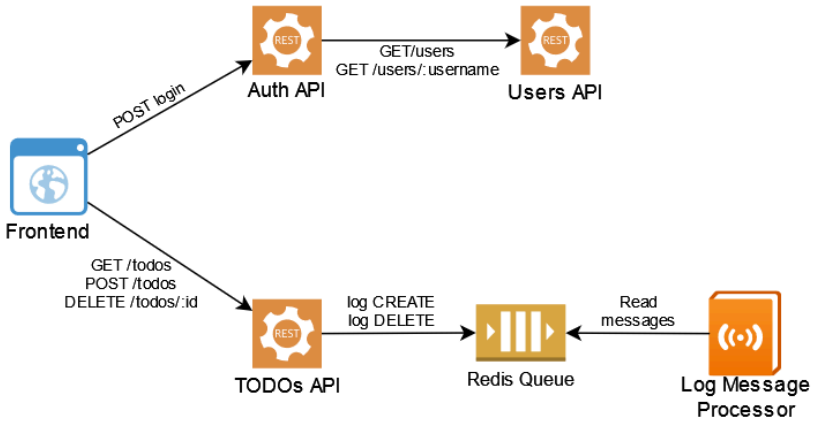
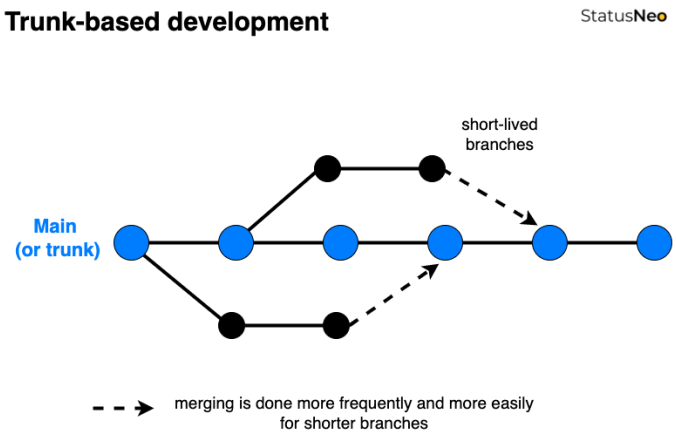
1. Estrategia de Branching para Desarrolladores



Tomando en cuenta que la organización actual del repositorio consiste en un "Monorepo", en cuanto cada una de las carpetas representa uno de los microservicios plasmada en el diagrama arquitectónico del enunciado. Y, también considerando que las branching strategies deberán basarse en los principios fundamentales de la cultura DevOps, en cuanto se busca eliminar o reducir las barreras entre los equipos de operaciones y desarrollo, además de reducir los tiempos en que un feature nueva se encuentra desplegada en producción, se optó por escojer la estrategia de branching conocida como *Trunk Based Branching (TBD): "Microsoft utiliza una estrategia de ramificación basada en troncos para ayudar a desarrollar productos rápidamente, implementarlos con regularidad y aplicar los cambios de forma segura a la producción."* (AWS, n.d.)



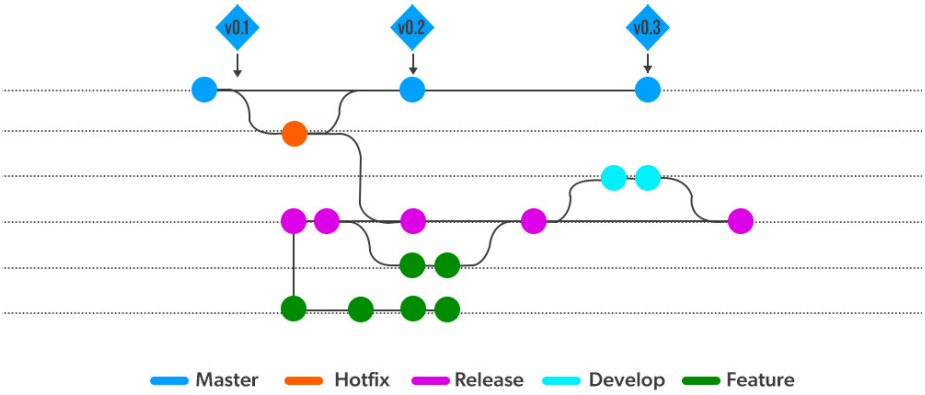
Fuente: Beginners Guide to Trunk-Based Development (TBD) - StatusNeo

Como se mencionó con anterioridad, las estrategias de branching para los equipos de desarrollo deben facilitar la integración continua y entrega continua, además de trabajar bajo el supuesto de que la main branch deberá estar siempre "buildable", por lo que las razones para usar dicha estrategia son:

