TECH CHALLENGE II

Documento de Especificação

Projeto: ToggleMaster (Microsserviços)
Fase 2: Transição do ToggleMaster de um monólito para microsserviços
Curso: DevOps e Arquitetura Cloud (AWS)
Professor(a):
Ano: 2026

Integrantes:
- Antonio Eduardo Silveira Demarchi
- Leonardo Alves Freitas

Projeto:
https://github.com/DemarchiWorking/toggle-master-microservice-kubernetes

Vídeo Apresentação: https://youtu.be/P505KNzWTBA
GitHub: https://github.com/FIAP-TCs

Sumário

Toggle Master

ESPECIFICAÇÃO DO PROJETO	
Nome:	**Toggle Master**
Área:	**Infraestrutura de Software e Engenharia de Plataforma** (DevOps/SRE/Infra/Cloud/Desenvolvimento)
Github:	https://github.com/FIAP-TCs
Projeto:	https://github.com/freitasleoalves/fiap-tech-challenge-2-v1
Vídeo:	https://youtu.be/P505KNzWTBA

CONTEXTO E PROBLEMA	
Problema:	Risco do Deploy Acoplado ao Lançamento (Inflexibilidade no Lançamento, Alto Tempo de Recuperação, Gargalo entre Produto e Engenharia)
Contexto:	Gerenciar Flags em Larga Escala e Alta Disponibilidade: Implementação de novas funcionalidades atrás de feature flags, para determinados usuários testarem e o pros demais não conseguirem enxergar essa nova funcionalidade até toda equipe ter chance de testar e aprovar).

ESCOPO DO PROJETO

Introdução:
O ToggleMaster evoluiu de um MVP monolítico para uma plataforma distribuída de gerenciamento de Feature Flags. Após o sucesso da fase de validação, a arquitetura foi reestruturada para suportar alta demanda, baixa latência e resiliência global.
Este documento descreve a Fase 2, onde a aplicação foi decomposta em 5 microsserviços especializados, conteinerizada com Docker e orquestrada via Azure Kubernetes Service (AKS).
Permitindo que equipes lancem novas funcionalidades de forma controlada, alternando funcionalidades e garantindo que o caminho (Hot Path) seja independente da gestão administrativa das flags.
Visão Geral da Aplicação:
A plataforma abandonou o modelo de unidade única para adotar uma arquitetura de sistemas distribuídos (cada serviço utiliza o banco de dados

mais eficiente para sua tarefa).
Objetivo:
Permitir o controle de funcionalidades em tempo real (Feature Toggles) para usuários específicos, eliminando o risco e a demora de novos deploys em produção, garantindo a agilidade do time de produto.

Evoluindo a plataforma ToggleMaster de um modelo monolítico para uma arquitetura de microsserviços nativa em nuvem (Cloud Native), utilizando a Azure para garantir que o sistema seja capaz de suportar um crescimento exponencial de acessos através de orquestração com Kubernetes (AKS), persistência e escalonamento.

Microsserviços Distribuídos: Desacoplamento de lógica de autenticação, CRUD, segmentação, avaliação e telemetria.

Tecnologias Core: Go (Alta performance), Python (Agilidade), Redis (Cache em memória), PostgreSQL (Relacional), Azure Cosmos DB (NoSQL) e Azure Service Bus (Mensageria) e Terraform (Infrastructure as Code).

Orquestração: Kubernetes (AKS) com escalonamento.

Situação:
Rodar a aplicação localmente para entender seu funcionamento e, em seguida, provisionará a infraestrutura necessária na nuvem para publicá-la, seguindo as melhores práticas iniciais de segurança e arquitetura.

Justificativa:
Necessidade de criar, ler, atualizar e deletar feature flags.
A necessidade transcende o CRUD básico, nesta fase, a justificativa para a separação em microsserviços é garantir a integridade e o isolamento de domínios. Cada serviço (Auth, Flag, Targeting) possui seu próprio ciclo de vida de dados no Azure Database for PostgreSQL, seguindo o princípio de menor Privilégio.

Conectividade: Foi necessário configurar tabelas de rotas específicas para garantir que a EC2 na rede pública pudesse rotear tráfego para a sub-rede privada. Diferente da estrutura simples da Fase 1, a conectividade agora utiliza uma topologia com múltiplas subnets e service endpoints para garantir que apenas o cluster AKS tenha permissão de rota para as subnets privadas de dados (PostgreSQL e Redis). A comunicação entre o Evaluation Service e o Analytics Service foi justificada pelo uso de Mensageria (Service Bus), garantindo conectividade assíncrona e resiliência em picos de carga.

Público-Alvo:

- **Engenheiros de Software e DevOps**: Que buscam realizar deploys seguros.
- **Product Managers (PMs):** Que necessitam de autonomia para ativar funcionalidades para grupos específicos de usuários (segmentação) sem depender de janelas de manutenção.
- **Times de Operações (SRE):** Que utilizam as flags como para desativar funções instáveis instantaneamente, preservando a estabilidade do ecossistema.

Riscos para o Negócio:

- Indisponibilidade Sistêmica
- Latência no Hot Path
- Inconsistência de Lançamento
- Perda de Dados de Telemetria

Nomenclatura:

RF=Requisito Funcional
RNF=Requisito Não Funcional

VANTAGENS

De Monolito a Microsserviços (Granularidade):

Valida a operação em larga escala, o sistema não apenas funciona, mas que sobrevive a picos de tráfego, otimizando custos.

No modelo de microsserviços, cada função é independente. Se um microserviço (Ex:evaluation-service) estiver sobrecarregado, você escala apenas os necessários, eliminando a carga operacional de gerenciar máquinas virtuais.

Suportando picos de tráfego no serviço e processando filas de sem intervenção manual e otimizando o custo e a performance, pois cada serviço usa a linguagem e o banco de dados mais adequados para sua função.

O Kubernetes introduz o Self-healing (Auto-cura). Se um contêiner (Ex:auth-service) falhar, a ferramenta percebe e sobe um novo instantaneamente.

Permite o Event-driven Scaling no caso o analytics-service pode estar com 0 pods (custo zero) e, no momento em que uma mensagem cai na fila, o serviço acorda a aplicação para processar.

O Kubernetes distribui os pods de forma a extrair o máximo de valor de cada centavo gasto com a infraestrutura de máquinas.

