allow-Origin Method Response Header to GET method' (with a red error icon), '✗ Add Access-Control-Allow-Origin Integration Response Header Mapping to GET method' (with a red error icon), '✗ Add Access-Control-Allow-Origin Method Response Header to PUT method' (with a red error icon), and '✗ Add Access-Control-Allow-Origin Integration Response Header Mapping to PUT method' (with a red error icon). A note at the bottom states: 'Your resource has been configured for CORS. If you see any errors in the resulting output above please check the error message and if necessary attempt to execute the failed step manually via the Method Editor.'

Figure AWS.CORS.4

Some sites indicated that the correct Cloudformation format to enable CORS for API Gateway as shown below.

```
! template.yaml template.yaml
1   AWSTemplateFormatVersion: '2010-09-09'
2   Transform: AWS::Serverless-2016-10-31
3   Description: >
4     | Tallaght-Mortgages-App
5
6   Globals:
7     Function:
8       Timeout: 3
9     Api:
10       EndpointConfiguration: REGIONAL
11       Cors: "*"
12
```

Figure AWS.CORS.5

After much web surfing and trial and error the correct template is shown below:

```
! template.yaml template.yaml
1 AWSTemplateFormatVersion: '2010-09-09'
2 Transform: AWS::Serverless-2016-10-31
3 Description: >
4   Tallaght-Mortgages-App
5
6 Globals:
7   Function:
8     Timeout: 3
9   Api:
10    EndpointConfiguration: REGIONAL
11    Cors:
12      AllowMethods: "*"
13      AllowHeaders: "'Content-Type'"
14      AllowOrigin: "*"
15
16 Parameters:
```

Figure AWS.CORS.6

API Gateway with Lambda integration (rather than HTTP integration) does not fully abstract out CORS functionality. Individual Lambda function must add CORS headers as seen in Figure AWS.CORS.7

```
const findMortgageResponse = await dynamoDb.scan(findAllMortgages).promise();
console.log(findMortgageResponse);
return {
  statusCode: 200,
  headers: { 'Content-Type': 'application/json', "Access-Control-Allow-Origin": "*" },
  body: JSON.stringify(findMortgageResponse.Items)
};
```

Figure AWS.CORS.7

DEVELOPMENT – WORKFLOW IMPLEMENTATION – STEP FUNCTION.

The second iteration of the Step Function State Machine was shown in Figure AWS.STEP.1

The workflow has been extended only slightly this week but in the coming week I hope to have completed the workflow by adding the following steps:

- 1 Add Clarification, Rejection and Approval processes based on the human underwriter decision.
- 2 Pause the workflow when the Mortgage Application is assigned to a human underwriter.
- 3 Pause the workflow when the Mortgage Application is returned to the Broker for clarification.
- 4 Associate the workflow restart token with the mortgage Id and persist to DynamoDB.
- 5 Complete all of the Websocket and SNS notification paths.
- 6 Integrate with the Browser SPA, allow users to start and restart the workflow process.

The small enhancements that I have completed for the state machine this week have allowed me to become more familiar with the Step Function functionality and notation. It is very important to have a clear design for the input and outputs of each Task/Lambda function/method and the transformations that are required between Tasks to allow subsequent tasks in the workflow to access the required data.

The testing and bug fixing that I performed is summarised now in the following snapshots of two scenarios : Auto Assessment rejection and Auto Assessment success.

Auto Assessment Rejection

Submit a Mortgage Application that does not meet the Auto Assessment requirements. Figure AWS.AUTO.1

New execution

Start an execution using the latest definition of the state machine. [Learn more](#)

Enter an execution name - optional
Enter your execution id here
dcc34e2d-0996-590f-ff30-b79cc2413c34

Input - optional
Enter input values for this execution in JSON format

```
1 var {  
2   "mortgageId": "5000002"  
3 }
```

Open in a new browser tab Cancel **Start execution**

Figure AWS.AUTO.1

Invalid JSON path.

Execution History				
Step	Event	Timestamp	Duration	Log Stream
▼ 17	ExecutionFailed	-	820	Nov 16, 2019 05:30:31.402 PM
<pre>{ "error": "States.Runtime", "cause": "An error occurred while executing the state 'MortgageApplicationNotFound' (entered at the event id #16). The JSONPath '\$.error' specified for the field 'Message.\$' could not be found in the input '{\"statusCode\":400,\"approved\":false,\"reason\":\"Must be earning in excess of 100000 euro for autoapproval\"}'" }</pre>				

The InputPath on the Task must isolate the required JSON node to provide the correct input to the Lambda function. Figure AWS.AUTO.2

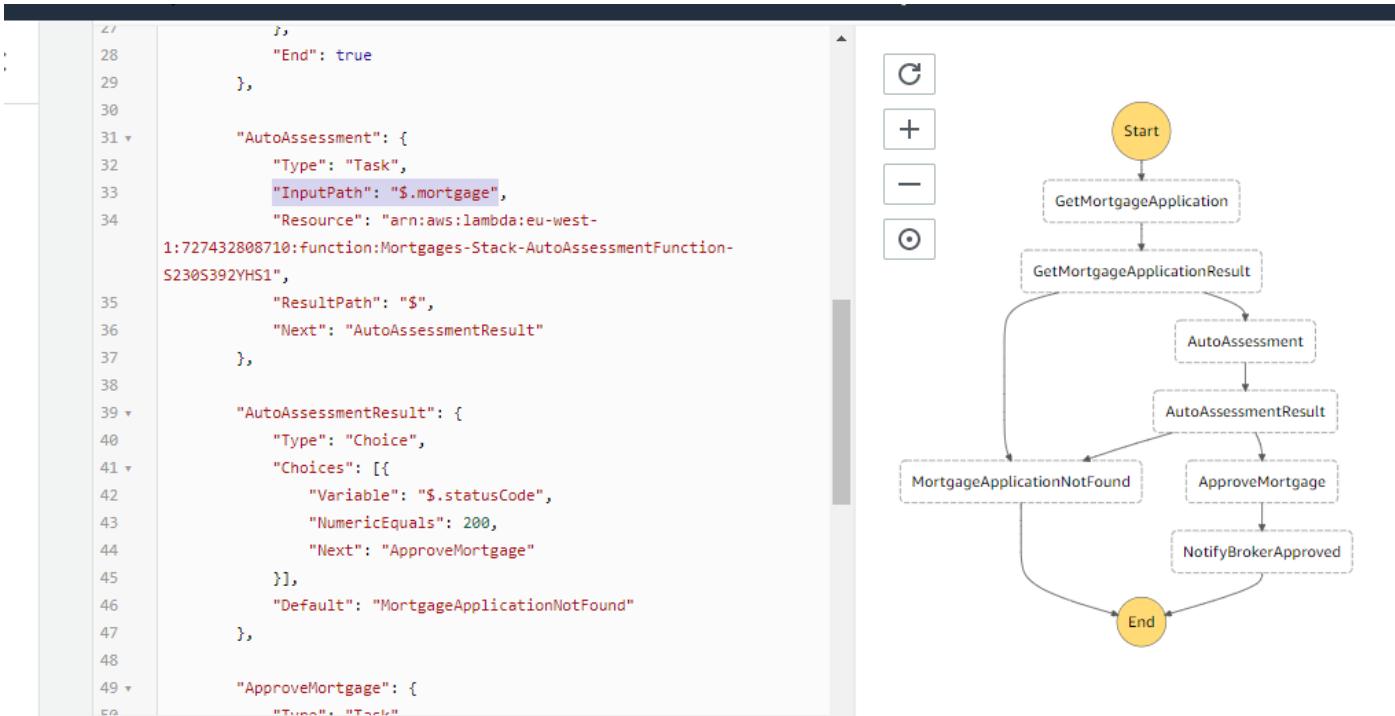


Figure AWS.AUTO.2

Also the Lambda Function must return any and all data that is required for the next task in the chain.

Figure AWS.AUTO.3

The screenshot shows a code editor for a Lambda function. The code is written in JavaScript and implements the logic for the 'AutoAssessment' task.

```

        }
    }
    if (mortgageApplication.salary != undefined && mortgageApplication.salary != null) {
        let salary = toInt(mortgageApplication.salary)
        if (100000 > salary) {
            return {
                statusCode: 400,
                approved: false,
                reason: 'Must be earning in excess of 100000 euro for autoapproval',
                mortgage: mortgageApplication
            };
        }
    }
}

```

Figure AWS.AUTO.3

If the input and output data is not correctly mediated then the Step Function execution will fail. Figure AWS.AUTO.4

New execution X

Start an execution using the latest definition of the state machine. [Learn more](#)

Enter an execution name - optional
Enter your execution id here

Input - optional
Enter input values for this execution in JSON format

```
1 * {  
2     "mortgageId": "5000002"  
3 }
```

Open in a new browser tab Cancel **Start execution**

Figure AWS.AUTO.4

The AWS Console will highlight failed states as shown below. Figure AWS.AUTO.5

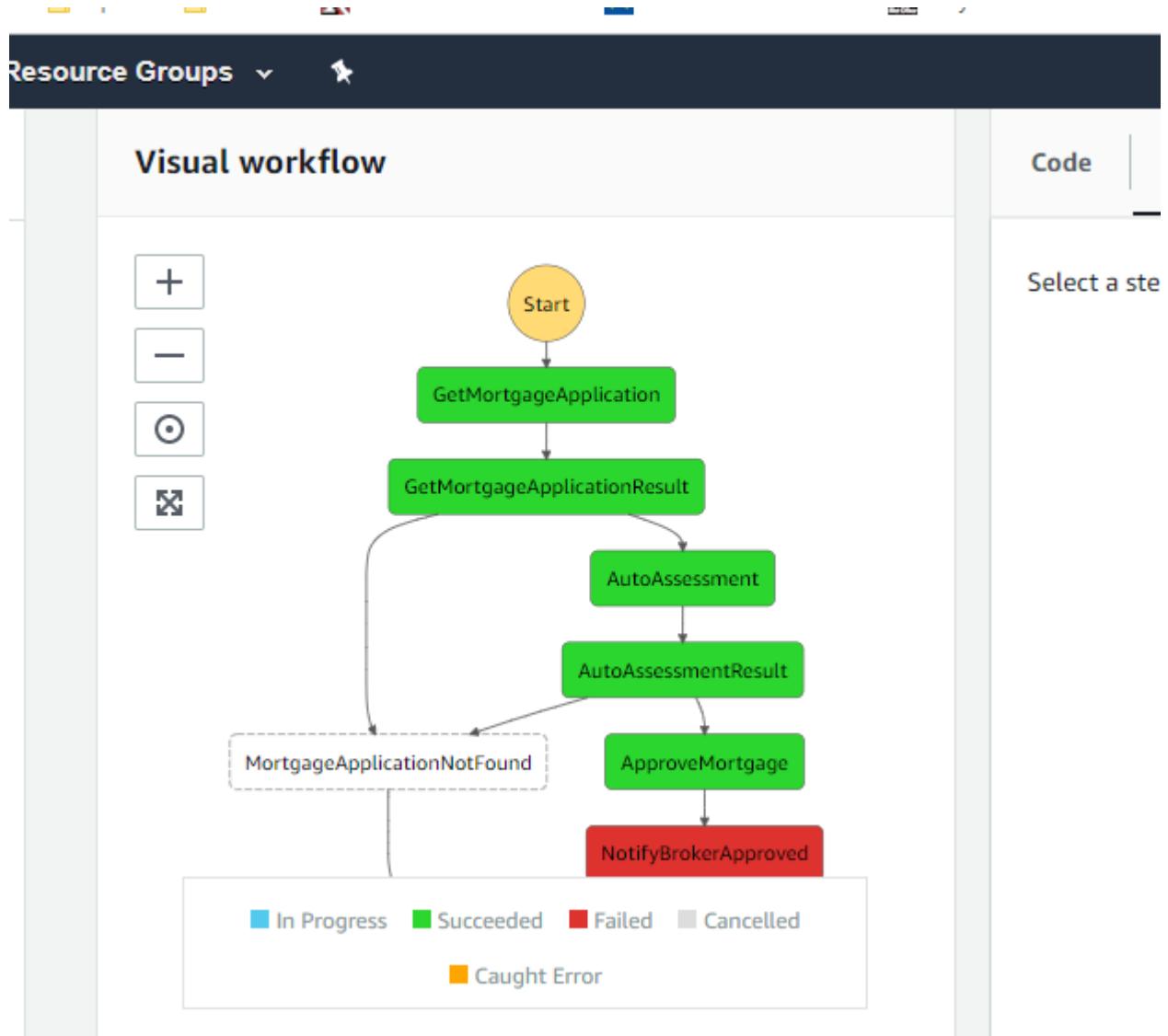


Figure AWS.AUTO.5

Another badly formatted Step Function Task input. Figure AWS.AUTO.6

▼ 17	ExecutionFailed	-	895	Nov 16, 2019 05:38:55.220 PM
{ "error": "States.Runtime", "cause": "An error occurred while executing the state 'AutoAssessmentRejection' (entered at the event id #16). The JSONPath '\$.mortgage' specified for the field 'mortgage.\$' could not be found in the input '{\"statusCode\":400,\"approved\":false,\"reason\":\"Must be earning in excess of 100000 euro for autoapproval\"}' }				

Figure AWS.AUTO.6

Once these errors are rectified then the Step Function execution will success. Figure AWS.AUTO.7

Execution details	
Execution Status	Started
 Succeeded	Nov 16, 2019 05:27:48.645 PM
Execution ARN	End Time
arn:aws:states:eu-west-1:727432808710:execution:MortgageApprovalWorkflow:29809572-eae6-7355-769a-9eda8698d9ec	Nov 16, 2019 05:27:54.104 PM
▼ Input	▼ Output
{ "mortgageId": "5000004" }	{ "statusCode": 200, "messageId": "2e2003f8-8547-5e39-b730-55b2b69ecd4c" }

Figure AWS.AUTO.5

The Step Function output correctly shows that the last Task returned a 200 status completion code and also the ID of the SNS message that was send to the users.

However another coding error show that the new state is not correctly rendered in the email. Figure

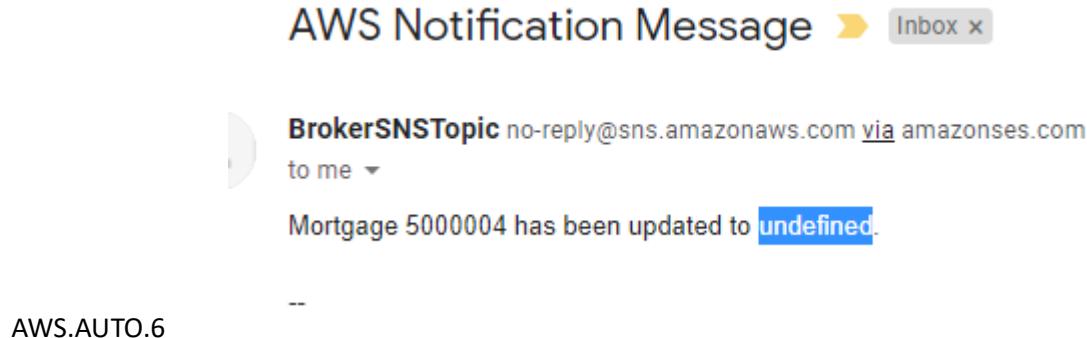


Figure AWS.AUTO.6

Once resolved we see the correct message in the email. Figure AWS.AUTO.7

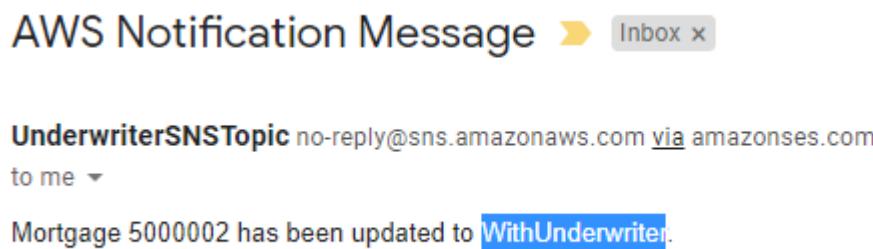


Figure AWS.AUTO.7

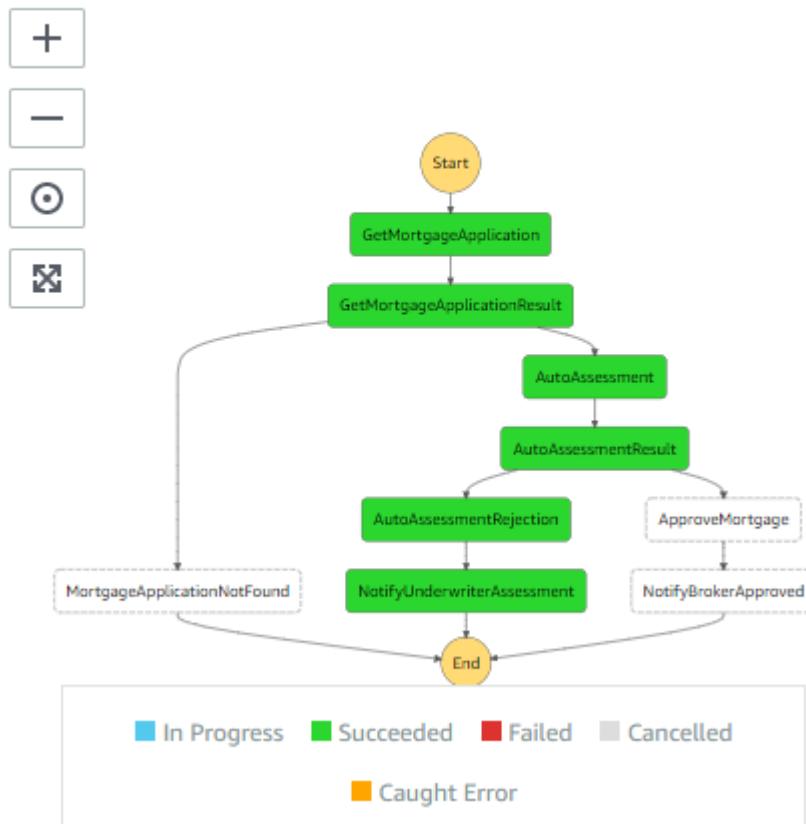
The mortgage application has also been correctly updated in DynamoDB. Figure AWS.AUTO.8

	mortgageId	customerId	employerName	loanAmount	mortgageStatus	salary
<input checked="" type="checkbox"/>	5000002	60002	Smurfit	300009	WithUnderwriter	50009
<input type="checkbox"/>	5000003	60003	Microsoft	200000	PreSubmission	100000
<input type="checkbox"/>	5000004	60003	AIB Ltd	999999	PreSubmission	100001

Figure AWS.AUTO.8

The Step Function execution shows the sequence of tasks.

