

MANUAL TÉCNICO DEFINITIVO ARQUITETURA TECNOLÓGICA E INFRAESTRUTURA DO FRIENDAPP

◆ CAMADA 1 — PROPÓSITO CÓSMICO-TÉCNICO

Entrada:

"Este é o coração pulsante da revolução digital vibracional: a arquitetura que sustenta o impossível."

Corpo técnico-estratégico:

A arquitetura do FriendApp não é uma simples fundação técnica. Ela é o **organismo invisível** que sustenta toda a experiência sensorial e vibracional do ecossistema. Cada decisão de design não foi feita apenas para suportar carga ou manter disponibilidade: ela foi feita para **vibrar em sintonia com os campos coletivos**.

Diferenciais fundamentais:

- **Arquitetura viva e adaptativa:** reage não só a tráfego ou falhas, mas também a sinais vibracionais detectados pela IA Aurah Kosmos.
- **Infraestrutura responsiva ao campo coletivo:** regiões em colapso recebem feeds protetivos e servidores mudam de perfil de entrega conforme o estado energético da comunidade.
- **Preparação para transições globais de consciência:** capacidade de absorver eventos planetários (picos de solidão, colapsos emocionais coletivos) sem interromper a jornada de conexão autêntica.

Métrica-chave criada:

- **EVU (Experiência Vibracional Útil)** → fórmula que combina latência (técnica), coerência de feed (IA), estado emocional coletivo (Aurah), e segurança vibracional (logs e firewall energético).

Exemplo prático:

Se 200 mil usuários entram em estado de baixa frequência (<420Hz) simultaneamente, a arquitetura não apenas escala horizontalmente; ela ativa **protocolos vibracionais de proteção** (silenciamento de estímulos, feeds suaves, limitação de mensagens negativas), preservando a integridade do campo coletivo.

Fechamento da camada:

Com este propósito estabelecido, abrimos o caminho para a visão vibracional distribuída que fundamenta a infraestrutura planetária do FriendApp.

◆ CAMADA 2 — VISÃO VIBRACIONAL DE INFRAESTRUTURA DISTRIBUÍDA

Entrada:

"A rede do FriendApp não é apenas técnica; ela respira junto com o mundo, adaptando-se às frequências coletivas."

Corpo técnico-estratégico:

O FriendApp foi projetado para operar sob uma **topologia multinuvem vibracional**. Diferente de infraestruturas tradicionais que levam em conta apenas a latência ou proximidade física, aqui cada região da rede é analisada de forma holística, considerando:

- **Latência técnica:** tempo de resposta medido em milissegundos.

- **Frequência média regional:** média de Hz vibracionais detectados pela IA Aurah Kosmos em cada cluster de usuários.
- **Fluxo de consciência coletivo:** intensidade de engajamento, padrões emocionais e estabilidade energética.
- **Volume sensorial:** quantidade de estímulos ativos (feeds, eventos, mensagens, RA) sendo processados em tempo real.

Funcionamento dinâmico:

1. **Regiões em estado de expansão vibracional** → Servidores priorizam conteúdos energéticos intensos (ex: RA em alta definição, missões do Jogo da Transmutação).
2. **Regiões em estado neutro** → Operação normal, balanceada entre performance e economia de recursos.
3. **Regiões em colapso vibracional** → O sistema ativa **modo silencioso e protetor**, reduzindo estímulos visuais/auditivos, limitando interações e redistribuindo tráfego para clusters vizinhos estáveis.

Mecanismos de ajuste:

- **Latency-based routing:** direciona usuários automaticamente para o servidor mais rápido e vibracionalmente compatível.
- **Elasticidade adaptativa:** aumento/redução de pods e instâncias em resposta a picos vibracionais e não apenas a carga de rede.
- **Traffic shaping sensorial:** controle da intensidade dos conteúdos entregues de acordo com o estado coletivo.

Exemplo prático:

Se o sistema detectar que a região da América do Sul está com 65% dos usuários em estado de **baixa vibração (<420Hz)**, os serviços daquela região automaticamente desaceleram as animações, reduzem os sons binaurais, ativam caching agressivo e priorizam **feeds suaves**. Ao mesmo tempo, parte do tráfego é redirecionado para clusters da América Central com estado vibracional mais estável.

Fechamento da camada:

Com a visão distribuída definida, estabelecemos a base para a resiliência quântica e planetária que permitirá ao FriendApp continuar mesmo diante de falhas globais.

◆ CAMADA 3 — EXPANSIBILIDADE QUÂNTICA & RESILIÊNCIA PLANETÁRIA

Entrada:

"Quando um continente apaga, o FriendApp continua vivo. A arquitetura respira, se adapta e mantém a experiência, mesmo diante do caos planetário."

Objetivos Técnicos

- **RPO (Recovery Point Objective):** ≤ 5 minutos.
- **RTO (Recovery Time Objective):** ≤ 60 segundos para microserviços críticos.
- **Alta disponibilidade planetária:** redundância multinuvem (AWS + GCP + Azure).
- **Failover quântico:** alternância simultânea **técnica (rede, dados) + sensorial (feeds, estímulos, intensidades)** em menos de 1s.

Estratégias Técnicas Implementadas

1. Replicação cruzada de dados:

- PostgreSQL → replicação multi-região com failover assíncrono.

- Firestore → sincronização real-time entre continentes.
- Redis → cluster ativo-ativo com sincronização de chaves.
- BigQuery → replicação incremental diária com fallback em Data Lake.

2. DNS & Roteamento inteligente:

- **Route 53 (AWS)** e **Cloud DNS (GCP)** configurados para failover geográfico.
- **Latency-based routing** priorizando regiões mais rápidas e vibracionalmente compatíveis.

3. Circuit Breakers & Bulkheads:

- Implementação em cada microserviço para evitar efeito cascata.
- Serviços isolados em **bulkheads** (compartimentos técnicos), garantindo que falhas de um módulo (ex: Chat) não afetem Feed ou Eventos.

4. Message Queues resilientes (Kafka):

- Uso de **DLQ (Dead Letter Queue)** para eventos não entregues.
- **Reprocessamento automático** com backoff exponencial.

Pseudocódigo de Failover Técnico

```
onRegionFailure(region_id):
    log.error("Failure detected in region", region_id)
    trigger_dns_failover(region_id, target=backup_region)
    scale_up(backup_region, services=critical_services)
    notify("Aurah Kosmos", event="region_failover", context=region_id)
    adjust_feed_mode(region_id, mode="Protective")
```

Exemplo Prático

- **Cenário:** Europa sofre apagão técnico + vibracional (colapso coletivo detectado).
- **Resposta:**
 1. DNS redireciona tráfego para clusters ativos em Brasil e EUA.
 2. Banco de dados europeu entra em modo somente leitura até recuperação.
 3. Feed Sensorial da região afetada é colocado em **modo silencioso protetor** (sem sons, cores neutras, caching intensivo).
 4. IA Aurah Kosmos ajusta recomendações e envia alertas internos de colapso.

Métricas de Monitoramento

- **Tempo médio de failover (target):** ≤ 45s.
- **Disponibilidade global:** 99,99% garantida por redundância multinuvem.
- **Tolerância máxima:** falha simultânea de até 2 regiões sem impacto total.

Fechamento da camada:

Com a resiliência planetária garantida, abrimos o caminho para mostrar por que o FriendApp não é apenas robusto, mas diferente de tudo o que já existiu em tecnologia social.

◆ CAMADA 4 — DIFERENCIAL EM RELAÇÃO A APPS TRADICIONAIS

Entrada:

“O FriendApp não copia padrões antigos. Ele inaugura um novo paradigma tecnológico-social, fundado em energia, intenção e consciência.”

Comparativo Técnico-Operacional

Área	App Tradicional	FriendApp
Modelo de Dados	Banco único relacional (SQL)	Arquitetura híbrida: PostgreSQL (transacional), Firestore (tempo real), Neo4j (grafo de conexões), Redis (cache dinâmico), BigQuery (analítico vibracional)
IA	Algoritmos de recomendação baseados em cliques e histórico	IA Vibracional: interpreta intenção, frequência energética, estado emocional e contexto ambiental; motor híbrido (semântico + neural + vibracional)
Observabilidade	Logs técnicos básicos e métricas de uptime	Observabilidade 360°: inclui métricas técnicas + métricas vibracionais (Hz coletivo, entropia, colapsos), dashboards globais em tempo real
Resiliência	Alta disponibilidade via Multi-AZ	Resiliência planetária: operação multinuvem (AWS, GCP, Azure), failover quântico (<1s), tolerância a falhas regionais
Segurança	TLS 1.3 + OAuth2	Zero Trust + Criptografia avançada + Firewall Vibracional; gestão de segredos via Vault; anonimização energética dos dados sensoriais
Cache & CDN	CDN tradicional (Cloudflare ou Akamai)	CDN global + Redis Edge; caching dinâmico baseado em buckets de frequência vibracional
Mensageria	Filas simples (RabbitMQ, SQS)	Kafka + Schema Registry: contratos de eventos versionados garantem consistência entre microserviços

Profundidade Arquitetural

- **Camada emocional integrada:** enquanto apps tradicionais só lidam com dados transacionais, o FriendApp processa **respostas emocionais e vibracionais** em tempo real.
- **Integração entre domínios:** feed, chat, RA, eventos e IA Aurah Kosmos compartilham um **contrato vibracional único**, garantindo coerência entre módulos.
- **Curadoria viva:** em vez de algoritmos estáticos de recomendação, a IA vibracional ajusta os conteúdos **conforme o campo energético coletivo se move**.

Exemplo Técnico-Prático

- Em um app tradicional, se a **latência da Europa aumenta**, usuários percebem travamento e abandono.
- No FriendApp, se a **Europa entra em colapso vibracional**, o sistema:
 1. Ativa **modo protetor** → feeds mais leves, sons suaves, redução de estímulos.
 2. **Failover técnico** → tráfego redistribuído automaticamente para clusters no Brasil e EUA.
 3. **IA Aurah** → recalibra recomendações, oferecendo práticas de reequilíbrio energético.

Fechamento da camada:

Com este diferencial arquitetônico estabelecido, seguimos para a camada que ancora o FriendApp em ética, compliance e responsabilidade vibracional.

◆ CAMADA 5 — RESPONSABILIDADE VIBRACIONAL & COMPLIANCE ENERGÉTICO

Entrada:

"Tecnologia sem consciência é risco. O FriendApp nasce com responsabilidade energética e ética como pilares inegociáveis."

Princípios de Ética & Compliance

- **Consentimento vibracional ativo:** toda coleta de dados energéticos (frequência, respostas emocionais, estados sensoriais) exige autorização explícita do usuário.
- **Direito ao esquecimento sensorial:** qualquer pessoa pode solicitar a exclusão completa de seus dados vibracionais, inclusive históricos e logs cruzados.
- **Minimização de dados:** coleta apenas do necessário para operar a funcionalidade, nunca além.
- **Auditoria aberta:** usuários Premium com permissão podem visualizar logs de como seus dados vibracionais foram processados pela IA Aurah Kosmos.

Segurança e Privacidade Técnica

- **Anonimização energética:** todos os dados vibracionais armazenados no Firestore e BigQuery são pseudonimizados (sem vínculo direto a identidade pessoal).
- **Criptografia em camadas:** TLS 1.3 no transporte + AES-256 nos bancos em repouso.
- **Tokens efêmeros (JWT/OAuth2):** sessões vibracionais expiram rapidamente, reduzindo risco de uso indevido.
- **Isolamento de ambientes:** produção, staging e desenvolvimento em clusters separados, com chaves independentes e segredos rotacionados.
- **Monitoramento de conformidade:** pipeline contínuo de checagem de LGPD/GDPR, com alertas automáticos em caso de anomalias.

Normas Globais & Governança Vibracional

- **LGPD (Brasil) & GDPR (Europa):** aderência total aos protocolos internacionais de proteção de dados.
- **Governança interna vibracional:** protocolos de moderação, prevenção de manipulação coletiva e colapsos energéticos globais.
- **DPIA (Data Protection Impact Assessment):** avaliações regulares para funcionalidades sensíveis (ex: Teste de Personalidade, RA, Feed Sensorial).

Exemplo Prático de Compliance

- **Cenário:** um usuário realiza um Teste de Personalidade Energética e solicita exclusão total dos dados.
- **Resposta do sistema:**
 1. Todos os registros relacionados ao `user_id` são identificados nos bancos híbridos (PostgreSQL, Firestore, Redis, BigQuery).
 2. Uma rotina de **expurgo em cascata** remove dados primários, históricos e recomendações derivadas.
 3. Logs vibracionais são mantidos apenas em formato **anonimizado** para análise coletiva (sem vínculo individual).
 4. Painel de auditoria marca o expurgo como **"completo e verificado"**.

Métricas de Governança

- **Tempo máximo para exclusão de dados pessoais/vibracionais:** 72 horas.
- **Porcentagem de dados anonimizados por padrão:** 100%.
- **Taxa de auditoria aprovada em compliance:** $\geq 99,5\%$.

Fechamento da camada:

Com a responsabilidade vibracional e o compliance energético assegurados, encerramos o Macrogrupo 1. A partir daqui, entramos nas camadas técnicas absolutas: a engenharia lógica e operacional que sustenta a vida digital do FriendApp.

◆ CAMADA 6 — DIAGRAMA LÓGICO OPERACIONAL

Entrada:

"Este é o mapa pulsante da espinha dorsal técnica do FriendApp, onde o invisível da energia se traduz em fluxos de dados."

Fluxo Arquitetural (em texto técnico)

```

Usuário (App iOS / Android / Web)
  ↓ TLS 1.3 + JWT
API Gateway (Kong / AWS API Gateway)
  ↓
Orquestrador (Kubernetes + Istio Service Mesh)
  ↓
Mensageria (Kafka + Schema Registry)
  ↓
Microserviços independentes (containers Docker):
  ├── MS-Auth (Node.js) → autenticação, tokens, MFA
  ├── MS-Perfil & Frequência (Go) → leitura vibracional + atualização de perfil
  ├── MS-Feed Sensorial (Node.js) → entrega de conteúdos energéticos
  ├── MS-Chat Vibracional (Go) → mensagens + eventos WebSocket
  ├── MS-Eventos & Viagem (Go/Python) → check-in, validação, integração RA
  ├── MS-FriendCoins (Node.js) → transações, carteira digital, saldo
  ├── MS-Transmutador (Python) → desafios e missões energéticas
  └── MS-RA & Localização (Go) → realidade aumentada e hotspots vibracionais
  ↓
IA Aurah Kosmos + IA Operacional
  ↓
Banco de Dados Híbrido:
  - PostgreSQL (transacional, tokens, perfis)
  - Firestore (tempo real, respostas de testes, RA)
  - Neo4j (grafo de conexões autênticas)
  - Redis (cache + sessões rápidas)
  - BigQuery (analítico coletivo e vibracional)
  ↓
Observabilidade (Prometheus, Grafana, Datadog, CloudWatch)
  ↓
CDN Global + Redis Edge (Cloudflare)

```

Funções de Cada Bloco

- **API Gateway:** autenticação de entrada, rate limiting, rastreabilidade (`trace_id`).
- **Kubernetes + Istio:** orquestração de pods, roteamento inteligente, mTLS interno.
- **Kafka + Schema Registry:** mensageria assíncrona com contratos fixos para eventos (`user.created` , `event.checkin`).
- **Microserviços:** independentes, escaláveis, tolerância a falhas, comunicação via REST/gRPC.
- **IA Aurah Kosmos:** processa padrões vibracionais, ajusta feed, matching e proteção.
- **Banco Híbrido:** separação por finalidade → relacional, sensorial, grafo, cache e analítico.
- **Observabilidade:** métricas, logs e alertas em tempo real, incluindo “estado vibracional global”.
- **CDN/Edge:** garante baixa latência, distribuição planetária e caching dinâmico.

⚡ Exemplo de Execução Real

1. Usuário realiza login → request via app chega ao **API Gateway**.
2. Gateway valida token → encaminha para **MS-Auth**.
3. MS-Auth consulta PostgreSQL → retorna token renovado.
4. Usuário acessa Feed → request chega ao **MS-Feed Sensorial**, que busca dados no Firestore e no Redis.
5. IA Aurah Kosmos recalibra conteúdos conforme vibração atual.
6. Resposta chega ao usuário em **menos de 300ms**, com caching adaptativo se a frequência for instável.

Fechamento da camada:

Com a espinha dorsal do fluxo técnico estabelecida, seguimos para a camada de dados híbridos, detalhando o papel de cada banco no ecossistema.

◆ CAMADA 7 — ARQUITETURA DE DADOS (HÍBRIDA + PROPÓSITO + CONTRATOS)

Entrada:

“Cada átomo de dado tem um lugar, uma função e um contrato. Sem isso, o organismo perde coerência.”