PRÉ-REQUISITOS E FERRAMENTAS (LOCAL)		
Docker Desktop	24.x	Containers e build de imagens
Docker Compose	v2.x	Orquestração local dos 9 containers
curl	qualquer	Testes manuais dos endpoints

CORREÇÕES NECESSÁRIAS	
Go	Comentar ou remover essa linha do “go.mod”: > //github.com/jackc/pgx/v4/stdlib v4.18.3 // indirect Comentar ou remover essas linha do “handlers.go” > // "crypto/sha256" > // "encoding/hex" Comentar ou remover essas linha do “key.go” e do “main.go” >// "fmt" _ "github.com/jackc/pgx/v4/stdlib" // blank import Rodar esse commando: > go mod tidy
Python	Nos arquivos requirements.txt, adicionar > Werkzeug==2.2.2 Flask==2.2.2 Werkzeug==2.2.2 psycopg2-binary==2.9.5 gunicorn==20.1.0 python-dotenv==0.21.0 requests==2.28.1

STACK DE TECNOLOGIAS	
Go (Golang)	Utilizado nos serviços auth-service e evaluation-service pela sua alta performance, baixo consumo de memória e suporte nativo à concorrência. **Net/HTTP:** Pacote nativo do Go para criação de servidores HTTP de alta performance.
Python	Utilizado nos serviços flag-service, targeting-service e analytics-service pela flexibilidade e ricas bibliotecas de integração (Flask, Boto3, Psycopg2). **Flask:** Micro-framework web para as APIs em Python.
Docker	Criação de imagens utilizando builds para otimização de tamanho e segurança. **Docker Compose:** Orquestração local de 9 contêineres para ambiente de desenvolvimento e testes.
Kubernetes (AKS - Azure Kubern etes Service)	Orquestrador de contêineres gerenciado para produção. **Kubctl**: Ferramentas de linha de comando para gestão da nuvem e do cluster. **Lens**: Agiliza a administração de kubernetes.
Azure Database for PostgreSQL	Banco de dados relacional gerenciado para armazenamento de flags, chaves de API (hashes) e regras de segmentação.
Redis	Armazenamento em memória para o Hot Path de avaliação, garantindo latência reduzida.
Azure Cosmos DB (API NoSQL):	Banco de dados de documentos para persistência de grandes volumes de eventos de telemetria e analytics.
Azure Service Bus (Queues):	Fila de mensagens assíncronas para desacoplar a avaliação da flag (escrita rápida) do processamento de analytics (persistência posterior).
HPA (Horizontal Pod Autoscaler):	Escalabilidade baseada em métricas de CPU/Memória para os demais serviços.
Nginx Ingress Controller:	Ponto de entrada único para o cluster, gerenciando roteamento por path
Azure Container Registry (ACR)	Repositório privado e seguro para armazenamento das imagens Docker.
Terraform	Provisiona a infraestrutura na Azure.

DOCKER

Introdução e Análise de Conteinerização

O projeto faz parte de uma evolução de MVP monolítico para uma arquitetura de microsserviços distribuídos, visando escalabilidade e resiliência.

Análise e Conteinerização (Docker)

Estratégia de Dockerfile: Para todos os 5 microsserviços foram construídas o arquivo Dockerfile especificando as configurações necessárias para garantir que você consiga executar e entender todo o ecossistema localmente.

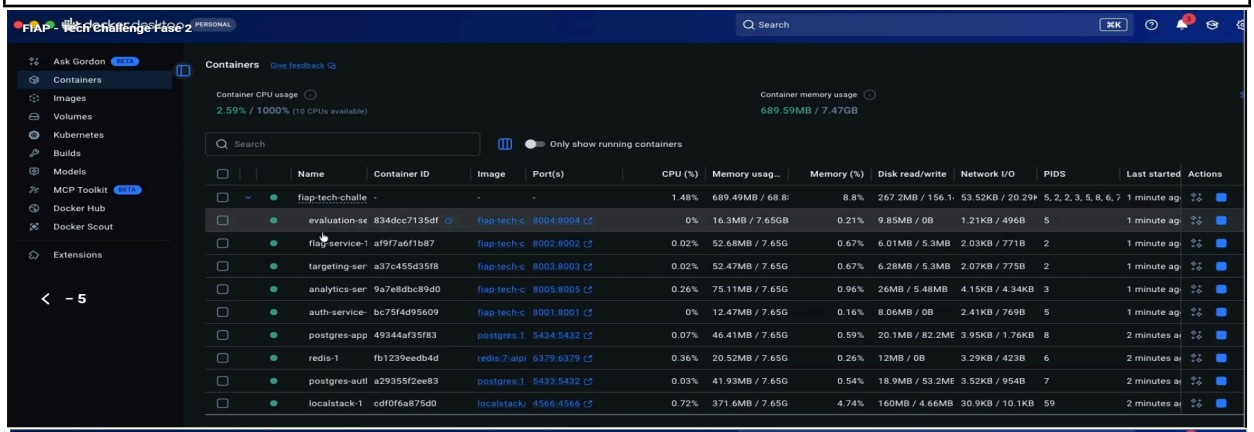
Ambiente Local (Docker Compose)

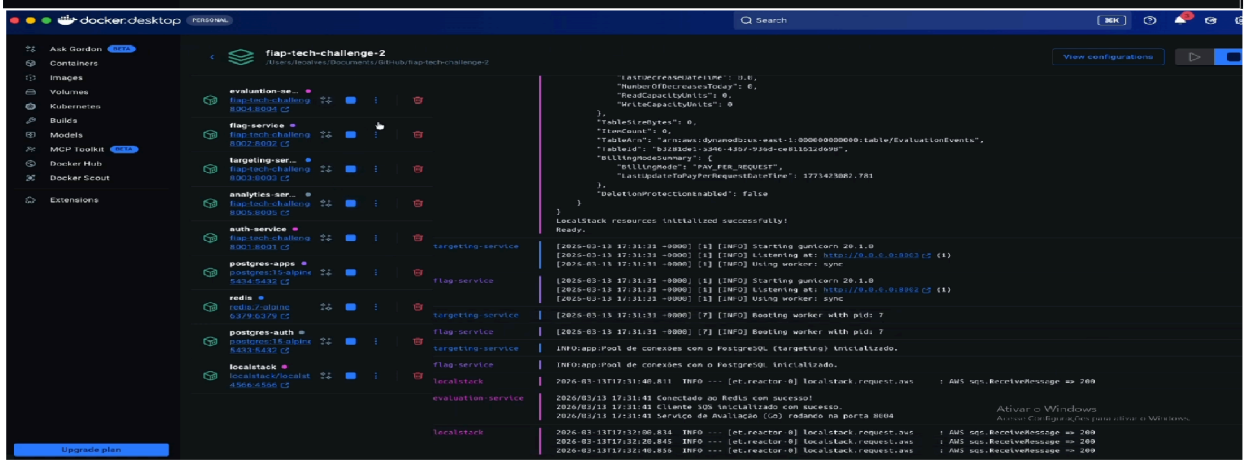
O arquivo docker-compose.yml orquestra diversos contêineres:

Apps: auth, flag, targeting, evaluation, analytics.

Infra Local: PostgreSQL, Redis, DynamoDB Local.

Rede: Criada uma network dedicada para comunicação interna do Docker.





FUNCIONALIDADES PRINCIPAIS	
Auth-Service (Go) Segurança, gerenciando chaves de API e autenticação. (Banco de Dados: PostgreSQL)	- Gestão de API Keys: Geração de chaves(256 bits) - Armazenamento Seguro (hashing) - Acesso: Validar se a chave enviada está ativa. - Administração Restrita: Endpoint de criação de chaves
Flag-Service (Python/Flask) CRUD das definições das feature flags. (Banco de Dados: PostgreSQL)	- Catálogo de Feature Flags: Operações de CRUD completo (Criar, Ler, Atualizar, Deletar) para as definições de flags. - Interruptor de Emergência: Capacidade de habilitar ou desabilitar globalmente uma funcionalidade - Consumo de Autenticação: Validar permissões antes de qualquer operação.
Targeting-Service (Python/Flask) Gerencia regras de segmentação. (Banco de Dados: PostgreSQL)	- Segmentação Dinâmica:Permite criar regras de público-alvo vinculadas a uma flag específica - Persistência de Regras JSON:Armazenamento de regras, permitindo evoluir a lógica de segmentação sem alterar o schema
Evaluation-Service (Go) O caminho quente (hot path) de alta performance que retorna a decisão final (true/false). (Cache: Redis)	- Orquestração de Decisão: Centraliza a lógica que decide se o usuário verá ou não. - Cache de Alta Performance: Utilização de Cache para evitar chamadas repetitivas aos bancos de dados. - Processamento Concorrente: Busca informações simultaneamente no flag-service e targeting-service para otimizar o tempo de resposta.
Analytics-Service (Python/Boto3) Consome eventos de uma fila e salva dados de análise.	- Processamento Assíncrono: Atua como um Worker que processa mensagens de telemetria em segundo plano, sem travar a experiência do usuário. - Carregamento de Dados NoSQL: Consome eventos de filas e os persiste.

