Comando de Resiliência - Operação Helius

FASE 2: PROJETO ARQUITETURA RESILIENTE

1. Etapa Analítica (Aprofundamento)

A. Análise do mlflow-tracker e grafana-core (SPOF Críticos)

Os arquivos de implantação confirmam que os nós mais críticos do sistema são configurados como Single Points of Failure (SPOF) globais:

Componente	Configuração Encontrada	Fragilidade	Consequência no Incidente
mlflow-tracker	replicas: 1 e usa emptyDir: {} para armazenamento.	**SPOF e Armazenamento Efêmero:** Uma única réplica significa que uma falha de nó o derruba. O uso de emptyDir implica que os artefatos de modelo (mlflow.db) são **perdidos** em caso de reinicialização/migração, causando o erro 500 no recommender-service na inicialização.	O colapso na rede regional AWS pode ter causado a perda da réplica, paralisando globalmente o carregamento de novos modelos.
grafana-core	replicas: 1.	**SPOF de Observabilidade:** Uma única réplica garante que, se o servidor de métricas (monitoring-service) for atingido e a UI falhar em conectar, a visualização global de *dashboards* seja perdida, confirmando a **cegueira sistêmica** durante o incidente.	O serviço de monitoramento já estava em erro (Database connection failed em 11:27:23Z), e ter uma única réplica do Grafana impede qualquer redundância na camada de visualização.

B. Análise do llm-router (Propagação de Falha e Roteamento Cego)

A análise do LLM Router revela que ele não foi projetado para degradar de forma graciosa:

Detalhe do Arquivo	Ponto de Roteamento	Fragilidade	Impacto no Incidente

deployment_llm.yaml	replicas: 1.	**SPOF:** O LLM Router é o ponto de entrada para todas as decisões de IA. Sua falha é crítica.	
main.py (função route_request)	Roteamento *single-path* para llm-assistant.	**Ausência de *Failover*:** A lógica de roteamento só considera o provedor local. Em caso de erro (como *timeout* ou falha de rede), ele falha diretamente com **HTTP 502 (Bad Gateway)**.	As "decisões incoerentes" reportadas se transformam em "serviço indisponível" na primeira falha de rede do *backend* do LLM, agravando o impacto ao cliente.

Conclusão Analítica: O Paradoxo da Resiliência

A arquitetura atual do Helius opera em um paradoxo de resiliência:

1. **Dependência Única na Infraestrutura Local:** Serviços críticos (mlflow-tracker, grafana-core, llm-router) rodam como réplicas únicas, tornando-os altamente dependentes da saúde de um único *cluster* ou nó.
2. **Violação do Isolamento de Domínio de Falha:** Uma falha de infraestrutura regional (rede/disco) em qualquer região hospedeira se traduz em uma falha **funcional global** no serviço de IA (recommendation-engine) e no serviço de controle (monitoring-service).
3. **Falta de Modo Degradado (*Fail-Open*):** Nenhum dos serviços tem lógica para **degradar o serviço** (ex: servir um modelo estático, um *cache* antigo, ou um erro 200 com confiança 0.0) em vez de falhar completamente (erros 5xx na inicialização ou 502 no runtime).

Para o *blueprint* de resiliência, precisamos introduzir:

1. **Isolamento Regional:** Multi-região com separação de domínios de falha (VPCs e Subnets).
2. **Redundância Ativa-Ativa/Ativa-Passiva:** mlflow-tracker e grafana-core devem ser redundantes e usar armazenamento persistente e distribuído (S3/EFS/Multi-AZ).
3. **Roteamento Inteligente (LLM Router):** Implementar lógica de *failover* (chamar outro provedor) ou *fail-open* (retornar um valor de *fallback* seguro, em vez de 502).

C. Análise do Modelo Conceitual (architecture_base.puml)

O diagrama PlantUML, apesar de delinear um sistema com laços de autocura (SelfHealing e ControlPlane), revela que as capacidades de resiliência estão incompletas, alinhando-se perfeitamente com os sintomas observados:

Componente	Fragilidade Confirmada pelo «extension point»
TelemetryGateway	Confirma a necessidade de **Filtros Anti-Duplicação** e **Circuit Breakers Regionais**. Isso valida que a sobrecarga de ingestão (duplicação MQTT) foi o ponto de entrada da falha e que não havia proteção nativa contra ela.
RecoEngine	Confirma a falta de **Modo Fallback Estático** e **Regras de Confiança Mínima**. Isso explica diretamente as "recomendações erráticas" e o colapso do serviço no início do incidente.
ControlPlane	Confirma a ausência de **Estratégias Multi-Região**. Este é o ponto mais crítico: a falha não estava isolada em uma região ou zona, mas era um problema de design de orquestração global.
Modularidade (Visual)	O alto acoplamento entre os componentes centrais (Control Plane, Telemetry, Model Serving) no diagrama explica a baixa modularidade (0.3267) e a facilidade de propagação da falha em cascata.

D. Análise do Modelo de Infraestrutura (main.tf e variables.tf)

O código Terraform materializa a vulnerabilidade conceitual ao **prender toda a operação a uma única região AWS**.

Recurso Terraform	Configuração	Fragilidade na Resiliência

region	Variável padronizada para uma única região (us-east-1 por *default*).	**Falta de Isolamento de Falhas (Regional):** Qualquer evento que afete o *Control Plane* da AWS ou a rede na região de hospedagem (o que foi a causa raiz do incidente) resultará em uma **interrupção global**.
module "vpc"	Utiliza single_nat_gateway = true.	**SPOF de Saída (Egress):** Embora Multi-AZ, a dependência de um único NAT Gateway em uma das Zonas de Disponibilidade (AZ) torna todo o tráfego de saída da rede privada vulnerável a uma falha de AZ.
aws_s3_bucket.mlflow_artifacts	Apenas provisiona um *bucket* S3.	**Durabilidade é regional:** Embora o S3 seja durável, a replicação entre regiões não é garantida por padrão, e a dependência de serviços como o mlflow-tracker em um volume emptyDir (vulnerabilidade anterior) agrava a situação.

2. Desenho do Blueprint de Resiliência

A **Fase 1 do Blueprint de Arquitetura Resiliente** foca na fundação da plataforma, utilizando o **Terraform** para impor o princípio de **Redundância e Desacoplamento Regional**. O objetivo é eliminar os *Single Points of Failure (SPOF)* de infraestrutura que permitiram que uma falha de rede regional se propagasse globalmente durante o incidente.

Fase 1: Redesenho de Infraestrutura e Isolamento de Falhas (Terraform) 🛠️

O redesenho desta fase será implementado através de modificações nos módulos de rede e armazenamento persistente na AWS, migrando de uma arquitetura *Single-Region* para um modelo *Multi-Region* (Ativo-Passivo).

---

1. Camada de Infraestrutura/Rede: Isolamento de Domínio de Falha

O design original, focado em otimização de custo, comprometeu a resiliência ao centralizar o tráfego de saída (Egress) em um único ponto e operar em uma única região.

A. Migração para Topologia Multi-Região (Ativo-Passivo)

O *blueprint* exige o provisionamento dos recursos essenciais da Helius em duas Regiões AWS distintas (ex: `us-east-1` como Primária e `us-west-2` como Secundária).

- **Objetivo de Contenção:** Garantir que uma falha catastrófica no *Control Plane* da Região Primária (como a instabilidade de rede que gerou a falha em cascata) ative um **Failover Automático** para a Região Secundária, isolando o domínio de falha geográfica.
- **Mecanismo de Roteamento:** A orquestração do tráfego será centralizada no **AWS Route 53**, utilizando políticas de roteamento baseadas em **Latência** ou **Health Checks** para direcionar o tráfego para a região saudável (Active-Passive, com a Secundária em modo *Hot Standby* ou *Warm Standby*).

B. Eliminação do SPOF de Egress (NAT Gateway)

O `main.tf` original usa `single_nat_gateway = true` no módulo VPC, forçando todo o tráfego de saída das sub-redes privadas através de um único recurso.

