Universidad San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
Sistemas Operativos 1, Sección N
Catedrático: Ing. Sergio Mendez
Aux: Leonel Aguilar

USAC
TRICENTENARIA
Universidad de San Carlos de Guatemala

MANUAL TÉCNICO PROYECTO 2

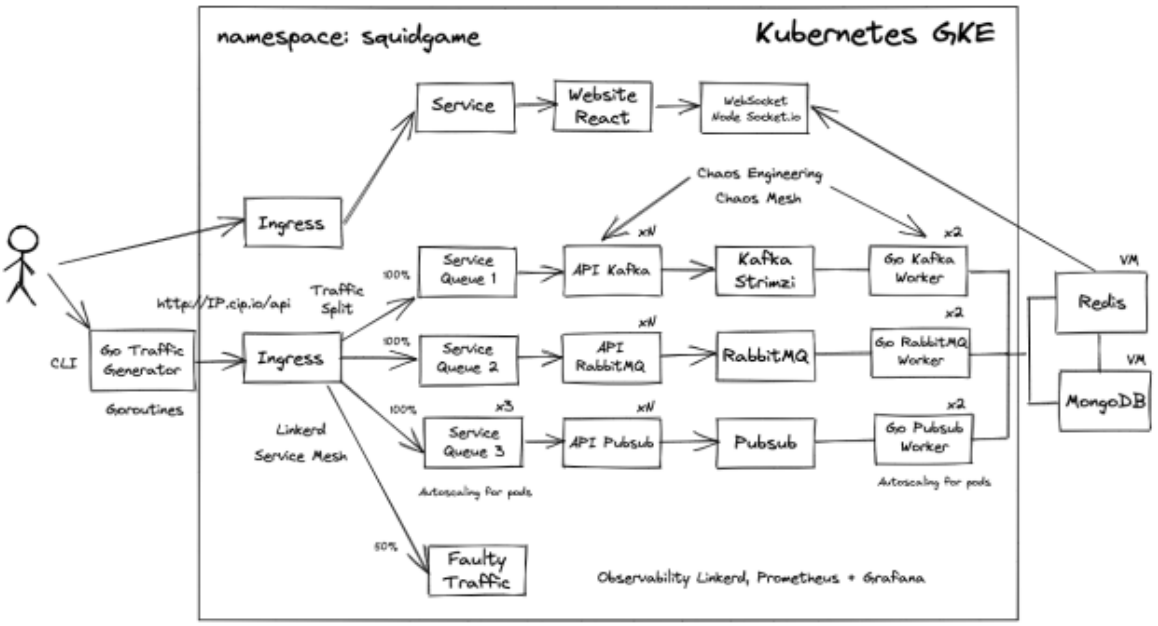
Nombre	Carne
Edi Yovani Tomas Reynoso	201503783
Luis Alfonso Ordóñez Carrillo	201603127
Helmut Efraín Najarro Álvarez	201712350

