Architecture Decision Records - Gerenciador de Oficina

Projeto: Gerenciador de Oficina - Fase 3
Autor: Manus AI
Data de Criação: 23 de janeiro de 2025
Status: Documentação Oficial

Índice

Introdução

Este documento consolida as decisões arquiteturais críticas do projeto Gerenciador de Oficina - Fase 3. O projeto evoluiu para uma arquitetura distribuída e cloud-native,

composta por múltiplos repositórios independentes, cada um com seu próprio ciclo de vida e responsabilidades bem definidas.

A arquitetura foi projetada para suportar escalabilidade horizontal, alta disponibilidade, segurança enterprise-grade e facilidade de manutenção. As decisões documentadas refletem a avaliação cuidadosa de alternativas e o alinhamento com as melhores práticas da indústria para sistemas distribuídos modernos.

ADR-001: Arquitetura Multi-Repositório

Data: 23 de janeiro de 2025
Status: Aceito
Decisores: Equipe de Arquitetura
Categoria: Arquitetura

Contexto

O projeto Gerenciador de Oficina - Fase 3 evoluiu para uma arquitetura distribuída com múltiplos componentes independentes: aplicação core, funções serverless, infraestrutura Kubernetes, banco de dados e API Gateway. A decisão sobre como organizar o código-fonte e a infraestrutura impacta diretamente a manutenibilidade, velocidade de desenvolvimento, CI/CD e autonomia das equipes.

Precisávamos definir se manteríamos todos os componentes em um único repositório monolítico (monorepo) ou dividiríamos em múltiplos repositórios independentes (multi-repo).

Decisão

Adotamos uma **arquitetura multi-repositório** com cinco repositórios independentes:

Repositório	Descrição	Responsabilidade
gerenciador-oficina-core-fase-3	Aplicação principal	Regras de negócio, APIs REST, integração com módulos
gerenciador-oficina-lambda-fase-3	Funções serverless	Processamento assíncrono, notificações, automações event-driven
gerenciador-oficina-k8s-infra-fase-3	Manifests Kubernetes	Deployments, services, ingress, autoscaling
gerenciador-oficina-db-infra-fase-3	Infraestrutura do banco	RDS PostgreSQL, versionamento com Terraform
gerenciador-oficina-api-gateway-infra-fase-3	API Gateway	Rate limiting, roteamento, monitoramento

Cada repositório possui seu próprio ciclo de vida, versionamento semântico, pipeline CI/CD e documentação específica.

Consequências

Positivas:

- **Autonomia de equipes**: Times diferentes podem trabalhar em módulos distintos sem conflitos de merge, permitindo paralelização de desenvolvimento
- **Deploy independente**: Cada componente pode ser versionado e deployado separadamente, reduzindo risco de falhas em cascata
- **CI/CD otimizado**: Pipelines executam apenas para o código modificado, reduzindo tempo de build e feedback mais rápido
- **Segurança granular**: Permissões de acesso podem ser configuradas por repositório, implementando princípio de least privilege
- **Clareza de responsabilidades**: Cada repositório tem um propósito bem definido, facilitando onboarding de novos desenvolvedores
- **Facilita manutenção**: Mudanças em infraestrutura não afetam código da aplicação, reduzindo risco de regressões

Negativas:

- **Complexidade de coordenação**: Mudanças que afetam múltiplos repositórios requerem sincronização cuidadosa entre equipes
- **Duplicação de configurações**: Alguns arquivos de configuração (ex: .gitignore, dependências comuns) podem ser replicados
- **Gestão de dependências**: Versionamento entre módulos precisa ser gerenciado manualmente através de tags e releases
- **Curva de aprendizado**: Novos desenvolvedores precisam entender a estrutura distribuída e como os componentes se integram

Alternativas Consideradas

Monorepo: Manteria todo o código em um único repositório com estrutura de diretórios. Refatorações cross-module seriam mais simples, mas pipelines CI/CD ficariam lentos (build de todo o código sempre), aumentando risco de conflitos de merge e dificultando aplicação de permissões granulares.

Monorepo com ferramentas especializadas (Nx, Turborepo): Melhoraria gerenciamento de dependências internas, mas adicionaria complexidade de tooling não justificada para o tamanho atual do projeto e não é padrão no ecossistema Java/Spring Boot.

Notas Adicionais

- Integração entre repositórios é feita via pipelines CI/CD e configurações declarativas
- Versionamento segue Semantic Versioning 2.0.0
- Tags Git são usadas para releases coordenadas entre módulos
- Cada repositório mantém sua própria documentação README.md

ADR-002: Stack Backend Java/Spring Boot

Data: 23 de janeiro de 2025
Status: Aceito
Decisores: Equipe de Desenvolvimento Backend
Categoria: Backend

Contexto

O Gerenciador de Oficina requer uma API REST robusta para gerenciar autenticação, controle de estoque, ordens de serviço e integração com sistemas externos. A escolha da linguagem de programação e framework backend é fundamental para a produtividade, manutenibilidade, performance e disponibilidade de recursos humanos.

Precisávamos de uma solução que oferecesse ecossistema maduro, suporte robusto para APIs REST, integração nativa com bancos de dados relacionais, segurança enterprise-grade, facilidade de containerização e disponibilidade de desenvolvedores no mercado.

Decisão

Adotamos **Java 17+ com Spring Boot 3.3** como stack principal do backend, incluindo:

Componente	Versão	Propósito
Java	17+ (LTS)	Linguagem principal com features modernas
Spring Boot	3.3	Framework para desenvolvimento rápido
Spring Security	Latest	Autenticação e autorização
Spring Data JPA	Latest	Persistência e ORM
Hibernate	Latest	Mapeamento objeto-relacional
Spring Web	Latest	APIs REST
OpenAPI/Swagger	Latest	Documentação automática de APIs

Consequências

Positivas:

- **Ecossistema maduro**: Vasta documentação, bibliotecas de terceiros e comunidade ativa com soluções para praticamente qualquer problema
- **Produtividade**: Spring Boot reduz boilerplate significativamente através de auto-configuração e starters, acelerando desenvolvimento

- **Segurança robusta**: Spring Security é padrão de mercado para autenticação/autorização em aplicações enterprise
- **Performance**: JVM otimizada com garbage collection sofisticado, suporte a programação reativa (WebFlux) se necessário
- **Integração nativa**: JPA/Hibernate simplifica persistência em PostgreSQL com mapeamento automático de entidades
- **Testabilidade**: Mockito e Spring Test facilitam testes unitários, de integração e end-to-end
- **Containerização**: Imagens Docker otimizadas disponíveis (Eclipse Temurin), com suporte a GraalVM Native Image
- **Mercado**: Grande disponibilidade de desenvolvedores Java/Spring Boot, facilitando contratação e escalabilidade de equipe

Negativas:

- **Consumo de memória**: JVM consome mais memória que linguagens nativas (Go, Rust), impactando custos de infraestrutura
- **Tempo de startup**: Aplicações Spring Boot têm startup mais lento que frameworks minimalistas, afetando cold starts em serverless
- **Curva de aprendizado**: Spring tem muitas abstrações e "magia" (annotations, AOP), exigindo tempo para dominar
- **Verbosidade**: Java é mais verboso que linguagens modernas (Kotlin, Go), aumentando volume de código

Alternativas Consideradas

Node.js + Express/NestJS: Startup rápido, ecossistema NPM vasto, JavaScript/TypeScript unificado. Rejeitado porque menos robusto para aplicações enterprise complexas, tipagem mais fraca e ORM menos maduro.

