UNIVERSIDADE REGIONAL DO NOROESTE DO ESTADO DO RIO GRANDE DO SUL — UNIJUÍ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

MARCOS JOSÉ KUCHAK FILHO

APLICAÇÃO DE CHAT EM TEMPO REAL NA SERVERLESS EDGE COMPUTING

IJUÍ
2022

MARCOS JOSÉ KUCHAK FILHO

Aplicação de Chat em Tempo Real na Serverless Edge Computing

Trabalho apresentado como um dos requisitos obrigatórios para aprovação na disciplina de Trabalho de Conclusão de Curso de Ciência da Computação da Universidade Regional do Noroeste do Estado do Rio Grande do Sul

Orientador: Prof. M.e Éder Paulo Pereira

Ijuí

2022

RESUMO

O trabalho trata da implementação de um sistema de chat em tempo real baseado na tecnologia Serverless Edge Computing. Esse sistema, diferente de outros do gênero, não utiliza a infraestrutura tradicional de computação em nuvem, o que traz inúmeras vantagens ao produto final, como a capacidade de gerenciar grandes volumes de tráfego, fornecer baixa latência e persistir os dados gerados próximos ao usuário. Para tanto, foram revisados trabalhos relacionados para identificar as principais soluções existentes para o problema a ser abordado, bem como seus principais limitadores. A partir daí, é apresentada a proposta da solução, detalhando todos os componentes do sistema e seus respectivos papéis. Em seguida, é realizada a implementação do sistema e, por fim, são apresentados os resultados alcançados, que se revelaram exitosos. Pudemos observar um sistema capaz de lidar com grandes volumes de dados em baixa latência. Além disso, o sistema pode ser facilmente escalonado para atender à demandas cada vez maiores, sem a necessidade de grandes investimentos em infraestrutura. Como resultado, o trabalho demonstrou a viabilidade de usar a tecnologia Serverless Edge Computing para a construção de um sistema de chat em tempo real, oferecendo uma solução dimensionável, de baixa latência e com baixo custo de operação.

Palavras-chave: Computação em Névoa. Computação de Borda. Aplicação em Tempo Real. Chat. Sem Servidor.

ABSTRACT

The work deals with the implementation of a real-time chat system based on Serverless Edge Computing technology. This system, unlike others of its kind, does not use traditional cloud computing infrastructure, which brings numerous advantages to the final product, such as the ability to manage large volumes of traffic, provide low latency and persist the generated data close to the user. To this end, related works were reviewed to identify the main existing solutions for the problem to be addressed, as well as their main limitations. From there, the solution proposal is presented, detailing all the system components and their respective roles. Then, the implementation of the system is carried out and, finally, the results achieved are presented, which proved to be successful. We were able to observe a system capable of handling large volumes of data at low latency. In addition, the system can be easily scaled to meet ever-increasing demands, without the need for major infrastructure investments. As a result, the work demonstrated the feasibility of using Serverless Edge Computing technology to build a real-time chat system, offering a scalable, low-latency and low-cost solution.

Keywords: Fog Computing. Edge Computing. Real Time Application. Chat. Serverless.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CDN	Content Delivery Network
CI/CD	Continuous Integration and Continuous Deployment
CLI	Command Line Interface
DO	Durable Objects
DOM	Document Object Model
DX	Developer Experience
EC2	Elastic Compute Cloud
HTML	Hypertext Markup Language
IaaS	Infrastructure as a Service
I/O	Input/Output
IoT	Internet of Things
JS	JavaScript
KV	Key-value
LTV	Lifetime Value
MEAN	MongoDB, Express.js, Angular.js, Node.js
MERN	MongoDB, Express.js, React, Node.js
PoP	Point of Presence
RAM	Random-access Memory
RDS	Relational Database Service
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol

TTL	Time-to-live
UI	User Interface
UX	User Experience
URL	Uniform Resource Locators
XML	Extensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

A computação de borda sem servidor é uma categoria de tecnologia recente que permite que as empresas executem seus aplicativos e serviços sem precisar gerenciar ou manter qualquer infraestrutura. Isso significa que as empresas podem economizar tempo e dinheiro ao não precisar se preocupar com manutenção, atualizações ou segurança de servidores.

Em um mundo tão conectado, muitas empresas estão procurando formas de melhorar o atendimento ao cliente. Uma maneira de fazer isso é usar aplicativos de bate-papo para permitir que os usuários se comuniquem com os atendentes em tempo real. Esse pode ser um ótimo artifício para melhorar a satisfação e a fidelidade do cliente, além de reduzir o tempo de espera e melhorar a eficiência nas vendas. Esse tipo de comunicação pode ser extremamente útil em cenários de atendimento ao cliente ou suporte. Por exemplo, se um cliente está tendo problemas para usar um recurso específico de um aplicativo, um atendente pode fornecer assistência rápida e fácil via chat.

Com a ascensão da computação de borda sem servidor, construir uma aplicação deste modelo tornou-se viável em larga escala e baixo gerenciamento de infraestrutura. Outro benefício da computação de borda sem servidor é a capacidade de lidar com grandes quantidades de dados com eficiência. Isso se deve ao fato de que a computação de borda sem servidor pode ser dimensionada com rapidez e eficácia para atender às demandas de um aplicativo em crescimento.

No geral, a computação de borda sem servidor é uma tecnologia emergente que oferece muitos benefícios para desenvolvedores e empresas. Portanto, os resultados deste trabalho de pesquisa e prototipação tornam-se úteis, pois poderão ser disponibilizados publicamente no futuro para benefício de engenheiros de *software* e pesquisadores.

Ademais, a Ciência da Computação é a área do conhecimento que atua na análise e desenvolvimento de programas para diferentes situações de uso, contemplando os mais diversos campos da Tecnologia da Informação. Neste sentido, o profissional de TI necessita ter competência em áreas como algoritmo, linguagem de programação, engenharia de *software*, banco de dados, interação humano-computador e segurança e auditoria de sistemas; esferas substanciais no desenvolvimento de aplicações e âmbito de atuação profissional.

O objetivo deste trabalho é investigar as vantagens do uso de Serverless na Edge Computing para a construção de um aplicativo de conversação capaz de lidar com a comunicação em tempo real entre dois ou mais usuários. Diante disso, a questão central desta pesquisa é a seguinte: a computação de borda sem servidor é uma opção viável para um aplicativo de chat em tempo real? Foram levantadas também questões secundárias, tais como:

1. Como os provedores de infraestrutura de borda sem servidor se comparam entre si?

2. Como a computação de borda sem servidor se compara à computação em nuvem tradicional em termos de desempenho, custo e escalabilidade?

3. Quais são as vantagens e desvantagens de usar a computação de borda sem servidor para um aplicativo de chat em tempo real?

Diferentemente de trabalhos como os de THARANIDHARAN et al. (2022), KUMAR; SINGH (2019), OBADJERE (2020), que focam sua concepção em arquiteturas tradicionais na nuvem, esse trabalho visa implementar um canal de comunicação de mensagens instantâneas em uma infraestrutura que não exige provisionamento, gerenciamento ou manutenção e permita comunicação entre clientes através de baixa latência na borda. Entre alguns provedores que fornecem esse serviço estão a Amazon Web Services, a Microsoft Azure, a Google Cloud Provider, a Fastly e a Cloudflare.

Dessa forma, para obter êxito neste trabalho, os seguintes objetivos foram traçados:

1. Inteirar-se dos principais serviços de infraestrutura e das principais bibliotecas, arquiteturas e tecnologias disponíveis para o desenvolvimento da aplicação a fim de identificar as melhores práticas de projeto e implementação;

2. Realizar a engenharia de requisitos da aplicação a ser desenvolvida utilizando técnicas orientadas à meta;

3. Elaborar o *wireframe* do *front-end* (cliente web) seguindo bons padrões de projeto para uma interface simples, porém responsiva;

4. Gerenciar e executar o processo de desenvolvimento seguindo princípios da metodologia ágil.

Seis partes compõem esse estudo. A Seção 1 discorre de forma geral sobre o problema abordado e a solução proposta. Na Seção 2 são descritas áreas de conhecimento, ferramentas, documentações e estudos acadêmicos relacionados. A Seção 3 define a metodologia utilizada para a elaboração do trabalho, e a Seção 4 explica o desenvolvimento da aplicação de chat em tempo real e os motivos das escolhas do autor. Finalmente, uma análise detalhada dos resultados, bem como considerações finais e ideias para expansão do trabalho são descritas nas Seções 5 e 6, respectivamente.

2 ESTADO DA ARTE

Esse capítulo tem por objetivo apresentar os principais conceitos abordados neste trabalho. Das Seções 2.1 a 2.5 são apresentados as áreas de conhecimento, os provedores de infraestrutura e as e ferramentas de *software* relacionados a aplicabilidade de *serverless* na computação de borda em relação ao tema deste trabalho.

Nas últimas décadas vários trabalhos foram publicados no campo de pesquisa da computação na nuvem, demonstrando as mais diversas aplicações práticas desta tecnologia. No âmbito de aplicações de bate-papo em tempo real, o mercado mostra que é uma prática adotada por uma variedade de empresas para prestar atendimento ao cliente. Ao utilizar esse tipo de aplicação, as companhias podem estabelecer um relacionamento mais próximo aos seus consumidores, aumentar a satisfação e a retenção na taxa de vendas. Os consumidores, da mesma forma, podem desfrutar de uma experiência mais agradável ao se conectar com a marca.

Além disso, as empresas buscam cada vez mais soluções inovadoras para melhorar a experiência do cliente e, em simultâneo, aumentar as vendas. É nesse contexto que o desenvolvimento de *chatbots* também está se tornando cada vez mais popular. Agentes artificiais podem atuar como interlocutores em conversas com humanos, auxiliando-os na obtenção de informações e esclarecimento de dúvidas, podendo também ser utilizados em áreas como marketing e educação.

Contudo, foi observado que a maior parte dos trabalhos relacionados a esse tema se baseia em conceitos contemporâneos, o que dificulta desenvolver, escalar e manter uma estrutura robusta capaz de atender muitos consumidores. A pesquisa deste trabalho traz uma abordagem diferente à computação em nuvem tradicional, sendo que o principal objetivo é entender a viabilidade desse tipo de aplicação em uma infraestrutura sem servidor na computação de borda.

2.1 Conceitos e Áreas de Conhecimento

Nessa Seção será apresentado a evolução teórica das diferentes categorias de provedores de infraestrutura e seus principais conceitos. Todas as áreas de conhecimento introduzidas aqui se conectam ou se complementam.

