1. Metodología Ágil

Para la gestión de este proyecto, se ha seleccionado la metodología ágil **Kanban**. La elección se basa en un análisis de las características del proyecto y la estructura del equipo de trabajo, compuesto por seis desarrolladores y dos operadores.

Justificación

El proyecto consiste en una aplicación de **microservicios** con un **tiempo de vida corto**, lo que exige una metodología que favorezca la velocidad, la eficiencia y la **entrega continua**. Kanban es la opción ideal por las siguientes razones:

1. **Foco en el Flujo de Entrega Continua:** El principio fundamental de Kanban es la optimización del flujo de trabajo. Para un equipo que gestiona múltiples microservicios independientes, este enfoque permite que las tareas (nuevas funcionalidades, correcciones, cambios de infraestructura) se muevan desde el backlog hasta el despliegue de manera constante y sin cuellos de botella. Esto se alinea perfectamente con el objetivo de construir pipelines de CI/CD eficientes.
2. **Flexibilidad para un Proyecto Dinámico:** A diferencia de marcos más rígidos como Scrum, que opera en Sprints de duración fija, Kanban ofrece una mayor flexibilidad. Para un proyecto corto, esta adaptabilidad es crucial. Permite al equipo priorizar y ejecutar tareas a medida que surgen y se completa el trabajo anterior, sin tener que esperar el final de un ciclo artificial.
3. **Gestión Eficiente del Trabajo en Paralelo:** Con seis desarrolladores, es probable que se trabaje en varios microservicios simultáneamente. Un **tablero Kanban** proporciona una visualización clara del estado de cada tarea, permitiendo al equipo gestionar el "Trabajo en Progreso" (WIP) de manera efectiva. Esto previene la sobrecarga y asegura que el equipo se enfoque en finalizar tareas antes de comenzar otras nuevas, acelerando el tiempo total de entrega.

2. Estrategia de Branching para Desarrolladores

Para el equipo de desarrollo, compuesto por seis integrantes, se ha seleccionado la metodología **GitHub Flow**. Esta estrategia se alinea con los principios de la metodología ágil Kanban y está optimizada para la entrega continua en un entorno de microservicios.

Justificación

La elección de GitHub Flow se fundamenta en su **simplicidad y eficiencia**, factores críticos para un proyecto de corta duración y un equipo de tamaño mediano. A diferencia de modelos más complejos como GitFlow, esta estrategia minimiza la sobrecarga administrativa y permite a los desarrolladores enfocarse en la entrega de valor. Facilita el trabajo en paralelo de los seis desarrolladores en diferentes funcionalidades o microservicios, integrando los

cambios de forma continua y segura a través de un flujo de trabajo claro y centrado en Pull Requests.

Flujo de Trabajo Técnico

El ciclo de vida del desarrollo de una nueva funcionalidad o la corrección de un error sigue un proceso estandarizado para garantizar la calidad y la estabilidad del código base.

1. **La Rama main como Fuente de Verdad Desplegable:** La rama main es la única fuente de la verdad y siempre debe contener una versión del código que sea estable, probada y esté lista para ser desplegada en producción.
2. **Desarrollo Aislado en Ramas de Característica:** Todo trabajo nuevo debe realizarse en una rama creada a partir de main. El nombre de la rama debe ser descriptivo para reflejar su propósito.
    - **Ejemplos:** feature/user-password-reset, bugfix/todo-api-delete-error.
3. **Pull Request (PR) como Centro de Calidad:** Una vez que el desarrollo en la rama de característica ha concluido, el desarrollador abre un Pull Request hacia main. El PR actúa como el mecanismo formal para proponer cambios y es el punto de entrada para los procesos de revisión y validación.
4. **Integración Continua (CI) Automatizada:** La apertura del PR dispara automáticamente el pipeline de Integración Continua. Este pipeline es responsable de:
    - Compilar el código del microservicio afectado.
    - Ejecutar las pruebas automatizadas (unitarias, de integración).
    - Construir la imagen de Docker correspondiente y almacenarla en un registro de contenedores.
5. **Revisión por Pares:** Es obligatorio que al menos otro miembro del equipo de desarrollo revise el código en el Pull Request. Este paso fomenta la colaboración, comparte el conocimiento y mejora la calidad general del software.
6. **Fusión y Despliegue Continuo (CD):** Solo después de que el pipeline de CI se complete con éxito y el PR sea aprobado por un par, el código se puede fusionar (merge) a la rama main. Esta acción desencadena el pipeline de Despliegue Continuo (CD), que toma la imagen de Docker recién construida y la despliega automáticamente en el entorno de nube correspondiente.

