



# placeholder

*Title:*

**Felipe Tomkiel Malacarne, Prof. Me. Marcos André Lucas aaaa**

<sup>1</sup> 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

**Abstract.** todo

**Keywords** Serverless Computing, Cloud Run, Financial Platforms, Performance Testing, Observability.

**Resumo.** todo

**Palavras-Chave** Computação Serverless, Cloud Run, Plataformas Financeiras, Testes de Carga, Observabilidade.

## 1. Introdução

### 1.1. Contextualização e Motivação

Aplicações digitais modernas, que abrangem desde plataformas de e-commerce e serviços financeiros até mídias sociais e sistemas de colaboração em tempo real, enfrentam um desafio operacional comum: a gestão de cargas de trabalho (workloads) voláteis e imprevisíveis [Google Cloud 2024]. Picos de tráfego, eventos contínuos de ingestão de dados e a necessidade de múltiplas integrações com sistemas externos demandam uma infraestrutura que reaja dinamicamente, muito além da capacidade de provisionamento manual ou pipelines de integração monolíticos e rigidamente acoplados.

A resposta da indústria a esse desafio é a *elasticidade na nuvem* (cloud elasticity), definida como a capacidade de uma infraestrutura de computação em nuvem alocar e desalocar recursos computacionais de forma automática e autônoma, com base na flutuação da demanda em tempo real [Google Cloud 2024]. Diferente da escalabilidade tradicional, que muitas vezes envolve intervenção humana para provisionar novas instâncias, a elasticidade é projetada para lidar com picos abruptos e vales de tráfego, garantindo tanto o desempenho quanto a eficiência de custos [Google Cloud 2024]. Os benefícios de negócios são diretos: (1) *Eficiência de Custo*, ao adotar um modelo *pay-as-you-go* que evita o superprovisionamento dispendioso de recursos ociosos; e (2) *Alta Disponibilidade e Desempenho*, ao assegurar que a aplicação permaneça responsável e confiável, preservando a experiência do usuário mesmo sob demanda extrema [Google Cloud 2024].

Contudo, a adoção de arquiteturas que possibilitam essa elasticidade — notadamente microserviços, contêineres e paradigmas *serverless* [Basteri 2023] — introduz um nível exponencial de complexidade. Sistemas distribuídos são intrinsecamente mais difíceis de depurar e monitorar do que monólitos. Em resposta, a *observabilidade* (observability) emergiu como uma prioridade estratégica [Basteri 2023]. A observabilidade transcende o monitoramento tradicional (que rastreia falhas *conhecidas*) ao fornecer *insights* sobre o estado interno do sistema a partir de seus outputs (logs, métricas e traces), permitindo a depuração de falhas *desconhecidas* [Basteri 2023].

Elasticidade e observabilidade não são, portanto, características independentes, mas um ciclo de *feedback* simbótico. A elasticidade automática gera uma complexidade que só pode ser gerenciada pela observabilidade. Por sua vez, a observabilidade fornece os dados (na forma de Indicadores de Nível de Serviço, ou SLIs) que alimentam e validam o próprio mecanismo de elasticidade, garantindo que o escalonamento automático atenda aos objetivos de negócio (SLOs) sem incorrer em custos desnecessários [Basteri 2023]. A falha de processos manuais é uma consequência direta da alta velocidade desse ciclo, que opera em milissegundos.

O ValorizeAI, objeto deste trabalho, nasce como um estudo de caso completo para investigar essa simbiose. Trata-se de uma aplicação web modular que centraliza fluxos complexos de ingestão de dados, processamento síncrono e assíncrono, e entrega de notificações e painéis em tempo real, utilizando uma *stack* de tecnologias modernas.

## 1.2. Justificativa e Problema de Pesquisa

Workloads transacionais que concentram ingestão massiva de dados, gerenciamento de estados compartilhados e interfaces colaborativas — como os simulados pelo ValorizeAI — impõem requisitos rigorosos de arquitetura. Tais sistemas exigem consistência forte nos dados, rastreabilidade completa para fins de auditoria e, fundamentalmente, respostas de baixa latência (respostas instantâneas), mesmo quando o tráfego varia abruptamente.