2.1.1 Computação em Nuvem

A computação em nuvem é um modelo de computação onde os recursos computacionais são entregues como um serviço, geralmente através da internet. Os usuários podem acessar esses recursos como um serviço (IaaS), sem se preocupar com a complexidade ou o gerenciamento dos recursos de computação, mas com a possibilidade de alocá-los entre máquinas virtualizadas.

Segundo BHOSALE (2021, tradução nossa):

> "(...) nuvem é a distribuição de serviços de computação - incluindo servidores, banco de dados, rede, armazenamento, *software*, análise e inteligência - pela internet ("a nuvem") para oferecer uma revolução mais rápida, recursos flexíveis e economias de escala. Trabalhar na nuvem é a entrega de serviços de computação, como servidores, armazenamento, banco de dados, rede, *software*, análise, inteligência e muito mais, pela nuvem (internet). A computação em nuvem oferece um *datacenter* diferente do local."

Organizações de todos os tipos e tamanhos usam a computação em nuvem para melhorar a produtividade, acelerar o tempo de lançamento no mercado e reduzir custos. Por exemplo, as empresas de saúde usam a nuvem para desenvolver tratamentos mais personalizados para os pacientes. As empresas de serviços financeiros usam a nuvem como base para detectar e prevenir fraudes em tempo real. E os fabricantes de videogames usam a nuvem para entregar jogos *online* a milhões de jogadores em todo o mundo.

A nuvem oferece às organizações acesso fácil a uma ampla gama de tecnologias para que possam inovar mais rapidamente e criar praticamente qualquer coisa que possam imaginar. Eles podem gerar recursos rapidamente conforme necessário, desde serviços de infraestrutura como computação, armazenamento e bancos de dados até a Internet das Coisas, aprendizado de máquina, *datalakes*, análise de dados e muito mais.

As organizações podem implantar serviços de tecnologia em minutos e passar da ideia à implementação com agilidade em velocidades muito maiores do que antes. Isso lhes dá a liberdade de experimentar, testar novas ideias para diferenciar as experiências dos clientes e transformar seus negócios. Ainda BHOSALE (2021, tradução nossa):

> "Do ponto de vista de provisionamento e preço de *hardware*, três aspectos são fundamentais na computação em nuvem, que são os seguintes: a chegada de recursos de computação ilimitados disponíveis sob demanda, suficientemente grande para acompanhar picos de carga, eliminando assim a necessidade de planejamento de usuários de computação em nuvem muito à frente para o provisionamento; a não necessidade de um compromisso inicial de locação, permitindo assim que as empresas comecem pouco e aumentem os recursos de *hardware* apenas quando houver um aumento em suas necessidades; a capacidade de pagar pelo uso de

> recursos de computação em curto prazo como necessários (por exemplo, processadores por hora e armazenamento por dia) e liberá-los conforme necessário, recompensando assim a conservação, deixando as máquinas e o armazenamento desocupados quando não forem úteis por muito tempo."

Com a computação em nuvem, as organizações não precisam provisionar recursos em excesso para absorver futuros picos de atividade em seus negócios. Em vez disso, eles provisionam a quantidade de recursos de que realmente precisam. Elas podem aumentar ou diminuir instantaneamente esses recursos para ajustar a capacidade à medida que suas necessidades de negócios evoluem.

A nuvem permite que as organizações troquem gastos de capital (*datacenters*, servidores físicos, etc.) por despesas variáveis e paguem apenas pela tecnologia consumida. Além disso, as despesas variáveis são muito menores do que as organizações pagariam por conta própria devido a economias em escala, manutenção de hardware, climatização para equipamentos, investimento em rede e retenção de profissionais qualificados.

Com a nuvem, as organizações podem escalar atividades para novas localidades e implantar aplicações globalmente em minutos. Por exemplo, muitos provedores têm infraestrutura em todo o mundo, o que permite que as organizações implantem aplicativos em vários locais físicos com apenas alguns cliques. Aproximar os aplicativos dos usuários finais reduz a latência e melhora a experiência deles.

2.1.2 Redes de Distribuição de Conteúdo

As redes de entrega de conteúdo (Content Delivery Networks) são uma parte crucial de qualquer aplicativo da web moderno. No passado, as CDNs apenas replicavam arquivos comumente solicitados (conteúdo estático) em um conjunto distribuído globalmente de servidores de *cache* para melhorar a entrega de conteúdo. No entanto, as CDNs se tornaram muito mais úteis ao longo do tempo.

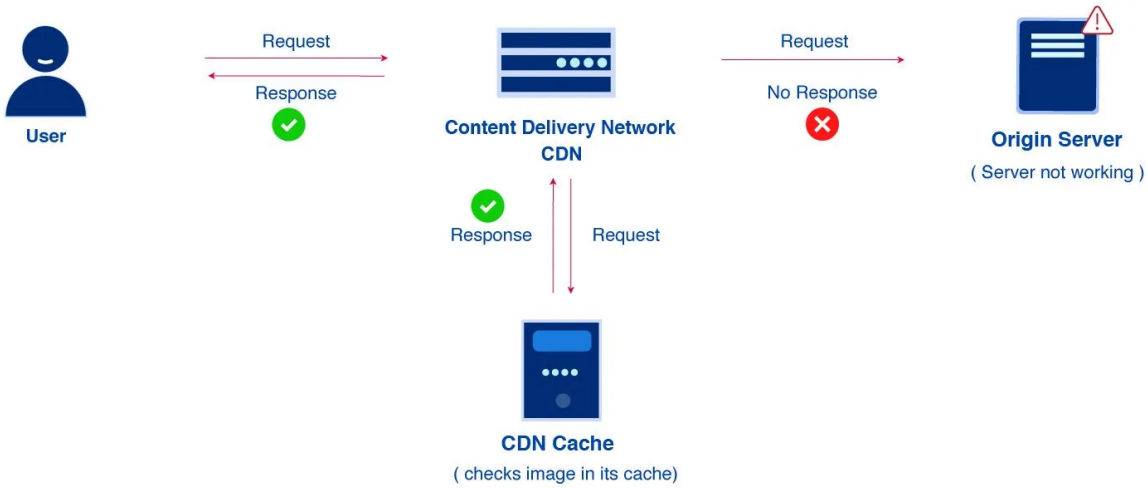
De acordo com GUPTA (2015, tradução nossa):

> "A Content Delivery Network é uma abordagem eficaz para aliviar o congestionamento na rede e nos servidores para melhorar a resposta aos usuários finais. Ele otimiza a entrega de conteúdo replicando o conteúdo em servidores substitutos colocados na borda da internet."

Para armazenamento em *cache*, uma CDN reduz a carga em uma origem de aplicativo e melhora a experiência do usuário, entregando uma cópia local do conteúdo a partir da borda, ou seja, do centro de entrega de *cache* mais próximo, ou ponto de presença (Point of Presence). A

origem do aplicativo — ou ainda, o servidor onde o processamento normalmente ocorre — não precisará abrir conexão e entregar o conteúdo para todas as requisições realizadas, pois a CDN cuidará disso. Como resultado, as origens do aplicativo não precisam ser dimensionadas para atender às demandas de conteúdo estático.

Figura 1 – A CDN entrega conteúdo de seu próprio *cache* enquanto o servidor de origem está inativo



Fonte: NANWANI (2021)

A CDN estabelece e mantém conexões seguras mais próximas do solicitante e, se a CDN estiver na mesma rede que a origem (como é o caso das CDNs baseadas em nuvem), o roteamento de volta à origem para recuperar o conteúdo dinâmico é extremamente rápido. Além disso, a CDN pode receber e enviar de volta conteúdo como dados de formulário, imagens e texto, aproveitando as conexões de baixa latência e o comportamento de *proxy* do PoP. Ao combinar a entrega de conteúdo estático e dinâmico, os clientes podem usar CDNs para fornecer interatividade e entrega próxima ao usuário de qualquer tipo de aplicativo web, o que é descrito por GUPTA (2015, tradução nossa):

> "O conceito de CDN baseia-se em colocar a réplica de conteúdos mais próxima dos utilizadores finais de forma a aumentar a escalabilidade, disponibilidade e acessibilidade dos conteúdos e como resultado melhorar o desempenho percebido pelo utilizador na recessão dos conteúdos solicitados. Ele entrega o conteúdo de forma transparente aos usuários finais em nome do servidor de origem. O algoritmo de redirecionamento de solicitação é usado para selecionar o melhor servidor de réplica e a solicitação do usuário é redirecionada para esse servidor. Uma CDN também oferece suporte para aprimorar o desempenho da Web durante

> o tráfego de intermitência. Também é usado para fornecer serviços de *streaming* de forma econômica e confiável."

Os *caches* também se tornaram muito mais inteligentes, fornecendo a capacidade de inspecionar as informações contidas no cabeçalho da solicitação e variar a resposta com base no tipo de dispositivo, informações do solicitante, *string* de consulta ou configurações de *cookies*. As CDNs podem ser direcionadas para recuperar objetos de várias fontes, impor políticas de protocolo, negociar conexões SSL e restringir o acesso a objetos com base na localização ou credenciais de autenticação.

Por fim, as CDNs também podem fornecer proteção no nível da rede e do aplicativo, evitando danos ou perda de serviço, filtrando o tráfego com *firewalls* de aplicativos da web e serviços de proteção contra DDoS integrados nos pontos de presença e na rede de armazenamento em *cache*. A combinação de controles de segurança e grandes quantidades de largura de banda de rede mantém *bots*, *scrapers* e *hackers* afastados sem comprometer a disponibilidade ou o desempenho do aplicativo.

2.1.3 Computação de Borda

A computação de borda é uma política de rede cujo objetivo é reduzir a latência e o uso da largura de banda, aproximando a computação da fonte de dados. Em outras palavras, a computação de borda, também conhecida como *edge computing* ou computação em névoa, tem como objetivo executar menos processamento na nuvem e mover esses processos para locais próximos, como no computador do próprio usuário, em um dispositivo IoT (Internet of Things) ou em um servidor de borda. Ao trazer a computação para a borda da rede, a quantidade de comunicação de longa distância que precisa acontecer entre um cliente e um servidor é minimizada. Conforme SULIEMAN (2022, tradução nossa):

> "A computação de borda é um paradigma de computação distribuída em que os dados do cliente são processados na periferia da rede, o mais próximo possível da fonte de origem. Desde que o século 21 passou a ser conhecido como o século dos dados (devido ao rápido aumento na quantidade de dados trocados em todo o mundo — especialmente em aplicações de cidades inteligentes, como veículos autônomos), coletando e processando esses dados de sensores e dispositivos de Internet das Coisas, operar em tempo real a partir de locais remotos e ambientes operacionais inóspitos em quase qualquer lugar do mundo é uma necessidade emergente relevante. De fato, a computação de borda está

> remodelando a tecnologia da informação e a computação empresarial."

