**SYSTEM ARCHITECTURE**

**Use Case**

Design and implement an event driven service, where an event of either true or false along with a random city of India is constantly pushed from one service to another service. Events are pushed once in every 2 minutes unless there is any update, if there is any update then the event is pushed instantly. The event receiving service should implement a messaging broker for handling the events and based upon the event value the service should do a specific task. Here the event true means that the fuel lid of the car is opened and false means its closed, when you get true then call an API to know the fuel cost at a given city (for instance {fuellid: true; city: Bangalore}) and assuming that the car can take 1 liter/30 seconds then calculate the amount of fuel got into the tank and calculate the price. For now, price can be logged inside the event receiving service. Car location can be random, for any random city try to get the fuel price from a 3rd party provider and calculate the amount of fuel cost. Explore an 3rd party API which can provide fuel cost of any given Indian city, if you were not able to find out the API then create a mocked service that returns fuel cost for a given Indian city. Considering the cost of fuel will change once in every 24 hours, please consider introducing caching to cache cost of fuel for a requested city if not already requested within 24 hours.

**Purpose**

This document provides high level architectural of the Data Logging in cloud.

**Scope**

This Software Architecture Document provides an architecture of all the components and a brief overview of all the Micro services used for the implementation of the Data Logging in cloud.

**Design Methodology**

**Structured design methodology** is used with hierarchical approach. Top down approach is followed wherein the design is broken down into smaller components and Micro-service Architecture is used for the service design of Integrated Energy Platform.

**Micro service architecture** - is an architectural style that structures an application as a collection of services that are

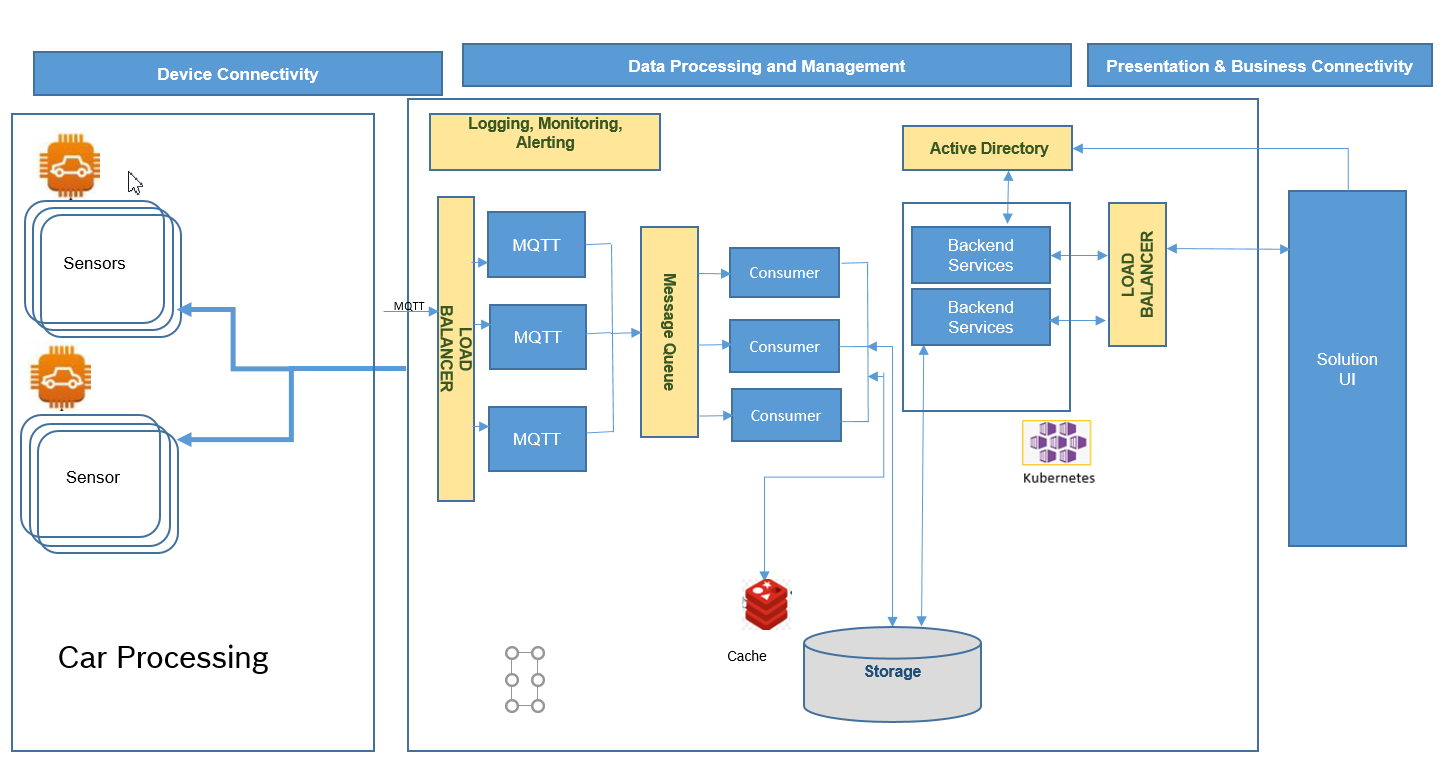
* Highly maintainable and testable
* Loosely coupled
* Independently deployable
* Organized around business capabilities
* Owned by a small team

