# Arquitectura planteada para Caso BP

## Descripción general

La arquitectura parte de la premisa de una arquitectura de microservicios en donde se deben tener componentes desacoplados y autoadministrables, esto con el objetivo de tener en cuenta pilares de reactividad en donde se toma como punto de partida del diseño que un sistema sea Responsivo, Elástico, Resiliente y Enfocado a mensajes. Con estos pilares se diseña la arquitectura en varios contenedores, cada uno con una responsabilidad especifica de acuerdo a su dominio, también se toman en cuenta aspectos de reusabilidad y escalabilidad aplicables a cada contenedor.

## Pilares de arquitectura

**Escalabilidad:** La plataforma está diseñada para soportar escalabilidad tanto vertical como horizontal. La escalabilidad vertical permite aumentar la capacidad de procesamiento o memoria de los nodos existentes, mientras que la escalabilidad horizontal permite añadir más nodos para distribuir la carga entre múltiples instancias. Estas estrategias se gestionan mediante un monitoreo continuo de los componentes, lo que permite determinar si es más óptimo aumentar la capacidad en un nodo específico o añadir nodos adicionales a ciertos componentes según su carga de trabajo y demanda.

**Disponibilidad:** La arquitectura modular de la plataforma facilita la alta disponibilidad a través de una supervisión constante de cada componente. Además, la configuración de redundancia de datos asegura que, en caso de fallo, los datos estén respaldados mediante almacenamiento con replicación en tiempo real. Esto garantiza que los sistemas de almacenamiento puedan reemplazarse automáticamente por réplicas sin afectar la continuidad del servicio ni la integridad de los datos.

**Rendimiento.** Al tratarse de una plataforma desacoplada, los componentes pueden configurarse y escalarse de manera independiente. Esto permite asignar más recursos a los grupos de componentes que experimentan una mayor demanda, optimizando así la distribución de cargas. Este diseño desacoplado admite tanto escalado vertical (incremento de recursos de hardware en un nodo o instancia) como escalado horizontal (añadiendo más instancias de componentes específicos en caso de cargas elevadas). Además, se aprovechan políticas de escalado automático que permiten ajustar los recursos en tiempo real, asegurando un rendimiento óptimo incluso en picos de demanda sin afectar a otros componentes del sistema.

**Mantenibilidad.** Al ser construida en microservicios, la plataforma tiene el concepto de que cada microservicio tiene una razón de ser y está construido con un solo propósito. Por lo tanto, si existe fallas en un solo componente, este puede ser modificado o a su vez puede ser actualizado en caso de que se requiera mas funcionalidades. Además, que en sus procesos de transaccionalidad este cimentado con colas de mensajería lo que permite añadir mas funcionalidad en caso de requerirse.

**Flexibilidad**. Al tratarse de una plataforma basada en microservicios, componentes serverless y comunicación desacoplada mediante mensajería asíncrona, permite desplegar y escalar cada componente de forma independiente. Esto no solo asegura que la plataforma sea flexible para cumplir con los requerimientos actuales, sino que también está diseñada para adaptarse rápidamente a cambios futuros sin afectar la estabilidad del sistema completo.

**Portabilidad**. La plataforma se basa en contenedores Docker, lo que permite que las aplicaciones puedan migrarse fácilmente a otras nubes o a entornos on-premise utilizando las mismas imágenes. Además, se han utilizado servicios que no dependen de características específicas de una nube en particular, facilitando su portabilidad entre diferentes proveedores de nube y entornos de infraestructura.

**Interoperabilidad**. La arquitectura incluye capas de aplicación diseñadas para integrarse con sistemas externos, incluyendo sistemas legados. Cada microservicio cuenta con una administración específica que permite implementar protocolos y estándares de interoperabilidad, garantizando una comunicación eficiente y segura con sistemas de terceros y otros componentes de la plataforma.

**Costos**. La estrategia de aprovisionamiento de recursos se enfoca en asignar capacidad a servicios de alta demanda, optimizando el uso de recursos y permitiendo pagar únicamente por lo que realmente se utiliza. Esto contribuye a un ahorro significativo, dado que se evita el gasto en servicios infrautilizados.

## Seguridad.

La seguridad es uno de los pilares mas fundamentales en una arquitectura, por lo que a continuación se detalla los aspectos a tomar en cuenta para la implementación de la plataforma

### 1.- Definir políticas de seguridad.

Se identifican todos los componentes a detalle de la arquitectura tales como roles, activos y riesgos en donde puede existir vulnerabilidades para según esto aplicar las políticas acordes a los lineamientos dados por la empresa desde su área de seguridad.

### 2.- Modelo de identidad y control de acceso.

Una vez identificado los componentes y roles, debe establecerse las políticas de acceso tomando como punto de partida el mínimo privilegio es decir sellar los componentes para que únicamente puedan ser accedidos por los recursos necesarios. Estos puntos podrían ayudar a tener una seguridad mas detallada sobre la arquitectura

* Los componentes de experiencia únicamente podrán ser llamados por el apim.
* Mediante network policies se debe asegurar los microservicios para que tengan acceso solo a los componentes necesarios
* La base de datos debe tener un usuario con permisos de lectura, escritura y edición, este usuario debe ser usado únicamente por el api correspondiente de data
* Los datos nunca deben tener borrado físico sino lógico

### 3.- Segmentación y aislamiento.

La arquitectura esta dividida en contenedores que corresponden a un dominio, los cuales están aislados unos de otros permitiendo asi su administración y gestión especifica sin causar impacto en toda la plataforma

### 4 implementación de monitoreo y detección de amenazas

Se pueden implementar la herramienta Azure sentinel que nos ayudara con las estadísticas de eventos ocurridos e información en la plataforma para prevenir con anticipación ataques que se estén planeando en contra de nuestra plataforma. Dado que Azure sentinel es una herramienta de SOAR nos permite el análisis de eventos con información de seguridad de diversas fuentes.