Python + FastAPI/Django: Sintaxe simples, excelente para prototipagem. Rejeitado porque GIL limita concorrência real, menos adotado em sistemas transacionais críticos, tipagem dinâmica aumenta riscos em produção.

Go + Gin/Echo: Performance excelente, baixo consumo de memória, concorrência nativa. Rejeitado porque ecossistema menos maduro para aplicações enterprise, menor disponibilidade de desenvolvedores, ORM menos robusto.

Kotlin + Spring Boot: Sintaxe moderna, interoperabilidade total com Java. Rejeitado porque equipe já domina Java e não há necessidade crítica de features Kotlin no momento.

Notas Adicionais

- Java 17 é versão LTS com suporte até setembro de 2029
- Spring Boot 3.3 requer Java 17+ e oferece suporte a GraalVM Native Image para futuro
- Possibilidade de migração incremental para Kotlin se necessário
- Spring Boot 3.x tem melhorias significativas de performance e observabilidade

---

ADR-003: Estratégia de Autenticação JWT

Data: 23 de janeiro de 2025
Status: Aceito
Decisores: Equipe de Segurança e Backend
Categoria: Segurança

Contexto

O sistema de gerenciamento de oficina requer autenticação segura para diferentes tipos de usuários (administradores, mecânicos, atendentes), cada um com permissões específicas. A aplicação é uma API REST stateless que será consumida por múltiplos clientes (web, mobile, integrações de terceiros). A estratégia de autenticação deve ser escalável, segura e compatível com arquiteturas distribuídas e microsserviços.

Decisão

Implementamos **autenticação baseada em JSON Web Tokens (JWT)** com as seguintes características:

A autenticação utiliza tokens JWT assinados com algoritmo HMAC-SHA256 (HS256), contendo claims customizados para identificação do usuário, roles e permissões. Os tokens são emitidos pelo endpoint de login após validação de credenciais e incluem tempo de expiração configurável. O Spring Security foi configurado com filtros

customizados para interceptar requisições, validar tokens JWT no header Authorization (formato Bearer) e popular o contexto de segurança.

A estratégia inclui dois tipos de tokens:

Tipo	Vida Útil	Propósito
Access Token	15 minutos	Autorização de requisições
Refresh Token	7 dias	Renovação de access tokens

Refresh tokens são armazenados no banco de dados PostgreSQL para permitir renovação de tokens expirados sem necessidade de reautenticação completa. Tokens são validados em cada requisição através de verificação de assinatura, expiração e revogação (blacklist em cache Redis para logout).

Consequências

Positivas:

- **Stateless e escalável**: Aplicação não mantém estado de sessão no servidor, permitindo escalabilidade horizontal sem sticky sessions ou compartilhamento de sessões entre instâncias Kubernetes
- **Performance superior**: Validação de tokens JWT é computacionalmente eficiente, envolvendo apenas verificação criptográfica local sem consultas ao banco em cada requisição
- **Compatibilidade com microsserviços**: Tokens JWT podem ser compartilhados entre diferentes serviços (core, Lambdas) sem centralização de sessões
- **Padrão de mercado**: JWT é amplamente adotado, suportado por bibliotecas maduras em todas as linguagens
- **Suporte nativo no Spring Security**: Framework oferece abstrações robustas, reduzindo código boilerplate e riscos de vulnerabilidades

Negativas:

- **Impossibilidade de revogação imediata**: Token JWT permanece válido até expiração, mesmo se usuário for desativado ou permissões alteradas (mitigado com blacklist Redis)

- **Tamanho dos tokens**: JWTs são maiores que session IDs, aumentando overhead de cada requisição HTTP
- **Gerenciamento de chaves**: Segurança depende totalmente da proteção da chave secreta usada para assinar tokens
- **Complexidade de refresh**: Implementação de refresh tokens adiciona complexidade, requerendo armazenamento persistente e lógica de renovação

Alternativas Consideradas

Sessões baseadas em cookies: Armazenaria estado no servidor em memória ou banco de dados, enviando session ID via cookie HTTP-only. Rejeitado porque requer sticky sessions ou armazenamento compartilhado (Redis) em ambientes distribuídos, adicionando complexidade de infraestrutura. Escalabilidade horizontal seria comprometida e haveria maior carga no banco de dados para validação de cada requisição.

OAuth 2.0 com servidor externo: Delegaria autenticação para provedores como Keycloak, Auth0 ou AWS Cognito. Rejeitado devido à complexidade adicional de infraestrutura e dependência de serviços externos, aumentando custos operacionais. Para o escopo atual, seria over-engineering sem necessidade de SSO ou integrações complexas.

Autenticação básica HTTP: Envio de credenciais em Base64 no header Authorization de cada requisição. Rejeitado imediatamente por razões de segurança: credenciais transmitidas em cada requisição aumentam superfície de ataque exponencialmente.

Notas Adicionais

- Chave secreta JWT é armazenada como variável de ambiente (JWT_SECRET) e gerenciada via AWS Secrets Manager em produção
- Implementação inclui rotação automática de refresh tokens (refresh token rotation) para maior segurança
- Cache Redis é utilizado para blacklist de tokens revogados, permitindo logout efetivo antes da expiração natural
- Migração futura para algoritmos assimétricos (RS256) está sendo considerada para cenários de microsserviços mais complexos

---

ADR-004: Banco de Dados PostgreSQL e RDS

Data: 23 de janeiro de 2025
Status: Aceito
Decisores: Equipe de Arquitetura e DBA
Categoria: Database

Contexto

O sistema de gerenciamento de oficina requer persistência de dados estruturados: usuários, estoque de peças, ordens de serviço, histórico de manutenções e relacionamentos complexos entre entidades. A escolha do banco de dados impacta integridade dos dados, performance de consultas, escalabilidade, custos operacionais e complexidade de manutenção.

Os requisitos principais incluem suporte a transações ACID para garantir consistência em operações críticas (baixa de estoque, faturamento), capacidade de modelar relacionamentos complexos, suporte a consultas analíticas para relatórios gerenciais e facilidade de integração com Spring Boot através de JPA/Hibernate.

Decisão

Adotamos **PostgreSQL 15+ como banco de dados relacional**, hospedado em **AWS RDS (Relational Database Service)** com as seguintes configurações:

Aspecto	Configuração
Versão	PostgreSQL 15.x
Hospedagem	AWS RDS
Classe de Instância (Dev)	db.t3.medium
Classe de Instância (Prod)	db.r6g.xlarge
Storage	SSD (gp3) com autoscaling
Backup	Automático (7 dias) + snapshots manuais
Disponibilidade	Multi-AZ em produção

O PostgreSQL foi escolhido pela robustez, conformidade com padrões SQL, suporte avançado a tipos de dados (JSON, arrays, enums) e extensibilidade. A hospedagem em AWS RDS oferece gerenciamento automatizado de backups, patches de segurança, monitoramento e alta disponibilidade através de Multi-AZ deployments.