REQUISITOS FUNCIONAIS (RF)	
RF01	O sistema deve permitir a criação de API Keys únicas para identificação de aplicações clientes.
RF02	O sistema deve validar a autenticidade de cada requisição via middleware centralizado.
RF03	O sistema deve permitir habilitar/desabilitar funcionalidades em tempo real sem novo deploy.
RF04	O sistema deve suportar a definição de regras de segmentação personalizadas por flag.
RF05	O sistema deve registrar cada evento de avaliação (hit/miss) para fins de auditoria e analytics.
RF06	O sistema deve fornecer endpoints de Health Check para monitoramento do cluster AKS.

REQUISITOS NÃO FUNCIONAIS (RNF)		
RNF01	Segurança	API Keys não devem ser armazenadas em texto plano (uso de SHA-256).
RNF02	Performance	O serviço de avaliação deve responder rápidamente (uso de Redis).
RNF03	Escalabilidade	Os serviços de Analytics e Evaluation devem escalar horizontalmente conforme a carga.
RNF04	Resiliência	A falha no serviço de Analytics não deve impedir a avaliação de flags (desacoplamento via filas).
RNF05	Observabilidade	

Serviço	Linguagem	Porta	Função
auth-service	Go 1.21	8001	Gerenciamento e validação de API Keys
flag-service	Python 3.11	8002	CRUD de Feature Flags

targeting-service	Python 3.11	8003	CRUD de regras de segmentação
evaluation-service	Go 1.21	8004	Avaliação de flags (com cache Redis)
analytics-service	Python 3.11	8005	Consumo de eventos e persistência

ARQUITETURA CLOUD NA AZURE

Design de Solução:

Para atender aos requisitos de persistência e mensageria na Azure, realizamos o seguinte mapeamento:

Recurso AWS	Recurso Azure	Finalidade
EKS	**AKS (Azure Kubernetes Service)**	Orquestrador de contêineres Managed.
ECR	**ACR (Azure Container Registry)**	Armazenamento privado de imagens Docker.
RDS (PostgreSQL)	**Azure Database for PostgreSQL**	Persistência de dados relacionais (Auth, Flag, Target).
ElastiCache (Redis)	**Azure Cache for Redis**	Cache de alta performance para o evaluation-service.
DynamoDB	**Azure Cosmos DB (com API NoSQL)**	Banco NoSQL para eventos de analytics.
SQS	**Azure Service Bus (Queues)**	Mensageria assíncrona entre Evaluation e Analytics.

FIAP

TERRAFORM

Provisionando a Infraestrutura na Nuvem

Usamos o Terraform para o provisionamento dos recursos na Azure (AKS, Bancos e Filas) e os Manifestos Kubernetes (YAML)

Adotando uma estrutura de diretórios baseada em Domínios de Serviço, garantindo que cada microsserviço seja uma unidade independente de deploy.

Implementamos o Horizontal Pod Autoscaler (HPA) nos serviços críticos.

Gatilho de Escala: Configurado para disparar quando a utilização média de CPU ultrapassar 70%.

Funcionamento: O AKS monitora as métricas via Metrics Server. Ao atingir o threshold, o Kubernetes instancia automaticamente novos Pods (dentro dos limites de minReplicas e maxReplicas) para distribuir a carga e manter a latência baixa.

Arquivo	Alteração
terraform/postgresql.tf	Adicionado atributo zone em cada PostgreSQL Flexible Server para fixar a zona de disponibilidade atribuída pela Azure (auth=3, flags=1, targeting=1), evitando drift no terraform apply

INFRAESTRUTURA AZURE (TERRAFORM)		
Ferramenta	**Versão mínima**	**Finalidade**
Terraform	1.5+	Provisionamento de infraestrutura
Azure CLI	2.50+	Autenticação e gestão da Azure
Conta Azure	—	Subscription ativa com permissões de Owner/Contributor

KUBERNETES
Configuração do Cluster Kubernetes **Nginx Ingress:** Usado para gerenciar o IP Público do e roteiar o tráfego com base nos paths: **Metrics Server:** Coletar métricas de CPU e Memória dos Pods em tempo real. **Namespaces:** Criado o namespace para isolamento.

DEPLOY KUBERNETES		
Ferramenta	**Versão mínima**	**Finalidade**
kubectl	1.28+	Provisionamento de infraestrutura
Docker Buildx	incluso no Docker Desktop	Build multi-platform (linux/amd64)

FLUXO PRINCIPAL	
1. Criar API Key	POST /auth/keys (protegido por MASTER_KEY)
2. Criar Flag	POST /flags/flags (autenticado via Authorization: <api-key>)
3. Criar Regra de Targeting	POST /targeting/rules (autenticado)
4. Avaliar Flag	GET /evaluate/evaluate?user_id=X&flag_name=Y - O evaluation-service consulta o flag-service e o targeting-service - Resultado é cacheado no Redis - Evento é enviado assincronamente para a fila de mensagens
5. Analytics	O analytics-service consome da fila e persiste no banco NoSQL

ESTRATÉGIA DE TESTES	
Teste 01	Teste de Health Check: Verificação os
Teste 02	Teste de Carga: Escalando Pods (HPA)

k6 (Grafana)	0.50+	Teste de carga e escalabilidade

ABSTRAÇÃO DO CLOUD PROVIDER

A variável de ambiente CLOUD_PROVIDER controla qual implementação é usada:
***Essa abstração existe no evaluation-service (interface MessageSender em Go) e no analytics-service (funções init_aws()/init_azure() em Python).**

CLOUD_PROVIDER	Fila	NoSQL	Uso
aws (default)	AWS SQS (LocalStack)	AWS DynamoDB (LocalStack)	Desenvolvimento local
azure	Azure Service Bus	Cosmos DB (Table API)	Produção (AKS)

COMUNICAÇÃO ENTRE SERVIÇOS	
evaluation-service	-> flag-service / targeting-service: HTTP com header Authorization usando a SERVICE_API_KEY pré-configurada
evaluation-service	-> fila: Assíncrono (goroutine)
fila	-> analytics-service: Long-polling (SQS) ou receiver loop (Service Bus)

COMPARATIVO DE DATA STORES		
PostgreSQL (auth-service, flag-service, targeting-service)	**Relacional**	Usado onde a consistência e relações complexas são necessárias (ex: regras de usuários e definição de flags). Garante que uma alteração em uma regra de flag ou um novo usuário de API seja gravado com total integridade. Relacionamentos: Ideal para o targeting-service, onde você tem regras complexas que se relacionam com diferentes flags e segmentos. Segurança: No auth-service, a estrutura rígida de tabelas ajuda a manter esquemas de chaves de API e permissões bem definidos. **Situação:** Quando o dado é estruturado e a precisão da informação é mais importante que a velocidade extrema de leitura.
Redis (evaluation-service)	**In-Memory**	Usado quando é necessário baixa latência: O Redis entrega dados em milissegundos, pois mantém informações na RAM. Performance: Toda vez que uma aplicação cliente pergunta se uma flag está ativa, esse serviço é chamado. Ler do disco (RDS) seria lento demais. **Situação:** Para dados que precisam de acesso extremamente rápido e que podem ser recalculados ou buscados no banco principal se o cache expirar.
Cosmos DB/NoSQL (analytics-service)	**Documento**	O analytics-service pode receber muitos eventos. Esse NoSql escala de forma automática para aguentar milhões de requisições por segundo. Esquema Flexível: Eventos de analytics podem mudar de formato rapidamente. Como é NoSQL, não é necessário migrar tabelas (migrations) toda vez que adicionar um campo novo no log. **Situação:** Volume massivo de escrita e necessidade de alta disponibilidade sem gerenciar servidores.

