|  |
| --- |
| 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](#_Toc117207397)

[2.0 API Implementation Strategy 4](#_Toc117207398)

[2.1 Architecture Overview Diagram 4](#_Toc117207399)

[2.2 Technological stack 5](#_Toc117207400)

[2.3 Unit Test Strategy 5](#_Toc117207401)

[2.4 Persistent volumes 5](#_Toc117207402)

[3.0 Implementation Roadmap 6](#_Toc117207403)

[4.0 Best Practices for Adjuster API’s 7](#_Toc117207404)

[4.1 Architecture 7](#_Toc117207405)

[4.2 Deployment Strategy 7](#_Toc117207406)

[5.0 Alternate Solution approaches Adjuster API’s 7](#_Toc117207407)

[5.2 Technological stack 8](#_Toc117207408)

[5.4 Technological stack 9](#_Toc117207409)

[6.0 Advanced features on each Azure services used. 9](#_Toc117207410)

[7.0 Design Principles 12](#_Toc117207411)

# 1.0 Executive Summary

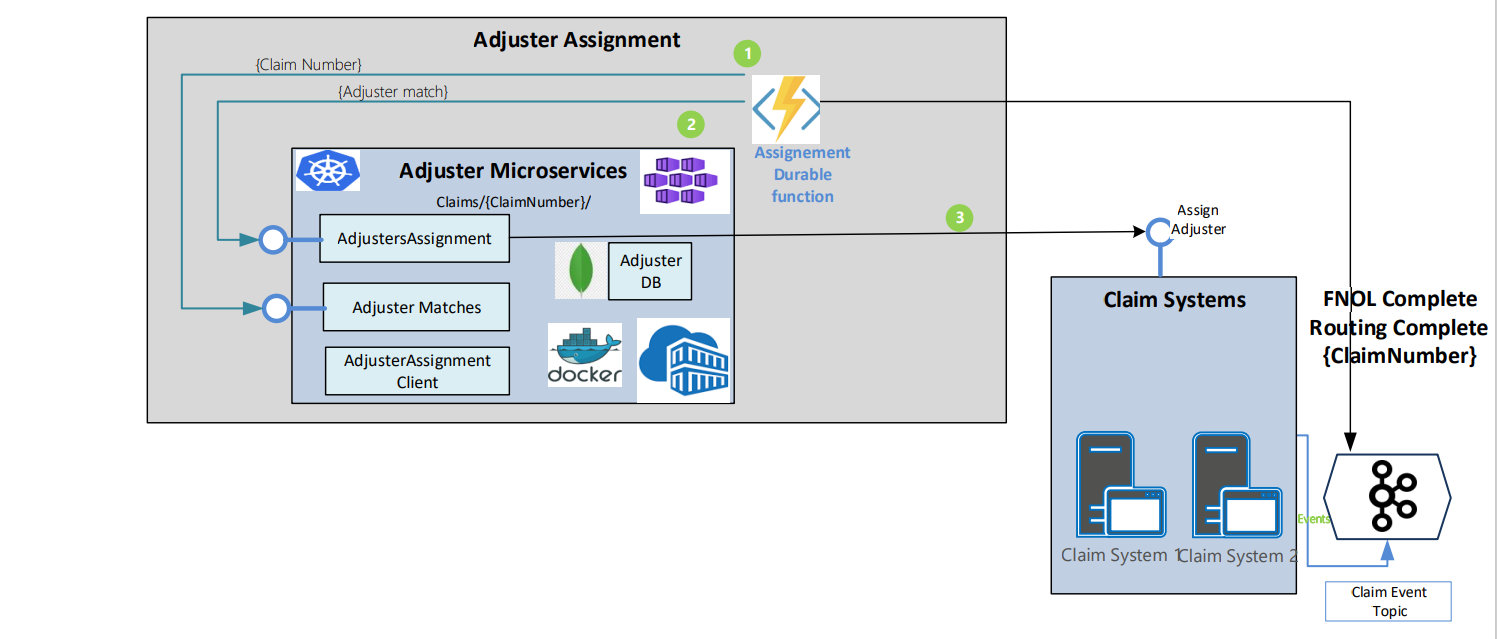
Adjuster assignment is currently using a lot of manual process to assign an adjuster to a given claim. The insurance company seeks to implement a real time automated assigning solution by capturing business rules into an API that will match incoming claims with the right adjustor based on claims characteristics, 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 to identify the right adjusters to process the insurance claims. Since the load of incoming claim data is too high in this specific business scenario, during the peak season, the load of data increases very rapidly, we need the APIs to be hosted on an orchestrated cluster environment as containerized microservices. The claim systems emit the claim creation events into KAFKA infrastructure in the form of TOPICS. 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 identify the matching 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.