Similarmente, PALLIS (2019, tradução nossa):

> "A computação de borda é um novo paradigma no qual os recursos de um servidor de borda são colocados na borda da internet, próximos a dispositivos móveis, sensores, usuários finais e a emergente IoT. Termos como "*cloudlets*", "micro *datacenters*" e "neblina" têm sido usados na literatura para se referir a esses tipos de *hardware* de computação pequeno e localizado na borda. Todos eles representam contrapontos ao tema da consolidação e dos *datacenters* massivos que dominaram o discurso na computação em nuvem."

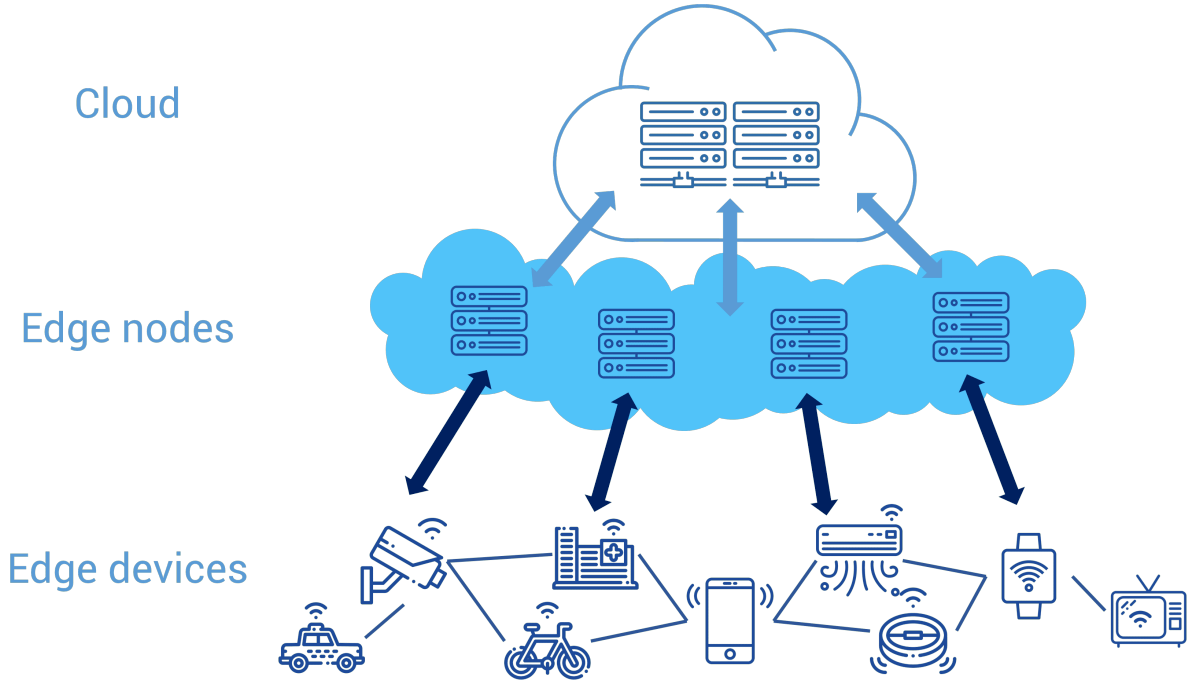
Assim, a *edge computing* pode ajudar a reduzir a quantidade de dados que precisam ser enviados para a nuvem, bem como a quantidade de poder de processamento e armazenamento necessários em servidores em nuvem. Isso é significativo, pois a quantidade de dados e dispositivos está aumentando continuamente, mas os recursos da nuvem são limitados e podem ser caros.

Um dos benefícios mais relevantes de mover processos para a borda é a redução da latência. Toda vez que um dispositivo precisa se comunicar com um servidor remoto em algum lugar, há um atraso. Por exemplo, quando dois colegas no mesmo escritório conversam por meio de um aplicativo de mensagens instantâneas, pode haver um atraso considerável porque cada mensagem deve ser roteada para fora do prédio, comunicada com um servidor em algum lugar do mundo e devolvida antes de aparecer na tela do destinatário. Se esse processo for movido para a borda e o roteador interno da empresa for responsável pelo roteamento das mensagens dentro do escritório, não haverá esse atraso perceptível. Da mesma forma, quando os usuários de outros tipos de aplicativos da web encontram processos que precisam se comunicar com um servidor externo, eles sofrem atrasos. A duração desses atrasos depende da largura de banda disponível e da localização do servidor, mas esses atrasos podem ser totalmente evitados movendo mais processos para a borda da rede.

A computação de borda tem o potencial de melhorar uma ampla gama de aplicativos, produtos e serviços (PRATT, 2021). Entre as possibilidades estão:

- *Cache* aprimorado: ao executar o código em uma rede de borda CDN, um aplicativo pode ajustar como o conteúdo é armazenado em *cache* e até personalizá-lo para fornecer conteúdo aos usuários com mais eficiência e relevância;

Figura 2 – A representação de névoa e sua comunicação entre servidores e dispositivos de borda



Fonte: AHMED (2021)

- Dispositivos de Internet das Coisas mais inteligentes: dispositivos de IoT podem se beneficiar da execução de código na borda ou no próprio dispositivo, e não na nuvem, para melhorar as interações do usuário. Isso torna os dispositivos mais inteligentes, pois eles podem processar informações e tomar decisões sem a necessidade de comunicação com um servidor;
- Dispositivos de monitoramento médico: os dispositivos médicos têm a capacidade de responder em tempo real sem ter que esperar pela resposta de um servidor mais distante na nuvem;
- Videoconferência: como o vídeo interativo ao vivo consome muita largura de banda, aproximar o processamento do *back-end* da fonte de vídeo pode reduzir atrasos e latência significativamente;
- Carros autônomos: veículos autônomos precisam ser capazes de responder em tempo real sem ter que esperar por instruções de um servidor; com essa tecnologia, esse problema é minimizado.

A computação em nuvem tradicional tem várias deficiências no cenário de IoT. Por exemplo, os aplicativos que estão inseridos no contexto de Internet das Coisas corriqueiramente têm requisitos em tempo real, e não estarem na borda pode causar graves problemas de latência. Ainda, a transmissão de grandes quantidades de dados gerados por dispositivos de borda para a nuvem em tempo real pode sobrecarregar a largura de banda da rede. Além disso, à medida que

mais e mais serviços de internet são implantados na nuvem, a disponibilidade desses serviços se tornou um fator importante a ser considerado, e uma camada entre cliente e servidor (a *edge computing*) pode ser benéfica ao garantir uma comunicação intermediária que supra demandas que não necessitam ir à nuvem.

Outro problema com a computação em nuvem é o consumo de energia – os *datacenters* consomem muita energia e, com o aumento da quantidade de computação e transmissão de dados, o gasto com energia se torna um gargalo que restringe o desenvolvimento de *datacenters* — não à toa muitos são instalados em locais remotos e gelados, como lugares próximos do Círculo Polar Ártico. Esse é o caso do Facebook, que construiu um de seus *datacenters* em Luleå, Suécia. O problema é tanto que a Microsoft, por exemplo, também já cogitou mergulhar servidores no mar para reduzir seus custos com a refrigeração (COSTA, 2016).

Por fim, a segurança e privacidade também são preocupações, pois os dados na interligação de milhares de domicílios estão intimamente relacionados à vida dos usuários. Isso pode incluir, por exemplo, câmeras internas transmitindo dados de vídeo da casa para a nuvem, o que aumenta o risco de vazamento de informações privadas e pode ser resolvido com o processamento na borda.

No geral, a computação de borda é uma tecnologia promissora que tem potencial para transformar muitas indústrias e setores de negócios, bem como tem o potencial de revolucionar a maneira como as empresas operam e fornecem serviços a seus clientes. Apesar dos muitos desafios que precisam ser resolvidos, como segurança de dados e posicionamento ideal dos pontos de presença (PoPs), os benefícios da implantação da computação de borda são claros. Ao mover o processamento e o armazenamento de dados para mais perto da borda da rede, as empresas podem melhorar seu desempenho enquanto atendem às expectativas de seus clientes com mais eficiência. Além disso, a computação de borda pode ajudar as empresas a reduzir sua dependência da nuvem, o que traz a possibilidade de reduzir custos e melhorar a segurança através dessa camada intermediária até a origem.

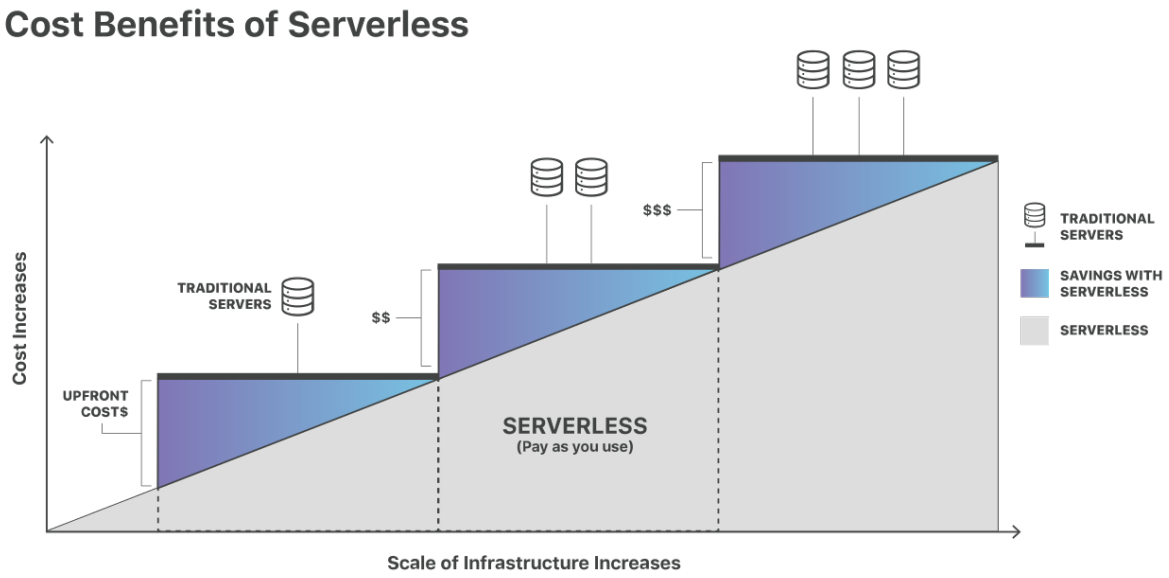
2.1.4 Computação Sem Servidor

A computação sem servidor é um modelo de computação em nuvem em execução no qual o provedor executa o servidor e aloca dinamicamente recursos de processamento sob demanda, em resposta a eventos e solicitações de entrada. Não há necessidade de um administrador de servidor para provisionar, dimensionar ou gerenciar quaisquer servidores. As tecnologias sem servidor apresentam dimensionamento automático, alta disponibilidade integrada e um modelo de cobrança com pagamento conforme o uso para aumentar a agilidade e otimizar os custos, sem pagar excessos.

