ValorizeAI: Documentação e Validação de uma Arquitetura Serverless Elasticamente Gerenciada

Title: ValorizeAI: Documenting and Validating a Managed Serverless Architecture

Felipe Tomkiel Malacarne, Prof. Me. Marcos André Lucas

[1] Universidade Regional Integrada do Alto Uruguai e das Missões
Departamento de Engenharias e Ciência da Computação
Caixa Postal 743 – 99.709-910 – Erechim – RS – Brasil

101090@uricer.edu.br, mlucas@uricer.edu.br

Resumo. *Plataformas financeiras modernas exigem arquiteturas elásticas capazes de absorver picos de carga transacional sem comprometer consistência forte ou observabilidade. Este trabalho documenta e valida o ValorizeAI, uma aplicação real de categorização automática construída sobre Google Cloud Run, Cloud SQL, Redis (Memorystore), Cloud Tasks e WebSockets. A pesquisa adota uma abordagem aplicada inspirada em SRE: os SLOs (latência P95 $\leq$ 300 ms, erro $<$ 0,5%, disponibilidade $\geq$ 99,5%) são definidos previamente, a infraestrutura é provisionada como código e os testes k6 abrangem cenários de leitura intensiva, leitura/escrita e o pipeline assíncrono. O cenário de leitura sustentou 1,000 usuários virtuais (média de 470 req/s, pico de $\approx$970 req/s) com P95 de 158 ms; o cenário misto manteve 226 req/s com 650 VUs e violou o SLO a partir de $\approx$539 VUs, elevando o P95 para 658 ms e o p99 para 2,67 s ao saturar a cota de 10 instâncias do Cloud Run. O pipeline assíncrono processou 51,58 mil tarefas em 10 minutos (86 tarefas/s) sem perdas. Conclui-se que a arquitetura atende confortavelmente workloads intensivos em leitura e que o suporte a escritas altamente concorrentes requer aumento de instâncias HTTP ou otimizações específicas no caminho de escrita.*
Palavras-Chave *Computação Serverless, Cloud Run, Plataformas Financeiras, Testes de Carga, Observabilidade.*

Abstract. *Modern financial workloads demand elastic architectures capable of sustaining variable traffic while preserving strong consistency and system observability. This paper documents and validates ValorizeAI, a real-world automated categorization platform built on Cloud Run, Cloud SQL, Redis, Cloud Tasks, and WebSockets. The study adopts an applied SRE-inspired methodology: SLOs (P95 latency $\leq$ 300 ms, error rate $<$ 0.5%, availability $\geq$ 99.5%) are defined beforehand, infrastructure is provisioned as code, and k6 executes read-only, mixed read/write, and asynchronous-pipeline load scenarios. The read scenario supported 1,000 virtual users (mean 470 req/s, peak $\approx$970 req/s) with P95 = 158 ms and zero failures; the mixed scenario sustained 226 req/s at 650 VUs and violated the SLO around $\approx$539 VUs, reaching P95 = 658 ms and p99 = 2.67 s once the 10-instance Cloud Run quota was saturated. The asynchronous pipeline processed 51.58k Cloud Tasks in 10 minutes (86 tasks/s) with no loss or duplication. The results show that the architecture comfortably supports read-heavy traffic, while higher write concurrency requires additional horizontal scaling or optimizations in the write path.*
Keywords *Serverless Computing, Cloud Run, Financial Platforms, Performance Testing, Observability.*

1. Introdução

Aplicações digitais modernas — incluindo plataformas de e-commerce, serviços financeiros, mídias sociais e sistemas colaborativos em tempo real — enfrentam um desafio operacional comum: a gestão de cargas de trabalho voláteis e imprevisíveis [Google Cloud 2024]. Picos de tráfego, ingestão contínua de dados e múltiplas integrações com sistemas externos exigem uma infraestrutura capaz de reagir dinamicamente, superando as limitações de provisionamento manual ou pipelines monolíticos e rigidamente acoplados.

A resposta predominante da indústria é a *elasticidade na nuvem* (*cloud elasticity*), definida como a capacidade de alocar e desalocar recursos de forma automática e autônoma conforme a demanda varia em tempo real [Google Cloud 2024]. Diferentemente da escalabilidade tradicional, que geralmente requer intervenção humana para expandir a infraestrutura, a elasticidade é projetada para lidar com picos abruptos e quedas de tráfego, garantindo simultaneamente desempenho e eficiência de custos. Seus benefícios vão além da alocação dinâmica: evitam-se desperdícios por superprovisionamento e assegura-se a responsividade da aplicação sob condições extremas.

Entretanto, arquiteturas que habilitam essa elasticidade — como microsserviços, contêineres e paradigmas *serverless* [Basteri 2023] — introduzem complexidade operacional significativa. Sistemas distribuídos são naturalmente mais difíceis de depurar, monitorar e validar. Nesse contexto, a *observabilidade* tornou-se um pilar estratégico [Basteri 2023]. Diferente do monitoramento tradicional, que rastreia falhas previamente conhecidas, a observabilidade fornece meios para inferir o estado interno do sistema a partir de logs, métricas e *traces*, permitindo detectar e diagnosticar comportamentos inesperados.

Elasticidade e observabilidade formam um ciclo de *feedback* essencial: arquiteturas elásticas geram comportamentos dinâmicos que só podem ser compreendidos por meio de dados observáveis; ao mesmo tempo, SLIs provenientes desses dados são utilizados para validar ou acionar mecanismos automáticos de escalonamento, garantindo que os SLOs sejam atendidos sem custos excessivos. Esse ciclo opera em escalas de tempo muito menores do que processos manuais conseguem acompanhar.

O ValorizeAI, objeto deste estudo, surge como um caso completo para investigar essa relação. Trata-se de uma aplicação web modular que integra ingestão de dados, processamento síncrono e assíncrono e comunicação em tempo real, utilizando uma *stack* moderna baseada em Cloud Run, Cloud SQL, Redis (Memorystore), Cloud Tasks, WebSockets e serviços auxiliares.

1.1. Justificativa e Problema de Pesquisa

Workloads transacionais que envolvem ingestão intensa de dados, estados compartilhados e interfaces colaborativas — como os simulados pelo ValorizeAI — impõem requisitos rigorosos: consistência forte, rastreabilidade para auditoria e respostas de baixa latência mesmo sob variações abruptas de tráfego. Para atender a esses requisitos, arquiteturas modernas combinam componentes especializados, como CDNs e balanceadores globais, filas assíncronas orientadas a eventos, cache distribuído e comunicação persistente via WebSockets [Barri 2025, Confluent 2024, Yadav 2019, Fernando e Engel 2025].

Embora existam estudos pontuais sobre esses componentes, a literatura apresenta lacunas quanto à validação integrada de arquiteturas híbridas (CaaS + filas + WebSockets) em cenários reprodutíveis de carga [Christidis et al. 2022, Abad et al. 2021]. Trabalhos existentes tendem a focar em comparações de ferramentas de IaC [Pessa 2023] ou no desempenho de microsserviços isolados [Hebbar 2025], raramente considerando o comportamento do sistema completo.

Este trabalho busca preencher essa lacuna ao documentar a arquitetura do ValorizeAI e validar seu comportamento sob estresse de carga frente a SLOs definidos. A questão central investigada é: *Como uma arquitetura híbrida e elástica — composta por Cloud Run, Redis, Cloud SQL, Cloud Tasks e WebSockets dedicados — se comporta sob condições intensas de carga, e como esse comportamento pode ser validado de forma reprodutível?*

1.2. Objetivo Geral

Demonstrar, por meio de documentação técnica e experimentos de desempenho, que a arquitetura do ValorizeAI — composta por CDN, contêineres escalados horizontalmente, processamento assíncrono em filas, servidor de WebSockets, armazenamento de artefatos em *buckets* e cache distribuído em Redis — sustenta os SLOs definidos para um produto transacional completo, mantendo código, infraestrutura e observabilidade versionados em repositório.