Adjusters Assignment client will be used to display the assignments happened, set up adjuster profile, Counterparty accounts, time, and availability of adjusters, reports etc.

## 2.2 Technological stack

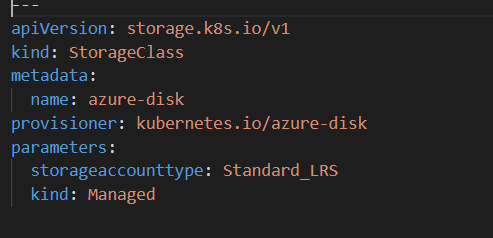
* Azure Kubernetes Service.
* Docker.
* Azure Container Registry.
* KAFKA hosted on AKS.
* Azure function app/Azure logic app.
* Microsoft Visual Studio 2022, Visual Studio Code
* MongoDB database.
* Docker- Compose
* Azure CLI
* Azure DEVOPS

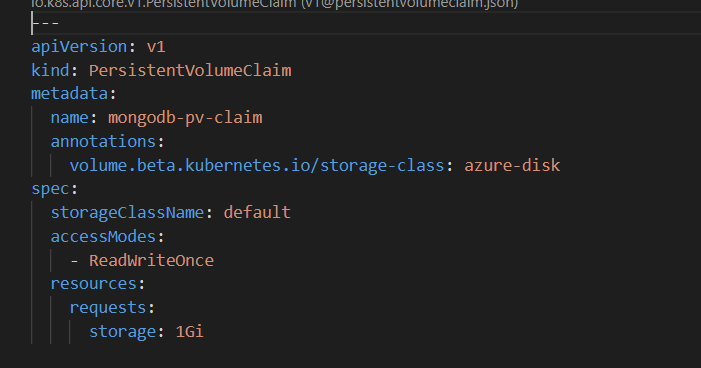
### 2.3 Unit Test Strategy

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.