Para atender a esses requisitos, arquiteturas modernas combinam múltiplos padrões especializados:

- **CDNs e Balanceamento Global:** Para reduzir a latência de entrega de *assets* estáticos, distribuindo o conteúdo para pontos de presença (PoPs) próximos ao usuário [Barri 2025].
- **Filas Assíncronas (EDA):** Para desacoplar tarefas pesadas (ex: processamento de relatórios, envio de e-mails) da resposta síncrona, garantindo resiliência e escalabilidade através de uma arquitetura orientada a eventos [Confluent 224].
- **Cache Distribuído (Redis):** Para armazenar dados frequentemente acessados em memória, reduzindo drasticamente a latência de leitura e a carga sobre o banco de dados principal [Yadav 2019].
- **WebSockets (Tempo Real):** Para comunicação bidirecional persistente, essencial para painéis e notificações em tempo real sem a sobrecarga do HTTP *polling* [Fernando e Engel 2025].

Embora existam tutoriais e artigos pontuais sobre cada uma dessas tecnologias, a justificativa deste trabalho reside na lacuna da literatura acadêmica e técnica [Christidis et al. 2022, Abad et al. 2021]. É raro encontrar material que conecte, de ponta a ponta (end-to-end), a implementação de uma arquitetura híbrida (CaaS + Filas + WebSockets) aos resultados práticos e reproduutíveis de testes de desempenho. A literatura existente tende a ser fragmentada, focando em comparações de ferramentas de IaC [Pessa 2023] ou na validação de microsserviços específicos [Hebbar 2025], mas raramente no sistema holístico.

A contribuição deste TCC é, portanto, metodológica e empírica. Ao construir o ValorizeAI e registrar rigorosamente os experimentos de carga e processamento assíncrono, este trabalho evidencia como as decisões arquiteturais se traduzem em métricas quantificáveis (ex: latência P95, throughput de tarefas, taxa de erro). O problema de pesquisa é: *Como uma arquitetura híbrida e elástica, composta por contêineres gerenciados, servidor de WebSockets dedicado e filas assíncronas, se comporta sob estresse de carga e qual a metodologia para validar seu desempenho de forma reproduutível contra SLOs pré-definidos?*

Documentar esse caminho, apoiado por Infraestrutura como Código (IaC) para reprodutibilidade e monitoramento de custo, fornece uma referência concreta para equipes técnicas que precisam justificar e implementar arquiteturas orientadas a eventos com elasticidade horizontal automática.

### 1.3. Objetivo Geral

Demonstrar, por meio de documentação técnica e experimentos de desempenho, que a arquitetura do ValorizeAI — composta por balanceador com CDN, contêineres escalados horizontalmente, processamento assíncrono em filas, servidor de WebSockets, *buckets* para artefatos e cache em Redis — sustenta os SLOs definidos para um produto transacional completo, mantendo todo o ciclo (modelagem, desenvolvimento, infraestrutura, observabilidade e testes) versionado no repositório.

### 1.4. Objetivos Específicos

1. **Mapear a arquitetura end-to-end**, destacando o papel do balanceador/CDN, das instâncias de contêineres, do servidor de WebSockets, das filas assíncronas, dos *buckets* de armazenamento e do Redis para garantir consistência e baixa latência.
2. **Documentar o desenvolvimento** do *backend* Laravel, do *frontend* React e dos fluxos síncronos/assíncronos, com foco nos módulos críticos (ingestão de dados, automações, notificações e painéis em tempo real).
3. **Planejar e executar os testes de carga** (k6, cenários de leitura e de leitura/escrita) e o teste de processamento assíncrono para validar horizontalmente a arquitetura frente aos SLOs e identificar gargalos.
4. **Interpretar os resultados e propor otimizações**, relacionando desempenho, elasticidade e custo (ex.: ajustes de limites de instância, estratégias de cache) e apontando como essas evidências fundamentam decisões para *workloads* transacionais de alta criticidade.

### 1.5. Estrutura do Trabalho

Este trabalho está organizado da seguinte forma: O Capítulo 2 apresenta uma revisão da literatura sobre paradigmas de execução em nuvem, padrões arquiteturais e metodologias de validação, identificando a lacuna que este TCC se propõe a preencher. O Capítulo 3 define os conceitos teóricos e técnicos canônicos que fundamentam o design do sistema e a metodologia de testes. O Capítulo 4 detalha a metodologia empregada, incluindo os SLOs definidos e a configuração dos testes. O Capítulo 5 descreve a implementação da arquitetura ValorizeAI. O Capítulo 6 apresenta e discute os resultados dos experimentos de desempenho. Finalmente, o Capítulo 7 conclui o trabalho e sugere direções para pesquisas futuras.