A infraestrutura do banco de dados é provisionada e versionada através de Terraform, garantindo reprodutibilidade e controle de mudanças. O controle de versão do schema é gerenciado pelo Flyway, que executa migrações automaticamente durante o deploy da aplicação.

Consequências

Positivas:

- **Integridade e consistência**: PostgreSQL oferece suporte completo a transações ACID, constraints (foreign keys, unique, check) e triggers, garantindo integridade referencial mesmo em cenários de concorrência alta
- **Modelo relacional robusto**: Capacidade de modelar relacionamentos complexos através de foreign keys e joins otimizados permite representar fielmente o domínio de negócio de uma oficina
- **Performance de consultas**: Otimizador de queries sofisticado, com suporte a índices avançados (B-tree, GiST, GIN) e estatísticas detalhadas
- **Integração nativa com Spring**: Spring Data JPA e Hibernate oferecem suporte de primeira classe para PostgreSQL com mapeamento automático
- **Gerenciamento simplificado com RDS**: AWS RDS elimina tarefas operacionais complexas (instalação, patching, backups, replicação, monitoramento)
- **Alta disponibilidade**: Multi-AZ deployment oferece failover automático com RPO de minutos e RTO de segundos, garantindo 99.95% de disponibilidade
- **Escalabilidade vertical**: RDS permite upgrade de instâncias com downtime mínimo, facilitando crescimento conforme demanda aumenta
- **Segurança**: Criptografia em repouso (AES-256) e em trânsito (TLS), integração com AWS IAM, auditoria via CloudWatch Logs

Negativas:

- **Custo operacional**: AWS RDS tem custo significativamente maior que instâncias EC2 autogerenciadas ou bancos de dados em containers

- **Vendor lock-in**: Dependência do RDS dificulta migração para outros provedores cloud ou on-premises
- **Limitações de configuração**: RDS não oferece acesso root ao servidor, limitando customizações avançadas e instalação de extensões não suportadas
- **Escalabilidade horizontal limitada**: PostgreSQL suporta read replicas, mas escalabilidade de escrita é limitada ao scaling vertical
- **Latência de rede**: Comunicação entre pods Kubernetes e RDS ocorre via rede, adicionando latência comparado a bancos locais

Alternativas Consideradas

MySQL/MariaDB no AWS RDS: Banco relacional open-source popular com ecossistema maduro. Rejeitado porque PostgreSQL oferece conformidade superior com padrões SQL, suporte mais robusto a tipos de dados complexos (JSON, arrays), melhor otimizador de queries para consultas analíticas e melhor suporte a transações complexas.

MongoDB (NoSQL) no AWS DocumentDB: Banco orientado a documentos com flexibilidade de schema. Rejeitado porque domínio de oficina é altamente relacional com relacionamentos complexos que se beneficiam de foreign keys e joins. Ausência de transações ACID multi-documento aumentaria riscos de inconsistência.

Amazon Aurora PostgreSQL: Banco compatível com PostgreSQL com performance até 3x superior. Rejeitado devido ao custo substancialmente maior (2-3x o custo do RDS PostgreSQL padrão). Para volume de dados esperado, performance adicional não justifica investimento.

PostgreSQL autogerenciado em Kubernetes: Execução em pods Kubernetes com StatefulSets e volumes persistentes. Rejeitado devido à complexidade operacional significativa: gerenciamento manual de backups, replicação, failover, patches de segurança e monitoramento. Risco de perda de dados por configuração incorreta é alto.

Notas Adicionais

- Versão do PostgreSQL é fixada em 15.x no Terraform para consistência entre ambientes

- Upgrades de versão major são planejados e testados em staging antes de produção
- Flyway gerencia migrações de schema através de scripts SQL versionados no repositório core
- Connection pooling implementado via HikariCP com configurações otimizadas para RDS
- Métricas de performance monitoradas via AWS CloudWatch e New Relic com alertas configurados

---

ADR-005: Infraestrutura Kubernetes no AWS EKS

Data: 23 de janeiro de 2025
Status: Aceito
Decisores: Equipe de DevOps e Arquitetura
Categoria: Infrastructure

Contexto

O Gerenciador de Oficina - Fase 3 requer uma plataforma de orquestração de containers que suporte escalabilidade automática, alta disponibilidade, rolling updates sem downtime e gerenciamento eficiente de recursos computacionais. A aplicação Spring Boot é containerizada com Docker e precisa ser deployada em um ambiente que permita múltiplas instâncias (pods) com balanceamento de carga, health checks e recuperação automática de falhas.

Os requisitos incluem capacidade de escalar horizontalmente conforme demanda, isolamento de ambientes (dev, staging, prod), gerenciamento declarativo de infraestrutura, integração com serviços AWS (RDS, IAM, VPC) e observabilidade através de logs centralizados e métricas.

Decisão

Adotamos **Kubernetes como plataforma de orquestração**, hospedado no **AWS EKS (Elastic Kubernetes Service)** com infraestrutura versionada em Terraform no repositório **gerenciador-oficina-k8s-infra-fase-3**.

A arquitetura Kubernetes inclui os seguintes componentes:

Componente	Configuração
Versão Kubernetes	1.28.x
Node Groups (Dev)	t3.medium, min 2, max 5 nodes
Node Groups (Prod)	m5.large, min 3, max 10 nodes
Application Deployment	3+ replicas com HPA
Load Balancer	AWS Application Load Balancer (ALB)
Ingress Controller	AWS Load Balancer Controller
Service Type	ClusterIP (interno) + Ingress (externo)
Autoscaling	HPA baseado em CPU e memória

A aplicação Spring Boot é deployada como Deployment com múltiplas réplicas, garantindo alta disponibilidade. Services do tipo ClusterIP expõem os pods internamente, enquanto um Ingress Controller gerencia o tráfego externo através de Application Load Balancer.

O Horizontal Pod Autoscaler (HPA) monitora métricas de CPU e memória, escalando automaticamente o número de réplicas conforme a carga. ConfigMaps armazenam configurações não sensíveis, enquanto Secrets gerenciam credenciais e tokens JWT. Health checks são implementados através de liveness e readiness probes, garantindo que apenas pods saudáveis recebam tráfego.

Namespaces separam ambientes (dev, staging, prod), com Resource Quotas e Limit Ranges prevenindo consumo excessivo de recursos. Network Policies restringem comunicação entre pods, seguindo princípios de zero-trust. O cluster integra-se com AWS IAM através de IRSA (IAM Roles for Service Accounts), permitindo que pods acessem serviços AWS sem credenciais hardcoded.