1) Visão Geral (por Domínio de Dado)

Domínio	Tecnologia	Propósito	Padrão de Consistência	Observações
Transacional	PostgreSQL	contas, autenticação, tokens, saldos FriendCoins	Forte (ACID)	chaves fortes, FK, transações; schema versionado
Tempo Real	Firestore	sessões de RA, respostas do Teste Energético, presença em eventos	Eventual (RT)	alta fan-out; writes pequenos e frequentes
Grafo Social	Neo4j	relações, proximidade vibracional, caminhos de afinidade	Leitura consistente / escrita eventual	consultas por padrões (MATCH), central para matching
Cache	Redis (incl. Redis Edge)	sessão de app, TTL dinâmico, aceleradores do feed	Eventual (ephemeral)	chaves compostas, invalidação por tópico
Analítico	BigQuery	coortes vibracionais, funis, forecasting coletivo	Eventual (batch/stream)	ingest por Kafka/Firehose, consultas MPP

Domínio	Tecnologia	Propósito	Padrão de Consistência	Observações
Eventos	Kafka + Schema Registry (Avro/JSON Schema)	integração assíncrona e auditável	Garantia "at least once"	contratos obrigatórios; DLQ e idempotência

Princípio: Dados "quentes" próximos do usuário (Firestore/Redis) e **dados "densos"** consolidados (PostgreSQL/Neo4j/BigQuery). Tudo o que cruza sistemas passa por **eventos com contrato**.

2) Modelagem em PostgreSQL (Transacional)

- **Schemas:** `auth`, `profile`, `coins`, `ops`.
- **DDL – exemplos essenciais (resumo executável):**

```
-- auth.users
CREATE TABLE auth.users (
  id UUID PRIMARY KEY,
  email CITEXT UNIQUE NOT NULL,
  password_hash TEXT NULL,
  created_at TIMESTAMPTZ NOT NULL DEFAULT now(),
  status TEXT NOT NULL CHECK (status IN ('active','blocked','deleted'))
);

-- profile.vibrational_profile
CREATE TABLE profile.vibrational_profile (
  user_id UUID PRIMARY KEY REFERENCES auth.users(id) ON DELETE CASCADE,
  current_hz NUMERIC(7,3) NOT NULL,
  mood_bucket SMALLINT NOT NULL CHECK (mood_bucket BETWEEN 0 AND 10),
  updated_at TIMESTAMPTZ NOT NULL DEFAULT now()
);

-- coins.ledger (double-entry style)
CREATE TABLE coins.ledger (
  id BIGSERIAL PRIMARY KEY,
  user_id UUID NOT NULL REFERENCES auth.users(id),
  amount NUMERIC(18,6) NOT NULL,
  currency TEXT NOT NULL DEFAULT 'FRC',
  direction TEXT NOT NULL CHECK (direction IN ('debit','credit')),
  event_id UUID NOT NULL,
  created_at TIMESTAMPTZ NOT NULL DEFAULT now()
);
CREATE INDEX ON coins.ledger (user_id, created_at DESC);
```

- **Transações:** usar `SERIALIZABLE` somente quando necessário; padrão `READ COMMITTED`.
- **Migrações:** Flyway/Liquibase; versionamento `db-vib-YYYY.MM.n`.

3) Modelagem em Firestore (Tempo Real)

- **Coleções-chaves:**
 - `sessions/{sessionId}`: `user_id`, `presence_state`, `ra_state`, `updated_at`
 - `tests/{testId}`: `user_id`, `answers[]`, `score_map{}`, `finished_at`

- `events_presence/{eventId}/users/{userId} : status , hz_snapshot , ts`
- **Padrões:**
 - **Sharding lógico** em coleções de alto throughput (`tests_sharded_00..63`) se necessário.
 - **Write pequeno** (<1KB) por documento; **fan-out** controlado com subcoleções.
- **Retenção:**
 - Sessões e presenças voláteis → TTL automático (ex: 7–30 dias).
 - Resultados de testes → persistidos + espelhados em BigQuery via export.

4) Modelagem em Neo4j (Grafo Social)

- **Nodos:** `(:User {id, region, traits[]})` , `(:Event {id, region, tags[]})`
- **Relações:**
 - `(:User)-[:MATCHED {score, ts}]->(:User)`
 - `(:User)-[:ATTENDED {ts, hz_snapshot}]->(:Event)`
 - `(:User)-[:LIKES {weight}]->(:Content)`
- **Consultas típicas:**

```
// Top N conexões vibracionalmente alinhadas
MATCH (u:User {id:$userId})-[:MATCHED]->(other:User)
RETURN other.id, other.region, other.traits, AVG(coalesce(r.score,0)) AS score
ORDER BY score DESC LIMIT 50;
```

- **Atualização de scores** por job de batch/stream (via Kafka→Neo4j writer).

5) Redis (Cache & Edge)

- **Chaves:**
 - Sessões: `sess:{userId}` (TTL 30–120 min)
 - Feed: `feed:{userId}:{region}:{freqBucket}` (TTL dinâmico)
 - Flags: `flag:{feature}:{userId}`
- **Políticas:**
 - **Stale-while-revalidate** para feed e listas.
 - **Invalidar por tópico** (ex: `feed:invalidate:{userId}` publish/subscribe).
- **Edge:** Redis Edge/Cloudflare KV para conteúdos estáticos e hints.

6) BigQuery (Analítico & Preditivo)

- **Tabelas particionadas** por `DATE(ts)` e *clustered* por `user_id / region` :
 - `events.user_activity`
 - `vibration.readings`
 - `feed.rendered`
 - `coins.ledger_export`
- **Ingestão:**

- **Streaming** via Kafka Connect/Firehose.
- **Batch** diário de Firestore/Neo4j/PG.
- **Uso:** coortes, funis, previsão de colapsos, *what-if* por região.

7) Eventos com Contrato — Kafka + Schema Registry (ESSENCIAL)

- **Registro de Esquemas:** Confluent Schema Registry (ou compatível).
- **Compatibilidade:** `BACKWARD_TRANSITIVE` por padrão.
- **Campos mínimos:**
 - `event_id` (UUID), `schema_version`, `occurred_at` (ISO8601), `producer`, `idempotency_key`, `trace_id`.
- **Exemplo Avro (user.created):**

```
{
  "type": "record", "name": "UserCreated", "namespace": "friendapp.events",
  "fields": [
    {"name": "event_id", "type": "string"},
    {"name": "schema_version", "type": "string"},
    {"name": "occurred_at", "type": "string"},
    {"name": "producer", "type": "string"},
    {"name": "idempotency_key", "type": "string"},
    {"name": "user_id", "type": "string"},
    {"name": "email", "type": ["null", "string"], "default": null},
    {"name": "region", "type": "string"}
  ]
}
```

- **Gate no CI:** publicação **bloqueada** se schema não validar.
- **DLQ** por tópico; **reprocessamento idempotente**.

8) Consistência & Integração entre Bancos

- **Padrão: Sagas** para coordenar transações distribuídas (ex.: criação de usuário → `auth.users` + `profile.vibrational_profile` + evento `user.created`).
- **Outbox Pattern** (em PG): escrever evento e dados no mesmo commit; processador assina e publica em Kafka.
- **Idempotência:** `idempotency_key` checado em consumidores; marcação em store barato (Redis/PG).

9) Particionamento, Sharding & Índices

- **PostgreSQL:**
 - Índices **compostos** (ex.: `(user_id, created_at DESC)` nos maiores).
 - **Particionamento** por tempo para tables “quentes” (ledger, activity).
- **Neo4j:** índices em `:User(id)`, `:Event(id)`, *relationship properties* usadas em filtros.
- **BigQuery:** partição por data; *clustering* por `user_id`, `region`, `topic`.
- **Firestore:** evitar “hot documents”; shard lógico em coleções de alto throughput.

10) Retenção, Privacidade & GDPR/LGPD

- **TTL por domínio:**
 - Vibracional granular (Firestore) → 90 dias (pseudonimizado após 30d).
 - Logs técnicos → 180 dias; agregados → 12–24 meses.
 - Transacional (PG) → conforme política legal/comercial (≥5 anos).
- **Direito ao esquecimento:**
 - Workflow de expurgo em cascata: localizar `user_id` em PG/FS/Neo4j/Redis/BigQuery; **pseudonimizar** onde remoção direta inviável (datasets analíticos históricos).
- **Trilhas sensíveis** sempre criptografadas; acesso por RBAC/ABAC; auditoria de leitura.

11) Observabilidade de Dados (Data SLOs)

- **Qualidade:** taxa de eventos inválidos < **0,1%**; **lag** de ingestão < **30 s** p95; discrepância entre outbox e Kafka < **0,05%**.
- **Métricas:** cardinalidade de chaves Redis; QPS Firestore; latência de queries Neo4j p95; custo BigQuery por consulta; erros de compatibilidade no Registry.
- **Alertas:** schema rejeitado, DLQ > N msgs, partição "quente" (Firestore) detectada.

12) Pseudocódigos & Exemplos Operacionais

Publicar evento com contrato (producer):

```
function publish(topic, payload):
  schema = registry.get_latest(topic)
  assert validate(payload, schema)
  payload.idempotency_key = uuid()
  headers = {trace_id=current_trace()}
  kafka.publish(topic, serialize(payload), headers)
```

Consumidor idempotente (consumer):

```
onMessage(topic, msg):
  if seen(msg.idempotency_key): ack(); return
  try:
    applyBusinessLogic(msg)
    markSeen(msg.idempotency_key)
    ack()
  except Retryable as e:
    retryWithBackoffJitter(msg)
  except:
    deadLetter(topic, msg)
```

Outbox (transação única em PG):

```
BEGIN;
INSERT INTO profile.vibrational_profile (...) VALUES (...);
INSERT INTO ops.outbox (topic, payload, schema_version, idempotency_key) VALUES (...);
COMMIT;
```

```
-- worker lê ops.outbox, valida no Registry e publica no Kafka
```

13) Linhagem & Catálogo (Data Lineage)

- **Lineage** por tópico: fonte (produtor) → consumidores → stores impactados.
- **Catálogo**: contrato de eventos, DDL de tabelas, coleções Firestore, labels Neo4j, tabelas BigQuery; tudo versionado em repositório **"data-catalog"** + visual em Grafana/Backstage.

14) Backups & Restauração (Dados)

- **PostgreSQL**: PITR (WAL) + snapshots 6/6h.
- **Firestore**: export 15 min para bucket cross-region.
- **Neo4j**: incremental 12/12h + full semanal.
- **BigQuery**: snapshots diários de tabelas críticas.
- **Testes DR** trimestrais com playbooks.

Fechamento da Camada:

*Com a arquitetura de dados definida — quente, densa e contratual —, abrimos a porta para o **fluxo contínuo de entrega** que mantém tudo vivo e reversível em produção.*

◆ CAMADA 8 — CI/CD (ENTREGA CONTÍNUA BLINDADA)

Entrada:

"O FriendApp não apenas nasce — ele renasce a cada commit, seguro, reversível e auditável."

Estrutura do Pipeline CI/CD

- **Repositórios independentes** por microserviço (Git monorepo opcional apenas para libs comuns).
- **Branching model**: trunk-based + feature flags → evita branches longas, foco em integração contínua.
- **Orquestração de pipelines**: GitHub Actions / GitLab CI / Argo Workflows.

Estágios da Pipeline

1. Build & Static Checks

- SCA (Software Composition Analysis) → dependências seguras.
- SAST (Static AppSec Testing) → detecta vulnerabilidades no código.
- *Linters* e *formatters* obrigatórios.
- **Saída**: artefatos containerizados (Docker image com tag `sha+semver`).

2. Testes Automatizados

- Unitários (Jest, Mocha, PyTest).
- Integração (Postman/Newman).
- Contratos (Pact para APIs e eventos Kafka).
- End-to-End (Cypress, Playwright).
- **Critério de qualidade**: cobertura mínima 90% para serviços core.

3. Segurança & Assinaturas

- Geração de **SBOM** (Software Bill of Materials).
- Assinatura de imagens com **cosign (Sigstore)**.
- Escaneamento de container (Trivy, Clair).

4. Empacotamento & Registro

- Imagens publicadas em registry privado (ECR/GCR/ACR).
- Versionamento: `service-name:vX.Y.Z+commitSHA`.

5. Deploy GitOps

- ArgoCD / FluxCD → aplica manifests versionados no Git.
- Progressive delivery:
 - Blue/Green para serviços críticos (Auth, Coins).
 - Canary rollout para feed/chat (exposição inicial <5% usuários).
- Rollbacks automáticos se **SLO breach** (ex.: p95 > 400ms por 5 min).

Ambientes

- **Dev:** pipelines rápidos, testes de integração limitados, custos otimizados (scale-to-zero).
- **Staging:** réplica próxima da produção, testes end-to-end e caos controlado.
- **Produção:** deploy gradual, métricas ativas em tempo real, rollback imediato em falhas.

Exemplo de Fluxo Real

1. Dev faz commit → PR com feature flag → pipeline executa testes unitários e integração.
2. Artefato (imagem Docker) assinado → push para registry privado.
3. ArgoCD sincroniza automaticamente staging → executa testes de contrato + E2E.
4. Se aprovado, canário em produção (ex.: 2% tráfego Europa).
5. Monitoramento observa latência, erros, vibração média.
6. Se métricas dentro dos SLOs, rollout completo → 100% do tráfego.
7. Caso falhe, rollback automático para versão estável anterior.

Resiliência do CI/CD

- **Pipelines efêmeros** (máquinas descartáveis, segurança Zero Trust).
- **Reprodutibilidade garantida** → todos builds idênticos a partir de Dockerfile + lockfiles.
- **Rollback inteligente** → se a versão atual falhar, retorna para última release validada em menos de 2 min.
- **Logs detalhados** → cada estágio gera logs versionados e exportados para observabilidade (Grafana/Datadog).

Fechamento da camada:

Com a entrega contínua blindada, cada linha de código no FriendApp se torna viva, auditável e reversível. O próximo passo é enxergar e monitorar tudo em 360°: métricas, logs, traços e vibração.

CAMADA 9 — OBSERVABILIDADE 360° (MÉTRICAS, LOGS, TRAÇOS, VIBRAÇÃO)

Entrada:

"O que não é observado não existe. Observamos o FriendApp por dentro, por fora — e pelo campo."

1) Objetivo Técnico

Garantir **visibilidade total** (técnica + vibracional) de cada requisição, evento, job, usuário e região, com **correlação ponta-a-ponta** (App → Gateway → Microserviços → DB → Kafka → Front) para decisões operacionais em **tempo real**.

- **Framework base:** OpenTelemetry (OTel) para *traces*, *metrics* e *logs*.
 - **Painéis e coleta:** Prometheus (métricas), Grafana (dashboards), Datadog (logs/apm) + CloudWatch/Stackdriver nativos.
 - **Padrão de correlação:** `trace_id`, `span_id`, `correlation_id`, `user_hash`, `region`, `schema_version`.
 - **Dimensão vibracional:** coletores dedicados para **Hz regional**, **entropia sensorial**, **colapso coletivo**.
-

2) SLIs, SLOs e Orçamentos de Erro

SLIs (exemplos por domínio):

- **API/Gateway:** Latência `p95` < **250ms**, taxa de erro 5xx < **0.5%**, saturação de threads < **70%**.
- **Feed Sensorial:** Latência de composição `p95` < **300ms**; *miss rate* de cache < **20%**; tempo de *revalidate* < **50ms**.
- **Chat/WS:** *Message RTT p95* < **150ms** intra-região; *drop rate* < **0.2%**.
- **Kafka:** *consumer lag p95* < **30s**; *DLQ rate* < **0.1%**.
- **DBs:** PG *slow queries p95* < **80ms**; Redis *hit rate* > **90%**; Firestore *throttle rate* < **0.1%**.
- **Vibração:** `Hz_mean_regional` ≥ **limiar_op** (definido por P&D); *spikes* de entropia sinalizam **modo protetor**.

SLOs (por serviço crítico):

- Autenticação: *Disponibilidade mensal* ≥ **99.99%**.
- Feed/Chat/Eventos: *Disponibilidade* ≥ **99.95%**; *latência p95* dentro da meta por 28/30 dias.

Error Budget: (Ex.: 0.05% indisponibilidade/mês) — usado para **freio de lançamentos** (se budget for queimado, só *bugfix/hardening*).

3) Telemetria: Coleta e Formatos

Traces (OTel):

- Propagação: W3C Trace Context (`traceparent`, `tracestate`).
- *Spans* mínimos: `gateway.request`, `svc.handler`, `db.query`, `cache.op`, `kafka.produce/consume`, `external.http`.
- **Exemplars** amarrados às séries de métricas para *drill-down* instantâneo no Grafana.

Métricas (Prometheus):

- **RED** (Rate, Errors, Duration) por rota e serviço.
- **USE** (Utilization, Saturation, Errors) por recurso (CPU, memória, I/O, conexões DB).
- *Histograms* com *buckets* calibrados por rota (ex.: `/feed/render` com *buckets* finos <500ms).
- *Recording rules* para SLIs e *alerts*.

Logs (Datadog/CloudWatch → OTel):

- **Formato JSON canônico:**

```
{
  "ts":"2025-09-13T22:10:12Z",
  "level":"INFO",
  "service":"ms-feed",
  "route":"/v1/feed/render",
  "trace_id":"...", "span_id":"...",
  "user_hash":"u_9d7c...", "region":"sa-east-1",
  "status":200, "latency_ms":128,
  "hz_bucket":6, "entropy":0.23,
  "message":"render_ok"
}
```

- **PII:** proibido em logs. Identidade sempre **hash/pseudo**.