## 2. Trabalhos Relacionados

A arquitetura proposta pelo ValorizeAI situa-se na interseção de três domínios de pesquisa em engenharia de software e sistemas distribuídos: (1) os paradigmas de execução em nuvem, (2) os padrões de design para resiliência e desempenho, e (3) as metodologias de validação empírica. Esta seção revisa o estado da arte em cada um desses eixos para posicionar a contribuição deste trabalho e fundamentar a lacuna de pesquisa identificada na Introdução.

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

A primeira decisão de design em arquiteturas elásticas modernas é a escolha do paradigma de computação. A literatura recente concentra-se no debate entre *Functions-as-a-Service* (FaaS) e *Containers-as-a-Service* (CaaS).

O paradigma FaaS, popularizado por serviços como AWS Lambda e Google Cloud Functions, abstrai completamente o gerenciamento de servidores, oferecendo um modelo de faturamento por execução e escalabilidade instantânea (incluindo *scale-to-zero*) [Sonawane et al. 2024]. Esse modelo é ideal para *workloads* reativos, *stateless* e de curta duração. No entanto, a literatura aponta desafios significativos: (1) a latência de inicialização (cold start), que pode impactar o desempenho de aplicações sensíveis à latência [Sonawane et al. 2024]; (2) a complexidade de monitoramento e observabilidade em um ambiente altamente efêmero [Sonawane et al. 2024]; e (3) a inadequação para processos *stateful* ou de longa duração, como conexões de banco de dados persistentes ou servidores WebSocket [Datadog 2024].

Em contrapartida, o CaaS, exemplificado por plataformas como Google Cloud Run e AWS Fargate, emerge como um meio-termo estratégico [Lloyd et al. 2018]. O CaaS combina a elasticidade e o modelo de *scale-to-zero* do FaaS com a portabilidade, consistência e controle de ambiente fornecidos pelos contêineres (ex: Docker) [Datadog 2024].

A arquitetura do ValorizeAI requer explicitamente um servidor de WebSockets dedicado (Reverb) para comunicação em tempo real — um processo *stateful* e de longa duração. A revisão da literatura [Datadog 2024, Sonawane et al. 2024] demonstra que o FaaS é um paradigma inadequado para esse requisito. O CaaS (especificamente o Cloud Run) foi, portanto, escolhido por ser o paradigma que permite a execução de processos persistentes (o contêiner do Reverb) enquanto ainda fornece a elasticidade horizontal automática e a abstração de infraestrutura desejadas para os serviços web *stateless*.

### 2.2. Padrões Arquiteturais para Desempenho e Resiliência

Para atender aos requisitos de um sistema transacional em tempo real, o ValorizeAI combina padrões de comunicação síncronos e assíncronos.

#### 2.2.1. Comunicação Assíncrona e Arquiteturas Orientadas a Eventos (EDA)

O processamento assíncrono por meio de filas, um pilar central do ValorizeAI, é a implementação prática de uma Arquitetura Orientada a Eventos (EDA). EDAs são definidas como sistemas que promovem o desacoplamento (*loose coupling*), a escalabilidade e a resiliência [Confluent 224]. Ao utilizar um *message broker* (como RabbitMQ, Kafka ou serviços gerenciados como Google Cloud Tasks), os serviços "produtores" podem enfileirar tarefas (eventos) sem esperar que os "consumidores" os processem [Confluent 224]. Isso permite que o sistema absorva picos de escrita e mantenha a responsividade da interface do usuário, além de garantir a entrega de tarefas mesmo que os serviços consumidores falhem temporariamente.

A pesquisa acadêmica neste domínio frequentemente se concentra em análises de desempenho comparativas dos *brokers* de mensagens. Por exemplo, estudos comparam o desempenho de RabbitMQ, Apache Kafka e Apache Pulsar em cenários de IoT, medindo *throughput* e latência sob diferentes tamanhos de mensagem [Thepphakan 2025]. Esses estudos validam que a escolha da tecnologia de fila deve estar alinhada com os requisitos específicos do *workload* (ex: baixa latência para mensagens pequenas vs. alto *throughput* para *streams* de dados).

## 2.2.2. Comunicação em Tempo Real (WebSockets) e Cache Distribuído (Redis)

Para os requisitos de "paineis em tempo real" e "notificações instantâneas", a arquitetura do ValorizeAI utiliza Laravel Reverb e Redis. A literatura não trata desses componentes isoladamente, mas sim como um padrão arquitetural combinado para escalar aplicações em tempo real.