1.3. Objetivos Específicos

1. Mapear a arquitetura *end-to-end*, destacando o papel do balanceador/CDN, instâncias de contêineres, servidor WebSockets, filas assíncronas, *buckets* e Redis.
2. Documentar o desenvolvimento dos módulos críticos do sistema, incluindo ingestão de dados, automações, notificações e painéis em tempo real.
3. Planejar e executar testes de carga (k6, cenários de leitura e leitura/escrita) e testes assíncronos, validando a arquitetura frente aos SLOs definidos.
4. Interpretar os resultados e propor otimizações relacionadas a desempenho, elasticidade e custo.

1.4. Contribuições Tangíveis

1. Arquitetura documentada e replicável.
2. Infraestrutura reprodutível (Terraform, Docker, Makefile).
3. Cenários de desempenho registrados e transparentes (k6).
4. Validação do pipeline assíncrono baseado em Cloud Tasks.

Em conjunto, essas contribuições formam um pacote completo de replicação — código, automações e experimentos — para avaliações futuras de workloads transacionais em ambientes CaaS.

O restante deste artigo está organizado da seguinte forma: a Seção 2 apresenta os trabalhos relacionados; a Seção 3 discute a fundamentação teórica; a Seção 4 descreve a metodologia experimental; a Seção 5 detalha a implementação do ValorizeAI; a Seção 6 consolida os resultados e análises; e, por fim, a Seção 7 apresenta as conclusões e trabalhos futuros.

2. Trabalhos Relacionados

A arquitetura investigada neste trabalho situa-se na interseção de três eixos centrais da literatura em sistemas distribuídos: (i) paradigmas de execução em nuvem, (ii) padrões arquiteturais para desempenho e resiliência e (iii) metodologias de validação empírica. Esta seção revisa o estado da arte nesses eixos para posicionar a contribuição do ValorizeAI e evidenciar a lacuna identificada na seção de Introdução.

2.1. Paradigmas de Execução: Serverless (FaaS) e Contêineres Gerenciados (CaaS)

A escolha do paradigma de execução é um dos fatores determinantes para sistemas elásticos modernos. A literatura recente compara amplamente *Functions-as-a-Service* (FaaS) e *Containers-as-a-Service* (CaaS). O FaaS, amplamente representado por AWS Lambda e Google Cloud Functions, oferece escalonamento automático e faturamento por execução, mas apresenta limitações para aplicações *stateful* ou de longa duração devido ao *cold start*, ao isolamento elevado e à efemeridade das instâncias [Sonawane et al. 2024, Datadog 2024]. Esses aspectos o tornam inadequado para componentes persistentes como servidores WebSocket.

Por outro lado, o CaaS — como Google Cloud Run ou AWS Fargate — mantém a elasticidade do FaaS, mas preserva o controle sobre o ambiente do contêiner e comporta processos contínuos [Lloyd et al. 2018]. Essa característica é essencial para o servidor de WebSockets do ValorizeAI (Laravel Reverb), que requer conexões persistentes e compartilhamento de estado via Redis. Assim, a literatura sustenta a escolha do CaaS como paradigma mais adequado para arquiteturas híbridas que combinam serviços *stateless* e componentes de longa duração.

2.2. Padrões Arquiteturais para Desempenho e Resiliência

Sistemas transacionais modernos combinam padrões síncronos e assíncronos para garantir responsividade, disponibilidade e escalabilidade.

2.2.1. Arquiteturas Orientadas a Eventos (EDA) e Filas Assíncronas

Arquiteturas orientadas a eventos promovem desacoplamento, resiliência e absorção de picos de carga ao delegar tarefas para filas assíncronas [Confluent 2024]. Estudos comparativos analisam o comportamento de diferentes *brokers* — como RabbitMQ, Apache Kafka e Pulsar — frente a cenários com variação de tamanho de mensagens e taxas de publicação [Thepphakan 2025]. Esses trabalhos demonstram que a escolha da fila deve refletir o perfil do *workload*: baixa latência para eventos pequenos ou alto *throughput* contínuo para fluxos intensivos.

A arquitetura do ValorizeAI segue essa abordagem ao integrar Cloud Tasks para desacoplar operações pesadas do ciclo de requisição HTTP, garantindo responsividade mesmo sob carga.

2.2.2. Comunicação em Tempo Real e Cache Distribuído

Para requisitos de tempo real, a literatura destaca o uso combinado de WebSockets e cache distribuído. Redis é amplamente utilizado como memória compartilhada de baixa latência e como mecanismo Pub/Sub para orquestrar a entrega de eventos entre múltiplas instâncias [Yadav 2019]. Em ambientes elásticos — como Cloud Run — o Redis atua como *backplane* que mantém a consistência entre instâncias efêmeras ao distribuir mensagens para clientes conectados.

Estudos recentes analisam o impacto desse modelo em métricas de *throughput* e latência RTT, validando suas vantagens para sistemas colaborativos e dashboards em tempo real [Fernando e Engel 2025, Twine 2022]. Esse padrão é consistente com o design do ValorizeAI, cujo servidor WebSocket persistente publica e assina eventos via Redis para garantir consistência e escalabilidade horizontal.

2.3. Metodologias de Validação Empírica

Validar arquiteturas distribuídas exige metodologias reprodutíveis e alinhadas a objetivos de serviço.

2.3.1. Infraestrutura como Código (IaC)

IaC é amplamente adotado para garantir reprodutibilidade e eliminar variações de ambiente em experimentos de desempenho [Pessa 2023]. Estudos analisam ferramentas como Terraform e AWS CDK em termos de eficiência, expressividade e redução de deriva de configuração [Guerriero et al. 2019]. Entretanto, tais estudos frequentemente focam na ferramenta, e não no sistema provisionado — o que contrasta com este trabalho, no qual IaC é utilizado como base para experimentos e revalidações sucessivas do ambiente completo (CaaS, filas, Redis e banco de dados).

2.3.2. Validação por SLOs e Testes de Carga

A Engenharia de Confiabilidade de Sites (SRE) recomenda validação orientada a SLOs para medir sucesso operacional [McCoy e Forsgren 2020]. A literatura recente adota ferramentas modernas como o k6 para simular perfis realistas de carga e avaliar latência, erro e saturação [Cervone 2024]. O trabalho de Hebbar [Hebbar 2025], por exemplo, utiliza k6 para validar priorização de tráfego em APIs reativas, monitorando percentis de latência e comportamento sob estresse.

A estratégia utilizada pelo ValorizeAI — definição de SLOs, instrumentação do sistema e execução de cenários de carga reprodutíveis — encontra suporte direto nesses estudos, reforçando a adequação da metodologia adotada.

2.4. Síntese e Lacuna de Pesquisa

A literatura revisada é rica, mas fragmentada: há estudos sobre FaaS vs. CaaS [Lloyd et al. 2018], comparações de *brokers* de EDA [Thepphakan 2025], análises de ferramentas de IaC [Pessa 2023] e validações de desempenho específicas usando k6 e SLOs [Hebbar 2025]. Contudo, conforme discutido por Abad et al. [Abad et al. 2021], falta uma análise integrada de arquiteturas híbridas que combinem todos esses elementos em um único sistema reprodutível.

A contribuição do ValorizeAI está exatamente nessa integração: uma arquitetura CaaS orientada a eventos com WebSockets persistentes, cache Redis e filas assíncronas, provisionada integralmente via IaC e validada com metodologia alinhada ao estado da arte. Os estudos analisados servem como blocos isolados; este trabalho, por sua vez, propõe uma validação *end-to-end* que abrange paradigma, padrões arquiteturais e metodologia experimental.

3. Fundamentação Teórica

Esta seção apresenta os conceitos que sustentam o design, a implementação e a validação empírica do ValorizeAI. São abordados princípios de design de software, fundamentos arquiteturais dos componentes utilizados e noções essenciais de Engenharia de Confiabilidade (SRE), que orientam a formulação dos SLOs e os experimentos descritos na metodologia.

3.1. Princípios de Design de Software

O desenvolvimento do ValorizeAI segue práticas consolidadas de engenharia de software, que visam reduzir acoplamento, aumentar coesão e facilitar evolução incremental da aplicação.