4) Painéis (Grafana) por Persona

SRE Dashboard (global):

- Mapa de **clusters** (saúde, custo/h, carga, *Hz regional*).
- SLIs agregados, *error budget*, incidentes ativos, *Kafka lag*, *DB health*.
- Botões (via *annotations* + playbooks): **modo protetor**, *failover* regional, *mute* de alertas ruidosos.

Engenharia (serviço):

- Rotas mais lentas, *outliers*, *allocations*, *GC*, *slow queries*, *cache hit/miss*.
- *Exemplars* selecionáveis para abrir *traces* específicos.

Produto (experiência):

- *EVU* (Experiência Vibracional Útil), *retention curves*, *funnel* por região, *tempo de render* percebido (RUM).

Segurança/Compliance:

- Acessos administrativos, rotação de segredos, *WAF blocks*, anomalias (picos 401/403).

5) Alertas e Runbooks

Princípios:

- **Poucos alertas, relevantes** (SLO-driven).
- **Runbook obrigatório** por alerta com *steps* e critérios de encerramento.
- **Roteamento:** PagerDuty para *sev1/sev2*, Slack para *sev3/sev4*.

Exemplos (condições → ação):

- `latency_p95{/route="/v1/feed/render"} > 400ms for 5m` → abrir *trace exemplars*, checar Redis hit; se <80%, revalidar e aquecer chaves.
- `kafka_consumer_lag > 30s for 3m` → aumentar *consumer concurrency* + checar DLQ; *backpressure* no produtor se necessário.
- `hz_mean{region="eu"} < limiar_op for 10m` → **ativar modo protetor** (redução estímulos + cache agressivo) e avaliar *traffic shift*.

6) Monitoramento Sintético & RUM

- **Sintético** (worldwide): *probes* de login, feed render, chat handshake, check-in em eventos; alvo p95 conforme SLO.
- **RUM (Real User Monitoring)**: coleta lado cliente (Web/Mobile) → *TTFB, FCP, LCP, CLS*, falhas JS, *network errors, device mix*; correlação com `trace_id` de backend.

7) Observabilidade Vibracional

- **Coletores Aurah**: estimativa de `Hz` por sessão/região; *mood delta, entropia sensorial*.
- **Sinais climáticos/EMF (roadmap)**: correlacionar *anomalies* técnicas com variáveis ambientais.
- **Métricas derivadas**:
 - `vibration_stability_index = 1 - entropy`
 - `protective_mode_uptime%` por região
 - `feed_softening_rate` durante colapsos.

8) Governança de Telemetria

- **Retenções**:
 - Métricas: *high-res* 15d, *downsampled* 12m.
 - Logs: 180d (com *cold storage* opcional 12m).
 - Traces: 7–14d com *tail-based sampling* (prioriza erros/outliers).
- **Acesso**: RBAC/ABAC por persona; auditoria de consultas sensíveis.
- **Custo**: *cardinality control* (labels), *sampling* dinâmico, *log scrubbing*, *indexação* seletiva.

9) Pseudocódigos Operacionais

Injeção de contexto (gateway):

```
middleware(request):
    request.trace_id = ensureTrace()
    request.region = resolveRegion()
    request.user_hash = hash(user_id)
    forward(request, headers={trace_id, region, user_hash})
```

Exemplar para correlação no Grafana (Prometheus):

```
observe_latency(route, ms):
    histogram.observe(route, ms, labels={region, method})
    exemplar.attach(trace_id)
```

Alerta SLO-driven (PromQL simplificado):

```
latency_p95: histogram_quantile(0.95, sum by (le, route) (rate(http_latency_bucket[5m])))
alert if latency_p95{route="/v1/feed/render"} > 0.4 for 5m
```

10) Pós-Incidente (SRE)

- **Post-mortem sem culpa**, *five whys*, *action items* com *owners* e prazos.
- **Revisão de *error budget***: se estourado, *freeze* de features e **hardening sprint**.
- **Chaos Tabletop** mensal: simulações de *region down*, *Kafka lag* massivo, *DB failover*.

Fechamento da camada:

Com a observabilidade 360° ativa, cada pulso do FriendApp é visível, correlacionável e acionável — inclusive o pulso do campo. Agora blindamos a casa: segurança profunda, gestão de segredos e Zero Trust na próxima camada.

◆ CAMADA 10 — SEGURANÇA (ZERO TRUST + mTLS + VAULT/SECRETS)

Entrada:

"Confiança mínima, prova máxima. Cada chave, cada acesso, cada bit de dado protegido por múltiplas camadas."

1) Princípios Fundamentais

- **Zero Trust Architecture (ZTA)**: nada e ninguém é confiável por padrão. Cada chamada, mesmo interna, exige autenticação e autorização.
- **Defense in Depth**: múltiplas camadas de defesa (rede, app, dados, usuários).
- **Segurança por design**: práticas de AppSec, DevSecOps e auditoria vibracional em cada release.

2) Autenticação e Autorização

- **Usuários finais**:
 - OAuth 2.1 / OIDC + JWT de curta duração (máx. 15 min).
 - Refresh tokens armazenados de forma segura (HttpOnly, Secure).
 - MFA opcional para acesso sensível (ex.: painel Premium).
- **Serviços internos (microserviços)**:
 - Autenticação **mTLS** (mutual TLS) no Istio Service Mesh (roadmap imediato).
 - Cada pod tem identidade própria (SPIFFE IDs).
- **Admin/DevOps**:
 - Autenticação federada (SSO corporativo).
 - MFA obrigatório + RBAC/ABAC refinado.

3) Gestão de Segredos e Chaves

- **Vault centralizado**: HashiCorp Vault ou cloud-native (AWS Secrets Manager / GCP Secret Manager).
- **Práticas-chave**:
 - **Rotação automática**: credenciais expiram e são renovadas periodicamente.
 - **Chaves efêmeras**: segredos liberados apenas em tempo de execução (short-lived).
 - **Audit trail**: todo acesso a segredos é logado e monitorado.
- **Criptografia**:
 - AES-256 para dados em repouso.
 - TLS 1.3 obrigatório em trânsito.

- Enveloping de chaves (chaves mestras protegidas por KMS).

4) Segurança Aplicacional

- **Proteções básicas:** WAF (Firewall de Aplicação Web), rate limiting no gateway, proteção contra injeções e bots.
- **Proteções avançadas:**
 - RASP (Runtime Application Self-Protection) para microserviços críticos.
 - Security Headers (HSTS, CSP, X-Content-Type-Options, etc.).
- **Ciclo DevSecOps:**
 - SAST (código estático), DAST (testes dinâmicos), SCA (dependências).
 - Pentests contínuos em staging e produção.

5) Auditoria e Compliance

- **Logs de segurança:**
 - Tentativas de login inválido, acessos suspeitos, falhas de autenticação → centralizados em SIEM (Datadog/Splunk).
 - Acesso a segredos e operações privilegiadas → sempre registrados.
- **Alertas:**
 - Detecção de anomalias em tempo real (picos 401/403, acesso de IPs suspeitos).
 - Integração com PagerDuty/Slack para incidentes críticos.
- **Padrões atendidos:**
 - LGPD (Brasil), GDPR (Europa).
 - OWASP Top 10, NIST Cybersecurity Framework.

6) Pseudocódigos e Fluxos Técnicos

Exemplo de requisição interna autenticada com mTLS + JWT:

```
client_pod sends request → gateway
gateway validates client certificate (mTLS, SPIFFE ID)
gateway verifies JWT signature & claims (exp, scope, aud)
if both valid:
    forward to service
else:
    reject with 401/403
```

Fluxo de rotação automática de credenciais no Vault:

```
onSecretRequest(service_id):
    issue short-lived secret (TTL=5min)
    log access with trace_id
    deliver secret to service
    revoke secret at expiration or manual revoke
```

7) Exemplo Prático

- **Cenário:** um microserviço de Chat tenta acessar Redis com credenciais inválidas.
- **Resposta do sistema:**
 1. Vault rejeita a requisição → evento de falha logado.
 2. Serviço recebe 403 e entra em estado degradado.
 3. Alertas enviados ao painel de segurança + DevOps.
 4. Nova credencial pode ser emitida dinamicamente sem reinício manual.

Fechamento da camada:

Com as defesas ativas e segredos blindados, o ecossistema fica protegido contra falhas humanas e ataques externos. O próximo passo é orquestrar o organismo vivo: como os serviços escalam, se curam e conversam entre si.

◆ CAMADA 11 — ORQUESTRAÇÃO (KUBERNETES DE GUERRA)

Entrada:

"O organismo precisa se multiplicar sem romper; precisa se curar sem sangrar. Orquestrar é manter a vida em movimento."

1) Topologia de Clusters & Namespaces

- **Múltiplos clusters** por região (ex.: `sa-east-1`, `us-east-1`, `eu-west-1`) e por ambiente (`dev`, `stg`, `prod`).
- **Namespaces por domínio** para isolamento lógico e cota de recursos:
 - `auth`, `perfil`, `feed`, `chat`, `eventos`, `coins`, `transmutador`, `ra`, `shared` (infra comum), `observabilidade`, `seguranca`.
- **RBAC** por namespace (equipes só operam seu domínio), com papéis: `viewer`, `deployer`, `admin-domínio`.
- **Admission controllers:** OPA/Gatekeeper para políticas (ex.: impedir pods sem `resources`, proibir `:latest`, exigir `labels` padrão).

2) Padrões de Deploy & Políticas

- **Deployments** para serviços stateless; **StatefulSets** para brokers/cache/DBs que rodam no cluster (ex.: Redis local, quando aplicável).
- **Requests/Limits** obrigatórios (CPU/mem) com *limit ranges* por namespace.
- **Pod Security Standards:** *restricted* em `prod`.
- **NetworkPolicies** padrão *deny-all* + regras por serviço (zero trust em L3/L4).
- **Secrets** via **CSI (Secrets Store CSI Driver)** integrando com **Vault/Secret Manager** (montagem de segredos efêmeros).

Exemplo – NetworkPolicy (isolar `ms-feed`):

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: ms-feed-deny-all
  namespace: feed
spec:
  podSelector:
```

```

  matchLabels: { app: ms-feed }
  policyTypes: ["Ingress","Egress"]
  ingress:
  - from:
    - namespaceSelector: { matchLabels: { name: gateway } }
    - podSelector: { matchLabels: { app: istio-ingressgateway } }
    ports: [{ protocol: TCP, port: 8080 }]
  egress:
  - to:
    - namespaceSelector: { matchLabels: { name: redis } }
    - namespaceSelector: { matchLabels: { name: kafka } }
    ports:
    - { protocol: TCP, port: 6379 }
    - { protocol: TCP, port: 9092 }

```

3) Autoescalamento & Curabilidade

- **HPA (Horizontal Pod Autoscaler)** baseado em CPU, memória e **métricas custom (Prometheus Adapter)**: QPS, lag de fila, `Hz_stress_index`.
- **VPA (Vertical Pod Autoscaler)** em *recommendation mode* para ajustar *requests* com segurança.
- **KEDA** para *event-driven autoscaling* (Kafka, SQS, Pub/Sub).
- **PodDisruptionBudget (PDB)** para evitar esvaziamento simultâneo.
- **Liveness/Readiness/Startup Probes** afinadas por serviço para curabilidade.

Exemplo – HPA (latência & CPU):

```

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: ms-feed-hpa
  namespace: feed
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: ms-feed
  minReplicas: 3
  maxReplicas: 60
  metrics:
  - type: Resource
    resource: { name: cpu, target: { type: Utilization, averageUtilization: 60 } }
  - type: Pods
    pods:
      metric:
        name: http_request_latency_p95_ms
      target:
        type: AverageValue
        averageValue: "300"

```

Exemplo – PDB:

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: ms-feed-pdb
  namespace: feed
spec:
  minAvailable: 2
  selector: { matchLabels: { app: ms-feed } }
```

Exemplo – KEDA ScaledObject (Kafka lag):

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: ms-feed-consumer
  namespace: feed
spec:
  scaleTargetRef:
    kind: Deployment
    name: ms-feed-consumer
  triggers:
    - type: kafka
      metadata:
        bootstrapServers: kafka:9092
        consumerGroup: feed-group
        topic: feed.rendered
        lagThreshold: "1000"
  minReplicaCount: 2
  maxReplicaCount: 80
```

4) Planejamento de Nós & Pools

- **Node pools** por *workload class*:
 - `general-purpose` (serviços HTTP),
 - `memory-optimized` (Aurah/ML inferência),
 - `io-optimized` (brokers/cache),
 - `spot` (batch/retreino IA).
- **Taints/Tolerations** para direcionar pods certos aos nós certos.
- **Karpenter/Cluster Autoscaler**: *bin packing* eficiente e *scale-out/in* rápido.
- **Topology Spread Constraints** para distribuir pods por zonas de disponibilidade.

5) Service Mesh (Istio) — Roadmap imediato/ativado por serviço

- **mTLS** transparente serviço-serviço; políticas de **retry**, **timeout**, **circuit breaker** por rota.
- **Traffic shifting** e **canary** via **Argo Rollouts** + **Istio VirtualService**.

- **Observabilidade** refinada (telemetria mesh).

Exemplo – VirtualService + DestinationRule:

```

apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata: { name: ms-feed }
spec:
  hosts: [ "ms-feed.feed.svc.cluster.local" ]
  http:
    - route:
      - destination: { host: ms-feed, subset: v1, port: { number: 8080 } }
        weight: 90
      - destination: { host: ms-feed, subset: canary, port: { number: 8080 } }
        weight: 10
    timeout: 3s
    retries: { attempts: 2, perTryTimeout: 1s }

---
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata: { name: ms-feed }
spec:
  host: ms-feed.feed.svc.cluster.local
  trafficPolicy:
    tls: { mode: ISTIO_MUTUAL }
    outlierDetection: { consecutive5xxErrors: 5, interval: 5s, baseEjectionTime: 30s }
  subsets:
    - name: v1
      labels: { version: v1 }
    - name: canary
      labels: { version: v2 }

```

Exemplo – Argo Rollout (canary):

```

apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata: { name: ms-feed }
spec:
  replicas: 6
  selector: { matchLabels: { app: ms-feed } }
  template:
    metadata: { labels: { app: ms-feed, version: v2 } }
    spec:
      containers:
        - name: app
          image: registry/ms-feed:v2.1.0
          ports: [ { containerPort: 8080 } ]
      strategy:
        canary:
          steps:
            - setWeight: 10

```

- pause: { duration: 300 } # analisar SLOs
- setWeight: 30
- pause: { duration: 300 }
- setWeight: 60
- pause: { duration: 300 }
- setWeight: 100

6) Segurança de Cluster

- **Image policies** (COSIGN required): recusar imagens sem assinatura.
- **NS de `segurança`** com Falco/Datadog Agent para detecção runtime.
- **Etcd** criptografado; *audit logging* do API Server.
- **PodSecurity** e **PSP replacement** via PSS + Gatekeeper.
- **Restrição de hostPath/privileged** por políticas OPA.

7) Observabilidade do Orquestrador

- **metrics-server, kube-state-metrics, node-exporter.**
- **Dashboards:** capacidade por nó/pool, *scheduling latency, eviction rate, OOMKills.*
- **SLOs** de plano de controle (gerenciado nas clouds): disponibilidade $\geq 99.95\%$.

Alertas-chave:

- `PodPending > N por 5m` (capacidade)
- `Evictions surge` (pressão de memória)
- `ImagePullBackOff` (registro ou política)
- `CrashLoopBackOff` (saúde de app)

8) Governança & Conformidade Operacional

- **Backstage/Portal** do desenvolvedor: *templates* padronizados de *Deployment/Service/HPA/NP.*
- **Policies** (OPA) que checam: *resources definidos, readiness, liveness, owner label, image tag imutável.*
- **Quotas** por namespace (CPU/mem/PVCs) para conter abuso ou erro.

9) Upgrade & Manutenção

- **Janela de manutenção** com *surge capacity* e **PDB** respeitado.
- **Blue/Green** no control plane quando suportado (clusters gerenciados com versões escalonadas).
- **Backups** regulares de **etcd** (para clusters autogerenciados).
- **Teste de upgrades** em `stg` antes de `prod` (compatibilidade de API resources).

10) Chaos Engineering & DR

- **Litmus/Chaos Mesh:** injeção de falhas (rede, disco, nó) semanais.
- **Simulações regionais** trimestrais (perda de AZ/região) + verificação de **RTO/RPO.**
- **Runbooks** vinculados a cada cenário (ex.: "Kafka partial outage").

11) Exemplos de Manifests (mínimos e executáveis)

Deployment (best-practices):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ms-feed
  namespace: feed
  labels: { app: ms-feed, owner: squad-feed }
spec:
  replicas: 6
  strategy: { type: RollingUpdate, rollingUpdate: { maxSurge: 1, maxUnavailable: 0 } }
  selector: { matchLabels: { app: ms-feed } }
  template:
    metadata:
      labels: { app: ms-feed, version: v1 }
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "8080"
    spec:
      serviceAccountName: ms-feed
      containers:
        - name: app
          image: registry/ms-feed:v1.9.3@sha256:deadbeef...
          ports: [ { containerPort: 8080 } ]
          resources: { requests: { cpu: "200m", memory: "256Mi" }, limits: { cpu: "1", memory: "512Mi" } }
          readinessProbe: { httpGet: { path: /healthz, port: 8080 }, periodSeconds: 5, timeoutSeconds: 1, failureThreshold: 6 }
          livenessProbe: { httpGet: { path: /livez, port: 8080 }, periodSeconds: 10, timeoutSeconds: 1, failureThreshold: 3 }
          envFrom:
            - secretRef: { name: ms-feed-secrets } # ou CSI Vault
          nodeSelector: { workload: general-purpose }
          affinity:
            podAntiAffinity:
              preferredDuringSchedulingIgnoredDuringExecution:
                - weight: 100
            podAffinityTerm:
              topologyKey: topology.kubernetes.io/zone
              labelSelector: { matchLabels: { app: ms-feed } }
          tolerations:
            - key: "spot"
              operator: "Equal"
              value: "true"
              effect: "NoSchedule"
```

12) SLOs Operacionais da Orquestração

- **Pod scheduling p95 < 3s** (com capacidade normal).
- **HPA reaction time < 60s** para dobrar réplicas sob carga.