O Redis é amplamente citado por seu papel como um *cache* distribuído em memória, fornecendo acesso a dados de baixa latência e alto *throughput*, o que reduz a carga sobre bancos de dados relacionais [Yadav 2019]. Servidores WebSocket (como o Reverb [Laravel Holdings Inc. 2025]) fornecem o canal de comunicação bidirecional persistente necessário para que o servidor envie dados aos clientes sem que eles precisem solicitá-los (push) [Twine 2022].

O desafio de escalar WebSockets reside na sua natureza *stateful* (o servidor deve manter o registro de cada conexão ativa). Em um ambiente de CaaS elástico como o Cloud Run, onde múltiplas instâncias *stateless* são criadas e destruídas dinamicamente, uma conexão WebSocket estabelecida com a "Instância A" não pode ser acessada pela "Instância B". A literatura e a documentação técnica [Laravel Holdings Inc. 2025] resolvem isso usando o mecanismo de Publicação/Subscrição (Pub/Sub) do Redis como um *backplane* de mensagens. Quando a Instância B precisa enviar uma mensagem para um usuário conectado à Instância A, ela publica a mensagem no canal Redis. A Instância A, que está inscrita (subscribed) nesse canal, recebe a mensagem e a retransmite ao cliente correto através de sua conexão WebSocket local.

Estudos de desempenho de bibliotecas WebSocket, como o de Fernando e Engel [Fernando e Engel 2025], validam a importância de escolhas de implementação leves, focando em métricas-chave como *throughput* (mensagens/segundo) e latência de *Round Trip Time* (RTT) para garantir o desempenho em tempo real.

## 2.3. Metodologias de Validação Empírica

Provar que uma arquitetura complexa atende aos seus requisitos de desempenho exige uma metodologia de validação rigorosa e, idealmente, reproduzível.

### 2.3.1. Infraestrutura como Código (IaC) para Reprodutibilidade

A Infraestrutura como Código (IaC) é uma prática de DevOps onde a infraestrutura de TI (redes, máquinas virtuais,平衡adores de carga) é provisionada e gerenciada usando arquivos de definição legíveis por máquina (ex: Terraform, AWS CDK), em vez de configuração manual [Pessa 2023]. No contexto da pesquisa acadêmica e de engenharia, o principal benefício do IaC é a *reprodutibilidade*. Ao versionar a configuração da infraestrutura juntamente com o código da aplicação, o IaC garante que o ambiente de teste possa ser recriado de forma consistente, eliminando a "deriva de configuração" e tornando os resultados dos testes de desempenho verificáveis [Guerriero et al. 2019].

A literatura sobre IaC, como o estudo de Pessa [Pessa 2023], muitas vezes foca na comparação das próprias ferramentas de IaC (ex: AWS CDK vs. Terraform) em termos de desempenho de provisionamento e experiência do desenvolvedor, em vez de usar o IaC como um *meio* para validar o desempenho da *aplicação* que ele provisiona.

### **2.3.2. Validação de SLOs com Testes de Carga (k6)**

A metodologia do ValorizeAI baseia-se nos princípios de Engenharia de Confiabilidade de Sites (SRE), onde o sucesso é medido pelo cumprimento dos Objetivos de Nível de Serviço (SLOs) [McCoy e Forsgren 2020]. A validação de SLOs requer testes empíricos sob carga.

A literatura acadêmica recente começa a adotar ferramentas de teste de carga modernas, como o k6 [Cervone 2024], para essa finalidade. Um exemplo notável é o trabalho de Hebbar [Hebbar 2025] sobre APIs reativas para o setor financeiro. Hebbar utiliza o k6 para criar perfis de carga (ex: rajada, estado estacionário) e simular tráfego contra um microsserviço Spring WebFlux [Hebbar 2025]. A contribuição desse estudo é a validação de que a arquitetura consegue aplicar priorização de tráfego (tiering de SLA) em tempo real, medindo métricas de latência, taxa de descarte e saturação [Hebbar 2025]. Este estudo serve como um "espelho" metodológico, validando a abordagem do ValorizeAI (uso de k6 para medir métricas de latência P95 contra SLOs definidos) como academicamente rigorosa e alinhada com o estado da arte da pesquisa em desempenho de sistemas.