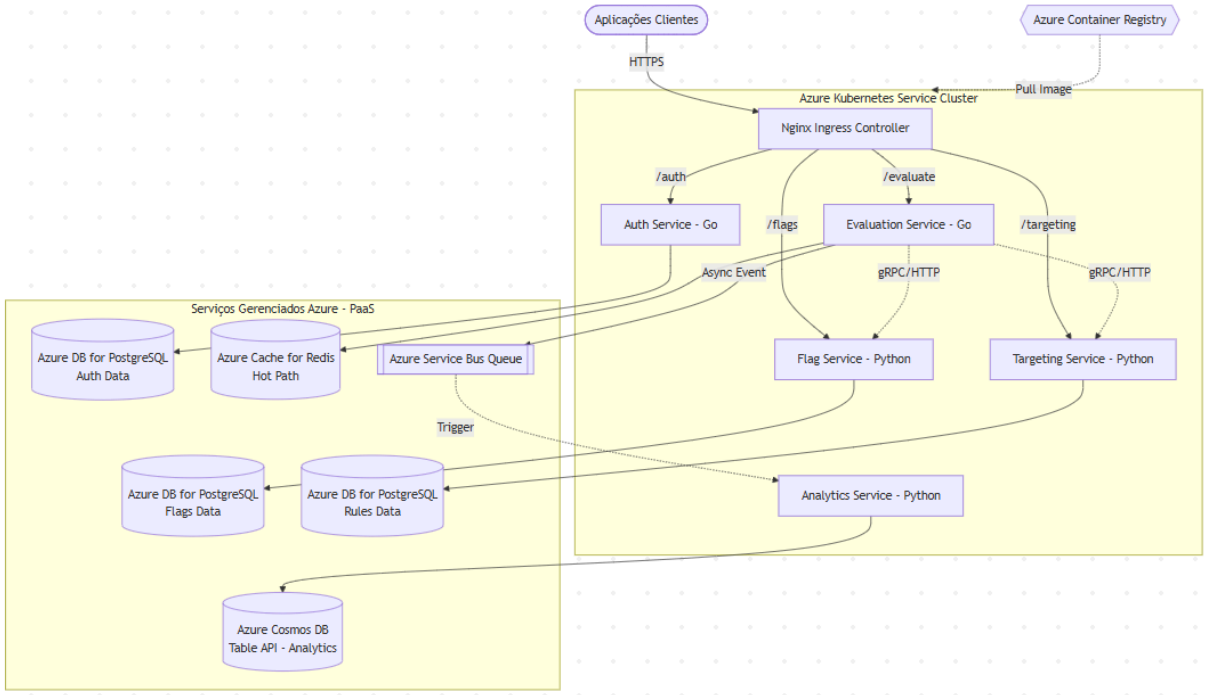
RECURSOS PRINCIPAIS

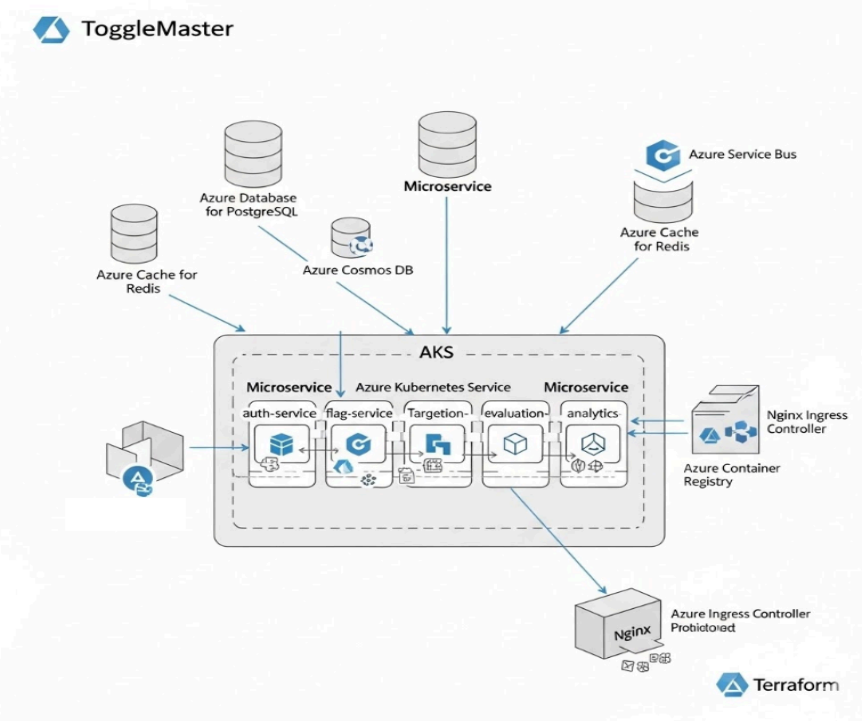
Azure Kubernetes Service (AKS)	**Todos os Microsserviços**
Azure Database for PostgreSQL	**Auth, Flag, Targeting**
Azure Cache for Redis	**Evaluation-Service**
Azure Cosmos DB (Table API)	**Analytics-Service**
Azure Service Bus	**Mensageria (Eval > Analyt)**
Azure Container Registry (ACR)	**CI/CD (Imagens Docker)**
Bandwidth / Ingress	**Nginx Ingress Controller**

CUSTOS DE INFRAESTRUTURA

Calculadora de Custos Azure Tech Challenge			
Requisito do Desafio	**Recurso Azure Correspondente**	**Configuração / SKU**	**Custo Mensal (Est.)**
1. Cluster Kubernetes	Azure Kubernetes Service (AKS)	2 a 4 Nodes (B2s) + Load Balancer	$78.00
2. RDS 1 (Auth)	Azure DB for PostgreSQL	Flexible Server (B1ms) + 32GB	$15.10
3. RDS 2 (Flags)	Azure DB for PostgreSQL	Flexible Server (B1ms) + 32GB	$15.10
4. RDS 3 (Targeting)	Azure DB for PostgreSQL	Flexible Server (B1ms) + 32GB	$15.10
5. ElastiCache (Redis)	Azure Cache for Redis	Tier Basic (C0 - 250MB)	$16.00
6. DynamoDB (Analytics)	Azure Cosmos DB	Table API (Serverless/400 RU)	$24.00
7. SQS (Fila)	Azure Service Bus	Tier Basic (Mensageria)	$0.05
8. ECR (Imagens)	Azure Container Registry	Tier Basic (5 Repositórios)	$5.00
Infra de Rede / Ingress	VNet + NAT Gateway + IP	Saída de dados e Ingress Nginx	$12.00
TOTAL ESTIMADO			**$180.35**
		Em média,	**R$957.66**

DESENHO ARQUITETURA





ARTEFATOS ENTREGÁVEIS DA FASE II	
01	Relatório de Entrega (.PDF ou .txt): Link da Documentação: https://drive.google.com/file/d/1NoaZmWwO4fLt5mIH6-CX4F-yam7RxNdt/view
02	Código GitHub: https://github.com/freitasleoalves/fiap-tech-challenge-2-v1
03	Vídeo de Demonstração (até 15 min): - Apresentação da aplicação rodando localmente. - Demonstração da aplicação rodando na Infraestrutura. https://youtu.be/P505KNzWTBA
04	Desenho da arquitetura:
05	Discussão sobre Banco de Dados:
06	Calculadora Estimativa de Custo Mensal
07	Breve resumo dos desafios encontrados e decisões tomadas.