Visual workflow



Auto Assessment Success

Select a Mortgage Application that meets ALL Auto Assessment criteria. Figure AWS.AUTO.9

New execution

Start an execution using the latest definition of the state machine. [Learn more](#)

Enter an execution name - optional
Enter your execution id here
29809572-eae6-7355-769a-9eda8698d9ec

Input - optional
Enter input values for this execution in JSON format

```
1 {  
2   "mortgageId": "5000004"  
3 }
```

Open in a new browser tab Cancel **Start execution**

Figure AWS.AUTO.9

Successful email SNS Notification. Figure AWS.AUTO.10

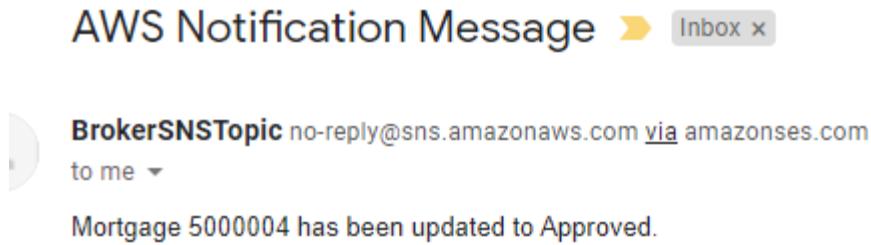


Figure AWS.AUTO.10

DynamoDB has also been updated. Figure AW.S.AUTO.11

	mortgageId	customerId	employerName	loanAmount	mortgageStatus	salary	ter
	5000002	60002	Smurfit	300009	PreSubmission	50009	39
	5000003	60003	Microsoft	200000	PreSubmission	100000	20
	5000004	60003	AIB Ltd	999999	Approved	100001	98

Figure AW.S.AUTO.11

Again the Step Function execution diagram shows the sequence of events. Figure AWS.AUTO.12

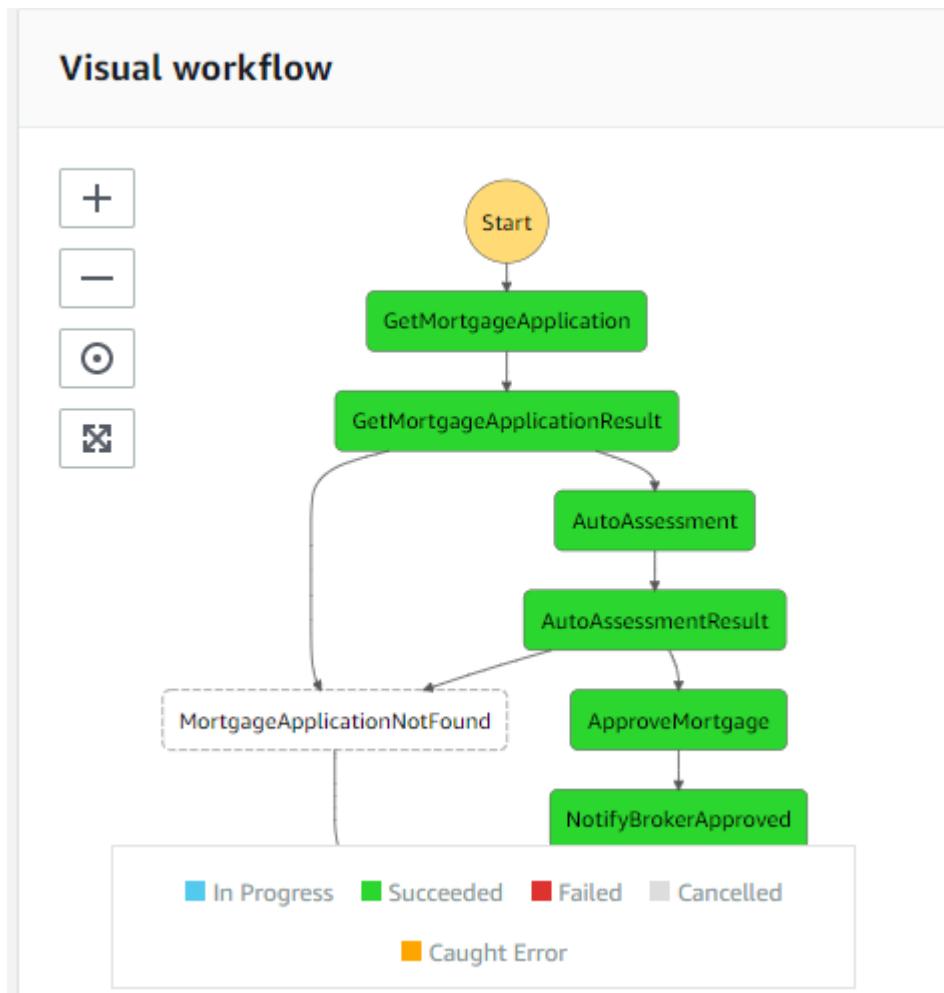


Figure AWS.AUTO.12

APPENDICES

The following sections list some key / interesting source code / template snippets.

WebSocket Cloudformation Template

```
# WEB SOCKET APIs

MortgageWebSocket:
  Type: AWS::ApiGatewayV2::Api
  Properties:
    Name: MortgageWebSocket
    ProtocolType: WEBSOCKET
    RouteSelectionExpression: "$request.body.message"
  ConnectRoute:
    Type: AWS::ApiGatewayV2::Route
    Properties:
      ApiId: !Ref MortgageWebSocket
      RouteKey: $connect
      AuthorizationType: NONE
      OperationName: ConnectRoute
      Target: !Join
        - '/'
        - - 'integrations'
        - !Ref MortgageConnectionIntegration
MortgageConnectionIntegration:
  Type: AWS::ApiGatewayV2::Integration
  Properties:
    ApiId: !Ref MortgageWebSocket
    Description: Connect Integration
    IntegrationType: AWS_PROXY
    IntegrationUri:
      Fn::Sub:
        arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${WebSocketConnectFunction.Arn}/invocations
  DisconnectRoute:
    Type: AWS::ApiGatewayV2::Route
    Properties:
      ApiId: !Ref MortgageWebSocket
      RouteKey: $disconnect
      AuthorizationType: NONE
      OperationName: DisconnectRoute
```

```
Target: !Join
  - '/'
  - - 'integrations'
  - !Ref MortgateDisconnectInteg
MortgateDisconnectInteg:
  Type: AWS::ApiGatewayV2::Integration
  Properties:
    ApiId: !Ref MortgageWebSocket
    Description: Disconnect Integration
    IntegrationType: AWS_PROXY
    IntegrationUri: 
      Fn::Sub:
        arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${WebSocketDisconnectFunction.Arn}/invocations
    SendRoute:
      Type: AWS::ApiGatewayV2::Route
      Properties:
        ApiId: !Ref MortgageWebSocket
        RouteKey: sendmessage
        AuthorizationType: NONE
        OperationName: SendRoute
        Target: !Join
          - '/'
          - - 'integrations'
          - !Ref SendInteg
SendInteg:
  Type: AWS::ApiGatewayV2::Integration
  Properties:
    ApiId: !Ref MortgageWebSocket
    Description: Send Integration
    IntegrationType: AWS_PROXY
    IntegrationUri: 
      Fn::Sub:
        arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${WebSocketSendFunction.Arn}/invocations
  Deployment:
    Type: AWS::ApiGatewayV2::Deployment
    DependsOn:
      - ConnectRoute
      - SendRoute
      - DisconnectRoute
    Properties:
      ApiId: !Ref MortgageWebSocket
Stage:
  Type: AWS::ApiGatewayV2::Stage
```

```

Properties:
StageName: Prod
Description: Prod Stage
DeploymentId: !Ref Deployment
ApiId: !Ref MortgageWebSocket

```

```

# WEBSOCKET DYNAMO DB TABLE
WebsocketConnectionsTable:
  Type: AWS::DynamoDB::Table
Properties:
  AttributeDefinitions:
    - AttributeName: "connectionId"
      AttributeType: "S"
  KeySchema:
    - AttributeName: "connectionId"
      KeyType: "HASH"
  ProvisionedThroughput:
    ReadCapacityUnits: 5
    WriteCapacityUnits: 5
  SSESpecification:
    SSEEnabled: True
  TableName: !Ref WebsocketConnectionTableName

```

```

# WEBSOCKET LAMBDA API
WebSocketConnectFunction:
  Type: AWS::Serverless::Function
Properties:
  CodeUri: websockets/onconnect/
  Handler: app.handler
  MemorySize: 256
  Runtime: nodejs10.x
  Environment:
    Variables:
      TABLE_NAME: !Ref WebsocketConnectionTableName
Policies:
  - DynamoDBCrudPolicy:
      TableName: !Ref WebsocketConnectionTableName
OnConnectPermission:
  Type: AWS::Lambda::Permission
  DependsOn:
    - MortgageWebSocket
    - WebSocketConnectFunction
Properties:
  Action: lambda:InvokeFunction

```

```
  FunctionName: !Ref WebSocketConnectFunction
    Principal: apigateway.amazonaws.com
  WebSocketDisconnectFunction:
    Type: AWS::Serverless::Function
  Properties:
    CodeUri: websockets/ondisconnect/
    Handler: app.handler
    MemorySize: 256
    Runtime: nodejs10.x
  Environment:
  Variables:
    TABLE_NAME: !Ref WebsocketConnectionTableName
  Policies:
    - DynamoDBCrudPolicy:
        TableName: !Ref WebsocketConnectionTableName
  OnDisconnectPermission:
    Type: AWS::Lambda::Permission
    DependsOn:
      - MortgageWebSocket
      - WebSocketDisconnectFunction
    Properties:
      Action: lambda:InvokeFunction
      FunctionName: !Ref WebSocketDisconnectFunction
      Principal: apigateway.amazonaws.com
  WebSocketSendFunction:
    Type: AWS::Serverless::Function
  Properties:
    CodeUri: websockets/sendmessage/
    Handler: app.handler
    MemorySize: 256
    Runtime: nodejs10.x
  Environment:
  Variables:
    TABLE_NAME: !Ref WebsocketConnectionTableName
    WEBSOCKET_ENDPOINT: !Join [ '', [ !Ref MortgageWebSocket, '.execute-api.', !Ref 'AWS::Region', '.amazonaws.com/Prod' ] ]
  Policies:
    - DynamoDBCrudPolicy:
        TableName: !Ref WebsocketConnectionTableName
    - Statement:
      - Effect: Allow
      Action:
        - 'execute-api:ManageConnections'
  Resource:
```

```

- !Sub 'arn:aws:execute-
api:${AWS::Region}:${AWS::AccountId}: ${MortgageWebSocket}/*'
  WebSocketSendPermission:
    Type: AWS::Lambda::Permission
    DependsOn:
      - MortgageWebSocket
      - WebSocketSendFunction
    Properties:
      Action: lambda:InvokeFunction
      FunctionName: !Ref WebSocketSendFunction
      Principal: apigateway.amazonaws.com

```

REFERENCE LIST AND BIBLIOGRAPHY

- [1] AWS Free Tier. <https://aws.amazon.com/free/>
- [2] AWS CLI. <https://docs.aws.amazon.com/cli/latest/userguide/install-windows.html>
- [3] AWS CLI SAM. <https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-install-windows.html>
- [4] Node JS. <https://nodejs.org/en/download/>
- [5] Configure AWS CLI. <https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html>
- [6] Visual Studio Code. <https://code.visualstudio.com/download>
- [7] Configure AWS on Visual Studio Code. <https://docs.aws.amazon.com/toolkit-for-vscode/latest/userguide/setup-toolkit.html>
- [8] Install Docker Toolbox. https://docs.docker.com/toolbox/toolbox_install_windows/
- [9] DynamoDB Cloudformation Template.
<https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.dynamo/dynamo-create.yml>
- [10] Lambda Cloudformation Template.
<https://github.com/GeneralYeager/tallaght.mortgages/blob/master/tud.mortgages/Tallaght-Mortgages-App/template.yaml>
- [11] Step Function State Machine Template.
<https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.workflow/StateMachineV3.txt>
- [12] <https://aws.amazon.com/blogs/compute/announcing-websocket-apis-in-amazon-api-gateway/>

Dublin Technological University,
Certificate, Cloud Solutions Architecture (2020)

Implementation Iteration 2.1
and
Implementation Iteration 2.2 (Page 28)

Student Name: Alan Duffin

Student ID: X00159409

SUBMISSION MATERIAL

PLEASE visit the video for demonstration of progress.

<https://s3-eu-west-1.amazonaws.com/appd.cloud.project/IT-Tallaght-Mortgage-Workflow-WebApp-2019-11-23+13-54-23.mp4>

Source Code Repository Details

Source Code NodeJS / Lambda	<u>https://github.com/GeneralYeager/tallaght.mortgages/tree/master/tud.mortgages/Tallaght-Mortgages-App</u>
Workflow Cloudformation	<u>https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.workflow/StateMachineV3.txt</u>
DynamoDB Cloudformation	<u>https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.dynamo/dynamo-create.yml</u>
Lambda / API Gateway Cloudformation	<u>https://github.com/GeneralYeager/tallaght.mortgages/blob/master/tud.mortgages/Tallaght-Mortgages-App/template.yaml</u>
Angular SPA Source Code	<u>https://github.com/GeneralYeager/tallaght.mortgages/tree/master/mort-app-web</u>

Table A.

ITERATION 2.1 TASKS

Building upon last weeks progress report, this document outlines the additional project tasks that have been completed for implementation phase 2. The follow tasks were completed this week.

Project Phase	Service / Area	Implementation Task
Build / Development	AWS Websockets	Human interaction requires notification with User Interface.
Build / Development	AWS Lambda	Build Lambda functions to manage API Gateway Websockets and notifications.
Build / Development	DynamoDB	Build table to manage API Websocket connection tokens.
Build / Development	Angular SPA	Build Browser based Front End to allow users to create/modify/submit Mortgage Applications for workflow assessment.
Test		Test Websocket Notification Events and subscriptions
Build / Development	AWS Lambda	Additiona Lambda functions to support Step Function execution.
Build / Development	AWS Step Functions	Extend Step Function state machine to support Human decision making.
Design	AWS Step Functions	Integrate Step Function generation into Cloudformation template.
Test	AWS Step Functions	Test State Transitions / Events

Table B.

I have extended the suite of Lambda services that will allow a the Step function state machine to transition Mortgage application states and allow notification to different user categories (Brokers and Underwriters). In order to support this I have had to build an intial Browser based Single Page Application using the Javascript Angular framework. At this point Brokers can create and amend mortgage applications. I have built Angular services that connect to the AWS API Gateway Websocket API that allows traffic to be initiated either by the client or the server. This is necessary so that uses can be notified of Mortgage application state changes. So users will be notified of State changes and required tasks via two channels SNS (email) and real time browser based websockets.

It has been necessary to build additional Lambda functions to support the integration between the Step Function State Machine and the API Gateway Websocket layer. Also some additional Lambda functions were developed specifically to transition the Mortgage Applications between states.

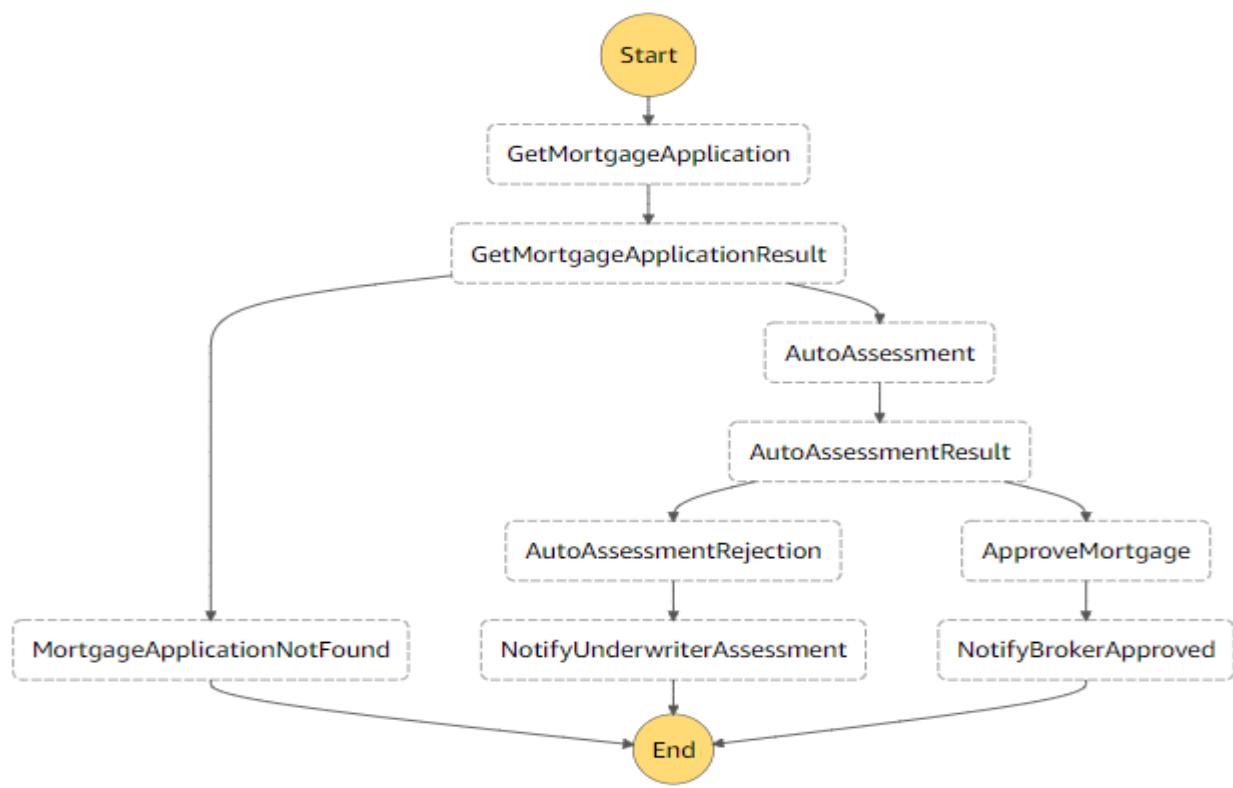
PLAN FOR ITERATION 2.2

Project Phase	Service / Area	Implementation Task
Build / Development	AWS Step Functions	Complete the Step Function Workflow state machine.
Build / Development	AWS Cloudformation	Integrate the deployment of the Workflow state machine into the project Cloud formation template.
Build / Development	AWS Lamda	Build the Lambda Functions to support pausing the workflow to await human decisions. Also manage the Step Function tokens.
Build / Development	DynamoDB	Implement new DB schema to map mortgage application to Step Function execution id. This is required to allow users approve or reject applications without reference to the step function execution id.
Build / Development	Angular	Extends the Browser application to allow submission of mortgages for assessment. Allow for clarification requests and Underwriter decision making.
Test	AWS Step Functions	Test State Transitions / Events / Human interaction

Table C.

The workflow state machine currently exists as seen in Figure AWS.STEP.1

Figure AWS.STEP.1



SERVLERLESS / LAMBDA FUNCTIONS

The following Lambda functions have now been completed as of this progress report. Figure AWS.LAMBDA.1

Mortgages-Stack-AutoAssessmentFunction-S230S392YHS1
Mortgages-Stack-ChangeMortgageStatusFunction-2NK931AA2H2N
Mortgages-Stack-CreateCustomerFunction-NC15CDP9GDUI
Mortgages-Stack-CreateMortgageFunction-ZC5021VBH5GA
Mortgages-Stack-DeleteMortgageFunction-L0XFGM8ONTYQ
Mortgages-Stack-GetCustomerFunction-P1ADM0V52KA0
Mortgages-Stack-GetMortgageFunction-14BLETUMUYZEJ
Mortgages-Stack-ListByStatusMortgageFunction-SFRBQ8ABULIO
Mortgages-Stack-MortgageSNSFunction-1TAXBKNV0FSUX
Mortgages-Stack-MortgageStatusSNSFunction-5UVPVE4OU9PF
Mortgages-Stack-NotifyUsersSNSFunction-216EXFJBHO0V
Mortgages-Stack-StartWorkflowFunction-4H7HZIPCE3XX
Mortgages-Stack-UpdateCustomerFunction-QXUSA0YPACZE
Mortgages-Stack-UpdateMortgageFunction-15LUWGKY2ZX2Q
Mortgages-Stack-WebSocketConnectFunction-1GH8JPQCLA8YJ
Mortgages-Stack-WebSocketDisconnectFunction-1PPRHIDN8QRG3
Mortgages-Stack-WebSocketSendFunction-H1E7UBOP0KFV
Mortgages-Stack-WorkflowGetCustomerFunction-1JZFBVZ6QABZG
Mortgages-Stack-WorkflowGetMortgageFunction-16L3UM67HNU4N
Mortgages-Stack-WorkflowUpdateMortgageFunction-1EKLNK5DXIDZG

Figure AWS.LAMBDA.1

This table lists the purpose of each of the completed Lambda Functions. Table AWS.LAMBDA.1

Lambda Function	Purpose
Mortgages-Stack-AutoAssessmentFunction	Perform Mortgage Rules check
Mortgages-Stack-ChangeMortgageStatusFunction	Modify the Mortgage Application Status and store.
Mortgages-Stack-CreateCustomerFunction	HTTP API Create Customer
Mortgages-Stack-CreateMortgageFunction	HTTP API Create Mortgage Application
Mortgages-Stack>DeleteMortgageFunction	HTTP API Delete Mortgage Application
Mortgages-Stack-GetCustomerFunction	HTTP API Find By PK Customer
Mortgages-Stack-GetMortgageFunction	HTTP API Find By PK Mortgage Application
Mortgages-Stack-ListByStatusMortgageFunction	HTTP API Find By Status Mortgage Application
Mortgages-Stack-MortgageStatusSNSFunction	Step Function Update Mortgage Status
Mortgages-Stack-NotifyUsersSNSFunction	Step Function Notify State Change
Mortgages-Stack-StartWorkflowFunction	HTTP API Execute Mortgage Workflow
Mortgages-Stack-UpdateCustomerFunction	HTTP API Update Customer
Mortgages-Stack-UpdateMortgageFunction	HTTP API Update Mortgage
Mortgages-Stack-WebSocketConnectFunction	Manage Websocket connection and store token.
Mortgages-Stack-WebSocketDisconnectFunction	Manage Websocket disconnect and delete token.
Mortgages-Stack-WebSocketSendFunction	Step Function send message to API Gateway Websocket for Browser broadcast.
Mortgages-Stack-WorkflowGetCustomerFunction	Step Function Find By PK Customer
Mortgages-Stack-WorkflowGetMortgageFunction	Step Function Find By PK Mortgage
Mortgages-Stack-WorkflowUpdateMortgageFunction	Step Function Update Mortgage

Table AWS.LAMBDA.1

The source code and deployment template for these functions are available at the GitHub repository listed in Table A or in the Appendix.