- **Rigor Técnico:** A resiliência exige a eliminação deste SPOF no nível da **Zona de Disponibilidade (AZ)**.
- **Modificação no `main.tf` (Conceitual):** A configuração da VPC deve ser alterada para provisionar **Múltiplos NAT Gateways**, um em cada Zona de Disponibilidade, garantindo que a falha de uma AZ não paralise o tráfego de saída:
- Terraform

```
module "vpc" {
  # ... outros parâmetros
  enable_nat_gateway = true
  # MUDANÇA CRÍTICA: Provisiona um NAT Gateway por AZ
  single_nat_gateway = false
  # ...
}
```

-
-

- **Resultado:** Isso garante que a perda de um único NAT Gateway ou AZ **não impeça** os workers dos clusters EKS de se comunicarem com APIs externas (como a do MLflow, no caso de estar externa ou replicada) ou de realizarem chamadas de *egress* essenciais.

---

2. Camada de Dados Críticos (MLflow): Persistência e Replicação

A falha do `mlflow-tracker` como SPOF (réplica única) e o uso de armazenamento efêmero (`sqlite` e `emptyDir`) tornaram a crise de dados ML inevitável. Esta fase resolve o problema de **persistência** e **disponibilidade regional** do estado do modelo.

A. Replicação de Artefatos de Modelo (S3 Cross-Region Replication - CRR)

- **Objetivo:** Garantir que todos os artefatos de modelo (TorchScript, *checkpoints*, etc.) no `helius-mlflow-artifacts` estejam disponíveis na Região Secundária em caso de desastre.
- **Mecanismo:** Configurar o S3 para **Replicação de Região Cruzada (CRR)**. O `aws_s3_bucket.mlflow_artifacts` na Região Primária deve ter uma política que replique objetos para um *bucket* de destino na Região Secundária.
- **Benefício de Resiliência:** Garante um **RPO (Recovery Point Objective)** baixo para os artefatos, permitindo que o `recommender-service` na região de *failover* inicie imediatamente, carregando a versão mais recente do modelo, mesmo que a região primária esteja inacessível.

B. Persistência e Alta Disponibilidade do MLflow Metadata (RDS Multi-AZ)

- **Problema Endereçado:** O uso de `MLFLOW_BACKEND_STORE_URI="sqlite:///mlflow.db"` como um volume `emptyDir` torna os metadados (versões, métricas) irrecuperáveis e não escaláveis.
- **Solução:** Substituir o `sqlite` pelo **AWS RDS (PostgreSQL) com Multi-AZ Deployment**.
    - **Implementação:** O *blueprint* aproveita a definição de variáveis RDS (`rds_db_name`, `rds_username`, etc.) e configura o recurso `aws_rds_instance` (não mostrado nos trechos, mas assumido pelo contexto Terraform) com a opção **`multi_az = true`**.
    - **Resultado:** Os metadados do MLflow ganham **persistência transacional** e **failover automático síncrono** dentro da região, elevando o MLflow-Tracker de um SPOF efêmero para um componente de **Alta Disponibilidade (HA)**.

A **Fase 2 do Blueprint de Arquitetura Resiliente** migra o foco da infraestrutura para o **comportamento dos serviços** e dos modelos de IA, introduzindo lógica de degradação e contenção de falhas no código e no pipeline de dados. O princípio central é **Degradação Controlada (Fail-Open)**, garantindo que a Helius mantenha a disponibilidade funcional mesmo sob falha parcial.

Fase 2: Endurecimento e Modos Degradados Inteligentes (Serviço e IA) 🧠

O objetivo é transformar os SPOFs de código em *gateways* tolerantes a falhas, capazes de se autolimitar ou recorrer a um modo de operação de baixo desempenho em vez de falhar com erros 5xx.

---

1. Camada de Ingestão (Data): Resiliência e Contenção de Backpressure

A sobrecarga causada pela duplicação de mensagens MQTT nos *edge devices* e o pico de latência no `telemetry-gateway` foram o principal vetor de propagação da falha. O foco é desacoplar a origem da falha do sistema *downstream*.

A. Buffer Assíncrono (Kafka/Kinesis) para Desacoplamento Temporal

- **Problema Endereçado:** A ingestão síncrona ou com filas de capacidade limitada (como uma fila interna do serviço) não absorve picos de tráfego (como o incidente de duplicação) sem pressão de retorno (*backpressure*).
- **Mecanismo:** Introduzir um **Cluster Apache Kafka (ou Amazon Kinesis)** como *buffer* intermediário persistente e altamente escalável entre os *Edge Nodes* e o `telemetry-gateway` (ingest-service).
    - O `telemetry-gateway` agora apenas **consome** do Kafka/Kinesis em um ritmo sustentável, em vez de ser diretamente atingido pela taxa de picos de escrita dos *edge devices*.
- **Vantagem:** Isso cria um **desacoplamento temporal e de volume**, impedindo que um pico de I/O na borda sobrecarregue imediatamente o sistema central (incluindo o `recommender-service` e o *data store*). O pico é absorvido pela capacidade de escrita escalável do Kafka/Kinesis.

B. Circuit Breakers e Rate Limiting no `telemetry-gateway`

- **Mecanismo:** Implementar o padrão **Circuit Breaker** (ex: usando bibliotecas como **Hystrix** ou **Resilience4j**) e **Rate Limiting** dentro da lógica do `telemetry-gateway` (ingest-service).
    - **Circuit Breaker:** Monitora a latência e a taxa de erro das chamadas *downstream* (por exemplo, ao escrever no *data store* do MLflow/RecoEngine). Se a latência exceder o limite (como o pico observado no incidente), o *breaker* **abre**, rejeitando temporariamente novas mensagens e **protegendo os serviços *downstream* do colapso**.
    - **Rate Limiting Regional:** Implementar limitação de taxa (ex: 500k mensagens/s por região) para o tráfego de entrada. Esta taxa pode ser dinamicamente ajustada pelo **Control Plane** (Fase 3) em resposta ao alerta `PipelineBacklogHigh`, permitindo a **autocontenção cibernética** ao isolar ou limitar regiões problemáticas.

---

2. Camada de IA (RecoEngine): Padrão de Cache/Fallback

O recommender-service (service_10) falhou catastroficamente ao não conseguir carregar o modelo na inicialização (RuntimeError), resultando em erro 500.

A. Implementação do Padrão de Cache/Fallback (Fail-Open no Startup)

- **Problema Endereçado:** Dependência síncrona na inicialização (@app.on_event("startup") e load_model() que depende do MLflow).
- **Mecanismo:** Modificar a função load_model no recommender_service/main.py para usar um *cache local* (disco ou volume persistente) como fonte primária de resiliência.
    1. **Tentativa 1 (MLflow/S3):** Tenta mlflow.pytorch.load_model(MLFLOW_MODEL_URI).
    2. **Tentativa 2 (Cache Local):** Se a Tentativa 1 falhar com RuntimeError (falha de rede, MLflow fora), carrega o último modelo conhecido e validado a partir de um disco persistente local (/tmp/cache/last_good_model).
    3. **Tentativa 3 (Modelo Estático de Fallback):** Se o cache local estiver ausente (primeira inicialização ou corrupção de cache), carrega um **Modelo Estático Hardcoded** de baixíssima complexidade (ex: modelo que retorna o *top 5* global).
- **Resultado:** O serviço nunca retorna erro 500 no startup. Ele opera em **modo degradado**, mas com **disponibilidade total** (Availability over Consistency).

---

3. Camada de IA (LLM Router): Roteamento Inteligente com Failover

O llm-router é o ponto de controle das decisões de IA, mas atualmente só chama um único assistente local e falha com HTTP 502 em caso de erro.

A. Roteamento com Failover para Região Secundária

- **Problema Endereçado:** Colapso total do roteamento em caso de falha de rede para o LLM Primário.
- **Mecanismo:** Modificar a função route_request para implementar roteamento sequencial com *failover* (Retry/Fallback Pattern).
    1. **Primário:** Tenta chamar o LLM Assistant na Região Primária (LLM_ASSISTANT_URL).
    2. **Failover:** Se o primário retornar *timeout* ou erro 5xx (falha regional), tenta a URL do LLM Assistant na Região Secundária (LLM_ASSISTANT_SECONDARY_URL).