DIFICULDADES

A tarefa mais complexa deste desafio foi a segregação dos serviços em namespaces Kubernetes independentes e toda a cadeia de impactos que essa mudança provocou.

Por que foi complexo?
Comunicação cross-namespace: Quando todos os serviços compartilhavam um único namespace (togglemaster), a comunicação entre eles usava simplesmente o nome do Service (ex: http://auth-service:8001). Ao mover cada serviço para seu próprio namespace, todas as referências de DNS internas precisaram ser atualizadas para o formato FQDN (http://auth-service.togglemaster-auth.svc.cluster.local:8001). Identificar e atualizar cada ponto de comunicação — incluindo variáveis de ambiente em Deployments, ConfigMaps e Secrets — exigiu mapear cuidadosamente a árvore de dependências entre os microsserviços.

Distribuição granular de Secrets e ConfigMaps: No modelo anterior, todos os Secrets estavam em um único namespace e todos os serviços tinham acesso a todos eles. Com a segregação, foi necessário redesenhar a distribuição de Secrets para que cada namespace contenha apenas o que seu serviço precisa (princípio do menor privilégio). Por exemplo, o auth-service precisa de credenciais do banco e da master-key, mas não de credenciais do Redis ou Cosmos DB. Esse mapeamento granular de "quem precisa de quê" envolveu análise do código-fonte de cada serviço.

Ingress por namespace: O NGINX Ingress Controller faz merge automático de múltiplos recursos Ingress, mas cada recurso precisa estar no namespace correspondente ao Service que referencia. Isso exigiu dividir um único Ingress em 5 recursos separados, um por namespace, mantendo as anotações consistentes e garantindo que nenhuma rota quebrasse.

Jobs de inicialização de banco: Os Jobs que executam init.sql nos bancos PostgreSQL precisaram ser movidos para seus respectivos namespaces, pois cada Job referencia Secrets locais (credenciais do banco). Garantir que a sequência de deploy continuasse funcionando (Jobs completam antes dos Deployments subirem) adicionou complexidade ao fluxo.

Atualização em cascata da documentação: Cada mudança nos manifests K8s propagou alterações nos comandos de deploy, validação, monitoramento e na documentação geral. Manter tudo sincronizado — incluindo referências a

namespaces em 30+ arquivos YAML e no README — foi um exercício de disciplina e atenção a detalhes.

Reorganização dos manifests em diretórios por serviço: Após a segregação por namespace, ficou evidente que manter todos os manifests em arquivos flat numerados (00-namespace.yaml, 01-secrets.yaml, ...) não escalava bem. A migração para uma estrutura de diretórios por serviço (k8s/auth-service/, k8s/flag-service/, etc.) exigiu:

Dividir arquivos monolíticos (ex: 01-secrets.yaml com secrets de 5 serviços) em arquivos individuais por diretório
Reescrever completamente a pipeline CI/CD para usar kubectl apply -f k8s/<service>/ ao invés de referências a arquivos individuais
Mapear service → namespace na pipeline para operações como kubectl set image e kubectl rollout status
Garantir que a ordem de deploy fosse preservada (namespaces → serviços com DB init → serviços sem DB init)
Lições aprendidas
Planejar antes de mover: Mapear todas as dependências entre serviços antes de iniciar a segregação evita surpresas com DNS e Secrets.
FQDN desde o início: Usar FQDNs (service.namespace.svc.cluster.local) como padrão desde o início do projeto facilita futuras migrações de namespace.
Menor privilégio: Segregar Secrets por namespace é uma boa prática de segurança que, embora exija mais trabalho inicial, reduz a superfície de ataque em caso de comprometimento de um serviço.
Estrutura de diretórios por serviço: Organizar manifests K8s em diretórios por serviço torna o projeto mais navegável, facilita code review e permite kubectl apply -f k8s/<service>/ para deploy atômico de cada componente.

ESTRUTURA DO PROJETO

```
fiap-tech-challenge-2/
├── README.md                    # Este documento
├── docker-compose.yml           # Orquestração local (9 containers)
│
├── auth-service/                # Microsserviço de autenticação (Go)
│   ├── Dockerfile
```

```
│   ├── .dockerignore
│   ├── main.go / handlers.go / key.go
│   ├── go.mod
│   └── db/init.sql                # Schema + seed da API Key interna
│
├── flag-service/                  # Microsserviço de flags (Python)
│   ├── Dockerfile
│   ├── .dockerignore
│   ├── app.py
│   ├── requirements.txt
│   └── db/init.sql                # Schema da tabela flags
│
├── targeting-service/             # Microsserviço de segmentação (Python)
│   ├── Dockerfile
│   ├── .dockerignore
│   ├── app.py
│   ├── requirements.txt
│   └── db/init.sql                # Schema da tabela targeting_rules
│
├── evaluation-service/            # Microsserviço de avaliação (Go)
│   ├── Dockerfile
│   ├── .dockerignore
│   ├── main.go / handlers.go / evaluator.go
│   ├── queue.go                   # Interface MessageSender
│   ├── sqs.go                     # Implementação AWS SQS
│   ├── servicebus.go              # Implementação Azure Service Bus
│   ├── types.go
│   └── go.mod
│
├── analytics-service/             # Microsserviço de analytics (Python)
│   ├── Dockerfile
│   ├── .dockerignore
│   ├── app.py                     # Dual-provider (AWS / Azure)
│   └── requirements.txt
│
├── localstack/
│   └── init-resources.sh          # Cria fila SQS e tabela DynamoDB no LocalStack
│
├── scripts/
```

```
│   ├── init-postgres-apps.sql      # Init do banco flags_db
│   └── init-targeting-db.sh        # Init do banco targeting_db
│
├── .github/
│   └── workflows/
│       └── deploy.yml              # CI/CD pipeline (GitHub Actions)
│
├── terraform/                      # IaC para Azure
│   ├── providers.tf                # Provider azurerm ~> 3.100
│   ├── variables.tf                # Variáveis de configuração
│   ├── terraform.tfvars            # Valores das variáveis
│   ├── main.tf                     # Resource Group + locals
│   ├── aks.tf                      # ACR + AKS Cluster
│   ├── postgresql.tf               # 3x PostgreSQL Flexible Server
│   ├── redis.tf                    # Azure Cache for Redis
│   ├── servicebus.tf               # Azure Service Bus + Queue
│   ├── cosmosdb.tf                 # Cosmos DB (Table API)
│   └── outputs.tf                  # Connection strings e endpoints
│
├── load-test/                      # Teste de carga
│   ├── k6-load-test.js             # Script k6 (smoke, ramp-up, stress)
│   └── README.md                   # Documentação do teste de carga
│
└── k8s/                            # Manifests Kubernetes (por serviço)
    ├── base/
    │   └── namespaces.yaml         # 5 Namespaces (1 por serviço)
    ├── auth-service/
    │   ├── secrets.yaml            # Secrets (DB + master-key)
    │   ├── configmap.yaml          # ConfigMap SQL init
    │   ├── db-init-job.yaml        # Job inicialização do banco
    │   ├── deployment.yaml         # Deployment + Service
    │   └── ingress.yaml            # Ingress /auth
    ├── flag-service/
    │   ├── secrets.yaml            # Secrets (DB)
    │   ├── configmap.yaml          # ConfigMap SQL init
    │   ├── db-init-job.yaml        # Job inicialização do banco
    │   ├── deployment.yaml         # Deployment + Service
    │   └── ingress.yaml            # Ingress /flags
    ├── targeting-service/
```

```
│   ├── secrets.yaml          # Secrets (DB)
│   ├── configmap.yaml          # ConfigMap SQL init
│   ├── db-init-job.yaml        # Job inicialização do banco
│   ├── deployment.yaml           # Deployment + Service
│   └── ingress.yaml          # Ingress /targeting
├── evaluation-service/
│   ├── secrets.yaml          # Secrets (Redis, Service Bus, API Key)
│   ├── configmap.yaml          # ConfigMap app-config
│   ├── deployment.yaml           # Deployment + Service
│   ├── ingress.yaml          # Ingress /evaluate
│   └── hpa.yaml              # HPA (2-10 replicas, 70% CPU)
├── analytics-service/
│   ├── secrets.yaml          # Secrets (Service Bus, Cosmos DB)
│   ├── configmap.yaml          # ConfigMap app-config
│   ├── deployment.yaml           # Deployment + Service
│   ├── ingress.yaml          # Ingress /analytics
│   └── hpa.yaml              # HPA (2-10 replicas, 70% CPU)
└── load-test/
    ├── namespace.yaml          # Namespace togglemaster-loadtest
    ├── configmap.yaml        # Script k6 embutido
    └── job.yaml            # Job k6 para teste de carga
```

