|  |
| --- |
| Adjuster Assignment - API Design Document- v0.1 |
| Project: Adjuster Assignment - API  IBM |
| V0.1 |
| **October 18, 2022** |

**Revision Status: Version 1**

**Document Control Information**

**Document Approver(s):**

|  |  |
| --- | --- |
| **Approver Name** | **Role** |
|  |  |
|  |  |
|  |  |

**Document Reviewers:** (Records of each required reviewer must be maintained.)

|  |  |
| --- | --- |
| **Reviewer Name** | **Role** |
|  |  |
|  |  |
|  |  |

NOTE: All Reviewers in the list are considered required unless explicitly listed as Optional.

| **Revision** | **Date** | **Created by** | **Short Description of Changes** |
| --- | --- | --- | --- |
| 0.1 | 18/10/2022 | IBM | Initial Draft |
|  |  |  |  |

**Document Source:**

Table of Contents

[1.0 Executive Summary 3](#_Toc117034812)

[2.0 API Implementation Strategy 4](#_Toc117034813)

[2.1 Architecture Overview Diagram 4](#_Toc117034814)

[2.2 Technological stack 5](#_Toc117034815)

[2.3 Unit Test Strategy 5](#_Toc117034816)

[3.0 Implementation Roadmap 5](#_Toc117034817)

[4.0 Best Practices for Adjuster API’s 6](#_Toc117034818)

[4.1 Architecture 6](#_Toc117034819)

[4.2 Deployment Strategy 6](#_Toc117034820)

[5.0 Alternate Solution approaches Adjuster API’s 6](#_Toc117034821)

[5.2 Technological stack 7](#_Toc117034822)

[5.4 Technological stack 8](#_Toc117034823)

[6.0 Design Principles 8](#_Toc117034824)

# 

# 1.0 Executive Summary

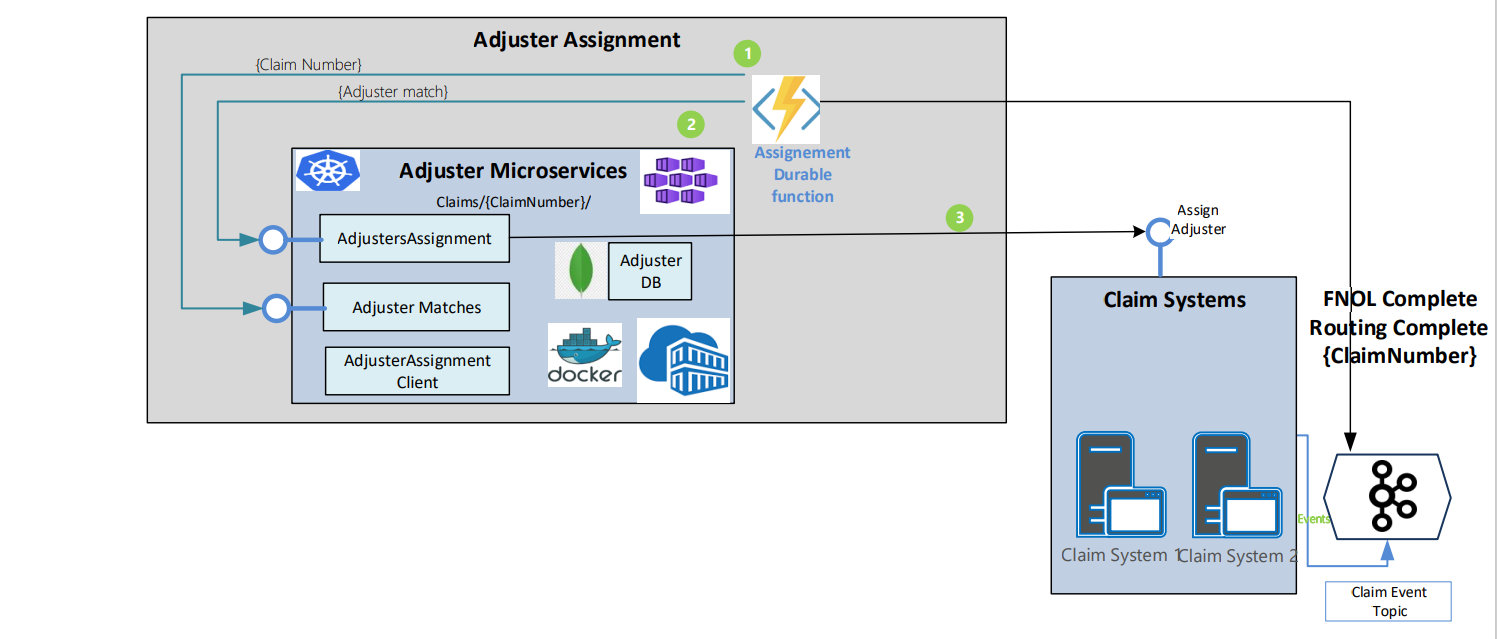
Adjuster assignment is currently using a lot of manual process to assign an adjuster to a given claim. The insurance company seeks to implement a real time automated assigning solution by capturing business rules into an API that will match incoming claims with the right adjustor based on claims characteristics, eligibility, and availability. The insurance company decided to develop the adjuster match service which essentially identifies a pool of adjusters who belong to similar claim group, account, department, track history to settle the claims etc. This document describes the strategy and plan that the IBM team will use to develop the matching service using Microsoft based systems. It discusses the implementation roadmap, estimates for the migration effort, initial Risks, and the mitigation plan.