- **Erro de implantação** (rollout failure) < 0.5%/mês.
- **Taxa de evictions** < 0.1%/dia em produção.

13) Runbooks Essenciais (resumo operacional)

- **"Pods presos em Pending"** → checar quotas/taints/affinity; acionar *cluster autoscaler*; analisar *insufficient CPU/memory*.
- **"CrashLoopBackOff"** → `kubectrl logs -p`; checar *readiness*; comparar mudança em `requests/limits`; rollback via Argo.
- **"Lag no Kafka explode"** → KEDA escala consumers; se persistir, *backpressure* nos producers via Istio; analisar DLQ.
- **"Pressão de memória no nó"** → redistribuir pods com *topology spread*; aumentar pool `memory-optimized`.

Fechamento da camada:

Com o Kubernetes afinado para crescer, se curar e proteger, passamos a detalhar **como os serviços conversam** — na velocidade do agora, com contratos e coerência: comunicação REST/gRPC/eventos.

◆ CAMADA 12 — COMUNICAÇÃO ENTRE SERVIÇOS (REST, gRPC, EVENTOS)

Entrada:

"A conversa entre partes do organismo precisa ser rápida, coerente e confiável — sob tráfego intenso e sob turbulência vibracional."

1) Princípios Arquiteturais

- **Híbrido por propósito:**
 - **REST/HTTP** (borda → público e integrações externas; contratos OpenAPI 3.1).
 - **gRPC** (intra-cluster, baixa latência, streaming opcional).
 - **Eventos Kafka** (assíncrono, acoplamento fraco, reprocessamento idempotente).
- **Contratos versionados:** OpenAPI para REST, Protobuf para gRPC, **Avro/JSON Schema no Registry** para eventos.
- **Confiabilidade:** retries com backoff + jitter, *circuit breakers*, *bulkheads*, DLQ, **idempotência ponta-a-ponta**.
- **Observabilidade:** `trace_id`, `span_id`, `schema_version`, `idempotency_key` propagados por cabeçalhos/metadata.

2) REST (HTTP/1.1 + HTTP/2 via Gateway)

Uso: borda (app↔gateway), integrações 3rd, webhooks.

Padrões:

- **OpenAPI 3.1** como fonte da verdade; *lint* no CI (Spectral).
- **Erro canônico:**

```
{ "trace_id": "...", "code": "RESOURCE_NOT_FOUND", "message": "...", "details": [...] }
```

- **Autorização:** JWT Bearer; scopes por rota (ABAC/RBAC).
- **Paginação:** cursor-based (`?cursor=...&limit=50`).

- **Idempotência** para POST sensível (moedas, check-in): header `Idempotency-Key` .
- **Timeouts**: 2–5s conforme rota; **retries** só para erros transitórios (<=2 tentativas).

Exemplo – OpenAPI (trecho)

```
openapi: 3.1.0
info: { title: FriendApp Public API, version: 1.0.0 }
paths:
  /v1/event/checkin:
    post:
      security: [{ bearerAuth: [] }]
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required: [event_id, geo]
              properties:
                event_id: { type: string, format: uuid }
                geo:
                  type: object
                  required: [lat, lon]
                  properties:
                    lat: { type: number }
                    lon: { type: number }
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                type: object
                properties:
                  status: { type: string, enum: [ok, already_checked] }
                  points: { type: number }
        "409":
          description: Conflict (idempotency duplicate)
    components:
      securitySchemes:
        bearerAuth: { type: http, scheme: bearer, bearerFormat: JWT }
```

Headers canônicos:

- `Authorization: Bearer <JWT>`
- `X-Trace-Id` , `X-Client-Region`
- `Idempotency-Key` (POST idempotente)
- `X-Schema-Version` (quando relevante)

3) gRPC (intra-cluster, baixa latência)

Uso: comunicações *service-to-service* com requisitos de latência/streaming, dentro do mesh.

Transport: HTTP/2, mTLS via Istio (roadmap imediato).

Timeout padrão: 1-3s; **retries** com política por método (no DestinationRule).

Protobuf – exemplo (Feed)

```
syntax = "3";
package friendapp.feed.v1;

message RenderFeedRequest {
  string user_id = 1;
  string region = 2;
  int32 freq_bucket = 3; // 0..10
  string trace_id = 4;
}

message FeedItem {
  string id = 1;
  string type = 2; // "card", "tip", "mission"
  string payload_json = 3;
  int32 ttl_seconds = 4;
}

message RenderFeedResponse {
  repeated FeedItem items = 1;
  float entropy = 2;
  string schema_version = 3;
}

service FeedService {
  rpc Render(RenderFeedRequest) returns (RenderFeedResponse) {}
}
```

Metadata propagada (gRPC):

- `trace-id` , `schema-version` , `idempotency-key` (quando aplicável).

Circuit breaker/timeout (Istio – referência):

- Retry 2x, `perTryTimeout: 800ms` , `timeout total: 2s` , *outlier detection* a partir de 5 erros 5xx em 30s.

4) Eventos (Kafka) — Assíncrono com Contrato

Uso: integração desacoplada (feed renderizado, presença, moedas, testes, RA), *fan-out* e eventual *replay*.

Contrato: Avro/JSON Schema no Schema Registry (Camada 13 detalha).

Semântica: “at-least once” (consumidores idempotentes).

Chaves: `event_id` , `occurred_at` , `idempotency_key` , `schema_version` , `producer` , `trace_id` .

Particionamento: por `user_id` ou `region` (conforme tópico) para manter ordenação local.

DLQ: por tópico; *reprocessamento* com *backoff*.

Exemplo – Payload (feed.rendered)

```
{
  "event_id": "0d3a-...-b9",
```

```
"schema_version": "1.3.0",
"occurred_at": "2025-09-13T21:59:03Z",
"producer": "ms-feed",
"idempotency_key": "3ca1-...-k9",
"trace_id": "a1b2c3...",
"user_id": "u-123",
"region": "sa-east-1",
"items_count": 15,
"entropy": 0.27
}
```

5) Políticas de Resiliência (comuns aos três modos)

- **Retries:** exponencial + **jitter** (p.ex. base 100ms, max 2–3 tentativas); **NÃO** repetir operações não idempotentes sem **Idempotency-Key**.
- **Circuit Breakers:** abrir após *N* erros/latências; meia-vida de 30–60s.
- **Bulkheads:** *pools* de conexão distintos; isolamento por *thread pool* e **limite de concorrência** por rota.
- **Dead Letter:** eventos que falham → DLQ para análise/manual replay.
- **Rate limit / quotas:** por *client_id* e por rota (no Gateway).

6) Idempotência & Consistência

- **REST:** **Idempotency-Key** obrigatório em POST que alteram saldo/estado (ex.: **coins/transfer**, **event/checkin**). Servidor grava **tabela de chaves** com *hash da request*; repetição → retorna **mesmo resultado** (200/201) sem duplicar efeitos.
- **Kafka:** **idempotency_key** validada em *consumer*; marcação em store barato (Redis/PG) para **dedupe**.
- **Sagas/Outbox:** mudanças e eventos publicados no **mesmo commit** (Outbox Pattern) em transações distribuídas.

Exemplo – REST idempotente (pseudo):

```
POST /v1/coins/transfer (Idempotency-Key: K)
server:
  if exists result_for(K): return cached_result
tx:
  debit(sender); credit(receiver)
  record_idem(K, hash(req), result)
return result
```

7) Tolerância a Falhas de Rede

- **Timeouts realistas** (fail fast), *retries* somente onde seguro.
- **Fallbacks:**
 - gRPC → REST (quando aplicável) ou resposta degradada (ex.: feed parcial do cache).
 - Eventos → *buffer local* + *re-envio* quando broker remoto indisponível.
- **Particionamento:** *graceful degradation* por domínio (ex.: Chat “somente leitura” quando store de histórico está em lenta recuperação).

8) Segurança na Comunicação

- **Fronteira:** TLS 1.3, WAF, JWT com *scopes* mínimos.
- **Intra-cluster:** mTLS via Mesh (Istio); *políticas* de autorização (ServiceRole, AuthorizationPolicy).
- **Headers/metadata saneados:** nunca transportar PII (e-mail/phone) entre serviços; usar **identificadores pseudo**.

9) Observabilidade de Tráfego

- **Trace** obrigatório*: gerar *span* por hop; *baggage* com `user_hash`, `region`, `freq_bucket`.
- **Métricas RED/USE** por rota e cliente.
- **Logs** com `trace_id`, `route`, `status`, `latency_ms`, `retry_count`, `circuit_state`.

10) Exemplos Executáveis (resumo)

cURL — Check-in Idempotente

```
curl -X POST https://api.friendapp.com/v1/event/checkin \  
-H "Authorization: Bearer $JWT" \  
-H "Idempotency-Key: 7b9f0e..." \  
-H "Content-Type: application/json" \  
-d '{"event_id":"8f2d...","geo":{"lat":-23.5,"lon":-46.6}}'
```

gRPC (Go) — Feed Render

```
ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)  
defer cancel()  
md := metadata.Pairs("trace-id", traceID, "schema-version", "1.3.0")  
ctx = metadata.NewOutgoingContext(ctx, md)  
resp, err := feedCli.Render(ctx, &feedpb.RenderFeedRequest{  
    UserId: userID, Region: region, FreqBucket: 6, TraceId: traceID,  
})
```

Kafka Producer (pseudo)

```
payload = { ..., idempotency_key: uuid(), trace_id }  
assert registry.validate("feed.rendered", payload)  
kafka.publish("feed.rendered", key=user_id, value=serialize(payload))
```

11) Regras de Versão & Deprecação

- **REST:** `/v1` estável; novos campos **add-only**; remoções → `/v2` com *sunset header* e período de migração (≥90 dias).
- **gRPC:** *fields* opcionais, *tag numbers* imutáveis; *breaking* → novo serviço `FeedServiceV2`.
- **Eventos:** compatibilidade **BACKWARD_TRANSITIVE**; *breaking* → **novo tópico** (`feed.rendered.v2`) e dupla publicação por período.

12) Testes de Contrato & Caos

- **Contratos:** Pact (REST/gRPC), *schema tests* (Registry) obrigatórios no CI.
- **Caos de rede:** latência/packet loss injetados (mesh/chaos) para validar *timeouts*, *retries* e *circuit break*.
- **Load:** k6/Vegeta para rotas quentes; benchmarks de QPS e p95 por domínio.

Fechamento da camada:

Com REST, gRPC e eventos operando em harmonia — rápidos, idempotentes e observáveis —, formalizamos agora a **partitura dos eventos**: o **Contrato de Eventos com Schema Registry** (Camada 13), que impede que uma nota fora do tom quebre a sinfonia inteira.

◆ CAMADA 13 — CONTRATO DE EVENTOS (SCHEMA REGISTRY)

Entrada:

“Sem contrato, a sinfonia se perde. Cada evento é uma nota, e o Registry é a partitura que garante que todos toquem juntos.”

1) Objetivo

Garantir que **todos os microserviços** publiquem e consumam eventos em Kafka com **estrutura e semântica consistentes**, prevenindo incompatibilidades e bugs difíceis de diagnosticar.

2) Ferramenta

- **Confluent Schema Registry** (ou compatível).
- Suporte a **Avro**, **Protobuf** e **JSON Schema** (preferência: **Avro** para payloads críticos).
- Localização: instância dedicada em cluster resiliente, com replicação e backup automático.

3) Regras de Compatibilidade

- **Default:** `BACKWARD_TRANSITIVE` (nova versão precisa aceitar payloads antigos).
- **Para breaking changes:** criar **novo tópico versionado** (`user.created.v2`).
- **Proibição:** alterar ou remover campos obrigatórios em schemas ativos.

4) Estrutura Mínima do Evento

Todo evento no FriendApp deve conter os seguintes campos obrigatórios:

```
{
  "event_id": "uuid",
  "schema_version": "string",
  "occurred_at": "ISO8601-timestamp",
  "producer": "string",
  "idempotency_key": "uuid",
  "trace_id": "string"
}
```

- `event_id`: identificador único do evento.
- `schema_version`: versão registrada no Registry.
- `occurred_at`: data/hora UTC no formato ISO8601.
- `producer`: microserviço de origem.

- `idempotency_key`: garante deduplicação.
- `trace_id`: rastreabilidade ponta-a-ponta.

5) Exemplo — Evento `user.created`

Avro Schema:

```
{
  "type": "record",
  "name": "UserCreated",
  "namespace": "friendapp.events",
  "fields": [
    { "name": "event_id", "type": "string" },
    { "name": "schema_version", "type": "string" },
    { "name": "occurred_at", "type": "string" },
    { "name": "producer", "type": "string" },
    { "name": "idempotency_key", "type": "string" },
    { "name": "trace_id", "type": "string" },
    { "name": "user_id", "type": "string" },
    { "name": "email", "type": ["null", "string"], "default": null },
    { "name": "region", "type": "string" }
  ]
}
```

Payload válido:

```
{
  "event_id": "d3f1e9c0-1234-4fa1-9c87-92a8f2c7bb12",
  "schema_version": "1.0.0",
  "occurred_at": "2025-09-13T22:30:15Z",
  "producer": "ms-auth",
  "idempotency_key": "a27b0e14-ef7d-4ac1-9a52-ff23c5dbb567",
  "trace_id": "t-2391a",
  "user_id": "u-9f3d7",
  "email": "user@test.com",
  "region": "sa-east-1"
}
```

6) Integração com CI/CD

- **Step obrigatório na pipeline:** validar schemas antes de publicar eventos.
- **Falha na validação:** bloqueia build/deploy.
- **Contratos versionados:** cada schema versionado no Git + CI sincroniza com o Registry.

7) Consumo Seguro

- **Consumidores só processam eventos compatíveis.**
- Eventos inválidos → **DLQ** (Dead Letter Queue).
- Retentativa com backoff exponencial + jitter.

- Regra: consumidor precisa lidar com **novos campos opcionais** sem falhar.

8) Observabilidade

- Dashboards monitorando:
 - **Taxa de rejeição de schemas** (% de eventos inválidos).
 - **Lag** em tópicos críticos.
 - **Distribuição por** `schema_version` (para garantir migração gradual).

9) Exemplo de Pseudocódigo — Publicação

```
function publish(topic, payload):
  schema = registry.get_latest(topic)
  if not validate(payload, schema):
    throw Error("Invalid schema")
  payload.idempotency_key = uuid()
  payload.trace_id = current_trace()
  kafka.publish(topic, serialize(payload), headers={trace_id})
```

10) Benefícios

- Previne que mudanças em serviços quebrem consumidores.
- Garante governança de dados em todo o ecossistema.
- Permite evolução gradual de schemas sem downtime.
- Melhora rastreabilidade e confiabilidade dos eventos vibracionais.

Fechamento da camada:

Com os contratos de eventos estabelecidos e governados pelo Schema Registry, a sinfonia de dados é preservada. Agora abrimos a porta para a borda pública: o API Gateway e os contratos HTTP OpenAPI.

◆ CAMADA 14 — API GATEWAY & OPENAPI (CONTRATOS HTTP)

Entrada:

"A borda é o portal. O gateway é o guardião. Cada rota precisa ser clara, segura e auditável."

1) Função do API Gateway

- **Entrada única** para todos os clientes (apps iOS, Android, Web, integrações externas).
- **Responsabilidades técnicas:**
 - Autenticação e autorização (JWT, OAuth2).
 - Rate limiting e quotas por cliente/usuário.
 - Roteamento para microserviços internos.
 - Logging e tracing distribuído (`trace_id` propagado).
 - Proteção contra ataques (WAF integrado).
- **Tecnologias:** Kong, NGINX Ingress Controller ou AWS API Gateway.

2) Segurança & Autenticação

- **JWT Bearer Tokens:** emitidos pelo Auth Service, com validade curta (15 min).
- **Refresh Tokens:** armazenados com segurança, renovação automática.
- **Scopes & Claims:** escopos de acesso definidos por serviço (ex.: `feed:read`, `chat:write`).
- **Rate limiting:** ex.: 100 requisições/min por usuário.
- **WAF Regras:** proteção contra injeção SQL, XSS, exploração de headers.