Razón	Argumento
Eliminación de Merge Debt y Conflictos de Integración	En un monorepo con microservicios organizados por carpetas, TBD reduce los conflictos de merge que pueden surgir cuando múltiples equipos trabajan simultáneamente en diferentes servicios, manteniendo la sincronización entre componentes interdependientes.
Detección Temprana de Errores y Shift Left	El flujo de pull request proporciona un punto común para hacer cumplir las pruebas, revisión de código y detección de errores temprano en el pipeline. Esta estrategia de desplazamiento hacia la izquierda ayuda a acortar el ciclo de retroalimentación a los desarrolladores reduciendo el tiempo de detección de errores. En el caso de microservicios se identifican inmediatamente durante la integración.
Frecuencia de Integración y Testing Continuo	TBD facilita que múltiples desarrolladores integren cambios frecuentemente al trunk principal, lo que resulta en ciclos de feedback rápidos y detección temprana de problemas de integración. En el caso de un monorepo con microservicios, esto significa que cada cambio en cualquier servicio se valida inmediatamente contra el estado actual de todos los demás servicios, manteniendo la compatibilidad del sistema completo y permitiendo que los equipos detecten dependencias rotas o efectos colaterales antes de que lleguen a producción.
Despliegues Independientes por Microservicio	Los equipos individuales deben poder liberar servicios de forma rápida y confiable, sin interrumpir a otros equipos o desestabilizar la aplicación como un todo. TBD facilita que cada microservicio mantenga su ciclo de release independiente desde un trunk común estable.

2. Estrategia de Branching para Operaciones

Para el equipo de operaciones, enfocado en Infraestructura como Código (IaC) con Terraform en Azure (e.g., AKS, Redis), se optó por GitHub Flow adaptado tras analizar alternativas en guías de Microsoft y AWS, priorizando estabilidad y control en cambios menos frecuentes que la infraestructura requiere. Junto a lo anterior, se tomó en cuenta el tipo de branching strategy que se había seleccionado para los desarrolladores, toda vez que al ya no ser silos de trabajo independientes, se buscó que una estrategia no perjudique a la otra.



Fuente: Git Flow vs Github Flow - GeeksforGeeks

Tomando en cuenta que la infraestructura del proyecto sería gestionada bajo el principio de "Infrastructure as Code", se necesitaba entonces una estrategia que permitiese una buena trazabilidad, auditabilidad, y versatilidad al momento de gestionar cambios de infraestructura; que de hacerse, se efectuarán bajo pipelines automatizados. Por lo que se optó por la estrategia de GitHub Flow, adaptado a los procesos del equipo de operaciones en cuanto se mantiene un main branch estable que refleja el estado actual de producción de toda la infraestructura, feature branches a para cambios específicos como actualizaciones de configuración o posibles políticas de autoscaling en los recursos, y Pull Requests con validación automatizada que ejecutan terraform plan y terraform validate antes del merge. Dicha estrategia traería las siguientes ventajas sobre las otras:

Basándome en las fuentes oficiales consultadas, aquí están las 4 principales ventajas de GitHub Flow adaptado para el equipo de operaciones:

Tomando en cuenta que la infraestructura del proyecto sería gestionada bajo el principio de "Infrastructure as Code", se necesitaba entonces una estrategia que permitiese una buena trazabilidad, auditabilidad, y versatilidad al momento de gestionar cambios de infraestructura; que de hacerse, se efectuarán bajo pipelines automatizados. Por lo que se optó por la estrategia de GitHub Flow, adaptado a los procesos del equipo de operaciones en cuanto se mantiene un main branch estable que refleja el estado actual de producción de toda la infraestructura, feature branches para cambios específicos como actualizaciones de configuración o posibles políticas de autoscaling en los recursos, y Pull Requests con validación automatizada que ejecutan terraform plan y terraform validate antes del merge. Dicha estrategia traería las siguientes ventajas sobre las otras:

Razón	Argumento
Prevención de Bloqueos de Promoción	Evita que los cambios de infraestructura se bloqueen mutuamente cuando utilizan branches dedicados por entorno. Si un cambio de configuración de Redis no está listo para producción, esto no impide que una actualización de políticas de autoscaling se despliegue independientemente, manteniendo el flujo continuo de mejoras de infraestructura.
Separación Lógica de Entornos sin Complejidad	Organiza la infraestructura mediante estructura de carpetas por entorno en lugar de branches separados, manteniendo todos los entornos sincronizados en un solo repositorio. Esto evita que el código de desarrollo, staging y producción diverja con el tiempo, simplificando la gestión y reduciendo errores de configuración entre entornos.
Pipeline de Bajo Contacto Alineado con Naturaleza de IaC	Reconoce que los recursos de infraestructura como redes, bases de datos y servicios de cache cambian con menor frecuencia que el código de aplicación. La estrategia proporciona un nivel de control apropiado para estos cambios menos frecuentes pero más críticos, sin imponer la complejidad de múltiples branches de larga duración que serían subutilizados.
Trazabilidad y Control Mejorados	Aplica los mismos estándares de calidad y control de versiones que el código de aplicación, pero con controles adicionales específicos para infraestructura crítica. Cada cambio queda documentado en Pull Requests con validaciones automatizadas de Terraform y aprobaciones manuales, creando un registro completo de quién, cuándo y por qué se modificó cada recurso.