I have had to implement new Lambda functions to manage API Gateway Websocket functionality.

API GATEWAY WEBSOCKETS

Last year AWS announced API Gateway support for the Websocket protocol [12]. This allows for the creation of ...

"bidirectional communication applications using WebSocket APIs in Amazon API Gateway without having to provision and manage any servers.

HTTP-based APIs use a request/response model with a client sending a request to a service and the service responding synchronously back to the client. WebSocket-based APIs are bidirectional in nature. This means that a client can send messages to a service and services can independently send messages to its clients.

This bidirectional behavior allows for richer types of client/service interactions because services can push data to clients without a client needing to make an explicit request. WebSocket APIs are often used in real-time applications such as chat applications, collaboration platforms, multiplayer games, and financial trading platforms."

The following API Gateway Websocket API has been created. Figure AWS.API.1

The screenshot shows the AWS API Gateway console interface. At the top, there's a navigation bar with the 'Amazon API Gateway' logo, followed by 'APIs > MortgageWebSocket (h9uk1z65s6) > Routes'. On the left, a sidebar lists 'APIs', 'Custom Domain Names', and an 'API: MortgageWebSoc...' entry which is expanded to show 'Routes', 'Stages', and 'Authorizers'. The main content area is titled 'Routes' and contains an 'Actions' dropdown. It shows a 'Route Selection Expression' set to '\$request.body.message' with a pencil icon for editing. Below it is a 'New Route Key' input field. A list of route keys is shown: '\$connect', '\$disconnect', 'sendmessage', and '\$default' (preceded by a plus sign). The entire interface has a light gray background with blue and orange highlights for selected items.

Figure AWS.API.1

The screenshot shows the AWS API Gateway interface. At the top, it displays the path: APIs > MortgageWebSocket (h9uk1z65s6) > Stages > Prod. On the right, there are buttons for 'Show all hints' and a question mark icon. Below this, the 'Prod Stage Editor' title is centered, with 'Stages' and 'Create' buttons on the left and 'Delete Stage' and 'Configure' buttons on the right. A sidebar on the left shows 'Prod' under the stages. The main content area has a blue header bar with the text 'WebSocket URL: wss://h9uk1z65s6.execute-api.eu-west-1.amazonaws.com/Prod' and 'Connection URL: https://h9uk1z65s6.execute-api.eu-west-1.amazonaws.com/Prod/@connections'. Below this are tabs for 'Settings' (which is selected), 'Logs/Tracing', 'Stage Variables', and 'Deployment History'. The 'Default Route Throttling' section contains a note about account-level throttling rates (10000 messages per second with a burst of 5000). There is also an 'Enable throttling' checkbox and a help icon.

Figure AWS.API.2

Because API Gateway support for Websockets is very recent, the online help is minimal. It took many hours to get the deployment and development correct.

There were many errors encountered such as incorrect endpoint format and futile attempts to get Lambda functions to establish wss protocol connections.

This error was caused by an invalid endpoint format.

The screenshot shows the AWS Lambda function execution results for 'Mortgages-Stack-WebSocketSend...'. The top navigation bar includes 'Throttle', 'Qualifiers ▾', 'Actions ▾', 'TestWebsocketSend ▾', 'Test', and a save icon. The main area shows a green success icon and the message 'Execution result: succeeded (logs)'. A 'Details' section is expanded, showing the message 'The area below shows the result returned by your function execution. Learn more about returning results from your function.' Below this is a large code block containing a JSON object with a 'statusCode': 500 and a detailed error message about an 'UnknownEndpoint' error related to the 'wss' protocol.

```
{
  "statusCode": 500,
  "body": "UnknownEndpoint: Inaccessible host: `wss'. This service may not be available in the `eu-west-1` region.\n      at Request.callListeners\n      (/var/runtime/node_modules/aws-sdk/lib/event_listeners.js:494:46)\n      at Request.emit\n      (/var/runtime/node_modules/aws-sdk/lib/sequential_executor.js:106:20)\n      at Request.emit\n      (/var/runtime/node_modules/aws-sdk/lib/sequential_executor.js:78:10)\n      at Request.emit\n      (/var/runtime/node_modules/aws-sdk/lib/request.js:683:14)\n      at ClientRequest.error\n      (/var/runtime/node_modules/aws-sdk/lib/event_listeners.js:333:22)\n      at ClientRequest.<anonymous>\n      (/var/runtime/node_modules/aws-sdk/lib/http/node.js:96:19)\n      at ClientRequest.emit\n      (events.js:198:13)\n      at ClientRequest.EventEmitter.emit\n      (domain.js:448:20)\n      at TLSSocket.socketErrorListener\n      (_http_client.js:392:9)\n      at TLSSocket.emit\n      (events.js:198:13)"
}
```

Figure AWS.SOCKET.1

It is not possible for Lambda functions to connect to an external WSS endpoint. They must directly use the 'AWS.ApiGatewayManagementApi' sdk and connect via the API Gateway instance.

```
395     Runtime: nodejs10.x
396     Environment:
397       Variables:
398         TABLE_NAME: !Ref WebSocketConnectionTableName
399         # WEB SOCKET _ENDPOINT: !Join [ '', [ 'wss://', !Ref MortgageWebSocket, '.execute-api.', !Ref 'A
400         WEB SOCKET _ENDPOINT: !Join [ '', [ !Ref MortgageWebSocket, '.execute-api.', !Ref 'AWS::Region
401       Policies:
402         !Ref DyanmicRPCcloudPolicy

```

Figure AWS.SOCKET.2

Websocket Success. Figure AWS.SOCKET.3



Tallaght Mortgages Underwriters Page

Home
underwriter-component works!

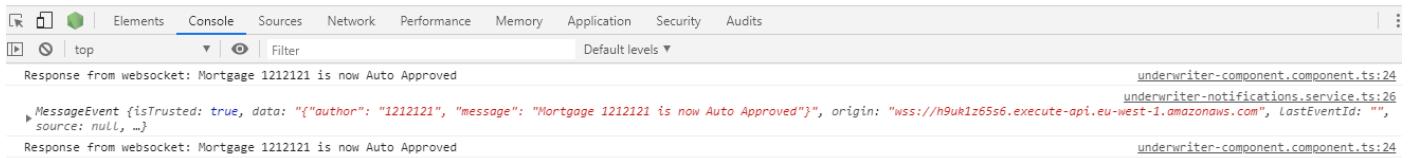


Figure AWS.SOCKET.3

Socket Broadcast Source Code:

```
exports.handler = async (event, context) => {

  let connectionData;
  console.log(event);
  try {
    connectionData = await ddb.scan({ TableName: TABLE_NAME, ProjectionExpression: 'connectionId' }).promise();
  } catch (e) {
    return { statusCode: 500, body: e.stack };
  }

  const apigwManagementApi = new AWS.ApiGatewayManagementApi({
    apiVersion: '2018-11-29',
    endpoint: WEBSOCKET_ENDPOINT //event.requestContext.domainName + '/' + event.requestContext.stage
  });

  //const postData = JSON.parse(event.body).data;
  const postData = '{\"author\": \"1212121\", \"message\": \"Mortgage 1212121 is now Auto Approved\"}';

  const postCalls = connectionData.Items.map(async ({ connectionId }) => {
    try {
      await apigwManagementApi.postToConnection({ ConnectionId: connectionId, Data: postData }).promise();
    } catch (e) {
      if (e.statusCode === 410) {
        console.log(`Found stale connection, deleting ${connectionId}`);
        await ddb.delete({ TableName: TABLE_NAME, Key: { connectionId } }).promise();
      } else {
        throw e;
      }
    }
  });

  try {
    await Promise.all(postCalls);
  } catch (e) {
    return { statusCode: 500, body: e.stack };
  }
}
```

```
    return { statusCode: 200, body: 'Data sent.' };
};
```

Web Browser Websocket Connection Source Code:

```
import { Injectable } from '@angular/core';

import { Observable, Observer, Subject } from 'rxjs'

@Injectable({
  providedIn: 'root'
})
export class WebsocketService {
  constructor() {}

  private subject: Subject<MessageEvent>;

  public connect(url): Subject<MessageEvent> {
    console.log("connect url");
    if (!this.subject) {
      this.subject = this.create(url);
      console.log("Successfully connected: " + url);
    }
    return this.subject;
  }

  private create(url): Subject<MessageEvent> {
    let ws = new WebSocket(url);
    console.log("create ws");
    let observable = Observable.create((obs: Observer<MessageEvent>) => {
      console.log("in obs");
      //ws.onmessage = obs.next.bind(obs);
      ws.onmessage = function(event) {
        console.log("WebSocket message received:", event);
        obs.next(event);
      };
      ws.onerror = obs.error.bind(obs);
      ws.onclose = obs.complete.bind(obs);
      return ws.close.bind(ws);
    });
    let observer = {
      next: (data: Object) => {
        console.log("in next");
        if (ws.readyState === WebSocket.OPEN) {
          ws.send(JSON.stringify(data));
        }
      }
    };
    return observable;
  }
}
```

```
        }
    }
};

return Subject.create(observer, observable);
}
}
```

ANGULAR SINGLE PAGE APPLICATION

In order to build a Javascript based Bowser application to demonstrate and interact with the Step Function Workflow I have built a first iteration of an Angular SPA application.

Steps to set up the development environment are:

- 1 Install npm. <https://nodejs.org/en/>
- 2 Install Angular CLI. <https://angular.io/guide/setup-local>
- 3 Install Angular Material Design. <https://material.angular.io/guide/getting-started>
- 4 Install Websocket NPM Package. <https://www.npmjs.com/package/websocket>
- 5 The source code completed so far is stored in Git Hub.

At the moment the SPA allows users to create / modify / list mortgage application. View Websocket notifications.

The look and feel is very basic but will be sufficient to provide a working system.

Run the SPA locally – shown below.

```
C:\Users\User\aws\sam\git\tallaght.mortgages\mort-app-web>ng serve
Your global Angular CLI version (8.3.18) is greater than your local
version (8.3.17). The local Angular CLI version is used.

To disable this warning use "ng config --cli.warnings.versionMismatch false".
10% building 3/3 modules 0 active i ?wds?: Project is running at http://localhost:4200/webpack-dev-server/
i ?wds?: webpack output is served from /
i ?wds?: 404s will fallback to //index.html

chunk {main} main.js, main.js.map {main} 77.4 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map {polyfills} 264 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map {runtime} 6.15 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map {styles} 338 kB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map {vendor} 6.97 MB [initial] [rendered]
Date: 2019-11-17T14:58:39.235Z - Hash: 1c4a25c725afce1cf961 - Time: 55294ms
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **
i ?wdm?: Compiled successfully.
```

Figure AWS.ANGULAR.1

Home Page. Figure AWS.ANGLAR.2

The screenshot shows a web browser window with the URL `localhost:4200/home`. The page title is "Tallaght Mortgages Home Page". Below the title, there are two tabs: "Brokers" (selected) and "Underwriters". The main content area displays a table titled "Tallaght Mortgages Brokers Page". The table has a header row with columns: mortgagelId, customerId, employerName, loanAmount, mortgageStatus, salary, term, and yearsInEmployment. There are five data rows below the header, each representing a mortgage broker. The data is as follows:

mortgagelId	customerId	employerName	loanAmount	mortgageStatus	salary	term	yearsInEmployment
5000005	60002	Ryanair	222000	PreSubmission	22001	22	22
5000004	60003	AIB Ltd	999999	PreSubmission	100001	91	99
5000002	60002	Smurfit	300009	WithUnderwriter	50009	39	29
5000003	60003	Microsoft	200000	PreSubmission	100000	20	11

Broker Page. Figure AWS.ANGLAR.3

Modify Mortgages. Figure AWS.ANGULAR.4

The screenshot shows a web application for managing mortgages. At the top, there's a navigation bar with links for 'Apps', 'Allergy', 'Banking', 'IT', 'Sports', 'Twitter', and a logo. Below the navigation is a large header 'Tallaght Mortgages Broker'. Underneath the header is a table with columns: mortgageId, customerId, employerName, loanAmount, mortgageStatus, salary, term, and yearsInEmployment. There are two rows of data. The first row corresponds to the modal dialog, which contains the message 'localhost:4200 says Update Completed' and an 'OK' button. The second row shows data for a mortgage with customerId 60002, employerName Ryanair, loanAmount 222000, mortgageStatus PreSubmission, salary 22001, term 22, and yearsInEmployment 23. Below this table is another table with columns: Loan Amount, Term (Years), and two rows of data (222000, 22). At the bottom left is a 'Save' button. Below the tables is a section titled 'Save' containing three more rows of data: 5000004, 60003, AIB Ltd, 999999, PreSubmission, 100001, 91, 99; 5000002, 60002, Smurfit, 300009, WithUnderwriter, 50009, 39, 29; and 5000003, 60003, Microsoft, 200000, PreSubmission, 100000, 20, 11.

Recieve Mortgage Status Notifications. Figure AWS.LAMBDA.5

The screenshot shows a web application for underwriting mortgages. At the top, there's a green notification bar with the message 'Mortgage 1212121 is now Auto Approved'. Below the notification is a heading 'Tallaght Mortgages Underwriters Page'. Underneath the heading is a sub-section with the text 'underwriter-component works!'. The rest of the page is mostly blank.

Cross-Origin Resource Sharing ([CORS](#)) is a mechanism that uses additional [HTTP](#) headers to tell browsers to give a web application running at one [origin](#) access to selected resources from a different origin. A web application executes a **cross-origin HTTP request** when it requests a resource that has a different origin (domain, protocol, or port) from its own.

<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

The AWS documentation enabling CORS on API Gateway via Cloudformation is lacking. It took a couple of days for me to get the HTTP API calls working from my local development environment.

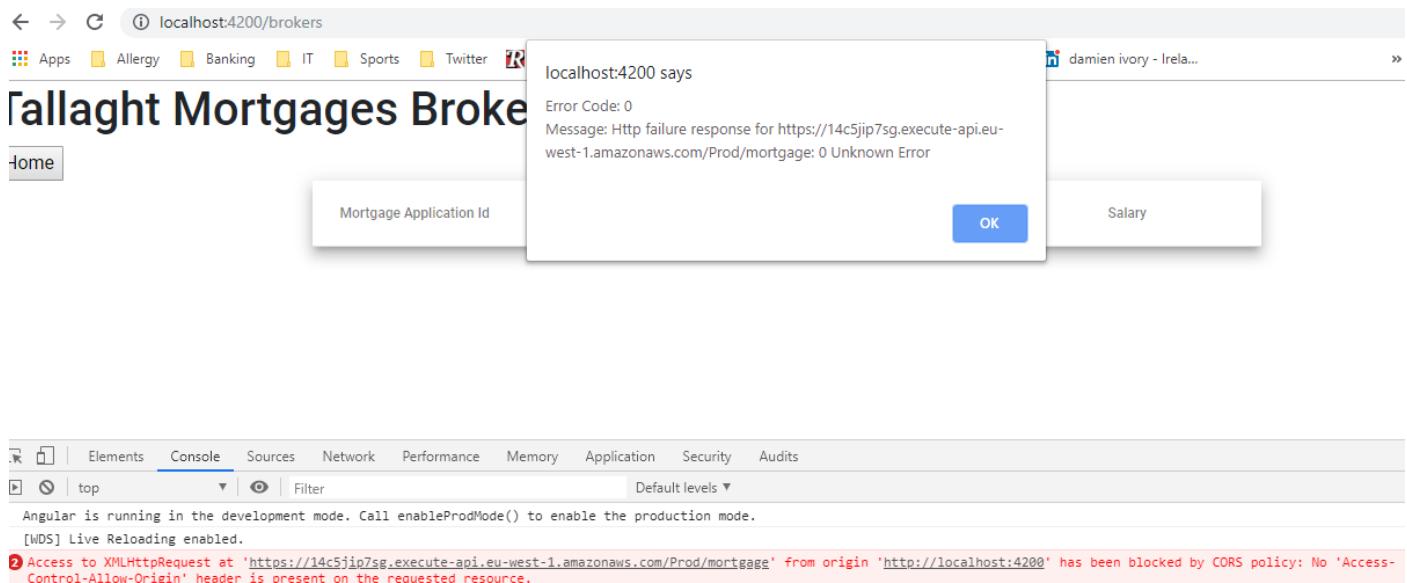


Figure AWS.CORS.1

Attempt to enable CORS post deployment.

The screenshot shows the AWS API Gateway console interface. The top navigation bar indicates the path: APIs > Mortgages-Stack (14c5jp7sg) > Resources > /mortgag... . On the left, a sidebar menu lists various API components: APIs, Mortgages-Stack, Resources (which is selected), Stages, Authorizers, Gateway Responses, Models, Resource Policy, Documentation, Dashboard, Settings, and MortgageWebSocket. The main content area displays a hierarchical tree of resources under the path /mortgag... . The tree includes nodes for /, /customer, /{id}, and /workflow. Under each node, there are METHOD ACTIONS (Edit Method Documentation, Delete Method), RESOURCE ACTIONS (Create Method, Create Resource, Enable CORS, Edit Resource Documentation, Delete Resource), and API ACTIONS (Deploy API, Import API, Edit API Documentation, Delete API). A context menu is open over the /mortgag... node, specifically over the GET method. This menu also lists the same three categories of actions. To the right of the tree, there are two side panels: 'Method I...' which shows 'Auth: NOI' and 'ARN: arn:aws:apigateway:us-east-1::method/...'; and 'Method R...' which says 'Select an i...'. The title bar of the browser window also reflects the current URL: 'Amazon API Gateway | APIs > Mortgages-Stack (14c5jp7sg) > Resources > /mortgag... '.

Figure AWS.CORS.2

The screenshot shows the AWS Lambda API Gateway CORS configuration page. On the left, a tree view lists resources: / (customer, {id}, mortgage). Under /mortgage, methods are listed: GET, OPTIONS, PUT, and a sub-item {id} with DELETE and GET. On the right, the 'Enable CORS' configuration panel is shown. It includes sections for 'Gateway Responses for Mortgages-Stack API' (with checkboxes for DEFAULT 4XX and DEFAULT 5XX), 'Methods' (checkboxes for GET, OPTIONS, PUT), 'Access-Control-Allow-Methods' (set to GET, OPTIONS, PUT), 'Access-Control-Allow-Headers' (set to 'Content-Type,X-Amz-Date,Authorization'), and 'Access-Control-Allow-Origin*' (set to '*'). A 'Advanced' section is collapsed. At the bottom is a blue button labeled 'Enable CORS and replace existing CORS headers'.

Figure AWS.CORS.3

The screenshot shows the AWS Lambda API Gateway CORS configuration page for the /mortgage resource. The left sidebar shows the same resource structure as Figure AWS.CORS.3. The right panel displays a summary of successful configurations: 'Add Access-Control-Allow-Headers, Access-Control-Allow-Methods, Access-Control-Allow-Origin Method Response Headers to OPTIONS method' (green checkmark) and 'Add Access-Control-Allow-Headers, Access-Control-Allow-Methods, Access-Control-Allow-Origin Integration Response Header Mappings to OPTIONS method' (green checkmark). Below this, there are four error messages with red exclamation marks: 'Add Access-Control-Allow-Origin Method Response Header to GET method', 'Add Access-Control-Allow-Origin Integration Response Header Mapping to GET method', 'Add Access-Control-Allow-Origin Method Response Header to PUT method', and 'Add Access-Control-Allow-Origin Integration Response Header Mapping to PUT method'. A note at the bottom states: 'Your resource has been configured for CORS. If you see any errors in the resulting output above please check the error message and if necessary attempt to execute the failed step manually via the Method Editor.'

Figure AWS.CORS.4

Some sites indicated that the correct Cloudformation format to enable CORS for API Gateway as shown below.

```
! template.yaml template.yaml
1 AWSTemplateFormatVersion: '2010-09-09'
2 Transform: AWS::Serverless-2016-10-31
3 Description: >
4 | Tallaght-Mortgages-App
5
6 Globals:
7 Function:
8 | Timeout: 3
9 Api:
10 | EndpointConfiguration: REGIONAL
11 | Cors: "*"
12 |
```

Figure AWS.CORS.5

After much web surfing and trial and error the correct template is shown below:

```
! template.yaml template.yaml
1 AWSTemplateFormatVersion: '2010-09-09'
2 Transform: AWS::Serverless-2016-10-31
3 Description: >
4 | Tallaght-Mortgages-App
5
6 Globals:
7 Function:
8 | Timeout: 3
9 Api:
10 | EndpointConfiguration: REGIONAL
11 | Cors:
12 | | AllowMethods: "*"
13 | | AllowHeaders: "'Content-Type'"
14 | | AllowOrigin: "*"
15
16 Parameters:
```

Figure AWS.CORS.6

API Gateway with Lambda integration (rather than HTTP integration) does not fully abstract out CORS functionality. Individual Lambda function must add CORS headers as seen in Figure AWS.CORS.7

```
const findMortgageResponse = await dynamoDb.scan(findAllMortgages).promise();
console.log(findMortgageResponse);
return {
  statusCode: 200,
  headers: [ 'Content-Type': 'application/json', "Access-Control-Allow-Origin": "*" ],
  body: JSON.stringify(findMortgageResponse.Items)
};
```

Figure AWS.CORS.7

DEVELOPMENT – WORKFLOW IMPLEMENTATION – STEP FUNCTION.