3.1.1. Clean Architecture

A *Clean Architecture*, proposta por Martin [Martin 2017], defende a separação de responsabilidades através da *Regra da Dependência*: detalhes variáveis, como frameworks e tecnologias de persistência, devem depender das regras de negócio, e não o contrário. No ValorizeAI, essa diretriz se manifesta na separação explícita entre lógica de domínio (Actions e Queries) e elementos de infraestrutura (controladores, Reverb, Redis e Cloud SQL).

3.1.2. Domain-Driven Design (DDD)

O *Domain-Driven Design* (DDD), de Evans [Evans 2003], fornece estrutura semântica para lidar com domínios complexos. Os principais elementos aplicados são:

- **Linguagem Ubíqua:** Modelo conceitual usado consistentemente por desenvolvedores e especialistas de domínio.
- **Contextos Delimitados:** Fronteiras que evitam ambiguidade e definem onde cada modelo é válido.
- **Agregados:** Unidades atômicas de consistência, manipuladas exclusivamente via sua *Aggregate Root*.

Esses conceitos organizam o domínio do ValorizeAI e estruturam a modelagem de transações, contas e categorias.

3.1.3. Segregação de Responsabilidades (CQRS e DTOs)

Para otimizar caminhos de leitura e escrita, o sistema adota *Command Query Responsibility Segregation* (CQRS) [Fowler 2011]. Consultas são tratadas por *Queries* especializadas, que exploram Redis para acesso rápido; enquanto escritas passam por *Actions*, que encapsulam regras de negócio, validações e persistência.

Complementarmente, são utilizados *Data Transfer Objects* (DTOs) [Fowler 2002], que isolam os dados expostos pela API dos modelos internos, reduzindo acoplamento e controlando a estrutura retornada ao front-end.

3.2. Arquitetura e Componentes da Aplicação

A arquitetura do ValorizeAI combina serviços gerenciados que oferecem elasticidade, isolamento, baixo acoplamento e capacidade de processamento assíncrono.

3.2.1. Google Cloud Run e Cloud Tasks

O Google Cloud Run é uma plataforma CaaS que executa contêineres *stateless* com escalonamento automático, incluindo *scale-to-zero* [Google Cloud 2024]. No ValorizeAI, duas classes de contêineres são executadas: o serviço HTTP principal e o servidor WebSocket.

O Cloud Tasks implementa o componente assíncrono de EDA, enfileirando tarefas que exigem maior tempo de processamento. Isso permite que a API principal permaneça responsiva mesmo durante operações intensivas, delegando trabalho para *workers* dedicados.

3.2.2. Laravel Reverb (WebSockets)

O Laravel Reverb [Laravel Holdings Inc. 2025] fornece um canal WebSocket de alta performance, apoiado no protocolo Pusher. O recurso central para este trabalho é sua capacidade de operar em múltiplas instâncias. Para permitir comunicação consistente entre esses nós efêmeros do Cloud Run, o Reverb utiliza Redis como *backplane*, garantindo que eventos emitidos por qualquer instância sejam entregues aos clientes conectados em outra.

3.2.3. Redis (Cache e Pub/Sub)

O Redis, um armazenamento em memória de baixa latência [Kleppmann 2017], exerce dois papéis complementares:

1. **Cache de leitura:** Acelera consultas críticas, reduzindo carga sobre o Cloud SQL.
2. **Pub/Sub:** Atua como barramento de mensagens para sincronizar múltiplas instâncias do Reverb.

Essas funções são essenciais para garantir que operações de leitura permaneçam dentro dos SLOs e que notificações em tempo real sejam entregues sem violação de consistência.

3.3. Engenharia de Confiabilidade de Sites (SRE)

A metodologia de validação do ValorizeAI segue os princípios de SRE apresentados pelo Google [Beyer et al. 2016]. O SRE trata confiabilidade como uma disciplina quantitativa, guiada por indicadores e metas formalizadas.

3.3.1. SLIs, SLOs e Orçamento de Erro

Três conceitos embasam os experimentos:

- **SLI (Service Level Indicator):** Métrica quantitativa que representa a qualidade de uma operação, como latência P95, taxa de erro ou *throughput* [McCoy e Forsgren 2020].
- **SLO (Service Level Objective):** Meta operacional definida para um SLI. Ex.: “95% das leituras com latência <300 ms”.
- **Orçamento de Erro:** Tolerância máxima para falhas dentro do período do SLO, definindo quando priorizar evolução ou estabilidade.

Esses conceitos orientam os testes de carga apresentados na Seção 4, guiando a análise de saturação, violações de SLO e comportamento do sistema sob tráfego intenso.

4. Metodologia

A metodologia adotada neste trabalho é aplicada, experimental e orientada a reprodutibilidade. Todas as configurações de infraestrutura, código e instrumentação foram versionadas no repositório do ValorizeAI, permitindo que os experimentos de carga sejam reproduzidos de forma determinística.

4.1. Tipo de Pesquisa e Estratégia Geral

O estudo caracteriza-se como uma **pesquisa aplicada** conduzida sob a forma de **estudo de caso** de uma aplicação real em operação. A estratégia seguiu quatro fases iterativas:

1. **Planejamento:** definição dos SLOs (latência P95 $\leq$ 300 ms, taxa de erro $<$ 0,5%, disponibilidade $\geq$ 99,5%). Consideraram-se também as cotas do Cloud Run (10 instâncias simultâneas de 1 vCPU/1 GiB), que estabeleceram o limite teórico de throughput.
2. **Preparação do ambiente:** módulos Terraform provisionaram a infraestrutura (VPC, Cloud Run, Cloud SQL, Redis, Cloud Tasks e balanceador). O Docker Compose reproduziu localmente PostgreSQL, Redis e ferramentas de observabilidade. O Makefile encapsulou rotinas de lint, build e execução dos cenários k6.
3. **Execução controlada:** os cenários k6 (leitura intensiva e leitura/escrita) foram executados contra o domínio público da API, enquanto o pipeline assíncrono processava um lote adicional de tarefas enviadas ao Cloud Tasks.
4. **Coleta e análise:** métricas de latência, throughput e taxa de erro foram extraídas dos CSVs do k6 e correlacionadas com séries temporais do Cloud Monitoring (CPU, memória, backlog de tarefas). O comportamento do pipeline assíncrono foi documentado qualitativamente e quantitativamente.

4.2. Arquitetura do Ambiente Experimental

A Figura 1 sintetiza o ambiente utilizado nos experimentos. O tráfego HTTP/HTTPS é roteado pelo **Cloud Load Balancer** com **Cloud CDN**, que distribui *assets* estáticos e aplica políticas de segurança (WAF). A arquitetura executa três serviços Cloud Run:

- **API Laravel:** responsável pelos endpoints REST exercitados nos testes.
- **Laravel Reverb:** servidor WebSocket dedicado, escalado independentemente.
- **Workers HTTP:** consumidores assíncronos acionados pelo Cloud Tasks.

O Redis (Memorystore) atua como *cache* para consultas críticas e como *backplane* Pub/Sub para a sincronização do Reverb. O banco transacional permanece no **Cloud SQL for PostgreSQL**, configurado como instância de 2 vCPU e 16 GiB de RAM para refletir o ambiente produtivo. O **Cloud Tasks** executa o pipeline assíncrono por meio de requisições *push*, permitindo escalabilidade automática dos workers. O **Cloud Storage** integra o sistema, mas não foi exercitado nos testes.

4.3. Desenvolvimento da Aplicação e Modelo de Dados

O backend Laravel concentra o domínio financeiro e expõe APIs consumidas pelo front-end em React. Os workers HTTP processam tarefas intensivas, como importações de extratos, geração de relatórios e envio de notificações. O modelo relacional multi-inquilino, mostrado na Figura 2, organiza usuários, contas, transações, categorias, orçamentos e embeddings vetoriais, permitindo cenários realistas durante os testes.

4.4. Ferramentas e Processo de Preparação

A reprodutibilidade foi garantida por três pilares:

- **Infraestrutura como Código (IaC):** cada componente (Cloud SQL, Redis, Cloud Run, Tasks e balanceador) possui módulo Terraform independente. Outputs conectam automaticamente credenciais, redes e endpoints entre serviços, evitando divergências.