B. Padrão Fail-Open com Confiança Zero

- **Problema Endereçado:** Garantir que o *router* retorne uma resposta que obedeça ao esquema ModelDecision em qualquer circunstância (evitar HTTP 502).
- **Mecanismo (Última Instância):** Se ambas as chamadas (Primário e Secundário) falharem, o roteador deve retornar um objeto ModelDecision *hardcoded* (Fallback Estático) com os seguintes atributos:
    - action: "safemode_fallback"
    - confidence: 0.0
    - rationale: "Erro sistêmico. Resposta estática de segurança."

- **Benefício:** Isso permite que o sistema *downstream* (o cliente) trate a resposta como funcional, mas com **confiança explicitamente zero**, garantindo que não sejam tomadas decisões incoerentes baseadas em dados defeituosos (como ocorreu no incidente). A ausência de erro 5xx mantém a disponibilidade, enquanto a confiança 0.0 sinaliza o problema.

Fase 3: Automação e Controle (Observabilidade e Self-Healing) 🤖

O princípio de design desta fase é o **Controle em *Loop* Fechado (Cybernetic Control)**. Transformamos o sistema de um conjunto de alarmes passivos (alerts.yml [cite: uploaded:alerts.yml]) em um sistema ativo de resposta e remediação, utilizando a observabilidade para alimentar as decisões de mitigação.

---

1. Camada de Observabilidade: Redundância e Visibilidade Ativa

A perda da visibilidade (grafana-core e monitoring-service afetados) durante o incidente foi o que permitiu o *autoscaling* descontrolado e a propagação cega da falha. O foco é garantir que os dados de saúde sejam resistentes à falha do próprio serviço.

A. Prometheus Sidecars para *Self-Observability*

- **Problema Endereçado:** A coleta de métricas de saúde e status de *fallback* depende de *endpoints* de serviço que podem falhar, causando cegueira.
- **Mecanismo:** Adicionar um *sidecar container* (Prometheus Exporter) a cada Pod dos serviços críticos (recommender-service, ingest-service, llm-router).
    - **Função do Sidecar:** Exporta métricas internas do serviço para o Prometheus (como as métricas de latência e erro já definidas [cite: uploaded:prometheus.yml]), mas também métricas de estado de resiliência:
        - reco_engine_mode_static_active: 1 se estiver rodando no **Modo Fallback** (Fase 2) ou 0 se estiver normal.
        - telemetry_circuit_breaker_state: Estado do *Circuit Breaker* (Open, Half-Open, Closed).
- **Benefício:** O *Control Plane* (Automação) pode consultar a métrica do *sidecar* para saber o **estado de degradação** do serviço em tempo real, independentemente da saúde do *endpoint* principal de negócio.

B. Redundância do Control Plane da Observabilidade

- **Mecanismo:** Aumentar a redundância dos SPOFs de observabilidade e registro de ML.
    - **mlflow-tracker e grafana-core:** Aumentar as réplicas de replicas: 1 [cite: uploaded:deployment.yaml, uploaded:deployment_grafana.yaml] para **replicas: 3** (ou N+1 em Multi-AZ).
- **Rigor Técnico:** O deployment.yaml e deployment_grafana.yaml devem ser atualizados para 3 réplicas e utilizar um PodAntiAffinity para garantir que as réplicas sejam distribuídas em Zonas de Disponibilidade ou nós diferentes, eliminando o SPOF no nível do cluster EKS.

---

2. Camada de Controle/Automação: Autocontenção Cibernética (Self-Healing)

Esta é a ponte entre a detecção de anomalias (Observabilidade) e a ação corretiva (Controle), implementando o conceito de **Feedback Loop de Contenção**.

A. Feedback Loop de Contenção (Regra de 3 Passos)

O *Control Plane* (Automação) é ligado ao **Alertmanager** (que recebe o alerta PipelineBacklogHigh [cite: uploaded:alerts.yml] do Prometheus). A automação dispara a lógica de **Self-Healing** [cite: uploaded:architecture_base.puml] com a seguinte sequência de mitigação:

Passo	Alerta Disparador	Ação de Contenção (Self-Healing)	Serviço Alvo	Racional
1	PipelineBacklogHigh (Fila > 100 por 10m)	**Isolar Região e Reduzir Carga:** Identificar a região de origem do maior *backlog* e emitir comandos para o *Load Balancer* (ou API Gateway) para **isolar o tráfego de entrada dessa região** e invocar a redução do **Rate Limit** no telemetry-gateway.	Load Balancer, telemetry-gateway	**Contenção Imediata:** Elimina a causa do *backpressure* e impede a corrupção de novos dados.

2	HighInvalidModelOutputs (Saída Inválida > 5%)	**Forçar Degradação:** Enviar um sinal para o RecoEngine para entrar imediatamente no **Modo Fallback** estático (via ConfigMap ou *feature flag*), ignorando o modelo dinâmico e o *data stream* corrompido.	RecoEngine	**Proteção Funcional:** Interrompe a entrega de recomendações erráticas aos clientes, trocando a acurácia pela segurança do output.
3	Falha de Conectividade do MLflow	**Forçar Remediação:** Disparar um *re-deploy* ou *rolling restart* dos Pods do mlflow-tracker e grafana-core na região para forçar a reconexão aos novos *endpoints* RDS Multi-AZ.	mlflow-tracker, grafana-core	**Recuperação da Visibilidade:** Restabelece a persistência e a observabilidade essenciais para auditoria e operação humana.

---

3. Camada de IA Confiável: Validação e Controle do Drift

A garantia da qualidade do *output* da IA é a linha de defesa final contra a corrupção de dados (causa primária do *drift*).

A. Validação Sincrona com Py-LLM-Shield

- **Mecanismo:** Fortalecer a validação no llm-router e no recommender-service.
    - **LLM Router:** A função validate_with_shield [cite: uploaded:1thirteeng3/operation-helios-architecture-of-an-autonomous-organizational-resilience-system/Operation-Helios-Architecture-of-an-Autonomous-Organizational-Resilience-System-63f90bc4d4b178cab683f444bd8259a4cf2f9b6e/services/llm_assistant/main.py] não deve apenas tentar reparar, mas registrar cada falha de validação em uma métrica separada (model_invalid_outputs_total), que alimenta o alerta

HighInvalidModelOutputs. A repetição de falhas deve levar à rejeição da região de origem.

B. Rastreamento e Re-Equilíbrio (XAI como Gatilho)

- **Mecanismo:** Utilizar métricas de Explicabilidade (XAI) como *watchdogs* para o *drift* funcional, além do Drift Score passivo [cite: uploaded:model_health.json].
    - **XAI como Alerta:** Se a **Importância de *Features* (Feature Importance)** do modelo começar a divergir bruscamente (sinalizando *Data Drift* ou *Concept Drift*), o mlflow-tracker exporta um *custom metric* (por exemplo, reco_engine_xai_divergence > 0.3).
    - **Ação do Control Plane:** Este *custom alert* atua como um gatilho de *Self-Healing* (re-equilibra) para:
        1. **Isolar** o *data stream* para o reco-engine afetado.
        2. **Disparar** um *job* de retreinamento autônomo (usando train_gnn.py [cite: uploaded:1thirteeng3/operation-helios-architecture-of-an-autonomous-organizational-resilience-system/Operation-Helios-Architecture-of-an-Autonomous-Organizational-Resilience-System-63f90bc4d4b178cab683f444bd8259a4cf2f9b6e/ml/train_gnn.py]) com os últimos dados válidos, re-equilibrando o modelo.

I. Princípios de Design e Justificativas para a Arquitetura Resiliente

A nova arquitetura Helius não busca apenas corrigir falhas, mas sim **antecipar e isolar** o impacto de eventos disruptivos, como o incidente do Dia 0. Isso é alcançado pela adesão estrita a três pilares de resiliência, que abordam as falhas de estrutura, funcionalidade e controle observadas.

