INSTITUTO FEDERAL
Rondônia

MINISTÉRIO DA EDUCAÇÃO
Secretaria de Educação Profissional e Tecnológica
Instituto Federal de Educação, Ciência e Tecnologia de Rondônia

Campus Vilhena

Coordenação do Curso Superior em Tecnologia em Análise e Desenvolvimento de Sistemas

RUAN DE OLIVEIRA LOPES DA SILVA

Desenvolvimento de um serviço para envio de e-mails transacionais para serviços terceiros

VILHENA - RO

2026

RUAN DE OLIVEIRA LOPES DA SILVA

Desenvolvimento de um serviço para envio de e-mails transacionais para serviços terceiros

Artigo entregue como Trabalho de Conclusão de Curso apresentado ao Instituto Federal de Educação, Ciência e Tecnologia de Rondônia (IFRO), Campus Vilhena, como requisito parcial para obtenção do grau de Tecnólogo, junto ao Curso Superior em Tecnologia em Análise e Desenvolvimento de Sistemas, sob a orientação do professor Marco Antônio Augusto de Andrade.

VILHENA - RO

2026

Ficha catalográfica elaborada pelo Sistema Gerador de Ficha Catalográfica do IFRO.

Silva, Ruan de Oliveira Lopes da.
Desenvolvimento de um serviço para envio de e-mails transacionais para serviços terceiros / Ruan de Oliveira Lopes da Silva. - Vilhena, 2026.
30 f. : il.

Orientador(a): Prof. Me. Marco Antonio Augusto de Andrade.

Trabalho de Conclusão de Curso (Superior de Tecnologia em Análise e Desenvolvimento de Sistemas) – Instituto Federal de Educação, Ciência e Tecnologia de Rondônia - IFRO, Vilhena, 2026.

1. Arquitetura de software. 2. Gateway de e-mails. 3. Mensageria assíncrona. 4. Reds / BullMQ. 5. MJML. I. Andrade, Marco Antonio Augusto de (orient.). II. Instituto Federal de Educação, Ciência e Tecnologia de Rondônia - IFRO. III. Título.

Bibliotecário(a) Responsável: Rosilene Maria do Couto Marques, CRB-11/321

RUAN DE OLIVEIRA LOPES DA SILVA

Desenvolvimento de um serviço para envio de e-mails transacionais para serviços terceiros

Artigo entregue como Trabalho de Conclusão de Curso apresentado ao Instituto Federal de Educação, Ciência e Tecnologia de Rondônia (IFRO), Campus Vilhena, como requisito parcial para obtenção do grau de Tecnólogo, junto ao Curso Superior em Tecnologia em Análise e Desenvolvimento de Sistemas, sob a orientação do professor Marco Antônio Augusto de Andrade.

Aprovado em 06/11/2026 pela banca examinadora.

gov.br
Documento assinado digitalmente
ERICK LEONARDO WEIL
Data: 19/11/2026 09:28:14-0300
Verifique em https://validar.iti.gov.br

Prof. Especialista Erick Leonardo Weil

Examinador interno

gov.br
Documento assinado digitalmente
GILBERTO PEREIRA DA SILVA
Data: 18/11/2026 17:36:39-0300
Verifique em https://validar.iti.gov.br

Prof. Mestre Gilberto Pereira da Silva

Examinador interno

gov.br
Documento assinado digitalmente
MARCO ANTONIO AUGUSTO DE ANDRADE
Data: 18/11/2026 17:38:38-0300
Verifique em https://validar.iti.gov.br

Prof. Mestre Marco Antonio Augusto de Andrade

Orientador

Resumo

O envio síncrono de e-mails transacionais em aplicações monolíticas gera gargalos de latência e alto risco de falhas em cadeia. Este trabalho apresenta o Hermes, um ecossistema desacoplado de gateway de e-mails e mensageria assíncrona. Desenvolvido sob a metodologia Design Science Research (DSR), o sistema integra uma API REST e Workers em Node.js (hermes-api), um painel em Next.js (hermes-front) e um SDK em TypeScript (hermes-client), utilizando Redis, BullMQ, validação sintática Ahead-of-Time (AOT) e cache em memória de templates MJML/Handlebars. Testes de carga demonstraram latência média de 64,05 ms no enfileiramento (HTTP 201 Created) e vazão de até 717,33 e-mails/s em lote. O Hermes entrega um modelo arquitetural de código aberto, escalável e defensivo para infraestruturas de comunicação transacional.

Palavras-chave: Arquitetura de software, Gateway de e-mails, Mensageria assíncrona, Reds / BullMQ, MJML

Abstract

Synchronous transmission of transactional emails in monolithic applications creates latency bottlenecks and a high risk of cascading failures. This work presents Hermes, a decoupled ecosystem consisting of an email gateway and an asynchronous messaging system. Developed using the Design Science Research (DSR) methodology, the system integrates a Node.js-based REST API and workers (hermes-api), a Next.js dashboard (hermes-front), and a TypeScript SDK (hermes-client), leveraging Redis, BullMQ, Ahead-of-Time (AOT) syntax validation, and in-memory caching of MJML/Handlebars templates. Load tests showed a mean queuing latency of 64.05 ms (HTTP 201 Created) and a throughput of up to 717.33 emails/s in bulk ingestion. Hermes provides an open-source, scalable, and defensive architectural model for transactional communication infrastructure.

Keywords: Software architecture, Email gateway, Asynchronous messaging, Redis / BullMQ, MJML

1. Introdução

A crescente digitalização das organizações levou as aplicações Web a demandar arquiteturas resilientes e flexíveis (Lima, 2025). Nesse contexto, a Engenharia de Software vem migrando gradualmente do modelo monolítico para a arquitetura de microsserviços (Razzaq; Ghayyur, 2023), que segmenta o sistema em serviços autônomos, modulares e fracamente acoplados, facilitando a evolução e a manutenção do código (Picioli; Dezani; Carlson Filho, 2022; Valente, 2022).

Nesses sistemas, o envio de e-mails transacionais, como confirmações de cadastro, alertas de segurança e faturamento, é uma capacidade essencial. Contudo, rotinas síncronas de disparo acopladas à aplicação principal geram gargalos de rede, disputa por recursos do servidor e, sob picos de carga, indisponibilidade temporária do sistema (Picioli; Dezani; Carlson Filho, 2022). A lentidão ou a falha de um servidor SMTP remoto não deve comprometer a navegação do usuário, o que justifica isolar essa camada em um ecossistema assíncrono e orientado a eventos.

Para isso, este trabalho apresenta o Hermes, um gateway de e-mails transacionais composto por uma API assíncrona baseada em filas persistentes Redis/ BullMQ, um console administrativo em Next.js e um SDK cliente em TypeScript. O objetivo geral é desenvolver e avaliar um gateway que desacople o envio de notificações da aplicação consumidora. Os objetivos específicos são: (i) implementar a API e os workers com filas persistentes; (ii) construir o console e o SDK, com retentativas e rotação segura de credenciais; (iii) avaliar latência, vazão, escalabilidade horizontal e resiliência em testes de carga; e (iv) discutir as limitações da solução.

A hipótese central de pesquisa estabelece que o desacoplamento assíncrono mediado por filas persistentes Redis/BullMQ, aliado à pré-compilação Ahead-of-Time (AOT) de templates e ao tratamento defensivo no SDK, reduz significativamente a latência percebida pelas aplicações consumidoras e previne falhas em cadeia provocadas por oscilações na rede SMTP. O escopo desta investigação delimita-se à arquitetura de gateways auto-hospedados para envio de e-mails transacionais via REST/SMTP, focando na resiliência da camada de mensageria e na segurança da gestão de credenciais, sem abranger a gestão externa de reputação de IP ou entregabilidade em redes de destinos (DNSBL/IP warming).

O artigo está organizado em mais cinco seções: a Seção 2 apresenta a fundamentação teórica; a Seção 3 aborda os trabalhos relacionados; a Seção 4 detalha a metodologia e os materiais utilizados; a Seção 5 expõe o desenvolvimento e a implantação da arquitetura; a Seção 6 traz os resultados e discussões; e a Seção 7 encerra com as considerações finais e trabalhos futuros.

2. Fundamentação Teórica

2.1. Protocolo SMTP (Simple Mail Transfer Protocol)

O Simple Mail Transfer Protocol (SMTP) é o protocolo da camada de aplicação TCP/IP para a transferência de mensagens de correio eletrônico na Internet (Klensin, 2008). Opera no modelo cliente/servidor: a porta 25 é tradicionalmente usada na

retransmissão entre servidores de correio (Mail Transfer Agents), enquanto a submissão autenticada por aplicações e gateways, como o Hermes, utiliza as portas 587 (com STARTTLS, conforme RFC 6409) ou 465 (com TLS implícito, conforme RFC 8314) (Instituto Metrópole Digital, [s.d.]). Concebido sob o pressuposto de confiança entre redes acadêmicas, o protocolo original não verifica criptograficamente o remetente, o que viabilizou falsificação de cabeçalhos, SMTP smuggling e phishing (Longin, 2023; Wang et al., 2025) e motivou a adoção de SMTP AUTH, SPF, DKIM e DMARC.

2.2. Arquitetura de Microsserviços e Desacoplamento

Segundo Fowler e Lewis (2014), a arquitetura de microsserviços decompõe uma aplicação em pequenos serviços autônomos e fracamente acoplados, cada um executando em seu próprio processo e comunicando-se por protocolos leves, como HTTP/REST ou mensageria (Paiva; Santos, 2025). Diferentemente do monólito, concentrado em uma única unidade executável e sujeito a falhas em cadeia, essa abordagem favorece a resiliência, a manutenção e a implantação independente (Blinowski; Ojdowska; Przybyłek, 2022), ao custo de transferir a complexidade para a rede: latência, consistência eventual e observabilidade distribuída.