Consequências

Positivas:

- **Escalabilidade automática**: Horizontal Pod Autoscaler permite que aplicação responda automaticamente a picos de demanda, escalando o número de réplicas baseado em métricas de CPU, memória ou custom metrics
- **Alta disponibilidade**: Distribuição de pods em múltiplos nodes e availability zones, combinada com health checks e self-healing, garante que aplicação permaneça disponível mesmo durante falhas de hardware ou atualizações
- **Deploy sem downtime**: Rolling updates permitem atualizar aplicação gradualmente, substituindo pods antigos por novos sem interrupção do serviço. Rollback automático ocorre se novos pods falharem health checks
- **Gerenciamento declarativo**: Toda a infraestrutura Kubernetes é definida em manifests YAML versionados no Git, permitindo revisão de mudanças, auditoria e reprodutibilidade
- **Isolamento e segurança**: Namespaces, Network Policies e RBAC garantem isolamento entre ambientes e componentes. Pods executam com privilégios mínimos através de Security Contexts
- **Integração nativa com AWS**: EKS é totalmente gerenciado pela AWS, eliminando overhead de gerenciar o control plane do Kubernetes
- **Portabilidade**: Embora hospedado no EKS, aplicação usa APIs padrão do Kubernetes, facilitando migração futura para outros provedores (GKE, AKS) ou on-premises
- **Ecossistema rico**: Kubernetes tem ecossistema vasto de ferramentas para observabilidade (Prometheus, Grafana), service mesh (Istio), CI/CD (ArgoCD)

Negativas:

- **Complexidade operacional**: Kubernetes tem curva de aprendizado íngreme com muitos conceitos (pods, services, ingress, persistent volumes, operators) que requerem expertise especializada
- **Overhead de recursos**: Kubernetes consome recursos computacionais para seus componentes (kubelet, kube-proxy, CoreDNS), reduzindo capacidade disponível para aplicações
- **Custo do EKS**: AWS EKS cobra $0.10/horaporcluster$(73/mês) além dos custos de nodes EC2, Load Balancers e outros recursos
- **Latência de startup**: Pods Kubernetes têm latência de startup maior que containers standalone devido a scheduling, pull de imagens e health checks

- **Debugging complexo**: Diagnosticar problemas em ambientes Kubernetes requer ferramentas especializadas (kubectl, k9s, Lens) e compreensão de logs distribuídos

Alternativas Consideradas

AWS ECS (Elastic Container Service): Serviço de orquestração proprietário da AWS, mais simples que Kubernetes com integração nativa com serviços AWS. Rejeitado devido ao vendor lock-in severo, tornando migração futura para outros provedores extremamente custosa. Kubernetes oferece portabilidade e ecossistema significativamente mais rico.

Docker Swarm: Solução nativa de orquestração do Docker, mais simples que Kubernetes. Rejeitado porque está praticamente descontinuado pela comunidade com desenvolvimento estagnado e adoção decrescente. Ecossistema de ferramentas é limitado.

Nomad (HashiCorp): Orquestrador flexível que suporta containers, VMs e binários standalone. Rejeitado devido à adoção limitada comparada ao Kubernetes, menor disponibilidade de profissionais no mercado e ecossistema de ferramentas menos maduro.

AWS Fargate (serverless containers): Permite executar containers sem gerenciar nodes EC2. Rejeitado devido ao custo significativamente maior para workloads contínuos (até 40% mais caro que EC2), latência de cold start mais alta (até 60 segundos) e limitações de customização de rede e storage.

Notas Adicionais

- Cluster EKS é configurado com versão 1.28 do Kubernetes, com upgrades planejados seguindo ciclo de releases do EKS
- Node groups utilizam Amazon Linux 2 com EKS-optimized AMI
- AWS Load Balancer Controller é instalado via Helm para gerenciar Application Load Balancers automaticamente
- Metrics Server é instalado para suportar Horizontal Pod Autoscaler
- Logs de pods são enviados para CloudWatch Logs através do Fluent Bit DaemonSet

- Cluster Autoscaler gerencia scaling de nodes EC2 baseado em pods pending
- IRSA (IAM Roles for Service Accounts) é configurado para permitir que pods acessem RDS e Secrets Manager sem credenciais hardcoded
- Network Policies são implementadas usando AWS VPC CNI plugin
- Estratégia de deploy utiliza rolling updates com maxSurge=1 e maxUnavailable=0 para garantir zero downtime

---

ADR-006: Funções Lambda para Processamento Assíncrono

Data: 23 de janeiro de 2025
Status: Aceito
Decisores: Equipe de Arquitetura e Backend
Categoria: Backend

Contexto

O sistema de gerenciamento de oficina possui operações que não precisam ser executadas em tempo real e podem impactar a latência da API principal se processadas sincronicamente. Exemplos incluem envio de notificações por e-mail/SMS, geração de relatórios complexos, processamento de imagens de veículos e integração com sistemas de terceiros. A execução síncrona dessas tarefas aumentaria o tempo de resposta das APIs, degradaria a experiência do usuário e acoplaria a aplicação principal a serviços externos que podem ser lentos ou indisponíveis.

Precisávamos de uma solução para processamento assíncrono que fosse escalável, econômica, desacoplada da aplicação principal e gerenciada com o mínimo de overhead operacional.

Decisão

Adotamos **AWS Lambda para processamento assíncrono e event-driven**, com o código-fonte das funções gerenciado no repositório **gerenciador-oficina-lambda-fase-3**.

A arquitetura utiliza um padrão de comunicação assíncrona baseado em eventos. A aplicação principal publica eventos em um tópico AWS SNS (Simple Notification Service) sempre que uma operação assíncrona é necessária. Múltiplas filas AWS SQS (Simple Queue Service) são inscritas nesse tópico, cada uma correspondendo a um tipo de tarefa (ex: `email-notifications-queue`, `report-generation-queue`). Cada fila SQS serve como trigger para uma função Lambda específica, que consome a mensagem e executa a lógica de negócio correspondente.

As funções Lambda são escritas em Java 17 com o framework Quarkus para otimizar o tempo de startup (cold start) e o consumo de memória. O código é empacotado como um JAR nativo usando GraalVM, resultando em performance superior e custos reduzidos. As funções são provisionadas e configuradas via Terraform, incluindo permissões IAM, triggers e variáveis de ambiente.