# 2.0 API Implementation Strategy

## 2.1 Architecture Overview Diagram

The adjuster assignment application is an automated system. The load of incoming claim data is too high in this case, during peak season the load of data increases very rapidly and due to that reason, we need the APIs to be hosted on an orchestrated cluster environment as microservices. The claim systems emit the claim creation events into KAFKA infrastructure. Once event reached to KAFKA will be notified by an Azure function app. The function app intakes the KAFKA topic to process the claim and gets into the process to identify the adjuster for the claim.

Please find the Overall Conceptual Architecture of the Automated Adjuster Assignment.



**Figure 1: Architecture Overview Diagram**

The AdjusterMatches component makes a synchronous call (with Claim details) to the Adjuster Match engine, which returns a list of potentially matching adjusters. Note that the goal is to enhance and tune the rules for adjuster matching over time to return a single adjuster. The AdjusterMatches component returns the list of matches to the Assignment Service.

During the matching process, the Adjuster Match API avails the Supplementary Claim Details to obtain additional Claim details from the Claim system such as the data associated with the routing.

 Adjusters Assignment will be used to Post the collection of matching adjusters to the Claim systems via an endpoint exposed by the claim systems. It also records the assignment into a non-sql database.

## 2.2 Technological stack

* Azure Kubernetes Service.
* Docker.
* Azure Container Registry.
* KAFKA hosted on AKS.
* Azure function app/Azure logic app.
* Microsoft Visual Studio 2022, Visual Studio Code
* MongoDB database.
* Docker- Compose
* Azure CLI
* Azure DEVOPS

### 2.3 Unit Test Strategy

A testing strategy and test scenarios will be identified to formulate a set of acceptance criteria that will have to be met. Basically, in each microservices the Test-driven design must be adopted and thereby ensure that required code coverage of 80% is met. We will be using NUnit framework for our unit testing strategy.

# 3.0 Implementation Roadmap

Assignment API’s will be developed from end-to-end which includes the following steps:

1. Implement the Object Model
2. Develop the decision service in the matching API
3. Develop the assignment service.
4. Unit test the API’s developed using TDD approach.
5. Integrate the claim service to the assignment service.
6. Signoff by vendor

# 4.0 Best Practices for Adjuster API’s

## 4.1 Architecture

Three types of architecture can be set up for Adjuster API’s:

* Using AKS based container orchestrated microservices with a back end.
* Using app services based purely a PAAS solution deployed with API’s with a back end.
* Using Serverless azure workloads with function apps and Azure app service UI with a back end.

There are benefits and disadvantages for each, the choice depending on the cost of set up and maintenance. In all the cases, the MongoDB database will be used as the backend.