2.3. Escalabilidade Horizontal

Escalabilidade é a capacidade de absorver o aumento da demanda sem degradação de desempenho (Picioli; Dezani; Carlson Filho, 2022). A escalabilidade vertical adiciona recursos a um único servidor, o que limita o ganho e cria um ponto único de falha, enquanto a horizontal distribui a carga entre novas instâncias em múltiplos nós ou contêineres (Adrio et al., 2023). Com o desacoplamento, apenas os módulos sujeitos a gargalos de I/O ou de processamento, como os workers de mensageria, precisam ser replicados.

2.4. Integração Assíncrona e Sistemas de Mensageria

A comunicação síncrona bloqueia o emissor até a resposta do destinatário, tornando o sistema vulnerável a oscilações de rede e a indisponibilidades em cadeia (Costa; Queiroz, 2022). Como alternativa, o middleware orientado a mensagens (MOM), intermediado por um message broker, oferece dois níveis de desacoplamento: no espaço, pois emissor e consumidor não precisam conhecer a localização um do outro; e no tempo, pois ambos não precisam estar ativos simultaneamente, já que o broker retém as mensagens em filas até o consumo (Curry, 2004).

2.5. Conteinerização de Serviços

Contêineres empacotam a aplicação e suas dependências em ambientes isolados e portáteis (Docker Docs, 2021). Diferentemente das máquinas virtuais, compartilham o kernel do sistema hospedeiro, o que reduz o consumo de recursos e acelera a inicialização (Cavalcante, 2023). O fluxo baseia-se em Dockerfiles para gerar imagens e em ferramentas como o Docker Compose para gerenciar o ciclo de vida de múltiplos contêineres interdependentes (Docker Docs, 2021).

2.6. Linguagens Declarativas para E-mail Responsivo (MJML)

Os clientes de e-mail (Outlook, Apple Mail, Gmail) usam motores de renderização heterogêneos, o que dificulta layouts responsivos consistentes. O MJML é uma linguagem declarativa baseada em XML que abstrai tabelas HTML aninhadas e CSS inline, convertendo tags de alto nível em HTML compatível com diversos dispositivos e clientes (MJML, 2024).

2.7. Semântica de Entrega, Idempotência e Padrão Transactional Outbox

Em sistemas distribuídos de mensageria, garantir a entrega exata de uma mensagem sem duplicação representa um desafio clássico (Curry, 2004). A semântica de entrega at-least-once assegura que toda mensagem será processada ao menos uma vez pelo consumidor, exigindo a implementação de mecanismos de idempotência no receptor para neutralizar o impacto de retransmissões decorrentes de falhas de confirmação (ack). Além disso, a coordenação de alterações no estado relacional e o enfileiramento de mensagens em brokers distintos enfrenta o problema da gravação dupla (dual-write). Para mitigar inconsistentemente falhas de rede entre a base relacional e a fila assíncrona, o padrão Transactional Outbox insere o evento de mensageria na mesma transação ACID do banco de dados relacional, delegando a leitura e o despacho assíncrono a um processo relé independente (poller ou Change Data Capture - CDC).

2.8. Primitivos Criptográficos e Cifragem Autenticada

A proteção de credenciais em trânsito e em repouso exige primitivos criptográficos robustos contra ataques modernos. Para o armazenamento e verificação de chaves de acesso, a função memory-hard Argon2id (Biryukov et al., RFC 9106) combina resistência a ataques de canal lateral e otimização para execução em hardware genérico, prevenindo a aceleração de ataques de força bruta com GPUs ou ASICs. Para a proteção de dados confidenciais armazenados, o algoritmo AES (Advanced Encryption Standard) operando no modo Galois/Counter Mode (AES-256-GCM, conforme NIST SP 800-38D) provê Cifragem Autenticada com Dados Associados (AEAD) (Dworkin, 2007), garantindo simultaneamente a confidencialidade do conteúdo e a integridade da mensagem via tags de autenticação.

3. Trabalhos Relacionados

Entre as soluções comerciais em nuvem, o Amazon SES destaca-se pelo baixo custo por volume e pela escalabilidade da infraestrutura da AWS (Amazon Web Services, 2024), mas implica dependência de fornecedor, limites de envio em sandbox e aprovação manual para elevação de cotas, além de delegar à aplicação cliente as filas de retentativa, a concorrência e a compilação de layouts responsivos. SendGrid e Resend oferecem APIs REST ergonômicas, bibliotecas em TypeScript e rastreamento de entregabilidade (SendGrid, 2024; Resend, 2024). Em contrapartida, operam em nuvem fechada com precificação por faixa de envio e trafegam conteúdo e credenciais por servidores de terceiros, o que levanta questões de soberania de dados e de conformidade com a LGPD e o GDPR.

Entre as soluções de código aberto, o Postal é uma alternativa auto-hospedada ao SendGrid, com MTA dedicado e painel administrativo (Postal, 2024). Porém, transfere à organização a gestão da reputação de IP, o aquecimento de domínios (IP warming) e o tratamento de listas negras (DNSBL), e não oferece compilação antecipada de templates nem SDK com rotação de segredos sem indisponibilidade. O Novu orquestra notificações multicanal (e-mail, SMS, mensagens instantâneas e in-app) com múltiplos provedores (Novu, 2024), mas sua amplitude implica uma arquitetura e um consumo de recursos maiores do que o necessário para um gateway dedicado a e-mails transacionais, e não trata a compilação de MJML na ingestão.

Na literatura, o uso de mensageria assíncrona para mitigar latência e promover resiliência é bem documentado. Costa e Queiroz (2022) relatam um estudo de caso de fila de mensagens e orquestração de microsserviços em uma linha de produtos de software, e Blinowski, Ojdowska e Przybyłek (2022) e Adrio et al. (2023) comparam arquiteturas monolíticas e de microsserviços quanto a desempenho e escalabilidade. Esses trabalhos, contudo, tratam de filas genéricas de domínio, e aspectos próprios da comunicação transacional externa, como a latência do handshake SMTP, o custo de CPU da renderização de templates e os riscos de timing attack na rotação de chaves, são pouco explorados nas fontes consultadas. O Hermes posiciona-se, assim, entre a soberania das soluções self-hosted e a ergonomia dos serviços SaaS, com quatro diferenciais: (i) autonomia e neutralidade de provedores, com suporte a SMTP corporativo e OAuth2; (ii) validação MJML/Handlebars antecipada (AOT) no cadastro e cache da função pré-compilada nos workers; (iii) segurança em camadas (Seção 5.3.2); e (iv) SDK resiliente e compatível com Edge Runtimes. A Tabela 3 compara o Hermes com as alternativas analisadas.

A Tabela 3 sintetiza a análise comparativa entre o ecossistema Hermes e as soluções correlatas analisadas.

Tabela 3 - Análise comparativa entre o ecossistema Hermes e soluções correlatas

Solução	Implantação	Código Aberto	Mensageria	MJML Nativo	Backoff & Jitter
Amazon SES	Nuvem SaaS	Não (Fechado)	Parcial (via SQS)	Inexistente	Não nativo
SendGrid / Resend	Nuvem SaaS	Não (Fechado)	Sim (Interna)	Inexistente (JIT)	Básico
Postal	Auto-Hospedado	Sim (MIT)	Sim (RabbitMQ)	Inexistente	Não nativo
Novu	Híbrido	Sim (MIT)	Sim (Redis/ BullMQ)	Inexistente	Parcial
Hermes (Proposto)	**Auto-Hospedado**	**Sim (ISC)**	**Sim (Redis/ BullMQ)**	**Sim (validação AOT + cache)**	**Sim (jitter ±25%)**

Fonte: Autoria própria (2026).

4. Metodologia, Materiais e Métodos

O desenvolvimento do Hermes caracteriza-se como pesquisa aplicada e experimental, voltada aos gargalos de acoplamento e latência no disparo de e-mails transacionais em sistemas distribuídos. Adotou-se a Design Science Research (DSR) (Wieringa, 2014; Peffers et al., 2007), método orientado à resolução de problemas práticos que coincidem com demandas reais de partes interessadas, por meio da criação sistemática de artefatos de TI (Wieringa, 2014). O artefato é o ecossistema Hermes (API, console e SDK), avaliado para verificar a hipótese de que o desacoplamento assíncrono por filas persistentes Redis/BullMQ melhora o desempenho percebido e a tolerância a falhas das aplicações integradas.

4.1. Abordagem de Desenvolvimento e Ciclo de Vida

Adotou-se o desenvolvimento iterativo e incremental (Pressman; Maxim, 2016), que entrega o software em parcelas funcionais progressivas e permite adaptar-se a novos requisitos e desafios técnicos. Por ser um projeto de autoria individual, não se seguiram cerimônias de gestão de equipes. Os ciclos combinaram: (i) exploração e design de cada incremento (rotas HTTP, esquema relacional mínimo e contratos do SDK e do painel); (ii) implementação incremental, na ordem da cadeia de dependências, iniciando pelo núcleo de ingestão e filas (hermes-api), seguido do painel (hermes-front) e do SDK (hermes-client); (iii) integração e versionamento em Git, com commits incrementais e empacotamento em imagens Docker; e (iv) verificação empírica por testes de carga com Autocannon e servidor SMTP de captura Mailpit, em contêineres locais, para aferir os percentis de latência (p50, p90, p95 e p99), a vazão entre os nós (API, Redis, Worker e SMTP) e o comportamento dos webhooks.

4.2. Materiais e Tecnologias Utilizadas

O ecossistema divide-se em três componentes desacoplados: o back-end (hermes-api), o front-end (hermes-front) e a biblioteca cliente (hermes-client).

4.2.1. Back-end (Hermes - API)

O back-end e os workers foram desenvolvidos em Node.js com TypeScript e Express. A persistência usa PostgreSQL (PostgreSQL Development Group, 2022) via Drizzle ORM, e o enfileiramento assíncrono apoia-se em Redis (Redis, 2023) e na biblioteca BullMQ (BullMQ, 2024). A Tabela 1 sintetiza o acervo tecnológico.