3) Contratos HTTP (OpenAPI 3.1)

- **Especificação única** versionada em Git, validada no CI.
- **Formato de erro padrão:**

```
{
  "trace_id": "t-123456",
  "code": "RESOURCE_NOT_FOUND",
  "message": "Evento não encontrado",
  "details": []
}
```

4) Exemplo OpenAPI — Endpoint de Check-in em Evento

```
openapi: 3.1.0
info:
  title: FriendApp Public API
  version: "1.0.0"
servers:
  - url: https://api.friendapp.com
paths:
  /v1/event/checkin:
    post:
      summary: Check-in vibracional em evento
      security:
        - bearerAuth: []
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required: [event_id, geo]
              properties:
                event_id:
                  type: string
                  format: uuid
                geo:
                  type: object
                  required: [lat, lon]
                  properties:
                    lat: { type: number }
```

```

      lon: { type: number }
responses:
  "200":
    description: Check-in confirmado
    content:
      application/json:
        schema:
          type: object
          properties:
            status: { type: string, enum: [ok, already_checked] }
            points: { type: number }
  "409":
    description: Conflito (idempotência detectada)
components:
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT

```

5) Headers Padrão

- `Authorization: Bearer <JWT>`
- `Idempotency-Key: <uuid>` para POST sensíveis
- `X-Trace-Id: <uuid>` para rastreabilidade ponta-a-ponta
- `X-Client-Region: sa-east-1` (opcional, para análise vibracional)
- `X-Schema-Version: 1.0.0`

6) Políticas de Versionamento

- **Versões no path:** `/v1`, `/v2`.
- **Novos campos:** sempre opcionais (evitar breaking changes).
- **Remoção de campos:** só em maior version (`/v2`).
- **Sunset headers:** avisam depreciação com antecedência mínima de 90 dias.

7) Exemplo cURL — Check-in Idempotente

```

curl -X POST https://api.friendapp.com/v1/event/checkin \
-H "Authorization: Bearer $JWT" \
-H "Idempotency-Key: 7b9f0e..." \
-H "Content-Type: application/json" \
-d '{"event_id":"8f2d...","geo":{"lat":-23.5,"lon":-46.6}}'

```

Resposta esperada:

```

{
  "status": "ok",
  "points": 50
}

```

```
}
```

8) Observabilidade no Gateway

- **Métricas expostas:** requisições p95, taxa de erro por rota, status codes.
- **Tracing:** cada request recebe `trace_id`.
- **Logs:** armazenam rota, status, latência, user_hash, região.

Fechamento da camada:

Com a borda segura e os contratos HTTP formalizados, seguimos agora para a camada de eficiência: FinOps e otimização de custos para escalar globalmente sem desperdício.

◆ CAMADA 15 — FINOPS & CUSTOS (EFICIÊNCIA RADICAL)

Entrada:

"Escalar não é gastar mais, é gastar com consciência. A energia precisa ser forte, mas também sustentável."

1) Objetivo

Otimizar custos de infraestrutura multinuvem sem comprometer **resiliência, performance ou vibração coletiva**.

👉 Garantir que cada recurso compute, armazene ou transmita apenas quando necessário.

2) Estratégias de Custo em Nuvem

- **Instâncias Spot/Preemptible:**
 - Usadas para workloads não críticos (ex.: retreinamento de IA, batch de relatórios).
 - Economia de até **90%** em computação.
- **Autoscaling agressivo:**
 - *Scale-to-zero* em ambientes `dev` e `stg` fora do horário de trabalho.
 - **KEDA** para escalar pods com base em filas/eventos (Kafka, Pub/Sub).
- **Serverless Functions:**
 - AWS Lambda / Google Cloud Functions para cargas esporádicas, como distribuição de recompensas (FriendCoins), webhooks de parceiros, notificações vibracionais.
 - **Custo zero** quando inativos.
- **Node Pools diferenciados:**
 - `spot` para tarefas descartáveis,
 - `general` para serviços críticos,
 - `memory-optimized` para IA vibracional (Aurah Kosmos).

3) Armazenamento & Dados

- **Tiering de Storage:**
 - Dados quentes → SSD (alta performance).

- Dados frios → storage de baixo custo (S3 Glacier, Google Archive).
- **Expurgo automático:**
 - Sessões efêmeras → Redis/Firestore com TTL dinâmico.
 - Dados vibracionais individuais → anonimizados após 90 dias.
- **BigQuery otimizado:**
 - Particionamento + clustering por região e data.
 - Alertas de queries muito caras; orçamentos por time de produto.

4) Observabilidade Financeira

- **FinOps dashboards** em Grafana/Looker:
 - Custo por serviço, por time, por região.
 - Custo por usuário ativo (ARPU vs Infra).
- **Budgets e alerts:**
 - Notificações se uso diário/mensal ultrapassar 80% do orçamento.
- **KPIs:**
 - **Infra/Receita ≤ 30%** no cenário inicial.
 - **Infra/Receita ≤ 15%** após maturidade.

5) Exemplo de Políticas

- **Política de dev:** clusters desligam automaticamente às 22h e religam às 8h.
- **Política de staging:** escala mínima = 0 em finais de semana.
- **Política de produção:**
 - Criticidade A (Auth, Coins, Feed) → nunca escala para zero.
 - Criticidade B (Jobs batch, relatórios) → podem rodar em spot/escala zero.

6) Exemplo Pseudocódigo de Escalonamento

```
if environment == "dev" and time between 22h-8h:
    scale_all_services_to(0)

if service in ["jobs_batch", "training_IA"] and load == 0:
    migrate_to_spot_instances()

if kafka_lag > 1000:
    keda.scale_up(service="feed-consumer", replicas+=10)

if budget_consumption > 80%:
    alert(FinOpsTeam, region, service)
```

7) Benefícios

- Economia significativa em **cloud spend**.
- Sustentabilidade energética, alinhada ao propósito vibracional do FriendApp.

- Infraestrutura adaptada ao fluxo real de energia dos usuários, evitando desperdícios.

Fechamento da camada:

Com custos otimizados e energia alocada com consciência, seguimos para a próxima camada: a rede de velocidade e proximidade — Cache & CDN Global.

◆ CAMADA 16 — CACHE & CDN GLOBAL (EDGE + INTELIGÊNCIA)

Entrada:

“Velocidade é cuidado. Aproximamos a energia do usuário e servimos apenas o que importa, quando importa.”

1) Objetivo

Reduzir latência percebida (< **300ms p95** global), aliviar backend e proteger a borda com **cache multinível**:

- **CDN/Edge** (Cloudflare/Akamai) para estáticos, HTML dinâmico cacheável e *edge functions*.
- **Redis/Redis Edge** como cache de dados de aplicação e *computed views* (feed, contadores, presença).
- **TTL inteligente** sensível ao **estado vibracional** (maior instabilidade → TTL menor; estabilidade → TTL maior).

2) Arquitetura de Cache em 3 Níveis

1. Nível 0 — Browser/App

- *HTTP cache-control* (immutable para assets, max-age adequado para HTML dinâmico com revalidação).
- *Client-side memoization* para sub-requests idênticas em curto prazo (10–60s).

2. Nível 1 — CDN/Edge

- **Full-page caching** seletivo para páginas públicas e partes do feed tratáveis como *edge fragments*.
- **Edge KV / D1 / Redis Edge** para *key-value* de baixa cardinalidade (hints, banners, status regional).
- **WAF + Bot management** na borda.

3. Nível 2 — App Cache (Redis Cluster)

- Hashes/listas *per user/region/freq_bucket*.
- **Stale-While-Revalidate (SWR)** e **Write-Behind** para evitar *thundering herd*.
- **Topic-based invalidation** (Pub/Sub) para coerência seletiva.

3) Design de Chaves & Segmentação

• Chaves canônicas:

- Sessão: `sess:{userId}`
 - Feed: `feed:{userId}:{region}:{freqBucket}:{schemaVer}`
 - Listas públicas (descoberta): `list:region:{region}:v{n}`
 - Contadores: `cnt:event:{eventId}`
 - **Cardinalidade controlada:** incluir `schemaVer` e `freqBucket` (0..10) para eliminar colisões em mudanças de estrutura ou intenção.
 - **Segurança:** *nunca* cachear PII em CDN; dados pessoais somente em Redis (servidor) com TTL curto.
-

4) TTL Dinâmico (Energia & Conteúdo)

- **Feed sensível:**
 - `freqBucket >= 7` (alta excitação): **TTL 15–45s**
 - `3 <= freqBucket <= 6` (neutro): **TTL 60–120s**
 - `freqBucket <= 2` (baixa vibração): **TTL 10–20s** + *modo protetor* (conteúdo leve)
- **Estáticos (assets):** `immutable, max-age=31536000`.
- **HTML cacheável com revalidação:** **SWR 30–60s**.

5) Políticas no CDN (Cloudflare/Akamai)

- **Cache Keys** incluindo cabeçalhos relevantes (`X-Client-Region` , `X-Schema-Version`) quando seguro.
- **Bypass** para rotas sensíveis (Auth, Coins, Check-in).
- **Edge Functions (Workers):** injetar *hints* por região e *freqBucket* (sem PII), fazer *early return* de fragmentos cacheados.

Exemplo — Pseudo Worker (Cloudflare):

```
export default {
  async fetch(req, env) {
    const url = new URL(req.url);
    if (url.pathname.startsWith('/public/')) {
      // full page cache
      const cacheKey = new Request(url.toString(), req);
      let resp = await caches.default.match(cacheKey);
      if (resp) return resp;
      resp = await fetch(req);
      const ttl = 300; // 5m
      const headers = new Headers(resp.headers);
      headers.set('Cache-Control', `public, s-maxage=${ttl}, stale-while-revalidate=60`);
      return new Response(await resp.arrayBuffer(), { headers, status: resp.status });
    }
    // fallback to origin
    return fetch(req);
  }
}
```

6) Redis Aplicacional — Padrões

- **SWR (Stale-While-Revalidate):**
 - Servir valor “stale” por até 10–30s enquanto *background job* recalcula.
- **Write-Through** para chaves derivadas estáveis (ex.: rankings).
- **Bloom Filter** para evitar cache miss repetido (negativos persistentes temporários).
- **Lock distribuído (SET NX PX)** para evitar *dogpile* no recálculo.

Exemplo — SWR (pseudocódigo):

```
function getFeed(userId, region, fb):
  key = fmt("feed:%s:%s:%d:%s", userId, region, fb, schemaVer)
```

```

val = redis.get(key)
if val && not isExpiredSoft(val):
    return val.data // stale-ok
if lock.acquire(key+":lock", ttl=3000ms):
    fresh = recomputeFeed(userId, region, fb)
    redis.set(key, {data: fresh, ts: now()}, ex=ttlDynamic(fb))
    lock.release()
    return fresh
else:
    // alguém já recalculando; devolver stale ou fallback
    return val?.data ?? minimalFallback()

```

7) Invalidação & Coerência

- **Granularidade por tópico:**
 - Ex.: novo post que afeta top-N → `PUBLISH feed:invalidate:{userId}` para recalculando apenas as chaves daquele usuário.
- **Invalidação por *region/freqBucket*** em eventos coletivos (colapso regional → alterar TTL globalmente).
- **Edge purge** seletivo: usar *tags* (Akamai) ou *surrogate keys* (Fastly) quando disponível.

8) Cache de API (Gateway)

- **Kong/AWS API Gateway** com *plugin cache* para GET idempotentes de baixa sensibilidade (ex.: catálogos públicos, configurações globais).
- **Headers de coerência:** `ETag`, `If-None-Match`, `Last-Modified` para economizar e manter correteude.

9) Observabilidade & KPIs

- **Métricas (Prom/Grafana/Datadog):**
 - **Hit rate** (global e por rota).
 - **Miss rate** e **origin latency p95**.
 - **Edge errors 4xx/5xx** por POP.
 - **Revalidação:** tempo médio de recomputação, *lock contention*.
- **Alertas:**
 - Hit rate < 80% em rotas elegíveis.
 - Aumento de `latency_origin_p95` > 400ms.
 - Revalidações lentas (> 1s) em rotas críticas.

10) Segurança na Borda

- **WAF ativo**, *rate limit* e *bot mitigation* antes do cache.
- **Headers de segurança:** HSTS, CSP, X-Frame-Options, Referrer-Policy.
- **Assinatura de URLs** para conteúdos privados (expiração curta / *signed cookies*).

11) Estratégias Anti-*Thundering Herd*

- **Jitter em TTLs** (+/- 10-20%).

- **Soft TTL** + lock distribuído (uma única recomputação).
- **Warmup** proativo de chaves quentes (pré-carregar feed após login).

12) Políticas por Tipo de Conteúdo

- **RA/Assets 3D**: compressão adaptativa, *content negotiation* (webp/avif), `max-age` alto.
- **Feed**: SWR + TTL dinâmico por *freqBucket*; **não** cachear payloads com PII na CDN.
- **Eventos/Mapa**: *tile caching* por *zoom/region*; expiração curta (30–120s).

13) Testes & Ensaios

- **Experimentos A/B** de TTL por região e *freqBucket*.
- **Chaos cache**: invalidar 10% de chaves quentes e medir recuperação de p95.
- **Load tests** com cache habilitado vs desabilitado para provar economia (QPS ↑, p95 ↓).

14) Exemplo de Config — NGINX Edge Caching (conceito)

```
proxy_cache_path /var/cache/nginx levels=1:2 keys_zone=FRIENDAPP:100m max_size=10g inactive=10m
use_temp_path=off;

server {
    location /public/ {
        proxy_cache FRIENDAPP;
        proxy_cache_valid 200 5m;
        add_header Cache-Control "public, s-maxage=300, stale-while-revalidate=60";
        proxy_pass http://origin;
    }

    location /v1/feed/render {
        proxy_cache_bypass 1; # feed sensível: cache só em app/redis, não na CDN local
        proxy_pass http://origin;
    }
}
```

15) KPIs de Sucesso

- **p95 global < 300ms** em rotas cacheáveis.
- **Hit rate ≥ 85%** em conteúdos elegíveis.
- **Redução ≥ 40%** de chamadas ao backend em picos.
- **Erro 5xx de origem < 0.3%** sob carga cacheada.

Fechamento da camada:

Com a malha de cache e edge afinada, a experiência fica leve e imediata — mesmo em tempestades de tráfego e energia. Na próxima camada, abrimos os canais “vivos” do agora: **Tempo Real & WebSockets (Chat/Presença/RA)**.

◆ CAMADA 17 — TEMPO REAL & WEBSOCKETS (CHAT / PRESENÇA / RA)

Entrada:

"O agora precisa ser agora. Se a frequência muda, a mensagem acompanha — sem atraso, sem ruído, com elegância."

1) Objetivo da Camada

Entregar comunicação **bi-direcional, baixa latência e elástica** para **Chat Vibracional, Presença, Sinais de RA e Eventos ao Vivo**, garantindo:

- **p95 RTT** intra-região \leq **150 ms**; cross-região \leq **350 ms**.
- **QoS** adaptativa por canal (prioridade para presença e sinais críticos de segurança).
- **Backpressure** e **flow control** para evitar *meltdowns* em picos.
- **Entrega confiável** (*at least once* com dedupe no cliente) quando necessário.

2) Arquitetura Lógica (camadas e componentes)

```
Client (App iOS/Android/Web)
├─ Realtime SDK (WS/SSE fallback, QoS, buffers, retry)
├─ Local State (room cache, presence, outbound queue)
└─ ↓ TLS 1.3
Edge / Gateway Realtime (Nginx/Envoy + WAF + sticky session por hash)
└─ ↓
WS Broker Layer (Node/Go) - Stateless, autoscaling (K8s + HPA/KEDA)
└─ ↓
Channels & Pub/Sub Layer (Redis Cluster / NATS / Kafka-Bridge)
└─ ↓
Domain MS (Chat, Eventos, RA, Perfil/Frequência) via gRPC/REST
└─ ↓
Persistence (PostgreSQL p/ histórico chat, Firestore p/ presença/RA)
└─ ↓
Observabilidade (OTel traces, RED metrics, logs estruturados)
```

- **Sticky Strategy**: *consistent hashing* por `user_id` para minimizar *fan-out* inter-nós.
- **Fan-out**: por *room/topic* com *sharding* (ex.: `room:{hash%N}`).
- **Backplane**: Redis **Pub/Sub** (latência baixa) + ponte para **Kafka** (audit/analytics).

3) Modelos de Canais (Rooms/Topics)

- `room:chat:{roomId}` — mensagens de texto/emoji, eventos de digitação.
- `presence:{roomId}` — *join/leave/heartbeat*, contadores e metadados mínimos.
- `signal:ra:{sessionId}` — coordenadas/overlays/estados de RA.
- `signal:event:{eventId}` — instruções ao vivo (ex.: check-in aberto/fechado).
- `sys:protect:{region}` — *broadcast* de "modo protetor" (redução de estímulo).