## 4.2 Deployment Strategy

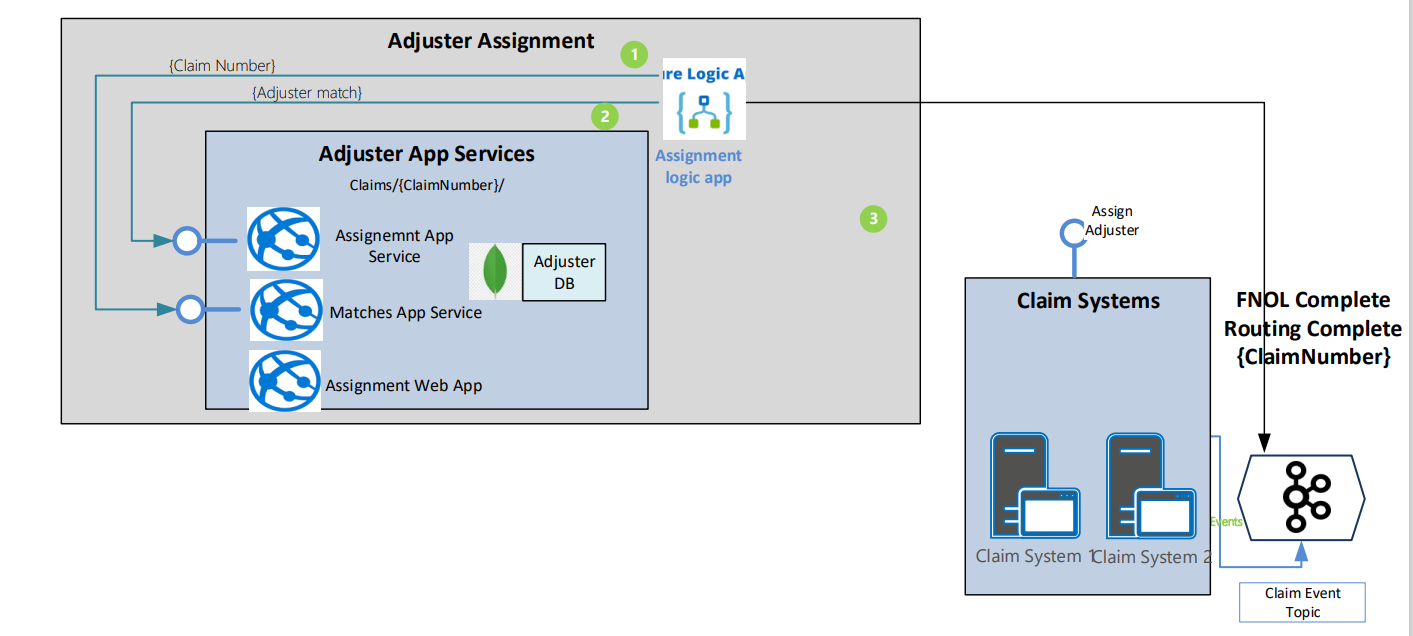
All API’s must be built and deployed using Azure DEVOPS/Azure CLI commands. The Azure DEVOPS pipeline will be based on a continuous integration and continuous development (CI/CD) approach.

The API uses standard tools for source code control and build management such as GIT and MS build and is a perfect candidate to be used alongside with CI/CD software and tools such as NUnit, Nexus, Jenkins.

# 5.0 Alternate Solution approaches Adjuster API’s

5.1 Alternate solution to host all API’s on to Azure App Services

This approach is all about hosting the API’s and UI application into Azure app services. The Mongo DB database is used as the backend system. This architecture demands for the 3 separate PAAS Azure app services and PAAS mongo DB database. This approach recommended when the load of the application is consistent and highly unlikely that the load varies rapidly.