Necessário Criar

Dockerfiles e Build

Arquivo	Descrição
auth-service/Dockerfile	Multi-stage build: golang:1.21-alpine → alpine:3.19
flag-service/Dockerfile	Multi-stage build: python:3.11-slim (build + runtime)
targeting-service/Dockerfile	Multi-stage build: python:3.11-slim (build + runtime)
evaluation-service/Dockerfile	Multi-stage build: golang:1.21-alpine → alpine:3.19
analytics-service/Dockerfile	Multi-stage build: python:3.11-slim (build + runtime)
*/.dockerignore	Ignora .git, __pycache__, .env, README.md, etc.

Docker Compose

Arquivo	Descrição
docker-compose.yml	Orquestra 9 containers: 2 PostgreSQL, 1 Redis, 1 LocalStack, 5 serviços de aplicação

Infraestrutura Local

Arquivo	Descrição
localstack/init-resources.sh	Script que cria fila SQS (evaluation-events) e tabela DynamoDB (EvaluationEvents) no LocalStack
scripts/init-postgres-apps.sql	Cria tabela flags + trigger no flags_db
scripts/init-targeting-db.sh	Cria banco targeting_db + tabela targeting_rules + trigger

Abstração de Cloud Provider

Arquivo	Descrição
evaluation-service/queue.go	Interface MessageSender com método SendEvent(userID, flagName, result)
evaluation-service/servicebus.go	ServiceBusSender — implementação Azure Service Bus

Terraform (Azure)

Arquivo	Recursos
terraform/providers.tf	Provider azurerm ~> 3.100
terraform/variables.tf	Variáveis: subscription, location, SKUs, credenciais
terraform/terraform.tfvars	Valores configurados
terraform/main.tf	Resource Group rg-togglemaster-prod
terraform/aks.tf	ACR (Basic) + AKS (Standard_B2s, autoscale 1-4 nodes)
terraform/postgresql.tf	3x PostgreSQL Flexible Server (B_Standard_B1ms, 32GB) em eastus2
terraform/redis.tf	Azure Cache for Redis (Basic C0)
terraform/servicebus.tf	Service Bus namespace (Basic) + queue evaluation-events
terraform/cosmosdb.tf	Cosmos DB (Table API) + tabela EvaluationEvents (400 RU)
terraform/outputs.tf	Connection strings, hostnames e endpoints

CI/CD

Arquivo	Descrição

.github/workflows/deploy.yml	Pipeline GitHub Actions: build matrix (5 serviços), push ACR, deploy AKS com rollout automático

Kubernetes Manifests (estrutura por serviço)

Cada serviço possui seu próprio diretório em k8s/ contendo todos os manifests necessários:

Diretório	Conteúdo
k8s/base/	namespaces.yaml — 5 Namespaces segregados (1 por serviço)
k8s/auth-service/	secrets, configmap (SQL init), db-init-job, deployment+service, ingress — namespace togglemaster-auth
k8s/flag-service/	secrets, configmap (SQL init), db-init-job, deployment+service, ingress — namespace togglemaster-flags
k8s/targeting-service/	secrets, configmap (SQL init), db-init-job, deployment+service, ingress — namespace togglemaster-targeting
k8s/evaluation-service/	secrets, configmap, deployment+service, ingress, hpa — namespace togglemaster-evaluation
k8s/analytics-service/	secrets, configmap, deployment+service, ingress, hpa — namespace togglemaster-analytics
k8s/load-test/	namespace, configmap (script k6), job — namespace togglemaster-loadtest
load-test/k6-load-test.js	Script k6 com cenários smoke, ramp-up e stress (execução local)

Frente 1 - Docker Compose (Local)

Subir o ambiente

docker compose up -d –build

Containers

Container	Imagem	Porta externa
postgres-auth	postgres:15-alpine	5433
postgres-apps	postgres:15-alpine	5434

redis	redis:7-alpine	6379
localstack	localstack/localstack:latest	4566
auth-service	build local	8001
flag-service	build local	8002
targeting-service	build local	8003
evaluation-service	build local	8004
analytics-service	build local	8005

Variáveis de Ambiente Importantes (Local)

CLOUD_PROVIDER não é definido -> default aws -> usa LocalStack
AWS_ENDPOINT_URL=http://localstack:4566 -> redireciona SDK da AWS para LocalStack
MASTER_KEY=master-key-local-dev
SERVICE_API_KEY=tm_key_service_internal_dev

Parar o ambiente

docker compose down -v

Frente 2 Terraform (Azure)

Recursos Provisionados

Recurso	Nome	Região
Resource Group	rg-togglemaster-prod	eastus
Container Registry	acrtogglemasterprod	eastus
AKS Cluster	aks-togglemaster-prod	eastus
PostgreSQL (auth)	pg-auth-togglemaster-prod	eastus2
PostgreSQL (flags)	pg-flags-togglemaster-prod	eastus2
PostgreSQL (targeting)	pg-targeting-togglemaster-prod	eastus2
Redis Cache	redis-togglemaster-prod	eastus
Service Bus	sb-togglemaster-prod	eastus
Cosmos DB	cosmos-togglemaster-prod	eastus

Nota: PostgreSQL usa eastus2 porque eastus está com restrição de quota para PostgreSQL Flexible Server.