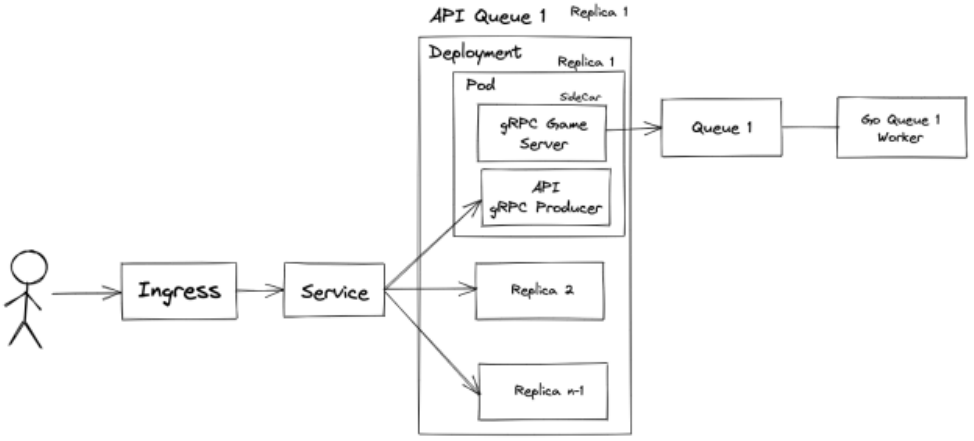
Objetivos

- Experimentar y probar tecnologías Cloud Native que ayudan a desarrollar sistemas distribuidos modernos.
- Monitorear el procesamiento distribuido utilizando tecnologías asociadas a la observabilidad y la telemetría.
- Implementar contenedores y orquestadores en sistemas distribuidos.
- Implementar Chaos Engineering.

Arquitectura

USAC Squid Game

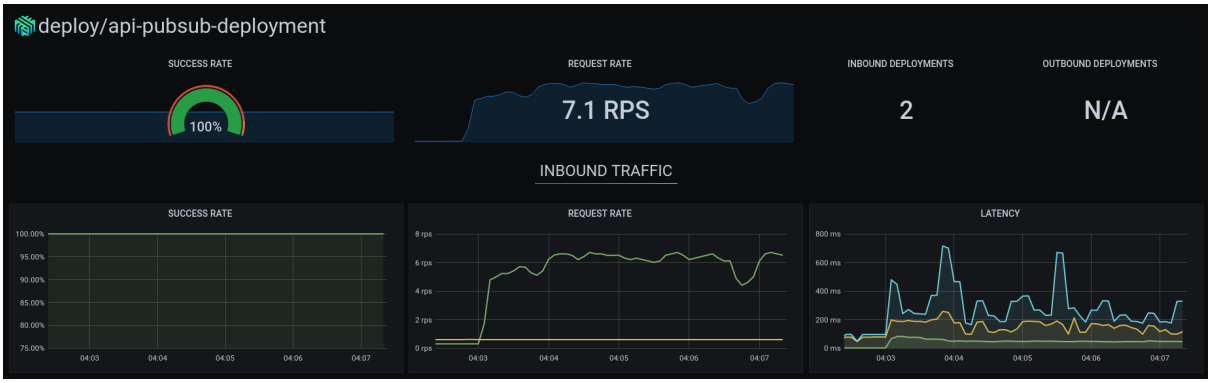




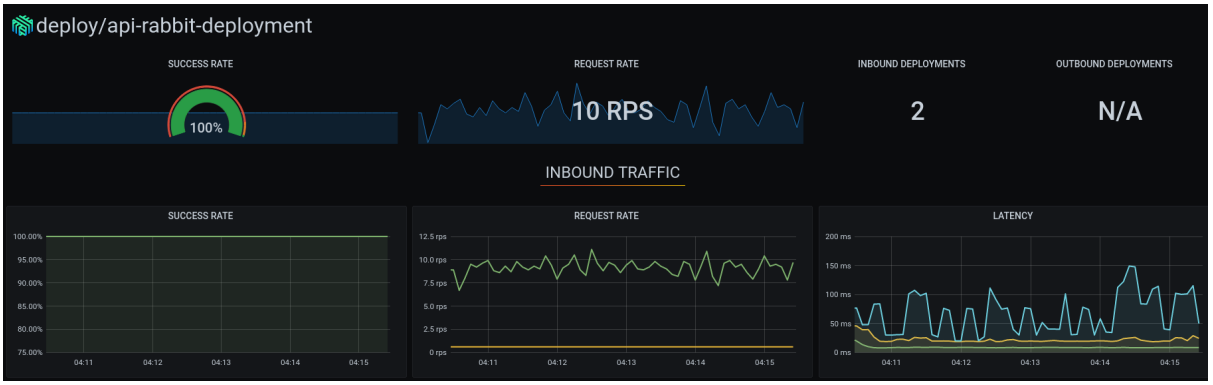
Considere responder las siguientes preguntas:

Cómo funcionan las métricas de oro, cómo puedes interpretar estas 7 pruebas de faulty traffic, usando como base los gráficos y métricas que muestra el tablero de Linkerd Grafana.