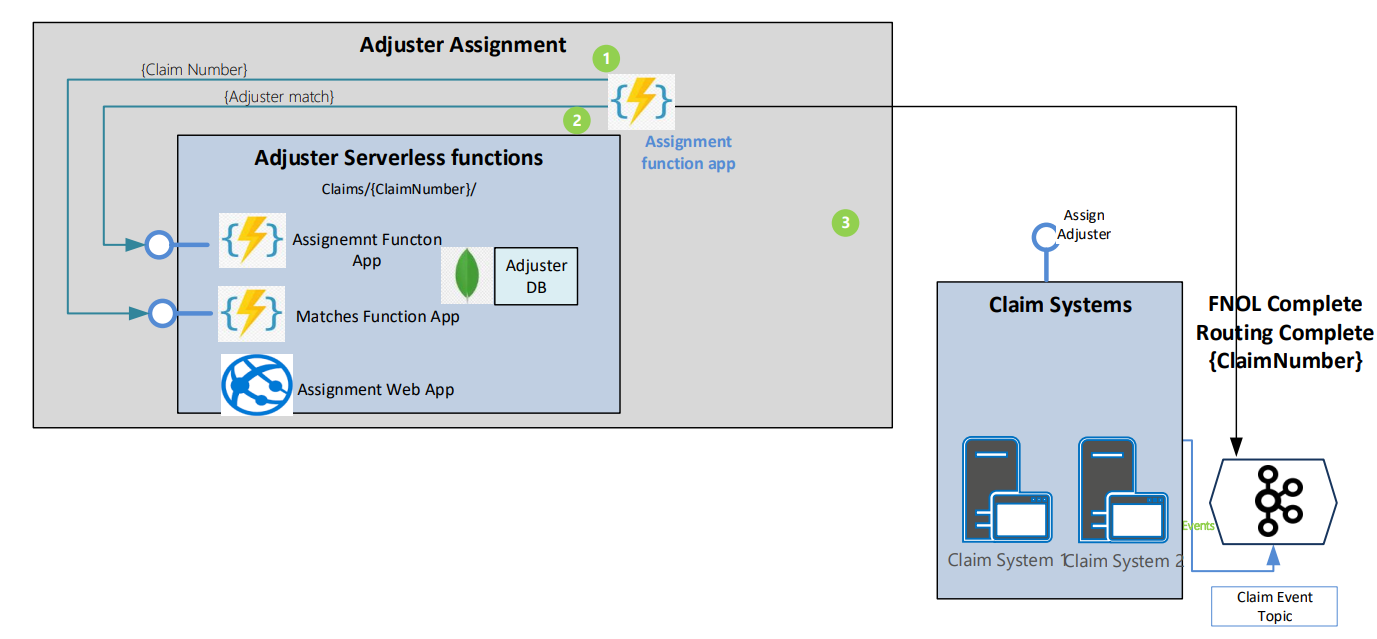


## 5.2 Technological stack

* Azure app service.
* KAFKA hosted on AKS.
* Azure logic app.
* Microsoft Visual Studio 2022, Visual Studio Code
* MongoDB database.
* Azure CLI
* Azure DEVOPS

5.3 Alternate solution to host all API’s on to serverless features

This approach is to host the API’s as serverless Azure functions app and UI application into Azure app services. The Mongo DB database is used as the backend system. 3 Azure functions hosts the API and Azure app services to host the UI application and a PAAS mongo DB database. This approach recommended when the load of the application is very less and due to that application requirement cost also must be reduced. This approach can minimize the infrastructure cost and the development team can concentrate more on application design and development.



## 5.4 Technological stack

* Azure app service.
* KAFKA hosted on AKS.
* Azure function.
* Microsoft Visual Studio 2022, Visual Studio Code
* MongoDB database.
* Azure CLI
* Azure DEVOPS

# 6.0 Design Principles

API design aims at these aspects

* Ability to produce a swagger file from code that follows the Open API Standard.
* Register REST API endpoint in APIM using swagger file.
* Use standard protocols to ensure platform independence
* Web APIs should be discoverable and must have the ability to add functionality independent of client applications.
* Consider the desired API consumer experience and target solutions, design your API gateways accordingly (number of gateways, access, ease of use, etc.)
* Organize APIs on the business entities that the web API exposes
* Use JSON as the primary exchange format
* Adopt a naming convention for resources via URI’s (Uniform Resource Identifier - string of characters used to identify a resource)
  + use plural nouns for URIs that reference collections.
  + Organize URIs for collections and items into a hierarchy
* A collection is a separate resource from the item within the collection and should have its own URI.
* Consider HATEOAS (Hypertext as the Engine of Application State) to enable navigation to related resources
* Avoid requiring resource URIs more complex than collection/item/collection
* Do not create dependencies between the web API and an underlying data source
* Implement a versioning strategy for APIs
* Consider a contract first Open API approach for 3rd Party endpoints
* Leverage **Swagger** to define APIs and generate documentation