O termo "sem servidor" refere-se a um método de fornecer serviços de *back-end* por uso. Assim, desenvolvedores podem criar e publicar código sem se preocupar com a infraestrutura subjacente. Como tudo é dimensionado automaticamente, uma organização que adquire serviços

de *back-end* de um provedor sem servidor é cobrada com base apenas em sua computação e não precisa reservar e pagar por uma quantidade fixa de largura de banda ou número de servidores. Na prática, servidores físicos ainda são usados (apesar da nomenclatura "*serverless*"), mesmo que desenvolvedores não precisem estar cientes deles.

Figura 3 – Exemplo de computação ociosa, que é desperdiçada na computação em nuvem tradicional



Fonte: CLOUDFLARE (2022d)

Organizações que alugavam unidades fixas de servidores, frequentemente adquiriam poder computacional adicional para garantir que um pico de tráfego ou atividade não excedesse as restrições mensais e causasse interrupções nos aplicativos (FEE, 2020). Isso significa que uma quantidade significativa de espaço de servidor pago geralmente era desperdiçada. Para resolver o problema, os fabricantes de nuvem ofereceram modelos de escalonamento automático, embora mesmo esse tipo de modelo possa ser muito caro se ocorrer um pico de atividade indesejado, como um ataque DDoS, ou o acesso em massa em uma promoção de evento. Essa é a principal razão para o surgimento da nova tecnologia.

Computação sem servidor, uma tecnologia que elimina tarefas como provisionamento de capacidade e segurança de servidores, permite com que as equipes de desenvolvimento possam se concentrar apenas em escrever códigos que atendam às regras de negócios de suas organizações, aumentando seu rendimento. Ela também pode contribuir na criação de aplicativos melhores; as principais vantagens são sua flexibilidade, escalabilidade e custo-benefício. Não há necessidade de um administrador de servidor, e é por esse motivo que *serverless* é uma maneira econômica de usar recursos de computação.

A computação nesse nicho também pode facilitar a criação de aplicativos melhores. Com tecnologias assim, é possível ir da ideia ao mercado mais rapidamente, eliminando a sobrecarga

operacional para que as equipes possam publicar projetos rapidamente, obter *feedback* e iterar para chegar ao mercado mais rapidamente com um produto novo. Por fim, os aplicativos desse porte são extremamente escaláveis e altamente personalizáveis, para que possam se adaptar às necessidades de cada tipo específico de projeto e tamanho de público mais rápido do que nunca. Entre tantos motivos para usar *serverless*, os que mais se destacam são (OKTA, 2021):

- É mais barato porque não precisa pagar por espaço não utilizado ou tempo de CPU;
- É mais fácil de dimensionar porque o fornecedor é quem cuida do provisionamento;
- É mais simples porque é possível criar pequenas funções para coisas diferentes — distribuindo responsabilidades;
- É mais rápido porque você pode implantar um novo código de maneira prática e ágil, bem como executar um processo de *rollback* para corrigir erros.

2.1.5 Computação de Borda Sem Servidor

A computação de borda sem servidor é um novo paradigma para computação em nuvem que promete revolucionar a maneira como implantamos e gerenciamos aplicativos. Serverless Edge Computing é um modelo de computação em nuvem no qual os aplicativos são implantados e gerenciados em um ambiente totalmente sem servidor, sem a necessidade de qualquer infraestrutura de servidor. Esse modelo permite que os desenvolvedores se concentrem em seu código e lógica de negócios, sem precisar se preocupar com o gerenciamento ou provisionamento do servidor (BANDAKKANAVAR, 2022). A computação de borda sem servidor é perfeita para aplicativos que são orientados a eventos ou têm uma carga de trabalho muito intensa. MEULEN (2018, tradução nossa), colaborador no Gartner, estima que a maior parte dos dados serão processados fora de *datacenter* e das *clouds* tradicionais em poucos anos:

> "Cerca de 10% dos dados gerados pela empresa são criados e processados fora de um *datacenter* ou nuvem centralizado tradicional. Até 2025, o Gartner prevê que esse número chegará a 75%"

A computação de borda sem servidor tem muitos benefícios em relação às implantações tradicionais baseadas em servidor. Primeiro, é muito mais barato, pois você não precisa pagar por nenhuma infraestrutura de servidor. Segundo, é muito mais fácil de gerenciar, pois não há servidores para provisionar ou manter. Terceiro, é muito mais escalável, pois o ambiente sem servidor pode aumentar ou diminuir automaticamente o seu aplicativo com base na demanda. Finalmente, é mais confiável, pois não há um único ponto de falha (KEMPF, 2021).

Sobre a evolução da CDN até a chegada da Serverless Edge Computing, ELLERBY (2021, tradução nossa) afirma:

> "As redes de distribuição de conteúdo representam a primeira onda de computação de borda. Com uma CDN, os dados são armazenados em vários pontos de presença mais próximos dos usuários, reduzindo o tempo para carregar e melhorando o desempenho. À medida que as CDNs se tornaram mais avançadas, a maturidade das tecnologias de virtualização permitiu que o código e o armazenamento saíssem dos limites da nuvem tradicional. Isso permitiu que os recursos de computação surgissem como um serviço nos pontos de presença. (...) O uso típico da computação de borda é para processamento de dados instantâneo e em tempo real. Ao contrário da nuvem, onde *big data* é o nome do jogo, há muito mais foco em dados instantâneos em tempo real à medida que avançamos para o Edge em conjunto com Serverless."

A computação de borda sem servidor é uma ferramenta poderosa que pode ser usada para gerenciar e monitorar aplicativos com uso intenso de dados. É uma maneira econômica de melhorar o desempenho e a escalabilidade do aplicativo. Estas são algumas das principais vantagens de usar o Serverless Edge Computing (ELLERBY, 2021):

- Melhor desempenho do aplicativo: pode ajudar a melhorar o desempenho do aplicativo, reduzindo a necessidade de enviar e receber dados entre o servidor e o cliente. Isso pode ajudar a melhorar os tempos de resposta e reduzir a latência.
- Maior escalabilidade: pode ajudar a aumentar a escalabilidade de um aplicativo, permitindo que ele seja distribuído por vários servidores intermediários. Isso pode ajudar a reduzir o risco de sobrecarregar um único servidor e melhorar o desempenho geral do aplicativo.
- Econômico: pode ser uma maneira econômica de melhorar infraestrutura do aplicativo. Ele pode ajudar a reduzir a necessidade de *hardware* caro e pode ser usado para pagar apenas os recursos usados.
- Flexível: é uma solução flexível que pode ser usada para gerenciar e monitorar uma ampla variedade de aplicativos com uso intenso de dados. Ele pode ser facilmente personalizado para atender às necessidades específicas de uma organização e maleável a ponto de ajustar-se à demanda.
- Fácil de usar: é uma solução fácil de usar que pode servir para gerenciar e monitorar aplicativos com uso intenso de dados.

O que difere a CDN da Serverless Edge Computing é a capacidade de executar a lógica na borda, e não apenas repassar conteúdo estático. Isso abre um conjunto enorme de possibilidades, já que é possível fazer um *proxy* com o servidor de origem e repassar apenas as solicitações que realmente são necessárias, e não apenas criar uma camada de *cache* entre servidores. O que pode ser ainda mais disruptivo é a capacidade de processar dados de maneira intensa, transferindo a necessidade de um servidor de origem para que tudo seja realizado na borda. Em outros termos, agora podemos criar um serviço de *back-end* completamente na borda, próximo aos usuários da aplicação e com latência mínima.

2.2 Sistemas de Chat em Tempo Real

O termo chat, ou sala de bate-papo, é usado principalmente para descrever uma forma de conferência síncrona. Ocasionalmente também é entendido como conferência assíncrona, portanto, o termo pode significar distintas tecnologias, desde bate-papo em tempo real e interação com estranhos (por exemplo, fóruns *online*) até ambientes sociais gráficos totalmente imersivos como o novo metaverso. Com o aumento da popularidade da internet nos anos 2000, também houve uma explosão de sistemas de salas de bate-papo *online*. Foi assim que surgiu uma gama de serviços de comunicação como IRC, MSN, ICQ e afins. Esses serviços permitiam que os usuários se conectassem a um ou mais servidores onde podiam conversar com outros usuários em "salas virtuais" públicas ou privadas.

Mais recentemente, novos aplicativos surgiram, como o WhatsApp, Facebook Messenger, Telegram, Signal, Discord, Slack, e tantos outros. E, embora a maioria desses aplicativos sejam feitos especificamente para telefones celulares, é possível usá-los em um computador através de um navegador ou através de um aplicativo *desktop* específico.

Ainda, em sistemas comerciais, essa tecnologia pode ser explorada de múltiplas maneiras. Um sistema desse gênero pode ser utilizado para diversos fins, desde prestar atendimento, responder a perguntas, resolver problemas e construir relacionamentos.

No âmbito comercial, há muitos benefícios de usar um sistema de chat ao vivo (SILVA, 2021), incluindo:

1. Maior satisfação do cliente

   Um dos benefícios mais importantes de usar um sistema de chat ao vivo é que ele pode ajudar a aumentar a satisfação dos clientes. Quando os clientes têm suas dúvidas respondidas em tempo real, é mais provável que fiquem satisfeitos com a experiência.

2. Aumento de vendas

   Outro benefício de usar um sistema de chat ao vivo é que ele pode ajudar a aumentar as vendas. Muitas vezes, os clientes usam o sistema de bate-papo para fazer perguntas sobre

produtos ou serviços antes de fazer uma compra. Ao fornecer respostas e orientações, você pode ajudar a aumentar a probabilidade de que eles façam uma compra.