- **Ambientes determinísticos:** Docker Compose e Makefile garantem que a aplicação seja testada sempre com a mesma versão do stack (PHP 8.4, PostgreSQL, Redis), evitando variabilidade entre execuções.
- **Observabilidade:** métricas de CPU, fila, memória e latência foram coletadas no Cloud Monitoring. Logs e traces complementaram a identificação dos pontos de saturação.

4.5. Planejamento dos SLOs e Desenho dos Cenários

Com base na literatura de SRE [McCoy e Forsgren 2020, Beyer et al. 2016], adotaram-se três metas: latência P95 $\leq$ 300 ms, erro $<$0,5% e disponibilidade $\geq$99,5%. As cotas do Cloud Run definiram o limite físico dos experimentos (10 instâncias de 1 vCPU). Os testes foram projetados para atingir e analisar o comportamento nessa zona de saturação.

Foram definidos três experimentos:

1. **Cenário de leitura intensiva:** 1.000 VUs acessando o endpoint `GET /api/transactions` durante 17 minutos. O roteiro alternou contas, filtros e um *think time* de 1 s para simular interação real.
2. **Cenário misto leitura/escrita:** 650 VUs alternando entre consultas e criação de transações (65% leitura, 20% escrita, 15% contas). Esse fluxo replica a proporção observada nos logs reais.
3. **Teste assíncrono:** publicação de 51.580 tarefas no Cloud Tasks, monitorando o tempo de drenagem e o pico de instâncias dos workers.

Os cenários foram estruturados com ramp-ups progressivos (150 $\rightarrow$ 1.000 VUs) para aquecer o cache e observar o comportamento nas proximidades do teto de instâncias.

4.6. Execução dos Experimentos

Cada rodada seguiu o protocolo:

1. **Preparação dos dados:** povoamento do PostgreSQL com seeds e factories, e pré-aquecimento do Redis.
2. **Disparo dos cenários k6:** execução automatizada via Makefile, registrando SLIs e eventos relevantes.
3. **Coleta automática:** exportação dos resultados do k6 para CSV (latência, erros, uso de VUs).
4. **Teste de filas:** disparo massivo de tarefas e monitoramento do backlog no Cloud Tasks.

4.7. Coleta e Integração das Evidências

As evidências utilizadas na análise incluem:

- **Tabelas e séries de latência/throughput** extraídas do k6;
- **Métricas de infraestrutura** (CPU, instâncias ativas, backlog da fila);
- **Logs e observações experimentais**, destacando comportamentos de saturação.

Essa abordagem garante rastreabilidade completa entre arquitetura, ambiente, experimentos e resultados, pois todos os artefatos — código, infraestrutura e cenários — estão versionados no repositório.

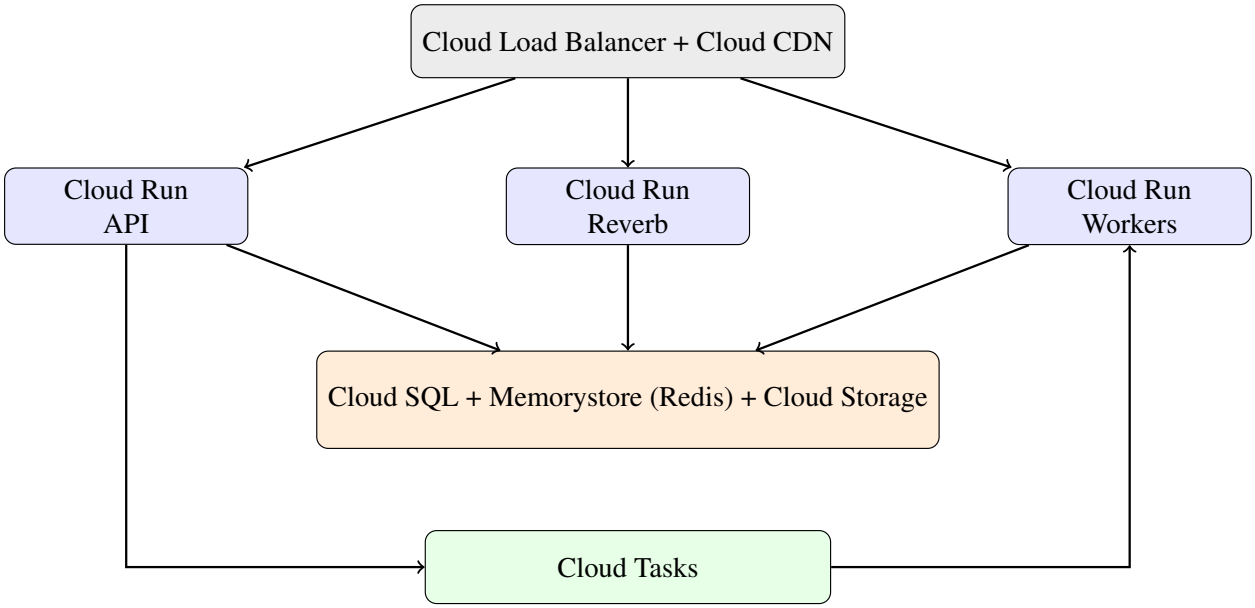

Figura 1. Arquitetura utilizada nos experimentos.

5. Implementação

Esta seção descreve a implementação do ValorizeAI, enfatizando os elementos arquiteturais, os fluxos de execução e os componentes que foram exercitados nos experimentos de carga. O objetivo é apresentar como o sistema opera internamente, de modo a contextualizar os resultados apresentados na próxima seção.

5.1. Arquitetura Lógica da Aplicação

A aplicação segue uma arquitetura modular composta por três subsistemas principais:

- **Serviço HTTP (API Laravel):** responsável pelas operações transacionais síncronas (consultas, criação de transações, ingestão de dados e automações).
- **Servidor WebSocket (Laravel Reverb):** dedicado ao envio de notificações em tempo real e atualização dos dashboards.
- **Workers HTTP (Cloud Run):** executores de tarefas assíncronas disparadas pelo Cloud Tasks.

Cada subsistema é implantado como serviço independente no Cloud Run, permitindo escalonamento isolado e controle fino de concorrência.

5.2. Fluxo Síncrono: API HTTP

O backend Laravel organiza o domínio financeiro em módulos que seguem Clean Architecture e DDD. As rotas públicas acessam controladores que delegam:

- **consultas** para classes *Query*, otimizadas para leitura e usando Redis como cache;
- **escritas** para classes *Action*, que encapsulam validações, regras de negócio, transações no PostgreSQL e emissão de eventos.

Os endpoints exercitados nos testes k6 incluem:

- `GET /api/transactions` — leitura intensiva de listas paginadas;

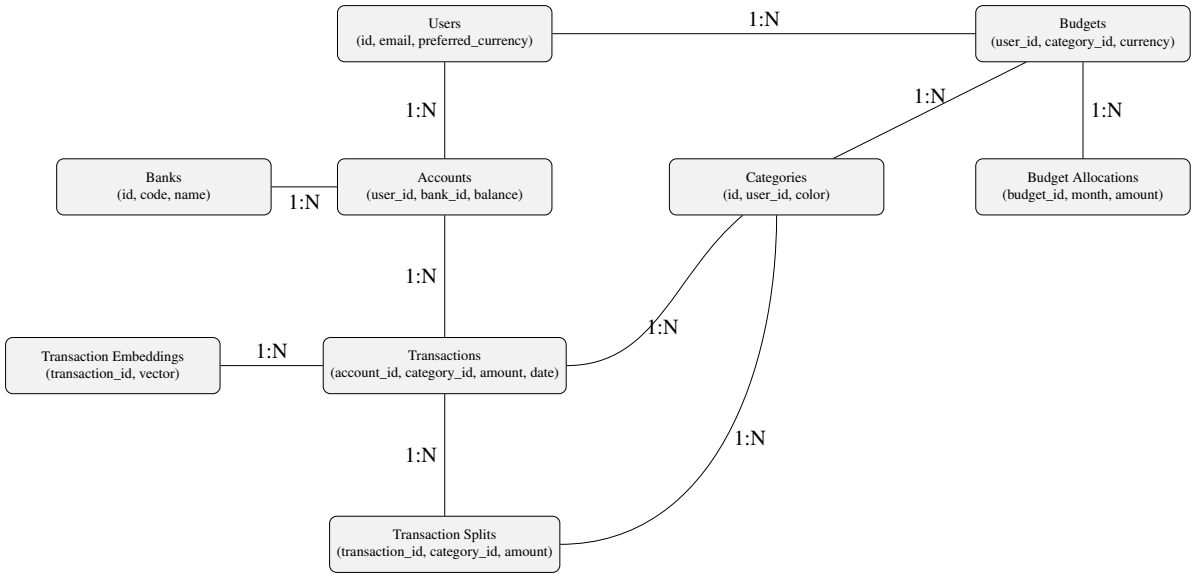

Figura 2. Modelo lógico central derivado do esquema relacional do ValorizeAI.