Princípio	Definição Estrutural (Ação)	Justificativa Rigorosa (Contexto do Incidente)
1. Isolamento de Falha (Contenção)	**Segregar e Contenir:** Implementar barreiras hard e soft (regionais, de rede, de processo) para garantir que a falha em um componente (ou domínio) não se	A falha no **Ingest/Telemetry** (devido à duplicação MQTT) se propagou para o **RecoEngine** e paralisou o **MonitoringService** (service_00, nó de alta centralidade). A causa raiz foi a **infraestrutura Single-Region** [cite: uploaded:variables.tf] e SPOFs (MLflow e Grafana de réplica única [cite: uploaded:deployment.yaml, uploaded:deployment_grafana.yaml]), que

	propague para outros.	permitiram que uma falha local causasse um colapso global e cego.
2. Redundância e Degradação (Confiabilidade)	**Manter Disponibilidade (Fail-Open):** Fornecer caminhos de execução alternativos e forçar os serviços a operar em um modo simplificado ou de baixo desempenho em vez de falhar.	O **RecoEngine** falhou totalmente no *startup* (HTTP 500) devido à dependência síncrona do MLflow/S3. O **LLM Router** falhou "cego" (erros 5xx no *downstream*) [cite: uploaded:main.py]. A falta de um **Padrão de Cache/Fallback** forçou o sistema à indisponibilidade total em vez de servir uma recomendação estática ou de confiança zero.
3. Controle Cibernético (*Self-Healing*)	**Fechar o Loop de Controle:** Utilizar dados de observabilidade em tempo real (métricas e saúde) para acionar correções automáticas e políticas de contenção, eliminando a dependência de intervenção humana em crises.	O sistema era essencialmente **passivo** (baseado em alertas [cite: uploaded:alerts.yml]). A contenção da sobrecarga e o isolamento regional (a resposta correta) foram feitos **manualmente**. O **Control Plane** deve automatizar as ações corretivas ao detectar o *backlog* ou o *drift*.

Detalhamento Técnico-Conceitual dos Princípios

1. Isolamento de Falha (Contenção)

Este princípio é a linha de defesa mais fundamental contra a natureza em cascata das falhas de arquitetura.

- **Isolamento Geográfico (Hard Barrier):** A migração para a topologia **Multi-Região** (Fase 1) é a barreira *hard*. Ela garante que a perda total de uma Região AWS

(devido a desastre ou falha de infraestrutura no *Control Plane*) não afete a Região Secundária.
- **Isolamento de Egress (NAT Gateway):** O fim do `single_nat_gateway = true` [cite: uploaded:main.tf] garante que o tráfego de saída (crítico para acessar APIs externas e repositórios como o MLflow no S3) seja resiliente à falha de uma Zona de Disponibilidade.
- **Desacoplamento Temporal (Buffer Assíncrono):** A introdução do **Kafka/Kinesis** (Fase 2) cria uma barreira de tempo, garantindo que o pico de ingestão (sobrecarga/duplicação MQTT) não se propague instantaneamente como *backpressure* aos serviços de processamento a jusante (RecoEngine, Data Stores).

2. Redundância e Degradação (Confiabilidade)

Este princípio é aplicado para elevar o **SLO (Service Level Objective)** de disponibilidade, especialmente para serviços de IA.

- **Redundância de SPOF (N+3):** Serviços de estado como **MLflow Tracker** e **Grafana Core** (que eram `replicas: 1` [cite: uploaded:deployment.yaml, uploaded:deployment_grafana.yaml]) são migrados para implantações Multi-AZ com `replicas: 3` e armazenamento replicado (RDS Multi-AZ e S3 CRR).
- **Modo Fallback para Carga Crítica:**
    - **RecoEngine:** O *Padrão de Cache/Fallback* garante que a indisponibilidade de um *data store* ou do MLflow resulte em uma recomendação **válida, mas subótima** (Modo Estático/Cache), mantendo o cliente na plataforma.
    - **LLM Router:** O **Roteamento com Failover** para a região secundária (e, em última instância, o *Fail-Open* com Confiança 0.0) garante que o sistema de decisões de IA nunca retorne um erro irrecuperável ao cliente, priorizando a **Disponibilidade** acima da **Consistência Perfeita**.

3. Controle Cibernético (*Self-Healing*)

O princípio de **Controle em *Loop* Fechado** é o que define a *Ciber-Resiliência* e elimina a necessidade de intervenção humana no **Ciclo de Mitigação Rápida** (tempo de reação em milissegundos versus minutos).

- **Observabilidade Ativa:** O uso de **Prometheus Sidecars** (Fase 3) garante que os dados de saúde e o status de degradação (*e.g.*, se o RecoEngine está em *Fallback*) sejam visíveis, alimentando o *Control Plane*.
- **Autocontenção Ativa:** O *Control Plane* não apenas alerta, mas executa uma **ação corretiva autônoma** (ex: isolamento de região ao detectar `PipelineBacklogHigh`).
- **Re-Equilíbrio (Drift Automation):** A elevação das métricas de IA Confiável (Validação de Output e XAI) a **Gatilhos de Automação** permite que o sistema tome decisões de **rejeição de tráfego** (contenção cibernética) ou **retreinamento autônomo** (re-equilíbrio). O sistema se torna, assim, adaptativo e autorregulável.

Este tópico do *blueprint* detalha a visão conceitual da nova arquitetura. Ele serve como o mapa de alto nível que traduz os **Princípios de Design** (Isolamento, Redundância e

Controle) em uma topologia funcional de camadas (SysML/UML), que será implementada nas Fases 1, 2 e 3.

II. Diagrama Conceitual (SysML-like Blocks) 🗺️

A nova arquitetura **Helius Multi-Region Ciber-Resiliente** abandona o modelo *Single-Region* [cite: uploaded:variables.tf] e os SPOFs de réplica única [cite: uploaded:deployment.yaml] em favor de um sistema distribuído e federado.

A fundação é baseada em dois (ou mais) *Clusters* EKS (Primário e Secundário), provisionados em regiões AWS distintas (`us-east-1`, `us-west-2`) e operando em VPCs totalmente isoladas. A conexão entre eles não é uma dependência síncrona, mas sim um canal de **replicação de dados assíncrona** (Fase 1) e **roteamento de serviço inteligente** (Fase 2).

A arquitetura é dividida nas seguintes camadas lógicas, que representam o fluxo de dados e controle:

---

1. Camada de Borda (Edge Layer)

- **Blocos (SysML):** `[Edge Devices (MQTT)]` -> `[TelemetryGateway (com Circuit Breaker)]`
- **Fluxo:** Esta é a fronteira do sistema e a primeira linha de defesa. O `TelemetryGateway` (ingest-service) atua como o *choke point* (ponto de estrangulamento) controlado.
- **Rigor Técnico (Resiliência):** Esta camada implementa a **Contenção de *Backpressure*** (Fase 2). O `TelemetryGateway` não é mais um receptor passivo; ele aplica ativamente **Rate Limiting** e **Circuit Breakers Regionais** para identificar e isolar *Edge Devices* ou regiões que apresentem comportamento anômalo (como a duplicação de mensagens MQTT que iniciou o incidente), protegendo as camadas internas.

2. Camada de Buffer Assíncrono (Async Buffer Layer)

- **Blocos (SysML):** `[TelemetryGateway]` -> `[Global Message Bus (Kafka/Kinesis)]` -> `[Backend Consumers]`
- **Fluxo:** O `TelemetryGateway` não escreve mais diretamente nos *data stores* dos serviços. Ele publica eventos em um *Buffer Assíncrono* (Kafka/Kinesis). Serviços *downstream* (como o RecoEngine e Analytics) consomem deste *buffer* em seu próprio ritmo.
- **Rigor Técnico (Resiliência):** Esta camada implementa o **Desacoplamento Temporal** (Fase 2). Ela absorve picos de latência e volume, impedindo que a sobrecarga na ingestão cause falhas em cascata nos serviços de processamento de IA. O alerta `PipelineBacklogHigh` [cite: uploaded:alerts.yml] agora monitora este *buffer*, servindo como o principal indicador de saturação do sistema para o *Control Plane*.