3. Melhor retenção de clientes

   Mais um ganho através do uso de sistemas de troca de mensagens, é que eles podem ajudar a melhorar a retenção de clientes. Quando os clientes têm uma experiência positiva com a marca, é mais provável que continuem fazendo negócios. Também é possível aplicar campanhas de marketing e *remarketing*, aumentando o LTV (Lifetime Value), uma estimativa da receita média que um cliente irá gerar ao longo de sua vida como consumidor da empresa.

4. Custos reduzidos de suporte ao cliente

   Ainda, um aplicativo de comunicação pode ajudar a reduzir os custos de suporte ao cliente. Quando há a possibilidade de responder às perguntas dos clientes em tempo real, pode-se evitar a necessidade de suporte telefônico dispendioso.

5. Aumento da fidelidade do cliente

   Por fim, outra vantagem de usar um sistema de chat ao vivo é que ele pode ajudar a aumentar a fidelidade do cliente. Quando os clientes têm suas dúvidas respondidas de forma rápida e eficiente, é mais provável que permaneçam fiéis à sua empresa.

De maneira geral, há muitas vantagens em usar um sistema de chat em tempo real. Ao fornecer suporte ao cliente ao vivo, é possível aumentar a satisfação, as vendas, a retenção e a fidelidade. Além disso, reduzir os custos de suporte ao cliente também torna-se factível (BOGU, 2021). Com tantos benesses, usar um sistema de bate-papo ao vivo é algo que deve ser considerado por empresas de varejo online.

Os sistemas de chat em tempo real normalmente são construídos em uma infraestrutura tradicional de cliente-servidor, onde o servidor mantém o estado do chat e envia eventos para os clientes, que o respondem. Os chats tradicionais geralmente mantém o estado do chat em um servidor central. Qualquer nova mensagem ou evento enviado para o servidor é propagado para todos os clientes conectados, incluindo o cliente que enviou a mensagem original. Devido a estas particularidades, esses sistemas geralmente são mais difíceis de escalar e também podem ser caros para manter.

Uma alternativa é o chat baseado na Serverless Edge Computing. Nesse tipo de rede, as mensagens são enviadas diretamente pelos clientes para um ponto de presença, sem envolver um servidor central de origem. Isso pode permitir que uma aplicação de chat seja implantada e distribuída em diversos servidores mais próximos dos usuários, em uma escala que seria impossível usando apenas um servidor central. Isso pode ser especialmente útil, já que uma aplicação deste porte pode se beneficiar da latência reduzida e do custo minimizado de uma implantação na borda da internet.

2.2.1 Browsers e Websockets

Os navegadores — também conhecidos como browsers — são *softwares* que permitem aos usuários acessar a internet, visualizar e interagir com o conteúdo *online*, incluindo páginas da web, vídeos e imagens. O termo "navegador" foi originalmente usado para descrever interfaces de usuário baseadas em texto que permitiam aos usuários navegar por arquivos de texto *online*. Hoje, os navegadores web são amplamente utilizados para acessar a internet e são vistos como uma ferramenta essencial para muitas pessoas em seu dia a dia.

O primeiro navegador foi criado em 1990 e se chamava WorldWideWeb. Já o primeiro navegador popular foi o Netscape Navigator, lançado em 1994. A Microsoft lançou o Internet Explorer em 1995 e dominou o mercado até o ano de 2005. O navegador mais popular hoje em dia, o Chrome, foi lançado pela Google em 2008. Ele é baseado no projeto de código aberto Chromium. O Google desenvolveu o Chrome porque queria criar um navegador rápido, simples e seguro.

Os navegadores da web funcionam em um modelo cliente/servidor. O cliente é o navegador, que roda no dispositivo do usuário, e faz requisições ao servidor web. O servidor então envia informações de volta ao navegador, que então interpreta e exibe uma interface no dispositivo do usuário.

Eles são compostos de várias partes interfuncionais, incluindo a interface do usuário (UI), o mecanismo do navegador, o mecanismo de renderização, a rede, um interpretador JavaScript e o *back-end* da interface do usuário. A UI é o nível em que o usuário interage com o navegador. O mecanismo do navegador consulta o mecanismo de renderização, que renderiza a página da web solicitada, interpretando os documentos HTML ou XML. A rede lida com a segurança e a comunicação na internet. O interpretador JavaScript é usado para interpretar e executar código JavaScript em um site. O *back-end* da interface do usuário é usado para criar *widgets*, como janelas. Além disso, uma camada de persistência, chamada *local storage* ou armazenamento de dados, gerencia dados, como marcadores, *caches* e *cookies*.

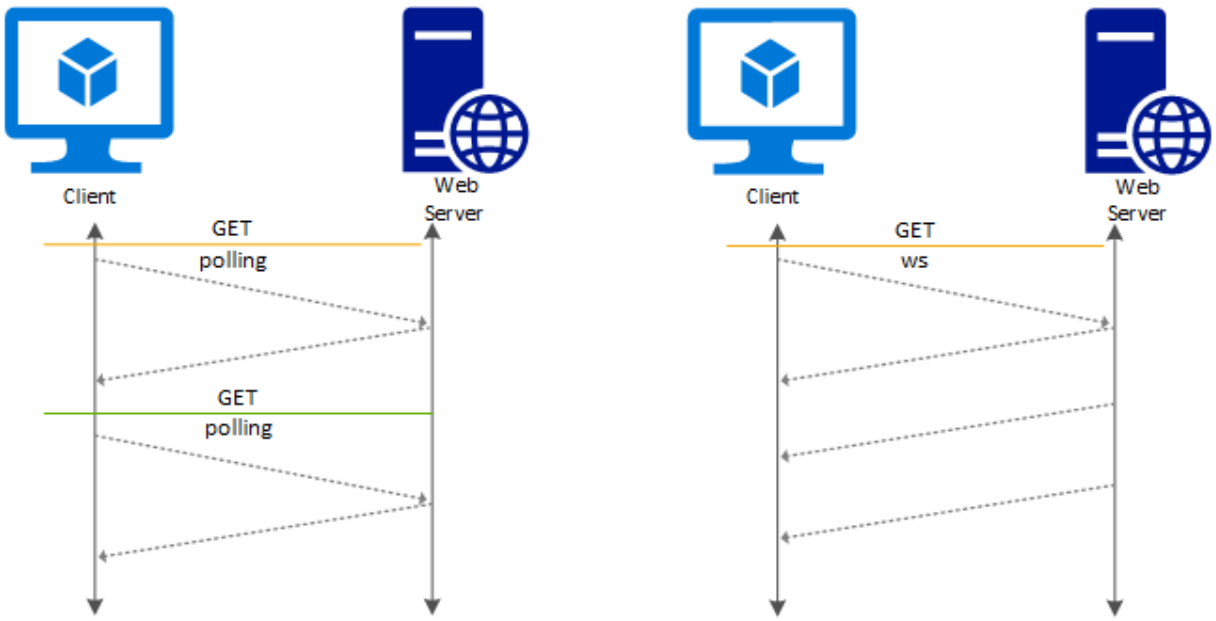
Existem alguns motivos pelos quais é melhor criar um produto para a web em vez de um aplicativo para dispositivos móveis. Primeiro, a web é mais acessível do que os aplicativos móveis. Qualquer pessoa pode acessar a web de qualquer dispositivo, independentemente de ter um aplicativo específico instalado. Em segundo lugar, a web é mais pesquisável do que os aplicativos móveis. Quando as pessoas estão procurando algo, é mais provável que procurem na web do que tentem encontrar um aplicativo específico para determinado fim.

A web é mais flexível do que os aplicativos móveis, é mais fácil alterar e atualizar um aplicativo web do que alterar e atualizar um aplicativo móvel que é distribuído e instalado em milhares de dispositivos, ao contrário da web, que é requisitado a partir de um endereço na internet e então é carregado. Por fim, navegadores tendem a ser mais seguros do que aplicações nativas, pois são protegidos por *sandbox* e têm menos privilégios por padrão.

WebSockets são um divisor de águas para aplicativos da web. Essa é uma tecnologia que permite a comunicação bidirecional entre um navegador da web e um servidor, oferecendo uma maneira padronizada de lidar com a transferência de dados em tempo real entre um navegador da web e um servidor da web.

O WebSocket foi projetado para ser usado em aplicativos da web onde uma grande quantidade de dados precisa ser transferida entre o navegador e o servidor — onde os dados precisam ser processados em tempo real. Eles fornecem um canal de comunicação bidirecional *full-duplex* em um único soquete TCP ou são implementados através do protocolo WebSocket-Over-HTTP, um protocolo simples baseado em texto para *gateway* entre um cliente WebSocket e um servidor HTTP convencional (SOOKOCHEFF, 2019). Isso significa que os dados podem fluir entre o navegador e o servidor sem precisar pesquisar constantemente novos dados.

Figura 4 – Como o WebSocket funciona em comparação as solicitações HTTP habituais



Fonte: Lindsay (2022)

Isso torna a criação de aplicativos em tempo real muito mais fácil. Por exemplo, um aplicativo de bate-papo agora pode ser facilmente construído usando WebSockets. O servidor pode enviar mensagens para o navegador à medida que elas chegam, e o navegador pode exibi-las sem precisar verificar periodicamente se há novos dados.

WebSockets também são muito mais eficientes do que os métodos tradicionais de pesquisa. Com o *polling*, o navegador precisa enviar uma solicitação ao servidor em intervalos regulares, mesmo que não haja novos dados para recuperar. Isso pode colocar muita pressão desnecessária no servidor.

Com WebSockets, a conexão só é estabelecida quando há dados a serem trocados. Isso significa que o servidor só precisa lidar com as solicitações quando realmente há algo a ser feito, o que é muito mais eficiente.

2.2.2 Linguagem de Desenvolvimento

Para desenvolver um aplicativo web completo, é importante ter uma forte compreensão de todas as diferentes tecnologias envolvidas no processo de desenvolvimento de *software*. Uma dessas tecnologias é a linguagem de programação JavaScript. Embora o JavaScript não seja a única linguagem que pode ser usada para desenvolver aplicativos da web, é uma linguagem que se adéqua à tarefa.

JavaScript é uma linguagem versátil que pode ser usada para criar uma ampla variedade de aplicativos. Um dos benefícios é que é uma linguagem que pode ser utilizada tanto no *front-end* quanto do *back-end* das aplicações. Além disso, existem muitas bibliotecas e estruturas diferentes disponíveis que podem ser usadas para estender seus recursos como módulos que são importados e exportados.