The second iteration of the Step Function State Machine was shown in Figure AWS.STEP.1

The workflow has been extended only slightly this week but in the coming week I hope to have completed the workflow by adding the following steps:

- 1 Add Clarification, Rejection and Approval processes based on the human underwriter decision.
- 2 Pause the workflow when the Mortgage Application is assigned to a human underwriter.
- 3 Pause the workflow when the Mortgage Application is returned to the Broker for clarification.
- 4 Associate the workflow restart token with the mortgage Id and persist to DynamoDB.
- 5 Complete all of the Websocket and SNS notification paths.
- 6 Integrate with the Browser SPA, allow users to start and restart the workflow process.

The small enhancements that I have completed for the state machine this week have allowed me to become more familiar with the Step Function functionality and notation. It is very important to have a clear design for the input and outputs of each Task/Lambda function/method and the transformations that are required between Tasks to allow subsequent tasks in the workflow to access the required data.

The testing and bug fixing that I performed is summarised now in the following snapshots of two scenarios : Auto Assessment rejection and Auto Assessment success.

Auto Assessment Rejection

Submit a Mortgage Application that does not meet the Auto Assessment requirements. Figure AWS.AUTO.1

New execution

Start an execution using the latest definition of the state machine. [Learn more](#)

Enter an execution name - optional
Enter your execution id here

Input - optional
Enter input values for this execution in JSON format

```

1 * {
2     "mortgageId": "5000002"
3 }
```

Open in a new browser tab Cancel **Start execution**

Figure AWS.AUTO.1

Invalid JSON path.

US.S0.S 1.402 PM			
▼ 17	ExecutionFailed	-	Nov 16, 2019 05:30:31.402 PM
<pre>{ "error": "States.Runtime", "cause": "An error occurred while executing the state 'MortgageApplicationNotFound' (entered at the event id #16). The JSONPath '\$.error' specified for the field 'Message.\$' could not be found in the input '{\"statusCode\":400,\"approved\":false,\"reason\":\"Must be earning in excess of 100000 euro for autoapproval\"}'" }</pre>			

The InputPath on the Task must isolate the required JSON node to provide the correct input to the Lambda function. Figure AWS.AUTO.2

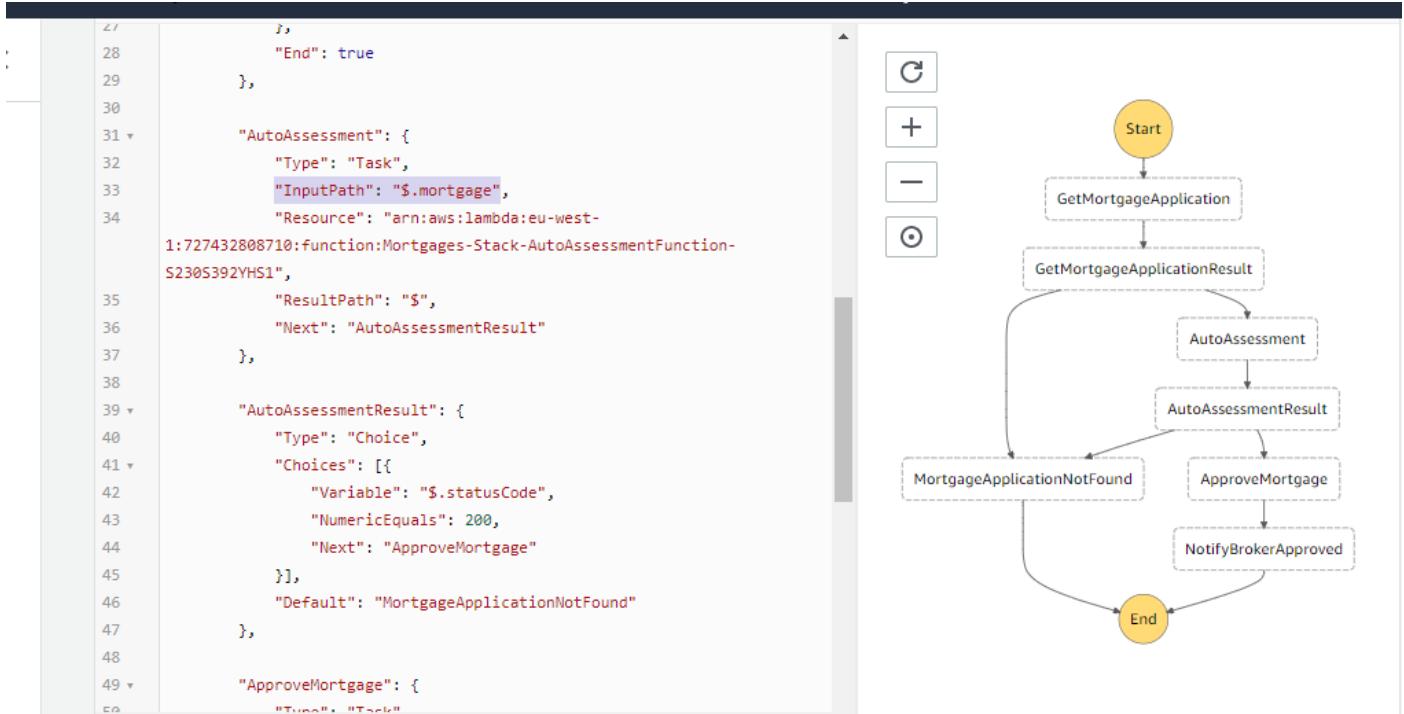


Figure AWS.AUTO.2

Also the Lambda Function must return any and all data that is required for the next task in the chain. Figure AWS.AUTO.3

The screenshot shows a code editor for a Lambda function. The code includes a conditional return statement that provides specific data for different salary levels.

```
        }
    }
    if (mortgageApplication.salary != undefined && mortgageApplication.salary != null) {
        let salary = toInt(mortgageApplication.salary)
        if (100000 > salary) {
            return {
                statusCode: 400,
                approved: false,
                reason: 'Must be earning in excess of 100000 euro for autoapproval',
                mortgage: mortgageApplication
            };
        }
    }
```

Figure AWS.AUTO.3

If the input and output data is not correctly mediated then the Step Function execution will fail. Figure AWS.AUTO.4

New execution X

Start an execution using the latest definition of the state machine. [Learn more](#)

Enter an execution name - optional
Enter your execution id here

Input - optional
Enter input values for this execution in JSON format

```
1 * {  
2     "mortgageId": "5000002"  
3 }
```

Open in a new browser tab Cancel **Start execution**

Figure AWS.AUTO.4

The AWS Console will highlight failed states as shown below. Figure AWS.AUTO.5

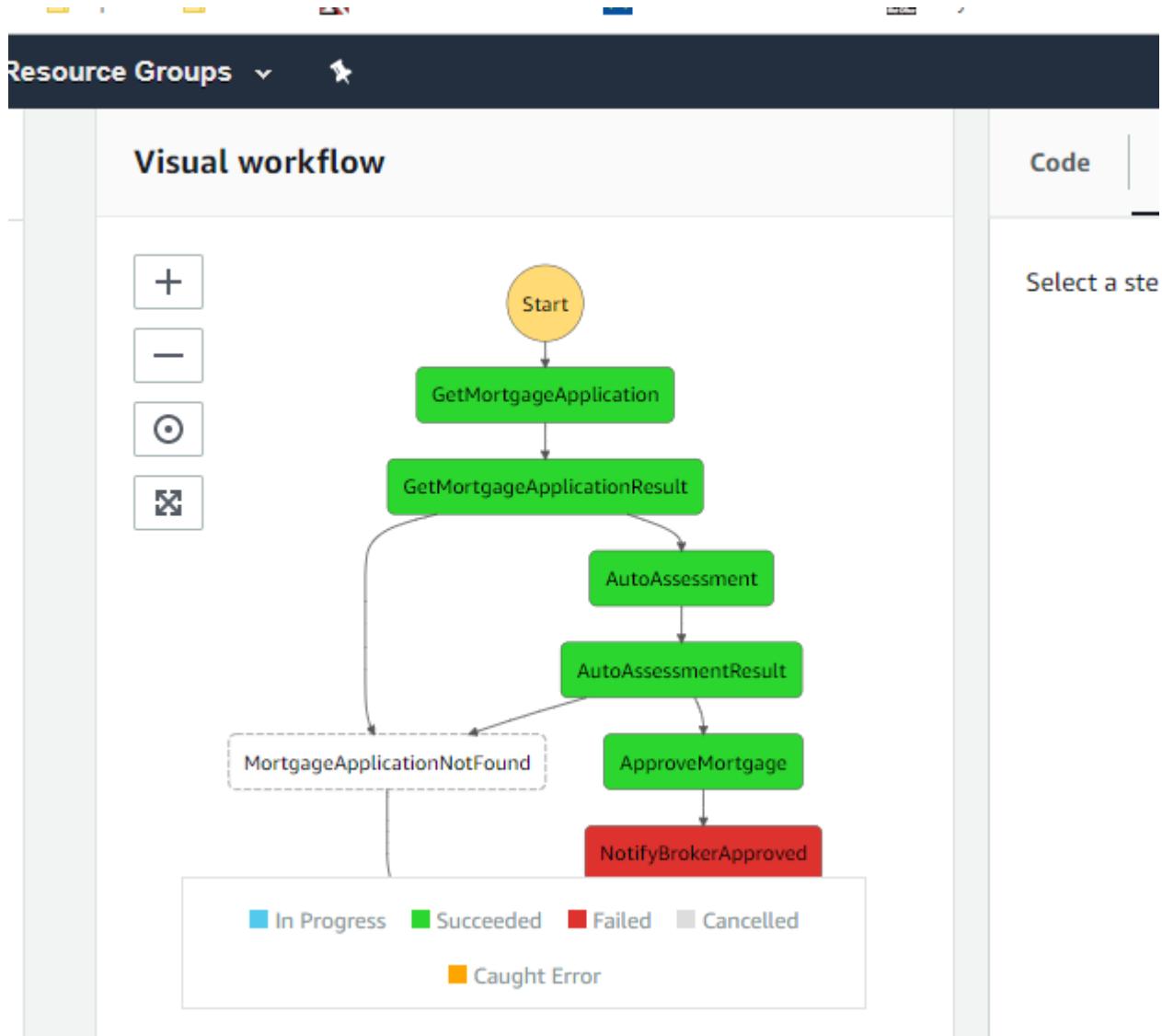


Figure AWS.AUTO.5

Another badly formatted Step Function Task input. Figure AWS.AUTO.6

▼ 17	ExecutionFailed	-	895	Nov 16, 2019 05:38:55.220 PM
{ "error": "States.Runtime", "cause": "An error occurred while executing the state 'AutoAssessmentRejection' (entered at the event id #16). The JSONPath '\$.mortgage' specified for the field 'mortgage.\$' could not be found in the input '{\"statusCode\":400,\"approved\":false,\"reason\":\"Must be earning in excess of 100000 euro for autoapproval\"}' }				

Figure AWS.AUTO.6

Once these errors are rectified then the Step Function execution will success. Figure AWS.AUTO.7

Execution details	
Execution Status	Started
<input checked="" type="checkbox"/> Succeeded	Nov 16, 2019 05:27:48.645 PM
Execution ARN	End Time
arn:aws:states:eu-west-1:727432808710:execution:MortgageApprovalWorkflow:29809572-eae6-7355-769a-9eda8698d9ec	Nov 16, 2019 05:27:54.104 PM
▼ Input	▼ Output
{ "mortgageId": "5000004" }	{ "statusCode": 200, "messageId": "2e2003f8-8547-5e39-b730-55b2b69ecd4c" }

Figure AWS.AUTO.5

The Step Function output correctly shows that the last Task returned a 200 status completion code and also the ID of the SNS message that was send to the users.

However another coding error show that the new state is not correctly rendered in the email. Figure

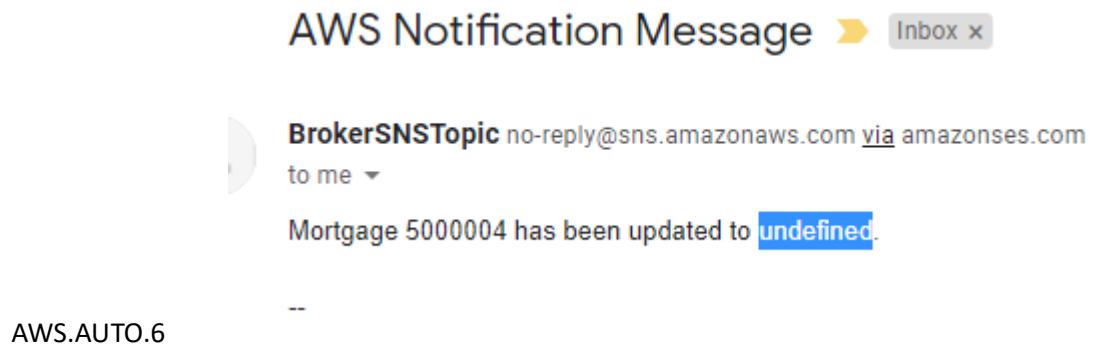


Figure AWS.AUTO.6

Once resolved we see the correct message in the email. Figure AWS.AUTO.7

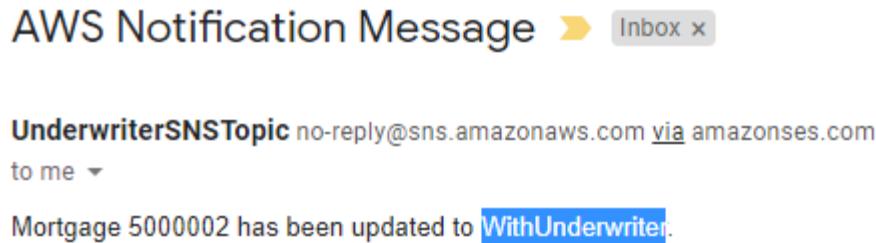


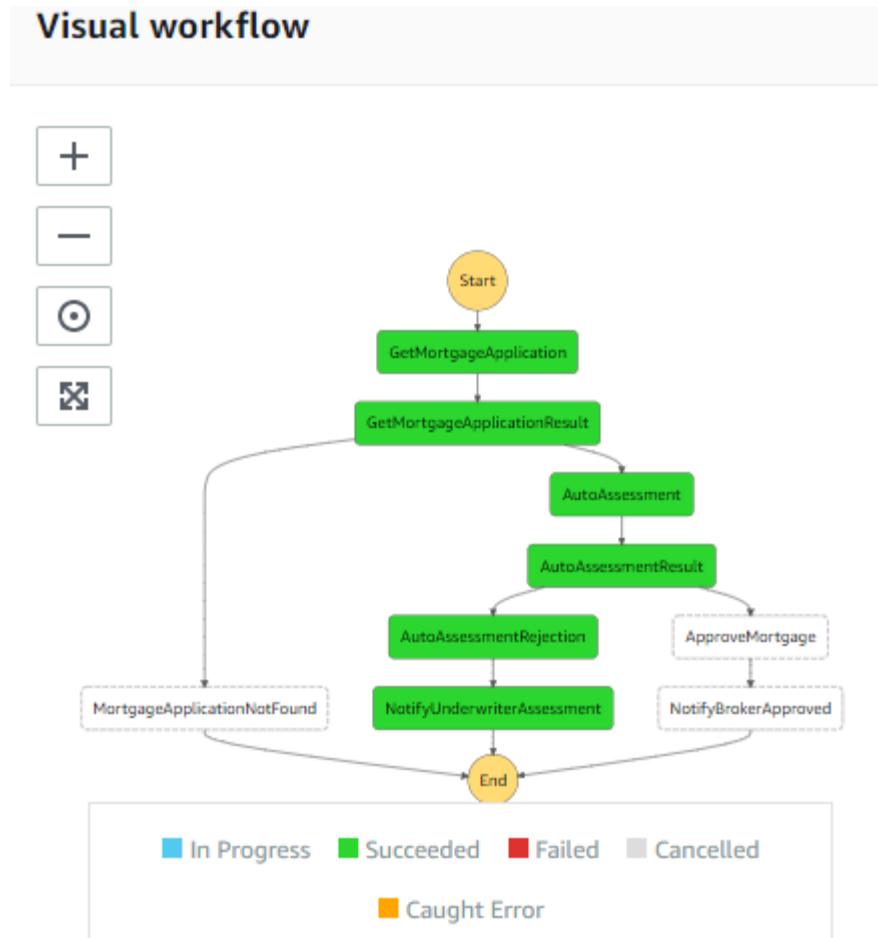
Figure AWS.AUTO.7

The mortgage application has also been correctly updated in DynamoDB. Figure AWS.AUTO.8

	mortgageld	customerId	employerName	loanAmount	mortgageStatus	salary
<input checked="" type="checkbox"/>	5000002	60002	Smurfit	300009	WithUnderwriter	50009
<input type="checkbox"/>	5000003	60003	Microsoft	200000	PreSubmission	100000
<input type="checkbox"/>	5000004	60003	AIB Ltd	999999	PreSubmission	100001

Figure AWS.AUTO.8

The Step Function execution shows the sequence of tasks.



Auto Assessment Success

Select a Mortgage Application that meets ALL Auto Assessment criteria. Figure AWS.AUTO.9

New execution

Start an execution using the latest definition of the state machine. [Learn more](#)

Enter an execution name - optional
Enter your execution id here
29809572-eae6-7355-769a-9eda8698d9ec

Input - optional
Enter input values for this execution in JSON format

```
1 {  
2   "mortgageId": "5000004"  
3 }
```

Open in a new browser tab Cancel **Start execution**

Figure AWS.AUTO.9

Successful email SNS Notification. Figure AWS.AUTO.10

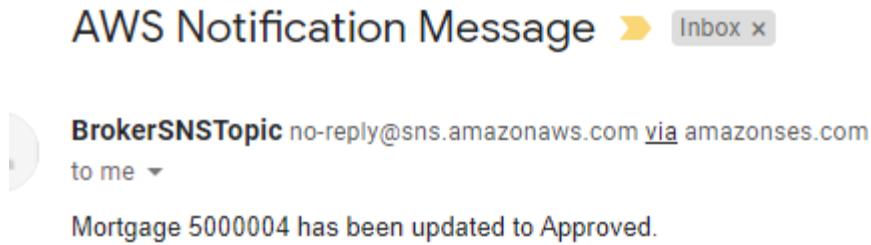


Figure AWS.AUTO.10

DynamoDB has also been updated. Figure AWS.AUTO.11

	mortgageId	customerId	employerName	loanAmount	mortgageStatus	salary	ter
	5000002	60002	Smurfit	300009	PreSubmission	50009	39
	5000003	60003	Microsoft	200000	PreSubmission	100000	20
<input checked="" type="checkbox"/>	5000004	60003	AIB Ltd	999999	Approved	100001	98

Figure AWS.AUTO.11

Again the Step Function execution diagram shows the sequence of events. Figure AWS.AUTO.12

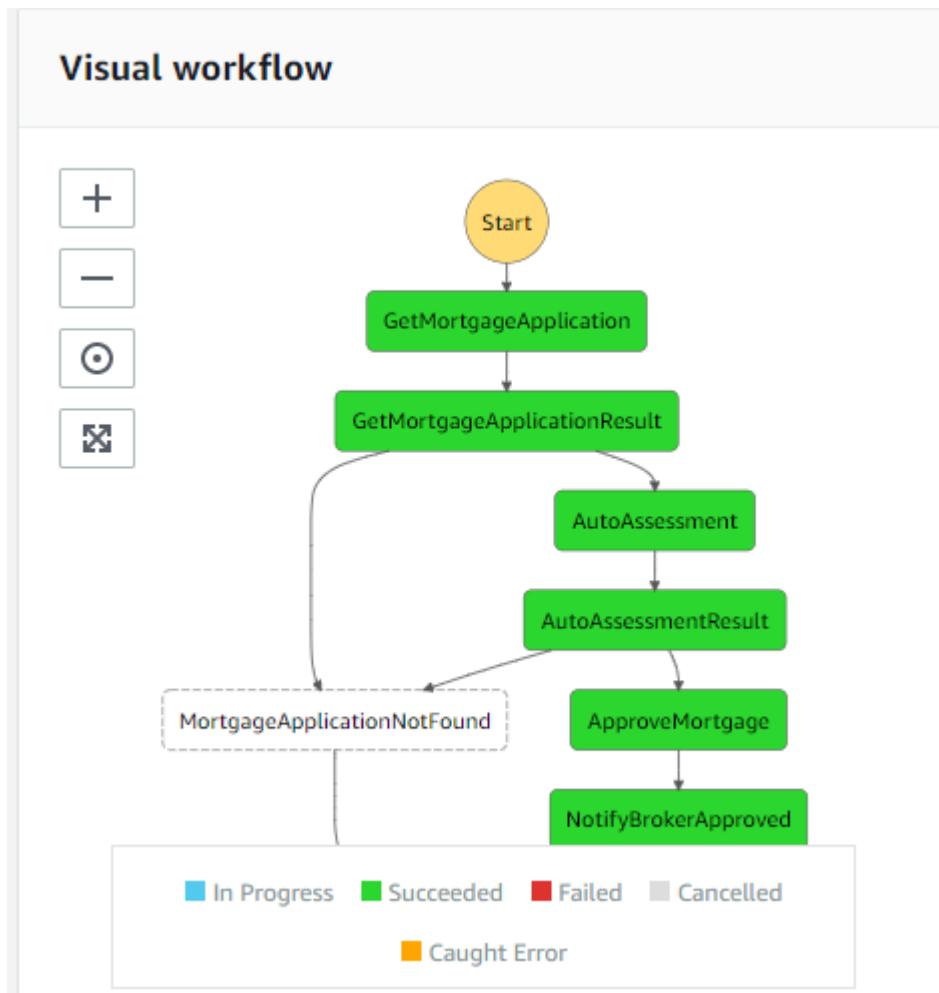


Figure AWS.AUTO.12

PROGRESS FOR SECOND WEEK OF ITERATION TWO.