### 5 Cumplimiento de auditorías regulares

Para este punto podemos utilizar el dynatrace, ya que a partir de esto se puede realizar una revisión exhaustiva de logs, ahora si no se quieren usar servicios de azure, siempre se puede optar por un componente para pistas de auditoria que trabaja de manera asíncrona y por eventos

### 6 Guardado de plantillas faciales

El NIST aconseja no guardar imágenes faciales en bruto por lo que una vez que azure face api realiza el análisis y comparación de una imagen, esta puede ser guardada, pero en forma de plantilla la misma que sale a partir de los puntos de referencia faciales y consecuentemente borrar la imagen en bruto.

## Recuperación de desastres

Puesto que la arquitectura esta concebida para contenedores sin estado, la recuperación de desastres es más rápida puesto que no requiere levantar información de sesiones de usuarios o datos pertinentes, además para los datos de cache de sesión y guardado de datos persistentes se usan bases de datos que no dependen de un componente, sino que son servicios serverless.

### Contenedores

Para los contenedores se puede utilizar despliegue multiregion permitiendo asi una alta disponibilidad en caso de que exista una colision en una región especifica, por lo que al existir una caída de un componente, se puede restaurar de forma rápida.

### Cache

La plataforma esta concebida para que sea sin estado ya que toda la información de sesiones se guarda en una redis cache para mantener los datos de sesión del usuario, a partir de esto se puede tener una replicación con Redis sentinel quien realiza un monitoreo del nodo primario para realizar un failover automático

### Sql DataBase

Puesto que en esta base de datos reposa información para el logeo de los clientes, es importante que se mantenga con una alta disponibilidad por lo cual podemos usar **Grupos de** Falla Autogeografiada que nos ayuda con la replica en multiples regiones, replicas de lectura y failover automático

## Restricciones

**Microservicios REST:** Los microservicios a implementarse deberán ser API REST. Para garantizar la integridad y consistencia en la comunicación, cada microservicio debe definir su contrato de API (mediante especificaciones como OpenAPI/Swagger) antes de iniciar la implementación. Esto facilitará la integración entre servicios y asegurará que todas las dependencias estén alineadas con las especificaciones de cada contrato. Los mismos deben implementarse en arquitectura hexagonal para que las integraciones que tengan puedan ser actualizadas con facilidad

**Base de Datos para Administración de Usuarios:** La base de datos destinada a la administración de usuarios será SQL, dada la naturaleza de los datos y las relaciones complejas entre tablas, como roles, perfiles y permisos. Este tipo de datos y la necesidad de integridad referencial hacen que una base de datos NoSQL no sea adecuada para este propósito.

**Dependencia de Servicios Externos:** El sistema no funcionará de forma autónoma, ya que depende de servicios proporcionados por el core para la ejecución de procesos clave. También se requiere la integración con servicios de terceros, tanto para obtener datos como para realizar procesos complementarios y transversales. Para mitigar los posibles riesgos asociados, se implementan mecanismos de control de fallos (por ejemplo, patrones de resiliencia como circuit breakers y retry patterns) para gestionar la dependencia de servicios externos.

**Time to Market en un Modelo de Coreografía:** Debido a que la plataforma se basa en una arquitectura de microservicios organizados mediante un modelo de coreografía, el tiempo de lanzamiento al mercado (time to market) puede verse afectado. Cada producto o funcionalidad debe desarrollarse en su totalidad para asegurar el correcto flujo de eventos entre servicios, lo cual puede limitar la velocidad de lanzamiento de un MVP.

## Decisiones de Arquitectura

**Base de Datos:** Se utiliza Azure SQL como base de datos principal, dado que es un servicio serverless que permite a Azure gestionar tareas administrativas, como actualizaciones y parches, reduciendo la carga operativa y garantizando alta disponibilidad y seguridad.

**Arquitectura Stateless**: Se adopta una arquitectura sin estado (stateless) para garantizar la resiliencia de la plataforma. En caso de caída de un nodo en Azure Kubernetes Service (AKS), el sistema puede restaurarse rápidamente sin afectar la experiencia del usuario, ya que no depende de un estado específico en cada instancia.

**Portabilidad en Servicios de Nube:** Los servicios de nube seleccionados no son específicos de un proveedor, asegurando portabilidad y flexibilidad para futuras migraciones. Por ejemplo, se podría utilizar Cosmos DB o DynamoDB sin comprometer la funcionalidad de la solución. Asimismo, para bases de datos SQL, se contempla tanto Azure SQL como Amazon RDS con SQL, manteniendo la capacidad de migrar entre nubes sin pérdida de funcionalidad.

**Mensajería Asíncrona:** Para la mensajería, se ha seleccionado Kafka, que también está disponible en otras nubes, incluyendo Azure, lo cual facilita la portabilidad del sistema. Esta decisión permite que el sistema pueda ser migrado a otra nube sin cambiar la lógica de mensajería ni el flujo de eventos.

**Orientación a Eventos y Patrones Reactivos:** Para asegurar el desacoplamiento de funciones y la alineación con los principios de diseño reactivo, se implementan patrones de arquitectura orientados a eventos. Esto permite que la plataforma esté basada en mensajes y eventos, promoviendo la independencia de cada función y garantizando la capacidad de respuesta ante cambios y demandas.