## **2.4. Síntese da Revisão e Identificação da Lacuna**

A revisão da literatura revela que a pesquisa é frequentemente especializada e fragmentada. Encontramos estudos que comparam FaaS vs. CaaS [Lloyd et al. 2018], analisam o desempenho de *brokers* de EDA [Thepphakan 2025], comparam ferramentas de IaC [Pessa 2023], ou validam um microsserviço específico usando k6 e SLOs [Hebbar 2025].

A lacuna na literatura, identificada em trabalhos como [Christidis et al. 2022] e [Abad et al. 2021], é a ausência de estudos de caso *end-to-end* que integrem *todos* esses componentes. Falta um trabalho que documente e valide empiricamente uma arquitetura holística e híbrida (CaaS + EDA + WebSockets + Cache) que seja:

1. Provisionada de forma reproduzível (via IaC).
2. Validada rigorosamente contra SLOs de latência e *throughput* (via k6).
3. Analisada em seus múltiplos componentes (fluxos síncronos e assíncronos).

A Tabela 1 visualiza essa lacuna. Enquanto trabalhos anteriores focam em colunas específicas, este TCC (ValorizeAI) é o único que propõe uma validação integrada de todos os eixos: Paradigma, Padrões Híbridos e Metodologia de Validação Completa. Este trabalho preenche, assim, a lacuna ao fornecer um "plano" de arquitetura e validação, completo e empiricamente verificado, para aplicações transacionais modernas em tempo real.

**Tabela 1. Quadro Comparativo de Estudos sobre Desempenho de Arquiteturas em Nuvem (2018-2025)**

<b>Estudo (Autor)</b>	<b>Paradigma</b>	<b>Padrões Analisados</b>	<b>Metodologia de Validação</b>
Lloyd et al. [Lloyd et al. 2018]	FaaS vs. CaaS	Fatores de desempenho em microsserviços simples.	Benchmarking de desempenho (latência, custo). Não foca em IaC ou SLOs formais.
Pessa [Pessa 2023]	N/A (Foco na ferramenta)	Provisionamento de infraestrutura (FaaS, CaaS).	Comparação de ferramentas de IaC (CDK vs. Terraform). Não valida desempenho da aplicação.
Hebbar [Hebbar 2025]	Microsserviço (Monolítico)	API Reativa (Spring WebFlux) com priorização.	<b>SLOs e k6.</b> Não foca em IaC, EDA ou WebSockets.
Thepphakan [Thepphakan 2025]	N/A (Foco no broker)	EDA (RabbitMQ vs. Pulsar).	Benchmarking de desempenho (latência, throughput). Não é um sistema E2E.
Abad et al. [Abad et al. 2021]	FaaS / Serverless	Revisão de aplicações serverless.	Análise de literatura. Aponta a <b>lacuna</b> em estudos E2E e workflows complexos.
<b>ValorizeAI (Este TCC)</b>	<b>CaaS Híbrido</b> (Cloud Run)	<b>E2E (EDA + WebSockets + Cache).</b>	<b>IaC (Terraform) + SLOs + k6.</b>

### 3. Fundamentação Teórica

Este capítulo estabelece o vocabulário canônico e os fundamentos conceituais utilizados no design, implementação e validação do sistema ValorizeAI. São apresentados os princípios de design de software, as arquiteturas de componentes tecnológicos e os conceitos de engenharia de confiabilidade que formam a base deste trabalho.

#### 3.1. Princípios de Design de Software

O ValorizeAI adota uma abordagem de "arquitetura limpa", segregando responsabilidades com base em princípios estabelecidos de design de software.

##### 3.1.1. Clean Architecture

Formalizada por Robert C. Martin, a *Clean Architecture* (Arquitetura Limpa) é um modelo arquitetural que advoga pela separação de interesses [Martin 2017]. Seu objetivo é criar sistemas que sejam: (1) Independentes de frameworks; (2) Testáveis; (3) Independentes da interface do usuário (UI); e (4) Independentes do banco de dados [Martin 2017].

O pilar central dessa arquitetura é a *Regra da Dependência* (The Dependency Rule). Esta regra estipula que as dependências do código-fonte devem apontar exclusivamente "para dentro—de camadas de baixo nível (detalhes voláteis, como frameworks e bancos de dados) para camadas de alto nível (políticas de negócio estáveis e abstrações)" [Martin 2017]. No ValorizeAI, isso se manifesta na separação das regras de negócio (localizadas em *Actions* ou *Queries*) da lógica do framework (Controladores Laravel) ou da persistência (Modelos Eloquent).