- `POST /api/transactions` — criação de transações, fluxo que ativa lógica de consistência e atualizações derivadas;
- `GET /api/accounts` — consulta leve usada para refletir mudanças de saldo.

O caminho de leitura foi otimizado com cache *cache-aside*: a primeira consulta popula Redis, e subsequentes retornam em baixa latência. O caminho de escrita, por sua vez, não usa cache e realiza operações ACID no PostgreSQL.

5.3. Fluxo Assíncrono: Cloud Tasks e Workers

Tarefas que exigem maior tempo de processamento são delegadas ao pipeline assíncrono. O processo ocorre em quatro etapas:

1. a API cria uma tarefa via Cloud Tasks, anexando o payload necessário;
2. o serviço envia uma requisição HTTP *push* para o endpoint do worker;
3. o Cloud Run instancia dinamicamente quantos workers forem necessários para consumir o backlog;
4. cada worker executa o processamento (ex.: importação de extratos, geração de relatórios, triggers de automações).

O uso de *push queues* elimina a necessidade de processos consumidores contínuos e garante elasticidade automática baseada no ritmo de produção.

5.4. Comunicação em Tempo Real: WebSockets com Reverb e Redis

O servidor WebSocket do Reverb mantém conexões persistentes com os clientes do painel em tempo real. Como o ambiente Cloud Run escala horizontalmente e instancia múltiplos contêineres, cada instância possui um conjunto próprio de clientes conectados.

Para distribuir eventos entre elas, utiliza-se Redis como *backplane* Pub/Sub:

- Instâncias publicam eventos em canais Redis.
- Todas as instâncias inscritas recebem as mensagens.
- A instância que possui o cliente conectado retransmite o evento via WebSocket.

Esse mecanismo assegura coerência e funcionamento correto mesmo em ambientes altamente elásticos.

5.5. Modelo de Dados e Otimizações

O modelo lógico (Figura 2) segue um desenho multi-inquilino composto por:

- usuários, contas, categorias e orçamentos;
- transações e divisões (*splits*);
- embeddings vetoriais, armazenados via `pgvector` para auxiliar classificação.

Algumas otimizações relevantes para os testes incluem:

- índices compostos em `transactions(date, account_id)`;
- padronização de paginação para reduzir fan-out;
- projeção de DTOs para minimizar transferência de dados;
- pré-aquecimento da cache com consultas frequentes.

5.6. Observabilidade e Instrumentação

A coleta de evidências utilizou:

- **Cloud Monitoring** para métricas de CPU, memória, instâncias ativas, latência e backlog do Cloud Tasks;
- **CSV automatizados do k6** para SLIs de latência e taxa de erro;
- **logs estruturados** para rastrear, por requisição, saturação do banco e falhas de escrita;
- **dashboards personalizados** para acompanhar as instâncias do Cloud Run durante os experimentos.

Essa infraestrutura permite correlacionar comportamentos de aplicação, banco, Redis, filas e WebSockets com precisão.

5.7. Síntese da Implementação

A integração entre:

- API síncrona,
- pipeline assíncrono via Cloud Tasks,
- cache Redis,
- servidor WebSocket escalável

fornece a base operacional necessária para avaliar os SLOs definidos. Os resultados obtidos dependem diretamente dessas decisões arquiteturais, que moldam o comportamento observado sob tráfego elevado.

6. Resultados e Discussão

Esta seção consolida as métricas obtidas nos testes de carga (k6) e no ensaio assíncrono com Cloud Tasks. As análises seguem os SLIs definidos na metodologia: latência P95, taxa de erro e comportamento das filas sob acúmulo de tarefas. O objetivo é verificar a aderência da arquitetura aos SLOs estabelecidos e compreender os pontos de saturação observados.

Os valores de *throughput* foram extraídos diretamente dos dashboards do Cloud Run, refletindo o número real de requisições por segundo atendidas por todas as instâncias durante cada estágio do teste. Assim, VUs e req/s representam perspectivas complementares do mesmo nível de pressão sobre o sistema.

Tabela 1. Resumo dos cenários de carga e conformidade com os SLOs

Cenário	Latência P95	Throughput médio	Throughput pico	Taxa de erro
Leitura intensiva	158 ms	470 req/s	970 req/s	0,00%
Mistura leitura/escrita	658 ms	226 req/s	450 req/s	0,00%

6.1. Cenário de Leitura Intensiva

O cenário de leitura mobilizou 1.000 usuários virtuais por 17 minutos, sustentando em média 470 requisições por segundo e alcançando um pico próximo a 970 req/s no trecho final ($900 \rightarrow 1.000$ VUs). A latência P95 permaneceu em 158 ms (Tabela 1), substancialmente abaixo do SLO de 300 ms, e nenhuma requisição apresentou erro. Esses resultados confirmam que o caminho otimizado de leitura — CDN, API em Cloud Run, Redis como cache e consultas eficientes no PostgreSQL — suporta picos significativos de tráfego sem degradação perceptível.

O Cloud Monitoring registrou o escalonamento completo das 10 instâncias de Cloud Run, atingindo CPU média de 72% e uso de memória em torno de 31%. Ou seja, a cota máxima de CPU foi totalmente utilizada, mas sem sinais de exaustão ou sobrecarga que pudessem comprometer as latências observadas.

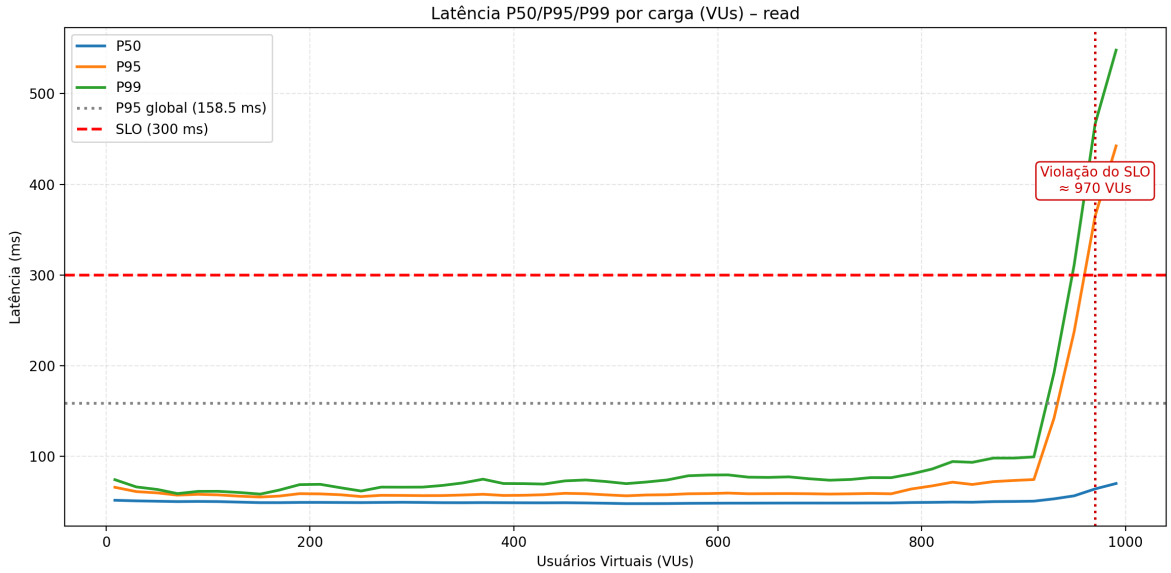

Figura 3. Latências P50/P95/P99 por carga (VUs) no cenário de leitura. A linha tracejada indica o SLO de 300 ms, não violado durante o teste.

