

# Pré-Projeto — Arquitetura 1: App Escalável com Balanceador + Banco Único

**Escolha de arquitetura:** *Scale-out* na camada de aplicação (múltiplas instâncias atrás de um **Load Balancer**) com **um banco de dados único gerenciado** (Postgres).

---

## 1) Resumo do Sistema

Entregar um pequeno backend (API REST) para uma entidade simples (ex.: `Usuario` e `Tarefa`) capaz de:

- Atender **concorrentemente** por múltiplas instâncias de aplicação;
  - Persistir dados em **um banco relacional** gerenciado;
  - Expor **métricas/logs** que evidenciem balanceamento e disponibilidade.
- 

## 2) Por que essa arquitetura?

- **Simplicidade máxima:** menos peças, menor chance de erro; curva de aprendizado curta.
  - **Distribuição suficiente para a disciplina:** várias instâncias atendendo em paralelo sob um **LB** com *health checks*.
  - **Boa narrativa didática:** mostra *scaling out*, **stateless**, pooling de conexões e limites práticos do DB.
  - **Viável no free tier:** PaaS gratuitas entregam HTTPS, LB e deploy fácil.
- 

## 3) Visão Lógica (alto nível)

Internet



[ Load Balancer / Gateway (gerenciado) ]

——► A1: API (stateless)

——► A2: API (stateless) (+ A3 se necessário)



——► Banco de Dados Único (Postgres gerenciado)

└(opcional) Read Replica para leituras

(opcional) Cache (Redis gerenciado) entre A\* e o DB

## Decisões-chave

- **Aplicação stateless:** sessões via JWT/cookie assinado → instâncias intercambiáveis.
  - **Banco único gerenciado:** ACID, backups automáticos; reduz complexidade.
  - **Observabilidade:** logs/metrics por instância para evidenciar o balanceamento.
- 

## 4) Componentes e Responsabilidades

- **Load Balancer/Gateway (plataforma):**
    - Distribuição (round-robin), *health checks*, TLS/HTTPS, *rate limiting* simples.
  - **Aplicações A1, A2 (API):**
    - Endpoints CRUD ( /usuarios , /tarefas ), validação, autenticação simples (JWT), *retry* leve para falhas transitórias.
  - **Banco de Dados (Postgres):**
    - Tabelas com chaves/índices essenciais; backups; política de *connection pooling*.
  - **Cache (opcional):**
    - Redis gerenciado para leituras quentes (TTL curto).
  - **Observabilidade:**
    - Logs estruturados; métricas de requisição, erro 5xx, latência p95; endpoint /healthz .
- 

## 5) Benefícios

- **Entrega rápida** e colaborativa (dupla trabalha em paralelo).
  - **Escala horizontal** na camada web/API.
  - **Resiliência prática:** queda de A1 não derruba o serviço (A2 continua).
  - **Custo baixo/zero** no free tier.
- 

## 6) Limitações (assumidas e justificadas)

- **DB como potencial gargalo/ponto único lógico:** aceito para MVP; mitigado por índices, cache e *read replica* opcional.
  - **Sem sharding/filas/eventos:** fora do escopo do pré-projeto para manter simplicidade.
  - **Quotas do free tier:** monitorar consumo e ajustar limites.
-

## 7) Serviços/Funcionalidades (Escopo)

1. API REST **stateless** para `Usuario` e `Tarefa`.
  2. Autenticação simples (JWT) e autorização básica.
  3. Validações (e.g., email único; limites por usuário).
  4. Logs & Métricas: requisições por instância (A1/A2), latência p95, taxa de erro.
  5. Health checks (`/healthz`) para remoção automática de instâncias pelo LB.
  6. (Opcional) Cache de leitura (Redis) para endpoints mais acessados.
- 

## 8) Riscos e Mitigações

- Sobrecarga no DB: índices corretos, *connection pooling* e cache.
  - Cold start (serverless): manter instância “quente” ou usar PaaS *always-on*.
  - Falhas intermitentes: *retry* exponencial na API + *timeouts* configurados.
- 

## 9) Critérios de Avaliação (propostos)

- Evidência de **balanceamento** (logs/metrics mostram A1 e A2 atendendo).
  - **Disponibilidade** sob falha simulada de uma instância.
  - Boas práticas: **stateless**, pooling, índices, HTTPS, variáveis de ambiente.
  - Clareza dos **trade-offs** e justificativas arquiteturais.
- 

## 10) Plataformas Gratuitas (curva de aprendizagem rápida)

- API / LB
    - **Vercel** (Serverless Functions) — deploy com 1 clique, HTTPS e observabilidade nativa.
    - **Render** (Web Service free) ou **Railway** — processos *always-on* simples.
  - Banco de Dados
    - **Neon** (Postgres serverless, free tier com autosleep).
    - **Supabase** (Postgres gerenciado + UI web para tabelas/SQL).
  - Observabilidade
    - **Sentry** (erros) e **Grafana Cloud Free** (dashboards) ou logs nativos da plataforma.
-

## 11) Organização da Dupla

- **Pessoa A (API)**: endpoints, autenticação, logs/métricas.
  - **Pessoa B (Dados/Infra)**: modelagem SQL, índices, provisionamento do Postgres, políticas de pool e *health checks*.
  - **Integração**: contrato de API e esquema SQL versionados; PRs curtas com revisão.
- 

## 12) Roadmap Conceitual

- **Fase 1 — Arquitetura & Contratos**: diagramas, modelo de dados, políticas (stateless, JWT, pooling).
  - **Fase 2 — Funcional**: CRUD + autenticação + observabilidade + health checks.
  - **Fase 3 — Qualidade**: cache opcional, índices e teste de *failover* (desligar A1 e observar A2).
  - **Fase 4 — Apresentação**: prints/gráficos de métricas evidenciando o balanceamento.
- 

## Anexo — Padrões e Decisões (resumo)

- **Stateless & Idempotência** em endpoints → facilita *retries* e escalabilidade.
  - **Pool de conexões** para respeitar limites do free tier.
  - **Índices essenciais** (e.g., `usuarios(email) UNIQUE`, `tarefas(usuario_id)` ).
  - **Segurança mínima**: HTTPS, secrets em variáveis de ambiente, CORS restrito.
-