A Tabela 1 sintetiza o acervo tecnológico empregado na API.

Tabela 1 - Acervo tecnológico utilizado no Back-end

Camada / Função	Tecnologia Adotada	Justificativa e Propósito
Runtime / Linguagem	*Node.js* / *TypeScript*	Ambiente assíncrono não bloqueante com tipagem estática e segurança em tempo de compilação.
Framework HTTP	*Express*	Servidor REST

Camada / Função	Tecnologia Adotada	Justificativa e Propósito
		minimalista de alto desempenho e fácil acoplamento de middlewares.
ORM / Query Builder	*Drizzle ORM*	Mapeador estritamente tipado que gera consultas SQL otimizadas sem overhead.
Banco de Dados	*PostgreSQL*	Persistência transacional segura para serviços, credenciais cifradas e logs de auditoria.
Fila e Cache	*Redis* / *BullMQ*	Gerenciamento de filas assíncronas em memória com suporte a retentativas e concorrência.
Autenticação	*Better Auth*	Gestão de sessões administrativas e suporte a login social via Google OAuth2.
Segurança & Hash	*Argon2id* / *AES-256-GCM*	Hash criptográfico para API Keys e cifragem simétrica em repouso de dados sensíveis.
Envio de E-mails	*Nodemailer*	Cliente para estabelecimento de conexões SMTP/TLS e envio de mensagens.
Template Engine	*MJML* + *Handlebars*	Validação sintática AOT e transpilação MJML combinadas com cache de AST e injeção dinâmica via Handlebars.
Sanitização/Validação	*DOMPurify* / *Zod*	Higienização de HTML contra ataques XSS e validação estrita de esquemas HTTP.

Fonte: Autoria Própria (2026)

4.2.2. Front-end (Hermes - Painel Administrativo)

O console foi construído em Next.js (App Router) (Vercel, 2026) com React, Tailwind CSS v4 e componentes Radix UI/shadcn/ui. O estado assíncrono é gerenciado com TanStack Query, e as métricas em tempo real usam Server-Sent Events (SSE) e gráficos ECharts. A Tabela 2 apresenta o acervo tecnológico da interface.

Tabela 2 - Acervo Tecnológico utilizado no Front-end

Camada / Função	Tecnologia Adotada	Justificativa e Propósito
Framework Base	*Next.js* (App Router)	Roteamento baseado em arquivos com otimização híbrida de renderização (SSR/CSR).
Biblioteca de UI	*React*	Renderização declarativa e componentização de interfaces reativas.
Estilização / Design	*Tailwind CSS* / *Radix UI*	Classes utilitárias de alto desempenho e componentes semânticos acessíveis.
Estado Assíncrono	*TanStack Query*	Gerenciamento de cache local e sincronização automática com a API REST.
Analytics e Gráficos	*Apache ECharts*	Gráficos vetoriais fluidos para acompanhamento de volumetria e status de envios.
Editor de Código	*Monaco Editor*	Editor integrado para confecção e visualização em tempo real de e-mails MJML.
Onboarding	*Driver.js*	Script de engajamento para tour guiado no primeiro acesso do usuário.

Fonte: Autoria Própria (2026)

4.2.3. SDK Client (Hermes - SDK)

Para facilitar a adoção por sistemas parceiros, desenvolveu-se o SDK oficial @ruanlopes1350/hermes-client, cujo núcleo é agnóstico ao ambiente de execução:

substitui o EventEmitter nativo do Node.js, indisponível sem polyfills em ambientes de borda, por uma implementação mínima (LiteEventEmitter) e isola o EnvAdapter, que depende do módulo fs, no sub-pacote @ruanlopes1350/hermes-client/node. Assim, o pacote base permanece compacto, compatível com tree-shaking e com Edge Runtimes (Cloudflare Workers, Vercel Edge e AWS Lambda@Edge). A resiliência é nativa: falhas transitórias (HTTP 408, 429, 500, 502, 503 e 504) disparam até 3 retentativas com backoff exponencial (atraso base de 1.000 ms, fator 2 e jitter de ±25%), o que evita o efeito thundering herd. O SDK inclui ainda middlewares para Express, Next.js e Fastify que validam webhooks pela Web Crypto API (crypto.subtle) com comparação em tempo constante por acumulação booleana, sem depender de node:crypto, e o utilitário HermesStream para status em tempo real via SSE. Os padrões de projeto e os mecanismos de segurança são detalhados na Seção 4.3.

4.2.4. Infraestrutura e Conteinerização

A infraestrutura baseia-se em Docker e Docker Compose (Docker Docs, 2021; Cavalcante, 2023). PostgreSQL 15 e Redis 7 operam em contêineres dedicados, isolados em uma rede virtual própria (bridge), sem exposição desnecessária de portas. A API e os workers (hermes-worker) usam multi-stage builds sobre imagens Linux Alpine, resultando em artefatos inferiores a 150 MB, com inicialização rápida e menor superfície de ataque.

5. Desenvolvimento e Implantação

Esta seção descreve o ciclo de vida do projeto, a arquitetura, os padrões de projeto, as decisões de segurança, os fluxos de processamento e o uso de Inteligência Artificial (IA) no desenvolvimento.

5.1. Ciclo de Vida e Evolução Temporal do Ecossistema

O ecossistema, de autoria individual e código aberto sob a licença ISC (https://github.com/RuanLopes1350), organiza-se em três repositórios Git (hermes-api, hermes-front e hermes-client) que somam 229 commits no primeiro semestre de 2026, em entregas incrementais (Pressman; Maxim, 2016). A Tabela 4 resume a distribuição. A evolução seguiu a cadeia de dependências: primeiro o hermes-api, com o esquema PostgreSQL (Drizzle ORM), a validação de domínios por registros DNS MX, a ingestão em fila Redis/BullMQ com orquestração dinâmica de réplicas (hermes_scaler) e as travas de autorização; depois o hermes-front, com o editor Monaco para templates MJML, a sanitização com DOMPurify na pré-visualização, o TanStack Query e a escuta de eventos por SSE; e, por fim, o SDK, com compilação dual (ESM/CJS), retentativas exponenciais e middlewares de webhook HMAC-SHA256.

Tabela 4 - Síntese dos repositórios do ecossistema Hermes

Repositório	Papel no Sistema	Commits	Tecnologias Principais
hermes-api	Back-end REST, Fila e Workers	103	*Node.js, Express, Postgres, Drizzle,*

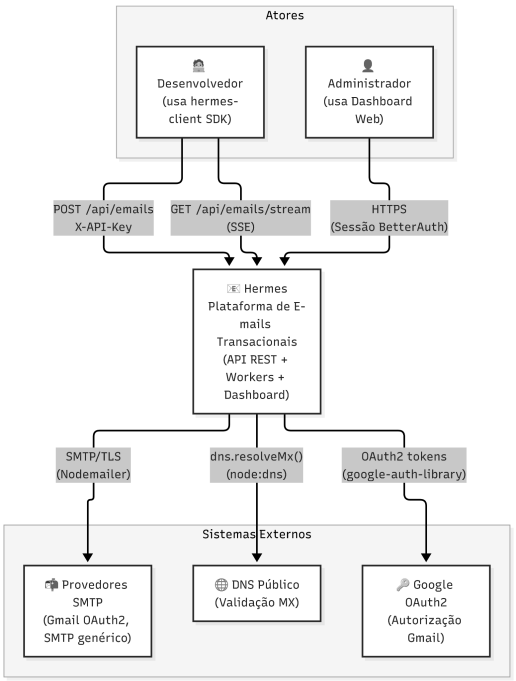
Repositório	Papel no Sistema	Commits	Tecnologias Principais
			Redis, BullMQ
hermes-front	Console e Dashboard Administrativo	93	*Next.js (App Router), React, Tailwind, ECharts*
hermes-client	SDK Cliente e Handler de Webhooks	33	*TypeScript, tsup (ESM/CJS), Web Crypto API*

Fonte: Autoria Própria (2026)

5.2. Arquitetura Proposta do Ecossistema Hermes

A arquitetura segue o paradigma de microsserviços fracamente acoplados e é descrita pelo Modelo C4 (Context, Containers, Components, Code). No Nível 1 (Figura 1), o Hermes funciona como plataforma centralizada de mensageria transacional que isola as aplicações consumidoras (via API REST ou SDK) e os administradores das interações diretas com provedores SMTP e serviços OAuth2.

Figura 1 - Arquitetura geral do ecossistema.

