Evaluación Módulo 5

Diseño de Arquitectura y Escalabilidad para una Plataforma de Gestión de Proyectos

Nombre: Juan Pablo Urra Jara

Curso: Fundamentos DevOps

Contenido

Análisis de requerimientos y elección de arquitectura

Identificación de los componentes clave del sistema.

La plataforma de gestión administrativa para ProManage debe incluir los siguientes componentes funcionales:

- **Autenticación y gestión de usuarios**: registro, inicio de sesión, recuperación de contraseña, y gestión de roles y permisos (por ejemplo: administrador, jefe de proyecto, colaborador).
- **Gestión de proyectos**: creación, modificación, visualización y eliminación de proyectos; asignación de responsables; definición de fechas de inicio y plazos de entrega; y estados del proyecto.
- **Presupuestos**: asignación de presupuesto por proyecto, posibilidad de modificarlo y mantener un historial de cambios, con alertas por sobreasignación.
- **Gestión de archivos**: carga y almacenamiento de documentos asociados a los proyectos, con soporte para diversos formatos (PDF, imágenes, hojas de cálculo).
- **Perfiles de usuario y control de acceso**: diferentes vistas y permisos según el rol del usuario, con restricciones sobre qué acciones pueden realizar.
- **Panel de control (dashboard)**: visualización de métricas relevantes como avance de los proyectos, consumo presupuestario y cumplimiento de fechas.

Justificación de arquitectura monolítica o basada en microservicios.

Una arquitectura monolítica permite desarrollar y desplegar la aplicación como una sola unidad. Tiene una curva de entrada más sencilla y es adecuada para equipos pequeños y sistemas con bajo nivel de complejidad inicial. Sin embargo, presenta problemas de escalabilidad y mantenimiento a largo plazo, ya que todos los componentes están acoplados.

Por el contrario, una arquitectura basada en microservicios divide el sistema en servicios independientes que se comunican entre sí, usualmente por medio de HTTP o mensajería. Esto permite escalar componentes de forma individual, facilita el mantenimiento, mejora la resiliencia ante fallos y permite una mayor flexibilidad tecnológica.

Dado que ProManage requiere una solución escalable, resiliente y segura, y considerando la naturaleza modular de sus funcionalidades (usuarios, proyectos, presupuestos, archivos), la arquitectura recomendada es la **basada en microservicios**. Esto permitirá mayor independencia entre componentes, despliegues desacoplados y una mejor capacidad de escalar bajo demanda.

Elección de estructura como servicio (PaaS, BaaS, FaaS, IaaS o combinación).

Existen cuatro principales modelos de servicio en la nube: IaaS (Infraestructura como Servicio), PaaS (Plataforma como Servicio), FaaS (Funciones como Servicio) y BaaS (Backend como Servicio). Cada uno presenta ventajas particulares.

- **PaaS** es adecuado para desplegar microservicios de forma gestionada, delegando tareas como balanceo de carga, escalado automático y actualizaciones de sistema operativo. Ejemplos incluyen Azure App Service o AWS Elastic Beanstalk.
- **BaaS** puede utilizarse para componentes no centrales del negocio como la autenticación (por ejemplo, Auth0 o Firebase Auth) y el almacenamiento de archivos (como Amazon S3 o Firebase Storage), acelerando el desarrollo sin comprometer la seguridad.
- **FaaS** resulta útil para tareas puntuales y event-driven como procesamiento de archivos al subirlos, generación de reportes o envío de notificaciones. Plataformas como AWS Lambda o Azure Functions permiten implementar estas tareas sin aprovisionar infraestructura permanente.
- **IaaS** ofrece control total sobre la infraestructura, pero implica una mayor carga operativa. No se recomienda su uso principal en este caso, salvo que existan requisitos muy específicos que lo justifiquen.

En conclusión, se recomienda una **combinación de PaaS, BaaS y FaaS**, donde:

- PaaS se utilice para desplegar los microservicios principales.
- BaaS para componentes como autenticación y almacenamiento de archivos.
- FaaS para tareas asincrónicas o bajo demanda.

Este enfoque permite una solución balanceada entre control, escalabilidad, velocidad de desarrollo y bajo mantenimiento operativo.

Diseño de infraestructura y escalabilidad

Uso de autoescalado y balanceo de carga.

Para garantizar que la plataforma pueda adaptarse dinámicamente a variaciones en la demanda, se debe implementar una estrategia de **autoescalado horizontal**. Esto implica aumentar o disminuir la cantidad de réplicas de los servicios (microservicios) según métricas específicas como uso de CPU, memoria o cantidad de solicitudes por segundo.