3. Estrategia de Branching para Operaciones

Para la gestión del ciclo de vida de la infraestructura, se ha seleccionado un enfoque basado en los principios de **GitOps**, utilizando **GitHub Flow** como el modelo de branching. Esta estrategia asegura que cada cambio en la infraestructura sea auditable, revisable y se despliegue de forma automatizada y consistente. En lugar de una herramienta declarativa compleja, la implementación se realizará mediante **scripts de Bash** que interactúan directamente con la **Interfaz de Línea de Comandos (CLI) de Azure**, priorizando la simplicidad y el control directo.

Principios Clave

- **Git como Única Fuente de la Verdad:** El repositorio de Git es el único lugar que define el estado deseado de la infraestructura. Cualquier cambio debe originarse como un commit en el repositorio.
- **Cambios a través de Pull Requests (PRs):** No se permiten cambios directos a la rama principal (main). Todo cambio debe ser propuesto, revisado y aprobado a través de un Pull Request.
- **Automatización Completa:** Los procesos de validación y despliegue de la infraestructura están completamente automatizados a través de pipelines de CI/CD.

Flujo de Trabajo Técnico

El ciclo de vida de un cambio en la infraestructura sigue un flujo de trabajo estructurado para garantizar la seguridad y la coherencia.

1. **Repositorio Dedicado:** Se utiliza un repositorio de Git exclusivo para todo el código de infraestructura. Esto separa las preocupaciones del código de la aplicación del código que define el entorno donde se ejecuta.
2. **Rama main como Reflejo del Entorno Real:** La rama main del repositorio representa el estado actual y aprobado de la infraestructura desplegada en Azure.
3. **Desarrollo en Ramas de Característica (feature branches):** Para proponer cualquier modificación —sea la creación de un nuevo recurso o la actualización de uno existente— el operador crea una nueva rama a partir de main. El nombre de la rama debe ser descriptivo de la tarea a realizar (ej. feature/create-virtual-network o fix/update-container-cpu-limit). En esta rama, el operador creará o modificará los scripts de Bash (.sh) que contienen los comandos az necesarios para aplicar el cambio.
4. **Pull Request y Proceso de Revisión Automatizada (CI):** Una vez que los scripts están listos, el operador abre un Pull Request (PR) hacia la rama main. La creación de este PR dispara automáticamente un pipeline de Integración Continua (CI) que realiza las siguientes validaciones:
    - **Linting de Scripts:** Se utiliza una herramienta como **shellcheck** para analizar estáticamente los scripts de Bash. Este paso previene errores de sintaxis y asegura que los scripts sigoen buenas prácticas antes de su ejecución.
    - **Revisión de Código por Pares:** Dado que los scripts son imperativos (describen una secuencia de acciones), la revisión humana es el control de calidad principal. El equipo debe inspeccionar los comandos az en el PR para confirmar que las acciones son correctas, seguras y cumplen con el objetivo deseado.
5. **Merge y Despliegue Automatizado (CD):** Tras la aprobación del PR y la validación exitosa del pipeline de CI, la rama se fusiona (merge) con main. Esta acción sirve como una aprobación final y dispara automáticamente el pipeline de Despliegue Continuo (CD), el cual ejecuta el script principal (ej. deploy-all.sh) en un entorno

seguro. El script se autentica en Azure y ejecuta los comandos para aplicar los cambios en la infraestructura real.

Claro, aquí tienes la redacción formal de los patrones de diseño seleccionados para tu informe. Este texto explica la justificación y la estrategia de implementación local para cada uno.