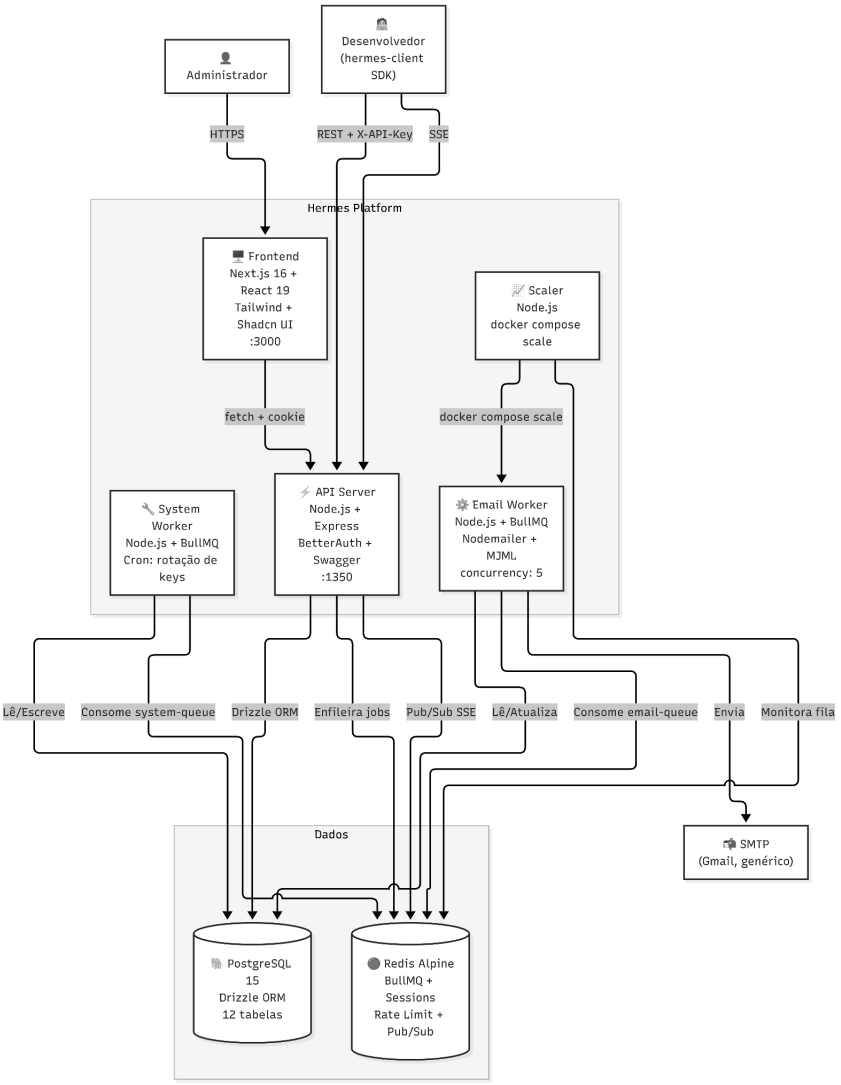


Fonte: Autoria Própria (2026)

No Nível 2 (Figura 2), o sistema se decompõe em unidades implantáveis de forma autônoma: o console (hermes-front), o servidor de ingestão REST (hermes-api), os workers e scalers de orquestração, o PostgreSQL 15 e o buffer de mensageria Redis

Alpine. Internamente (Nível 3), o back-end adota arquitetura em camadas (Routes, Controllers, Services e Repositories), com persistência via Drizzle ORM.

Figura 2 - Estrutura Hierárquica do Sistema

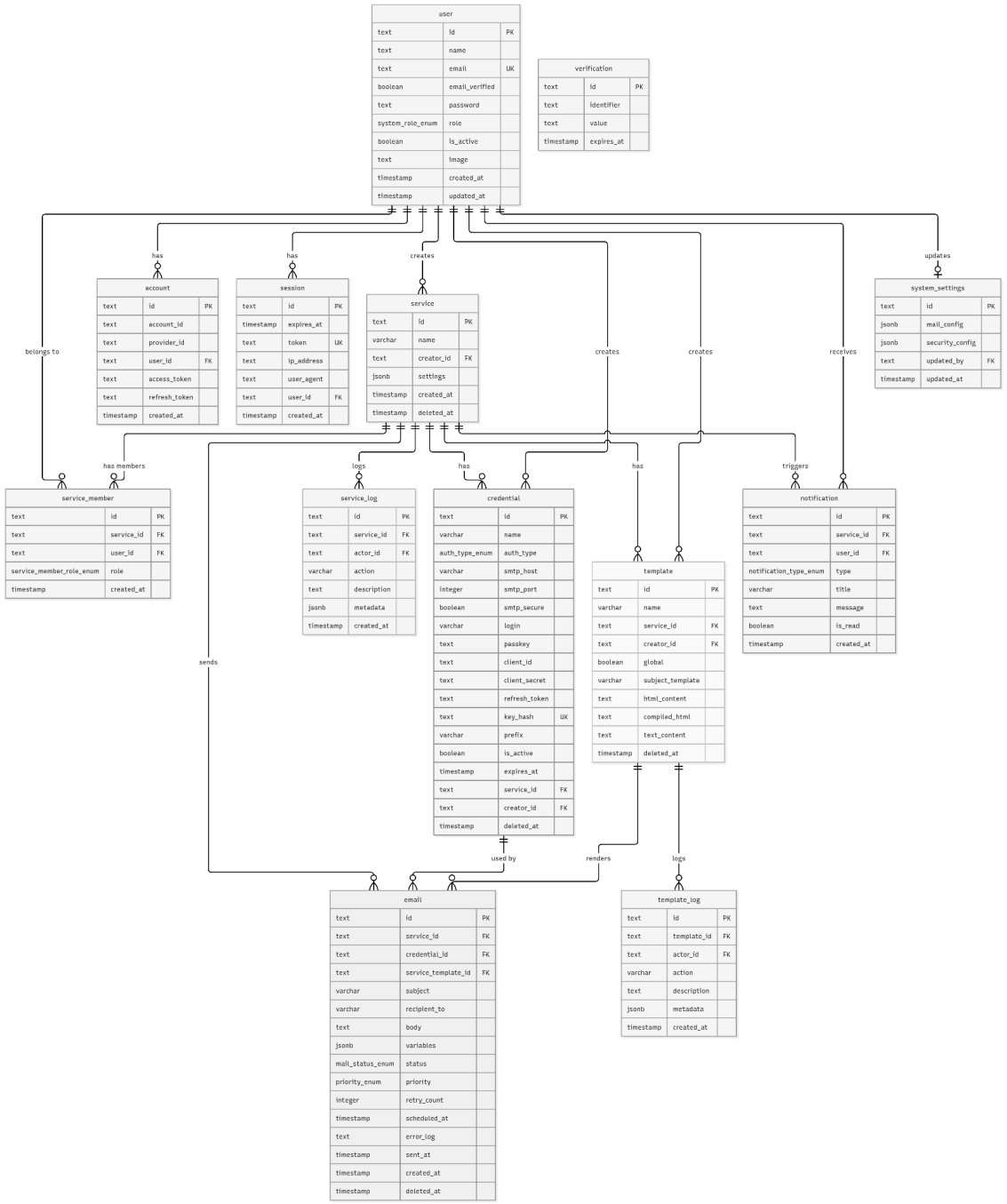


Fonte: Autoria Própria (2026)

A ingestão e a execução são estritamente separadas. Na ingestão, a API valida o payload com Zod, verifica os registros DNS MX do domínio do destinatário (cache em memória de 10 minutos), grava o e-mail com status pending no PostgreSQL e despacha o job para a fila BullMQ no Redis. O envio SMTP é delegado aos workers (emailWorker.ts), cujo número de contêineres é dimensionado pelo hermes_scaler conforme a volumetria da fila e o hardware detectado. Para otimizar os templates, combinam-se duas estratégias: (i) validação sintática estrita Ahead-of-Time (AOT) do MJML (modo strict) e das expressões Handlebars no cadastro ou edição, que impede o armazenamento de layouts defeituosos; e (ii) cache em memória por processo (templateCache.ts, TTL de 60 s) da função pré-compilada do Handlebars (Handlebars.TemplateDelegate) nos workers, que evita leituras repetidas ao banco e recompilações em envios em lote.

O modelo relacional (Figura 3) isola logicamente os clientes pela entidade service, que vincula credenciais cifradas, templates e logs de auditoria a namespaces independentes. O front-end usa Next.js (App Router), TanStack Query e SSE para atualizar o painel em tempo real, e a implantação é padronizada via Docker Compose com multi-stage builds sobre imagens Alpine.

Figura 3 - Diagrama Entidade Relacionamento



Fonte: Autoria Própria (2026)

5.3. Padrões de Projeto e Mecanismos de Segurança

5.3.1. Padrões de Projeto (Design Patterns)

Adotaram-se padrões do catálogo GoF (Gamma et al., 1994), adaptados ao ambiente assíncrono do TypeScript/Node.js: (i) Builder, nas classes EmailBuilder e BulkEmailBuilder do SDK, que permitem montar e-mails por encadeamento de métodos (.to().subject().useTemplate().send()), separando a representação do objeto de sua validação; (ii) Adapter, na interface StorageAdapter, que permite alternar entre armazenamento em memória (MemoryAdapter), em provedores chave-valor assíncronos para ambientes serverless e de borda (GenericKVAdapter, com Redis ou Cloudflare KV) e em variáveis de ambiente (EnvAdapter), sem alterar o código do cliente; (iii) Chain of Responsibility/Middleware, na API Express, em que cada requisição passa em sequência por sanitização de cabeçalhos, autorização do contexto do serviço, rate limiting e validação Zod, sendo abortada antes de alcançar a camada de serviço; e (iv) Observer, na classe LiteEventEmitter (eventos keyRotated, retry e error) e no módulo HermesStream (SSE), que mantêm os componentes desacoplados.

5.3.2. Mecanismos de Criptografia e Segurança da Informação

Como gateway centralizado, o Hermes gerencia credenciais de alto privilégio (chaves de API, senhas SMTP e tokens OAuth) e, por isso, adota defesa em profundidade:

1. **Criptografia simétrica em repouso (AES-256-GCM).** As senhas SMTP e os refresh tokens do Google OAuth2 são cifrados com AES-256 no modo Galois/ Counter (GCM) (NIST SP 800-38D; Dworkin, 2007), que provê criptografia autenticada (AEAD) com tag de 16 bytes e vetor de inicialização (IV) exclusivo por registro. Isso impede a leitura em caso de vazamento do banco e a adulteração silenciosa dos dados.
2. **Chaves de API com Argon2id.** As chaves seguem o formato hm_[prefixo]. [segredo] (prefixo público de 8 caracteres hexadecimais e segredo aleatório de 64 caracteres hexadecimais, 256 bits de entropia) e nunca são armazenadas em texto claro. O gateway aplica a função memory-hard Argon2id (RFC 9106; Biryukov et al., 2021), resistente a ataques de força bruta com GPU/ASIC. Embora chaves de alta entropia admitissem resumos rápidos (SHA-256 ou HMAC), optou-se pelo Argon2id como defesa em profundidade contra o comprometimento offline da base. Para não degradar a latência, apenas o prefixo hm_ mais 8 caracteres hexadecimais fica em texto claro e indexado (B-Tree) no PostgreSQL, de modo que cada requisição executa uma única verificação Argon2id.
3. **Webhooks assinados com HMAC-SHA256:** A rotação de credenciais envia POST aos webhooks dos clientes com o cabeçalho X-Hermes-Signature, o HMAC-SHA256 do corpo bruto calculado com o segredo compartilhado (Krawczyk; Bellare; Canetti, 1997). O HMAC garante autenticidade e integridade, mas não cifra o payload, que carrega a nova chave (newApiKey); por isso os endpoints de produção devem operar sob HTTPS (TLS). No SDK, a verificação usa a Web Crypto API (crypto.subtle) com comparação em tempo constante, mitigando ataques de temporização em qualquer runtime JavaScript.