Outra razão para escolher JavaScript para desenvolvimento web é que é uma linguagem muito popular. Isso significa que há uma grande comunidade de desenvolvedores que podem fornecer suporte e assistência. Há também muitos recursos disponíveis que podem ser usados em seu ecossistema, como *frameworks* com convenções de arquitetura estabelecidas para acelerar o desenvolvimento por completo.

Ademais, antigamente o JavaScript era executado apenas em navegadores, mas hoje em dia também é possível executá-lo no lado do servidor, favorecendo o desenvolvimento *back-end*. Existem alguns *runtimes* para o JavaScript no lado do servidor, como o Node.js, uma derivação do V8, que é um mecanismo criado pelo Google e usado pelo Chrome e outros navegadores. O Node.js é bastante rápido porque usa um modelo de I/O sem bloqueios baseado em eventos. Isso significa que ele pode lidar com mais solicitações simultâneas.

Além do Node.js, outros interpretadores de JavaScript, como Deno e Cloudflarer Workers (baseados nos Service Workers dos navegadores) — já anunciado que seu código-fonte será de código aberto para que qualquer pessoa possa revisar e contribuir — são frequentemente usados para executar JavaScript fora do navegador.

Essa capacidade de executar instruções no cliente e no servidor oferece um grande poder de compartilhamento e reutilização de código. Com interpretadores JS desse tipo, é possível escrever código uma vez e executá-lo no servidor e no cliente, usando a mesma estrutura e a mesma linguagem. Isso traz um enorme poder para criar aplicativos abrangentes e altamente interativos que podem se conectar e trabalhar com dados facilmente.

2.2.3 Persistência de Estado

Além das camadas de *front-end* e *back-end*, temos a camada de persistência de estado, normalmente gerida por um banco de dados, que é onde os dados são realmente armazenados. A persistência de dados é o processo de armazenamento de dados em um banco de dados para que possam ser recuperados posteriormente. Os dois tipos mais comuns de bancos de dados são

bancos de dados relacionais e bancos de dados não relacionais.

De acordo com LUTKEVICH (2021, tradução nossa):

> Um banco de dados é um conjunto de informação que é armazenada e configurada para ser acessada, gerenciada e atualizada facilmente. Os bancos de dados de computador normalmente armazenam agregações de registros de dados ou arquivos que contêm informações, como transações de vendas, dados de clientes, informações financeiras e de produtos. Os bancos de dados são usados para armazenar, manter e acessar qualquer tipo de dados. Eles coletam informações sobre pessoas, lugares ou coisas. Essas informações são reunidas em um só lugar para que possam ser observadas e analisadas. Os bancos de dados podem ser pensados como uma coleção organizada de informações.

Bancos de dados relacionais, como MySQL e Postgres, armazenam dados em tabelas com linhas e colunas. Eles são fáceis de dimensionar verticalmente, o que significa adicionar mais recursos de *hardware* a um único servidor, mas são mais difíceis de dimensionar horizontalmente, o que representa a necessidade de adicionar mais servidores à medida que o banco de dados cresce e tem mais acessos.

Um dos maiores desafios no dimensionamento de uma aplicação é dimensionar o banco de dados. Isso ocorre porque o banco de dados normalmente é o principal gargalo no aplicativo, ou seja, é a parte mais lenta e computacionalmente custosa. Existem algumas maneiras de dimensionar um banco de dados, mas geralmente é um processo difícil e caro.

Comumente, a melhor maneira de dimensionar um banco de dados é transformar aplicativos monolitos em microsserviços. Isto é, dividir o aplicativo em partes menores que podem ser executadas sem interferência direta das demais partes. Dessa forma, cada microsserviço pode ter seu próprio banco de dados, menor e mais fácil de dimensionar. No entanto, essa abordagem nem sempre é possível e, às vezes, não é a melhor solução.

Outra abordagem, é adotar o uso de um banco de dados não relacional NoSQL, como MongoDB, por exemplo. Esse tipo de banco armazena dados em documentos sem esquema predefinido. Os bancos de dados NoSQL são projetados para serem escaláveis e podem lidar com uma grande quantidade de dados. Porém, eles geralmente são mais complexos de usar e não são tão adequados para alguns aplicativos quanto os bancos de dados relacionais.

Os bancos de dados relacionais são o tipo mais comum de banco de dados e geralmente não são tão fáceis de escalar quanto os bancos de dados NoSQL. Contudo, existem, ainda, outras maneiras de realizar esta tarefa. Uma maneira é usar o *sharding*, o que significa dividir os dados

em partes menores (isto é, dividir o próprio banco, e não a aplicação) e distribuí-los por vários servidores. Outra maneira é usar a replicação, o que representa copiar os dados para vários servidores diferentes para realizar a leitura horizontalmente, deixando a escrita para um servidor central com maior poder de *hardware* vertical.

Ambas as abordagens podem ser difíceis de implementar e podem ser caras. Além disso, podem introduzir novos problemas, como inconsistência de dados. Como resultado, muitas vezes é difícil dimensionar um banco de dados relacional. Isso é especialmente um problema para a computação de borda, onde os dados podem ser espalhados por diferentes pontos de presença. Quando os dados precisam ser consistentes, pode ser mais difícil dimensionar porque cada servidor pode ter sua própria versão dos dados.

2.3 Ambientes de Desenvolvimento

Existem vários ambientes de desenvolvimento para trabalhar com aplicações web em diferentes linguagens. Para fins de contextualização, será realizada aqui uma breve comparação entre ambientes para a linguagem JavaScript. A principal comparação será entre Neovim, Sublime Text, WebStorm e Visual Studio Code.

Neovim é um editor de texto gratuito e de código aberto com suporte a Vimscript, uma linguagem de script com o propósito de modificar e estender o editor, criando novas ferramentas, simplificando tarefas comuns e melhorando recursos já existentes. Ele é um clone do antigo editor Vim, porém é focado na extensibilidade e usabilidade. A grande vantagem é que o Neovim é executado diretamente no emulador de terminal integrado, o que encurta tarefas devido a facilidade de executar comandos ao mesmo tempo em que edita texto (NEOVIM, 2022).

O Sublime Text é um editor de texto multiplataforma com licença proprietária também muito eficiente. Uma das maiores vantagens é como ele oferece aos usuários muitos atalhos e recursos que facilitam o trabalho para os desenvolvedores (SUBLIME HQ, 2022). Com o Sublime Text, é possível acessar quase todos os recursos usando apenas o teclado.

WebStorm é um IDE para desenvolvimento JavaScript da JetBrains. Ele traz um ambiente de desenvolvimento integrado para JavaScript e tecnologias relacionadas, fornecendo recursos que tornam a experiência de desenvolvimento mais agradável, automatizando o trabalho de rotina e ajudando a lidar com tarefas complexas com maior facilidade (JETBRAINS, 2022). O IDE executa dezenas de inspeções de código enquanto se digita e detecta possíveis problemas para que os resultados no trabalho sejam mais confiáveis. Ele ainda ajuda a refatorar toda a base de código com apenas alguns cliques, aproveitando a codificação produtiva com tudo o que é necessário para o desenvolvimento web. O WebStorm também simplifica tarefas desafiadoras, como trabalhar com Git e renomear componentes em todo o projeto, assim é possível se concentrar no quadro geral, e não em pequenas tarefas de rotina.

O Visual Studio Code é um editor de texto multiplataforma da Microsoft com vasto

suporte para extensões. É atualmente o ambiente de desenvolvimento mais amplamente utilizado no mundo (STACKOVERFLOW, 2021), oferecendo suporte a uma grande variedade de linguagens, incluindo JavaScript, bem como integração com ferramentas de depuração, versionamento de código e extensões fornecidas por terceiros ou pela própria mantenedora.

Por padrão, o Visual Studio Code possui um editor de código-fonte muito rápido, excelente para o uso diário. O VSCode oferece *feedback* instantâneo, destacando trechos escritos, parênteses, fornecendo indentação automática, caixas de seleção, *snippets* e muito mais. Atalhos intuitivos, fácil personalização e atalhos de acesso às múltiplas guias facilitam a navegação no momento da edição de código (MICROSOFT, 2021).

Para obter uma boa produtividade é necessário usar ferramentas que interpretam o código e vão além do entendimento de apenas blocos de texto. O Visual Code Studio *IntelliSense* oferece suporte de autocompletar código, compreensão de código semântico, refatoração de código e depuração de *hardware* em caso de problemas de codificação.

Todos os editores comparados aqui têm suporte para realce de código e sintaxe. Neovim e Sublime Text têm suporte básico para depuração, enquanto WebStorm e Visual Studio Code têm recursos de depuração mais abrangentes. O WebStorm também possui suporte integrado para trabalhar com estruturas JavaScript populares, como o *framework* Angular e a biblioteca React.

O Visual Studio Code e Neovim são os únicos editores totalmente gratuitos e de código aberto. Sublime Text pode ser baixado e avaliado gratuitamente, mas possui uma taxa para adquirir sua licença e continuar sua utilização depois de um período experimental. WebStorm é um IDE pago e também possui um período de avaliação de 30 dias.

Em termos de recursos, todos os editores comparados aqui oferecem um bom conjunto de ferramentas para trabalhar com aplicativos da web. No entanto, o Visual Studio Code e o WebStorm parecem oferecer um conjunto mais abrangente de recursos do que o Neovim e o Sublime Text.

2.4 Provedores de Infraestrutura

Visto que o objetivo principal do trabalho é criar um sistema de chat na *serverless edge computing*, essa Seção irá abordar algumas das possibilidades para a criação desse tipo de sistema no quesito infraestrutura. Existem muitos provedores de infraestrutura de computação de borda sem servidor, mas alguns dos mais populares incluem Amazon Web Services (AWS), Microsoft Azure e Google Cloud Platform (GCP).