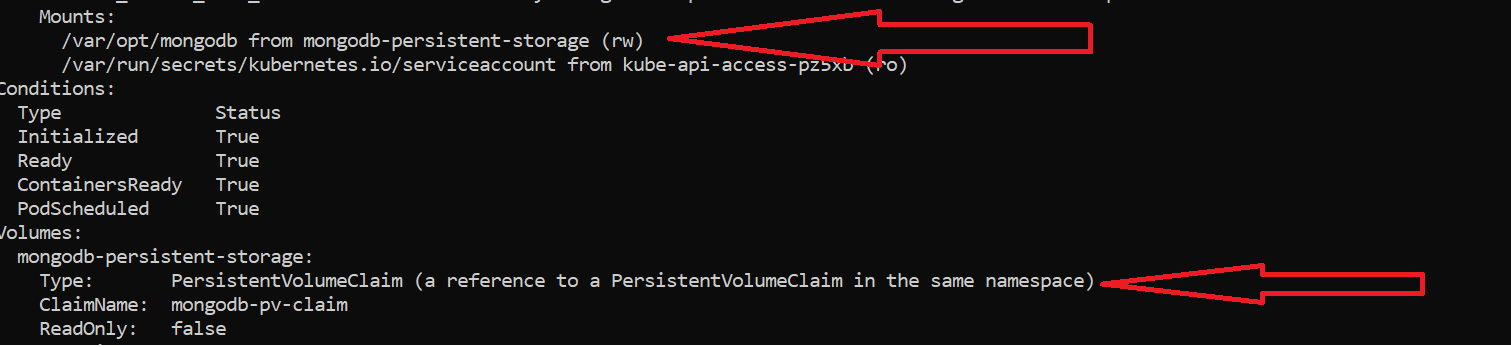
### 2.4 Persistent volumes

The application contains a database container, to persist the state of the data, the persistent volume must be created and attached to pod deployment. To create persistent volume first storage class must be created and then create the persistent volume claims with storage class information. The persistent volume claims information needed to add in the MongoDB POD deployment yaml file. Storage class and persistent volume claims are created as per the below Yaml structure.





After the POD deployed, the mount points start displayed in POD information. The MongoDB POD information has section Mount which shows the persistent storage path as shown below



To check the persistence volume is working as expected, followed these steps.

1. Add few assignments using the Assignment function.
2. Check the Mongo DB data using both Assignment API get/ Assignment UI app.
3. Restart the Mongo DB POD - **kubectl rollout restart deployment mongo-deployment**
4. Check the POD is back to normal using **kubectl get pods command.**
5. Check the assignment data again using Assignment API/ Assignment UI app. There should not be any loss in data which confirms that PV is working fine.

# 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

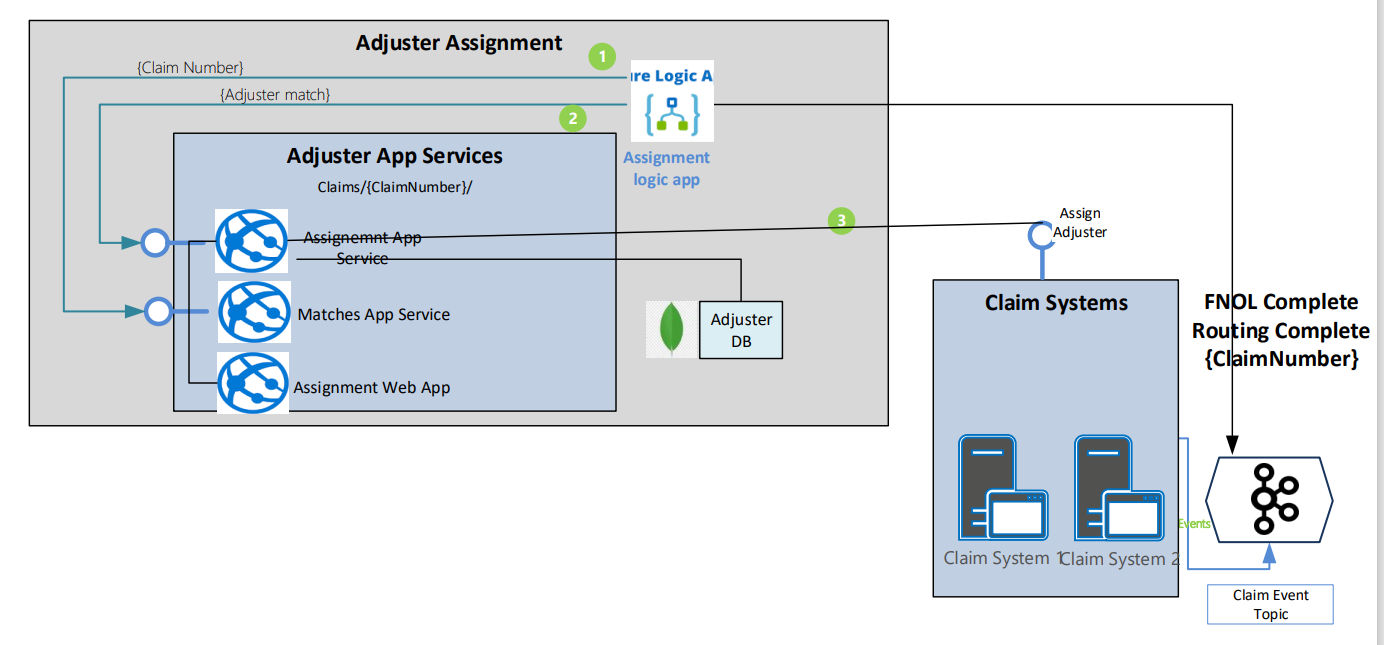
All API’s must be built and deployed using Azure DEVOPS/Azure CLI commands. The Azure DEVOPS pipeline will be based on a continuous integration and continuous development (CI/CD) approach.