The micro service architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack.

**Micro – Service Grouping**

| **Sl.No** | **Micro service** | **Features covered** |
| --- | --- | --- |
| 1 | Data Logging | Fuel Diary |

The above grouping can be extended to include more number of services in future if there is a need to accommodate more functionality in the same.



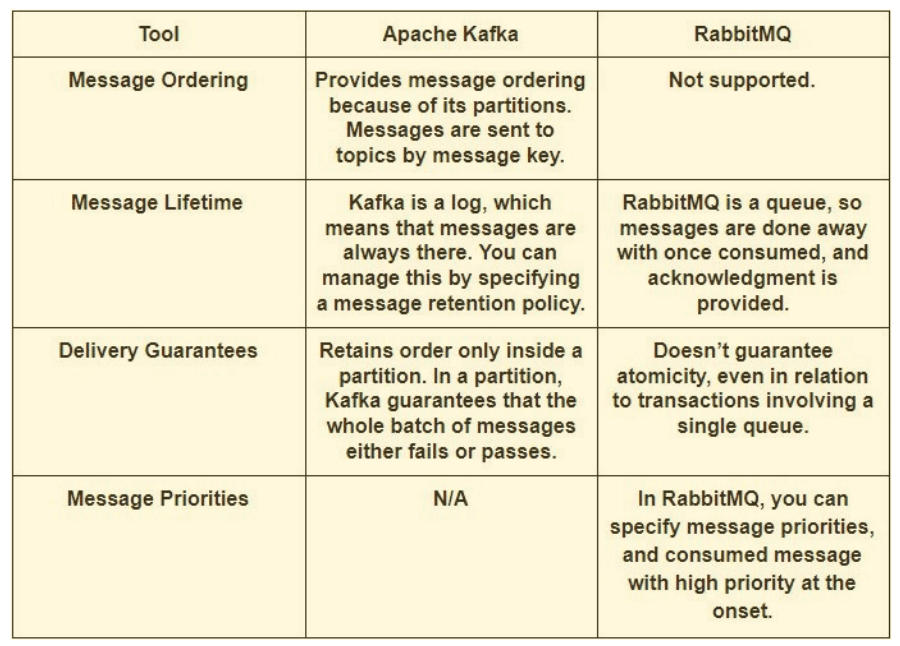
Above design, I have divided the same into three parts i.e.

1. Device Connectivity
2. Data Processing and Management
3. Presentation and Business Connectivity

**Design Notation**

1. **Sensors :** Device which is connected in the Cars to give the status of fuel lid
2. **Car Processing:** Code/Program used to read the values of the sensors connected in the cars.
3. **Load Balancer:** For high availability between sensors to cloud it can be achieved by adding a new HTTP Proxy server to Load Balancer.
4. **Message Queue:** It provides a lightweight buffer which temporarily stores **messages**, and endpoints that allow software components to connect to the **queue** in order to send and receive **messages**.