PubSub 100%



Rabbit 100%



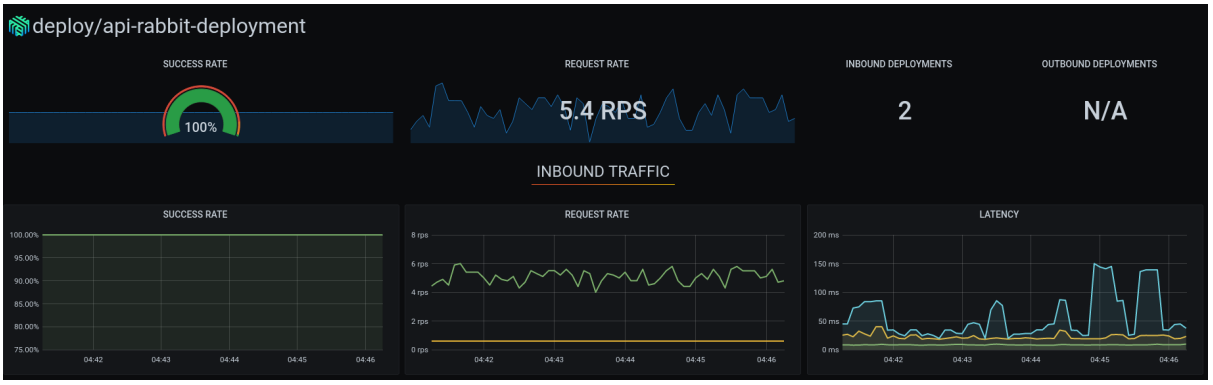
Kafka 100%

PubSub se mantiene en una alta request, pero por momento se caen y tiene una mayor latencia.

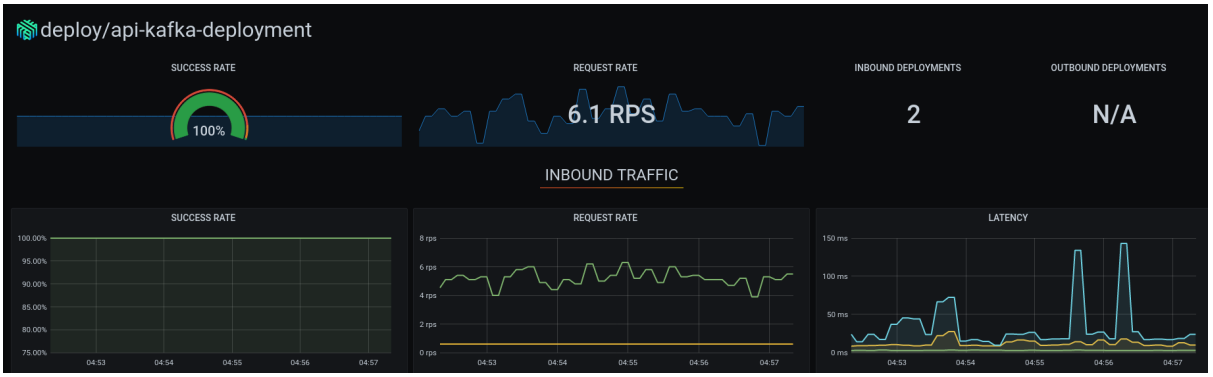
Rabbit tiene menor latencia,pero tiene mejor Request que Pubsub, y la tasa de exito es muy alta.
Kafka tiene una latencia mas pequeña que Rabbit y tiene un Request del mismo parámetro que rabbit y por momento se caen.

PubSub en 50% tiene una latencia de mayor al 100 y request medio alto por momento sube y cae Error el error injection es la tasa más baja en latencia y en request

50% rabbit 50% error



Rabbit en 50% rabbit se cae los request al principio y luego intenta subir los request con la mitad de su capacidad y su latencia se mantiene. y luego se mantuvo su request
Error le quita la mitad de la carga.