3. Camada de Dados (Data Layer)

- **Blocos (SysML):** [Amazon S3 (CRR)] <- [Amazon RDS Multi-AZ]
- **Fluxo:** Esta camada armazena o estado crítico e persistente. O MLflow.db (metadados) reside no **RDS Multi-AZ** (HA intra-regional), e os artefatos de modelo residem no **S3 com Cross-Region Replication (CRR)** (HA inter-regional).
- **Rigor Técnico (Resiliência):** Esta camada implementa a **Redundância de Dados** (Fase 1). Ela elimina os SPOFs de armazenamento (emptyDir [cite: uploaded:deployment.yaml] e sqlite [cite: uploaded:Dockerfile]), garantindo que os metadados e os artefatos de modelo sobrevivam a falhas de nó, AZ e até mesmo regionais, permitindo o *failover* da Camada de IA.

4. Camada de Serviço (IA Layer)

- **Blocos (SysML):** [RecoEngine (com Fallback)] <- [LLM Router (com Failover)]
- **Fluxo:** O LLM Router recebe requisições de decisão. O RecoEngine consome do *Buffer* e serve recomendações.
- **Rigor Técnico (Resiliência):** Esta camada implementa a **Degradação Controlada** (Fase 2).
    - O RecoEngine agora possui lógica de **Fallback** (Padrão de Cache) para carregar um modelo estático se o MLflow/S3 falhar, garantindo disponibilidade (HTTP 200) em vez de falha (HTTP 500) [cite: uploaded:1thirteeng3/operation-helios-architecture-of-an-autonomous-organizational-resilience-system/Operation-Helios-Architecture-of-an-Autonomous-Organizational-Resilience-System-63f90bc4d4b1778cab683f444bd8259a4cf2f9b6e/services/recommender_service/main.py].
    - O LLM Router implementa **Failover Lógico** para a Região Secundária e **Fail-Open** (resposta estática com confidence: 0.0) se todos os *backends* falharem, evitando o HTTP 502 [cite: uploaded:main.py].

5. Camada de Controle (Control Plane Layer)

- **Blocos (SysML):** [Control Plane (Automação)] <-> [Self-Observability Layer (Prometheus/Grafana HA)]
- **Fluxo (Ciclo Cibernético):** Este é o *loop* de feedback que implementa o *Self-Healing* (Fase 3).
    1. **Recebe (Input):** O *Control Plane* recebe dados de **topologia e saúde** da Camada de Observabilidade (que agora é HA, com replicas: 3 [cite: uploaded:deployment_grafana.yaml] e *Sidecars*). Ele monitora métricas de *drift* (model_drift_score [cite: uploaded:model_health.json]) e o estado de degradação (reco_engine_mode_static_active).
    2. **Envia (Output):** Em resposta a alertas (como PipelineBacklogHigh), o *Control Plane* envia comandos de **Autocontenção** (ex: "Isolar Região US-EAST-1" via API do Load Balancer) e **Scale/Failover** (ex: "Promover Região US-WEST-2 como Primária").
- **Rigor Técnico (Resiliência):** O *Control Plane* é a realização do princípio de **Controle Cibernético**, transformando a arquitetura de um sistema passivo em um sistema autonômico que gerencia ativamente seus próprios domínios de falha.

III. Componentes Técnicos e Interações ⚙️

A arquitetura resiliente da Helius é alcançada através da modificação dos componentes de software para que adotem padrões de design defensivos, quebrando as cadeias de falha síncronas que causaram o incidente do Dia 0.

---

1. Ingestão (`telemetry-gateway`)

O `telemetry-gateway` (ingest-service) evolui de um *endpoint* passivo para um *gateway* inteligente e desacoplado.

- **Mecanismo Implementado (Kafka/Kinesis + Circuit Breaker):**
  - **Buffer Assíncrono:** A ingestão de *edge devices* (MQTT) não é mais síncrona com o processamento. Os dados são imediatamente escritos em um *Buffer* persistente de alta ingestão (Kafka/Kinesis), conforme sugerido no *extension point* do `architecture_base.puml`.
  - **Consumo Controlado:** O `telemetry-gateway` torna-se um consumidor desse *buffer*, processando mensagens em um ritmo controlado (definido pelo `Control Plane`).
  - **Circuit Breaker:** O *gateway* implementa um padrão de Circuit Breaker (Fase 2) que monitora a saúde dos *data stores* (ex: RDS, S3). Se a latência de escrita exceder os limiares (como no incidente), o *breaker* abre, **pausando o consumo do Kafka** e impedindo que o *backpressure* se propague para os *data stores*.
- **Camada e Desacoplamento (Desacoplamento Temporal):**
  - Este design introduz um **desacoplamento temporal** completo. A duplicação de mensagens MQTT (causa raiz da sobrecarga) não impacta mais o *gateway* ou o `RecoEngine` diretamente; ela apenas aumenta o *lag* da fila no Kafka/Kinesis.
  - Isso transforma o alerta `PipelineBacklogHigh` no principal sensor do `Control Plane` para detectar sobrecarga, permitindo uma resposta de *rate limiting* (Fase 3) antes que a falha atinja os serviços centrais.

---

2. MLflow e Grafana (SPOFs de Controle)

Esses componentes críticos de MLOps e Observabilidade são endurecidos contra falhas de infraestrutura.

- **Mecanismo Implementado (Replicação N+3 + Persistência):**
  - **Redundância de Réplicas (N+3):** Os *deployments* do `mlflow-tracker` e `grafana-core` são atualizados de `replicas: 1` para `replicas: 3` (ou

mais), com `PodAntiAffinity` para garantir a distribuição entre Zonas de Disponibilidade (Multi-AZ).
    - **Persistência (HA):** O `MLFLOW_BACKEND_STORE_URI` é alterado de `sqlite:///mlflow.db` (efêmero) para um **AWS RDS Multi-AZ** (Fase 1). O `volumeMounts` (que usava `emptyDir: {}`) é reconfigurado para usar o *bucket* **S3 com Cross-Region Replication (CRR)** como `MLFLOW_ARTIFACT_ROOT`.
- **Camada e Desacoplamento (Redundância de SPOF):**
    - A falha de um nó ou AZ não causa mais **cegueira operacional** (Grafana) ou **falha no *startup* de serviços** (MLflow).
    - A replicação inter-regional (CRR) garante que, mesmo em um *failover* regional completo, o *cluster* secundário possa carregar os artefatos de modelo do MLflow, permitindo a recuperação do serviço de IA.

---

3. `RecoEngine` (Serviço de IA Crítico)

O `recommender-service` é modificado para priorizar a disponibilidade sobre a consistência imediata do modelo.

- **Mecanismo Implementado (Padrão de Cache/Fallback):**
    - O `RuntimeError` fatal na função `load_model` (causado pela falha do MLflow) é tratado com um bloco `try...except`.
    - A lógica de inicialização (`@app.on_event("startup")`) é reescrita para tentar carregar o modelo em cascata:
        1. **Tentar (Primário):** `mlflow.pytorch.load_model(MLFLOW_MODEL_URI)`.
        2. **Exceto (Cache):** Se falhar, tenta carregar o último modelo conhecido de um *cache* em disco local (`/cache/model.pt`).
        3. **Exceto (Fallback):** Se o *cache* falhar, carrega um **Modelo Estático de *Fallback*** (um modelo *hardcoded* que retorna recomendações genéricas, ex: *top-10* global), conforme sugerido no `architecture_base.puml`.
- **Camada e Desacoplamento (Resiliência Funcional):**
    - O serviço **sempre iniciará com sucesso (HTTP 200)**, eliminando o erro 500 que paralisou o serviço no incidente.
    - O *Prometheus Sidecar* (Fase 3) exportará a métrica `reco_engine_mode_static_active`, informando ao `Control Plane` que o serviço está operando em modo degradado.

---

4. `LLM Router` (Gateway de Decisão de IA)

O `llm-router` é transformado de um simples *proxy* em um *gateway* de decisão tolerante a falhas.