3. Patrones de Diseño de Nube

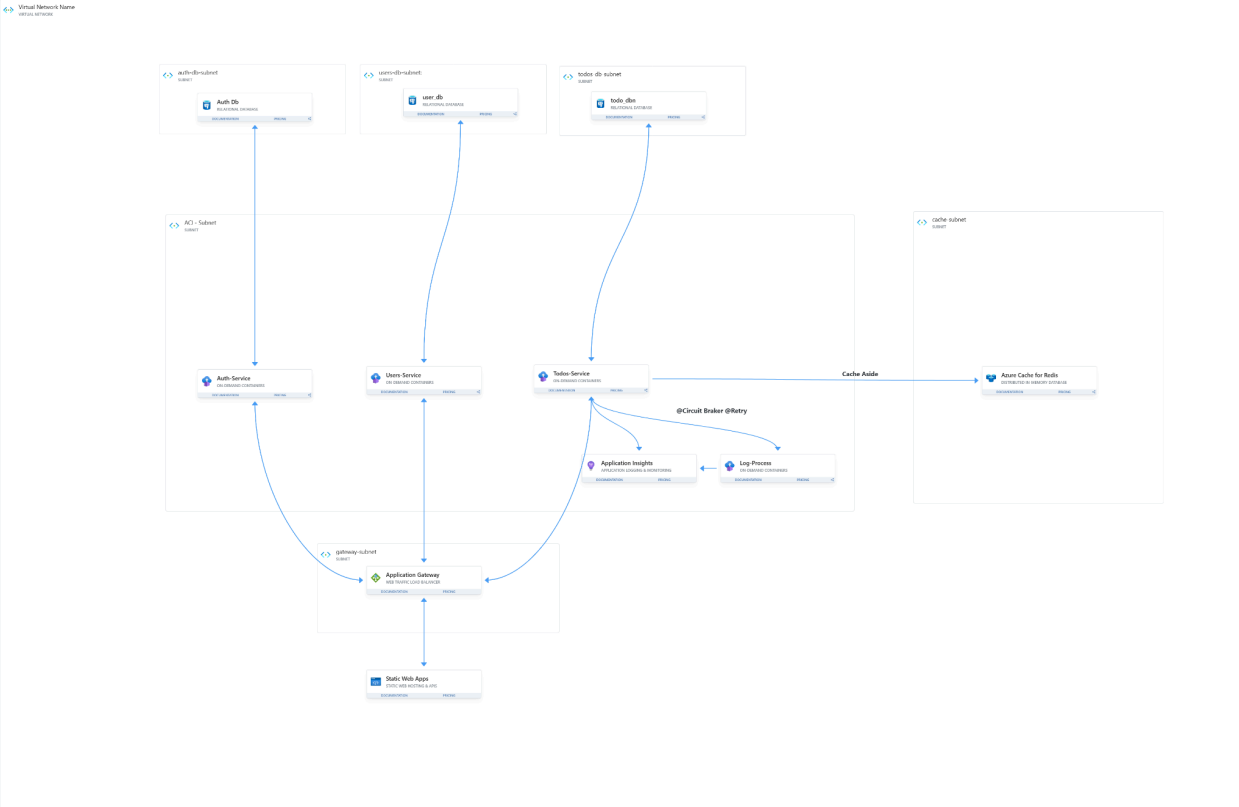
Basándose en que el estado actual del repositorio supone que existe una alta acoplación entre los servicios, ya que hay dependencias donde el TODOs API depende directamente del Users API para validar que un usuario tenga permisos para realizar operaciones CRUD sobre sus tareas, y el Frontend requiere constante comunicación con el Auth API para mantener tokens JWT válidos, se identificaron varios riesgos operacionales que requerían mitigación: El primer criterio fue gestionar fallos en cascada, considerando que si el Users API (Spring Boot) experimenta problemas, todas las operaciones de TODOs quedarían bloqueadas, afectando la experiencia completa del usuario. El segundo criterio fue manejar la heterogeneidad de respuesta, ya que la arquitectura polyglot presenta diferentes patrones de comportamiento: el Auth API en Go puede responder en microsegundos, mientras que el Users API en Spring Boot puede tomar más tiempo durante picos de carga, creando posibles inconsistencias en la experiencia de usuario. El tercer criterio fue optimizar recursos de infraestructura, aprovechando que Redis ya procesa logs de operaciones de TODOs a través del Log Message

Processor en Python, representando una oportunidad de reutilización sin añadir complejidad arquitectónica adicional.