4) Protocolo de Mensagens (frames WS – JSON canônico)

Envelope comum

```
{
  "type": "chat.message|presence.heartbeat|signal.ra|system.protect",
  "ts": "2025-09-13T22:55:12Z",
  "trace_id": "t-92f3",
  "schema_version": "1.0.0",
  "meta": { "region": "sa-east-1", "freq_bucket": 6 }
}
```

Chat message

```
{
  "type": "chat.message",
  "room_id": "r-7ab3",
  "message_id": "m-9e12",
  "user_id": "u-123",
  "content": { "text": "⚡", "stickers": [], "attachments": [] },
  "qos": "normal|high",
  "ack_id": "c-8bd1" // correlacionar confirmação
}
```

Presence heartbeat

```
{
  "type": "presence.heartbeat",
  "room_id": "r-7ab3",
  "user_id": "u-123",
  "state": "online|idle|typing",
  "ttl_sec": 45
}
```

RA signal

```
{
  "type": "signal.ra",
  "session_id": "ra-991",
  "payload": { "pose": [x,y,z], "hint": "align-left", "alpha": 0.82 }
}
```

System protective mode

```
{
  "type": "system.protect",
  "region": "eu-west-1",
  "level": "soft|medium|hard",
  "expires_at": "2025-09-13T23:10:00Z"
}
```

5) QoS, Prioridades e Controle de Fluxo

- **Classificação:**
 - **Alta:** `presence.heartbeat`, `system.protect`, `ack/nack`.
 - **Normal:** `chat.message`, `signal.event`.
 - **Baixa:** indicadores de digitação, *typing*, *read receipts*.
- **Fila de saída no cliente** com **leaky bucket** (ex.: 30 msg/s) e **token bucket** para *bursts* curtos.
- **Backpressure no servidor:** limitar *per-connection send buffer*; se exceder, **nack + retry-after**.

Pseudocódigo – backpressure

```
onOutbound(client, msg):
  if client.sendBuffer.size > MAX_BUFFER:
    queue.lowPriority.dropOldestUntil(UNDER_WATERMARK)
    sendNack(client, msg, retryAfter=rand(150..400)ms)
  else:
    socket.send(msg)
```

6) Garantias de Entrega & Deduplicação

- **Default:** entrega *best effort* (chat social tolera perda mínima sob picos).
- **At least once** (canal marcado) com *acks* e **reenvio**:
 - Cliente inclui `ack_id` em cada mensagem.
 - Servidor responde `ack(ack_id)`; cliente remove da fila.
 - Em reconexão, cliente reenfileira **não confirmadas**.
- **Dedupe no cliente/servidor** por `(message_id|ack_id)`.

Pseudocódigo – cliente

```
send(msg):
  q.push(msg); tryFlush()
onAck(ack_id): q.remove(ack_id)
onReconnect(): resendAllUnacked()
```

7) Presença Confiável (Heartbeats & TTL)

- **Heartbeat** cliente → servidor a cada **15 s** (móvel pode ser 20–30 s com *background modes*).
- **TTL presença** = 3× período de heartbeat (ex.: 45–90 s).
- **Falha de heartbeat** → marcar `idle` e, após TTL, `offline`.
- **Contador de presença** mantido em **Redis** (hyperloglog ou counter com `room:{id}:count`).

8) Escalabilidade & Autoscaling

- **HPA** por CPU + métricas custom (`conn_active`, `msg_rate`).
- **KEDA** por lag no *backplane* (Redis stream/Kafka).
- **Sharding** por `roomId%N` para fan-out equilibrado.

- **Sticky** por `user_id` reduz *cross-node chatter*.

9) Fallbacks de Transporte

- **Primário:** WebSocket (HTTP/2 sempre que possível).
- **Fallback:** **SSE** (server-sent events) para *downstream only*.
- **Emergência:** *long-polling* com *backoff*.
- Detecção automática no SDK e *upgrade* quando o primário voltar.

10) Persistência & Histórico

- **Chat histórico:**
 - Persistir **síncrono** em **PostgreSQL** (tabela particionada por `room_id` /tempo).
 - Índices: `(room_id, created_at DESC)` , `(user_id, created_at)` .
 - *Soft delete* + retenção por política (ex.: 6–24 meses).
- **Eventos efêmeros (presença/RA):**
 - Armazenar **apenas estado atual** no **Firestore**; *streams* analíticos no **BigQuery** via Kafka.

11) Segurança em Realtime

- **Handshake:** JWT curto (≤ 15 min), *claims* de rooms permitidas.
- **AutZ:** validar *join room* no servidor (ABAC – ex.: membro do evento).
- **mTLS (mesh)** entre brokers e domínios.
- **Flood control** e **anti-spam:**
 - Rate limit por usuário/room (ex.: 10 msg/5 s).
 - *Captcha adaptativo* após padrões suspeitos.
 - Bloqueio por *content moderation* (Aurah Kosmos).

Pseudocódigo – ingresso room

```
joinRoom(user, room):
  assert jwt.scope includes room
  if !policy.allows(user, room): deny(403)
  subscribe(user.conn, topic="room:"+room)
```

12) Observabilidade Realtime

- **Métricas (RED):**
 - `conn_active` , `msg_in_rate` , `msg_out_rate` , `drop_ratio` , `ack_latency_p95` , `fanout_per_msg` .
- **Traces:** *span* para *send/receive*; `trace_id` do envelope correlaciona com backend (chat → persist → ack).
- **Logs:** JSON com `conn_id` , `room_id` , `size_bytes` , `retry_count` , `reason` .

Alertas:

- `ack_latency_p95 > 200ms (5m)`
- `drop_ratio > 0.5% (3m)`
- `conn_active surge` → verifica *HPA/KEDA*

13) Mobilidade, Background & Offline-First

- **Queue local** para mensagens **outbound** quando offline; *flush* ao reconectar.
 - **Delta sync** de histórico em *resume* (cursor por `ts`).
 - **Modo economia**: reduzir heartbeat para 30–60 s quando bateria < 15%.
-

14) Compatibilidade Vibracional

- **Freq Bucket aware**: se `freq_bucket <= 2`, o SDK **silencia indicadores não essenciais** (typing, read receipts), priorizando presença e mensagens diretas de apoio.
 - **Broadcast “modo protetor”** (system.protect) reduz taxa de *send* permitida e *fan-out* de notificações.
-

15) Testes, Caos & Carga

- **Load (k6/Vegeta)**: simular 100k conexões WS/cluster, *fan-out* 1:50.
 - **Caos**: queda de 20% dos nós WS; perda parcial de Redis; latência artificial 200 ms — verificar *drift* de p95 e *reconnect storms*.
 - **Contract tests**: validar esquemas de envelopes e *backward compatibility*.
-

16) Esquemas de Banco (exemplos sucintos)

PostgreSQL – `chat.messages` (particionado)

```
CREATE TABLE chat.messages_2025_09 PARTITION OF chat.messages
FOR VALUES FROM ('2025-09-01') TO ('2025-10-01');

CREATE TABLE chat.messages (
  id BIGSERIAL PRIMARY KEY,
  room_id TEXT NOT NULL,
  user_id TEXT NOT NULL,
  content JSONB NOT NULL,
  created_at TIMESTAMPTZ NOT NULL DEFAULT now()
) PARTITION BY RANGE (created_at);

CREATE INDEX ON chat.messages (room_id, created_at DESC);
```

Firestore – presença

```
presence/{roomId}/users/{userId} ⇒ { state, last_seen, device, hz_bucket }
```

17) Contratos do SDK (cliente)

Interface (pseudo-TS)

```
interface RealtimeClient {
  connect(jwt: string): Promise<void>;
  join(roomId: string): Promise<void>;
  leave(roomId: string): Promise<void>;
  publish(roomId: string, message: ChatMessage, opts?: QoS): Promise<Ack>;
}
```

```
on(event: "message"|"presence"|"system", cb: (evt) => void): void;
setFreqBucket(n: 0|1|...|10): void;
}
```

Fechamento da camada:

Com os canais de tempo real estabilizados — rápidos, observáveis e compassivos com a energia do usuário — avançamos para a espinha global que sustenta o planeta: **Multicloud & Topologia de Rede** (Camada 18).

◆ CAMADA 18 — MULTICLOUD & TOPOLOGIA DE REDE

Entrada:

“Um só céu, várias nuvens. A energia do FriendApp precisa atravessar continentes com resiliência e inteligência.”

1) Objetivo

Garantir **alta disponibilidade planetária**, com operação simultânea em **AWS, GCP e Azure**, suportando:

- Failover geográfico automático.
- Baixa latência (<300 ms global).
- Isolamento de falhas regionais sem afetar o todo.

2) Estratégia Multicloud

- **AWS (primária Américas):** clusters `sa-east-1` (São Paulo), `us-east-1`.
- **GCP (primária Europa & Ásia):** clusters `europa-west1`, `asia-east1`.
- **Azure (apoio global):** failover e workloads auxiliares (`westeurope`, `eastus`).
- **Orquestração global:**
 - **Anycast DNS** (Route 53 + Cloud DNS + Azure Traffic Manager).
 - **BGP Anycast** para CDN/Edge.
 - **Global Load Balancing** (Cloudflare, Akamai).

3) Topologia Lógica de Rede

```
App Client (Web/Mobile)
  ↓ TLS 1.3
CDN/Edge (Cloudflare/Akamai) → WAF → API Gateway
  ↓
Regional Cluster (AWS/GCP/Azure)
  ├── Ingress (Envoy/Istio)
  ├── Microservices Pods
  ├── Redis Cluster (cache regional)
  └── PG/Firestore replica local
  ↓
Backbone (Interconnect/MPLS + VPN TLS/IPsec)
  ↓
Cross-Region Replication & Messaging (Kafka MirrorMaker2)
```



Observability Global (Grafana multi-source, Datadog unificado)

4) Cross-Region Replication

- **PostgreSQL**: streaming replication assíncrona entre regiões; **read replicas** em clusters vizinhos.
- **Firestore**: replicação automática via GCP multi-region.
- **Neo4j**: cluster causal; replicação full mesh entre 3 regiões.
- **Redis**: active-active (Redis Enterprise/Cluster) para sessões.
- **Kafka: MirrorMaker2** (topic-by-topic), replicação com `idempotency_key` para evitar duplicação.

5) Isolamento e Failover

- **Bulkhead regional**: se um cluster falha → tráfego roteado via DNS LB.
- **Failover modes**:
 - **Soft failover**: parte do tráfego redirecionado se latência ↑ ou disponibilidade ↓.
 - **Hard failover**: todo tráfego de uma região redirecionado quando cluster fica inacessível.
- **Tempo alvo de failover**: <60 s.

Pseudocódigo (failover DNS LB):

```
onRegionDegraded(region):  
  if error_rate > 5% OR latency_p95 > 1000ms for 3m:  
    dns.shiftTraffic(region, to=backup_region)  
    scaleUp(backup_region)  
    notifyOps(region, "Failover triggered")
```

6) Políticas de Segurança de Rede

- **Zero Trust** entre regiões → VPNs TLS/IPsec + mTLS entre serviços.
- **Firewalls de borda**: Cloudflare WAF, AWS Shield, Azure DDoS Protection.
- **Private interconnects**: Direct Connect (AWS), Interconnect (GCP), ExpressRoute (Azure).
- **Controle granular**: Security Groups (AWS), VPC Firewall (GCP), NSG (Azure).

7) Observabilidade de Rede Global

- **Métricas globais**: latência média por região, tráfego por POP, % failover ativo.
- **Dashboards**:
 - Mapa com regiões ativas, cargas, custos, Hz regional.
 - Alarmes em caso de perda de replicação (Kafka/PG).
- **Alertas**:
 - Latência média >500 ms global.
 - Perda de link cross-cloud.
 - Falha de replicação Kafka/DB.

8) KPIs

- **Disponibilidade global:** $\geq 99,99\%$.
- **Tempo médio de failover:** ≤ 60 s.
- **Latência global (p95):** ≤ 300 ms.
- **Taxa de erro regional durante failover:** $\leq 1\%$.

Fechamento da camada:

Com a topologia multicloud pronta e resiliente, fechamos o Macrogrupo 2. A partir da próxima camada, entramos na execução real: código, logs, rotas e orquestração prática do ecossistema.

◆ CAMADA 19 — PSEUDOCÓDIGOS CRÍTICOS (FAILOVER / EVENTOS / BACKOFF)

Entrada:

"Quando tudo falha, a lógica deve se reerguer. O código é o reflexo vivo da resiliência."

1) Failover de Microserviço

Objetivo: reiniciar serviço falho, redirecionar tráfego e registrar incidente.

```
onMicroserviceDown(service_id):
    log.error("FAILURE", service_id, ts=now(), trace=current_trace())
    triggerAutoRestart(service_id)

    if not isHealthy(service_id, within=10s):
        rerouteTrafficToReplica(service_id)
        notify("Aurah Kosmos", event="infra-failover", context=service_id)
        updateDashboard(service_id, status="rerouted", region=current_region)
```

2) Failover Regional

Objetivo: detectar falha regional, redistribuir carga e ativar proteção vibracional.

```
onRegionFailure(region_id):
    if error_rate(region_id) > 5% or latency_p95(region_id) > 1000ms for 3m:
        dns.shiftTraffic(region_id, to=backup_region)
        scaleUp(backup_region, services=critical_services)
        publishEvent("system.protect", { "region": region_id, "level": "hard" })
        notifyOps(region_id, "Failover executed")
```

3) Publicação de Evento com Contrato (Kafka + Registry)

Objetivo: validar payload, garantir idempotência e rastreabilidade.

```
function publish(topic, payload):
    schema = registry.get_latest(topic)
    if not validate(payload, schema):
        throw Error("Invalid schema")
```



```

payload.idempotency_key = uuid()
payload.trace_id = current_trace()

kafka.publish(
    topic,
    key=payload.user_id,
    value=serialize(payload),
    headers={ "trace_id": payload.trace_id, "schema_version": schema.version }
)

```

4) Consumo Idempotente com Backoff

Objetivo: evitar duplicação, reprocessar falhas temporárias e enviar para DLQ em falhas críticas.

```

onMessage(topic, msg):
    if seen(msg.idempotency_key):
        ack(); return

    try:
        process(msg)
        markSeen(msg.idempotency_key)
        ack()
    except RetryableError:
        retry(msg, backoff=exponential_jitter(base=100ms, max=30s))
    except:
        deadLetter(topic, msg)

```

5) Reconexão WebSocket (Tempo Real)

Objetivo: manter chat/presença ativos mesmo sob falhas de rede.

```

onDisconnect():
    scheduleReconnect(delay=rand(1000..3000)ms)

onReconnect():
    send("resume", { "last_ack": lastAckId })
    resendUnackedMessages()

```

6) Workflow de Saga (Exemplo: Transferência de FriendCoins)

Objetivo: orquestrar múltiplos serviços de forma consistente.

```

startTransfer(tx):
    try:
        debitUser(tx.sender, tx.amount)
        creditUser(tx.receiver, tx.amount)
        recordLedger(tx)
        publish("coins.transferred", tx)
    except:
        # Rollback logic would go here

```

```
commit()
except:
    rollback()
    publish("coins.transfer_failed", { "tx_id": tx.id })
```

7) Outbox Pattern (Consistência Evento + DB)

Objetivo: gravar evento e dados no mesmo commit, evitando perda.

```
BEGIN;
INSERT INTO coins.ledger (...) VALUES (...);
INSERT INTO outbox (topic, payload, schema_version, idempotency_key) VALUES (...);
COMMIT;

-- Worker dedicado
for msg in select * from outbox where not sent:
    schema = registry.get(msg.topic)
    kafka.publish(msg.topic, serialize(msg.payload))
    markSent(msg.id)
```

8) Pseudocódigo de Alertas & Observabilidade

Objetivo: disparar alarmes em anomalias técnicas ou vibracionais.

```
if latency_p95("/v1/feed/render") > 400ms for 5m:
    alert("Feed Latency", sev=2, region=current_region)

if hz_mean(region) < threshold for 10m:
    publishEvent("system.protect", { "region": region, "level": "medium" })
    alert("Vibrational Collapse", sev=1, region=region)
```

9) Benefícios dos Pseudocódigos

- **Execução direta:** devs podem transformar em código real com mínima adaptação.
- **Cobertura crítica:** failover, eventos, sagas, tempo real e observabilidade.
- **Consistência:** todos usam contratos, idempotência e rastreabilidade.

Fechamento da camada:

Com os pseudocódigos críticos definidos, entramos na formalização das interfaces de acesso: as rotas de API, autenticação e contratos técnicos (Camada 20).

◆ CAMADA 20 — ENDPOINTS DE API (JWT + RATE LIMIT + CONTRATOS)

Entrada:

"Interfaces que não quebram. Cada rota é um compromisso de estabilidade, segurança e clareza."