Dois outros provedores que vêm se destacando são Fastly e Cloudflare com suas tecnologias de *edge computing* mais maduras que as concorrentes. Cada um desses provedores oferece uma variedade de serviços e recursos que podem ser usados para criar aplicativos sem servidor

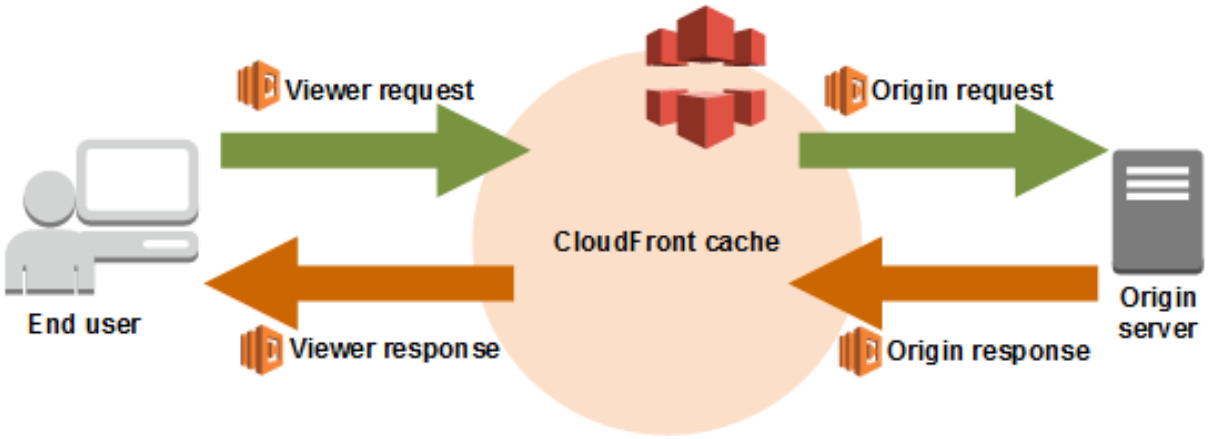
na borda. Analisaremos as três opções nesse nicho mais bem estabelecidas comercialmente: AWS, Fastly e Cloudflare.

2.4.1 AWS Lambda@Edge

Lambda@Edge é um recurso do Amazon Web Services CloudFront que permite que o código seja executado mais próximo dos usuários de um aplicativo, o que melhora o desempenho e reduz a latência. Com esse mecanismo, não há necessidade de provisionar ou gerenciar infraestrutura em vários locais ao redor do mundo. Dessa forma, paga-se apenas pelo tempo de computação que é consumido — não há cobrança quando o código não está em execução.

O Lambda@Edge pode ser usado para melhorar os aplicativos da web, tornando-os distribuídos globalmente e melhorando seu desempenho — tudo sem administração de nenhum servidor. Essa tecnologia executa seu código em resposta a eventos gerados pela rede de entrega de conteúdo (CDN) do Amazon CloudFront. O código é simplesmente carregado no AWS Lambda, que cuida de tudo o que é necessário para executar e dimensionar a aplicação com alta disponibilidade em um local da AWS mais próximo do usuário final.

Figura 5 – Diagrama de requisição de um usuário passando por um ponto de presença e, se necessário, passando pela origem



Fonte: AWS (2022c)

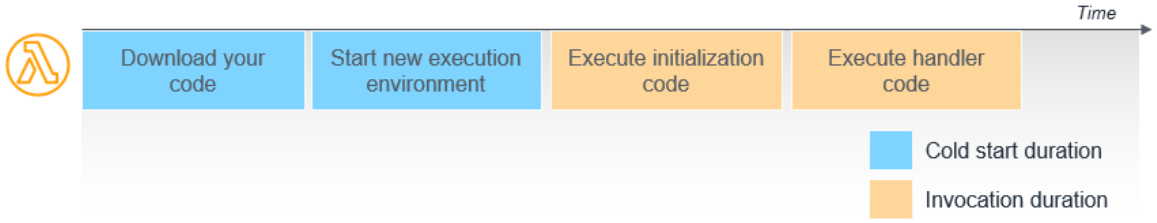
São mais de 225 pontos de presença distribuídos pela América do Norte, América do Sul, Europa, Ásia e África em 47 diferentes países (CHAN, 2021). Por isso, essa solução de alta disponibilidade e baixa latência fornece respostas rápidas aos usuários, independentemente de sua localização.

O Lambda@Edge permite executar código em JavaScript a partir do *runtime* Node.js e Python em suas versões mais novas (3.7, 3.8 e 3.9). Isso é feito executando as funções, que são armazenadas no bucket da Amazon S3, e distribuídas e invocadas a partir do ponto de presença mais próximo ao invocador.

Devido a essa implementação de arquitetura, em que todo o código é armazenado e distribuído em diferentes *buckets*, as funções *serverless* da AWS sofrem com um atraso em

sua primeira invocação — isso é chamado de Cold Start (ou inicialização a frio). A duração de um Cold Start varia de menos de 100 ms a mais de 1 segundo (BESWICK, 2021). Como o serviço Lambda reutiliza ambientes "aquecidos" para invocações subsequentes, as inicializações a frio são normalmente mais comuns em funções de desenvolvimento e teste do que em cargas de trabalho de produção. Entretanto, isso pode ser prejudicial para serviços que são invocados eventualmente e precisam ser processados rapidamente — justamente o propósito da *edge computing*.

Figura 6 – Ciclo de invocação do início ao fim de uma função na Lambda@Edge



Fonte: BESWICK (2021)

2.4.2 Fastly Compute@Edge

O Fastly Compute@Edge é uma plataforma sem servidor que facilita a implantação e o gerenciamento de aplicativos na borda. O Compute@Edge é construído sobre a rede de borda da Fastly, que possui mais de 100 pontos de presença em todo o mundo. Isso dá aos desenvolvedores a capacidade de implantar aplicativos mais próximos dos usuários, o que pode melhorar o desempenho e reduzir a latência. Fastly Compute@Edge é uma plataforma totalmente auto-gerenciada, para que os desenvolvedores não precisem se preocupar com provisionamento ou manutenção de servidores. O Fastly cuida de todo o gerenciamento, dimensionamento e segurança da infraestrutura. Em sua documentação, FASTLY afirma (2022b, tradução nossa):

> "Compute@Edge é uma plataforma de computação que pode executar binários personalizados que você compila em seus próprios sistemas e carrega no Fastly. A segurança e a portabilidade são fornecidas compilando seu código para WebAssembly, que executamos na borda usando o Lucet, um *runtime* do WebAssembly de código aberto criado pela Fastly."

Por ser uma plataforma baseada na WebAssembly System Interface (WASI), é possível escrever aplicações em qualquer linguagem compatível com WASI. Assim, linguagens como C/C++, C#, .NET, Rust, Java, Python, Elixir e Go, além da tradicional JavaScript, também são suportadas. Por causa disso, esse provedor não é capaz de executar módulos de plataformas

nativas de seus respectivos *runtimes*, deixando de fora alguns recursos úteis como, por exemplo, acessar o sistema de arquivos.

No geral, a Fastly oferece uma boa opção para desenvolver aplicativos na borda. Sua plataforma é rápida, é espalhada por pontos de presença bem distribuídos — apesar da quantidade ainda limitada — e, devido sua implementação, não há inicialização a frio ou atrasos de ida e volta: apenas computação rápida e sempre ativa.

Entretanto, a Fastly ainda carece de mais instruções sobre seu produto em sua documentação, o que pode ser um desafio para iniciantes. A Fastly talvez não ofereça uma plataforma que esteja pronta para todos. Enquanto ela é ideal para desenvolvedores experientes, iniciantes terão dificuldades para configurar seu primeiro serviço. Além disso, a plataforma ainda demanda de alguns recursos, como uma interface de usuário gráfica e um painel de gerenciamento mais intuitivos.

2.4.3 Cloudflare Workers

Semelhante a Fastly, a Cloudflare também desenvolveu seu próprio motor de tempo de execução para aplicações de borda. Enquanto a Fastly usou um *runtime* do WebAssemlby, a Cloudflare optou por utilizar o V8 do projeto de código aberto Chromium (assim como o Node.js). Isso significa que o Cloudflare Workers é capaz de executar scripts JavaScript de seus clientes com segurança em seus servidores da mesma maneira que o Chrome executa scripts de várias aplicações em seu navegador.

Diferente do Node.js, a plataforma Cloudflare Workers não usa o V8 por inteiro, mas se aproveita principalmente da API do Service Workers — essa é a origem de seu nome. Service Workers são um recurso implementado por navegadores modernos que permitem carregar um script que intercepta solicitações da web destinadas ao seu servidor. Isso oferece a oportunidade de reescrevê-los, redirecioná-los ou até mesmo responder diretamente às requisições. Os Service Workers foram projetados para serem executados em navegadores, mas também podem ser usados na borda.

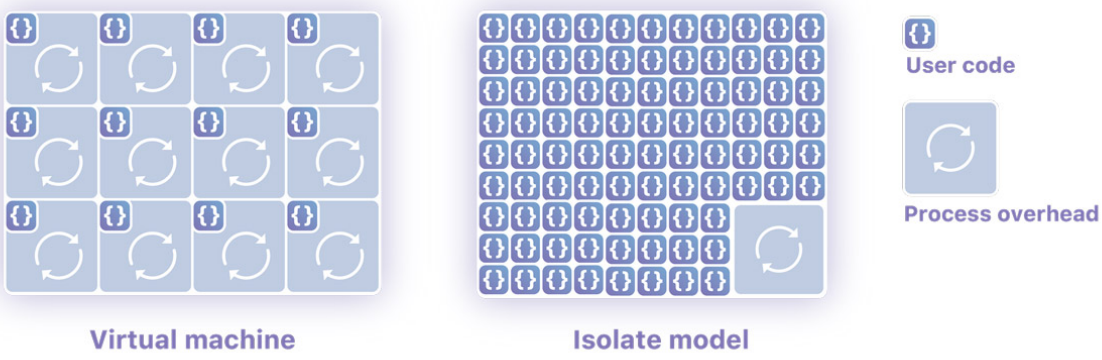
Além de tudo, o V8 ainda inclui o WebAssembly pronto para uso, isso permite ao Workers implantar código escrito em outras linguagens, assim como seus concorrentes. Entre as diversas possíveis decisões, como máquinas virtuais, contêineres, e o próprio Node.js, o uso dos Service Workers se mostrou uma opção acertada, já que atua de maneira isolada, ocupa pouca RAM e não tem Cold Start — assim afirma VARDA (2017, tradução nossa):

> “Node.js é um *runtime* JavaScript orientado a servidor que também usa V8. À primeira vista, parece fazer muito sentido reutilizar o Node.js em vez de construir diretamente no V8. No entanto, apesar de ser construído no V8, o Node não foi projetado para ser um *sandbox*. (...) Como tal, se fôssemos

> construir no Node, perderíamos os benefícios do *sandbox* do V8. Em vez disso, teríamos que fazer *sandboxing* em nível de processo (também conhecido como contêiner), que é menos seguro e menos eficiente."

Um único *runtime* pode executar centenas ou milhares de isolados, alternando perfeitamente entre eles, sem compartilhamento de memória. Dessa forma, cada parte de código executado é protegido de outro código executado em paralelo. O modelo com isolamento é projetado para iniciar rapidamente ao invés de criar uma máquina virtual para cada aplicação, assim, é criado um novo compartimento isolado dentro de um ambiente já existente. Esse modelo extingue qualquer problema de inicialização a frio do modelo tradicional de máquina virtual ou contêiner.