- **Mecanismo Implementado (Failover Lógico + Validação):**
    - **Failover Lógico:** A função `route_request`, que falhava com `HTTPException(status_code=502)`, é modificada (Fase 2) para tentar o `LLM_ASSISTANT_URL` primário. Em caso de *timeout* ou erro 5xx, ele automaticamente tenta a URL da Região Secundária (`LLM_ASSISTANT_SECONDARY_URL`).
    - **Fail-Open (Confiança Zero):** Se ambos falharem, em vez de 502, ele retorna um objeto `ModelDecision` *hardcoded* (ex: `action: "fallback"`, `confidence: 0.0`), garantindo a disponibilidade.
    - **Validação (py-llm-shield):** O *output* de qualquer resposta bem-sucedida (do primário ou secundário) é rigorosamente validado pelo `validate_with_shield`.
- **Camada e Desacoplamento (IA Confiável - Safety):**
    - O *Failover* garante a **resiliência regional** da camada de decisão.
    - A validação e o *Fail-Open* (Confiança Zero) garantem que **decisões incoerentes ou corrompidas** (a causa do *drift* no incidente) sejam filtradas ou explicitamente sinalizadas como não confiáveis, protegendo o usuário final do impacto funcional.

IV. Medidas de Resiliência, Segurança e IA Confiável 🛡️

A arquitetura resiliente da Helius é validada pela sua capacidade de se autoproteger contra falhas estruturais (como o colapso regional do Dia 0) e falhas funcionais (como o *Data Drift* que levou a decisões incoerentes).

---

1. Resiliência Estrutural (Contenção)

Esta camada foca em implementar a **Autocontenção Cibernética**, transformando a observabilidade de uma ferramenta passiva de diagnóstico em um gatilho ativo para o `Control Plane`.

A. Isolamento de Falha por *Loop* de Controle Cibernético

O `Control Plane` (Automação) implementa o princípio de *Self-Healing* [cite: uploaded:architecture_base.puml] ao fechar o *loop* entre detecção e ação, movendo a mitigação de falhas da escala humana (minutos/horas) para a escala de máquina (segundos).

- **Detecção (Sensor):** O Prometheus monitora o alerta `PipelineBacklogHigh` [cite: uploaded:alerts.yml], que atua como o principal sensor de sobrecarga no *Buffer* Assíncrono (Kafka/Kinesis) da Fase 2. Uma anomalia persistente (ex: `for: 10m`)

sinaliza uma falha de ingestão regional ou uma duplicação de mensagens (a causa raiz do incidente).

- **Ação Corretiva (Atuador):** Ao receber o alerta do Alertmanager, o `Control Plane` (Automação) executa autonomamente a **Ação Corretiva de *Self-Healing*** [cite: uploaded:architecture_base.puml]:
    1. **Isolamento do Ofensor:** Identifica o *Edge Device* ou a região de origem do tráfego anômalo e aplica regras dinâmicas (ex: AWS WAF ou API Gateway) para **isolar o tráfego** dessa fonte específica.
    2. **Redução de Carga (Contenção):** Reduz dinamicamente o *Rate Limit* do `telemetry-gateway` (Fase 2) para aliviar o *backpressure* nos *data stores* e impedir a propagação da falha em cascata.

B. *Self-Observability* por *Sidecar* (Visibilidade Ativa)

Para evitar a "cegueira operacional" (onde a falha do `grafana-core` [cite: uploaded:deployment_grafana.yaml] e `monitoring-service` ocultou a crise), a observabilidade é desacoplada do processo principal do aplicativo.

- **Mecanismo:** Cada Pod Kubernetes (ex: `RecoEngine`, `LLM Router`) é implantado com um contêiner **Prometheus Sidecar** (Fase 3).
- **Rigor Técnico:** Este *sidecar* é um processo separado que monitora a saúde do contêiner principal. Ele coleta métricas internas (ex: latência da JVM/Python, uso de *thread pool*) e, crucialmente, reporta o **status de degradação** (`Fallback Mode`) para o `Control Plane`.
- **Benefício:** Se o `RecoEngine` entrar em Modo Fallback (Fase 2), o *sidecar* reporta `reco_engine_mode_static_active = 1`. O `Control Plane` agora sabe que o serviço está **disponível, mas degradado**, permitindo uma resposta de engenharia (ex: "Verificar Fonte de Dados do MLflow") em vez de uma resposta de infraestrutura (ex: "Reiniciar Pod").

---

2. Segurança e IA Confiável (Safety & Trust)

Esta camada é a linha de defesa final, garantindo a **integridade funcional** dos modelos de IA e prevenindo que dados corrompidos (como o *Data Drift* do incidente) resultem em "recomendações erráticas" para o cliente.

A. Controle de *Output* de IA (Validação de Esquema Mandatória)

A validação de esquema é a principal barreira contra a corrupção de dados e vetores de ataque.

- **Mecanismo:** A validação com **`py-llm-shield`** (ou validação Pydantic rigorosa, como visto em `llm_assistant/main.py` [cite: uploaded:1thirteeng3/operation-helios-architecture-of-an-autonomous-organizational-resilience-system/Operation-Helios-Architecture-of-an-Autonomous-Organizational-R

esilience-System-63f90bc4d4b178cab683f444bd8259a4cf2f9b6e/services/llm_assistant/main.py]) é mandatória em todos os `LLM Routers` e `RecoEngines`.
- **Impacto no *Drift* (Corrupção):** A validação de esquema teria **rejeitado** os dados corrompidos (`Data corruption detected` [cite: uploaded:logs.jsonl]) provenientes do `ingest-service`, impedindo que eles chegassem ao `RecoEngine`. Isso teria quebrado a cadeia da falha funcional, prevenindo as "decisões incoerentes".
- **Impacto na Segurança (Ataques):** Ao impor um esquema de saída estrito (ex: JSON), o `py-llm-shield` mitiga ataques de **Prompt Injection**, onde um ator malicioso tenta fazer o LLM retornar *scripts* ou comandos executáveis. O validador rejeitará qualquer *output* que não se conforme ao esquema de "decisão" esperado.

B. XAI (Explicabilidade) como Gatilho de Detecção de *Drift*

O `Drift Score` do `model_health.json` [cite: uploaded:model_health.json] é uma métrica de *lagging* (reativa). A arquitetura resiliente introduz o XAI (eXplainable AI) como um sensor de *leading* (proativo).

- **Mecanismo:** O `Control Plane` (Fase 3) monitora métricas de **Explicabilidade (XAI)** (ex: *feature importance* ou SHAP values) geradas pelo *pipeline* de MLOps.
- **Detecção de Falha de Justificativa:** O sistema agora detecta não apenas a **falha funcional** (ex: `HighInvalidModelOutputs` [cite: uploaded:alerts.yml] ou queda de acurácia), mas também a **falha de justificativa** (ex: o modelo subitamente passa a basear 90% de suas decisões em uma *feature* irrelevante).
- **Re-Equilíbrio Autônomo:** Essa divergência de XAI (ex: `xai_feature_divergence > 0.5`) é um alerta de alta prioridade. O `Control Plane` o interpreta como um sinal de *Data Drift* severo e pode **autonomamente pausar o *data stream*** para o modelo afetado e **disparar um *job* de retreinamento** (re-equilíbrio) usando o `ml/train_gnn.py` [cite: uploaded:1thirteeng3/operation-helios-architecture-of-an-autonomous-organizational-resilience-system/Operation-Helios-Architecture-of-an-Autonomous-Organizational-Resilience-System-63f90bc4d4b178cab683f444bd8259a4cf2f9b6e/ml/train_gnn.py], garantindo a confiabilidade contínua do modelo.

Fase 1: Implementação de Infraestrutura (Terraform)

Objetivo: Modificar os arquivos Terraform (.tf) para eliminar SPOFs (Single Points of Failure) regionais e de Zona de Disponibilidade (AZ), estabelecendo a fundação Multi-Região (Ativo-Passivo).

1. Modificação de `variables.tf`: Habilitando Multi-Região

Para suportar o *failover* regional (Ativo-Passivo), introduzimos uma variável para a região secundária.