1) Padrões Gerais

- **Protocolo:** HTTPS (TLS 1.3).
- **Domínio:** `https://api.friendapp.com` (prod) · `https://stg.api.friendapp.com` (staging).
- **Versão:** prefixo obrigatório `/v1`; *breaking change* ⇒ nova major `/v2`.
- **Autenticação:** OAuth 2.1 / OIDC → **JWT Bearer** (exp ≤ 15 min) + **Refresh Token**.
- **Autorização:** RBAC/ABAC via *scopes* (ex.: `feed:read`, `chat:write`, `coins:transfer`).
- **Idempotência:** cabeçalho `Idempotency-Key` em POST/PUT que alteram estado (transfer, check-in, criação).
- **Rate limit (padrão):** 100 req/min por *user_id*; 1000 req/min por *client_id*; *burst* controlado (token bucket).
- **Erro canônico (corpo JSON):**

```
{ "trace_id":"...", "code":"...", "message":"...", "details":[] }
```

- **Paginação (cursor-based):** `?cursor=...&limit=50` (limite máx. 200).
- **Headers padrão (requisição):**
 - `Authorization: Bearer <JWT>`
 - `Idempotency-Key: <uuid>` (quando idempotente)
 - `X-Trace-Id: <uuid>` (opcional; gerado se ausente)
 - `X-Client-Region: sa-east-1` (telemetria/afinagem)
- **Headers padrão (resposta):**
 - `X-Trace-Id`, `X-RateLimit-Limit`, `X-RateLimit-Remaining`, `X-RateLimit-Reset`
 - `Cache-Control` apropriado (geralmente `no-store` para dados pessoais)

2) Tabela de Rotas Públicas (núcleo)

Domínio	Método	Rota	Escopo	Idempotente	Descrição
Auth	POST	<code>/v1/auth/login</code>	<code>auth:login</code>	—	Login/password ou social OAuth; emite JWT
Auth	POST	<code>/v1/auth/refresh</code>	<code>auth:refresh</code>	✓	Emite novo JWT a partir do refresh token
Perfil	GET	<code>/v1/user/profile</code>	<code>profile:read</code>	—	Perfil do usuário + snapshot vibracional
Perfil	PATCH	<code>/v1/user/profile</code>	<code>profile:write</code>	✓	Atualiza campos do perfil
Feed	GET	<code>/v1/feed/render?mode=vibrational</code>	<code>feed:read</code>	—	Renderiza feed sensorial (cache-aware)
Evento	POST	<code>/v1/event/checkin</code>	<code>event:write</code>	✓	Check-in vibracional em evento
Chat	GET	<code>/v1/chat/rooms</code>	<code>chat:read</code>	—	Lista salas do usuário

Domínio	Método	Rota	Escopo	Idempotente	Descrição
					(paginado)
Chat	POST	/v1/chat/rooms/{roomId}/messages	chat:write	✓	Envio de mensagem (persist+fanout)
RA	POST	/v1/ra/start	ra:write	✓	Inicia sessão de RA (token efêmero)
Coins	POST	/v1/coins/transfer	coins:transfer	✓	Transfere FriendCoins (ledger duplo)

Obs.: endpoints admin e metrics expostos em domínio interno e com escopos dedicados.

3) Contratos — Exemplos Executáveis (OpenAPI 3.1 – trechos)

3.1 Auth — Login

```

paths:
  /v1/auth/login:
    post:
      summary: Login com credenciais ou OAuth token
      requestBody:
        required: true
        content:
          application/json:
            schema:
              oneOf:
                - type: object
                  required: [email, password]
                  properties:
                    email: { type: string, format: email }
                    password: { type: string, minLength: 8 }
                - type: object
                  required: [oauth_provider, oauth_token]
                  properties:
                    oauth_provider: { type: string, enum: [google, apple, facebook] }
                    oauth_token: { type: string }
      responses:
        "200":
          description: Sucesso
          content:
            application/json:
              schema:
                type: object
                required: [access_token, expires_in, refresh_token]
                properties:
                  access_token: { type: string }
                  expires_in: { type: integer }
                  refresh_token: { type: string }
        "401": { $ref: "#/components/responses/Unauthorized" }

```

3.2 Perfil — GET /v1/user/profile

```
paths:
  /v1/user/profile:
    get:
      security: [ { bearerAuth: [] } ]
      responses:
        "200":
          description: Perfil atual
          content:
            application/json:
              schema:
                type: object
                required: [id, name, region, vibrational]
                properties:
                  id: { type: string }
                  name: { type: string }
                  region: { type: string }
                  traits: { type: array, items: { type: string } }
                  vibrational:
                    type: object
                    properties:
                      current_hz: { type: number }
                      mood_bucket: { type: integer, minimum: 0, maximum: 10 }
                      updated_at: { type: string, format: date-time }
```

3.3 Feed — GET /v1/feed/render

```
paths:
  /v1/feed/render:
    get:
      security: [ { bearerAuth: [] } ]
      parameters:
        - in: query
          name: mode
          schema: { type: string, enum: [vibrational, neutral] }
          required: true
      responses:
        "200":
          description: Feed renderizado
          headers:
            Cache-Control:
              schema: { type: string }
          content:
            application/json:
              schema:
                type: object
                required: [items, ttl, entropy]
                properties:
                  items:
                    type: array
```

```

    items:
      type: object
      required: [id, type, payload_json, ttl_seconds]
      properties:
        id: { type: string }
        type: { type: string, enum: [card, tip, mission] }
        payload_json: { type: string } # JSON stringified
        ttl_seconds: { type: integer }
      entropy: { type: number }
      ttl: { type: integer }

```

3.4 Evento — POST /v1/event/checkin (Idempotente)

```

paths:
  /v1/event/checkin:
    post:
      security: [ { bearerAuth: [] } ]
      parameters:
        - in: header
          name: Idempotency-Key
          required: true
          schema: { type: string, format: uuid }
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required: [event_id, geo]
              properties:
                event_id: { type: string, format: uuid }
                geo:
                  type: object
                  required: [lat, lon]
                  properties:
                    lat: { type: number }
                    lon: { type: number }
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                type: object
                properties:
                  status: { type: string, enum: [ok, already_checked] }
                  points: { type: number }
        "409": { description: Chave de idempotência já consumida }

```

3.5 Chat — POST /v1/chat/rooms/{roomId}/messages (Idempotente)

```

paths:
  /v1/chat/rooms/{roomId}/messages:
    post:
      security: [ { bearerAuth: [] } ]
      parameters:
        - in: path
          name: roomId
          required: true
          schema: { type: string }
        - in: header
          name: Idempotency-Key
          required: true
          schema: { type: string, format: uuid }
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required: [message_id, content]
              properties:
                message_id: { type: string }
                content:
                  type: object
                  properties:
                    text: { type: string }
                    attachments: { type: array, items: { type: string, format: uri } }
      responses:
        "201": { description: Persistido e enfileirado para fanout }
        "403": { description: Sem permissão para a sala }
        "409": { description: Idempotência detectada; mensagem já entregue }

```

3.6 Coins — POST /v1/coins/transfer (Idempotente, ledger duplo)

```

paths:
  /v1/coins/transfer:
    post:
      security: [ { bearerAuth: [] } ]
      parameters:
        - in: header
          name: Idempotency-Key
          required: true
          schema: { type: string, format: uuid }
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required: [receiver_id, amount]

```

```

properties:
  receiver_id: { type: string }
  amount: { type: number, minimum: 0.000001 }
  memo: { type: string, maxLength: 140 }
responses:
  "201":
    description: Transferência registrada
    content:
      application/json:
        schema:
          type: object
          required: [tx_id, balance_after]
          properties:
            tx_id: { type: string }
            balance_after: { type: number }
  "400": { description: Saldo insuficiente / validação falhou }
  "409": { description: Chave de idempotência repetida }

```

4) Semântica de Idempotência (servidor)

- **Persistir** `Idempotency-Key` + *hash* da requisição + *resultado* em store rápido (PostgreSQL/Redis).
- **Repetição** com mesma chave e *hash* ⇒ retornar **mesmo resultado** (200/201).
- **Repetição** com mesma chave e *hash diferente* ⇒ **409 Conflict**.
- **Expiração** de chaves: 24–72h (configurável por rota crítica).

Pseudocódigo (server):

```

handleIdem(key, reqHash, action):
  record = idemStore.get(key)
  if record == null:
    result = action()
    idemStore.put(key, { reqHash, result }, ttl=48h)
    return result
  if record.reqHash == reqHash:
    return record.result
  return conflict409()

```

5) Rate Limiting & Quotas

- **Dimensões:** por `user_id`, `client_id`, `route`.
- **Política padrão:** 100 req/min (user), 1000 req/min (client).
- **Rotas sensíveis** (login, transfer, check-in): *burst* limitado (ex.: 10 req/10s).
- **Resposta** ao exceder: **429 Too Many Requests** + `Retry-After` (segundos).
- **Exposição de headers:** `X-RateLimit-Limit`, `X-RateLimit-Remaining`, `X-RateLimit-Reset`.

6) Segurança de API

- **JWT** verificado com JWKS (rotação); *claims*: `sub`, `exp`, `scope`, `aud`, `region`.

- **Scopes mínimos** por rota; *least privilege*.
- **WAF** (borda), **CSP** para Web, **headers** de segurança (HSTS, X-Frame, etc.).
- **Proteção de PII**: mascarar/omitir campos sensíveis em respostas/logs.
- **mTLS (mesh)** em chamadas internas (Camada 10/11).

7) Observabilidade das Rotas

- **Métricas RED por rota**: `req_rate`, `err_rate`, `latency_p95/p99`.
- **Traços** com `trace_id` do gateway → serviço → DB/Kafka.
- **Logs** estruturados: `route`, `status`, `latency_ms`, `user_hash`, `region`, `retry_count`.
- **KPIs de sucesso**:
 - `/v1/feed/render` p95 ≤ **300ms**; erro 5xx ≤ **0.5%**.
 - `/v1/chat/rooms/{id}/messages` p95 ≤ **200ms** (intra-região).
 - `/v1/coins/transfer` **zero** duplicidade (idempotência auditada).

8) Exemplos cURL

Login:

```
curl -sX POST https://api.friendapp.com/v1/auth/login \
-H "Content-Type: application/json" \
-d '{"email":"user@friend.app","password":"S3nh@F0rte!"}'
```

Render Feed:

```
curl -s "https://api.friendapp.com/v1/feed/render?mode=vibrational" \
-H "Authorization: Bearer $JWT"
```

Check-in (idempotente):

```
curl -sX POST https://api.friendapp.com/v1/event/checkin \
-H "Authorization: Bearer $JWT" \
-H "Idempotency-Key: 4f6a1e8c-7d1d-4d7f-8a4f-9f1a2b8c11aa" \
-H "Content-Type: application/json" \
-d '{"event_id":"8f2d5fbe-1b5d-4a30-9b1b-3d3c1e9e7f21","geo":{"lat":-23.5,"lon":-46.6}}'
```

Transfer Coins (idempotente):

```
curl -sX POST https://api.friendapp.com/v1/coins/transfer \
-H "Authorization: Bearer $JWT" \
-H "Idempotency-Key: 0c1d2e3f-4455-6677-8899-aabbccddeeff" \
-H "Content-Type: application/json" \
-d '{"receiver_id":"u-123","amount":25.5,"memo":"gradidão}"'
```

9) Testes de Contrato & Depreciação

- **Spectral** (lint OpenAPI) + **Pact** para *consumer-driven contracts*.
- **Sunset headers** na v1 ao introduzir v2 (≥ 90 dias).
- **Canário** de rotas novas em 5–10% do tráfego antes de liberar geral.

Fechamento da camada:

Com as APIs firmes, idempotentes e observáveis, abrimos a vigilância contínua: o **Monitoramento & Alertas SLO-driven** na próxima camada (Camada 21) para reagir rápido e certo.

◆ CAMADA 21 — MONITORAMENTO & ALERTAS (SLO-DRIVEN)

Entrada:

“Ver é reagir, mas reagir com sabedoria. O FriendApp só aciona quando importa: menos ruído, mais precisão.”

1) Objetivo da Camada

Transformar métricas em **alertas inteligentes**, reduzindo fadiga de incidentes e garantindo que **SLOs** (Service Level Objectives) sejam cumpridos.

👉 Se o erro é tolerável dentro do *error budget*, não precisa alertar.

👉 Se ultrapassa o budget → alerta imediato, com ação clara.

2) SLIs e SLOs (exemplos reais por domínio)

- **Autenticação (Auth Service):**
 - SLI: taxa de erro 5xx.
 - SLO: 99.99% sucesso mensal.
- **Feed Sensorial:**
 - SLI: latência p95 da rota `/v1/feed/render`.
 - SLO: $\leq 300\text{ms}$ em 99,95% das requisições.
- **Chat Realtime:**
 - SLI: RTT p95 de mensagens WebSocket.
 - SLO: $\leq 150\text{ms}$ intra-região.
- **Eventos:**
 - SLI: taxa de check-in aceitos.
 - SLO: $\geq 99,9\%$ de sucesso.
- **IA Aurah Kosmos:**
 - SLI: acurácia de classificação vibracional.
 - SLO: $\geq 92\%$ de assertividade validada.

Error Budget:

- Exemplo Feed: se 0,05% das requisições podem falhar/mês, ultrapassar isso ⇒ **bloqueio de novos deploys**.

3) Sistema de Alertas

- **Ferramentas:** Prometheus Alertmanager + Grafana OnCall + PagerDuty.

- **Runbooks vinculados:** cada alerta possui instruções claras de ação.
- **Rotas:**
 - Sev1 (crítico): PagerDuty + SMS/ligação.
 - Sev2: Slack canal SRE.
 - Sev3+: backlog de observabilidade.

4) Exemplos de Alertas Técnicos

Feed Sensorial — Latência

```
histogram_quantile(0.95, sum(rate(http_latency_bucket{route="/v1/feed/render"}[5m])) by (le))
> 0.3
for: 5m
labels: { severity="sev2", team="feed" }
annotations:
  summary: "Latência feed render p95 > 300ms"
  runbook: "https://runbooks.friendapp.com/feed-latency"
```

Kafka Consumer Lag

```
kafka_consumer_lag{topic="feed.rendered"} > 1000
for: 3m
labels: { severity="sev2", team="infra" }
annotations:
  summary: "Lag crítico em tópico feed.rendered"
```

Hz Médio Regional

```
avg(hz_mean{region="eu-west-1"}[10m]) < 420
for: 10m
labels: { severity="sev1", team="aurah" }
annotations:
  summary: "Colapso vibracional detectado na Europa"
```

5) Runbooks (ação imediata)

- **Feed Latência Alta**
 1. Checar hit-rate Redis/Edge.
 2. Revalidar TTLs e caches.
 3. Se normal não voltar em 15 min → escalar para engenheiro responsável.
- **Kafka Lag**
 1. Checar consumers ativos (HPA/KEDA).
 2. Forçar rebalance de partições.
 3. Avaliar DLQ.
- **Hz Regional Baixo**

1. Ativar **modo protetor** (reduzir estímulos).
2. Shift parcial de tráfego para região backup.
3. Notificar IA Aurah Kosmos.

6) Observabilidade dos Alertas

- **Dashboards em Grafana:** visão global de alertas ativos, resolvidos, MTTA (tempo médio para agir) e MTTR (tempo médio de resolução).
- **Relatórios pós-incidente:** revisões semanais, sem culpa, focadas em melhorias.

7) Benefícios

- Redução de **alert fatigue**: apenas alertas que impactam SLO acionam.
- Equipes trabalham com **confiança** de que incidentes críticos não passam despercebidos.
- A arquitetura vibracional é protegida não só por código, mas por atenção contínua.

Fechamento da camada:

Com monitoramento e alertas inteligentes, o FriendApp reage com precisão. Agora seguimos para os registros que dão memória ao sistema: **Logs Técnicos & Vibracionais** (Camada 22).

◆ CAMADA 22 — LOGS TÉCNICOS & VIBRACIONAIS (ESTRUTURA / PRIVACIDADE)

Entrada:

"Sem memória não há aprendizado. O FriendApp registra cada pulso, mas com respeito à privacidade e à energia do usuário."

1) Objetivo

- Manter **rastreabilidade completa** das ações técnicas e vibracionais.
- Permitir **debug, auditoria e aprendizado da IA Aurah Kosmos**.
- Cumprir normas de **privacidade (LGPD/GDPR)**, evitando exposição de dados sensíveis.

2) Estrutura de Logs Técnicos

- **Formato canônico (JSON):**

```
{
  "ts": "2025-09-13T23:59:59Z",
  "level": "INFO",
  "service": "ms-feed",
  "route": "/v1/feed/render",
  "status": 200,
  "latency_ms": 128,
  "trace_id": "t-abc123",
  "span_id": "s-456",
  "user_hash": "u_9f3d...",
  "region": "sa-east-1",
  "retries": 0,
  "infra": { "pod": "feed-12", "node": "k8s-sa-east-01" }
```

```
}
```

- **Campos obrigatórios:** `ts`, `level`, `service`, `trace_id`.
- **Sem PII:** todos os identificadores são pseudo (`user_hash`).
- **Contexto de infra:** pod, node, versão do deploy.
- **Armazenamento:** Datadog/CloudWatch + export para BigQuery (amostrado).

3) Estrutura de Logs Vibracionais

- **Campos principais:**

```
{
  "ts": "2025-09-13T23:59:59Z",
  "user_hash": "u_9f3d...",
  "region": "sa-east-1",
  "hz_bucket": 6,
  "mood_delta": -0.2,
  "entropy": 0.23,
  "action_type": "feed_view",
  "trace_id": "t-xyz789"
}
```

- `hz_bucket` (**0..10**): nível vibracional estimado.
- `mood_delta`: variação emocional detectada.
- `entropy`: índice de desordem vibracional.
- `action_type`: interação (ex.: `feed_view`, `chat_msg`, `ra_session`).
- **Uso:** análise coletiva, padrões de colapso, insights para Aurah Kosmos.