ITERATION 2.2 TASKS

Building upon last weeks progress report, this document outlines the additional project tasks that have been completed this week for the final implementation phase 2 submission. The follow tasks were completed this week.

Project Phase	Service / Area	Implementation Task
Build / Development	AWS Step Functions	Significantly refactor the Step Function Workflow state machine. The original design of the workflow was done without a full understanding of the details and limitations of the State Machine Definition Language in terms of data state and mediation.
Build / Development	AWS Cloudformation	I have not had time to integrate the deployment of the Workflow state machine into the project Cloud formation template.
Build / Development	AWS Lamda	Completed the development of Lambda Functions to allow workflow to await human/underwriting decisions.
Build / Development	DynamoDB	Implement new DB schema to map mortgage application to Step Function execution id. This is required to allow users approve or reject applications without reference to the step function execution id.
Build / Development	Angular	Extends the Browser application to allow submission of mortgages for assessment. I have not had time to extend the functionality to allow for clarification requests.
Test	AWS Step Functions	Test State Transitions / Events / Human interaction

Table D.

Defining the data inputs and outputs of each State within the Step Function workflow requires working within the restrictions of the utility functions provided by the State Machine Definition Language [13]. Input and Output processing [14] must be done using the 3 processing methods : InputPath, ResultPath and OutputPath. These can be restrictive and documentation for complex scenarios is not easily available even within the Step Function Developer Guide [15]. Integration with existing Lambda functions (especially if they where written to used as HTTP endpoints) can require tricky data manipulation. For my Workflow definition I needed to maintain the Mortgage state data and pass it through every state, which does not happen by default as the Lambda output will overwrite the State data if the Lambda result is not saved to a distinct javascript element / node to avoid overwrites. Also removing stale or State specific data can require for example a Pass state that can build new data state for the next state.

PLAN FOR ITERATION 3.1

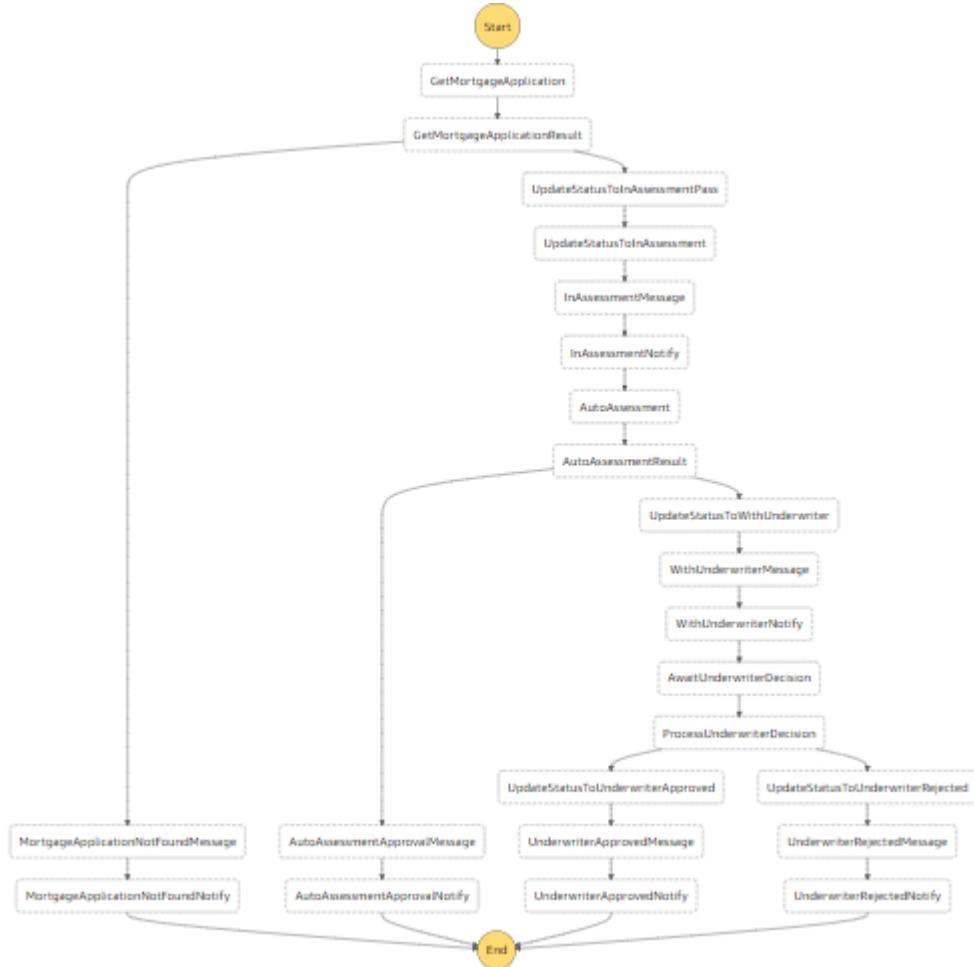
Project Phase	Service / Area	Implementation Task
Build / Development	AWS Step Functions	Extend the Step Function Workflow state machine to support Underwriting Clarification requests to the Broker to allow questions be submitted and answered regarding the mortgage application or applicant. The Broker must then be able to restart the workflow.
Build / Development	AWS Lambda	Add persistance to the Step Function tokens to allow users to restart the state machine execution from the Browser. Use either SQS or DynamoDB or both.
Build / Development	AWS S3 / Angular	Add Underwriter screens to the Front end app to allow the brokers to perform their tasks.
Build / Development	AWS S3 / Angular	Build new screens to allow underwriters to submit clarification questions and for the brokers to answer.
Build / Development	AWS Cloudformation	Integrate the deployment of the Workflow state machine into the project Cloud formation template.
Test	AWS Step Functions	Test State Transitions / Events / Human interaction

Table E.

ITERATION 2 SUBMISSION WORKFLOW STATE MACHINE

The workflow state machine currently exists as seen in Figure AWS.STEP.2.1

Figure AWS.STEP.2.1



ITERATION 2 SUBMISSION TECHNICAL ARCHITECTURE

The overall current technical architecture is outlined in Figure AWS.2.2

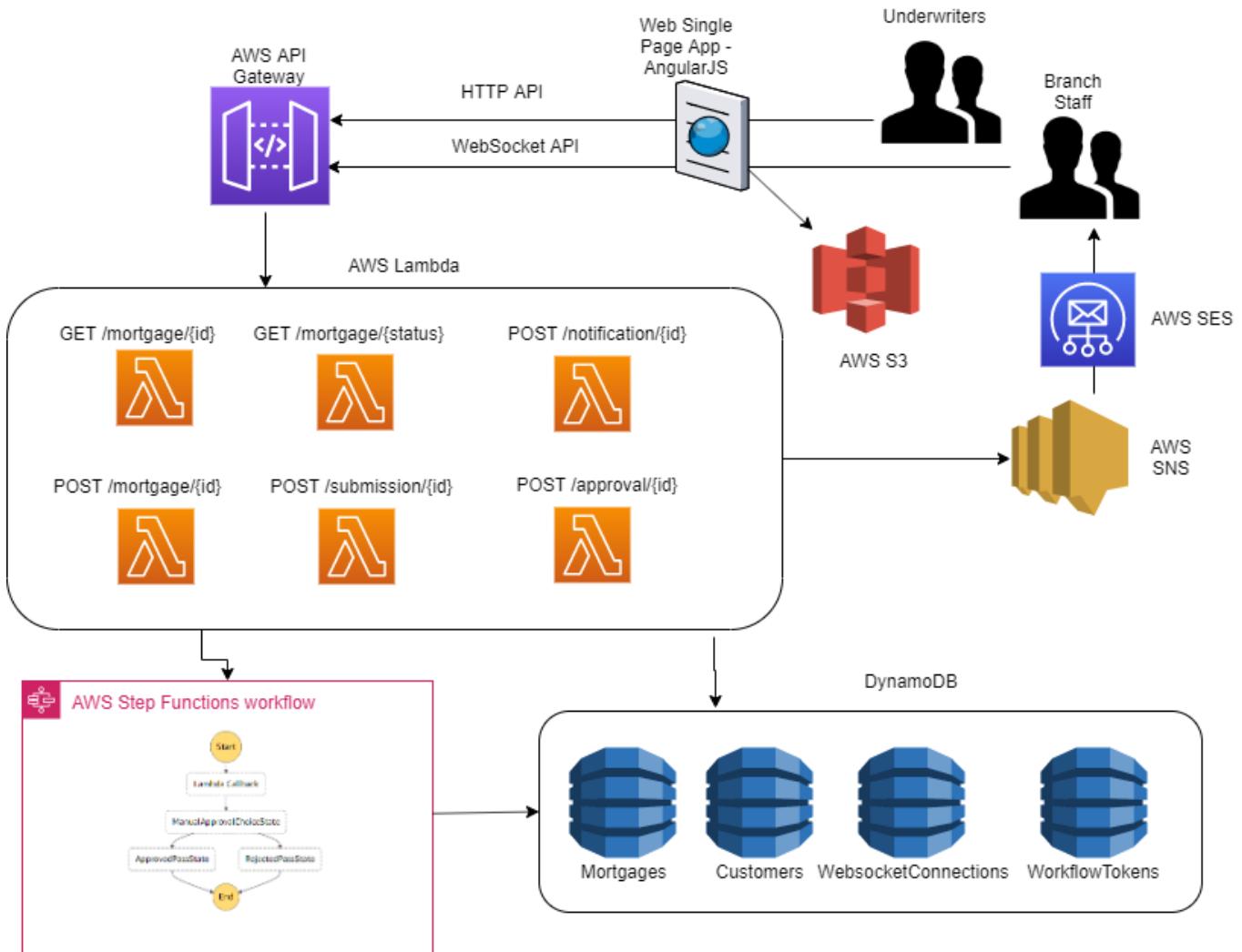


Figure AWS.2.2

ITERATION 2 LAMBDA FUNCTIONS

This table lists the new Lambda functions that I have developed this week in addition to those already developed (previously listed in Table AWS.LAMBDA.1). Table AWS.LAMBDA.2.1

Lambda Function	Purpose
CreateNotificationMessageFunction	This function is used by many states in the workflow to develop appropriate situational notification messages for brokers and underwriters to give them clarity on the current processing state of the mortgage application.
ReceiveUnderwriterDecisionEmailFunction	The Lambda function is exposed as a HTTP API via API Gateway to allow the Underwriter to submit their decision and restart the Workflow execution to process that decision.
SendNotificationMessageFunction	This function is used in combination with CreateNotificationMessageFunction to send the generated message to the appropriate SNS topic and the Websocket API for transmission to all connected browser apps.
SendUnderwriterDecisionEmailFunction	The Lambda function is called from the State Machine waitForTask State. It will generate the notification email to allow the underwriter to approve or reject the application. The email will contain the ReceiveUnderwriterDecisionEmailFunction HTTP API endpoint.

Table AWS.LAMBDA.2.1

The source code and deployment template for these functions are available at the GitHub repository listed in Table A or in the Appendix.

ITERATION 2 WORKFLOW HUMAN DECISIONS

This week I built out the functionality to allow the Step Function workflow to pause its execution to allow for Human decision making. Initially I followed to online documentation [16] to build a small test case. Figure AWS.STEP.2.1

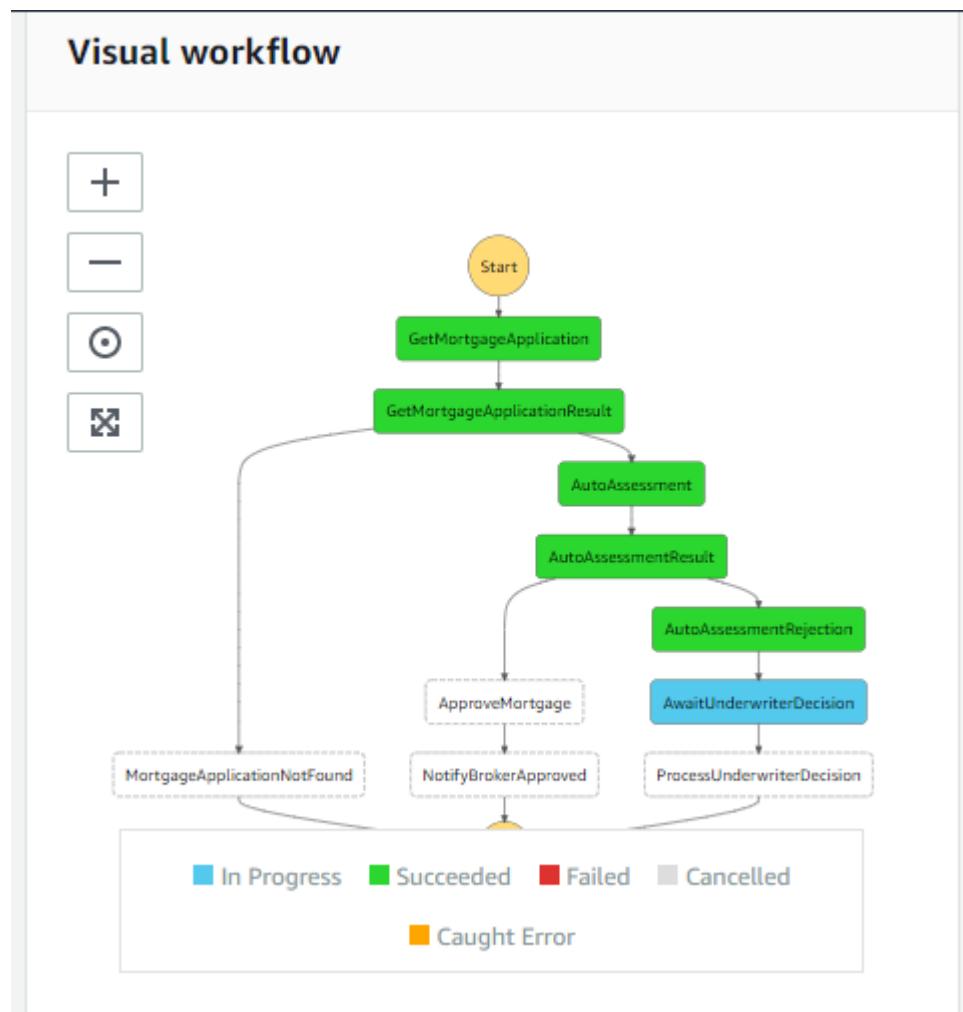


Figure AWS.STEP.2.1

The input to the execution is the Mortgage Id. Figure AWS.STEP.2.2

Execution details	
Execution Status	Started
 Succeeded	Nov 19, 2019 06:22:43.786 PM
Execution ARN	End Time
arn:aws:states:eu-west-1:727432808710:execution:HumanApprovalLambdaStateMachine-3FXyQ8OCHdyE:e6face65-b7b9-6f40-321c-1d7a14d949d5	Nov 19, 2019 06:23:43.275 PM
 Input	 Output
{ "Comment": "Testing the human approval tutorial." APPD" }	{ >Status": "Approved! Task approved by appduffin@gmail.com" }

Figure AWS.STEP.2.2

During execution we can see that the state will pause waiting for a Human decision. Figure AWS.STEP.2.3

▶ 23	TaskStarted	AwaitUnderwriterDecision	-	1264	Nov 20, 2019 06:24:50.409 PM
▶ 24	TaskSubmitted	AwaitUnderwriterDecision	-	2780	Nov 20, 2019 06:24:51.925 PM

Figure AWS.STEP.2.3

The email is delivered to the registered email address and contains embedded URLs that the underwriter can use to either Approve or reject the application. Figure AWS.STEP.2.4

Underwriting Decision required for Mortgage [].

UnderwriterSNSTopic no-reply@sns.amazonaws.com via amazonsns.com
to me ▾

6:24 PM (0 minutes ago)

This is an email requesting that the Underwriter perform a review of the Mortgage Application.

Please check the following information and click "Approve" link if you want to approve.

Execution Name -> 5ea7f199-a115-2cf0-2873-6aa80945a2c5

Approve <https://14c5jip7sg.execute-api.eu-west-1.amazonaws.com/Prod/workflow/execution?action=approve&ex=5ea7f199-a115-2cf0-2873-6aa80945a2c5&sm=MortgageApprovalWorkflow&taskToken=AAAAAKgAAAAIAAAAAAAAaWKVUD1wbXiWTdrI4bl62x%2BXq3MH9jV%2F0Uqy5GDKWDOSlr2ZgTbvcxmVnr2Q>
SijBHDGs3Hw67CgZPTi2Xsy9Up8a7p0Solxu1ZeRDx0HzLzRDzGkyM8D7YFFgip7OZvLdp9Rz%2BD4pazmkenosZJyipQnE9BPJxfJgh9BxINGRzwvUGKs
MGLh6qpr2wcqWsIEJyC%2BypanRaN2Qmti44E%2F1v4ko2ivkJXufXkquPxvxEWCzaGbfpuO8IU03TW17WG2RWAUxzlppCK0DVFXJMvaCrJX
aO49mQaxMXoeMcx4Qi47D0XpVTDLWrBCFcot%2FY9hhNWT9euFZRVzvMYQVZyBAssvMNtbJ4kOh%2FPqxWFuPGsBxblb4%2FsDjsOK6O97
s0DWB217qcQ80Xpl3uKXlxJFd05jhQ8q84sMp2Tyzkx3CZA3pURS2NkBvJw4%2BtHBkbdkytWfrTYTXZdwYjnHwKa7cgm9Y3a328fbFy9uJDdQPs0BrUzX6
rvSPnNOzLj1CITsKRX5Sc0tjAQ%2B3%2Fu%2Frkf56iOsx4hfsh%2FeV%2Fuy22%2Bd81oR%2FfQzn9UPowWPZZFdRck6Ok3MpuauR%
2B7O1goXCmlz6bQf9%2BI4u3vf5BZk9

Reject <https://14c5jip7sg.execute-api.eu-west-1.amazonaws.com/Prod/workflow/execution?action=reject&ex=5ea7f199-a115-2cf0-2873-6aa80945a2c5&sm=MortgageApprovalWorkflow&taskToken=AAAAAKgAAAAIAAAAAAAAaWKVUD1wbXiWTdrI4bl62x%2BXq3MH9jV%2F0Uqy5GDKWDOSlr2ZgTbvcxmVnr2Q>
SijBHDGs3Hw67CgZPTi2Xsy9Up8a7p0Solxu1ZeRDx0HzLzRDzGkyM8D7YFFgip7OZvLdp9Rz%2BD4pazmkenosZJyipQnE9BPJxfJgh9BxINGRzwvUGKs
MGLh6qpr2wcqWsIEJyC%2BypanRaN2Qmti44E%2F1v4ko2ivkJXufXkquPxvxEWCzaGbfpuO8IU03TW17WG2RWAUxzlppCK0DVFXJMvaCrJX
aO49mQaxMXoeMcx4Qi47D0XpVTDLWrBCFcot%2FY9hhNWT9euFZRVzvMYQVZyBAssvMNtbJ4kOh%2FPqxWFuPGsBxblb4%2FsDjsOK6O97
s0DWB217qcQ80Xpl3uKXlxJFd05jhQ8q84sMp2Tyzkx3CZA3pURS2NkBvJw4%2BtHBkbdkytWfrTYTXZdwYjnHwKa7cgm9Y3a328fbFy9uJDdQPs0BrUzX6

Figure AWS.STEP.2.4

APPENDICES

The following sections list some key / interesting source code / template snippets.