##### 3.1.2. Domain-Driven Design (DDD)

O *Domain-Driven Design* (DDD), introduzido por Eric Evans, é uma abordagem para o desenvolvimento de software que se concentra em modelar o software para corresponder a um domínio de negócio complexo [Evans 2003]. O DDD é essencial para gerenciar a complexidade em sistemas como o ValorizeAI. Os conceitos-chave utilizados neste trabalho incluem:

- **Linguagem Ubíqua (Ubiquitous Language):** Um vocabulário compartilhado e rigoroso, desenvolvido em colaboração entre os desenvolvedores e os especialistas do domínio (usuários). Essa linguagem é usada em todas as comunicações e reflete-se diretamente no código (nomes de classes, métodos e variáveis) [Evans 2003].
- **Contexto Delimitado (Bounded Context):** A fronteira explícita dentro da qual um modelo de domínio e sua Linguagem Ubíqua são aplicáveis e consistentes [Evans 2003].
- **Agregado (Aggregate):** Um cluster de objetos de domínio (Entidades e Objetos de Valor) que é tratado como uma única unidade para fins de consistência de dados. Um Agregado possui uma raiz (a *Aggregate Root*), que é o único ponto de entrada para modificações dentro do Agregado, garantindo que todas as regras de negócio (invariantes) sejam aplicadas [Evans 2003].

##### 3.1.3. Padrões de Comunicação e Segregação

Para implementar a Regra da Dependência e gerenciar o fluxo de dados, o ValorizeAI utiliza padrões de segregação e transferência de dados.

- **DTO (Data Transfer Object):** Conforme popularizado por Martin Fowler, um DTO é um objeto simples, sem comportamento, cujo único propósito é transferir dados entre subsistemas ou camadas [Fowler 2002]. Em arquiteturas distribuídas ou em camadas, os DTOs são usados para agregar múltiplas chamadas em uma única, reduzindo a latência da rede e desacoplando os modelos internos (domínio) dos modelos de visualização (API/UI).
- **CQRS (Command Query Responsibility Segregation):** Um padrão, descrito por Martin Fowler [Fowler 2011] e Greg Young, que propõe a segregação dos modelos de dados e da lógica de aplicação em duas categorias: *Commands* (operações que alteram o estado, ou seja, escritas) e *Queries* (operações que leem o estado). O ValorizeAI adota esse princípio através da separação explícita de *Actions* (Commands) e *Queries* (Queries), permitindo otimizações distintas para os caminhos de escrita e leitura.

### 3.2. Arquitetura e Componentes da Aplicação

A infraestrutura do ValorizeAI é composta por serviços gerenciados na nuvem, escolhidos por suas características de elasticidade e desempenho.

#### 3.2.1. Google Cloud Run e Cloud Tasks

O Google Cloud Run é uma plataforma de computação CaaS (Container-as-a-Service) totalmente gerenciada. Ele permite a execução de contêineres *stateless* que escalam horizontalmente de forma automática, com a capacidade de escalar até zero instâncias quando não há tráfego, eliminando custos ociosos [Google Cloud 2024]. Conforme discutido na Seção 2.1, o Cloud Run foi escolhido por combinar a elasticidade do FaaS com a flexibilidade dos contêineres, sendo capaz de executar tanto os serviços web *stateless* do ValorizeAI quanto o servidor *stateful* de WebSockets.

O Google Cloud Tasks é o serviço de enfileiramento de tarefas gerenciado. Ele é usado para implementar o processamento assíncrono (EDA), permitindo que a aplicação principal (síncrona) enfileire tarefas de longa duração (ex: processamento de lotes) para execução em *workers* separados, garantindo resiliência e baixa latência na resposta ao usuário.

#### 3.2.2. Laravel Reverb (WebSockets)

O Laravel Reverb é o servidor WebSocket oficial de primeira-parte para aplicações Laravel, projetado para comunicação em tempo real de alto desempenho [Laravel Holdings Inc. 2025]. Ele utiliza o protocolo Pusher, integrando-se nativamente ao sistema de *broadcasting* do Laravel para facilitar o envio de notificações *push* aos clientes conectados.