4. **Higienização contra XSS:** Na pré-visualização de templates (previewTemplate), o HTML resultante do MJML é sanitizado no back-end com DOMPurify sobre uma instância virtual de DOM (JSDOM), que remove tags <script>, atributos de evento em linha (onload, onerror) e URLs javascript:. Os caminhos internos de diretórios são suprimidos nas mensagens de erro de compilação, evitando vazar a estrutura de arquivos do servidor.
5. **Validação de domínios, autorização e SSRF:** A API audita os domínios de destino por DNS MX (node:dns/promises, cache de 10 minutos e concorrência limitada em lotes), rejeitando domínios inexistentes antes que ocupem a fila. O serviço emissor e a credencial ativa são extraídos exclusivamente do contexto autenticado da API Key (requireApiKey), o que anula a manipulação de parâmetros de rota para acessar recursos de outros serviços (falhas de autorização em nível de objeto e de função, BOLA e BFLA; OWASP, 2023). O cadastro de webhooks é restrito ao papel owner, com timeout de 5 s, mas a mitigação completa de SSRF e DNS rebinding exige validar o IP resolvido na abertura do socket TCP, aspecto discutido na Seção 6.2.

5.4. Fluxos de Processamento e Implementação Prática

Os dois fluxos centrais desacoplam a ingestão do trabalho pesado de rede: o envio assíncrono de e-mails e a rotação de chaves de API com notificação por webhook.

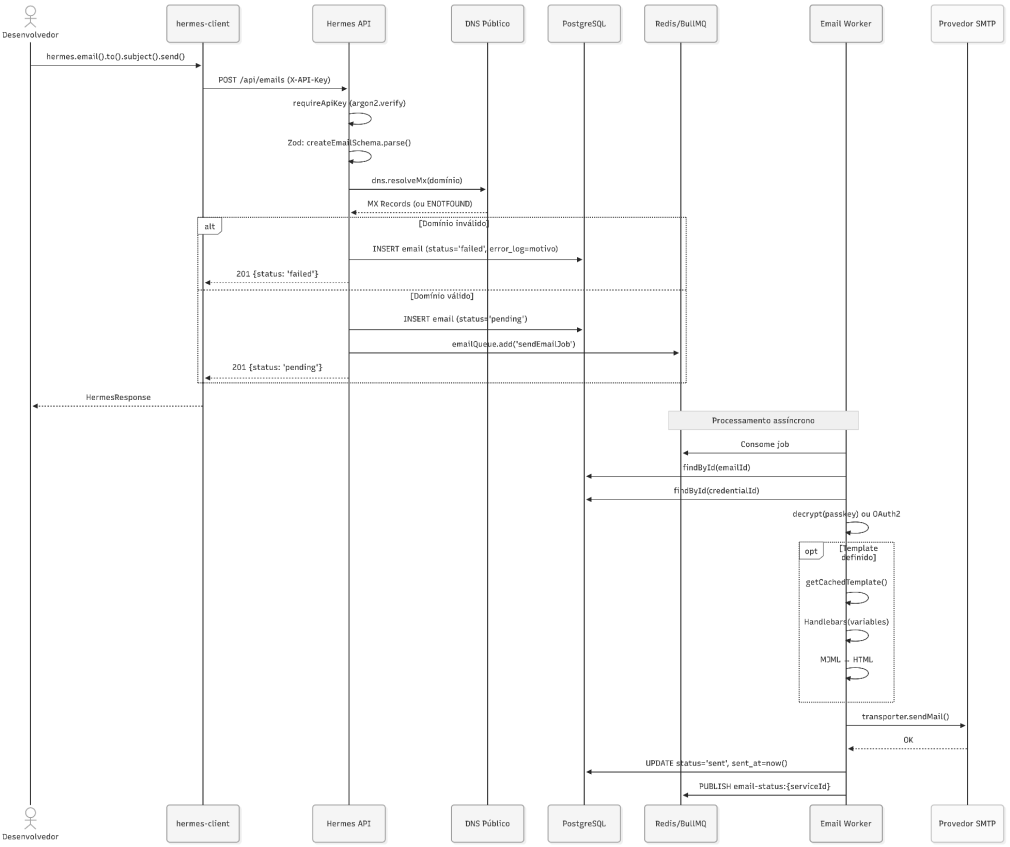
5.4.1. Fluxo de Processamento e Disparo de E-mails Transacionais

O fluxo recebe a requisição do cliente e garante a persistência e a entrega ao servidor SMTP sem bloquear a aplicação consumidora (Figuras 4 e 5):

1. **Ingestão e validação.** O cliente envia POST /emails (ou /emails/bulk) com o cabeçalho X-API-Key. O middleware localiza a credencial pelo prefixo indexado (hm_), verifica-a com Argon2id, carimba o contexto do serviço e da credencial e segue para a validação Zod e a checagem DNS MX do destinatário.
2. **Registro e enfileiramento.** Com MX válido, o serviço grava o e-mail como pending no PostgreSQL e despacha o job (emailId, serviceId e variáveis) ao BullMQ sobre Redis (persistência em volume RDB). A API responde 201 Created (via CommonResponse.created) indicando a criação bem-sucedida do recurso de e-mail no banco de dados com o emailId, liberando a aplicação cliente enquanto o processamento prossegue assincronamente.
3. **Processamento pelo worker.** Em segundo plano, com concorrência de 5 tarefas por instância, o emailWorker.ts consome o job. Verifica o estado no PostgreSQL (aborta se já for sent, como guarda de idempotência no consumo), recupera as configurações, descriptografa a credencial SMTP ou o refresh token OAuth2 (AES-256-GCM) e obtém o transportador Nodemailer, em cache com pool de até 5 conexões.
4. **Renderização.** Quando há template, o worker obtém a função pré-compilada do Handlebars no cache por processo (templateCache.ts, TTL de 60 s), interpola as variáveis no assunto e no corpo e, se o conteúdo for MJML, transpila para HTML responsivo e gera a alternativa em texto puro (html-to-text).
5. **Transmissão SMTP.** O worker abre conexão segura e executa EHLO, AUTH, MAIL FROM, RCPT TO e DATA via Nodemailer.

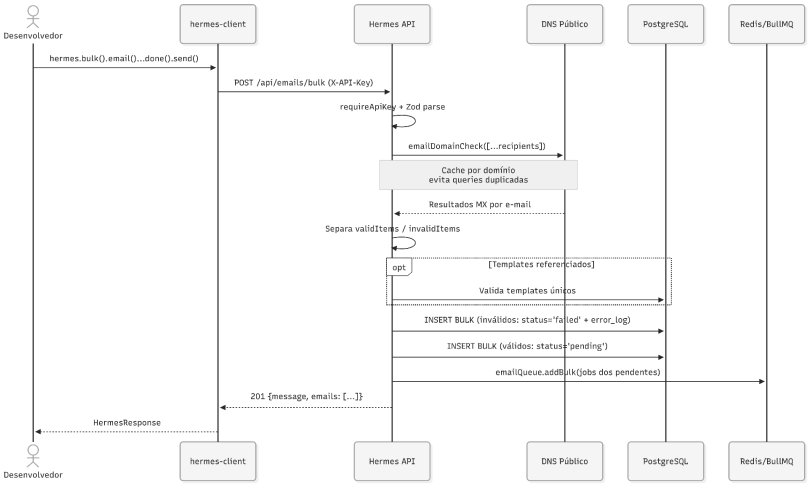
6. **Estados finais.** Na entrega, o status passa a sent com sent_at. Em falha transitória, o BullMQ faz até 3 retentativas com backoff exponencial (base de 5.000 ms), com status retrying; se persistir, o estado final é failed, com o diagnóstico em error_log e possibilidade de reenfileiramento manual no painel.

Figura 4 - Envio Assíncrono E-mail Único



Fonte: Autoria Própria (2026)

Figura 5 - Envio Assíncrono E-mail em Lote (Bulk)



Fonte: Autoria Própria (2026)

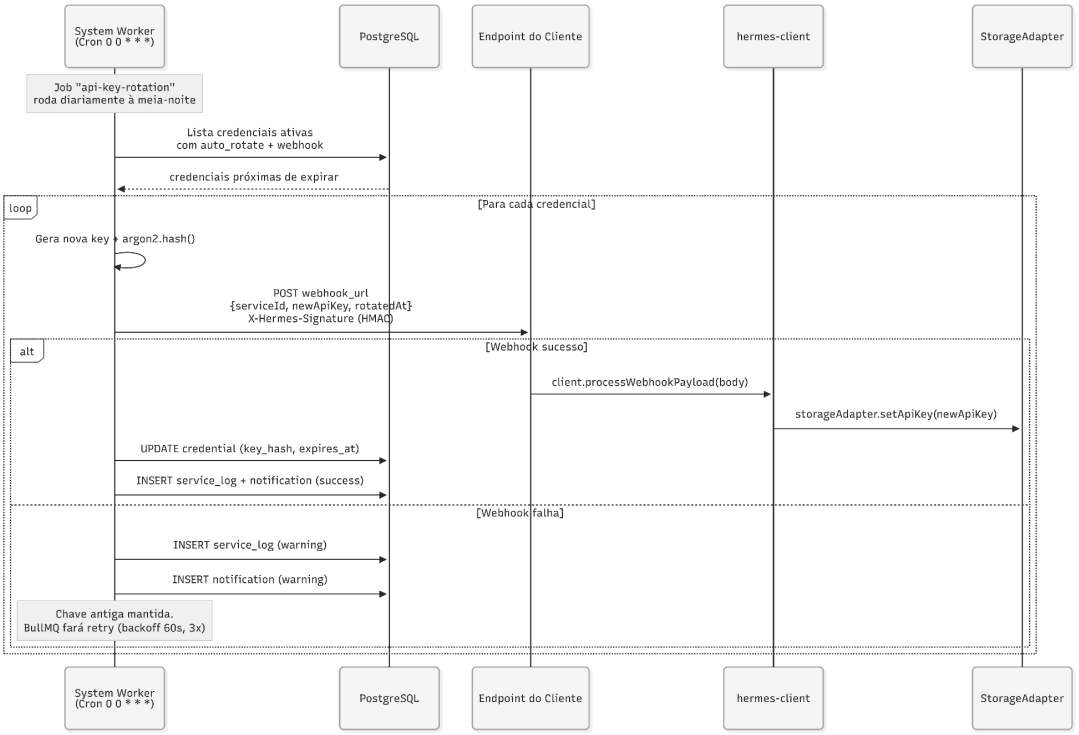
5.4.2. Fluxo de Rotação de Chaves de API e Notificação via Webhook