I am not elaborating which message queue should be used as this purely depends on the use case what will be there as a whole.



1. **Storage:** This means the database which also depends on the use case a whole and this database can be either SQL or NoSQL.

The five critical differences of SQL vs NoSQL:

* SQL databases are relational, NoSQL are non-relational.
* SQL databases use structured query language and have a predefined schema. NoSQL databases have dynamic schemas for unstructured data.
* SQL databases are vertically scalable, NoSQL databases are horizontally scalable.
* SQL databases are table based, while NoSQL databases are document, key-value, graph or wide-column stores.
* SQL databases are better for multi-row transactions, NoSQL are better for unstructured data like documents or JSON.

1. **Cache:** It can be used to store less frequent data also if you really need fast access to that data. We use cache to access the data very fast, so storing most frequent / least frequent data is just a matter of use case.

Caching also purely depends on the use case whether an in-memory cache or disk cache is required.

1. **Kubernetes:** It also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications.
2. **Load Balancer:** It is created when the NGINX ingress controller is deployed. The load balancer routes internet traffic to the ingress.
3. **Active Directory:** AD is recommended for user authentication and API authentication in client applications.

**Namespaces in Kubernetes**

Use namespaces for logically grouping the kubernetes resources such as pods and deployments. Namespaces are intended for use in environments with many users spread across multiple teams, or projects. Every object in a Kubernetes cluster belongs to a namespace. By default, when you create a new object, it goes into the default namespace. But it's a good practice to create namespaces that are more descriptive to help organize the resources in the cluster.

First, namespaces help prevent naming collisions. When multiple teams deploy micro services into the same cluster, with possibly hundreds of micro services, it gets hard to manage if they all go into the same namespace. In addition, namespaces allow you to:

* Apply resource constraints to a namespace, so that the total set of pods assigned to that namespace cannot exceed the resource quota of the namespace.
* Apply policies at the namespace level, including RBAC and security policies

So in this scope we will have 3 namespaces i.e.

* 1. **Dev**: For the development team for their development tasks.
  2. **QA**: For the QA team to perform their testing activities.
  3. **Staging**: For the environment which exactly resembles Production environment to test codes, builds and updates to ensure quality.

# **Security considerations**

### **Role based access control (RBAC)**

Kubernetes both have mechanisms for role-based access control (RBAC):

* RBAC controls access to resources in Azure, including the ability to create new Azure resources. Permissions can be assigned to users, groups, or service principals. (A service principal is a security identity used by applications.)
* Kubernetes RBAC controls permissions to the Kubernetes API. For example, creating pods and listing pods are actions that can be authorized (or denied) to a user through RBAC. To assign Kubernetes permissions to users, you create roles and role bindings:
  + A Role is a set of permissions that apply within a namespace. Permissions are defined as verbs (get, update, create, delete) on resources (pods, deployments, etc.).
  + A RoleBinding assigns users or groups to a Role.
  + There is also a ClusterRole object, which is like a Role but applies to the entire cluster, across all namespaces. To assign users or groups to a ClusterRole, create a ClusterRoleBinding.

AKS integrates these two RBAC mechanisms. When you create an AKS cluster, you can configure it to use Azure AD for user authentication.

### **Secrets management and application credentials**

Applications and services often need credentials that allow them to connect to external services such as Azure Storage or Cosmos Database. The challenge is to keep these credentials safe and not leak them.

**Azure Key Vault** will be used to serve the purpose of storing secrets securely, In AKS, we can mount one or more secrets from Key Vault as a volume. The volume reads the secrets from Key Vault. The pod can then read the secrets just like a regular volume.

**ConfigMaps** will be created using Kubernetes API. This ConfigMap can then be requested when you define a pod or deployment. ConfigMaps are stored within a given namespace and can only be accessed by pods within the same namespace.

**Active Directory** will be used to authenticate the APIs and User logins from the application side.

**Benefits of using Micro-Services:**

* 1. Improved Productivity
  2. Better resiliency
  3. Increased scalability
  4. Continuous Integration/ Continuous delivery(CI/CD)