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.

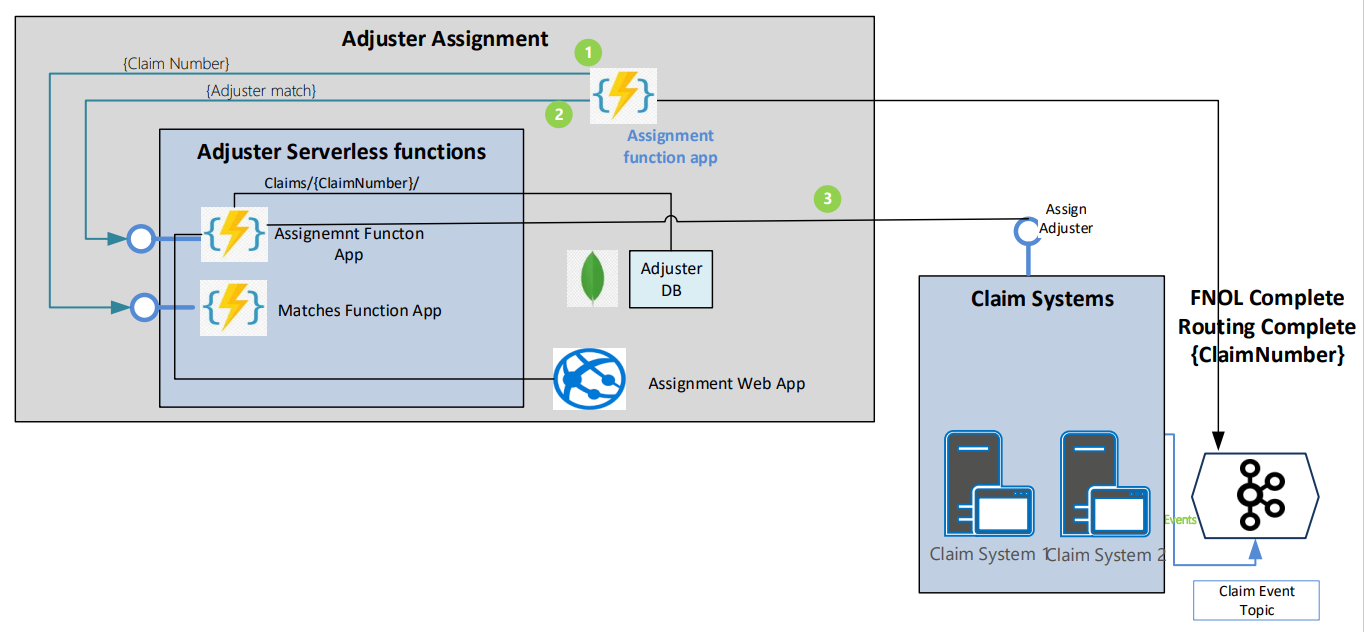


## 5.2 Technological stack

* 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 Advanced features on each Azure services used.

* Implement Autoscaling

Automatically scale your application to the number of pods required to handle the current load. This can be achieved by using Horizontal Pod Auto scaler for CPU & Memory or by using KEDA for scaling based on other sources

* Store secrets in Azure Key Vault

Secrets are not encrypted in etcd, prefer to store your secrets in a proper HSM like Azure Key Vault. You can then inject secrets using CSI provider.