Componente	Configuração
Runtime	Java 17 (Quarkus + GraalVM Native)
Timeout	900 segundos (15 minutos)
Memory	512-1024 MB (otimizado por função)
Triggers	AWS SQS (filas dedicadas)
Dead Letter Queue	Habilitada para mensagens com falha
Tracing	AWS X-Ray habilitado

Consequências

Positivas:

- **Desacoplamento e resiliência**: Comunicação via SNS/SQS desacopla aplicação principal das funções assíncronas. Se uma função Lambda falhar, a mensagem permanece na fila SQS e pode ser reprocessada através de uma Dead Letter Queue (DLQ), evitando perda de dados
- **Melhora na performance da API**: Delegação de tarefas demoradas para processamento assíncrono reduz drasticamente o tempo de resposta das APIs REST. Cliente recebe confirmação imediata (202 Accepted) enquanto tarefa é executada em background

- **Escalabilidade automática e econômica**: AWS Lambda escala automaticamente de zero a milhares de execuções concorrentes sem necessidade de provisionamento manual. Modelo de cobrança pay-per-use (pagamento por milissegundo de execução) é extremamente econômico para workloads esporádicos
- **Gerenciamento zero de servidores**: Lambda é um serviço serverless, eliminando completamente a necessidade de gerenciar servidores, sistemas operacionais, patches de segurança ou escalabilidade
- **Performance otimizada com Java nativo**: Uso de Quarkus e GraalVM para compilação nativa resulta em tempos de startup de milissegundos e consumo de memória até 10x menor que Java tradicional na JVM
- **Segurança granular**: Cada função Lambda possui sua própria IAM Role com permissões mínimas (princípio de least privilege), limitando o acesso a recursos AWS específicos

Negativas:

- **Complexidade de debugging e monitoramento**: Debugging de sistemas distribuídos e assíncronos é inerentemente mais complexo. Rastrear uma transação através de SNS, SQS e Lambda requer ferramentas de observabilidade como AWS X-Ray ou New Relic
- **Latência de cold start**: Embora otimizado com Java nativo, o primeiro request para uma função inativa ainda incorre em latência de cold start (provisionamento de ambiente). Para operações sensíveis à latência, pode ser necessário usar provisioned concurrency
- **Limitações de execução**: Funções Lambda têm limites de tempo de execução (máximo 15 minutos), tamanho do pacote de deploy (250 MB) e concorrência. Para tarefas de longa duração, pode ser necessário usar AWS Step Functions ou AWS Batch
- **Complexidade de estado**: Lambda é stateless, dificultando o gerenciamento de estado entre execuções. Para workflows complexos com múltiplos passos, é necessário usar um serviço de orquestração como AWS Step Functions

Alternativas Consideradas

Threads assíncronos na aplicação principal (@Async): Utilização da anotação `@Async` do Spring Boot para executar tarefas em threads separados dentro da mesma aplicação. Rejeitado porque acopla o processamento assíncrono ao ciclo de vida da aplicação principal. Se a aplicação reiniciar, as tarefas em andamento são perdidas. Scaling das tarefas assíncronas fica atrelado ao scaling da aplicação inteira, o que é ineficiente.

Fila de mensagens com workers em Kubernetes (RabbitMQ/Kafka): Implementação de um message broker como RabbitMQ ou Kafka no cluster Kubernetes, com um grupo de pods "workers" dedicados a consumir mensagens. Rejeitado devido à alta complexidade operacional. Equipe precisaria instalar, configurar, monitorar e gerenciar o message broker e os pods workers. Custo seria maior, pois os pods workers ficariam ociosos na maior parte do tempo.

AWS Batch: Serviço para execução de workloads de processamento em lote em larga escala. Rejeitado porque é otimizado para processamento em lote de grande volume, não para tarefas event-driven de baixa latência. Complexidade de configuração é maior e modelo de cobrança é baseado em instâncias EC2.

AWS Step Functions: Serviço de orquestração de workflows serverless que coordena múltiplas funções Lambda. Rejeitado porque para as tarefas assíncronas atuais (notificações, relatórios simples), seria over-engineering. Combinação SNS -> SQS -> Lambda é suficiente e mais simples.

Notas Adicionais

- Escolha do Quarkus com GraalVM foi fundamental para viabilizar uso de Java em Lambda de forma competitiva
- Funções Lambda são configuradas com Dead Letter Queues (DLQ) para capturar mensagens que falham após múltiplas tentativas
- Tracing distribuído é habilitado com AWS X-Ray para monitorar fluxo de eventos desde publicação no SNS até execução final no Lambda
- Repositório `gerenciador-oficina-lambda-fase-3` contém pipelines CI/CD separados para build e deploy das funções

---

ADR-007: API Gateway com Rate Limiting

Data: 23 de janeiro de 2025
Status: Aceito
Decisores: Equipe de Infraestrutura e Segurança
Categoria: Infrastructure

Contexto

A aplicação, exposta via Kubernetes Ingress, precisa de uma camada de gerenciamento centralizada para lidar com requisitos de segurança, roteamento, monitoramento e controle de tráfego. Expor o Ingress Controller (AWS Load Balancer) diretamente à internet cria desafios, como a implementação de rate limiting para prevenir abuso, autenticação de APIs, roteamento avançado baseado em path ou host, e monitoramento centralizado de requisições.

A ausência de um API Gateway forçaria a implementação dessas funcionalidades diretamente na aplicação ou no Ingress Controller, o que aumentaria a complexidade do código, dificultaria a manutenção e limitaria a flexibilidade para futuras integrações.

Decisão

Adotamos o **AWS API Gateway** como a camada de front-end para todas as APIs do sistema, com a infraestrutura provisionada e versionada via Terraform no repositório **gerenciador-oficina-api-gateway-infra-fase-3**.

A configuração utiliza um API Gateway do tipo REST com integração `HTTP_PROXY` para o Application Load Balancer (ALB) do cluster EKS. Isso permite que o API Gateway atue como um proxy reverso, encaminhando o tráfego para a aplicação Kubernetes sem expor o ALB diretamente.

As principais funcionalidades configuradas são:

Funcionalidade	Configuração
Rate Limiting	100 req/s com burst de 200
Throttling	Por API Key e por cliente
Roteamento	Paths (/v1/auth, /v1/inventory, etc)
Logging	CloudWatch Logs com retenção de 30 dias
Monitoramento	Métricas de latência, erros 4xx/5xx, volume
Segurança	API Keys, Lambda Authorizers (futuro)

Consequências

Positivas:

- **Segurança aprimorada**: Rate limiting e throttling protegem a aplicação backend contra picos de tráfego maliciosos ou inesperados, garantindo disponibilidade para usuários legítimos. Centralização do ponto de entrada também facilita a aplicação de outras políticas de segurança, como WAF (Web Application Firewall)
- **Gerenciamento centralizado**: Todas as políticas de API (roteamento, segurança, monitoramento) são gerenciadas em um único local, desacoplando essas preocupações da lógica de negócio da aplicação
- **Observabilidade nativa**: Integração com CloudWatch fornece visibilidade completa sobre o tráfego da API, permitindo identificar gargalos de performance, picos de erro e padrões de uso
- **Flexibilidade de roteamento**: API Gateway permite criar novas versões de API (ex: `/v2`), implementar canary releases e rotear tráfego para diferentes backends de forma transparente para o cliente
- **Ecossistema de integrações**: API Gateway se integra nativamente com todo o ecossistema AWS, incluindo Lambda, S3, Step Functions e IAM

Negativas:

- **Custo adicional**: AWS API Gateway tem um custo por milhão de requisições, além de custos de transferência de dados. Para APIs de alto volume, esse custo pode ser significativo

- **Latência adicional**: API Gateway introduz um hop de rede adicional no caminho da requisição, adicionando alguns milissegundos de latência
- **Complexidade de configuração**: Configuração do API Gateway, especialmente com Terraform, pode ser complexa e verbosa, com muitos recursos interconectados
- **Vendor lock-in**: Dependência do AWS API Gateway dificulta a migração para outros provedores de nuvem ou para uma solução on-premises

Alternativas Consideradas

Exposição direta do Kubernetes Ingress: Expor o Application Load Balancer criado pelo Ingress Controller diretamente à internet. Rejeitado porque carece de funcionalidades críticas de gerenciamento de API, como rate limiting, autenticação centralizada e monitoramento granular. Implementar essas funcionalidades no nível do Ingress ou da aplicação seria mais complexo e menos robusto.