WebSocket Cloudformation Template

```
AWSTemplateFormatVersion: '2010-09-09'

Transform: AWS::Serverless-2016-10-31
Description: >
    Tallaght-Mortgages-App

Globals:
Function:
    Timeout: 3
Api:
    EndpointConfiguration: REGIONAL
    Cors:
        AllowMethods: "'*'"
        AllowHeaders: "'Content-Type'"
        AllowOrigin: "'*'"


Parameters:
LambdaRoleParameter:
    Type: String
    Default: "arn:aws:iam::727432808710:role/TUD.Mortgage.Lambda.Role"
    Description: Enter the ARN of an applicable IAM Role for the Lambda Functions
StepFunctionArnParameter:
    Type: String
    Default: "arn:aws:states:eu-west-
1:727432808710:stateMachine:MortgageApprovalWorkflow"
    Description: Enter the ARN of the Workflow Step Function State Machine
WebsocketConnectionTableName:
    Type: String
    Default: 'WEBSOCKET_CONNECTIONS_TABLE'
StepFunctionTokenTableName:
    Type: String
    Default: 'STEPFUNCTION_TOKEN_TABLE'
   



Resources:

# SNS TOPICS
UnderwriterSNSTopic:
    Type: AWS::SNS::Topic
    Properties:
```

```
  DisplayName: UnderwriterSNSTopic
  Subscription:
    - Endpoint: appduffin@gmail.com
    Protocol: email
  TopicName: UnderwriterSNSTopic
BrokerSNSTopic:
  Type: AWS::SNS::Topic
  Properties:
    DisplayName: BrokerSNSTopic
    Subscription:
      - Endpoint: appduffin@gmail.com
      Protocol: email
    TopicName: BrokerSNSTopic
```

```
# CUSTOMER APIs
GetCustomerFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: customers/
    Handler: get.get
    Runtime: nodejs10.x
  Role:
    Ref: LambdaRoleParameter
  Events:
    GetCustomerEvent:
      Type: Api
      Properties:
        Path: /customer/{id}
        Method: GET
CreateCustomerFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: customers/
    Handler: create.create
    Runtime: nodejs10.x
  Role:
    Ref: LambdaRoleParameter
  Events:
    CreateCustomerEvent:
      Type: Api
      Properties:
        Path: /customer/
        Method: PUT
```

```
UpdateCustomerFunction:  
  Type: AWS::Serverless::Function  
  Properties:  
    CodeUri: customers/  
    Handler: update.update  
    Runtime: nodejs10.x  
    Role: [REDACTED]  
    Ref: LambdaRoleParameter  
  Events:  
    UpdateCustomerEvent:  
      Type: Api  
      Properties:  
        Path: /customer/{id}  
        Method: POST
```

```
# MORTGAGE APIs  
GetMortgageFunction:  
  Type: AWS::Serverless::Function  
  Properties:  
    CodeUri: mortgages/  
    Handler: get.get  
    Runtime: nodejs10.x  
    Role: [REDACTED]  
    Ref: LambdaRoleParameter  
  Events:  
    GetMortgageEvent:  
      Type: Api  
      Properties:  
        Path: /mortgage/{id}  
        Method: GET  
CreateMortgageFunction:  
  Type: AWS::Serverless::Function  
  Properties:  
    CodeUri: mortgages/  
    Handler: create.create  
    Runtime: nodejs10.x  
    Role: [REDACTED]  
    Ref: LambdaRoleParameter  
  Events:  
    CreateMortgageEvent:  
      Type: Api  
      Properties:  
        Path: /mortgage/  
        Method: PUT  
UpdateMortgageFunction:
```

```
Type: AWS::Serverless::Function
Properties:
  CodeUri: mortgages/
  Handler: update.update
  Runtime: nodejs10.x
  Role: [REDACTED]
    Ref: LambdaRoleParameter
Events:
  CreateMortgageEvent:
    Type: Api
    Properties:
      Path: /mortgage/{id}
      Method: POST
DeleteMortgageFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: mortgages/
    Handler: delete.delete
    Runtime: nodejs10.x
    Role: [REDACTED]
    Ref: LambdaRoleParameter
  Events:
    DeleteMortgageEvent:
      Type: Api
      Properties:
        Path: /mortgage/{id}
        Method: DELETE
ListByStatusMortgageFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: mortgages/
    Handler: list.list
    Runtime: nodejs10.x
    Role: [REDACTED]
    Ref: LambdaRoleParameter
  Events:
    ListByStatusMortgageEvent:
      Type: Api
      Properties:
        Path: /mortgage/
        Method: GET
```

```
# STEP FUNCTION HUMAN TOKEN DYNAMO DB TABLE
StepFunctionTokenTable:
  Type: AWS::DynamoDB::Table
```

```
Properties:  
  TableName: !Ref StepFunctionTokenTableName  
  AttributeDefinitions:  
    - AttributeName: "mortgageId"  
      AttributeType: "S"  
#      - AttributeName: "stepFunctionToken"  
#      AttributeType: "S"  
  KeySchema:  
    - AttributeName: "mortgageId"  
      KeyType: "HASH"  
  BillingMode: PAY_PER_REQUEST
```

```
# WORKFLOW APIs  
NotifyUsersSNSFunction:  
  Type: AWS::Serverless::Function  
  Properties:  
    Handler: stepfunctions/notifications/snsNotifications.handler  
    Runtime: nodejs10.x  
    Role: !Ref LambdaRoleParameter  
  Environment:  
    Variables:  
      UNDERWRITER_TOPIC_ARN: !Ref UnderwriterSNSTopic  
      BROKER_TOPIC_ARN: !Ref BrokerSNSTopic  
  DependsOn:  
    - UnderwriterSNSTopic  
    - BrokerSNSTopic
```

```
CreateNotificationMessageFunction:  
  Type: AWS::Serverless::Function  
  Properties:  
    Handler: stepfunctions/notifications/notificationMessages.handler  
    Runtime: nodejs10.x  
    Role: !Ref LambdaRoleParameter  
  Environment:  
    Variables:  
      UNDERWRITER_TOPIC_ARN: !Ref UnderwriterSNSTopic  
      BROKER_TOPIC_ARN: !Ref BrokerSNSTopic  
  DependsOn:  
    - UnderwriterSNSTopic  
    - BrokerSNSTopic
```

```
SendNotificationMessageFunction:  
  Type: AWS::Serverless::Function
```

```
Properties:  
  Handler: stepfunctions/notifications/sendNotifications.handler  
  Runtime: nodejs10.x  
  # Role:  
  # Ref: LambdaRoleParameter  
Environment:  
  Variables:  
    TABLE_NAME: !Ref WebsocketConnectionTableName  
    WEBSOCKET_ENDPOINT: !Join [ '', [ !Ref MortgageWebSocket, '.execute-  
api.', !Ref 'AWS::Region', '.amazonaws.com/Prod' ] ]  
Policies:  
  - DynamoDBCrudPolicy:  
    TableName: !Ref WebsocketConnectionTableName  
  - Statement:  
    - Effect: Allow  
      Action:  
        - 'sns:*'  
      Resource:  
        - '*'  
  - Statement:  
    - Effect: Allow  
      Action:  
        - 'execute-api:ManageConnections'  
      Resource:  
        - !Sub 'arn:aws:execute-  
api:${AWS::Region}:${AWS::AccountId}:${MortgageWebSocket}/*'  
DependsOn:  
  - UnderwriterSNSTopic  
  - BrokerSNSTopic  
  - WebsocketConnectionsTable  
  - MortgageWebSocket
```

```
WorkflowGetCustomerFunction:  
  Type: AWS::Serverless::Function  
  Properties:  
    Handler: stepfunctions/customers/findByPK.handler  
    Runtime: nodejs10.x  
    Role:  
      Ref: LambdaRoleParameter  
WorkflowGetMortgageFunction:  
  Type: AWS::Serverless::Function  
  Properties:  
    Handler: stepfunctions/mortgages/findByPK.handler  
    Runtime: nodejs10.x  
    Role:
```

```
Ref: LambdaRoleParameter
WorkflowUpdateMortgageFunction:
  Type: AWS::Serverless::Function
  Properties:
    Handler: stepfunctions/mortgages/updateMortgage.handler
    Runtime: nodejs10.x
    Role: [REDACTED]
      Ref: LambdaRoleParameter
StartWorkflowFunction:
  Type: AWS::Serverless::Function
  Properties:
    Handler: stepfunctions/startWorkflow.handler
    Runtime: nodejs10.x
    Role: [REDACTED]
      Ref: LambdaRoleParameter
Environment:
  Variables:
    MORTGAGE_WORKFLOW_ARN: !Ref StepFunctionArnParameter
Events:
  StartWorkflowEvent:
    Type: Api
    Properties:
      Path: /workflow/execution
      Method: PUT
AutoAssessmentFunction:
  Type: AWS::Serverless::Function
  Properties:
    Handler: workflow/autoassessment.handler
    Runtime: nodejs10.x
    Role: [REDACTED]
      Ref: LambdaRoleParameter
ChangeMortgageStatusFunction:
  Type: AWS::Serverless::Function
  Properties:
    Handler: stepfunctions/mortgages/changeMortgageStatus.handler
    Runtime: nodejs10.x
    Role: [REDACTED]
      Ref: LambdaRoleParameter
MortgageStatusSNSFunction:
  Type: AWS::Serverless::Function
  Properties:
    Handler: stepfunctions/notifications/mortgageStatusSend.handler
    Runtime: nodejs10.x
    Role: [REDACTED]
      Ref: LambdaRoleParameter
```

```
SendUnderwriterDecisionEmailFunction:  
Type: AWS::Serverless::Function  
Properties:  
  Handler: workflow/sendUnderwriterApprovalEmail.handler  
  Runtime: nodejs10.x  
  Role: !Ref LambdaRoleParameter  
  Environment:  
  Variables:  
    UNDERWRITER_DECISION_API: !Sub "https://${ServerlessRestApi}.execute-  
api.${AWS::Region}.amazonaws.com/Prod/workflow/execution"  
#      UNDERWRITER_DECISION_API: !Join [ 'https://', [ !Ref ServerlessRestA  
pi, '.execute-  
api.', !Ref 'AWS::Region', '.amazonaws.com/Prod/workflow/execution' ] ]  
    UNDERWRITER_TOPIC_ARN: !Ref UnderwriterSNSTopic  
  
ReceiveUnderwriterDecisionEmailFunction:  
Type: AWS::Serverless::Function  
Properties:  
  Handler: workflow/receiveUnderwriterDecision.handler  
  Runtime: nodejs10.x  
  Role: !Ref LambdaRoleParameter  
Events:  
  UnderwriterDecisionEvent:  
    Type: Api  
    Properties:  
      Path: /workflow/execution  
      Method: GET  
  
# WEBSOCKET APIs  
MortgageWebSocket:  
Type: AWS::ApiGatewayV2::Api  
Properties:  
  Name: MortgageWebSocket  
  ProtocolType: WEBSOCKET  
  RouteSelectionExpression: "$request.body.message"  
ConnectRoute:  
Type: AWS::ApiGatewayV2::Route  
Properties:  
  ApiId: !Ref MortgageWebSocket  
  RouteKey: $connect  
  AuthorizationType: NONE  
  OperationName: ConnectRoute  
  Target: !Join
```

```
- '/'
- - 'integrations'
- !Ref MortgateConnectionIntegration
MortgateConnectionIntegration:
Type: AWS::ApiGatewayV2::Integration
Properties:
  ApiId: !Ref MortgageWebSocket
  Description: Connect Integration
  IntegrationType: AWS_PROXY
  IntegrationUri:
  Fn::Sub:
    arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${WebSocketConnectFunction.Arn}/invocations
  DisconnectRoute:
    Type: AWS::ApiGatewayV2::Route
    Properties:
      ApiId: !Ref MortgageWebSocket
      RouteKey: $disconnect
      AuthorizationType: NONE
      OperationName: DisconnectRoute
      Target: !Join
        - '/'
        - - 'integrations'
        - !Ref MortgateDisconnectInteg
MortgateDisconnectInteg:
Type: AWS::ApiGatewayV2::Integration
Properties:
  ApiId: !Ref MortgageWebSocket
  Description: Disconnect Integration
  IntegrationType: AWS_PROXY
  IntegrationUri:
  Fn::Sub:
    arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-
31/functions/${WebSocketDisconnectFunction.Arn}/invocations
  SendRoute:
    Type: AWS::ApiGatewayV2::Route
    Properties:
      ApiId: !Ref MortgageWebSocket
      RouteKey: sendmessage
      AuthorizationType: NONE
      OperationName: SendRoute
      Target: !Join
        - '/'
        - - 'integrations'
        - !Ref SendInteg
```

```
SendInteg:  
  Type: AWS::ApiGatewayV2::Integration  
  Properties:  
    ApiId: !Ref MortgageWebSocket  
    Description: Send Integration  
    IntegrationType: AWS_PROXY  
    IntegrationUri: !Ref MortgageWebSocket  
    Fn::Sub:  
      arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-  
31/functions/${WebSocketSendFunction.Arn}/invocations  
  Deployment:  
    Type: AWS::ApiGatewayV2::Deployment  
    DependsOn:  
      - ConnectRoute  
      - SendRoute  
      - DisconnectRoute  
    Properties:  
      ApiId: !Ref MortgageWebSocket  
  Stage:  
    Type: AWS::ApiGatewayV2::Stage  
    Properties:  
      StageName: Prod  
      Description: Prod Stage  
      DeploymentId: !Ref Deployment  
      ApiId: !Ref MortgageWebSocket
```

```
# WEBSOCKET DYNAMO DB TABLE  
WebsocketConnectionsTable:  
  Type: AWS::DynamoDB::Table  
  Properties:  
    TableName: !Ref WebsocketConnectionTableName  
    AttributeDefinitions:  
      - AttributeName: "connectionId"  
        AttributeType: "S"  
    KeySchema:  
      - AttributeName: "connectionId"  
        KeyType: "HASH"  
    BillingMode: PAY_PER_REQUEST
```

```
# WEBSOCKET LAMBDA API  
WebSocketConnectFunction:  
  Type: AWS::Serverless::Function  
  Properties:  
    CodeUri: websockets/onconnect/  
    Handler: app.handler
```

```
MemorySize: 256
Runtime: nodejs10.x
Environment:
Variables:
  TABLE_NAME: !Ref WebsocketConnectionTableName
Policies:
  - DynamoDBCrudPolicy:
    TableName: !Ref WebsocketConnectionTableName
OnConnectPermission:
  Type: AWS::Lambda::Permission
  DependsOn:
    - MortgageWebSocket
    - WebSocketConnectFunction
Properties:
  Action: lambda:InvokeFunction
  FunctionName: !Ref WebSocketConnectFunction
  Principal: apigateway.amazonaws.com
WebSocketDisconnectFunction:
  Type: AWS::Serverless::Function
Properties:
  CodeUri: websockets/ondisconnect/
  Handler: app.handler
  MemorySize: 256
  Runtime: nodejs10.x
  Environment:
  Variables:
    TABLE_NAME: !Ref WebsocketConnectionTableName
Policies:
  - DynamoDBCrudPolicy:
    TableName: !Ref WebsocketConnectionTableName
OnDisconnectPermission:
  Type: AWS::Lambda::Permission
  DependsOn:
    - MortgageWebSocket
    - WebSocketDisconnectFunction
Properties:
  Action: lambda:InvokeFunction
  FunctionName: !Ref WebSocketDisconnectFunction
  Principal: apigateway.amazonaws.com
WebSocketSendFunction:
  Type: AWS::Serverless::Function
Properties:
  CodeUri: websockets/sendmessage/
  Handler: app.handler
  MemorySize: 256
```

```
Runtime: nodejs10.x
Environment:
Variables:
  TABLE_NAME: !Ref WebsocketConnectionTableName
  WEBSOCKET_ENDPOINT: !Join [ '', [ !Ref MortgageWebSocket, '.execute-
api.', !Ref 'AWS::Region', '.amazonaws.com/Prod' ] ]
Policies:
  - DynamoDBCrudPolicy:
    TableName: !Ref WebsocketConnectionTableName
  - Statement:
    - Effect: Allow
    Action:
      - 'execute-api:ManageConnections'
    Resource:
      - !Sub 'arn:aws:execute-
api:${AWS::Region}:${AWS::AccountId}: ${MortgageWebSocket}/*'
WebSocketSendPermission:
  Type: AWS::Lambda::Permission
  DependsOn:
    - MortgageWebSocket
    - WebSocketSendFunction
Properties:
  Action: lambda:InvokeFunction
  FunctionName: !Ref WebSocketSendFunction
  Principal: apigateway.amazonaws.com
```

```
Outputs:
  MortgageApi:
    Description: "API Gateway endpoint URL for Prod stage for GetMortgageFunctionR
ole"
    Value: !Sub "https://${ServerlessRestApi}.execute-
api.${AWS::Region}.amazonaws.com/Prod/mortgage/"
  GetCustomerFunction:
    Description: "GetCustomerFunctionARN"
    Value: !GetAtt GetCustomerFunction.Arn
  CreateCustomerFunction:
    Description: "CreateCustomerFunctionARN"
    Value: !GetAtt CreateCustomerFunction.Arn
  UpdateCustomerFunction:
    Description: "UpdateCustomerFunctionARN"
    Value: !GetAtt UpdateCustomerFunction.Arn
```

```
WebSocketConnectFunctionArn:
  Description: "Websocket OnConnect function ARN"
  Value: !GetAtt WebSocketConnectFunction.Arn
```

```
WebSocketDisconnectFunctionArn:  
  Description: "Websocket OnDisconnect function ARN"  
  Value: !GetAtt WebSocketDisconnectFunction.Arn
```

```
WebSocketSendFunctionArn:  
  Description: "Websocket SendMessage function ARN"  
  Value: !GetAtt WebSocketSendFunction.Arn
```

```
WebSocketURI:  
  Description: "Websocket WSS Protocol URI"  
  Value: !Join [ '', [ 'wss://', !Ref MortgageWebSocket, '.execute-  
api.', !Ref 'AWS::Region', '.amazonaws.com/', !Ref 'Stage' ] ]
```

REFERENCE LIST AND BIBLIOGRAPHY

- [1] AWS Free Tier. <https://aws.amazon.com/free/>
- [2] AWS CLI. <https://docs.aws.amazon.com/cli/latest/userguide/install-windows.html>
- [3] AWS CLI SAM. <https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-install-windows.html>
- [4] Node JS. <https://nodejs.org/en/download/>
- [5] Configure AWS CLI. <https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html>
- [6] Visual Studio Code. <https://code.visualstudio.com/download>
- [7] Configure AWS on Visual Studio Code. <https://docs.aws.amazon.com/toolkit-for-vscode/latest/userguide/setup-toolkit.html>
- [8] Install Docker Toolbox. https://docs.docker.com/toolbox/toolbox_install_windows/
- [9] DynamoDB Cloudformation Template.
<https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.dynamo/dynamo-create.yml>
- [10] Lambda Cloudformation Template.
<https://github.com/GeneralYeager/tallaght.mortgages/blob/master/tud.mortgages/Tallaght-Mortgages-App/template.yaml>
- [11] Step Function State Machine Template.
<https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.workflow/StateMachineV3.txt>
- [12] <https://aws.amazon.com/blogs/compute/announcing-websocket-apis-in-amazon-api-gateway/>
- [13] <https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html>
- [14] <https://docs.aws.amazon.com/step-functions/latest/dg/concepts-input-output-filtering.html>
- [15] <https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html>
- [16] <https://docs.aws.amazon.com/step-functions/latest/dg/connect-to-resource.html#connect-wait-token>

01/12/2019



ALAN DUFFIN

X00159409

CERTIFICATE, CLOUD SOLUTIONS ARCHITECTURE 2019, PROJECT

IMPLEMENTATION PHASE 3.1

TABLE OF CONTENTS

PROJECT PLAN FOR ITERATION 3	3
PROJECT PROGRESS	4
WORKFLOW / FRONTEND INTEGRATION AND TOKEN PERSISTENCE	5
FRONT END ENHANCEMENTS.....	8
STEP FUNCTIONS AND CLOUDFORMATION.....	10
REFERENCE LIST AND BIBLIOGRAPHY	11

PROJECT PLAN FOR ITERATION 3

The table below (Table A) lists the tasks that I will attempt to complete before the final Iteration 3 deadline.