Kafka en 50% al principio el rendimiento de kafka se reduce a la mitad y su request al principio esta muerta, al igual que la latencia, pero esperando algunos minutos para que se levante su rendimiento y empieza a mantenerse en la request y la latencia esta empezando a subir pero lo hace de manera muy lenta y en conclusión kafka le cuenta levantar su servicio. y a pesar que el error le quita la mitad de la carga.

Deployment	En la malla de servicios	Tasa de éxito	PPS	Latencia P50	Latencia P95	Latencia P99
api-kafka-deployment	3/3	100.00%	3.68	3 ms	11 ms	18 ms
api-pubsub-deployment	3/3	100.00%	3.17	60 ms	189 ms	269 ms
api-rabbit-deployment	3/3	100.00%	3.5	8 ms	67 ms	167 ms

33% de cada cola
kafka es primero en menor latencia
Rabbit tiene el segundo en latencia
Pubsub es el última en latencia
Pubsub tiene el peor desempeño, kafka tiene el mejor rendimiento y Rabbit se mantiene en el medio de los servicios de mensajeria

Menciona al menos 3 patrones de comportamiento que hayas descubierto.

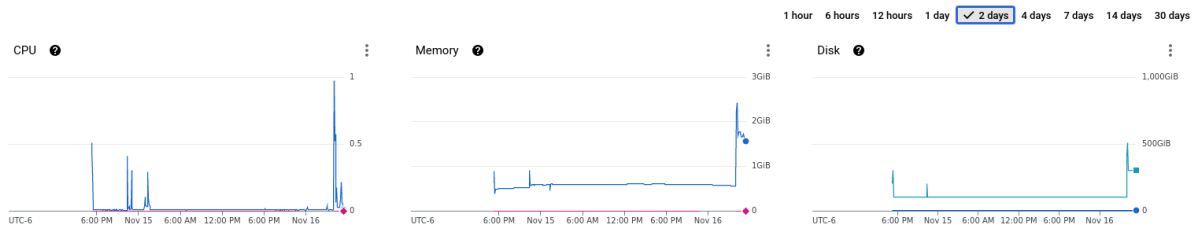
¿Qué sistema de mensajería es más rápido?

R// Kafka el mejor en mensajería por lo visto en la métrica

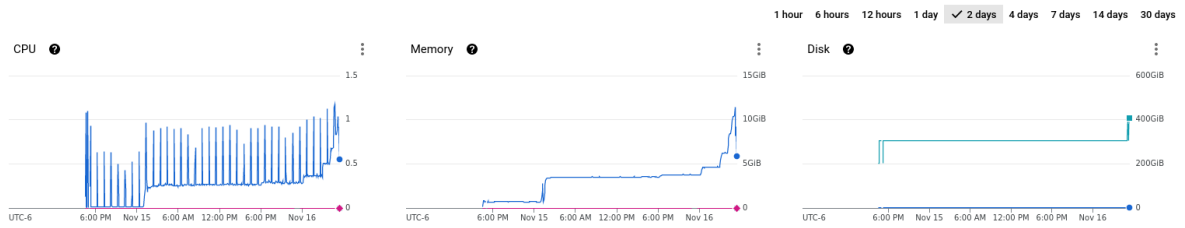
¿Cuántos recursos utiliza cada sistema? (Basándose en los resultados que muestra el Dashboard de Linkerd)

R// Kafka es el que mas consume mas recurso basado en la arquitectura de kafka, trabaja con dos mensajes servidores de mensajería y a parte el suscriber y el publisher

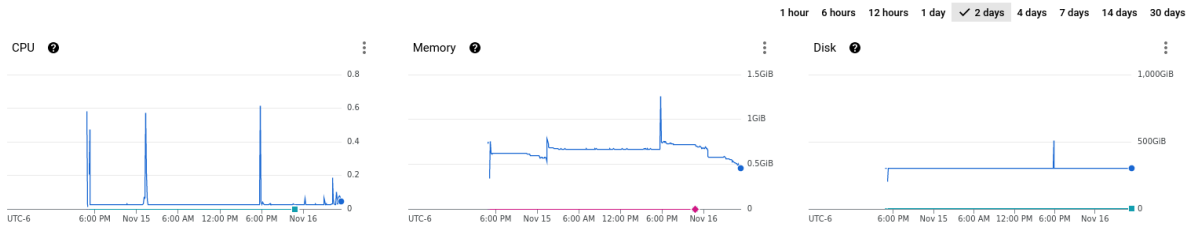
kafka



pubsub



rabbit



¿Cuáles son las ventajas y desventajas de cada sistema?