Los patrones implementados abordan cada uno de estos criterios de manera complementaria. El Circuit Breaker rompe la dependencia rígida entre TODOs API y Users API mediante la implementación de un mecanismo de protección que, tras detectar fallos consecutivos, temporalmente evita llamadas al servicio problemático y retorna respuestas predeterminadas, permitiendo que el sistema continúe funcionando parcialmente en lugar de fallar completamente. El patrón Retry maneja la variabilidad de respuesta entre los diferentes stacks tecnológicos, permitiendo que el sistema reintente operaciones fallidas hasta tres veces antes de activar el Circuit Breaker, lo cual es particularmente útil para las operaciones de autenticación donde las diferencias de latencia entre Go y los otros servicios pueden causar timeouts ocasionales. El Redis Cache (Cache-Aside) reduce el acoplamiento operacional al almacenar temporalmente perfiles de usuario frecuentemente consultados, disminuyendo las llamadas directas del TODOs API al Users API de cientos por minuto a solo unas pocas por hora para datos actualizados, mientras aprovecha la infraestructura Redis existente que ya maneja la cola de mensajes de logging, creando un sistema más resiliente y eficiente sin incrementar la complejidad de despliegue.

4.Diagramas de Arquitectura.

La arquitectura implementada presenta un diseño de microservicios polyglot desplegado en Azure Cloud dentro de una Virtual Network que proporciona aislamiento y seguridad de red. La solución está compuesta por tres subnets especializadas: auth-db-subnet que aloja el Auth Service desarrollado en Go, users-db-subnet que contiene el Users Service implementado en Spring Boot, y todos-db-subnet que hospeda el TODOs Service construido en NodeJS. Todos los servicios backend se comunican a través de un Application Gateway que actúa como punto de entrada unificado y balanceador de carga, mientras que el Frontend se despliega como una Static Web App independiente para servir la interfaz Vue.js. La infraestructura incluye Azure Cache for Redis que cumple una doble función: implementar el patrón Cache Aside para optimizar consultas de usuario y servir como cola de mensajes para el Log Processor desarrollado en Python que maneja los eventos de auditoría. Los patrones de resiliencia Circuit Breaker y Retry están integrados en las comunicaciones entre servicios para garantizar la estabilidad del sistema..



El flujo operacional inicia cuando un usuario accede a la Static Web App que sirve el Frontend Vue.js, la cual realiza peticiones HTTP al Application Gateway que enruta las solicitudes de autenticación hacia el Auth Service en la subnet correspondiente, protegido por el patrón Circuit Breaker con reintentos automáticos para manejar fallos transitorios. Una vez autenticado con token JWT válido, las operaciones CRUD de TODOs son dirigidas por el Gateway hacia el TODOs Service, el cual debe validar ownership consultando el Users Service a través de comunicación inter-subnet. Esta validación crítica está optimizada mediante el patrón Cache Aside en Azure Cache for Redis: primero se consulta el caché para obtener datos del usuario, y solo si no están disponibles se realiza la consulta al Users Service, almacenando el resultado para futuras consultas y reduciendo significativamente la latencia entre subnets. Simultáneamente, cada operación de creación y eliminación de TODOs genera eventos de auditoría que se almacenan en la cola Redis, donde el Log Processor los consume asincrónicamente desde su ubicación en Application Insights para procesamiento y registro. Esta arquitectura de red segmentada permite que cada servicio opere en su entorno aislado mientras mantiene comunicación eficiente a través del Gateway.

Referencias

AWS. (n.d.). *Branching strategies for IaC*. AWS Prescriptive Guidance. https://docs.aws.amazon.com/prescriptive-guidance/latest/designing-a-devsecops-mechanism/branching-strategies.html

Microsoft. (2024). *How Microsoft develops with DevOps*. Azure DevOps Documentation. https://learn.microsoft.com/en-us/devops/develop/how-microsoft-develops-devops

Microsoft. (2024). *Microservices CI/CD pipeline on Kubernetes with Azure DevOps and Helm*. Azure Architecture Center. https://learn.microsoft.com/en-us/azure/architecture/microservices/ci-cd-kubernetes

Microsoft. (2024). *Recommendations for using infrastructure as code*. Microsoft Azure Well-Architected Framework. https://learn.microsoft.com/en-us/azure/well-architected/operational-excellence/infrastructure-as-code-design