6.2. Cenário Misto (Leitura/Escrita)

O cenário misto utilizou provisionamento automático de contas e tokens por VU, eliminando o gargalo artificial presente em versões anteriores do teste. Com 650 usuários alternando entre 65% de leituras e 35% de escritas durante 21 minutos, o sistema sustentou 226 requisições por segundo ($\approx$ 283 mil iterações) sem erros e manteve o SLO de 300 ms até aproximadamente 450 RPS. A violação sistemática do SLO ocorre por volta de 539 VUs, ponto a partir do qual o Cloud Run volta a atingir o limite de 10 instâncias.

Nos estágios acima de 550 VUs, o P95 consolidado do ensaio alcançou 658 ms, enquanto o p99 atingiu 2,67 s. A origem da degradação é clara: o caminho de escrita demanda mais CPU

por requisição (validações, transações ACID e invalidação de cache), reduzindo a capacidade de requisições por instância quando o limite de contêineres é atingido. As métricas do Cloud SQL mostraram estabilidade, indicando que o banco relacional não foi o gargalo primário nesse experimento.

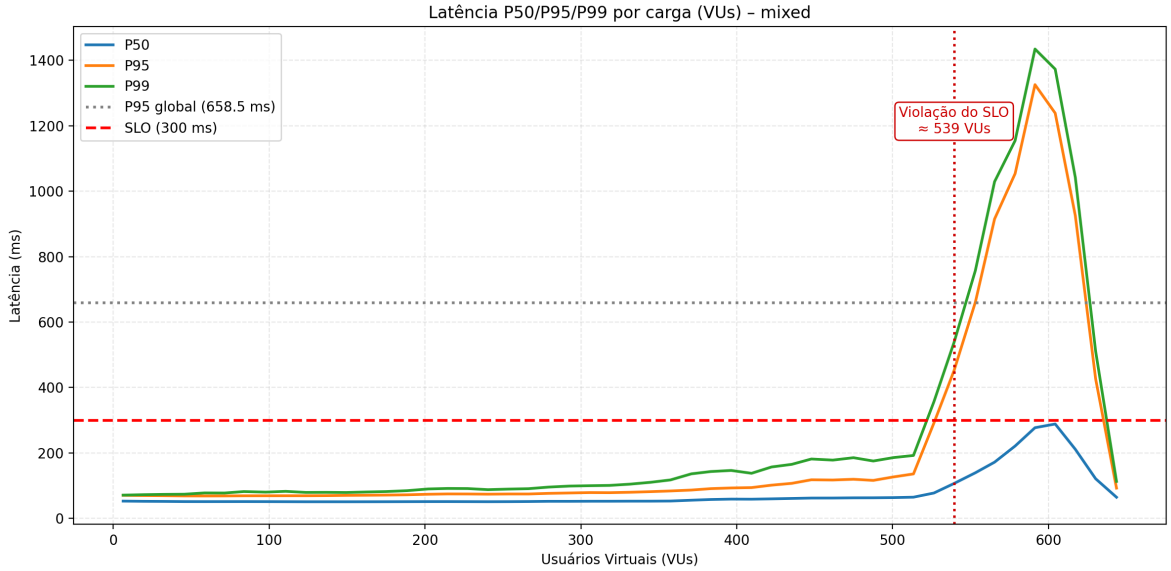

Figura 4. Latências P50/P95/P99 por carga (VUs) no cenário misto. A faixa tracejada marca o SLO de 300 ms; etapas acima de 539 VUs apresentam violações persistentes.

A Figura 5 ilustra o distanciamento entre os comportamentos das duas cargas. Enquanto o cenário de leitura mantém o P95 abaixo de 200 ms mesmo no pico de 1.000 VUs, o cenário misto diverge abruptamente após 500 VUs. Esse comportamento é consistente com modelos teóricos de sistemas transacionais [Kleppmann 2017], segundo os quais rotas de escrita têm custo marginal crescente e menor paralelização efetiva, sobretudo em sistemas baseados em contêineres com cotas rígidas de CPU.

Os resultados reforçam que, antes de adotar mecanismos mais complexos (particionamento, réplicas dedicadas ou CQRS físico), a arquitetura pode obter ganhos expressivos liberando o limite de instâncias HTTP no Cloud Run ou reduzindo o custo computacional por escrita.

6.3. Processamento Assíncrono com Cloud Tasks

O ensaio assíncrono publicou 51,58 mil tarefas em lote e monitorou a drenagem entre 01:08 e 01:18 ($\approx$ 10 minutos). Isso corresponde a aproximadamente 86 tarefas/segundo processadas em média, com variação alinhada ao número de instâncias de worker ativadas dinamicamente. O Cloud Run escalou horizontalmente enquanto havia backlog, reduzindo o número de instâncias assim que o volume de tarefas diminuiu.

Nenhuma entrega foi perdida ou duplicada. A latência ponta-a-ponta permaneceu estável porque:

- o Cloud Tasks opera em modo *push*, eliminando *polling*;
- o Redis atuou somente como *backplane* de eventos e não participou do pipeline crítico;
- cada worker pôde operar de forma idempotente e independente.

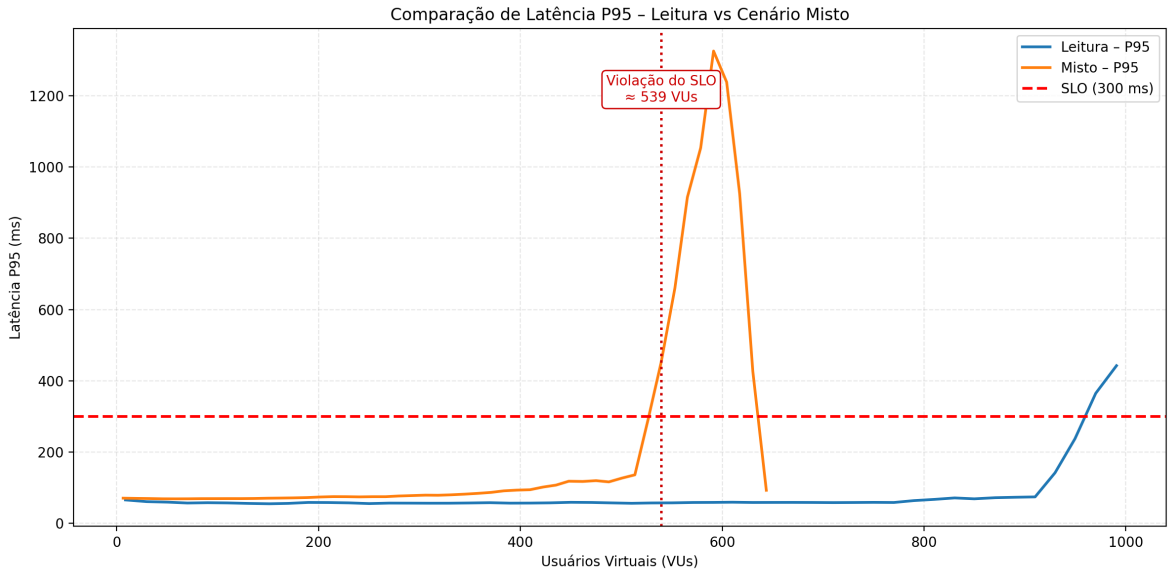

Figura 5. Comparação entre os P95 dos cenários de leitura e misto. A leitura permanece estável; a escrita viola o SLO acima de 539 VUs.

Esse comportamento demonstra elasticidade eficiente: a infraestrutura permanece mínima em períodos ociosos e escala agressivamente apenas durante janelas de pico, alinhando custo e demanda sem intervenção manual.