Kafka

Ventaja

- Su baja latencia, y su implementación de topics que pueden replicarse y particionarse

Desventaja

- Su dependendencia hacia el Zookeeper

Google Pubsub

Ventaja

- La facilidad con la que se programan los publicadores y suscriptores, además de la fácil configuración y monitorización de tópicos.

Desventaja

- Consume demasiados recursos y la lentitud del mismo, ya que depende de un servidor de mensajería en la nube.

Rabbit MQ

Ventaja

- Es fácil de implementar el código y tiene mejor documentación

Desventaja

- El desarrollo de Erlang es difícil de entender el código fuente.

¿Cuál es el mejor sistema?

Si nos limitamos por recursos es rabbitmq, Ahora depende del punto de vista de disponibilidad en la nube asi como pubsub.

¿Cuál de las dos bases se desempeña mejor y por qué?

La combinación de sockets con mongo es muy eficiente gracias al uso de una replica set proporcionado por mongo este nos permite tener informacion al instante.

Redis la información efímera y redundante es muy bueno utilizarlo y es muy bueno.

¿Cómo cada experimento se refleja en el gráfico de Linkerd, qué sucede?
Pod kill se manifiesta en el apartado de pod cuando el pod que mato no tiene métrica y kubernetes intenta levantarlo.
Pod feature refleja la tasa de éxito que se va reduciendo.
Container kill se encarga de matar un contenedor en específico.

¿Qué diferencia tienen los experimentos?

Cada uno ataca a cierta partes de la infraestructura del sistema.

¿Cuál de todos los experimentos es el más dañino?

Todos son dañinos dependiendo con cuánta frecuencia sucede este error.

Lenguaje de Programación

Para esta arquitectura se utilizó tres colas en lenguaje golang.

Kafka
Apache Kafka es una plataforma de transmisión de eventos distribuida por la comunidad capaz de manejar billones de eventos al día. Concebido inicialmente como una cola de mensajería, Kafka se basa en una abstracción de un registro de confirmación distribuido. Desde que fue creado y abierto por LinkedIn en 2011,

Kafka ha evolucionado rápidamente de la cola de mensajería a una plataforma de transmisión de eventos completa. Fundada por los desarrolladores originales de Apache Kafka, Confluent ofrece la distribución más completa de Kafka con Confluent Platform. Confluent Platform mejora Kafka con características comunitarias y comerciales adicionales diseñadas para mejorar la experiencia de transmisión tanto de operadores como de desarrolladores en producción, a escala masiva.

Rabbit

Es un software de intermediario de mensajes de código abierto (a veces llamado middleware orientado a mensajes) que originalmente implementó el Protocolo de cola de mensajes avanzado (AMQP) y desde entonces se ha ampliado con una arquitectura de complemento para admitir el Protocolo de mensajería orientada a texto en streaming (STOMP), Transporte de telemetría MQ (MQTT) y otros protocolos

PubSub

Pub/Sub, que significa publicador/suscriptor, permite que los servicios se comuniquen de forma asíncrona, con latencias de alrededor de 100 milisegundos.

Pub/Sub se usa para las canalizaciones de integración de datos y estadísticas de transmisión a fin de transferir y distribuir datos. Es igual de efectivo que el middleware orientado a la mensajería para la integración de servicios o como una cola con el fin de paralelizar las tareas.

Mongodb y Redis

MongoDB es un programa de base de datos orientado a documentos multiplataforma disponible en origen. Clasificado como un programa de base de datos NoSQL, MongoDB usa documentos similares a JSON con esquemas opcionales. MongoDB es desarrollado por MongoDB Inc. y tiene licencia de Server Side Public License (SSPL).