* Use Kubernetes namespaces to properly isolate the Kubernetes resources.

Namespaces give you the ability to create logical partitions and enforce separation of your resources as well as limit the scope of user permissions. Don't forget not to use the Default namespace.

* AAD Integration

Azure Kubernetes Service (AKS) can be configured to use Azure Active Directory (Azure AD) for user authentication. In this configuration, you can sign into an AKS cluster by using your Azure AD authentication token.

* Use AKS and ACR integration without password

For AKS to download/pull images from Azure Container Registry (ACR), it needs the ACR credentials including the password. To avoid saving the password in the cluster, you can simply activate the ACR integration on new or existing AKS cluster using SPN or Managed Identity.

* Choose the appropriate network model

While Kubenet is the default Kubernetes network plugin, the Container Networking Interface (CNI) is a vendor-neutral protocol that lets the container runtime make requests to a network provider. The Azure CNI assigns IP addresses to pods and nodes and provides IP address management (IPAM) features as you connect to existing Azure virtual networks. Each node and pod resource receives an IP address in the Azure virtual network, and no additional routing is needed to communicate with other resources or services.

* Ingress Controller

Ingress controllers bring traffic into the AKS cluster by creating ingress rules and routes, providing application services with reverse proxying, traffic routing/load balancing, and TLS termination. This allows us to evenly distribute traffic across our application services to ensure scalability and meet reliability requirements.

* Monitoring

Naturally, monitoring the day-to-day performance and operations of our AKS clusters is key to maintaining uptime and proactively solving potential issues. Using AKS’ toggle-based implementation, application services hosted on the AKS cluster can easily be monitored and debugged using Azure Monitor.

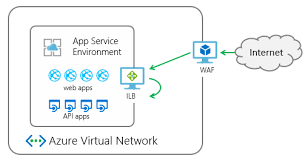
* Pod Identities

Instead of hardcoding static credentials within our containers, Pod Identity is deployed into the default namespace and dynamically assigns Managed Identities to the appropriate pods determined by label. This provides our example application the ability to write to Cosmos DB and our CI/CD pipelines the ability to deploy containers to production and stage clusters.

* App Service Environment

An App Service Environment (ASE) is a deployment of Azure App Service into a subnet in a customer's Azure Virtual Network instance. An ASE consists of:

* Front ends: Where HTTP or HTTPS terminates in an App Service Environment
* Workers: The resources that host your apps
* Database: Holds information that defines the environment
* Storage: Used to host the customer-published apps



The isolated App Service Plan tier, which allows to create a private ASE on a virtual network, allowed to control network security, enable direct site-to-site VPN/EXPRESSROUTE (WAN) connections, and allow App Services to interact with virtual machines at NIC performance speeds instead of Point-to-Site VPN (gateway) speeds. An ASE can be shared publicly using a virtual IP address (VIP or Azure public IP address) or privately using an Azure internal load balancer.

* Application Service Plan

An App Service plan defines a set of compute resources for a web app to run. These compute resources are analogous to the server farm in conventional web hosting. One or more apps can be configured to run on the same computing resources (or in the same App Service plan). App Service plan can be scaled up and down at any time. It is as simple as changing the pricing tier of the plan. We can choose a lower pricing tier at first and scale up later when we need more App Service features.

* Integration Service Environment (ISE)

An integration service environment is a fully isolated and dedicated environment for all enterprise-scale integration needs. When we create a new integration service environment, it’s injected into the Azure Virtual Network allowing us to deploy Logic Apps as a service in the VNET.

* Application Insights

Application Insights is an application performance management service for web applications that enables you to do all the monitoring of your website performance in Azure. It also has a powerful analytic tool that helps to diagnose issues and gain an insight of how people are using the application.

# 7.0 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