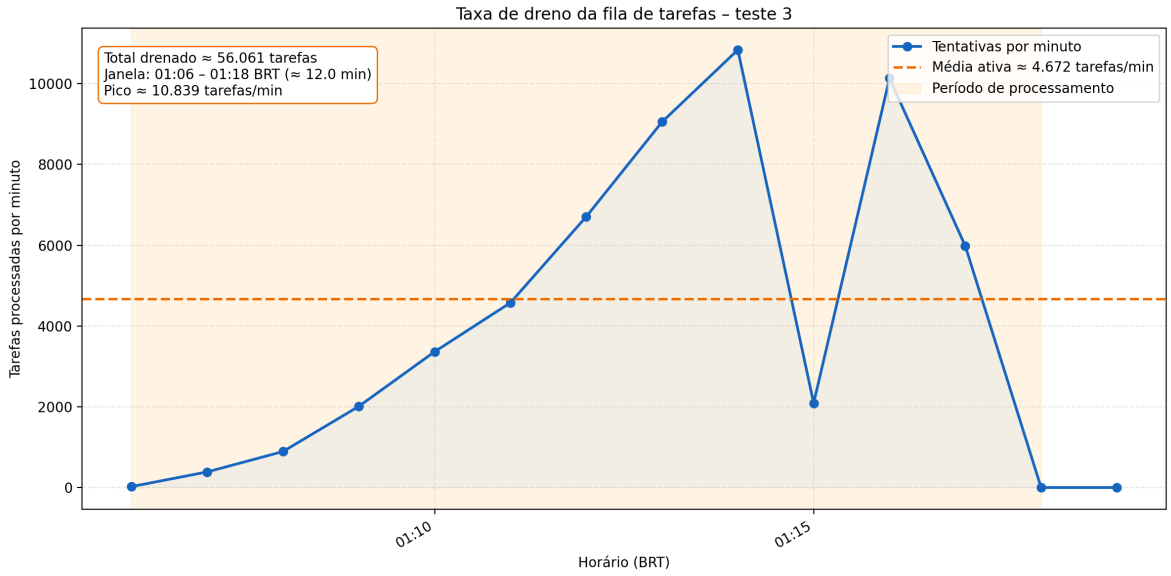

Figura 6. Taxa de processamento das filas (tarefas/min) e janela de drenagem entre 01:08 e 01:18 no ensaio com Cloud Tasks.

6.4. Síntese e Discussão Integrada

Os resultados demonstram que:

- **O caminho de leitura é altamente escalável**, atendendo 1.000 VUs com folga e sem violar SLOs.
- **O caminho de escrita é limitado pela cota de CPU do Cloud Run**, não pelo banco.

- **O pipeline assíncrono é elástico e confiável**, drenando grandes volumes sem perda de tarefas.
- **A arquitetura é adequada para workloads dominados por leitura**, mas exige ajustes para workloads de escrita concorrente.

Esses achados orientam recomendações para escalabilidade futura, incluindo aumento das cotas de instâncias, separação de serviços de escrita, redução do custo computacional das rotas críticas e, eventualmente, adoção de padrões arquiteturais como CQRS físico ou particionamento de dados.

6.5. Análise de Custos Operacionais e Impacto da Elasticidade

Para complementar a avaliação de desempenho, esta seção compara o custo computacional dos serviços serverless utilizados (Cloud Run) com alternativas tradicionais baseadas em máquinas virtuais (Compute Engine). Todas as referências de preço utilizam os valores **Default** da região **us-central1**, garantindo consistência nas comparações.

6.5.1. Custo por vCPU-second: Cloud Run vs. Compute Engine

O Cloud Run adota um modelo de cobrança proporcional ao tempo efetivo de uso da CPU, enquanto o Compute Engine mantém cobrança contínua durante toda a execução da máquina virtual. Os valores são:

- **Cloud Run (instance-based)**: \$0.000018 por vCPU-second.
- **Cloud Run (request-based, ativo)**: \$0.000024 por vCPU-second.
- **Compute Engine C2**: \$0.033982 por vCPU-hora.

Convertendo o custo do Compute Engine para a mesma unidade utilizada pelo Cloud Run:

$$\frac{0.033982 \text{ USD}}{3600 \text{ s}} = 0.000009439 \text{ USD/vCPU-second}.$$

Com os valores normalizados, é possível calcular o fator de diferença entre os modelos:

$$\frac{0.000018}{0.000009439} \approx 1.90, \qquad \frac{0.000024}{0.000009439} \approx 2.54.$$

Assim, o Cloud Run apresenta custo unitário aproximadamente **1,9× (instance-based)** a **2,5× (request-based)** superior ao de uma vCPU otimizada do tipo C2 no Compute Engine. Essa discrepância, no entanto, não implica necessariamente maior custo operacional total, como discutido a seguir.

6.5.2. Por que o Cloud Run pode custar menos mesmo tendo custo unitário mais alto?

O Compute Engine incorre em custo constante: ao provisionar uma VM, os vCPUs permanecem alocados e cobrados 24 horas por dia, independentemente da demanda. Dessa forma, workloads irregulares inevitavelmente produzem longos períodos de ociosidade, que se convertem diretamente em custo desperdiçado.

Por outro lado, o Cloud Run realiza *scale-to-zero* e cobra CPU e memória apenas durante o uso efetivo. Em sistemas como o ValorizeAI — cujo tráfego é altamente variável e sensível a

picos — essa característica elimina integralmente o custo ocioso. Na prática, o custo total mensal tende a ser menor, apesar do preço unitário mais alto por vCPU-second.

6.5.3. GKE como solução intermediária

O Google Kubernetes Engine (GKE) permite reduzir o custo unitário por utilizar nós Compute Engine, mas introduz novos desafios:

- maior complexidade operacional (HPA, VPA, node pools, autoscaling);
- tempo de reação mais lento a picos repentinos;
- probabilidade maior de manter vCPUs ociosas em períodos irregulares.

Como resultado, GKE costuma apresentar um custo total *intermediário* entre Compute Engine e Cloud Run, porém ao custo de maior responsabilidade operacional e menor granularidade de escalonamento.

6.5.4. Síntese comparativa

Tabela 2. Comparação econômica entre Cloud Run, GKE e Compute Engine (valores Default, us-central1).

Serviço	CPU (USD/vCPU-s)	Elasticidade	Ociosidade esperada
Cloud Run (request-based)	0.000024	Alta	Muito baixa
Cloud Run (instance-based)	0.000018	Alta	Baixa
Compute Engine (C2)	0.00000944	Nenhuma	Alta
GKE (nós C2)	0.00000944	Moderada	Moderada

Em síntese, mesmo possuindo custo unitário superior, o Cloud Run tende a apresentar custo operacional real menor para cargas irregulares ou sujeitas a picos abruptos — cenário típico de aplicações transacionais modernas. Já Compute Engine e GKE oferecem menor custo por vCPU-second, porém à custa de maior ociosidade ou maior complexidade operacional.

7. Conclusão

O ValorizeAI demonstrou que é possível documentar e validar, de ponta a ponta, uma arquitetura orientada a eventos construída integralmente sobre serviços gerenciados. O trabalho conectou decisões de design — Cloud Run, Redis, Cloud SQL, Cloud Tasks e Reverb — ao domínio transacional modelado no banco relacional, oferecendo um roteiro explícito para equipes que necessitam de elasticidade sem abrir mão de consistência forte. A metodologia experimental, fundamentada em princípios de SRE, complementou essa documentação ao mostrar como planejar SLOs, provisionar infraestrutura como código e conduzir testes reprodutíveis, reforçando o caráter aplicado da pesquisa.