Redis es un almacén de estructura de datos en memoria de código abierto (con licencia BSD), que se utiliza como base de datos, caché y agente de mensajes. Redis proporciona estructuras de datos como cadenas, hashes, listas, conjuntos, conjuntos ordenados con consultas de rango, mapas de bits, hiperloglogs, índices

geoespaciales y flujos. Redis tiene replicación incorporada, secuencias de comandos Lua, desalojo de LRU, transacciones y diferentes niveles de persistencia en disco, y proporciona alta disponibilidad a través de Redis Sentinel y particionamiento automático con Redis Cluster.

Conversión de mongoDB independiente en un conjunto de réplicas

En este caso de uso, el conjunto de réplicas se utilizará para escuchar todas las inserciones y actualizaciones que pasan por la base de datos, esto para informes en tiempo real.

Instalar mongo

El primer requisito es tener instalado mongodb. Se puede instalar siguiendo la documentación de la página oficial. Edición de la comunidad de Mongo - Ubuntu

Nota: Para esta guía, el servicio mongodb instalado fue MongoDB 5.0.3 Community Edition en Ubuntu 18

2 Preparando el archivo mongod.conf

Una vez que el servicio mongo se esté ejecutando correctamente, editemos el archivo mongod.conf. Este archivo se encuentra en la ruta predeterminada, que es /etc/mongod.conf. Después de abrir el archivo, agregaremos la siguiente configuración:

```
net:
        port: 27017
        bindIp: 0.0.0.0 --> Allow outside traffic

replication:
        replSetName: MyReplicaSetName
```

Luego guarde los cambios y reinicie el servicio mongo usando: sudo systemctl restart mongod Verifique el servicio mongo: sudo systemctl status mongod Si todo está bien, ahora abramos el servicio mongo escribiendo mongo o mongosh en la consola. Una vez que estemos dentro del shell mongo, ejecutemos el siguiente comando

```
rs.initiate({
        _id: "MyReplicaSetName",
        version: 1,
        members: [
                { _id: 0, host : "IP_ADDRESS" }
        ]
});
```

La salida debería ser "OK" si todo está en orden. Ahora, el caparazón de mongo se vería así:

```
MyReplicaSetName:PRIMARY>
```

Ahora que la instancia está usando la réplica primaria, podemos consumir la base de datos en el lado del cliente.

3. Cadena de conexión

Para conectar nuestro conjunto de réplicas a través de una cadena de conexión, el formato se vería así: mongodb: // host: 27017 / database? ReplicaSet = MyReplicaSetName connectTimeoutMS = 60000

- host: el miembro anfitrión base de datos:
- database: La base de datos que vamos a consumir.
- MyReplicaSetName: el nombre del conjunto de réplicas

Modelo de Redis y MongoDB

A continuación se presentan algunos de los modelos que se utilizó en las dos bases de datos NoSQL.

En esta imagen se muestra el primer modelo Log para introducir a la base de datos de mongo y se guarda en la colección de ‘games’

```
type Log struct {

    Request_number int `json:"request_number"`
        Game int `json:"game"`
        Gamename string `json:"gamename"`
        Winner string `json:"winner"`
        Players int `json:"players"`
        Worker string `json:"worker"`

}
```

En esta imagen se muestra como se hace el otro modelo para redis y mongo db este se guarda en la colección de ‘players’.

```
data := struct {
        JuegosGanados int
        Jugador string
        UltimoJuego string
        Estado string
}{
        players[newLog.Winner],
        newLog.Winner,
        newLog.Gamename,
        "Winner",
}
```