Executar

```
cd terraform
az login
terraform init
terraform plan
terraform apply
```

Outputs Importantes

```
terraform output aks_kube_config_command   # Comando para configurar kubectl
terraform output acr_login_server          # URL do registry
terraform output -json                     # Todos os outputs (connection strings sensíveis)
```

Frente 3 - Kubernetes (AKS)

Segregação por Namespaces

Cada microsserviço reside em seu próprio namespace, garantindo isolamento de recursos e segurança:

Serviço	Namespace	Recursos Isolados
auth-service	togglemaster-auth	Secrets (DB), ConfigMap (SQL init), Deployment, Service, Ingress
flag-service	togglemaster-flags	Secrets (DB), ConfigMap (SQL init), Deployment, Service, Ingress
targeting-service	togglemaster-targeting	Secrets (DB), ConfigMap (SQL init), Deployment, Service, Ingress
evaluation-service	togglemaster-evaluation	Secrets (Redis, Service Bus, API Key), ConfigMap,

		Deployment, Service, Ingress, HPA
analytics-service	togglemaster-analytics	Secrets (Service Bus, Cosmos DB), ConfigMap, Deployment, Service, Ingress, HPA

A comunicação entre serviços usa FQDN cross-namespace:

http://auth-service.togglemaster-auth.svc.cluster.local:8001
http://flag-service.togglemaster-flags.svc.cluster.local:8002
http://targeting-service.togglemaster-targeting.svc.cluster.local:8003

Pré-deploy: Build e Push das Imagens

As imagens devem ser compiladas para linux/amd64 (arquitetura dos nodes AKS):

```
az acr login --name acrtogglemasterprod

ACR=acrtogglemasterprod.azurecr.io
for svc in auth-service flag-service targeting-service evaluation-service
analytics-service; do
  docker buildx build --platform linux/amd64 -t $ACR/$svc:latest ./$svc --push
done
```

Deploy dos Manifests

```
az aks get-credentials --resource-group rg-togglemaster-prod --name
aks-togglemaster-prod

#Instalar NGINX Ingress Controller
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.9.6/deploy/stat
ic/provider/cloud/deploy.yaml
```

```
#Instalar Metrics Server (necessário para HPA)
kubectl apply -f
https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components
.yaml

#1. Criar namespaces
kubectl apply -f k8s/base/namespaces.yaml

#2. Deploy de cada serviço (aplica todo o diretório de uma vez)
for svc in auth-service flag-service targeting-service evaluation-service
analytics-service; do
  echo "Deploying $svc..."
  kubectl apply -f k8s/$svc/
done

#3. Aguardar jobs de init completarem
kubectl get jobs -n togglemaster-auth -w
kubectl get jobs -n togglemaster-flags -w
kubectl get jobs -n togglemaster-targeting -w
```

Características do Deploy

Namespaces: 5 namespaces isolados (1 por serviço)
Replicas: 2 por serviço (10 pods total)

- HPA: evaluation-service e analytics-service escalam de 2 a 10 replicas (70% CPU)
- Probes: Readiness (5s initial, 10s period) e Liveness (10s initial, 15s period) em /health
- Resources: requests 100m CPU / 64Mi RAM; limits 250-500m CPU / 128-256Mi RAM
- Ingress: 1 recurso Ingress NGINX por namespace (merge automático pelo controller)
- Secrets: segregados — cada namespace contém apenas os secrets necessários para seu serviço

Obter o IP Externo

kubectl get svc -n ingress-nginx ingress-nginx-controller -o jsonpath='{.status.loadBalancer.ingress[0].ip}'

CI/CD - GitHub Actions

Pipeline automatizado que faz build das imagens, push para o ACR e deploy no AKS a cada push na branch main.

Arquivo

.github/workflows/deploy.yml

Jobs

Job	Descrição
build-and-push	Build paralelo (matrix) dos 5 serviços com docker build --platform linux/amd64. Push para ACR com tags latest e SHA do commit.
deploy	Obtém credenciais AKS, aplica namespaces, Secrets (via GitHub Secrets), aplica cada diretório k8s/<service>/, atualiza tags das imagens para SHA atual por namespace, aguarda rollout e verifica health.

Triggers

- Push na branch main — execução automática
- workflow_dispatch — execução manual pelo GitHub UI

GitHub Secrets Necessários

Configurar em Settings -> Secrets and variables -> Actions no repositório:

Secret	Descrição	Exemplo
AZURE_CREDENTIALS	JSON do Service Principal Azure	Gerado via az ad sp create-for-rbac
DB_AUTH_URL	Connection string PostgreSQL (auth)	postgres://user:pass@host:5432/auth_db?sslmode=require
DB_FLAGS_URL	Connection string PostgreSQL (flags)	postgres://user:pass@host:5432/flags_db?sslmode=require
DB_TARGETING_URL	Connection string PostgreSQL (targeting)	postgres://user:pass@host:5432/targeting_db?sslmode=require
REDIS_URL	Connection string Redis	rediss://:key@host:6380/0
SERVICEBUS_CONNECTION_STRING	Connection string Azure Service Bus	Endpoint=sb://...
COSMOSDB_CONNECTION_STRING	Connection string Cosmos DB	DefaultEndpointsProtocol=https;...
MASTER_KEY	Chave mestre para auth-service	Qualquer string segura
SERVICE_API_KEY	API Key para comunicação entre serviços	Deve ter hash SHA256 no banco auth

Criar o Service Principal

```
az ad sp create-for-rbac \
  --name "github-actions-togglemaster" \
  --role contributor \
  --scopes /subscriptions/<SUBSCRIPTION_ID>/resourceGroups/rg-togglemaster-prod \
  --sdk-auth
```

Copie o JSON resultante e salve como secret AZURE_CREDENTIALS no GitHub.

Validações e Testes

Validação Local (Docker Compose)

1: Subir e Verificar Containers
docker compose up -d --build
docker compose ps
Resultado esperado = todos os 9 containers Up (healthy):

NAME	STATUS	PORTS
fiap-tech-challenge-2-analytics-service-1	Up (healthy)	0.0.0.0:8005->8005/tcp
fiap-tech-challenge-2-auth-service-1	Up (healthy)	.0.0.0:8001->8001/tcp
fiap-tech-challenge-2-evaluation-service-1	Up (healthy)	0.0.0.0:8004->8004/tcp
fiap-tech-challenge-2-flag-service-1	Up (healthy)	0.0.0.0:8002->8002/tcp
fiap-tech-challenge-2-localstack-1	Up (healthy)	0.0.0.0:4566->4566/tcp
fiap-tech-challenge-2-postgres-apps-1	Up (healthy)	0.0.0.0:5434->5432/tcp
fiap-tech-challenge-2-postgres-auth-1	Up (healthy)	0.0.0.0:5433->5432/tcp
fiap-tech-challenge-2-redis-1	Up (healthy)	0.0.0.0:6379->6379/tcp
fiap-tech-challenge-2-targeting-service-1	Up (healthy)	0.0.0.0:8003->8003/tcp

2: Health Check

```
for svc in 8001 8002 8003 8004 8005; do
  echo "Port $svc: $(curl -s http://localhost:$svc/health)"
done
```

Resultado esperado:

```
Port 8001: {"status":"ok"}
Port 8002: {"status":"ok"}
Port 8003: {"status":"ok"}
Port 8004: {"status":"ok"}
Port 8005: {"status":"ok"}
```

3:Teste E2E
No Docker Compose o acesso é direto por porta, sem prefixo de rota.