A característica arquitetural mais importante do Reverb para este TCC é seu suporte à escalabilidade horizontal. Para operar em um ambiente elástico como o Cloud Run (com múltiplas instâncias de servidor), o Reverb utiliza um *backplane* de mensagens, que no caso do ValorizeAI é implementado com o Redis (ver Seção 3.2.3) [Laravel Holdings Inc. 2025].

#### 3.2.3. Redis (Remote Dictionary Server)

O Redis (Remote Dictionary Server) é um armazenamento de estrutura de dados em memória, de código aberto, usado como banco de dados, *cache* e *message broker* [Kleppmann 2017]. No contexto da arquitetura ValorizeAI, o Redis desempenha dois papéis críticos e distintos, ambos fundamentais para o desempenho do sistema:

1. **Cache de Baixa Latência:** O Redis é usado como um *cache* para dados frequentemente acessados (ex: painéis, dados de sessão). Sua operação em memória permite latências de leitura e escrita na ordem de submilissegundos, reduzindo drasticamente a carga sobre o banco de dados PostgreSQL e melhorando a responsividade das *Queries* [Yadav 2019].
2. **Backplane Pub/Sub:** O Redis fornece um mecanismo de Publicação/Subscrição (Pub/Sub) de alto desempenho. Este mecanismo é utilizado como o *backplane* do Laravel Reverb. Quando uma instância do servidor (Instância A) precisa notificar um usuário que está conectado via WebSocket a outra instância (Instância B), a Instância A publica a mensagem em um canal Redis. Todas as outras instâncias, incluindo a Instância B, estão inscritas nesse canal, recebem a mensagem e a retransmitem aos seus clientes WebSocket conectados localmente.

### 3.3. Engenharia de Confiabilidade de Sites (SRE)

A metodologia de validação deste trabalho é baseada nos princípios de Engenharia de Confiabilidade de Sites (SRE), popularizados pelo Google [Beyer et al. 2016]. O SRE trata as operações de infraestrutura como um problema de engenharia de software, utilizando métricas rigorosas para equilibrar a inovação (velocidade de desenvolvimento) com a confiabilidade do serviço.

#### 3.3.1. SLIs, SLOs e Orçamentos de Erro

Os conceitos centrais do SRE utilizados para a validação do ValorizeAI são:

- **SLI (Service Level Indicator):** Um indicador de nível de serviço é uma medida quantitativa de um aspecto da qualidade do serviço fornecido [McCoy e Forsgren 2020]. Os SLIs são métricas diretas do desempenho do sistema, como latência de requisição, taxa de erro ou *throughput* do sistema [Beyer et al. 2016].
- **SLO (Service Level Objective):** Um objetivo de nível de serviço é um valor-alvo ou um intervalo de valores para um SLI, medido ao longo de um período [McCoy e Forsgren 2020]. Um SLO é a definição formal de "quão bom" o serviço precisa ser. Por exemplo, "95% das requisições de leitura (SLI: latência de leitura) devem ser concluídas em menos de 250ms (SLO) nos últimos 28 dias".
- **Orçamento de Erro (Error Budget):** O orçamento de erro é o complemento do SLO (ou seja,  $100\% - SLO\%$ ) [Beyer et al. 2016]. Ele representa a quantidade de falhas "permitidas" (ex: requisições lentas ou com erro) durante o período. O orçamento de erro é uma ferramenta de gerenciamento: enquanto houver orçamento, a equipe de desenvolvimento tem "permissão" para lançar novas funcionalidades (que inerentemente trazem risco); se o orçamento se esgotar, o foco da equipe deve mudar para a melhoria da confiabilidade [Beyer et al. 2016].

## 4. Metodologia

## 5. Resultados

## 6. Conclusão

### 6.1. Principais Contribuições

### 6.2. Resultados Alcançados

### 6.3. Limitações

### 6.4. Trabalhos Futuros

## 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. <https://queue-fair.com/website-scalability-for-large-traffic>, Acessado em: 2025-10-14.
- Basteri, A. (2023). What makes observability a priority. White Paper. <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. <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 (224). Event-driven architecture (eda): A complete introduction. <https://www.confluent.io/learn/event-driven-architecture/>, Acessado em: 2025-10-14.
- Datadog (2024). Serverless vs. containers: What's the difference? <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. Blog Post. <https://martinfowler.com/bliki/CQRS.html>, Acessado em: 2025-10-14.

- Google Cloud (2024). What is cloud elasticity? understanding elastic computing. <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.
- Laravel Holdings Inc. (2025). Laravel reverb official documentation. <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](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. <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.