Kubernetes

grcp

Se muestra el archivo del servicio de grcp

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: grpc-deployment-pubsub
  name: grpc-deployment-pubsub
  namespace: squidgame
spec:
  replicas: 3
  selector:
    matchLabels:
      app: grpc-deployment-pubsub
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: grpc-deployment-pubsub
      annotations:
        linkerd.io/inject: enabled
    spec:
      hostname: server
      containers:
      - name: grpc-game-server
        image: cascarus/grpc-game-server
        env:
          - name: SERVICE_TYPE
            value: "3"
        ports:
```

Archivo yml de Kafka

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: api-kafka-deployment
  name: api-kafka-deployment
  namespace: squidgame
spec:
  replicas: 1
  selector:
    matchLabels:
      app: api-kafka-deployment
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: api-kafka-deployment
      annotations:
        linkerd.io/inject: enabled
    spec:
      hostname: server-kafka
      containers:
      - name: api-kafka-server
        image: efraalv/kafka-publisher
        ports:
          - containerPort: 3000
      - name: api-kafka-client
        image: efraalv/kafka-subscriber
```

Archivo.yml de pubsub

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: api-pubsub-deployment
  name: api-pubsub-deployment
  namespace: squidgame
spec:
  replicas: 3
  selector:
    matchLabels:
      app: api-pubsub-deployment
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: api-pubsub-deployment
      annotations:
        linkerd.io/inject: enabled
    spec:
      hostname: server-pubsub
      containers:
      - name: api-pubsub-server
        image: efraalv/pubsub-publisher
        ports:
          - containerPort: 8080
      - name: api-pubsub-client
        image: efraalv/pubsub-subscriber
```

Archivo yml de Rabbit

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: api-rabbit-deployment
  name: api-rabbit-deployment
  namespace: squidgame
spec:
  replicas: 3
  selector:
    matchLabels:
      app: api-rabbit-deployment
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: api-rabbit-deployment
      annotations:
        linkerd.io/inject: enabled
    spec:
      hostname: server-pubsub
      containers:
      - name: api-rabbit-server
        image: 429a1ea681b0/gopublisher1
        ports:
          - containerPort: 8080
```

Archivo de Squidgame

```yaml
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: null
  name: squidgame
spec: {}
status: {}
---
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: grpc-deployment-pubsub
  name: grpc-deployment-pubsub
  namespace: squidgame
spec:
  replicas: 3
  selector:
    matchLabels:
      app: grpc-deployment-pubsub
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: grpc-deployment-pubsub
      annotations:
        linkerd.io/inject: enabled
```

Consulta de Mongo

```
export const MostrarInsersion = async (req, res) => {
    const datos = await Logs.aggregate().
        group({ _id: "$worker", Count: { $sum: 1 } })
    res.status(200).send(datos);
}

export const Ultimo10_Juegos = async (req, res) => {
    const datos = await Logs.find().
        limit(10)
    res.status(200).send(datos);
}

export const MostrarDatos = async (req, res) => {
    const datos = await Logs.find()
    res.status(200).send(datos);
}

export const AgruparJugadores = async (req, res) => {
    const datos = await players.aggregate().
        group({ _id: "$jugador", JuegosGanado: { $sum: 1 } }).
        sort({ JuegosGanado: -1 })

    res.status(200).send(datos);
}
```

Consulta Redis

```
    changeStream.on('change', async (data) => {

        client.hgetall("players:all", function (err, value) {
            console.log(value); // > "bar"

            socket.emit("squidgame", {
                games: data.fullDocument, players: value
            })
        })
        //console.log(data.fullDocument); // You could parse out the needed info and send only that data.
    });

}
```

DockerFile

```
FROM node:14

WORKDIR /user/src/app

COPY . .

RUN npm install

RUN npm i nodemon -g

CMD ["nodemon", "index.js"]
```

DockerCompose

```
version: "3.9"

services:

  node:
    build: ./
    image: efraalv/squidgame-backend
    ports:
      - "3500:3500"
    volumes:
        - .:/user/src/app
```

Conclusión

- Se aplico las tecnologia de Cloud Native para este proyecto.
- Se desarrollo varias estrategias para poder aplicar en este proyecto
- Se implemento los procesos de monitoreo en este proyecto.
- Se implemento kubernente como orquestador de contenedores para este proyecto.
- Se implemento Chaos Engineering para este proyecto.