Para reduzir o risco de chaves de longo prazo comprometidas, o Hermes implementa rotação preventiva com janela de antecedência e confirmação prévia por webhook (webhook-first commit) (Figura 6):

1. **Solicitação e agendamento.** A rotação é manual, pelo console (POST / services/:serviceId/credentials/:id/rotate), ou automática, pelo systemWorker.ts: um job repetível do BullMQ (api-key-rotation, cron 0 0 * * *, à meia-noite) identifica credenciais cujo expiresAt esteja dentro do limiar rotate_threshold_days (padrão de 3 dias) e enfileira um micro-job rotate-single-key com identificador determinístico diário (rotate-$credentialId$ −{YYYY-MM-DD}), evitando duplicidade.
2. **Geração da nova chave.** Em memória, gera-se uma chave no formato hm_[prefixo_8_hex].[segredo_64_hex] (32 bytes aleatórios para o segredo e 4 bytes para o prefixo) e calcula-se seu hash Argon2id, sem sobrescrever o registro ativo.
3. **Webhook-first commit.** A chave antiga permanece válida no PostgreSQL enquanto o worker tenta entregar a nova ao cliente. Se o endpoint estiver indisponível ou retornar erro, a atualização é abortada e o BullMQ agenda até 3 retentativas com backoff exponencial (base de 60 s), preservando a operação com a chave anterior.
4. **Efetivação atômica.** Após o HTTP 2xx do webhook (ou imediatamente, na rotação manual), o worker executa um UPDATE atômico na tabela credential, substituindo prefix e key_hash, prorrogando expiresAt por rotation_interval_days (padrão de 30 dias) e registrando o evento no log de auditoria e nas notificações SSE.
5. **Despacho assinado.** O webhookDispatcher.ts envia POST (timeout de 5 s) à webhook_url do serviço com o JSON { serviceId, credentialId, newApiKey, rotatedAt, expiresAt } e o cabeçalho X-Hermes-Signature (HMAC-SHA256 com o webhook_secret). Como o HMAC não garante confidencialidade, o endpoint em produção deve usar HTTPS.
6. **Recepção no SDK.** O middleware do SDK (Express, Next.js ou Fastify) lê o corpo bruto, valida a assinatura com Web Crypto em tempo constante e chama setApiKey() no StorageAdapter configurado, atualizando a credencial sem reiniciar a aplicação. No Express, a rota deve usar express.raw, para que o HMAC seja recalculado sobre a sequência exata de bytes recebida, sem alterações de parsers JSON.

Figura 6 - Fluxo de rotação de chaves com janela de antecedência

e validação por webhook



Fonte: Autoria Própria (2026)

5.5. Processo de Desenvolvimento Apoiado por Inteligência Artificial

Os Modelos de Linguagem de Larga Escala (LLMs) e os assistentes agênticos de codificação vêm atuando como parceiros virtuais de programação em pares na exploração arquitetural e na refatoração (Lima, 2025; Valente, 2022). No Hermes, utilizaram-se assistentes agênticos, com destaque para o Claude Code, no design de interface, no alinhamento estético e no suporte técnico, orientados por arquivos de contexto (AGENTS.md e CLAUDE.md). Para controlar a qualidade, adotaram-se revisão manual de código, inspeção estática de tipos e validação de conformidade técnica antes de cada commit. Os assistentes aceleraram a prototipagem de interface e a estruturação preliminar de componentes, enquanto as decisões críticas de arquitetura e segurança (AES-256-GCM, Argon2id indexado por prefixo, validação DNS MX, controle de autorização por middleware e dimensionamento dinâmico de filas no BullMQ) foram inspecionadas, refatoradas e validadas diretamente pelo desenvolvedor.

6. Resultados e Discussões

6.1 Ambiente de Testes e Cenários de Avaliação

A avaliação mediu o desempenho, a latência e a vazão do gateway sob diferentes regimes de concorrência (Blinowski; Ojdowska; Przybyłek, 2022). Os experimentos rodaram em uma estação com processador Intel Core i5-1334U de 13ª geração (12 vCPUs) e 7,6 GiB de RAM alocados ao WSL2 (Ubuntu 26.04 LTS, kernel Linux 6.6), com Node.js v22.22.1 e Docker Compose (Cavalcante, 2023). O ambiente reuniu

PostgreSQL 15, Redis Alpine, a hermes-api e instâncias escaláveis do hermes-worker (concorrência 5 por processo e pool de até 5 conexões SMTP no Nodemailer), que enviam a uma instância conteinerizada do Mailpit (axllent/mailpit:latest, SMTP na porta 1025 e API HTTP na 8025), o que permite auditar a entrega sem tráfego externo. A carga foi gerada com o Autocannon (v8.0) e scripts de telemetria em TypeScript. Mediram-se média, desvio padrão, p50, p90, p95 e p99 de latência, vazão (req/s), taxa de sucesso de enfileiramento (HTTP 201 Created) e tempo de drenagem das filas. Três cenários foram definidos:

- **Cenário A (baseline síncrono vs. ingestão assíncrona unitária):** N = 100 requisições sequenciais (1 VU) em três modalidades: (i) disparo síncrono direto ao Mailpit, com transpilação MJML na requisição; (ii) o mesmo disparo com RTT externo simulado (+240 ms de handshake TLS/SMTP típico de provedores em nuvem); e (iii) ingestão assíncrona em POST /api/emails, com Argon2id, Zod, DNS MX, persistência no PostgreSQL e enfileiramento no BullMQ.
- **Cenário B (carga concorrente e lote):** 500 requisições ao endpoint unitário com 10, 25 e 50 conexões simultâneas (VUs) e ingestão massiva em POST /api/emails/bulk, com 20 lotes de 50 destinatários (1.000 e-mails), usando inserção em lote no PostgreSQL e emailQueue.addBulk no Redis.
- **Cenário C (renderização e escalabilidade horizontal):** custo de CPU e latência da renderização de 500 templates MJML/Handlebars sob três estratégias de cache e tempo de drenagem de uma fila pré-carregada com 500 jobs reais com 1, 2 e 4 instâncias do hermes-worker (5, 10 e 20 slots simultâneos de envio ao Mailpit).

6.2 Desempenho da Ingestão de Requisições e Processamento de Filas

No fluxo síncrono de um monólito, cada envio faz o consumidor aguardar a compilação do template, o handshake SMTP e a confirmação do destino. No baseline local contra o Mailpit, a latência média foi de 71,42 ms (p95 de 77,36 ms). Com o RTT externo simulado de +240 ms, subiu para 310,41 ms (p95 de 321,90 ms), com vazão de 3,22 req/s. Como esse acréscimo é simulado, o resultado deve ser lido como estimativa para provedores SMTP externos, e não como medição em um provedor real.

No Cenário A, a hermes-api localizou a chave pelo prefixo indexado (hm_), verificou-a com Argon2id, validou o payload com Zod, checou o DNS MX (com cache), gravou o e-mail como pending no PostgreSQL via Drizzle ORM e alocou o job no BullMQ, retornando 201 Created com o emailId. A latência média foi de 64,05 ms (desvio padrão de 5,63 ms; p50 de 62,39 ms; p90 de 67,68 ms; p95 de 69,31 ms; p99 de 83,21 ms), com 100% de sucesso. Cerca de 58 ms desse tempo correspondem ao custo deliberado do Argon2id. Portanto, no baseline local o ganho unitário é pequeno (71,42 ms contra 64,05 ms), e a vantagem relevante aparece quando há latência de rede externa, que o enfileiramento retira do caminho da requisição.

Sob carga concorrente (Cenário B), a vazão estabilizou-se em cerca de 25 req/s (pico de 28 req/s), o teto imposto pelo Argon2id. As latências médias e p95 foram de 380,42 ms e 674,00 ms com 10 VUs (500/500 respostas 2xx), 952,61 ms e 1.126,00 ms com 25 VUs (500/500) e 1.817,71 ms e 2.037,00 ms com 50 VUs (499/500, 99,8%). O baseline síncrono foi medido apenas com 1 VU, de modo que a comparação sob

concorrência não foi realizada. Na ingestão em lote, uma única verificação Argon2id, um INSERT multilinha e um único addBulk por lote de 50 destinatários resultaram em latência média de 69,70 ms por lote (p50 de 67,55 ms; p90 de 71,09 ms; p95 de 74,95 ms), ou seja, 1,39 ms por e-mail e 717,33 e-mails/s de ingestão. A Tabela 5 consolida os Cenários A e B.