El **balanceo de carga** es necesario para distribuir equitativamente las peticiones entrantes entre las réplicas de los servicios disponibles. Este componente mejora el rendimiento del sistema, evita la sobrecarga en nodos individuales y contribuye a una experiencia de usuario consistente.

En el contexto de una arquitectura en la nube, se pueden emplear:

- **Balanceadores de carga gestionados**, como AWS Elastic Load Balancer, Azure Load Balancer o Google Cloud Load Balancing.
- **Autoescalado automático**, configurado mediante políticas en Kubernetes (Horizontal Pod Autoscaler) o directamente en el proveedor PaaS.

Ambos mecanismos son fundamentales para asegurar el correcto funcionamiento del sistema bajo distintos niveles de carga y garantizar la escalabilidad esperada.

Estrategia de resiliencia y alta disponibilidad.

Para lograr una solución resiliente y altamente disponible, se deben implementar las siguientes estrategias:

- **Despliegue multi-zona o multi-región**: alojar instancias del sistema en distintas zonas de disponibilidad dentro de una región (o incluso en regiones diferentes) para protegerse ante fallos de infraestructura localizados.
- **Redundancia de servicios**: mantener múltiples réplicas de cada microservicio corriendo en paralelo para asegurar continuidad ante fallos puntuales.
- **Tolerancia a fallos**: diseño de los microservicios para manejar errores de red, caídas de dependencias externas y reinicios inesperados. Se deben usar técnicas como reintentos automáticos, timeouts controlados y circuit breakers.
- **Base de datos replicada y con respaldo**: emplear bases de datos con replicación automática y mecanismos de backup periódico, idealmente con recuperación ante desastres (disaster recovery).
- **Monitoreo y alertas**: implementar herramientas de observabilidad como Prometheus, Grafana, ELK Stack o servicios gestionados para detectar incidentes en tiempo real.

Estas medidas aseguran que la plataforma siga operativa incluso en presencia de fallos parciales, lo que es esencial para cumplir con los requisitos de disponibilidad de un sistema crítico de gestión administrativa.

Justificación de uso de Kubernetes u otras herramientas de orquestación.

Para gestionar el ciclo de vida de los contenedores, se recomienda utilizar una plataforma de orquestación. La opción más adecuada en este caso es **Kubernetes**, por las siguientes razones:

- **Escalado automático**: permite escalar horizontalmente servicios según métricas en tiempo real.
- **Despliegues controlados (rolling updates, rollbacks)**: garantiza actualizaciones seguras sin interrumpir el servicio.
- **Gestión de recursos**: asigna recursos de forma eficiente entre servicios, evitando la saturación de nodos.
- **Alta disponibilidad**: reinicia automáticamente contenedores fallidos y los redistribuye si un nodo deja de funcionar.
- **Portabilidad y estandarización**: Kubernetes es agnóstico del proveedor de nube, permitiendo mover cargas entre entornos (on-premise, AWS, Azure, GCP).
- **Integración con herramientas de CI/CD, monitoreo y seguridad**: se integra fácilmente con soluciones modernas como ArgoCD, Prometheus, Istio, etc.

Alternativas como **Docker Swarm** o soluciones serverless puras son más simples, pero no ofrecen el mismo nivel de control, flexibilidad y escalabilidad.

Por tanto, para una plataforma compleja y con requisitos de alta disponibilidad y escalabilidad como la de ProManage, Kubernetes es la herramienta de orquestación más adecuada.

Contenedorización y orquestación

Elección de herramientas para la gestión de contenedores (Docker, Podman, etc.).

Para la gestión y creación de contenedores en el entorno de desarrollo y producción, se recomienda el uso de **Docker** como herramienta principal. Las razones son las siguientes:

- **Amplio soporte y comunidad**: Docker es la herramienta de contenedores más utilizada a nivel global, con abundante documentación y soporte en múltiples entornos.
- **Compatibilidad con Kubernetes**: Docker genera imágenes compatibles con Kubernetes sin necesidad de pasos adicionales.
- **Simplicidad en el flujo de trabajo**: su CLI es sencilla y está integrada en múltiples plataformas de desarrollo y CI/CD.
- **Integración con sistemas de orquestación y repositorios de imágenes**: permite una fácil conexión con servicios como AWS ECR, Docker Hub o GitHub Packages.

Alternativas como **Podman** ofrecen beneficios adicionales en cuanto a seguridad y ejecución rootless, pero no son aún tan ampliamente adoptadas en entornos de producción orquestados. Por lo tanto, se sugiere mantener Docker como herramienta estándar para el proyecto.

Diseño del proceso de orquestación con Kubernetes o alternativa.