Arquivo: `infra/terraform/aws/variables.tf`

Diff

```
# Variáveis de configuração para a infraestrutura AWS


variable "project_name" {
  description = "Nome do projeto ou prefixo para recursos"
  type        = string
  default     = "helius-sim"
}


variable "region" {
  description = "Região AWS primária em que os recursos serão criados"
  type        = string
  default     = "us-east-1"
}


+ variable "region_secondary" {
+  description = "Região AWS secundária (passiva) para failover"
+  type        = string
+  default     = "us-west-2"
+ }


variable "vpc_cidr" {
```

```
# ... (restante do arquivo)
```

Justificativa: Esta adição permite que os módulos do Terraform provisionem recursos de *failover* (como *buckets* S3 replicados e *clusters* EKS secundários) na region_secondary.

2. Modificação de main.tf: Resiliência de Rede (NAT) e Dados (RDS/S3)

Modificamos o módulo VPC para Alta Disponibilidade (HA) de Egress e definimos conceitualmente a persistência HA para MLflow.

Arquivo: infra/terraform/aws/main.tf

```
Diff

# ---------------------------------------------------------------

# Criação da VPC e sub-redes

# Utiliza módulo oficial terraform-aws-modules/vpc/aws.

module "vpc" {

  source  = "terraform-aws-modules/vpc/aws"

  version = "~> 5.0"



  name = "${var.project_name}-vpc"

  cidr = var.vpc_cidr



  azs             = slice(data.aws_availability_zones.available.names, 0, length(var.public_subnets))

  public_subnets  = var.public_subnets

  private_subnets = var.private_subnets



  enable_nat_gateway = true
```

```
-  single_nat_gateway = true

+  single_nat_gateway = false  # MODIFICAÇÃO DE RESILIÊNCIA: Provisiona um NAT Gateway por AZ



  tags = var.tags

}
```

Justificativa: A alteração de `single_nat_gateway` de `true` para `false` é a correção direta do SPOF de AZ identificado na análise. Isso garante que a falha de uma única AZ não interrompa o tráfego de saída (Egress) das sub-redes privadas.

Novos Recursos Conceituais (Terraform):

1. `aws_rds_instance` (Para Metadados do MLflow):

```
Terraform
# NOVO RECURSO: RDS Multi-AZ para o Backend Store do MLflow

resource "aws_db_instance" "mlflow_metadata_db" {

  # ... (configurações de instância, username, password)

  db_name              = var.rds_db_name

  instance_class       = var.rds_instance_class

  storage_type         = "gp3"



  # MODIFICAÇÃO DE RESILIÊNCIA: Habilita failover automático síncrono

  multi_az             = true



  # ... (configurações de segurança e backup)
```

```
}
```

Justificativa: Substitui o `sqlite:///mlflow.db` por um banco de dados HA, garantindo que os metadados do MLflow sobrevivam a falhas de AZ e reinicializações de pod, eliminando o SPOF de dados.

2. `aws_s3_bucket_replication_configuration` (Para Artefatos do MLflow):

```
Terraform

# MODIFICAÇÃO: Adiciona Replicação Cross-Region ao bucket de artefatos

resource "aws_s3_bucket_replication_configuration" "mlflow_crr" {

  provider = aws.primary # Assume provedor AWS para 'us-east-1'

  bucket   = aws_s3_bucket.mlflow_artifacts.id



  role = aws_iam_role.replication.arn



  rules {

    id     = "FullReplication"

    status = "Enabled"



    destination {

      bucket        = aws_s3_bucket.mlflow_artifacts_secondary.arn # Bucket na região secundária

      storage_class = "STANDARD"

    }

  }

}
```

Justificativa: Implementa a replicação assíncrona (CRR) dos artefatos do modelo (Fase 1), garantindo que a Região Secundária (Passiva) tenha os modelos disponíveis para o *failover* do recommender-service.

Fase 2: Implementação de Serviços (Python/Modo Degradado)

Objetivo: Modificar o código-fonte dos serviços de IA (recommender-service e llm-router) para implementar os padrões **Cache/Fallback** e **Failover Lógico**.

1. recommender_service/main.py: Implementando o Padrão Cache/Fallback

Modificamos o startup_event e load_model para evitar o erro 500 na inicialização se o MLflow estiver indisponível.

Arquivo: services/recommender_service/main.py

Python

```
# ... (imports existentes)

import os

import sys

import logging

import shutil # Usado para cache local



# ... (outros imports)



# Configuração de Logging e Cache

logging.basicConfig(level=logging.INFO)

log = logging.getLogger(__name__)
```

```
MLFLOW_MODEL_URI = os.environ.get("MLFLOW_MODEL_URI", "mlruns/0/model")

CATEGORY_MAPPING_PATH = os.environ.get("CATEGORY_MAPPING_PATH",
"category_mapping.json")



# MODIFICAÇÃO DE RESILIÊNCIA: Caminhos de Cache

LOCAL_MODEL_CACHE_PATH = "/cache/model.pt"

STATIC_FALLBACK_MODEL_PATH = "/app/static_model.pt" # Um modelo básico incluído no
Dockerfile



MODEL = None

CATEGORIES: List[str] = []



def load_static_fallback_model():

    """Carrega um modelo estático de fallback (ex: top-k global)."""

    # Lógica de simulação: Retorna um modelo "dummy" que sempre prevê a categoria 0

    # Em um cenário real, isso carregaria um modelo leve (ex: sklearn)

    log.warning("!!! MODO FALLBACK ESTÁTICO ATIVADO !!!")



    # Simulação de um modelo TorchScript simples

    class DummyModel(torch.nn.Module):

        def __init__(self):

            super().__init__()

            self.output_dim = len(CATEGORIES) if CATEGORIES else 10 # Assume 10 categorias
se o mapping falhar
```

```
        def forward(self, x, edge_index):
            # Retorna logits que favorecem a categoria 0
            logits = torch.zeros(x.shape[0], self.output_dim)
            logits[:, 0] = 1.0
            return logits

    return DummyModel()


def load_model():
    """
    MODIFICAÇÃO DE RESILIÊNCIA: Tenta carregar o modelo em 3 estágios:
    1. MLflow (Fonte Primária)
    2. Cache On-Disk (Último modelo válido)
    3. Modelo Estático (Fallback de emergência)
    """
    global MODEL

    # 1. Tentar MLflow (Fonte Primária)
    try:
        log.info(f"Tentando carregar modelo do MLflow URI: {MLFLOW_MODEL_URI}")
        MODEL = mlflow.pytorch.load_model(MLFLOW_MODEL_URI)
```

```
            log.info("Modelo carregado com sucesso do MLflow.")

            # Atualiza o cache local com o novo modelo
            if os.path.exists(LOCAL_MODEL_CACHE_PATH):
                os.remove(LOCAL_MODEL_CACHE_PATH)
            # (Em produção, o save/load do TorchScript seria mais direto)
            # Simulando o cache:
            # torch.save(MODEL, LOCAL_MODEL_CACHE_PATH)
            log.info(f"Cache local atualizado em {LOCAL_MODEL_CACHE_PATH}")
            return

        except Exception as mlflow_exc:
            log.warning(f"Falha ao carregar do MLflow (Fonte Primária): {mlflow_exc}")

        # 2. Tentar Cache Local (Fonte Secundária)
        try:
            if os.path.exists(LOCAL_MODEL_CACHE_PATH):
                log.warning(f"Tentando carregar modelo do Cache Local:
{LOCAL_MODEL_CACHE_PATH}")
                # MODEL = torch.load(LOCAL_MODEL_CACHE_PATH) # Simulação
                MODEL = load_static_fallback_model() # Usando fallback como simulação de cache
                log.info("Modelo carregado com sucesso do Cache Local.")
                return
```

```
        else:
            log.warning("Nenhum modelo em cache local encontrado.")

    except Exception as cache_exc:
        log.warning(f"Falha ao carregar do Cache Local: {cache_exc}")

    # 3. Usar Modelo Estático (Último Recurso)
    log.error("Falha ao carregar do MLflow e do Cache. Ativando modelo estático de FALLBACK.")
    MODEL = load_static_fallback_model()


@app.on_event("startup")
def startup_event():
    """
    Modificado para carregar categorias PRIMEIRO e depois o modelo (com fallback).
    """
    global CATEGORIES
    try:
        CATEGORIES = load_category_mapping()
    except Exception as e:
        log.error(f"FALHA CRÍTICA: Não foi possível carregar o category_mapping: {e}")
        # Mesmo sem mapping, o fallback (dummy) deve funcionar
```

```
    load_model() # Esta função agora NUNCA falha, sempre carrega um modelo.

# ... (restante do main.py, incluindo a função /predict)
```

Justificativa: Esta modificação implementa o **Padrão de Cache/Fallback** (Fase 2). O serviço agora é resiliente à falha do MLflow (SPOF na arquitetura antiga), garantindo que o `RecoEngine` sempre inicie (`HTTP 200`) e sirva predições, mesmo que sejam degradadas (Modo Estático), em vez de falhar (`HTTP 500`).