4. Patrones de Diseño de Nube Seleccionados

Para el diseño de la arquitectura de la aplicación, se han seleccionado e implementado tres patrones de diseño de nube fundamentales: **API Gateway**, **Cache-Aside (Respaldo de Caché)** y **Autoscaling (Autoescalado)**. Aunque el despliegue final se realiza en un entorno local simulado con Docker Compose, la implementación de estos patrones sigue las mejores prácticas de la industria y demuestra la preparación de la arquitectura para un eventual despliegue en un entorno de nube productivo.

4.1 Patrón de API Gateway

Justificación

En una arquitectura de microservicios, el cliente (en este caso, el **Frontend**) necesita interactuar con múltiples servicios backend independientes (Auth API, Users API, TODOs API). Exponer cada microservicio directamente al cliente presenta varios desafíos, como la complejidad en el código del frontend, problemas de seguridad y acoplamiento directo.

El patrón **API Gateway** resuelve estos problemas al actuar como un único punto de entrada (single entry point) para todas las solicitudes del cliente. Se ha implementado utilizando **Nginx** como un proxy inverso. Todas las peticiones del frontend se dirigen a Nginx, que se encarga de enrutarlas al microservicio interno correspondiente basándose en la ruta de la URL.

Los beneficios directos de este patrón son:

- **Simplificación del Cliente:** El frontend solo necesita conocer una única URL base (la del gateway), desacoplando su lógica del conocimiento de la ubicación de cada microservicio.
- **Centralización de Aspectos Transversales:** El gateway es el lugar ideal para centralizar la autenticación, el registro (logging) y el enrutamiento, liberando a los microservicios de estas responsabilidades.
- **Seguridad Mejorada:** Al no exponer los servicios internos directamente, se reduce la superficie de ataque de la aplicación.

Implementación Local

La simulación se logra añadiendo un servicio nginx a la configuración de docker-compose.yml. Un archivo nginx.conf define las reglas de proxy_pass que redirigen las peticiones (ej. /api/users/*) al nombre de servicio del contenedor correspondiente (ej. http://users-api:8082).

4.2 Patrón de Respaldo de Caché (Cache-Aside)

Justificación

El servicio **Users API** gestiona datos de perfiles de usuario, que son leídos con alta frecuencia pero modificados con poca regularidad. Depender exclusivamente de la base de datos para cada solicitud de lectura puede convertirla en un cuello de botella, afectando el rendimiento general.

El patrón **Cache-Aside** se implementa para mitigar esta carga. Se introduce una capa de caché de alta velocidad, en este caso, un contenedor de **Redis**, para almacenar temporalmente los datos de los usuarios. Cuando se solicita un perfil, la aplicación primero consulta la caché. Si los datos existen (**cache hit**), se devuelven instantáneamente. Si no existen (**cache miss**), la aplicación consulta la base de datos, almacena el resultado en la caché para futuras peticiones y luego devuelve la información.

Esta estrategia optimiza drásticamente el rendimiento al:

- **Reducir la latencia:** Las respuestas para datos cacheados son órdenes de magnitud más rápidas.
- **Disminuir la Carga en la Base de Datos:** Se minimiza el número de consultas de lectura, liberando recursos para operaciones de escritura.

Implementación Local

Se añade un servicio Redis al archivo docker-compose.yml. El Users API se configura con variables de entorno para conectarse al servicio redis-cache a través de la red interna de Docker. La lógica del patrón reside completamente en el código de la aplicación, haciendo la simulación idéntica a un despliegue en la nube.

4.3 Patrón de Autoescalado (Autoscaling)

Justificación

La carga de trabajo de una aplicación web es inherentemente variable. El patrón de autoescalado permite que la arquitectura se adapte dinámicamente a la demanda, asegurando tanto la disponibilidad del servicio como la eficiencia de los recursos.

Los servicios de API (TODOs API, Users API) son los principales candidatos para el autoescalado. Este patrón permite escalar horizontalmente (añadir más instancias o réplicas del servicio) durante picos de demanda para distribuir la carga y mantener un rendimiento óptimo. En períodos de baja actividad, permite escalar hacia adentro (reducir el número de instancias) para no desperdiciar recursos.

Los beneficios clave son:

- **Alta Disponibilidad y Resiliencia:** El sistema puede soportar picos de tráfico inesperados sin degradar el servicio.
- **Eficiencia de Recursos:** Se utilizan solo los recursos necesarios en cada momento, lo que en un entorno de nube se traduce en una optimización directa de costos.