Project Phase	Service / Area	Implementation Task
Build / Development	AWS Step Functions	Extend the Step Function Workflow state machine to support Underwriting Clarification requests to the Broker to allow questions be submitted and answered regarding the mortgage application or applicant. The Broker must then be able to restart the workflow.
Build / Development	AWS Lambda	Add persistence to the Step Function tokens to allow users to restart the state machine execution from the Browser. Use either SQS or DynamoDB or both.
Build / Development	AWS S3 / Angular	Add Underwriter screens to the Front end app to allow the brokers to perform their tasks.
Build / Development	AWS S3 / Angular	Build new screens to allow underwriters to submit clarification questions and for the brokers to answer.
Build / Development	AWS Cloudformation	Integrate the deployment of the Workflow state machine into the project Cloud formation template.
Test	AWS Step Functions	Test State Transitions / Events / Human interaction

Table A.

PROJECT PROGRESS

This week I completed the following tasks:

- Add persistance to the Step Function tokens to allow users to restart the state machine execution from the Browser. Use either SQS or DynamoDB or both.
- Add Underwriter screens to the Front end app to allow the brokers to perform their tasks.
- Build new screens to allow underwriters to submit clarification questions and for the brokers to answer.
- Test State Transitions / Events / Human interaction

Please review the video (MP4) at the following URL:

<https://s3-eu-west-1.amazonaws.com/appd.cloud.project/Cloud-Project-Implementation-Phase3.1-2019-12-01+11-44-04.mp4>

WORKFLOW / FRONTEND INTEGRATION AND TOKEN PERSISTENCE

In order to allow Underwriters approve / reject / clarify mortgage applications from the Browser based front end there must be a persistence mechanism to map the mortgage application id to the Step Function Execution restart token. So the first task is to build a DynamoDB table. This is done with the SAM CloudFormation template. Figure 3.1.1

```
# STEP FUNCTION HUMAN TOKEN DYNAMO DB TABLE

StepFunctionTokenTable:
  Type: AWS::DynamoDB::Table
  Properties:
    TableName: !Ref StepFunctionTokenTableName
    AttributeDefinitions:
      - AttributeName: "mortgageId"
        AttributeType: "S"
    KeySchema:
      - AttributeName: "mortgageId"
        KeyType: "HASH"
    BillingMode: PAY_PER_REQUEST
```

Figure 3.1.1

We then need to modify the Underwriter Decision process to store the token that is issued by the Step Function *waitForTaskToken* function. Figure 3.1.2.

```
"AwaitUnderwriterDecision": {
  "Type": "Task",
  "Resource": "arn:aws:states:::lambda:invoke.waitForTaskToken",
  "Parameters": {
    "FunctionName": "${SendUnderwriterDecisionEmailArn}",
    "Payload": {
      "mortgage.$": "$.mortgage",
      "ExecutionContext.$": "$$"
    }
  },
  "TimeoutSeconds": 180,
  "ResultPath": "$.lambdaresult",
  "Next": "ProcessUnderwriterDecision"
},
```

Figure 3.1.2

We must modify the Lambda functions that are involved in this process.

startUnderwriterDecisionProcess.js

```
var dbTokenParam = {
  TableName: STEP_FUNCTION_TABLE_NAME,
  Item: [
    'mortgageId': mortgage.mortgageId,
    'stepFunctionToken': taskToken
  ]
};

await dynamoDB.put(dbTokenParam).promise();
```

underwriterDecisionPOST.js

```
var dbTokenParam = {
  TableName: STEP_FUNCTION_TABLE_NAME,
  Key: {
    'mortgageId': decision.mortgageId
  }
};

const tokenItem = await dynamoDB.get(dbTokenParam).promise();
await dynamoDB.delete(dbTokenParam).promise();
```

underwriterDecisionGET.js

```
const action = event.queryStringParameters.action;
const taskToken = event.queryStringParameters.taskToken;
const mortgageId = event.queryStringParameters.mortgageId;

var dbTokenParam = {
  TableName: STEP_FUNCTION_TABLE_NAME,
  Key: [
    'mortgageId': mortgageId
  ]
};

const response = stepTokenUtil.restartWorkflow(mortgageId, taskToken, action);
await dynamoDB.delete(dbTokenParam).promise();
```

It was necessary to refactor the Step Function restart process because there are now two pathways for the Underwriters to submit a decision. When the Underwriter interacts with the State Machine execution via email the decision is submitted by embedded URLs with perform HTTP GET operations on API Gateway endpoints. Alternatively the Underwriter can submit the decision using the Browser based Angular frontend. In that scenario the decision is submitted via a HTTP POST operation.

In order to facilitate this two new Lambda Functions were created that act as a thin wrapper layer to extract the required data from the HTTP operations either from the HTTP POST body or the HTTP GET path variables. Both new functions then invoke common utility code to restart the Step Function workflow. This is found in the `/workflow/utils.restartWorkflow.js` source code file.

FRONT END ENHANCEMENTS

I have enhanced the front end Angular code to allow underwriters to view Mortgage Applications that require their attention. Also they can inspect the mortgage application information and approve or reject applications from within the browser web app.

Tallaght Mortgages Underwriters Page

Home	Reload Mortgages						
mortgageId	customerId	employerName	loanAmount	mortgageStatus	salary	term	yearsInEmployment

Figure 3.1.3

Underwriting Decision required for Mortgage [5000002].

Tallaght Mortgages Underwriters Page

Home	Reload Mortgages						
mortgageId	customerId	employerName	loanAmount	mortgageStatus	salary	term	yearsInEmployment
5000002	60002	Smurfit	300009	WithUnderwriter	50009	39	29

Figure 3.1.4

Underwriting Decision required for Mortgage [5000002].

Tallaght Mortgages Underwriters Page

[Home](#) | [Reload Mortgages](#)

mortagelId	customerId	employerName	loanAmount	mortgageStatus	salary	term	yearsInEmployment
5000002	60002	Smurfit	300009	WithUnderwriter	50009	39	29
employerName	Years in Employment				Salary		
Smurfit	29				50009		
Loan Amount	Term (Years)						
300009	39						

[Approve](#) | [Decline](#)

Figure 3.1.5

STEP FUNCTIONS AND CLOUDFORMATION

Integrate the deployment of the Workflow state machine into the project Cloud formation template.

Also this week I have hoped to integrate the Step Function State Machine definition into the CloudFormation stack template. However after spending 2 days investigating, numerous attempted deployments and restructuring of the template I have not been able to get this working.

To define a State Machine definition in the CloudFormation YAML template it is necessary to define a JSON definition string which is counter intuitive – to specify JSON in a YAML file.

Resources:

```
MortgagesStateMachine:  
  Type: AWS::StepFunctions::StateMachine  
  Properties:  
    StateMachineName: Tallaght-Workflow  
    DefinitionString:  
      !Sub  
        - |-  
        {  
          "StartAt": "GetMortgageApplication",  
          "States": {  
            "GetMortgageApplication": {  
              "Type": "Task",  
              "Resource": "${GetMortgageFunctionArn}",  
              "ResultPath": "$.lambdaresult",  
              "Next": "GetMortgageApplicationResult"  
            },  
          }  
        }  
    }
```

The advantages of having the Step Function definition in the CloudFormation template are that we can reference the ARNs of the Lambda Functions during stack deployment. Lambda functions that need to refer to the Step Function ARN could also capture the up to date ARN in Environment variables.

However after a number of hours correctly formatting the State Machine definition in the CloudFormation template, during the deployment I received Circular Dependency errors due to the dependancies between Lambda Functions, API Gateway and the Step Function definition. This points to a problem I have introduced somewhere. I believe the cause of this problem is the auto generation of the API Gateway Resouce operations within CloudFormation. By using the 'Event' definition on the Lambda functions to create HTTP endpoints, CloudFormation is automatically generating IAM policies/roles which may be required in some part of the definition before they are actually being created. The fix would be to explicitly define the HTTP Resources in an API Gateway Resource. However then I would need to add the other Resources in one at a time to find any conflicts. This will just take too long at the moment. The other solution would be to break the Cloudformation template into smaller logically defined areas, e.g. Workflow, Lambda, Database, etc and chain the Outputs and Inputs.

However after many attempts to find a solution (that does not include building the template again from the ground up) I have to acknowledge that I do not have sufficient time. I must focus on extending the State Machine next week to include the ability to request clarifications about the mortgage applications.

```
cmd Select Command Prompt - aws cloudformation deploy --template-file output.template.yaml --stack-name Mortgages-Stack --capabilities CAPABILITY_IAM
Successfully created/updated stack - Mortgages-Stack
C:\Users\user\aws\sam\git\tallaght.mortgages\tud.mortgages\tallaght-Mortgages-App>sam package --s3-bucket ad.aws.s3.codedeploy --output-template-file output.template.yaml
Uploading to 2d00caf1a4c6ab0938ab78e309e65ee2 55831 / 55831.0 <100.00>
Successfully packaged artifacts and wrote output template to file output.template.yaml.
Execute the following command to deploy the packaged template
aws cloudformation deploy --template-file C:\Users\user\aws\sam\git\tallaght.mortgages\tud.mortgages\tallaght-Mortgages-App\output.template.yaml --stack-name <YOUR STACK NAME>

C:\Users\user\aws\sam\git\tallaght.mortgages\tud.mortgages\tallaght-Mortgages-App>aws cloudformation deploy --template-file output.template.yaml --stack-name Mortgages-Stack --capabilities CAPABILITY_IAM
Waiting for changeset to be created..

Failed to create the changeset: Waiter ChangeSetCreateComplete failed: Waiter encountered a terminal failure state Status: FAILED, Reason: Circular dependency between resources: [GetCustomerFunctionGetCustomerEventPermissionProd, ListByStatusMortgageFunctionListByStatusMortgageEventPermissionProd, ReceiveUnderwriterDecisionPOSTFunctionGetMortgageEventPermissionUnderwriterDecisionPOSTEventPermissionProd, TallaghtStateMachine, StartWorkflowFunctionStartWorkflowEventPermissionProd, ReceiveUnderwriterDecisionGETFunctionUnderwriterDecisionGETEventPermissionProd, ServerlessRestApiProdStage, CreateCustomerFunctionCreateCustomerEventPermissionProd, GetMortgageFunctionGetMortgageEventPermissionProd, ServerlessRestApi, UpdateMortgageFunctionCreateMortgageEventPermissionProd, UpdateCustomerFunctionUpdateCustomerEventPermissionProd, ServerlessRestApiDeployed5a51308f, CreateMortgageFunctionCreateMortgageEventPermissionProd, DeleteMortgageFunctionDeleteMortgageEventPermissionProd, SendUnderwriterDecisionEmailFunction]
C:\Users\user\aws\sam\git\tallaght.mortgages\tud.mortgages\tallaght-Mortgages-App>sam package --s3-bucket ad.aws.s3.codedeploy --output-template-file output.template.yaml
Uploading to 3eb2fd3e5bd22dd2fe4a25eb0c917f540 55851 / 55851.0 <100.00>
Successfully packaged artifacts and wrote output template to file output.template.yaml.
Execute the following command to deploy the packaged template
aws cloudformation deploy --template-file C:\Users\user\aws\sam\git\tallaght.mortgages\tud.mortgages\tallaght-Mortgages-App\output.template.yaml --stack-name <YOUR STACK NAME>

C:\Users\user\aws\sam\git\tallaght.mortgages\tud.mortgages\tallaght-Mortgages-App>aws cloudformation deploy --template-file output.template.yaml --stack-name Mortgages-Stack --capabilities CAPABILITY_IAM
Waiting for changeset to be created..

Failed to create the changeset: Waiter ChangeSetCreateComplete failed: Waiter encountered a terminal failure state Status: FAILED, Reason: Circular dependency between resources: [GetCustomerFunctionGetCustomerEventPermissionProd, ListByStatusMortgageFunctionListByStatusMortgageEventPermissionProd, ReceiveUnderwriterDecisionPOSTFunctionGetMortgageEventPermissionUnderwriterDecisionPOSTEventPermissionProd, TallaghtStateMachine, StartWorkflowFunctionStartWorkflowEventPermissionProd, ReceiveUnderwriterDecisionGETFunctionUnderwriterDecisionGETEventPermissionProd, ServerlessRestApiProdStage, CreateCustomerFunctionCreateCustomerEventPermissionProd, GetMortgageFunctionGetMortgageEventPermissionProd, ServerlessRestApi, UpdateMortgageFunctionCreateMortgageEventPermissionProd, UpdateCustomerFunctionUpdateCustomerEventPermissionProd, ServerlessRestApiDeployed5a51308f, CreateMortgageFunctionCreateMortgageEventPermissionProd, DeleteMortgageFunctionDeleteMortgageEventPermissionProd, SendUnderwriterDecisionEmailFunction]
C:\Users\user\aws\sam\git\tallaght.mortgages\tud.mortgages\tallaght-Mortgages-App>sam package --s3-bucket ad.aws.s3.codedeploy --output-template-file output.template.yaml
Uploading to 0e230f2ca5056ab54999369444117a56 55802 / 55802.0 <100.00>
Successfully packaged artifacts and wrote output template to file output.template.yaml.
Execute the following command to deploy the packaged template
aws cloudformation deploy --template-file C:\Users\user\aws\sam\git\tallaght.mortgages\tud.mortgages\tallaght-Mortgages-App\output.template.yaml --stack-name <YOUR STACK NAME>

C:\Users\user\aws\sam\git\tallaght.mortgages\tud.mortgages\tallaght-Mortgages-App>aws cloudformation deploy --template-file output.template.yaml --stack-name Mortgages-Stack --capabilities CAPABILITY_IAM
Waiting for changeset to be created..
```

Figure 3.1.6

NEXT TASKS

My main priority this week is to implement the Clarification functionality. This will require code changes in the Step Function definition, new Lambda Functions, modifications to existing ones and Angular front end changes. So there is a significant amount of coding still remaining.

If time permits then I will revisit the Circular dependency error that I am encountering with the CloudFormation Step Function definition.

Project Phase	Service / Area	Implementation Task
Build / Development	AWS Step Functions	Extend the Step Function Workflow state machine to support Underwriting Clarification requests to the Broker to allow questions be submitted and answered regarding the mortgage application or applicant. The Broker must then be able to restart the workflow.
Build / Development	AWS S3 / Angular	Build new screens to allow underwriters to submit clarification questions and for the brokers to answer.
Build / Development	AWS Cloudformation	Integrate the deployment of the Workflow state machine into the project Cloud formation template.
Test	AWS Step Functions	Test State Transitions / Events / Human interaction

REFERENCE LIST AND BIBLIOGRAPHY

- [1] AWS Free Tier. <https://aws.amazon.com/free/>
- [2] AWS CLI. <https://docs.aws.amazon.com/cli/latest/userguide/install-windows.html>
- [3] AWS CLI SAM. <https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-install-windows.html>
- [4] Node JS. <https://nodejs.org/en/download/>
- [5] Configure AWS CLI. <https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html>
- [6] Visual Studio Code. <https://code.visualstudio.com/download>
- [7] Configure AWS on Visual Studio Code. <https://docs.aws.amazon.com/toolkit-for-vscode/latest/userguide/setup-toolkit.html>
- [8] Install Docker Toolbox. https://docs.docker.com/toolbox/toolbox_install_windows/
- [9] DynamoDB Cloudformation Template.
<https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.dynamo/dynamo-create.yml>
- [10] Lambda Cloudformation Template.
<https://github.com/GeneralYeager/tallaght.mortgages/blob/master/tud.mortgages/Tallaght-Mortgages-App/template.yaml>
- [11] Step Function State Machine Template.
https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.workflow/StateMachine_V3.txt
- [12] <https://aws.amazon.com/blogs/compute/announcing-websocket-apis-in-amazon-api-gateway/>
- [13] <https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html>
- [14] <https://docs.aws.amazon.com/step-functions/latest/dg/concepts-input-output-filtering.html>
- [15] <https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html>
- [16] <https://docs.aws.amazon.com/step-functions/latest/dg/connect-to-resource.html#connect-wait-token>

VERSION 1.0

07/12/2019



ALAN DUFFIN

X00159409

CERTIFICATE, CLOUD SOLUTIONS ARCHITECTURE 2019, PROJECT

IMPLEMENTATION PHASE 3.2

TABLE OF CONTENTS

PROJECT PLAN FOR ITERATION 3	3
PROJECT PROGRESS	4
WORKFLOW / FRONTEND INTEGRATION AND TOKEN PERSISTENCE	5
FRONT END ENHANCEMENTS.....	8
STEP FUNCTIONS AND CLOUDFORMATION.....	10
NEXT TASKS	12
REFERENCE LIST AND BIBLIOGRAPHY	13

PROJECT PLAN FOR ITERATION 3.2

The table below (Table A) lists the tasks that I planned to complete for the final implementation deadline.

Project Phase	Service / Area	Implementation Task
Build / Development	AWS Step Functions	Extend the Step Function Workflow state machine to support Underwriting Clarification requests to the Broker.
Build / Development	AWS Lambda	Develop 4 new Lambda functions to support the clarification use cases. These include functions to support CRUD operations and functions for notifications and pause / restart the Step Function workflow.
Build / Development	AWS Angular	Implement new Front end screens to support the clarifications use case.
Build / Development	AWS Cloudformation	Integrate the deployment of the Workflow state machine into the project Cloud formation template.
Test	AWS Step Functions	Test State Transitions / Events / Human interaction

Table A.

PROJECT PROGRESS

This week I completed the following tasks:

- Extend the Step Function Workflow state machine to support Underwriting Clarification requests to the Broker.
- Develop 4 new Lambda functions to support the clarification use cases. These include functions to support CRUD operations and functions for notifications and pause / restart the Step Function workflow.
- Implement new Front end screens to support the clarifications use case.
- I have tested all new state Transitions / Events / Human interaction

Please review the video (MP4) demonstrating the final version of this project at the following URL:

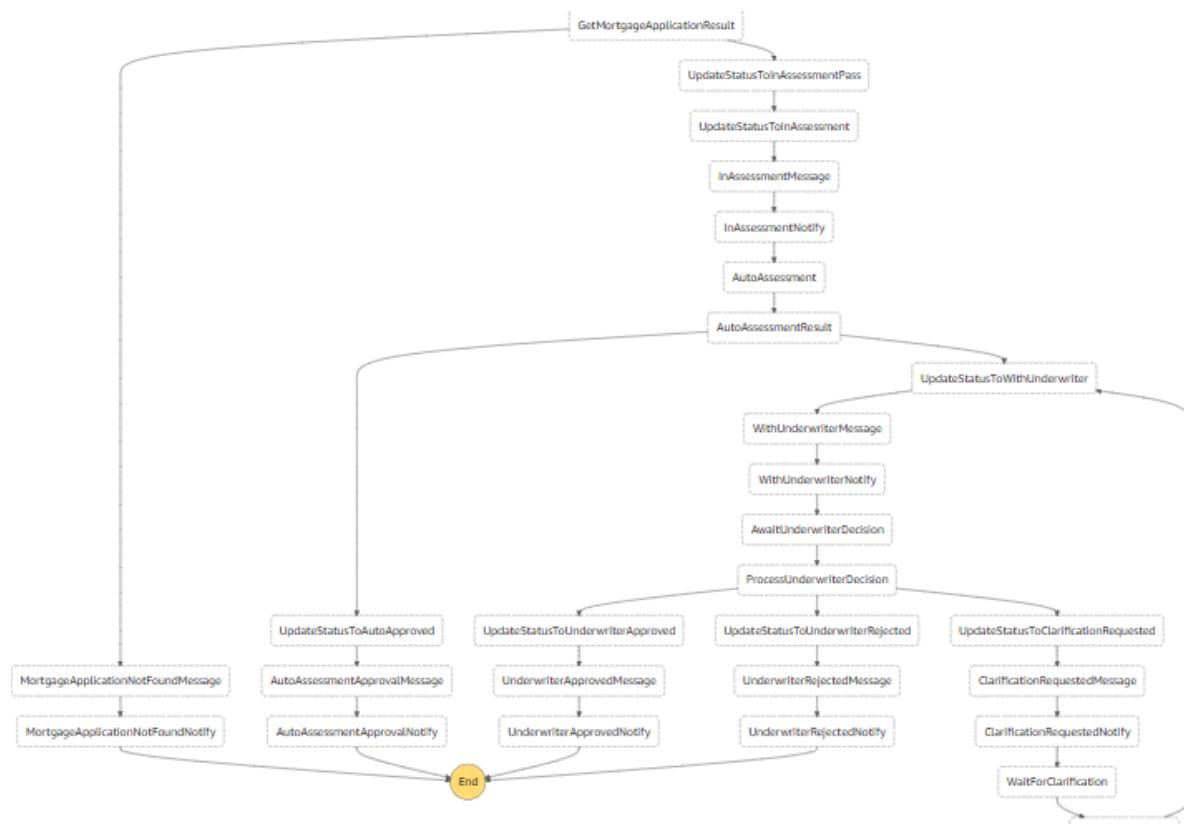
<https://s3-eu-west-1.amazonaws.com/appd.cloud.project/IT-Tallaght-Mortgage-Workflow-WebApp-2019-12-08.mp4>

Due to time constraints I was unable to refactor the Cloudformation Serverless Application Model template to integrate the Step Function deployment. The current iteration of the CloudFormation template includes the packaging and deployment of the following AWS components:

- WebSocket API Gateway Resources
- HTTP Restful API Gateway Resources
- DynamoDB Tables: Mortgages, Customers, Step Function Tokens, Clarifications, WebSocket connections, ID generation.
- Lambda Websocket Functions
- Lambda Mortgage / Customer / Clarification CRUD Functions
- Lambda Step Functions Support Functions
- Lambda SNS Functions
- SNS Topics
- SES Endpoints

STEP FUNCTIONS – CLARIFICATIONS - FINAL VERSION

The final version of my Step Function workflow is shown in Figure AWS.STEP.1. In the past week it have been enhanced to include the branches that allow clarifications between the humans involved in the Mortgage applicatoin process. It allows an unlimited number of question / response clarifications to happen between the Underwriter and the Broker to provide a clear understanding of the financial / personal circumstances of the applicants.