API Gateway open-source em Kubernetes (Kong, Tyk, Gloo): Instalar um API Gateway open-source como Kong, Tyk ou Gloo diretamente no cluster Kubernetes. Rejeitado devido à maior complexidade operacional. Equipe seria responsável por instalar, configurar, escalar e manter o API Gateway, incluindo seu banco de dados de configuração (geralmente Cassandra ou PostgreSQL).

Service Mesh com Ingress Gateway (Istio): Implementar um service mesh como o Istio, que inclui um Ingress Gateway para gerenciar o tráfego de entrada. Rejeitado porque seria over-engineering para o estágio atual do projeto. Complexidade de instalar e gerenciar um service mesh completo é muito alta.

Notas Adicionais

- Configuração do Terraform para o API Gateway é modularizada para permitir reutilização e facilitar adição de novos endpoints
- API Keys são gerenciadas via AWS Secrets Manager e distribuídas de forma segura para os clientes da API
- Logs do API Gateway são enviados para CloudWatch Logs e retidos por 30 dias, com métricas chave sendo encaminhadas para o New Relic

---

ADR-008: Infrastructure as Code com Terraform

Data: 23 de janeiro de 2025
Status: Aceito
Decisores: Equipe de Infraestrutura e DevOps
Categoria: DevOps

Contexto

O projeto Gerenciador de Oficina possui uma infraestrutura de nuvem complexa na AWS, abrangendo múltiplos serviços: EKS, RDS, VPC, IAM, API Gateway e Lambda. O gerenciamento manual dessa infraestrutura através do Console AWS é propenso a erros, lento, difícil de auditar e impossível de replicar de forma consistente entre diferentes ambientes (desenvolvimento, staging, produção).

A falta de automação no provisionamento de infraestrutura leva a configurações divergentes (configuration drift), dificulta a recuperação de desastres e torna o processo de criação de novos ambientes um gargalo significativo.

Decisão

Adotamos **Terraform como a ferramenta de Infrastructure as Code (IaC)** para provisionar e gerenciar toda a infraestrutura AWS. O código Terraform é organizado em repositórios dedicados, alinhados com a arquitetura multi-repositório:

Repositório	Responsabilidade
gerenciador-oficina-k8s-infra-fase-3	Cluster EKS, node groups, VPC, subnets, configurações de rede
gerenciador-oficina-db-infra-fase-3	Instância RDS PostgreSQL, security groups, parâmetros de banco
gerenciador-oficina-api-gateway-infra-fase-3	API Gateway, rotas, usage plans, integrações

O estado do Terraform (`terraform.tfstate`) é armazenado remotamente em um bucket S3 com versionamento e bloqueio de estado (state locking) via DynamoDB para

prevenir corrupção de estado em execuções concorrentes. O código é estruturado em módulos reutilizáveis para promover a padronização e reduzir a duplicação.

Consequências

Positivas:

- **Automação e reprodutibilidade**: Infraestrutura pode ser provisionada e atualizada de forma totalmente automatizada, garantindo que todos os ambientes sejam idênticos e consistentes. Criação de um novo ambiente de ponta a ponta torna-se uma tarefa rápida e confiável
- **Versionamento e auditoria**: Todo o código de infraestrutura é versionado no Git, permitindo rastrear cada mudança, revisar alterações através de pull requests e reverter para versões anteriores se necessário
- **Prevenção de configuration drift**: Ao usar o Terraform como a única fonte da verdade, evitamos que configurações manuais não rastreadas causem divergências entre o estado desejado e o estado real da infraestrutura
- **Recuperação de desastres**: Em caso de falha catastrófica de uma região AWS, a infraestrutura pode ser recriada rapidamente em outra região executando o mesmo código Terraform
- **Planejamento de mudanças (terraform plan)**: Comando `terraform plan` permite visualizar o impacto de uma mudança antes de aplicá-la, mostrando quais recursos serão criados, modificados ou destruídos
- **Agnóstico de nuvem**: Embora estejamos usando o provedor AWS, o Terraform suporta múltiplos provedores de nuvem (Azure, GCP) e on-premises. Sintaxe e workflow são os mesmos
- **Ecossistema e comunidade**: Terraform é o padrão de mercado para IaC, com comunidade vasta, documentação extensa e um grande número de módulos pré-construídos no Terraform Registry

Negativas:

- **Curva de aprendizado**: Linguagem do Terraform (HCL - HashiCorp Configuration Language) e seus conceitos (estado, provedores, módulos) requerem um tempo de aprendizado para a equipe

- **Gerenciamento de estado**: Arquivo de estado é um componente crítico e sensível. Se for corrompido ou perdido, o Terraform perde a referência da infraestrutura gerenciada, exigindo uma importação manual complexa e arriscada
- **Refatoração complexa**: Mover ou renomear recursos no código Terraform pode ser complexo e arriscado, muitas vezes exigindo manipulação manual do arquivo de estado (`terraform state mv`)
- **Lentidão em infraestruturas grandes**: Em projetos com milhares de recursos, execução do `terraform plan` e `terraform apply` pode se tornar lenta

Alternativas Consideradas

AWS CloudFormation: Solução de IaC nativa da AWS que utiliza templates em formato JSON ou YAML. Rejeitado pela sintaxe verbosa e menos legível (especialmente em JSON), dificuldade de criar módulos reutilizáveis e forte dependência da AWS (vendor lock-in). Terraform oferece uma experiência de desenvolvimento superior e portabilidade.

Pulumi: Ferramenta de IaC que permite definir infraestrutura usando linguagens de programação de propósito geral (TypeScript, Python, Go). Rejeitado porque a abordagem declarativa do Terraform é geralmente considerada mais segura e previsível para infraestrutura. Uso de linguagens imperativas pode levar a resultados inesperados.

Scripts de CLI (AWS CLI, Bash): Utilizar scripts customizados que invocam a AWS CLI para provisionar recursos. Rejeitado porque rapidamente se torna insustentável. Scripts se tornam complexos, difíceis de manter e não têm gerenciamento de estado, o que significa que não saberiam como atualizar ou destruir recursos existentes.