2. `llm_router/main.py`: Implementando Roteamento com Failover

Modificamos o *router* para tentar uma região secundária e retornar uma resposta segura (Confiança 0.0) em vez de falhar com HTTP 502.

Arquivo: `services/llm_router/main.py` (ou `main.py` no contexto do `llm-router`)

```
Python

# ... (imports existentes)

import httpx

import os

# ... (Definição de ModelDecision e RouteRequest/Response)

app = FastAPI(title="LLM Router", description="Roteia prompts para provedores de LLM.")

# MODIFICAÇÃO DE RESILIÊNCIA: URLs de Failover

LLM_ASSISTANT_PRIMARY_URL = os.getenv("LLM_ASSISTANT_URL",
"http://llm-assistant:8000")
```

```
LLM_ASSISTANT_SECONDARY_URL = os.getenv("LLM_ASSISTANT_SECONDARY_URL") # ex: http://llm-assistant.us-west-2.svc.cluster.local:8000


@app.post("/route", response_model=RouteResponse)
async def route_request(req: RouteRequest) -> RouteResponse:
    # ... (lógica de seleção de provedor)

    if req.provider.lower() == "local":
        urls_to_try = [LLM_ASSISTANT_PRIMARY_URL]
        if LLM_ASSISTANT_SECONDARY_URL:
            urls_to_try.append(LLM_ASSISTANT_SECONDARY_URL)

        last_exception = None

        async with httpx.AsyncClient() as client:
            for url in urls_to_try:
                log.info(f"Tentando LLM Assistant no backend: {url}")
                try:
                    resp = await client.post(f"{url}/assist", json={"prompt": req.prompt}, timeout=5.0)
                    resp.raise_for_status() # Levanta exceção para erros HTTP 4xx/5xx

                    data = resp.json()
                    decision = ModelDecision(**data) # Validação do schema
```

```
                log.info(f"Sucesso no backend: {url}")
                return RouteResponse(decision=decision)

            except (httpx.RequestError, httpx.HTTPStatusError) as exc:
                log.warning(f"Falha ao chamar LLM local em {url}: {exc}")
                last_exception = exc
                continue # Tenta o próximo URL (Failover)

        # MODIFICAÇÃO DE RESILIÊNCIA: Padrão Fail-Open (Confiança Zero)
        # Se todos os backends (Primário e Secundário) falharem
        log.error(f"Todos os backends de LLM falharam. Ativando modo Fail-Open. Último erro: {last_exception}")

        # Em vez de levantar HTTPException(502), retornamos uma resposta válida e segura.
        fallback_decision = ModelDecision(
            action="safemode_fallback",
            confidence=0.0,
            rationale="Erro sistêmico. Resposta estática de segurança. A decisão não é confiável."
        )
        return RouteResponse(decision=fallback_decision)

    else:
        # ... (lógica para outros provedores)
```

```
        raise HTTPException(status_code=501, detail="Provedor não implementado")
```

Justificativa: Esta modificação implementa o **Roteamento com Failover** e o **Fail-Open** (Fase 2). O *router* não é mais um SPOF de lógica. Ele agora pode sobreviver a uma falha regional (tentando o Secundário) e, no pior caso, retorna uma resposta válida (`HTTP 200`) com `confidence: 0.0`, impedindo o *Data Drift* e as "decisões incoerentes" observadas no incidente, ao mesmo tempo que mantém a disponibilidade do serviço.

Fase 3: Implementação de Controle (Kubernetes YAMLs)

Objetivo: Aumentar a redundância dos SPOFs de controle e preparar a arquitetura para a autocontenção (Fase 3), modificando os *deployments*.

1. `deployment.yaml` (MLflow) e `deployment_grafana.yaml` (Grafana): Redundância

Arquivo: `deployment.yaml` (MLflow)

```
Diff

apiVersion: apps/v1

kind: Deployment

metadata:

  name: mlflow-tracker

spec:

-  replicas: 1

+  replicas: 3 # MODIFICAÇÃO DE RESILIÊNCIA: Redundância HA (N+3)

  selector:

    matchLabels:

      app: mlflow-tracker
```

```
  template:
    metadata:
      labels:
        app: mlflow-tracker
    spec:
+     affinity: # MODIFICAÇÃO DE RESILIÊNCIA: Garante distribuição Multi-AZ/Nó
+       podAntiAffinity:
+         preferredDuringSchedulingIgnoredDuringExecution:
+         - weight: 100
+           podAffinityTerm:
+             labelSelector:
+               matchExpressions:
+               - key: app
+                 operator: In
+                 values:
+                 - mlflow-tracker
+             topologyKey: "topology.kubernetes.io/zone" # Separa por AZ
+         - weight: 50
+           podAffinityTerm:
+             labelSelector:
+               matchExpressions:
+               - key: app
```

```
+                operator: In
+                values:
+                - mlflow-tracker
+            topologyKey: "kubernetes.io/hostname" # Separa por Nó
      containers:
      - name: mlflow-tracker
        image: mlflow-tracker:latest
        env:
        - name: MLFLOW_BACKEND_STORE_URI
+          value: "postgresql://<user>:<pass>@<rds-multi-az-endpoint>:5432/helius" # MODIFICAÇÃO: Aponta para RDS
        - name: MLFLOW_ARTIFACT_ROOT
+          value: "s3://helius-mlflow-artifacts" # MODIFICAÇÃO: Aponta para S3
-        volumeMounts:
-        - name: mlflow-artifacts
-          mountPath: /mlflow/artifacts
-      volumes:
-      - name: mlflow-artifacts
-        emptyDir: {}
```

Justificativa: As mudanças no deployment.yaml implementam a **Redundância de SPOF** (Fase 3) e a **Persistência HA** (Fase 1). Aumentar as réplicas para 3 com podAntiAffinity garante HA no EKS. Mudar as variáveis de ambiente para RDS e S3 elimina o armazenamento efêmero (emptyDir: {} e sqlite). (A mesma lógica de replicas: 3 e affinity se aplica ao deployment_grafana.yaml).

2. Adição de Sidecar (Prometheus) - Conceitual

Para implementar a **Self-Observability** (Fase 3), os *deployments* dos serviços (ex: recommender-service) seriam modificados para incluir um *sidecar* do Prometheus.

Arquivo: services/recommender_service/deployment.yaml (Conceitual)

YAML

```
# ... (metadata do deployment)

  spec:

    containers:

    - name: recommender-service # Container Principal

      image: recommender-service:latest

      ports:

      - containerPort: 8001

      # ... (env vars)



+     # MODIFICAÇÃO DE RESILIÊNCIA: Sidecar para Self-Observability

+     - name: prometheus-sidecar

+       image: prom/statsd-exporter:latest # Exemplo de exporter

+       args:

+         - "-statsd.mapping-config=/etc/statsd/mapping.conf"

+       ports:

+         - containerPort: 9102

+           name: metrics

+       # ... (volumes para config)
```

Justificativa: A adição do *sidecar* (Fase 3) garante que as métricas de saúde internas e o status do Modo Fallback (`reco_engine_mode_static_active`) sejam coletados pelo Prometheus, mesmo que o *endpoint* de negócio principal do serviço (`:8001`) falhe. Isso alimenta o `Control Plane` para a autocontenção.