Os cenários de carga confirmaram a eficácia das otimizações de leitura: com 1.000 usuários virtuais (≈ 470 req/s em média e pico de ≈ 970 req/s), o sistema manteve latência P95 de 158 ms e taxa de erro nula, atendendo plenamente aos SLOs estabelecidos. Já o cenário misto revelou o limiar atual da solução: o patamar de 650 VUs (≈ 226 req/s) elevou o P95 para 658 ms e o p99 para 2,67 s, resultado diretamente relacionado ao limite de 10 instâncias de Cloud Run e ao maior

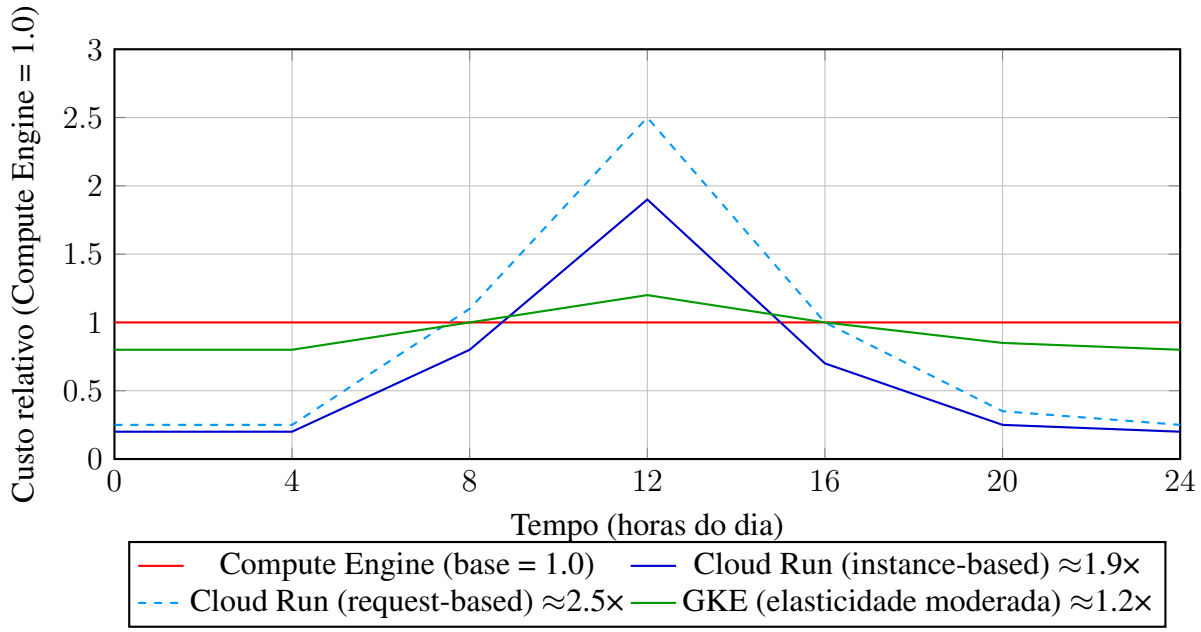

Figura 7. Custo relativo entre Compute Engine, Cloud Run e GKE, normalizado para Compute Engine = 1.0.

custo computacional das rotas de escrita (validações, transações ACID e invalidação de cache). O Cloud SQL manteve tempos estáveis ao longo do teste, indicando que o gargalo predominante permanece na camada HTTP. No plano assíncrono, a drenagem de 51,58 mil tarefas em 10 min comprovou que a combinação Cloud Tasks + workers em Cloud Run sustenta rajadas intensas sem intervenção manual, preservando elasticidade e rastreabilidade do processamento.

7.1. Limitações

Duas limitações principais moldaram os resultados. Primeiro, a infraestrutura operou com as cotas padrão de um ambiente recém-provisionado (10 instâncias de 1 vCPU), o que restringiu a observação de comportamentos em escalas superiores. Segundo, o banco de dados permaneceu centralizado em uma única instância regional do Cloud SQL; embora suficiente para o tráfego exercitado, essa configuração pode induzir contenção em cenários de escrita simultânea em grande escala.

7.2. Trabalhos Futuros

Como continuidade, pretende-se ampliar progressivamente a cota de Cloud Run, repetindo os testes sob 20 ou 40 vCPU para identificar novos gargalos. Também se planeja avaliar alternativas para aliviar operações analíticas, como réplicas de leitura e ajustes verticais das instâncias primárias para absorver picos de escrita. Outro eixo de evolução envolve mecanismos de *failover* multi-região para banco e Redis, aumentando a resiliência global da plataforma. No plano funcional, o pipeline assíncrono poderá ser estendido com orquestração baseada em eventos (por exemplo, Workflows), além da expansão do uso de embeddings e automações de categorização para casos como detecção de anomalias e recomendações transacionais.

Referências

Abad, C., Foster, I. T., Herbst, N., e Iosup, A. (2021). Serverless computing: One step forward, two steps back. *IEEE Computer*, 54(3):48–58. Refereciado por [1].

Barri (2025). Mastering website scalability for large traffic: Preparing for surges. Technical report, Queue-Fair. https://queue-fair.com/website-scalability-for-large-traffic, Acessado em: 2025-10-14.

Basteri, A. (2023). What makes observability a priority. Technical report, New Relic. https://newrelic.com/resources/white-papers/observability-as-a-priority, Acessado em: 2025-10-14.

Beyer, B., Jones, C., Petoff, J., e Murphy, N. R. (2016). *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., Sebastopol, CA, USA.

Cervone, V. (2024). k6 - performance testing for developers. Technical report, Grafana Labs. https://k6.io/, Acessado em: 2025-10-14.

Christidis, K. et al. (2022). Serverless cloud architectures for machine learning model deployment: A systematic review and case study. *World Journal of Advanced Engineering and Technology (WJAETS)*, 5(2). Disponível em: https://wjaets.com/sites/default/files/WJAETS-2022-0025.pdf.

Confluent (2024). Event-driven architecture (eda): A complete introduction. Technical report, Confluent Inc. https://www.confluent.io/learn/event-driven-architecture/, Acessado em: 2025-10-14.

Datadog (2024). Serverless vs. containers: What's the difference? Technical report, Datadog Research. https://www.datadoghq.com/knowledge-center/serverless-architecture/serverless-vs-containers/, Acessado em: 2025-10-14.

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, Boston, MA, USA.

Fernando, L. e Engel, M. M. (2025). Comparative performance benchmarking of websocket libraries on node.js and golang. *Sinkron : Jurnal dan Penelitian Teknik Informatika*, 9(4).

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, Boston, MA, USA.

Fowler, M. (2011). Cqrs. Technical report, martinfowler.com. https://martinfowler.com/bliki/CQRS.html, Acessado em: 2025-10-14.

Google Cloud (2024). What is cloud elasticity? understanding elastic computing. Technical report, Google Cloud. https://cloud.google.com/discover/what-is-cloud-elasticity, Acessado em: 2025-10-14.

Guerriero, A. et al. (2019). Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

Hebbar, K. S. (2025). Priority-aware reactive apis: Leveraging spring webflux for sla-tiered traffic in financial services. *European Journal of Electrical Engineering and Computer Science*, 9(5).

Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media, Inc.

Laravel Holdings Inc. (2025). Laravel reverb official documentation. Technical report, Laravel Holdings Inc. https://reverb.laravel.com, Acessado em: 2025-10-14.

Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., e Pallickara, S. (2018). Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*.

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, Hoboken, NJ, USA.

McCoy, J. e Forsgren, N. (2020). *SLO Adoption and Usage in Site Reliability Engineering*. O'Reilly Media, Inc., Sebastopol, CA, USA.

Pessa, A. (2023). Comparative study of infrastructure as code tools for amazon web services. Master's thesis, Tampere University. Disponível em: https://trepo.tuni.fi/bitstream/handle/10024/149567/PessaAntti.pdf.

Sonawane, S. et al. (2024). The role of serverless architecture in scalable and efficient web development. *International Journal of Scientific Research in Science and Technology*. Disponível em: https://www.researchgate.net/publication/389615854_The_Role_of_Serverless_Architecture_in_Scalable_and_Efficient_Web_Development.

Thepphakan, A. (2025). Study of real-time data communication using pulsar and rabbitmq (case study of stock price in lao securities exchange). *International Journal of Advanced Research in Computer Science and Software Engineering*.

Twine (2022). Twine case study: A scalable and open-source raas. Technical report, Twine Realtime Initiative. https://twine-realtime.github.io/case-study, Acessado em: 2025-10-14.

Yadav, P. S. (2019). Designing a high-performance real-time leaderboard system using redis: Scalability, efficiency, and fault tolerance. *Journal of Scientific and Engineering Research*, 6(3):313–320.