Notas Adicionais

- Pipeline de CI/CD no GitHub Actions inclui etapas para `terraform fmt` (formatação), `terraform validate` (validação de sintaxe) e `terraform plan` em cada pull request
- `terraform apply` é executado manualmente ou automaticamente após o merge para a branch principal, dependendo do ambiente

- Variáveis sensíveis, como chaves de API, são injetadas no Terraform através de variáveis de ambiente no pipeline, e não são armazenadas no código-fonte

---

ADR-009: Pipeline CI/CD com GitHub Actions

Data: 23 de janeiro de 2025
Status: Aceito
Decisores: Equipe de DevOps e Desenvolvimento
Categoria: DevOps

Contexto

Com uma arquitetura multi-repositório e a necessidade de entregas rápidas e seguras, um processo manual de build, teste e deploy é inviável. A falta de um pipeline de Integração Contínua (CI) e Entrega Contínua (CD) levaria a builds inconsistentes, falhas de teste não detectadas, deploys manuais arriscados e um ciclo de feedback lento para os desenvolvedores.

Precisávamos de uma solução de CI/CD que se integrasse nativamente com o GitHub (onde o código-fonte está hospedado), que fosse flexível para suportar os diferentes tipos de repositórios (aplicação Java, infraestrutura Terraform, funções Lambda) e que permitisse a automação de todo o ciclo de vida do desenvolvimento.

Decisão

Adotamos **GitHub Actions como a plataforma de CI/CD** para todos os repositórios do projeto. Cada repositório possui seu próprio workflow (`.github/workflows/main.yml`) customizado para suas necessidades específicas.

Repositório Core (Java/Spring Boot): O pipeline é disparado a cada push na branch `main` ou em pull requests. As etapas incluem: checkout do código, setup do JDK 17, cache de dependências do Maven, compilação, execução de testes unitários e de integração com Mockito, análise de qualidade de código com SonarQube, build da imagem Docker e push para o AWS ECR (Elastic Container Registry). O deploy no EKS é acionado manualmente (em produção) ou automaticamente (em staging) após a imagem ser publicada.

Repositórios de Infraestrutura (Terraform): O pipeline é disparado em pull requests. As etapas incluem: checkout do código, setup do Terraform, `terraform init`, `terraform validate` e `terraform plan`. O resultado do `plan` é postado como um comentário no pull request para revisão. O `terraform apply` é executado manualmente após o merge.

Repositório Lambda (Java/Quarkus): O pipeline é semelhante ao do core, mas inclui a etapa de compilação nativa com GraalVM, empacotamento do ZIP da função e deploy via AWS CLI (`aws lambda update-function-code`).

Consequências

Positivas:

- **Integração nativa com GitHub**: GitHub Actions é perfeitamente integrado ao ecossistema GitHub, permitindo acionar workflows a partir de eventos do Git (push, pull request), gerenciar segredos de forma segura e visualizar o status do pipeline diretamente na interface do GitHub
- **Automação e padronização**: Todo o processo de build, teste e deploy é automatizado, garantindo que as mesmas etapas sejam executadas de forma consistente a cada mudança
- **Feedback rápido**: Desenvolvedores recebem feedback imediato sobre a qualidade e o sucesso dos testes de seu código a cada pull request, permitindo corrigir problemas mais cedo no ciclo de desenvolvimento
- **Flexibilidade e customização**: Workflows em YAML são altamente customizáveis, permitindo a criação de pipelines complexos com matrizes de build (para testar em diferentes versões), etapas condicionais e integração com milhares de ações disponíveis no GitHub Marketplace
- **Custo-benefício**: GitHub Actions oferece um generoso plano gratuito para repositórios públicos e privados, e os custos para minutos de execução adicionais são competitivos

Negativas:

- **Vendor lock-in (parcial)**: Embora os scripts de build e deploy possam ser reutilizados, a sintaxe do workflow YAML é específica do GitHub Actions, dificultando a migração para outras plataformas de CI/CD como GitLab CI ou Jenkins

- **Gerenciamento de runners auto-hospedados (Self-hosted runners)**: Para tarefas que exigem acesso a redes privadas ou hardware específico, pode ser necessário configurar e gerenciar runners auto-hospedados, o que adiciona complexidade operacional
- **Debugging de workflows**: Diagnosticar falhas em workflows complexos pode ser desafiador, exigindo análise de logs extensos e, por vezes, execução de ferramentas de debugging via SSH

Alternativas Consideradas

Jenkins: Ferramenta de CI/CD open-source mais tradicional e poderosa, com um ecossistema de plugins gigantesco. Rejeitado devido à alta complexidade operacional. Jenkins requer o gerenciamento de um servidor dedicado, plugins, segurança e backups, o que consumiria um tempo significativo da equipe de DevOps.

GitLab CI/CD: Solução de CI/CD integrada e poderosa, semelhante ao GitHub Actions. Rejeitado porque o código do projeto já está hospedado no GitHub, e a migração para o GitLab seria um esforço desnecessário. GitHub Actions oferece funcionalidades equivalentes com a vantagem da integração nativa.

AWS CodePipeline / CodeBuild: Conjunto de ferramentas de CI/CD da AWS (CodeCommit, CodeBuild, CodeDeploy, CodePipeline). Rejeitado por ser menos amigável ao desenvolvedor e mais complexo de configurar que o GitHub Actions. A definição dos pipelines é mais verbosa e a interface do usuário é menos intuitiva.

Notas Adicionais

- Segredos, como `AWS_ACCESS_KEY_ID`, `SONAR_TOKEN` e `DOCKER_PASSWORD`, são armazenados como segredos criptografados no GitHub e injetados no workflow como variáveis de ambiente
- Cache de dependências (Maven, node_modules) é utilizado para acelerar a execução dos pipelines
- Análise do SonarQube é configurada para falhar o pipeline se as metas de qualidade (Quality Gate) não forem atingidas

---

ADR-010: Observabilidade com New Relic

Data: 23 de janeiro de 2025
Status: Aceito
Decisores: Equipe de SRE e Arquitetura
Categoria: DevOps

Contexto

Em uma arquitetura distribuída com Kubernetes, Lambdas e múltiplos serviços AWS, monitorar a saúde e a performance do sistema é um desafio complexo. Logs e métricas isolados de cada componente (aplicação, banco de dados, infraestrutura) não fornecem uma visão unificada do comportamento do sistema. Para diagnosticar problemas rapidamente, otimizar a performance e entender a experiência do usuário, é essencial ter uma plataforma de observabilidade que correlacione métricas, logs e traces (rastreamento distribuído).

Precisávamos de uma solução que oferecesse APM (Application Performance Monitoring) para a aplicação Spring Boot, monitoramento de infraestrutura para Kubernetes e RDS, e tracing distribuído para rastrear requisições através dos diferentes serviços.

Decisão

Adotamos a plataforma **New Relic** como a solução de observabilidade completa para o projeto.

A implementação inclui:

Componente	Configuração
New Relic APM Agent	Agente Java integrado à aplicação Spring Boot
Infrastructure Integration	Agente de infraestrutura como DaemonSet no EKS
AWS Integration	CloudWatch Metric Streams para serviços AWS
Log Management	Logs centralizados com análise e correlação
Dashboards	Customizados para visualizar KPIs do sistema
Alertas	Baseados em anomalias e violações de SLOs

New Relic APM Agent: O agente Java é integrado à aplicação Spring Boot para coletar automaticamente métricas detalhadas da JVM, transações web, queries de banco de dados e chamadas para serviços externos. O tracing distribuído é habilitado para rastrear requisições entre a aplicação e as funções Lambda.