Tabela 5 - Métricas de latência e vazão na camada de ingestão (Cenários A e B)

Cenário / Operação	Amostra / VUs	Média ± DP (ms)	p50 (ms)	p90 (ms)	p95 (ms)	p99 (ms)	Vazão (req/s)	Sucesso
Baseline Síncrono Local (Mailpit)	N=100 / 1 VU	71,42 ± 7,62	69,34	73,22	77,36	99,88	14,00	100%
Baseline Síncrono SMTP Externo (RTT simulado)	N=100 / 1 VU	310,41 ± 9,47	309,50	319,08	321,90	347,08	3,22	100%
Cenário A: Hermes Unitário	N=100 / 1 VU	64,05 ± 5,63	62,39	67,68	69,31	83,21	15,61	100%
Cenário B: Concorrente 10 VUs	N=500 / 10 VUs	380,42 ± 89,81	365,00	464,00	674,00	741,00	25,00	100%
Cenário B: Concorrente 25 VUs	N=500 / 25 VUs	952,61 ± 122,01	960,00	1.060,00	1.126,00	1.157,00	25,00	100%
Cenário B: Concorrente 50 VUs	N=500 / 50 VUs	1.817,71 ± 376,84	1.910,00	2.003,00	2.037,00	2.061,00	24,95	99,8%
Cenário B: Bulk (20x50 = 1.000)	20 lotes / 1 VU	69,70 ± 8,78 (1,39/msg)	67,55	71,09	74,95	106,72	14,35 (717,3 msg/s)	100%

Fonte: Autoria Própria (2026)

No Cenário C, a renderização de 500 templates de boas-vindas (MJML + Handlebars + html-to-text) sem cache (SELECT no PostgreSQL, Handlebars.compile e mjml2html a cada mensagem) levou 1.733,86 ms (3,46 ms por mensagem; p95 de 4,86 ms; 288,37 renderizações/s). Com o cache da função pré-compilada do Handlebars por 60 s (templateCache.ts) mais mjml2html, o tempo caiu para 826,12 ms (1,65 ms por mensagem; p95 de 2,85 ms; 605,24 renderizações/s), um ganho de 2,10x e redução de 52,3%. Como referência, o HTML pré-compilado em modo AOT (compiled_html interpolado pela função em cache, sem mjml2html) concluiu em 208,43 ms (0,42 ms por mensagem; p95 de 0,96 ms; 2.398,94 renderizações/s; 8,32x), o que quantifica o trade-off discutido na Seção 6.2.

Na drenagem de 500 e-mails reais pré-enfileirados (com descriptografia AES-256-GCM, renderização, envio Nodemailer em pool e atualização para sent), 1 worker (concorrência 5) levou 6,81 s (73,46 e-mails/s; latência média por job de 67,94 ms; p95 de 101,00 ms). Com 2 workers, o tempo caiu para 4,29 s (116,54 e-mails/s; speedup de 1,59x) e, com 4 workers (20 slots), para 3,03 s (164,79 e-mails/s; speedup de 2,25x), uma redução de 55,5%. O ganho é sublinear em relação ao número de workers, e a latência média por job cresce de 67,94 ms para 119,03 ms, indício de contenção de recursos na estação única, que não foi analisada em detalhe. Em todas as baterias houve 100% de entrega auditada (500/500 registros sent no PostgreSQL e 500/500 mensagens verificadas na API do Mailpit). A Tabela 6 consolida o Cenário C.

Tabela 6 - Desempenho de renderização de templates e escalabilidade horizontal dos Workers (Cenário C)

Experimento / Configuração	Volume (Jobs)	Tempo Total	Média ± DP (ms/ job)	p50 / p95 (ms)	Vazão (msg/s)	Speed up	Entregues (DB/ SMTP)
Render: Sem Cache (DB + MJML)	500 renders	1,73 s	3,46 ± 1,02	3,16 / 4,86	288,37	1,00x	500 / 500
Render: Cache AST + MJML (Hermes)	500 renders	0,83 s	1,65 ± 0,51	1,46 / 2,85	605,24	2,10x	500 / 500
Render: HTML AOT + Cache AST	500 renders	0,21 s	0,42 ± 0,21	0,35 / 0,96	2.398,94	8,32x	500 / 500
Fila BullMQ: 1 Worker (conc=5)	500 e-mails	6,81 s	67,94 ± 15,30	63,00 / 101,00	73,46	1,00x	500 / 500 (100%)
Fila BullMQ: 2 Workers (conc=10)	500 e-mails	4,29 s	85,14 ± 23,56	80,00 / 122,00	116,54	1,59x	500 / 500 (100%)
Fila BullMQ: 4 Workers (conc=20)	500 e-mails	3,03 s	119,03 ± 42,48	109,00 / 164,00	164,79	2,25x	500 / 500 (100%)

Fonte: Autoria Própria (2026)

6.3 Avaliação de Resiliência, Retentativas e Rotação de Credenciais

A resiliência foi avaliada por injeção controlada de falhas transitórias, instabilidades do servidor SMTP e rotação de credenciais sob tráfego ativo. Indisponibilidades momentâneas de I/O são eventos inevitáveis em microsserviços (Costa; Queiroz, 2022); sem proteção no cliente, resultam em perda de requisições ou em esgotamento do servidor por retentativas desordenadas (thundering herd).

O limitador de taxa da API (emailApiRateLimiter, 100 requisições por minuto por credencial) e o SDK foram submetidos a estresse e a injeção de erros HTTP (408, 429, 500, 502, 503 e 504). Em uma rajada de 110 requisições consecutivas em POST / api/emails, o gateway aceitou exatamente as 100 primeiras (201 Created) e bloqueou as 10 excedentes com 429 Too Many Requests. Em 10 execuções do algoritmo withRetry diante de HTTP 503 simulado, o backoff exponencial com jitter (base de 1.000 ms, fator 2, limite de 3 tentativas e dispersão de ±25%) comportou-se conforme o esperado: a 1ª retentativa (janela teórica de 750 a 1.250 ms) teve atraso médio de 1.043,50 ms (desvio padrão de 132,18 ms; mínimo de 821,00 ms; p50 de 1.056,00 ms; máximo de 1.210,00 ms), e a 2ª (janela de 1.500 a 2.500 ms) teve média de 1.878,60 ms (desvio padrão de 331,35 ms; mínimo de 1.538,00 ms; p50 de 1.778,00 ms; máximo de 2.461,00 ms). A entrega foi restabelecida na 3ª tentativa em 100% dos casos, sem intervenção humana.

Na rotação de chaves, um experimento com 50 requisições contínuas em trânsito e rotação disparada na 25ª requisição mostrou que o protocolo webhook-first commit notificou primeiro o endpoint do SDK, com webhook assinado por HMAC-SHA256. A verificação da assinatura via Web Crypto API, com comparação em tempo constante, teve tempo médio de 0,13 ms (p50 de 0,10 ms; p95 de 0,25 ms; N = 100). O ciclo completo (geração da chave Argon2id, despacho do webhook, atualização em memória pelo MemoryAdapter e persistência do novo hash no PostgreSQL) concluiu-se em 69,45 ms, com 50/50 respostas 201 Created e nenhum erro 401 Unauthorized durante a transição.

7. Conclusão

7.1. Considerações Finais

Sob a metodologia DSR, o Hermes entregou um artefato funcional composto por API e Workers assíncronos (hermes-api), console em Next.js (hermes-front) e SDK em TypeScript (hermes-client). Nos testes em ambiente conteinerizado, o desacoplamento por filas Redis/BullMQ reduziu a latência percebida pelo cliente de 310,41 ms (no disparo SMTP síncrono com RTT externo simulado) para 64,05 ms (p95 de 69,31 ms) em envios unitários, e para 1,39 ms por mensagem em lote (717,33 e-mails/s), com retorno imediato de 201 Created. No processamento assíncrono, o cache da função pré-compilada do Handlebars reduziu em 52,3% o tempo de renderização, e o escalonamento de 1 para 4 workers diminuiu a drenagem de 500 e-mails de 6,81 s (73,46 msg/s) para 3,03 s (164,79 msg/s), speedup de 2,25x, com 100% de entrega auditada no Mailpit. O SDK respeitou as janelas teóricas de backoff (médias de 1.043,50 ms e 1.878,60 ms na 1ª e 2ª retentativas), e a rotação de chaves por webhook-first commit foi concluída em 69,45 ms (verificação HMAC em 0,13 ms), sem erros em 50 requisições em trânsito. Ressalvadas as limitações da Seção 6.2, o Hermes consolida-se como uma solução resiliente e escalável para comunicação transacional.

7.2. Limitações Arquiteturais e Discussão de Trade-offs

Em conformidade com a DSR (Peffers et al., 2007; Wieringa, 2014), registram-se as limitações e os trade-offs da versão atual:

1. **Gravação dupla (dual-write) e persistência do Redis.** A gravação do e-mail como pending no PostgreSQL e o enfileiramento no Redis/BullMQ (emailService.ts) ocorrem em duas operações sem coordenação transacional. Se o Redis ficar indisponível logo após o commit, o registro permanece órfão. Além disso, o Redis usa snapshots RDB, sem o modo Append-Only File (AOF com fsync a cada segundo), o que pode perder jobs recém-enfileirados em um desligamento abrupto. A solução fundamenta-se no padrão Transactional Outbox (Curry, 2004), gravando o evento na mesma transação ACID do PostgreSQL com leitura assíncrona por um processo relé (poller/CDC), mais a ativação do AOF.
2. **Entrega at-least-once e idempotência na ingestão.** O worker adota semântica de entrega at-least-once com guarda de idempotência no consumo, e a rotação usa chaves determinísticas diárias, mas o endpoint POST /emails ainda não exige o cabeçalho Idempotency-Key. Se a resposta 201 for interrompida após a gravação, a retentativa do SDK (withRetry) criará um segundo registro e um envio duplicado.
3. **Custo do Argon2id na autenticação.** O Argon2id defende contra a exfiltração do banco, e o prefixo indexado mitiga a busca. Porém, como as chaves têm alta entropia (256 bits, em apiKeyGenerate.ts), a execução a cada requisição adiciona dezenas de milissegundos de CPU (cerca de 58 ms), que um resumo rápido (HMAC-SHA256) ou um cache de curta duração de chaves recém-verificadas evitaria. Esse custo define o teto de ~25 req/s do Cenário B.
4. **Transpilação MJML no worker a cada envio.** A injeção de variáveis pelo Handlebars precede o mjml2html() para permitir expressões dinâmicas em

atributos MJML, de modo que a transpilação ocorre no worker a cada e-mail. Para templates sem condicionais sobre tags MJML, a pré-compilação para HTML estático no cadastro (mantendo os placeholders do Handlebars) eliminaria esse custo, como indica o ganho de 8,32x da Seção 5.2.