Implementación Local

Dado que Docker Compose no soporta el autoescalado automático basado en métricas, el concepto se demuestra a través de una **simulación manual**. La arquitectura se prepara para la escalabilidad y se comprueba su funcionamiento:

1. **Escalado Manual:** Se utiliza el comando docker-compose up --scale <servicio>=N para aumentar manualmente el número de réplicas de un microservicio (ej. todos-api=3).
2. **Balanceo de Carga:** El **API Gateway (Nginx)**, al redirigir el tráfico al nombre del servicio escalado, utiliza el balanceador de carga integrado de Docker para distribuir las peticiones de manera equitativa (round-robin) entre todas las réplicas disponibles.

Esta simulación prueba que la arquitectura está correctamente diseñada para escalar horizontalmente y que el balanceo de carga funciona como se espera, validando la implementación del patrón.

Claro, aquí tienes un resumen de ambas estrategias de pipeline, redactado en tercera persona y en un tono formal, listo para ser añadido a tu informe.

5. Estrategia de Pipelines de Automatización

Para garantizar la calidad, velocidad y fiabilidad en la entrega del software, el proyecto implementa una estrategia de automatización robusta que separa la **Integración Continua (CI)** de la **Entrega Continua (CD)**. Esta separación se articula a través de dos tipos de pipelines especializados: un pipeline de CI por cada microservicio y un pipeline de CD centralizado para la integración.

Estrategia de Pipeline de Integración Continua (CI)

La estrategia de Integración Continua se enfoca en la autonomía y la validación individual de cada microservicio. El objetivo principal de este pipeline es asegurar que cualquier cambio nuevo en el código de un servicio específico resulte en un artefacto desplegable, probado y versionado, sin afectar el ciclo de vida de los otros servicios.

El flujo de trabajo técnico es el siguiente:

1. **Activación por Cambio:** El pipeline se activa automáticamente cuando un desarrollador fusiona cambios en la rama principal (main) de un microservicio.
2. **Compilación y Pruebas Unitarias:** El pipeline compila el código fuente del servicio afectado y ejecuta su conjunto de pruebas automatizadas para validar la lógica interna.
3. **Empaquetado como Contenedor:** Una vez que las pruebas pasan, el servicio se empaqueta en una imagen de Docker. Esta imagen constituye el artefacto final de la fase de CI, asegurando que el microservicio sea portable y esté listo para su despliegue en cualquier entorno.
4. **Versionado y Publicación:** La imagen de Docker se etiqueta con un identificador único y se publica en un registro de contenedores centralizado.

Al finalizar este pipeline con éxito, se tiene la certeza de que la nueva versión del microservicio es funcional de manera aislada y está lista para ser validada en el contexto del ecosistema completo.

Estrategia de Pipeline de Entrega Continua (CD)

El pipeline de Entrega Continua actúa como la fase final de validación, asegurando que la nueva versión de un microservicio no solo funciona por sí misma, sino que también se integra y colabora correctamente con el resto de los servicios del sistema.

El flujo de trabajo es el siguiente:

1. **Activación Post-CI:** Este pipeline es disparado automáticamente solo después de que el pipeline de CI de un microservicio se haya completado exitosamente en la rama principal.
2. **Aprovisionamiento del Entorno:** El pipeline utiliza Docker Compose para levantar un entorno de pruebas completo y efímero. Este entorno replica la arquitectura de producción, desplegando la nueva versión del microservicio junto con las últimas versiones estables de todos los demás servicios.
3. **Ejecución de Pruebas de Integración:** Se ejecuta un conjunto de pruebas automatizadas que simulan flujos de usuario reales a través del sistema. Por ejemplo, se prueba si un usuario puede autenticarse a través del **API Gateway**, obtener un token del Auth API y usar ese token para acceder a recursos protegidos en el TODOs API.
4. **Validación y Limpieza:** Si todas las pruebas de integración pasan, el pipeline se considera exitoso, validando que la nueva versión es segura para ser desplegada. Independientemente del resultado, el pipeline concluye destruyendo el entorno de pruebas para liberar recursos.