New Relic Infrastructure Integration for Kubernetes: O agente de infraestrutura é deployado como um DaemonSet no cluster EKS para coletar métricas de nós, pods, deployments e outros recursos Kubernetes.

New Relic AWS Integration: Integração com a AWS via CloudWatch Metric Streams permite coletar métricas de serviços como RDS, Lambda, API Gateway e ELB.

Log Management: Logs da aplicação e da infraestrutura são encaminhados para o New Relic para análise e correlação com traces e métricas.

Dashboards e Alertas: Dashboards customizados são criados para visualizar os principais indicadores de performance (KPIs) do sistema. Alertas são configurados para notificar a equipe sobre anomalias, erros e violações de SLOs (Service Level Objectives).

Consequências

Positivas:

- **Visão unificada**: New Relic fornece uma plataforma única para visualizar e correlacionar todos os dados de telemetria (métricas, logs, traces), permitindo uma análise holística da saúde do sistema

- **Diagnóstico rápido de problemas**: Tracing distribuído permite identificar rapidamente a causa raiz de latência ou erros, mostrando o tempo gasto em cada componente de uma requisição. APM aponta gargalos de performance diretamente no código
- **Monitoramento proativo**: Alertas baseados em anomalias (AI-driven) permitem detectar problemas antes que eles impactem os usuários. Equipe pode ser notificada sobre degradação de performance ou aumento de taxas de erro
- **Otimização de performance**: Dados coletados pelo APM permitem que desenvolvedores identifiquem e otimizem queries lentas, métodos de código ineficientes e problemas de consumo de memória
- **Visibilidade da experiência do usuário**: New Relic permite monitorar a performance do ponto de vista do cliente, medindo o tempo de resposta real das APIs e a taxa de sucesso das transações

Negativas:

- **Custo**: New Relic é uma solução comercial com um modelo de precificação baseado no volume de dados ingeridos e no número de usuários. Para sistemas de grande escala, o custo pode ser significativo
- **Overhead do agente**: Agente APM adiciona um pequeno overhead de CPU e memória à aplicação. Embora geralmente seja mínimo, em sistemas de alta performance, isso precisa ser medido e considerado
- **Vendor lock-in**: Dependência de uma plataforma de observabilidade proprietária pode dificultar a migração para outras soluções no futuro. Instrumentação e dashboards são específicos do New Relic

Alternativas Consideradas

Stack open-source (Prometheus, Grafana, Jaeger): Implementar uma solução de observabilidade baseada em ferramentas open-source: Prometheus para métricas, Grafana para dashboards e Jaeger para tracing distribuído. Rejeitado devido à alta complexidade operacional. Equipe precisaria instalar, configurar, escalar e manter cada um desses componentes, além de gerenciar o armazenamento de dados a longo prazo. Correlação entre métricas, logs e traces não é tão fluida quanto em uma plataforma unificada.

AWS CloudWatch: Utilizar exclusivamente os serviços de monitoramento da AWS: CloudWatch Logs, CloudWatch Metrics e AWS X-Ray para tracing. Rejeitado porque CloudWatch foi considerado insuficiente como uma solução de APM completa. Interface do usuário é menos amigável, correlação entre logs, métricas e traces é mais manual, e funcionalidades de análise de performance de código não são tão avançadas.

Datadog: Concorrente direto do New Relic, oferecendo uma plataforma de observabilidade unificada com funcionalidades semelhantes. A escolha entre New Relic e Datadog foi baseada em uma avaliação de funcionalidades, usabilidade e preço. Ambas são soluções excelentes. New Relic foi ligeiramente preferido pela sua força histórica em APM Java e pela estrutura de preços que se adequava melhor ao caso de uso.

Notas Adicionais

- Configuração do New Relic é gerenciada via Terraform sempre que possível
- Agente Java é incluído na imagem Docker da aplicação
- Alertas críticos são enviados para um canal dedicado no Slack para garantir uma resposta rápida da equipe de plantão (on-call)

Matriz de Decisões

A tabela abaixo apresenta um resumo das decisões arquiteturais, facilitando a comparação e referência rápida:

ADR	Título	Status	Categoria	Decisão Principal	Alternativa Rejeitada
001	Arquitetura Multi-Repositório	Aceito	Arquitetura	5 repositórios independentes	Monorepo
002	Stack Backend Java/Spring Boot	Aceito	Backend	Java 17 + Spring Boot 3.3	Node.js, Python, Go
003	Autenticação JWT	Aceito	Segurança	JWT com tokens de curta duração	Sessões, OAuth 2.0
004	PostgreSQL e RDS	Aceito	Database	PostgreSQL 15+ no AWS RDS	MySQL, MongoDB, Aurora
005	Kubernetes no AWS EKS	Aceito	Infrastructure	EKS com node groups gerenciados	ECS, Docker Swarm, Nomad
006	Lambda para Processamento Assíncrono	Aceito	Backend	AWS Lambda com SNS/SQS	@Async, RabbitMQ/Kafka
007	API Gateway com Rate Limiting	Aceito	Infrastructure	AWS API Gateway REST	Ingress direto, Kong, Istio
008	Infrastructure as Code com Terraform	Aceito	DevOps	Terraform com estado remoto	CloudFormation, Pulumi
009	Pipeline CI/CD com GitHub Actions	Aceito	DevOps	GitHub Actions workflows	Jenkins, GitLab CI, AWS CodePipeline
010	Observabilidade com New Relic	Aceito	DevOps	New Relic APM + Infraestrutura	Prometheus/Grafana, CloudWatch, Datadog

---

Conclusão

As decisões arquiteturais documentadas neste ADR refletem uma abordagem cuidadosa e bem fundamentada para construir um sistema de gerenciamento de oficina robusto, escalável e mantível. A arquitetura multi-repositório permite autonomia de equipes e deploy independente. O stack Java/Spring Boot oferece ecossistema maduro e segurança enterprise-grade. A infraestrutura cloud-native no AWS EKS com Kubernetes garante alta disponibilidade e escalabilidade automática.

A combinação de JWT para autenticação, PostgreSQL para persistência, Lambda para processamento assíncrono, API Gateway para gerenciamento centralizado, Terraform para IaC, GitHub Actions para CI/CD e New Relic para observabilidade cria uma arquitetura moderna, bem integrada e pronta para produção.

Essas decisões foram tomadas considerando cuidadosamente as alternativas disponíveis, os trade-offs envolvidos e o alinhamento com as melhores práticas da indústria para sistemas distribuídos. A documentação deste ADR serve como referência para futuras decisões arquiteturais e para facilitar o onboarding de novos membros da equipe.

Documento preparado por: Manus AI
Data de Conclusão: 23 de janeiro de 2025
Versão: 1.0
Status: Publicado