El proceso de orquestación define cómo se gestionan los servicios contenedorizados en producción. En este caso, se propone utilizar **Kubernetes** para la orquestación, con el siguiente diseño:

- **Agrupación en Pods**: cada microservicio se ejecuta dentro de uno o más Pods, con su propia configuración, variables de entorno y volúmenes.
- **Despliegue mediante Deployment y ReplicaSets**: se asegura que haya una cantidad mínima de instancias por servicio, con actualizaciones seguras y rollback automático.
- **Servicios (Services)**: exponen los Pods internamente mediante un sistema de nombres y balanceo de carga interno.
- **Ingress Controller**: gestiona el acceso HTTP/HTTPS desde el exterior a través de un único punto de entrada, con rutas definidas por dominio o subruta.
- **Autoescalado horizontal (HPA)**: basado en uso de CPU/memoria o métricas personalizadas.
- **ConfigMaps y Secrets**: para gestión de configuraciones y credenciales sin necesidad de incrustarlas en las imágenes.
- **Persistencia**: mediante volúmenes (PersistentVolume y PersistentVolumeClaim) para servicios que requieren almacenamiento duradero (como bases de datos o almacenamiento de archivos temporales).

Este diseño asegura flexibilidad, eficiencia operativa y soporte para crecimiento futuro del sistema.

Explicación de la gestión de registros de contenedores (AWS ECR, Docker Hub, GitHub Packages, etc.).

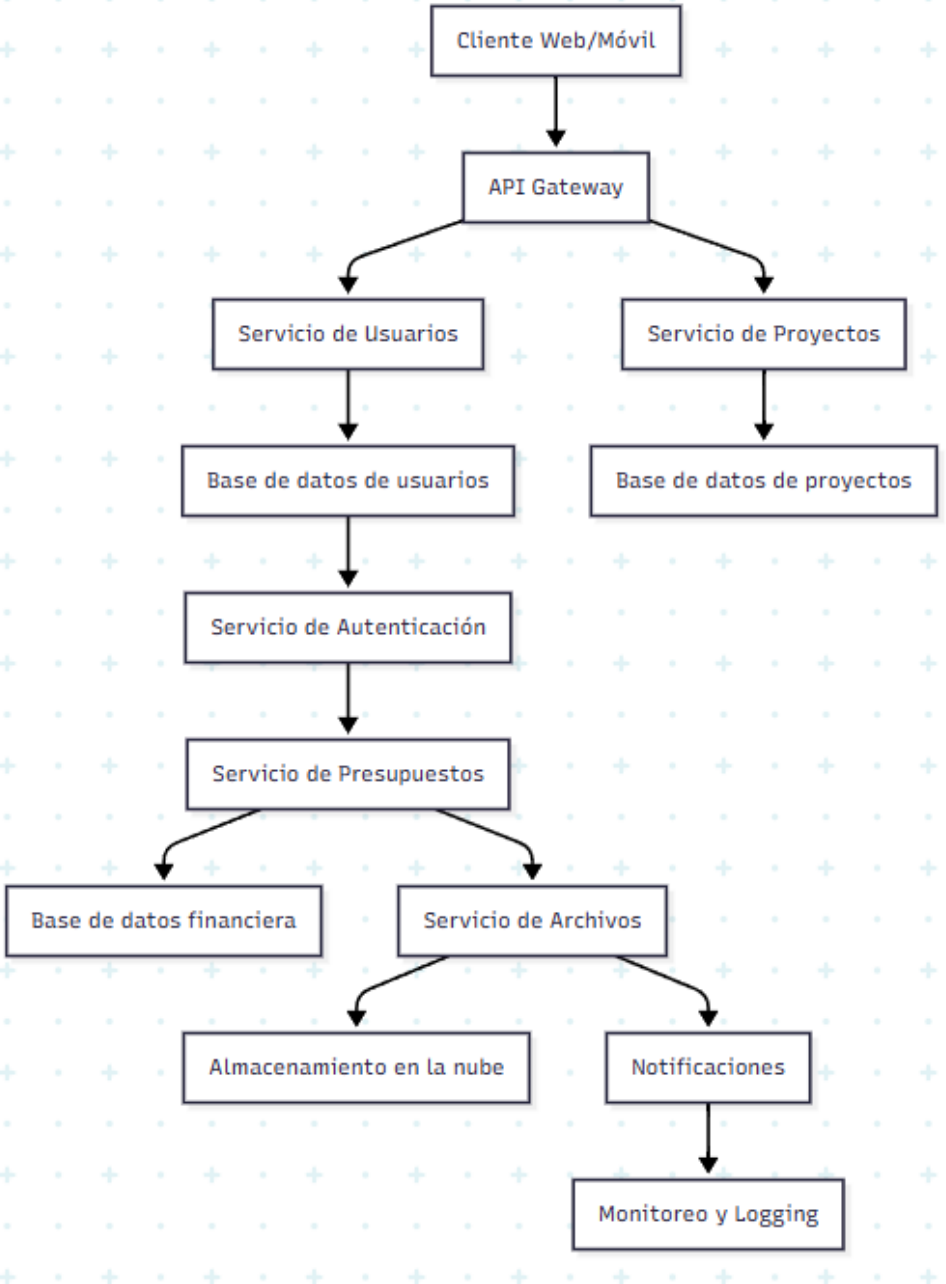
Los registros de contenedores (container registries) son servicios donde se almacenan y gestionan las imágenes de contenedores utilizadas para el despliegue. Existen varias opciones viables para este proyecto:

- **Docker Hub**: es el registro público más conocido, gratuito para proyectos open source, pero con limitaciones en automatización y seguridad en versiones gratuitas.
- **GitHub Packages / GitHub Container Registry (GHCR)**: integrado con GitHub Actions, facilita el versionado y despliegue automatizado desde el repositorio de código.
- **Amazon Elastic Container Registry (AWS ECR)**: altamente recomendado para proyectos desplegados en AWS. Ofrece integración directa con ECS y EKS, control de acceso mediante IAM, y cifrado automático.
- **Google Artifact Registry o Azure Container Registry**: ideales si el entorno se despliega en GCP o Azure, respectivamente.

Para este proyecto, si se despliega en AWS, la opción preferida sería **AWS ECR**, ya que permite un flujo automatizado y seguro entre el pipeline de CI/CD, el almacenamiento de imágenes y el clúster Kubernetes (EKS). En entornos agnósticos o de menor escala, **GitHub Container Registry** es una buena alternativa por su integración con los repositorios de código y flujos CI/CD mediante GitHub Actions.

Diagramas de arquitectura y herramientas utilizadas

Creación de diagrama de arquitectura en Draw.io, Lucidchart o Mermaid.



Representación de componentes, flujos de comunicación y almacenamiento de datos.

Los **componentes** principales de la arquitectura son:

- **Clientes web/móviles**: interactúan mediante HTTPS.
- **API Gateway / Ingress**: actúa como único punto de entrada, aplicando reglas de routing y seguridad.
- **Microservicios**: gestionan funcionalidades específicas (usuarios, proyectos, archivos, presupuestos, autenticación).
- **Bases de datos independientes por servicio** (patrón Database-per-service).
- **Almacenamiento externo** para archivos (S3 o equivalente).
- **Servicios auxiliares**: funciones event-driven, notificaciones, monitoreo, logging, backups.

Los **flujos de comunicación** siguen el patrón de microservicios desacoplados:

- Las peticiones del cliente se enrutan al servicio correspondiente por medio del gateway.
- Cada servicio interactúa con su propia base de datos.
- Se puede utilizar mensajería asincrónica (por ejemplo, RabbitMQ, Kafka o SQS) para desacoplar procesos pesados como generación de reportes o envío de correos.

Los **datos** se almacenan en sistemas especializados:

- **Usuarios y autenticación**: base de datos relacional (PostgreSQL o MySQL) o gestionada (Firebase Auth, Auth0).
- **Proyectos y presupuestos**: base de datos relacional por su estructura transaccional.
- **Archivos**: almacenados en buckets S3 (AWS), Blob Storage (Azure) o equivalente.
- **Logs y métricas**: en bases de datos de series de tiempo (Prometheus) o índices de búsqueda (Elasticsearch).

Definición de las herramientas que se integrarán en la plataforma (bases de datos, APIs, almacenamiento).

Categoría	Herramienta recomendada	Justificación
Contenedores	Docker	Amplia compatibilidad, estándar del sector.
Orquestación	Kubernetes	Escalabilidad, resiliencia y gestión avanzada de microservicios.
API Gateway	NGINX Ingress Controller / Istio Gateway	Control de rutas, seguridad y balanceo de carga.
Base de datos	PostgreSQL / MySQL	Modelo relacional robusto para proyectos y presupuestos.
Almacenamiento	AWS S3 / Azure Blob Storage	Alta disponibilidad y escalabilidad para archivos.
Autenticación	Auth0 / Firebase Auth	Solución externa segura, con autenticación multifactor y gestión de sesiones.
Mensajería	RabbitMQ / AWS SQS	Procesamiento asincrónico y desacople de servicios.
Observabilidad	Prometheus + Grafana / ELK Stack	Monitoreo de métricas, trazas y análisis de logs.
CI/CD	GitHub Actions / GitLab CI	Automatización del proceso de compilación, pruebas y despliegue.
Registro de contenedores	AWS ECR / GitHub Container Registry	Gestión segura y centralizada de imágenes de contenedor.

El desarrollo de los puntos anteriores se puede encontrar en el siguiente repositorio de Github: https://github.com/ojitxslml/Eva5-devops-adalid