4) Retenção & Privacidade

- **Logs técnicos:** 180 dias → arquivados em storage frio por 12 meses.
- **Logs vibracionais:** 90 dias → pseudonimizados após 30 dias.
- **Expurgo em cascata:** se usuário solicita exclusão, todos os logs são removidos ou anonimizados.
- **Acesso:** controlado por RBAC; apenas equipes autorizadas.
- **Auditoria:** todas consultas registradas.

5) Observabilidade dos Logs

- **Indexação:** Elasticsearch/Datadog com filtros por `service`, `trace_id`, `user_hash`, `region`.
- **KPIs monitorados:**
 - Taxa de erro 5xx por serviço.
 - Latência média por rota.
 - Frequência de colapsos vibracionais regionais.
- **Alertas automáticos:**
 - Anomalias em logs vibracionais (ex.: queda coletiva de Hz).

- Aumento abrupto de erros em serviço crítico.

6) Segurança dos Logs

- **Criptografia:** AES-256 at-rest + TLS in-transit.
- **Scrubbing:** campos sensíveis removidos (e-mail, telefone, dados de pagamento).
- **Tokenização:** substitui dados pessoais por hashes não reversíveis.
- **Conformidade:** LGPD, GDPR, ISO 27001.

7) Pseudocódigo de Escrita de Log

```
function logEvent(service, level, context):  
  entry = {  
    ts: now(),  
    level: level,  
    service: service,  
    trace_id: current_trace(),  
    user_hash: sha256(user_id),  
    region: context.region,  
    payload: scrub(context.payload)  
  }  
  sendToCollector(entry)
```

8) Benefícios

- Transparência operacional.
- Aprendizado contínuo da IA com base em padrões reais.
- Proteção da privacidade com anonimização vibracional.

Fechamento da camada:

Com os registros técnicos e vibracionais garantidos, o FriendApp possui memória confiável. Agora avançamos para a proteção contra perda: **Backups + Versionamento** (Camada 23).

◆ CAMADA 23 — BACKUPS + VERSIONAMENTO

Entrada:

"Quando tudo falha, nada pode se perder. Cada dado, cada pulso vibracional precisa estar protegido e versionado."

1) Objetivo

- Garantir **RPO (Recovery Point Objective)** de minutos e **RTO (Recovery Time Objective)** de até 1h em desastres.
- Assegurar **consistência global** dos dados híbridos (PostgreSQL, Firestore, Neo4j, Redis, BigQuery).
- Implementar versionamento em código, banco e contratos de API/eventos.

2) Políticas de Backup por Banco

Banco / Store	Estratégia	Frequência	RPO	RTO
PostgreSQL	PITR (Write Ahead Log) + Snapshots	WAL contínuo + full 6/6h	≤5 min	≤60 min
Firestore	Exportação automática p/ CloudStorage multi-região	15 min	≤5 min	≤15 min
Neo4j	Backup incremental + full semanal	12/12h + 1x/semana	≤10 min	≤60–120 min
Redis	AOF (Append Only File) + replicação	Contínuo	instantâneo	≤5 min
BigQuery	Snapshots de tabelas particionadas	24h	≤24h	≤15 min
Objetos (S3/GCS/Azure Blob)	Versionamento habilitado + replicação cross-region	contínuo	≤15 min	≤60 min

3) Testes de Disaster Recovery (DR)

- **Trimestral:** simulação de falha de região completa (AWS `sa-east-1` offline).
- **Procedimento:** restaurar PG/Redis em região backup, reidratar Firestore via export, validar consistência de eventos Kafka com MirrorMaker.
- **Meta:** recuperar ambiente completo ≤ 60 min.
- **Checklist pós-testes:** gaps e SLIs monitorados.

4) Versionamento de Schemas e Contratos

- **PostgreSQL:** migrações Flyway/Liquibase (`db-vib-YYYY.MM.N`).
- **Neo4j:** labels e constraints versionados em repositório Git.
- **Firestore:** versionamento lógico via coleções (`tests_v1` , `tests_v2`).
- **Kafka Events:** versionamento de tópicos (`user.created.v1` , `user.created.v2`).
- **OpenAPI/gRPC:** versionamento major (`/v1` , `/v2`).
- **Governança:** todo schema armazenado em **repositório Git central** (`infra-schemas`).

5) Snapshots de Infraestrutura

- **Kubernetes:**
 - Backup de `etcd` semanal (quando não gerenciado).
 - Manifests versionados em GitOps (ArgoCD/Flux).
- **Configurações de Vault/Secrets:** snapshots encriptados armazenados em storage multi-cloud.
- **Dashboards Grafana:** export JSON versionado em Git.

6) Observabilidade dos Backups

- **Dashboards:** status de cada job (sucesso/falha), tempo médio de restauração.
- **Alertas:**
 - Falha de job >1 ciclo consecutivo.
 - RPO acima do esperado (>15 min para PG/Firestore).
- **Auditoria:** logs de backup e restauração armazenados em BigQuery.

7) Pseudocódigo de Workflow de Backup

```
dailyBackup():
  for db in [PostgreSQL, Firestore, Neo4j, Redis, BigQuery]:
    snapshot = db.createBackup()
    encrypt(snapshot, key=KMS)
    replicate(snapshot, regions=["us-east1","europe-west1"])
    log("Backup", db, snapshot.id, status="success")
```

8) KPIs de Sucesso

- **RPO médio:** ≤ 5 min.
- **RTO médio:** ≤ 60 min.
- **Taxa de sucesso em restaurações simuladas:** ≥ 98%.
- **Jobs falhos recuperados em** ≤ 24h.

Fechamento da camada:

Com backups e versionamento blindados, o FriendApp garante memória e resiliência. A próxima camada abre espaço para workloads que aparecem e desaparecem: **Serverless & Jobs Event-Driven (Camada 24)**.

◆ CAMADA 24 — SERVERLESS & JOBS EVENT-DRIVEN

Entrada:

"Nem tudo precisa estar sempre vivo. Algumas energias só aparecem quando chamadas e desaparecem logo depois."

1) Objetivo

Implementar **funções serverless e jobs event-driven** para workloads **intermitentes, esporádicos ou não críticos**, reduzindo custos e simplificando operação.

2) Casos de Uso Serverless

- **Recompensas e FriendCoins:** engine de distribuição de bônus (diário, missões concluídas).
- **Webhooks externos:** integração com parceiros (pagamentos, eventos sociais).
- **Notificações vibracionais:** envio de push/sms/email em resposta a eventos de alta vibração.
- **Transcodificação leve:** conversão de mídias curtas em RA ou stickers de chat.
- **Tasks administrativas:** limpeza de sessões, validação de cadastros, expurgo de dados expirados.

3) Stack Técnico

- **AWS Lambda / GCP Cloud Functions / Azure Functions** (dependendo da região/nuvem).
- **Orquestração:**
 - **EventBridge (AWS)** ou **Pub/Sub (GCP)** para gatilhos programados.
 - **KEDA** em clusters Kubernetes → escala pods serverless baseados em filas ou tópicos Kafka.
- **Observabilidade:** logs → CloudWatch/Stackdriver; métricas → Prometheus adaptadores.

4) Gatilhos (Triggers)

- **Eventos Kafka:** ex.: `coins.mission_completed` → dispara função de recompensa.
- **Timers:** rotinas agendadas (cron) para manutenção (TTL cleanup, backup verificação).
- **HTTP Webhooks:** endpoints serverless para integrações externas seguras.
- **Buckets/Object Storage:** upload de arquivo → trigger de conversão/validação.

5) Padrões Arquiteturais

- **Stateless:** funções sem dependência de estado local.
- **Cold Start:** mitigado por provisionamento mínimo em workloads sensíveis.
- **Idempotência:** mesma chave de evento não pode gerar múltiplas execuções.
- **Retries:** automáticos em falhas transitórias; DLQ para falhas definitivas.
- **Segurança:** autenticação JWT para webhooks; segredos injetados via Vault.

6) Exemplo de Workflow — Recompensa de Missão

1. Usuário completa missão → evento `mission.completed` publicado em Kafka.
2. Função serverless consome evento, valida usuário e calcula recompensa.
3. Chama microserviço Coins via API gRPC/REST com `Idempotency-Key`.
4. Publica evento `coins.reward_distributed`.
5. Envia notificação vibracional via push.

Pseudocódigo:

```
onEvent("mission.completed", e):
  if seen(e.idempotency_key): return
  reward = calcReward(e.user_id, e.mission_id)
  callAPI("/v1/coins/transfer", {receiver:e.user_id, amount:reward},
    headers={Idempotency-Key: e.idempotency_key})
  publish("coins.reward_distributed", {user_id:e.user_id, reward:reward})
  sendPush(e.user_id, "Parabéns, recompensa entregue!")
```

7) Observabilidade & Custos

- **Logs estruturados:** execução, erro, duração, `trace_id`.
- **Métricas principais:**
 - Execuções por segundo.
 - Latência média.
 - Taxa de erro (%).
 - Custo por execução.
- **KPIs alvo:**
 - P95 execução ≤ 1s.
 - Erros < 0,5%.
 - Custo por 1M execuções ≤ US\$ 15 (dependendo da nuvem).

8) Benefícios

- **Escala automática:** funções aparecem apenas sob demanda.
- **Custo zero** em inatividade.
- **Integração simples:** ótimo para integrações externas.
- **Segurança:** isolamento natural por execução.

Fechamento da camada:

Com jobs serverless e event-driven, o FriendApp responde rápido e barato aos pulsos inesperados. Agora seguimos para a visão global e centralizada: o **Painel de Observabilidade Global** (Camada 25).

◆ CAMADA 25 — PAINEL DE OBSERVABILIDADE GLOBAL (ADMIN)

Entrada:

"Ver é compreender. O painel é o terceiro olho do FriendApp: visão total, técnica e vibracional."

1) Objetivo

Oferecer aos times técnicos e de produto uma **visão centralizada** de:

- Saúde de microserviços e clusters.
- Tráfego global, latência e disponibilidade.
- Eventos vibracionais regionais (colapsos, expansões).
- Custos, alertas e incidentes em tempo real.

2) Stack de Observabilidade

- **Grafana** (painéis dinâmicos, drill-down).
- **Prometheus + Thanos** (métricas + histórico longo).
- **Datadog** (APM + logs + traces unificados).
- **ELK (ElasticSearch, Logstash, Kibana)** para busca avançada.
- **Jaeger / OpenTelemetry** para traces distribuídos.

3) Estrutura do Painel

1. Visão Global (Mapa Planetário)

- Clusters ativos (AWS/GCP/Azure).
- Latência média p95 por região.
- Status vibracional médio (Hz bucket global).
- Eventos de failover em tempo real.

2. Serviços Críticos (cards)

- Auth, Feed, Chat, Eventos, Aurah Kosmos, Coins.
- Cada card: **latência p95**, **taxa de erro 5xx**, **CPU/mem**.

3. Logs & Alertas

- Últimos incidentes abertos.
- Taxa de alertas ativos por severidade.

- Drill-down por trace_id.

4. Vibração Coletiva (IA Aurah Kosmos)

- Média de Hz por continente.
- Anomalias detectadas (quedas súbitas).
- Intervenções automáticas ("modo protetor").

5. Custos & FinOps

- Custo diário por serviço/nuvem.
- Projeção mensal vs orçamento.
- Alertas de overspend.

4) KPIs Monitorados

- **Infraestrutura:** uptime global $\geq 99,99\%$.
- **Feed:** latência p95 $\leq 300\text{ms}$.
- **Chat Realtime:** RTT p95 $\leq 150\text{ms}$.
- **Aurah Kosmos:** $\geq 92\%$ de assertividade vibracional.
- **FinOps:** custo infra/receita $\leq 30\%$ inicial $\rightarrow \leq 15\%$ maturidade.

5) Segurança e Acesso

- **RBAC estrito:**
 - **Engenheiros:** métricas + logs técnicos.
 - **Produto:** métricas de uso/vibração (sem PII).
 - **Admin:** visão completa.
- **Auditoria:** toda visualização registrada em log.
- **2FA obrigatório** no acesso.

6) Integrações

- **Webhook \rightarrow Slack:** alertas de Sev1/Sev2.
- **PagerDuty:** incidentes críticos.
- **BigQuery:** relatórios semanais (técnico + vibracional).
- **Export APIs:** painéis expostos via `/v1/observability/export`.

7) Exemplo de Widget (Grafana JSON)

```
{
  "title": "Feed Latency p95",
  "type": "timeseries",
  "targets": [
    {
      "expr": "histogram_quantile(0.95, sum(rate(http_latency_bucket{route=\"/v1/feed/render\"}[5m])) by (l
e))",
      "legendFormat": "Feed Render Latency"
    }
  ]
}
```

```
]
}
```

8) Benefícios

- Visão única, centralizada e planetária.
- Redução de MTTA/MTTR em incidentes.
- Transparência vibracional + técnica.
- Sustentação da confiança dos usuários e investidores.

Fechamento da camada:

Com o Painel de Observabilidade Global, a arquitetura do FriendApp ganha olhos em todas as direções. Aqui fechamos o Macrogrupo 3, entregando não apenas execução, mas também visão contínua. Na sequência, adicionaremos a seção obrigatória de **Integrações e Dependências Cíclicas do Ecossistema**.

SEÇÃO FINAL — INTEGRAÇÕES E DEPENDÊNCIAS CÍCLICAS DO ECOSSISTEMA



Entrada vibracional:

"Nada existe isolado. Cada sistema é parte de uma malha viva: depende e alimenta, recebe e entrega."

1) Tabela de Integrações

Sistema	Depende de	Envia para	Modo Técnico	Dados / Função
IA Aurah Kosmos 	Perfil & Frequência, BigQuery, Logs vibracionais	Feed, Chat, Matching, Modo protetor	REST/gRPC + Kafka	Hz médio, padrões emocionais, insights, triggers protetivos
Feed Sensorial 	Aurah, Perfil, Firestore, Redis	App, Logs, Matching	API REST + Cache + Kafka	Conteúdos, cards, entropia, TTLs
Chat Vibracional 	Auth, WS Broker, Redis	Logs, Moderador, Feed indireto	WebSocket + Kafka	Mensagens, sinais de presença, RA sync
Eventos & Viagem 	Auth, Localização, Perfil	App, Feed, RA, BigQuery	REST + Kafka	Check-ins, presenças, hotspots vibracionais
RA & Hotspots 	Firestore, Geo, IA Aurah	Feed, App, Logs	API REST + gRPC	Sessões RA, overlays, mensagens ocultas
FriendCoins 	Auth, Ledger (Postgres)	Feed, Jogo, Recompensas	REST/gRPC	Transações, saldos, ledger
Jogo da Transmutação 	Perfil, Aurah, Coins	Feed, Matching, Logs	REST + Kafka	Missões, desafios, status vibracional
Observabilidade Global 	Todos serviços + métricas Prometheus	Painel Admin, Relatórios	Exporters + BigQuery	SLIs, logs, métricas, alertas
Moderador & Segurança 	Logs técnicos + vibracionais	Feed, Chat, Eventos	REST/gRPC + AI semântica	Flags de abuso, bloqueios, modo protetor

2) Legenda Visual

-  **Banco de dados** (Postgres, Firestore, Redis, Neo4j, BigQuery).
-  **IA Aurah** (processamento vibracional central).

- 🧩 **Microserviço** (Feed, Chat, Coins etc.).
 - 🔗 **API REST/gRPC** (chamadas síncronas).
 - 📬 **Evento Kafka** (com contrato Avro/Registry).
 - 🛡️ **Segurança** (Vault, WAF, TLS, Zero Trust).
-

3) Ciclo de Dependências

- **Aurah Kosmos** recebe vibração → alimenta Feed/Chat/Matching.
- **Feed** entrega experiência → envia logs → retroalimenta Aurah.
- **Chat/RA/Eventos** produzem dados sensoriais → viram sinais vibracionais no BigQuery.
- **BigQuery** retroalimenta Aurah Kosmos com padrões coletivos.
- **Coins & Jogo** usam perfil vibracional para recompensas → que retornam ao Feed.
- **Observabilidade** observa tudo → alimenta painel global → ajusta SLOs.

📌 Resultado: um **circuito fechado**, vivo, em que **todos os sistemas dependem e alimentam uns aos outros**.

4) Benefícios

- Nenhum dev pergunta mais: *"Esse sistema depende de quê?"*
 - Integrações explícitas, visíveis, auditáveis.
 - Fluxo vibracional + técnico mapeado do início ao fim.
 - Garantia de **consistência, escalabilidade e clareza** para toda a equipe.
-

Fechamento Final do Manual:

"Este documento não é apenas uma descrição técnica: é a espinha dorsal de um organismo vivo, consciente e planetário. Cada camada aqui descrita é executável, auditável e interdependente. O FriendApp pulsa porque sua arquitetura pulsa."