AWS.STEP.1

LAMBDA FUNCTIONS - CLARIFICATIONS

In order to support the enhancements to the Workflow that allow Underwriters approve / reject / clarify mortgage applications from the Browser based front end a number of additional Lambda functions are required.

The following Lambda Functions have been added :

- Mortgages-Stack-BrokerClarificationPOSTFunction
- Mortgages-Stack-CreateNotificationMessageFunction
- Mortgages-Stack-AddClarificationFunction
- Mortgages-Stack-BrokerClarificationPOSTFunction
- Mortgages-Stack-DeleteClarificationFunction
- Mortgages-Stack-GetClarificationsFunction
- Mortgages-Stack-WaitForBrokerClarificationFunction

Example Code.

Source Code on Git Hub: <https://github.com/GeneralYeager/tallaght.mortgages>

ClarificationDB.js ; Utility Function.

```
'use strict';

const AWS = require('aws-sdk');
const ddb = new AWS.DynamoDB.DocumentClient({ apiVersion: '2012-08-10' });
const { CLARIFICATION_TABLE_NAME } = process.env;

exports.queryClarifications = async function (mortgageId) {

    try {
        var clarificationParams = {
            TableName : CLARIFICATION_TABLE_NAME,
            KeyConditionExpression: "mortgageId = :mortId",
            ExpressionAttributeValues: {
                ":mortId": mortgageId
            }
    }
}
```

07/12/2019

Cloud Solution Architecture, Project

```
    };

    let clarifications = await ddb.query(clarificationParams).promise();
    return clarifications;
} catch (error) {
    console.log(error);
    return null;
}
};

exports.addClarification = async function (mortgageId, currMessageId, text) {

try {
    var newMessage = {
        TableName: CLARIFICATION_TABLE_NAME,
        Item: {
            "mortgageId": mortgageId,
            "messageId": currMessageId + 1,
            "text": text
        }
    };
    const result = await ddb.put(newMessage).promise();
    return result;
} catch (error) {
    return null;
}
};

exports.deleteClarifications = async function (clarification) {
try {
    var params = {
        TableName: CLARIFICATION_TABLE_NAME,
        Key: {
            'mortgageId' : clarification.mortgageId,
            'messageId' : clarification.messageId
        }
    };
    const deleteResult = await ddb.delete(params).promise();
    return deleteResult;
} catch (error) {
    return null;
}
};
}
```

FRONT END CLARIFICATIONS

I have enhanced the front end Angular code to allow brokers and underwriters to interact via a new real time clarification process.

The new screenflow is shown in the Screen shots below.

The Broker submits a Mortgage application for Decision. Figure ANGULAR.1.

The screenshot shows a web application for managing mortgages. At the top, there's a navigation bar with links like 'Apps', 'Allergy', 'Banking', 'IT', 'Sports', 'Twitter', and a profile icon for 'damien ivory'. Below the navigation, the main title is 'allaght Mortgages Broker'. There are three buttons: 'Home', 'Create New Mortgage' (which is highlighted), and 'Reload Mortgages'. The main content area displays a table of existing mortgage applications:

mortgageId	customerId	employerName	loanAmount	mortgage			
5000005	60002	Ryanair	222000	PreSubmission			
5000004	60003	AIB Ltd	999999	PreSubmission	100001	91	99
5000002	60002	Smurfit	300009	PreSubmission	50009	39	29

Below the table, there are input fields for a new application:

- Employer Name: Smurfit
- Years in Employment: 29
- Salary: 50009
- Loan Amount: 300009
- Term (Years): 39

At the bottom, there are two buttons: 'Save' and 'Submit For Approval'.

A modal dialog box is overlaid on the page, containing the following text:

```
...-project.frontend.s3-website-eu-west-1.amazonaws.com says
Mortgage 5000002 successfully submitted.
{"executionArn":"arn:aws:states:eu-west-1:727432808710:execution:MortgageApprovalWorkflow:54e9cb84-0b49-4aa6-bcd1-e400c1c54f60","startDate":"2019-12-07T12:08:15.071Z"}
```

There is an 'OK' button at the bottom right of the modal.

Figure ANGULAR.1.

The Underwriter receives a notification of the requirement to approve or reject the mortgage application if the application was not automatically passed by the Auto Assessment rules. Figure ANGULAR.2

The screenshot shows a web-based mortgage underwriting interface. At the top, there is a navigation bar with various links: Apps, Allergy, Banking, II, Sports, Twitter, GUU Greatest Album..., Cars Northern Irela..., and City of London Web. Below the navigation bar, a yellow header bar displays the text "Underwriting Decision required for Mortgage [5000002].". The main content area has a title "Tallaght Mortgages Underwriters Page". Below the title, there are two tabs: "Home" (selected) and "Reload Mortgages". The main form contains the following data:

mortagageld	customerId	employerName	loanAmount	mortgageStatus	salary	term	yearsInEmployment
5000002	60002	Smurfit	300009	WithUnderwriter	50009	39	29
employerName		Years in Employment		Salary			
Smurfit		29		50009			

Below the table, there are input fields for "Loan Amount" (300009) and "Term (Years)" (39). At the bottom of the form are three buttons: "Approve", "Decline", and "Clarify".

Figure ANGULAR.2.

The Underwriter can input a question or request if they require clarification or action by the Broker that is dealing with the applicants. Figure Angular.3

The screenshot shows a simple form for inputting a question or clarification. At the top left is a "Back" button. To its right is a text input field with the placeholder text "Type here." Below the input field is a question: "How much do the applicants pay in rent per month?". At the bottom of the form is a "Submit Clarification" button.

Figure Angular.3

The Broker will receive notification that the Underwriter has changed the status of the application to 'ClarificationRequested' and that the need to respond or take action regarding documentation. FIGURE ANGULAR.4

The Mortgage [5000002] has been referred to an Underwriter for further review.

The Underwriter has requested clarification about certain aspects of Mortgage [5000002].

Tallaght Mortgages Brokers Page

[Home](#) [Create New Mortgage](#) [Reload Mortgages](#)

mortgageId	customerId	employerName	loanAmount	mortgageStatus	salary	term	yearsInEmployment			
5000005	60002	Ryanair	222000	PreSubmission	220001	34	23			
5000004	60003	AIB Ltd	999999	PreSubmission	100001	91	99			
5000002	60002	Smurfit	300009	ClarificationRequested	50009	39	29			
employerName		Years in Employment			Salary					
Smurfit		29			50009					
Loan Amount			Term (Years)							
300009			39							
Save		Respond to Clarification								

FIGURE ANGULAR.4

Back

How much do the applicants pay in rent per month?

Type here.
They pay 1500 euro

[Submit Clarification](#)

FIGURE ANGULAR.5

Once the Broker has completed the Clarification task and resubmitted the mortgage application the Workflow returns the control to the Underwriter for further clarifications or final assessment. FIGURE ANGULAR.6

Underwriting Decision required for Mortgage [5000002].

Tallaght Mortgages Underwriters Page

Home Reload Mortgages

mortagagelId	customerId	employerName	loanAmount	mortgageStatus	salary	term	yearsInEmployment
5000002	60002	Smurfit	300009	WithUnderwriter	50009	39	29
employerName	Years in Employment	Salary					
Smurfit	29	50009					
Loan Amount	Term (Years)						
300009	39						

FIGURE ANGULAR.6

The Underwriter can view the Broker feedback. FIGURE ANGULAR.7

Back

How much do the applicants pay in rent per month?

They pay 1500 euro

Type here.

FIGURE ANGULAR.7

In this case the Underwriter has received clarification about the applicant rental payments and decides to Approve the application. FIGURE ANGULAR.8

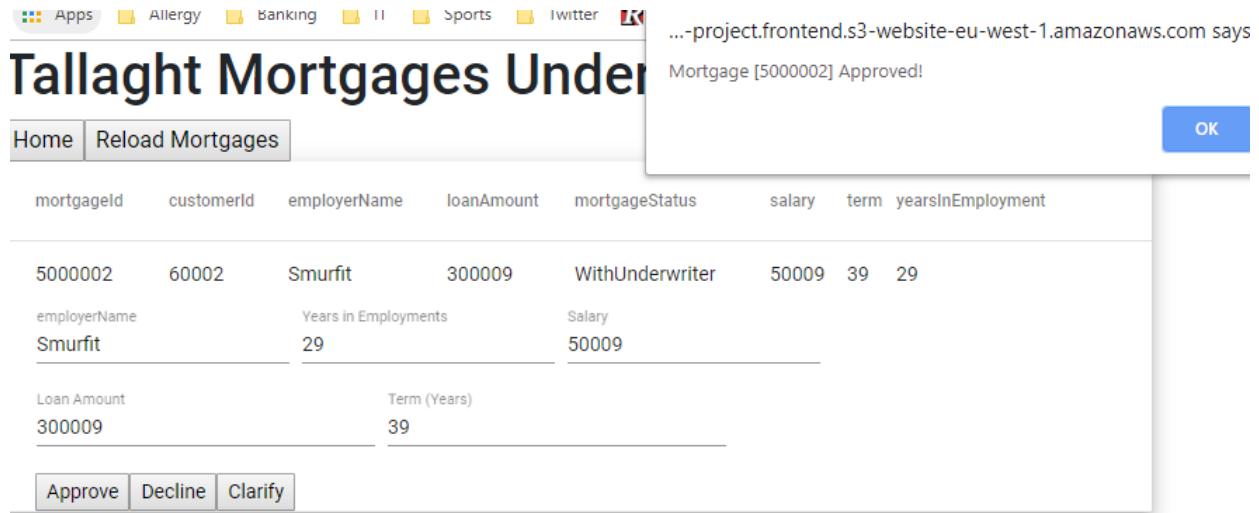


FIGURE ANGULAR.8

Finally the Broker receives a notification that the Mortgage Application has been Approved. FIGURE ANGULAR.9

The Mortgage [5000002] has been referred to an Underwriter for further review.

The Mortgage [5000002] has been approved by the Underwriter.

Tallaght Mortgages Brokers Page

mortgageId	customerId	employerName	loanAmount	mortgageStatus	salary	term	yearsInEmployment
5000005	60002	Ryanair	222000	PreSubmission	220001	34	23
5000004	60003	AIB Ltd	999999	PreSubmission	100001	91	99
5000002	60002	Smurfit	300009	Approved	50009	39	29
5000003	60003	Microsoft	200000	PreSubmission	100000	20	11

FIGURE ANGULAR.9

REFERENCE LIST AND BIBLIOGRAPHY

- [1] AWS Free Tier. <https://aws.amazon.com/free/>
- [2] AWS CLI. <https://docs.aws.amazon.com/cli/latest/userguide/install-windows.html>
- [3] AWS CLI SAM. <https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-install-windows.html>
- [4] Node JS. <https://nodejs.org/en/download/>
- [5] Configure AWS CLI. <https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html>
- [6] Visual Studio Code. <https://code.visualstudio.com/download>
- [7] Configure AWS on Visual Studio Code. <https://docs.aws.amazon.com/toolkit-for-vscode/latest/userguide/setup-toolkit.html>
- [8] Install Docker Toolbox. https://docs.docker.com/toolbox/toolbox_install_windows/
- [9] DynamoDB Cloudformation Template.
<https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.dynamo/dynamo-create.yml>
- [10] Lambda Cloudformation Template.
<https://github.com/GeneralYeager/tallaght.mortgages/blob/master/tud.mortgages/Tallaght-Mortgages-App/template.yaml>
- [11] Step Function State Machine Template.
https://github.com/GeneralYeager/tallaght.mortgages/blob/master/mortgage.workflow/StateMachine_V3.txt
- [12] <https://aws.amazon.com/blogs/compute/announcing-websocket-apis-in-amazon-api-gateway/>
- [13] <https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html>
- [14] <https://docs.aws.amazon.com/step-functions/latest/dg/concepts-input-output-filtering.html>
- [15] <https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html>
- [16] <https://docs.aws.amazon.com/step-functions/latest/dg/connect-to-resource.html#connect-wait-token>

CERT.CLOUD.SOLUTIONS FINAL PROJECT CONCLUSION

Serverless architecture is a development paradigm where applications (composed of many small, independent services/components) are run in cloud PaaS services perhaps on containers temporarily created for executing individual functions. Serverless is also referred to as Function as a Service (FaaS).

The advantages are rapid and automatic scaling and the cost advantages of paying only for what you use, i.e. the number of requests and function execution duration.

There are however also significant drawbacks. These include:

- Vendor lock-in: Once you have invested in building the software and the operations processes around a serverless application like AWS Lambda it will be expensive and time-consuming to port to any other vendor. The workaround for this is to create a separate function for your business logic and then call that from a Lambda handler. Although there are some ways to mitigate vendor lock-in, there are none for vendor control.
- Execution Constraints: The temporary containers used in AWS Lambda result in limits on the amount of memory and execution time. Currently Lambda functions are limited 128 KB for event request body. Memory is limited at 2GB RAM and 5 minutes of execution time. Disk space is also limited to 512 MB. So for functions that may be short lived but need bursts of memory, perhaps to read / transform large files would not be suitable for AWS Lambda. The workaround for limited execution time could be to use a step function (which we discuss later) or SQS to build a sequence of small functions or using Lambda to start and AWS Batch process.
- Cold Starts: AWS Lambda uses temporarily created containers, so in order to execute your function as soon as the client submits the request AWS Lambda must create a temporary container, deploy all dependencies and run the required code. After the request is completed, the container may be destroyed. This process can take between 100 milliseconds to 2 minutes. This Cold Start can cause significant response delays. The workaround is cumbersome in that it requires the developer to try to keep the function in a Warm state for example by pinging the function at regular intervals to prevent AWS Lambda reclaiming the container.

Some of these restrictions may be viewed as enforcing good architectural principles. Long running processes are better suited to Beanstalk, AWS Batch or EC2. Large payload requests in many cases indicates a function / service is trying to do too much. A review of how to refactor the function into more modular, composable services/endpoints should be undertaken. Some of these limitation prevent developers trying to abuse the Serverless architecture by deploying monolithic applications in an unsuitable way.

07/12/2019

Cloud Solution Architecture, Project

Lambda is a very versatile managed service but I would suggest that at this point it has a set of suitable problem domains and does not at this point provide a platform that will accommodate all use cases. The most suitable use cases that I am aware of at the moment are listed below.

- AWS Lambda and S3 hosted static websites. Host the web frontend files (html, javascript, images etc) on S3, and use content delivery via CloudFront. Lambda functions can host the logic for the API Gateway HTTP endpoints and persist data to a managed Database. Lambda, API Gateway and S3 pricing is pay only for the traffic, so the only fixed cost will be the database.
- Processing S3 Objects. Using S3 event notifications as triggers for Lambda execution. This solution is very cost effective and scaling is managed by AWS.
- Data Transformation. AWS Lambda is highly scalable which provides an ideal platform for ETL type processing and transferring data between S3, Redshift, Kinesis and database services.

Many user of AWS Lambda have found gaps whenever they need the following features:

- Imposing a sequence on a set of Functions
- Error handling / retry behaviour over that set of Functions
- Choose Function execution based on real time data.
- Parallelism
- Long running tasks

Manage state between stateless functions involve setting up queues and databases. This increases overhead and can be time-consuming and complex. AWS Step Functions provide a managed service that allow Scaling, State managements, error handling and auditing. Step Functions provide state machines that manage workflows of many Serverless functions using extensive JSON declarative language. They provide a resilient way to iterate though a sequence of AWS services that allows building multi-step applications. It allows the combination of synchronous external invocations such as HTTP API operations via AWS API Gateway with robust asynchronous state machine processing.

The AWS Lambda team restrict the use of recursive Lambda functions because it is very prone to error. An orchestration service like Step Functions shoud be used instead. You can put explicit branching checks in place and enforce timeouts at the workflow level. It helps prevent accidental infinite recursions.

The barrier to entry is cost. Step Fuction pricing at \$0.025 per 1,000 executions, is 100x times more expensive per invocation than Lambda. This price level makes it difficult to be cost-effective to replace existing custom Lambda solutions that maintain state.

Step Functions have many useful features for the business process use cases - executions can span 12 months, allow task timeouts and interrupts, configure tasks with heartbeats. However, it would be much easier to adopt if it supported low cost, high-scalable state machines using an event-driven paradigm. Modern software development encourages loosely coupled, event driven design. However, Step Functions Workers (called activities) must poll for tasks, and Lambda invocations by Step Functions are synchronous.

One feature that I think would be very useful is to allow the specification of tasks as an ARN or HTTP endpoint (e.g. asynchronous Lambda, SNS topic, HTTP endpoint, etc.), with a URL provided in the payload for posting back the output. This would allow Step Function state machines to more fully integrate with event-driven architectures. The current Step Function solution to this allows developers to call the `waitForTaskToken` method from the State Machine task and pause until a developer calls the Step Function SDK using a token to resume the execution. It would be neater if this could be done directly by using a specific execution HTTP endpoint.

Step Functions makes it easy to set up state management early on, and it continues to work well as your application scales and you add more services into the mix. The state machine can be seen as used as a visual construct in the AWS Admin console and is easy to understand. The States Language supports re-try, loop, back-off, timeout, parallel and choice constructs.

However, the lack of tooling / external IDEs may be an impediment to adoption. It is relatively quick to create complex sequences of tasks Orchestrating a sequence individual Serverless functions.

For complex workflows that involve many different states and branching logic, the visual workflow is an intuitive design and diagnostic tool. Developers and DevOps can look at the workflow diagram for running or completed execution and diagnose. It is intuitive to understand the state of the system without knowing the details of its implementation. It is possible because the workflow rule and design decisions in the workflow have been lifted out of the code and made explicit in a visual format that is easy to follow. It is also good practice to decouple workflow logic from business logic. To do otherwise increases the complexity of the applications and reduces their reuse value. In addition, managing state separately from serverless functions allows developers to focus on business logic.

While AWS Step Functions provides solutions to a number of design challenges in the Serverless framework, transitioning all your orchestration layers may not be suitable for all use cases.

The drawbacks of Step Functions include:

- Configuration with the Amazon States Language Amazon States Language is quite complex; its syntax is based on JSON and therefore optimized for machine readability instead of readability by humans. Learning the language is not practical for non-developers. It is also AWS specific and not open source.
- Decoupling services from the orchestration provides scalable, composable functions, but the developers need to learn an entirely separate service.

However, if you need rapid iteration and all your application actions are performed using AWS Lambda functions, AWS Step Functions is an ideal solution. It will reduce the time spent developing orchestration logic and therefore allow you to focus more on your business logic.

Step Functions doesn't support schedules, and AWS Lambda already has an integrated scheduling system. It's better to use Serverless and Lambda scheduling functionality directly.

Step Functions supports more integration with other AWS Services than it was possible for me to investigate in this project. For example it is possible to directly access DynamoDB, SQS, and SNS without writing Lambda functions for that purpose. There are trade-offs in that the Lambda functions would be available for reuse by other components.

Other points to note:

Pricing. For complex workflow, AWS Step Functions is the only available AWS solution but the price is high. AWS even charge for states that only wait or just pass to the next one.

Modern software development favours event driven architecture and AWS services encourage that model. If Step Functions allowed us to initiate execution in response to an event or automatically pull messages from an Amazon SQS Queue.