#1) Criar uma feature flag

```
curl -s -X POST http://localhost:8002/flags \
  -H "Content-Type: application/json" \
  -H "Authorization: tm_key_service_internal_dev" \
  -d '{"name":"test-checkout","description":"Teste do fluxo de
checkout","is_enabled":true}'
```

Resultado esperado:

```
{
    "id": 1,
    "name": "test-checkout",
    "description": "Teste do fluxo de checkout",
    "is_enabled": true,
    "created_at": "...",
    "updated_at": "..."
}
```

#2) Criar regra de targeting (100% dos usuários)

```
curl -s -X POST http://localhost:8003/rules \
  -H "Content-Type: application/json" \
  -H "Authorization: tm_key_service_internal_dev" \
```

```
  -d
'{"flag_name":"test-checkout","is_enabled":true,"rules":{"type":"PERCENTAGE","value":
100}}'
```

Resultado esperado:

```
{
    "id": 1,
    "flag_name": "test-checkout",
    "is_enabled": true,
    "rules": {"type": "PERCENTAGE", "value": 100},
    "created_at": "...",
    "updated_at": "..."
}
```

#3) Avaliar a flag para um usuário

```
curl -s "http://localhost:8004/evaluate?user_id=user-123&flag_name=test-checkout"
```

Resultado esperado:

```
{"flag_name": "test-checkout", "user_id": "user-123", "result": true}
```

4:Verificar Pipeline Assíncrona (SQS → DynamoDB via LocalStack)

```
#Aguardar ~3s após a avaliação e consultar o DynamoDB
docker exec fiap-tech-challenge-2-localstack-1 \
  awslocal dynamodb scan \
  --table-name EvaluationEvents \
  --region us-east-1
```

Resultado esperado:

```
{
    "Items": [
        {
            "result": {"BOOL": true},
            "event_id": {"S": "<uuid>"},
            "user_id": {"S": "user-123"},
            "flag_name": {"S": "test-checkout"},
            "timestamp": {"S": "2026-03-12T00:17:54.167Z"}
```

```
        }
    ],
    "Count": 1,
    "ScannedCount": 1
}
```

Isso confirma que o fluxo completo funcionou:

1. evaluation-service avaliou a flag
2. Evento foi enviado assincronamente para fila SQS (LocalStack)
3. analytics-service consumiu da fila e persistiu no DynamoDB

Validação Azure (Terraform + AKS)

1:Verificar Terraform

```
cd terraform
terraform plan  # Deve mostrar "No changes"
```

Resultado esperado: No changes. Your infrastructure matches the configuration.

2:Verificar Cluster e Pods

```
# Nodes do cluster
kubectl get nodes

#Pods de todos os namespaces da aplicação
for ns in togglemaster-auth togglemaster-flags togglemaster-targeting
togglemaster-evaluation togglemaster-analytics; do
  echo "=== $ns ==="
  kubectl get pods -n $ns
done
```

Resultado esperado: 2 nodes Ready + 10 pods Running (2 por namespace, 1/1 Ready) + 3 jobs Completed.

3:Health Check

```
INGRESS_IP=$(kubectl get svc -n ingress-nginx ingress-nginx-controller \
  -o jsonpath='{.status.loadBalancer.ingress[0].ip}')

for svc in auth flags targeting evaluate analytics; do
  echo "$svc: $(curl -s http://$INGRESS_IP/$svc/health)"
```

done
Resultado esperado — todos retornam {"status":"ok"}.

4:Teste E2E
No AKS os endpoints possuem prefixo de rota (/auth/, /flags/, /targeting/, /evaluate/, /analytics/) pois passam pelo Ingress NGINX com rewrite.

```
INGRESS_IP=$(kubectl get svc -n ingress-nginx ingress-nginx-controller \
  -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
BASE=http://$INGRESS_IP
```

#1) Criar flag
```
curl -s -X POST $BASE/flags/flags \
  -H "Content-Type: application/json" \
  -H "Authorization: tm_key_service_internal_dev" \
  -d '{"name":"test-checkout","description":"Teste do fluxo de
checkout","is_enabled":true}'
```

#2) Criar regra de targeting
```
curl -s -X POST $BASE/targeting/rules \
  -H "Content-Type: application/json" \
  -H "Authorization: tm_key_service_internal_dev" \
  -d
'{"flag_name":"test-checkout","is_enabled":true,"rules":{"type":"PERCENTAGE","value":
100}}'
```

#3) Avaliar flag
```
curl -s "$BASE/evaluate/evaluate?user_id=user-123&flag_name=test-checkout"
```
Resultado esperado (mesmos do Docker Compose).

5:Verificar Pipeline Assíncrona (Service Bus → Cosmos DB)
```
kubectl logs -n togglemaster-analytics deployment/analytics-service | grep "salvo"
```
Saída esperada:

Evento <uuid> (Flag: test-checkout) salvo no Cosmos DB Table.

6:Verificar HPA
```
kubectl get hpa -n togglemaster-evaluation
kubectl get hpa -n togglemaster-analytics
```

Resultado esperado:

NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS
evaluation-service-hpa Deployment/evaluation-service <cpu>/70% 2 10 2
analytics-service-hpa Deployment/analytics-service <cpu>/70% 2 10 2

Teste de Carga - k6

Utilizamos o k6 (Grafana) para validar a escalabilidade da plataforma, especialmente o evaluation-service que possui HPA configurado.

Cenários

Cenário	Descrição	VUs Máx	Duração
Smoke	Validação básica com carga mínima	5	30s
Ramp-up	Carga crescente para observar autoscaling (HPA)	200	6min
Stress	Pico de 300 VUs simultâneos	300	3min

Thresholds (critérios de sucesso)

- p(95) latência < 2s para todas as requests
- < 5% de falhas HTTP
- p(99) latência do /evaluate < 3s
- < 10% de erros no endpoint /evaluate

Execução Local
#Instalar k6 (macOS)
brew install k6

#Executar contra o Ingress (substituir IP)
k6 run \
  -e BASE_URL=http://<INGRESS_EXTERNAL_IP> \
  -e API_KEY=tm_key_service_internal_dev \
  load-test/k6-load-test.js

Execução no Cluster (K8s Job)
#Aplicar o Job do k6 no cluster
kubectl apply -f k8s/load-test/

#Acompanhar os logs do teste
kubectl logs -f job/k6-load-test -n togglemaster-loadtest

#Limpar após o teste
kubectl delete -f k8s/load-test/
Observando o HPA durante o teste
#Em terminais separados:
kubectl get hpa -n togglemaster-evaluation -w
kubectl get pods -n togglemaster-evaluation -w
O HPA deve escalar o evaluation-service de 2 para até 10 réplicas conforme a CPU aumenta, e reduzir gradualmente após o fim do teste.

Implementar/Melhorar:

MODELO DE DOMINIO

DIAGRAMA DE CLASSE

ARQUITETURA - DESENHO

DIAGRAMA DE SEQUÊNCIA

IMPLANTAÇÃO - PASSOS