5. **Transporte de webhooks e DNS rebinding.** O cabeçalho X-Hermes-Signature garante integridade e origem, mas a newApiKey trafega no corpo JSON, o que torna indispensável impor https:// na validação Zod das URLs de webhook. Contra SSRF por DNS rebinding (domínio que resolve primeiro para IP público e depois para loopback ou faixas RFC 1918), a verificação do IP deve ocorrer no evento de lookup do socket TCP (http.Agent/https.Agent customizado) no momento da conexão.
6. **Escopo da avaliação.** Todos os componentes e o gerador de carga compartilharam uma única estação (WSL2, 12 vCPUs, 7,6 GiB), e o destino de envio foi um Mailpit local, com o RTT externo simulado em +240 ms. O baseline síncrono foi medido apenas com 1 VU, o speedup de 2,25x com 4 workers é sublinear e sua causa não foi analisada, e a rotação foi testada em uma única instância com 50 requisições sequenciais. Não foram realizados testes de segurança ofensivos (XSS, autorização, SSRF), apenas a verificação do limitador de taxa e da rotação de chaves.

7.3. Trabalhos Futuros

1. **Orquestração multi-nó e consistência transacional.** Evoluir o hermes_scaler (Docker Compose em nó único) para Docker Swarm ou Kubernetes, distribuindo réplicas do worker entre servidores. Implementar o Transactional Outbox com Idempotency-Key na ingestão, a pré-compilação estática opcional de MJML e um agente HTTP customizado que bloqueie IPs privados no nível do socket.
2. **Observabilidade.** Instrumentar o ecossistema com OpenTelemetry, Prometheus e Grafana, para rastreamento distribuído de ponta a ponta, dashboards da ocupação das filas e dos recursos dos workers e telemetria das taxas de sucesso e falha SMTP.
3. **Interface gRPC.** Incorporar gRPC/Protocol Buffers na hermes-api como alternativa de alto desempenho ao HTTP/REST, otimizando a serialização e a latência entre microsserviços parceiros.
4. **Entregabilidade e reputação de IP.** Desenvolver um módulo analítico de bounces e feedback loops, com throttling adaptativo para preservar a reputação do domínio e do IP dos servidores SMTP emissores.

REFERÊNCIAS

ADRIO, K. et al. **Comparative analysis of monolith, microservice api gateway and microservice federated gateway on web-based application using graphql api**. In: INTERNATIONAL CONFERENCE ON ELECTRICAL ENGINEERING, COMPUTER SCIENCE AND INFORMATICS (EECSI), 10., 2023, Palembang.Proceedings [...]. IEEE, 2023. p. 654-660.

AMAZON WEB SERVICES. **Amazon Simple Email Service (SES): desenvolvedor guia**. Seattle: AWS, 2024. Disponível em: https://aws.amazon.com/ses/.

BIRYUKOV, Alex et al. **Argon2: the memory-hard function for password hashing and other applications**. RFC 9106. Sep. 2021. Disponível em: https://www.rfc-editor.org/rfc/rfc9106.

BLINOWSKI, Grzegorz; OJDOWSKA, Anna; PRZYBYŁEK, Adam. **Monolithic vs. microservice architecture: a performance and scalability evaluation**. IEEE Access, v. 10, p. 20357-20374, 2022. DOI: 10.1109/ACCESS.2022.3152803.

BULLMQ. **BullMQ: message queue and batch processing for NodeJS and TypeScript based on Redis**. 2024. Disponível em: https://bullmq.io.

CAVALCANTE, João Phelype da Silva. **Conteinerização e gerenciamento de aplicações industriais de uma arquitetura orientada a microsserviços para a Indústria 4.0**. 2023. 55 f. Trabalho de Conclusão de Curso (Graduação em Engenharia de Controle e Automação) - Instituto de Ciência e Tecnologia de Sorocaba, Universidade Estadual Paulista "Júlio de Mesquita Filho", Sorocaba, 2023.

COSTA, Marcos Vinicius Lima da; QUEIROZ, Paulo Gabriel Gadelha. **Um estudo de caso sobre desenvolvimento, fila de mensagens e orquestração de microsserviços em uma linha de produtos de softwares de clínicas médicas**. 2022. 16 f. Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Universidade Federal Rural do Semi-Árido, Mossoró, 2022.

CURRY, Edward. **Message-oriented middleware**. Middleware for communications, Chichester: John Wiley & Sons, p. 1-28, 2004.

DOCKER DOCS. **Docker overview: understand the Docker architecture and main concepts**. San Francisco: Docker Inc., 2021. Disponível em: https://docs.docker.com/get-started/overview.

DWORKIN, Morris. **Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC**. NIST Special Publication 800-38D. Gaithersburg: National Institute of Standards and Technology, 2007. DOI: https://doi.org/10.6028/NIST.SP.800-38D.

FOWLER, Martin; LEWIS, James. **Microservices: a definition of this new architectural term**. Martinfowler.com, 25 mar. 2014. Disponível em: https://martinfowler.com/articles/microservices.html.

GAMMA, Erich et al. **Design Patterns: elements of reusable object-oriented software**. Reading: Addison-Wesley, 1994.

INSTITUTO METRÓPOLE DIGITAL. **Redes de Computadores II: Aula 12 - Correio Eletrônico - Parte II**. Versão 5.0. Natal: IMD/UFRN, [s.d.].

KLENSIN, John C. **Simple Mail Transfer Protocol**. RFC 5321. October 2008. Disponível em: https://www.rfc-editor.org/rfc/rfc5321.

KRAWCZYK, Hugo et al. **HMAC: keyed-hashing for message authentication**. RFC 2104. Feb. 1997. Disponível em: https://www.rfc-editor.org/rfc/rfc2104.

LIMA, William Bruno Sales de Paula. **Um estudo comparativo entre arquiteturas de software para aplicações Web**. 2025. 89 f. Trabalho de Conclusão de Curso (Bacharelado em Engenharia de Computação) - Universidade Federal do Ceará, Sobral, 2025.

LONGIN, Timo. **SMTP Smuggling: spoofing e-mails worldwide**. Vienna: SEC Consult Vulnerability Lab, 2023. Disponível em: https://sec-consult.com/blog/detail/smtp-smuggling-spoofing-e-mails-worldwide.

MJML. **MJML: the only framework that makes responsive email easy**. 2024. Disponível em: https://mjml.io.

NOVU. **Novu: the open-source notification infrastructure**. 2024. Disponível em: https://novu.co.

OWASP FOUNDATION. **OWASP Top 10 API Security Risks - 2023**. OWASP, 2023. Disponível em: https://owasp.org/API-Security/editions/2023/en/0x11-t10/.

PAIVA, Gabriel R. C.; SANTOS, Paulo C. dos. **Arquitetura de Software: microsserviços, escalabilidade e práticas eficientes na comunicação entre sistemas**. Muzambinho: IFSULDEMINAS, 2025.

PEFFERS, Ken et al. **A Design Science Research Methodology for Information Systems Research**. Journal of Management Information Systems, v. 24, n. 3, p. 45-77, 2007.

PICIOLI, Matheus Augusto; DEZANI, Henrique; CARLSON FILHO, Carlos Magnus. **Arquitetura microsserviços voltada para escalabilidade**. RIC-CPS, Taquaritinga, 2022. Disponível em: https://ric.cps.sp.gov.br/handle/123456789/24878.

POSTAL. **Postal: a fully featured open source mail delivery platform**. 2024. Disponível em: https://postalserver.io.

POSTGRESQL DEVELOPMENT GROUP. **PostgreSQL: the world's most advanced open-source relational database**. 2022. Disponível em: https://www.postgresql.org.

PRESSMAN, Roger S.; MAXIM, Bruce R. **Engenharia de software: uma abordagem profissional**. 8. ed. Porto Alegre: AMGH, 2016.

RAZZAQ, Abdul; GHAYYUR, Shahbaz A. K. **A systematic mapping study: the new age of software architecture from monolithic to microservice architecture-awareness and challenges**. Computer Applications in Engineering Education, v. 31, n. 2, p. 421-451, 2023. DOI: 10.1002/cae.22586.

REDIS. **Getting started with Redis**. 2023. Disponível em: https://redis.io/docs/getting-started.

RESEND. **Resend: email for developers**. 2024. Disponível em: https://resend.com.

SENDGRID. **SendGrid Email API: cloud-based email delivery platform**. Denver: Twilio SendGrid, 2024. Disponível em: https://sendgrid.com.

VALENTE, Marco Tulio. **Engenharia de Software Moderna**. [S. l.]: Independente, 2022.

VERCEL. **Next.js Documentation: App Router and React Server Components**. San Francisco: Vercel, 2026. Disponível em: https://nextjs.org/docs.

WANG, Chuhan et al. **Email spoofing with SMTP smuggling: how the shared email infrastructures magnify this vulnerability**. In: USENIX Security Symposium, 34., 2025, Seattle. Proceedings [...]. Seattle: USENIX Association, 2025. p. 1-18. Disponível em: https://www.usenix.org/conference/usenixsecurity25/presentation/wang-chuhan.

WIERINGA, Roel J. **Design Science Methodology for Information Systems and Software Engineering**. [S. l.]: Springer, 2014. DOI: https://doi.org/10.1007/978-3-662-43839-8.