Figura 7 – Comparação entre máquina virtual e modelo com isolamento



Fonte: CLOUDFLARE (2022c)

Recentemente a Cloudflare anunciou que sua plataforma Workers será de código aberto (KOZLOV, 2022). O principal motivo para tal é a preocupação dos desenvolvedores em ficarem presos na tecnologia e se tornarem reféns. O segundo motivo é a dificuldade em criar um ambiente de desenvolvimento local fácil de iterar e testar rapidamente as alterações do projeto. Acertadamente, a Cloudflare tomou a posição de liberar o código-fonte e iniciou o desenvolvimento de ferramentas, bibliotecas e CLIs (Command Line Interfaces) que facilitam a experiência do desenvolvedor (DX).

Mesmo oferecendo um motor de tempo de execução inédito e original — e, até então fechado —, ainda que com algumas particularidades, utilizando-se de bons padrões de arquitetura (como Clean Architecture, proposto por Robert Martin), padrões de design (Design Patterns) e uma abordagem orientada a domínio (Domain-Driven Design), é possível desacoplar o código em camadas específicas nos aplicativos implementados na Cloudflare Workers. Consequentemente, torna-se viável criar uma aplicação que independa da infraestrutura, deixando-a flexível e removendo a sensação de estar preso ao provedor.

Assim, a Cloudflare oferece um poduto comercial que promete — e cumpre — implantar código *serverless* instantaneamente em todo o mundo (CLOUDFLARE, 2022b), oferecendo desempenho, confiabilidade e escala que são realmente excepcionais. Com essa plataforma, não é necessário configurar balanceadores de carga ou pagar por capacidade de processamento subutilizado; o tráfego é roteado e a carga é balanceada automaticamente em milhares de servidores distribuídos globalmente, sempre próximo do usuário, ou seja, todo o tráfego dos clientes é processado no *datacenter* mais próximo da origem.

Figura 8 – Distribuição dos *datacenters* da Cloudflare no mundo

Fonte: Cloudflare (2022a)

A rede conta pelo menos 270 cidades alcançadas em mais de 100 países (incluindo China continental) e promete latência máxima de 50 milissegundos para 95% da população mundial conectada à internet. Somente no Brasil são 23 cidades (CLOUDFLARE, 2022a): Americana, Belo Horizonte, Belém, Blumenau, Brasília, Campinas, Caçador, Curitiba, Florianópolis, Fortaleza, Goiânia, Itajaí, Joinville, Juazeiro do Norte, Manaus, Porto Alegre, Ribeirão Preto, Rio de Janeiro, Salvador, Sorocaba, São José do Rio Preto, São Paulo e Uberlândia.

Para iniciar, a empresa garante um custo bastante acessível, disponibilizando 100 mil solicitações diárias gratuitas, chegando a cerca de 3 milhões de requisições mensais sem necessidade de pagamento. As solicitações sem custo têm a capacidade de até 10 milissegundos de computação para completar sua carga de trabalho. O plano pago começa em um pacote de US$ 5,00 por 10 milhões de solicitações, com capacidade de computação de até 50 milissegundos. E, em um plano mais recente, é permitido uma carga de computação de até 30 segundos, ainda

assim com um custo muito atraente. Desse modo, a Cloudflare assegura que o modelo de precificação do Workers é até dez vezes mais barato que outras plataformas *serverless*, todavia com a vantagem de atuar na borda.

A maioria dos aplicativos do mundo real são *stateful*, isto é, possuem estado, salvam e carregam informações. Incontestavelmente isso se torna um grande problema quando estamos lidando com aplicações que ficam na borda, dado que toda a rede é bem distribuída e pensada para ser acessada e computada perto dos usuários finais. Um banco de dados, como visto na Seção 2.2.3, tende a ser centralizado ao menos em sua réplica de escrita. Isso torna difícil baixar a latência, já que muitos usuários podem estar distantes do servidor onde o estado é armazenado. Pensando nisso, a Cloudflare trouxe algumas soluções, dando aos desenvolvedores acesso a vários tipos de armazenamento: Workers KV, Durable Objects, R2 Storage e, mais recentemente, anunciou o D1 Database.

2.4.3.1 Workers KV

O Workers KV (Key-Value) oferece armazenamento de dados do tipo valor-chave em escala global e com baixa latência. Nele é possível armazenar e acessar dados e ativos estáticos distribuídos em todos os pontos de presença da Cloudflare, como se estivessem centralizados em um único ponto. Ele é perfeitamente escalável para suportar aplicativos que atendem a dezenas ou milhões de usuários.

Por ser eventualmente consistente — leia-se atraso nas gravações — devido a sua distribuição global, esse serviço é especialmente útil para armazenar informações que são pouco modificadas, mas acessadas frequentemente. Essas informações podem incluir autenticação de usuários, regras de controle de acesso, dados de configurações, camada de *cache* do banco de dados, arquivos pequenos, entre outras.

O KV também permite definir o TTL (Time-to-live) para que as chaves sejam expiradas em determinado momento, assim, a aplicação pode renovar as informações periodicamente sem a necessidade de fazer uma busca manual pelas chaves obsoletas (MCKEON, 2022). O limite de tamanho para cada chave-valor armazenado é de até 25MB, capacidade suficientemente grande para cobrir diversos casos de uso.

2.4.3.2 Durable Objects

Durable Objects (DO) ou Objetos Duráveis foi o primeiro passo da Cloudflare para uma abordagem *serverless* com persistência de estado consistente. Com ele é possível armazenar a instância de uma classe que gera um objeto. Essa classe pode conter métodos e estados, sejam públicos ou privados. Assim como o KV, cada objeto possui um identificador globalmente exclusivo, sendo que ele existe em apenas um local no mundo inteiro de cada vez, no entanto, após persistido, ele é distribuído migrando automaticamente para estar perto de onde é mais usado.

O objeto, quando criado, é instanciado na memória e, posteriormente, é desligado quando ocioso para ser recriado mais tarde sob demanda. Cada objeto possui armazenamento que é anexado e persistido, sendo que o armazenamento é sempre co-localizado com o objeto. Isso permite armazenamento muito rápido com consistência transacional forte, como se um grande banco de dados monolítico tradicional fosse dividido em muitas pequenas unidades lógicas. O grande ganho desta abordagem é a obtenção da enorme capacidade de dimensionamento sem esforço com zero trabalho de manutenção em relação à infraestrutura.

A maior aplicabilidade dessa tecnologia nos serviços que estão na borda é a capacidade de ter comunicação rápida com um centro de processamento (*back-end*) e, para isso, é indispensável trabalhar com dados em tempo real. Durable Objects traz a possibilidade de lidar com coordenação entre dois ou mais clientes, conectando-os em uma única instância do Workers: "solicitações relacionadas ao mesmo tópico podem ser encaminhadas para o mesmo objeto, que pode então coordenar ações entre eles" (VARDA, 2020, tradução nossa). A coordenação dos Objetos Duráveis é feita majoritariamente através de WebSockets, recurso introduzido neste serviço para funcionar em conjunto com o Workers. É através da comunicação em tempo real que a aplicação de chat se torna viável.

A diferença mais significativa entre Workers KV e DO é que o primeiro segue sendo o melhor método para fornecer conteúdo estático ou em *cache*, onde os dados são raramente alterados, mas muito consultados. Por outro lado, o segundo, é melhor para gerenciar o estado dinâmico e a coordenação entre múltiplos clientes.

2.4.3.3 R2 Storage

O Cloudflare R2 Storage é uma plataforma de armazenamento de arquivos que podem ser arbitrariamente grandes e não estruturados, distribuído e projetado para ser altamente disponível, escalável e extremamente confiável — 99,999999999% (onze "noves") de *uptime*, demonstrando a baixa probabilidade de perda de dados (MCKEON, 2021a). Ele é baseado na rede Cloudflare CDN e usa várias técnicas para fornecer esses recursos, incluindo o uso de vários nós de armazenamento em locais diferentes, a replicação de dados nesses nós e o uso de processos internos para proteger os dados contra falhas. O R2 Storage foi projetado para ser usado por aplicativos que exigem alta disponibilidade e escalabilidade, como *streaming* de vídeo, entrega de conteúdo e armazenamento em nuvem.

Esse serviço foi introduzido para competir diretamente com o Amazon S3, tendo a vantagem de não haver taxas de saída. Isso significa que a largura de banda de saída (geralmente o que incide no maior fator de cobrança) não é contabilizada nos custos do R2, mas apenas as operações e o tamanho do armazenamento. Isso decorre em razão da Cloudflare fazer parte do Bandwidth Alliance (Aliança de Largura de Banda) — um conjunto de provedores de nuvem que, em parceria, reduzem as taxas de transferência de dados a fim de diminuir custos desnecessários.

Uma plataforma sem servidor não pode armazenar arquivos, dado que é um serviço

efêmero, tradicionalmente sem estado, instanciado uma vez e logo após o uso, removido. Assim, o R2 soluciona este problema prometendo armazenamento “infinito” na borda, consumindo um grande volume de dados e mantendo-os de maneira distribuída a um baixo custo.

2.4.3.4 D1 Database

Mais recentemente a Cloudflare anunciou seu primeiro banco de dados relacional SQL, o D1. Ele é construído em cima do SQLite, um banco de dados de código aberto criado para ser pequeno, rápido, independente, de alta confiabilidade e com todos os principais recursos de banco de dados tradicionais, como Postgres e MySQL. O SQLite se auto denominava uma tecnologia sem servidor antes mesmo do termo se popularizar e foi pensado para ser executado no cliente, o que combina com a própria origem do Cloudflare Workers — oriunda do Service Workers do navegador.

Na verdade, os bancos de dados relacionais tradicionais oferecem desafios únicos que tornam as conexões *serverless* difíceis. A maioria dos bancos de dados exigem conexões TCP estáveis de longa duração entre o servidor da aplicação e o servidor do banco de dados, por outro lado, o *runtime* do Workers, devido a sua arquitetura e origem, não suporta conexões TCP e nem é estável visto que, uma vez utilizado, é finalizado. Dessa forma, para utilizar bancos tradicionais é necessário o uso de *proxy* para lidar com o *pool* de conexões através do protocolo HTTP.

Bancos de dados relacionais sem servidor não são uma tecnologia nova na nuvem, embora ainda não sejam comuns. No entanto, um banco de dados *serverless* distribuído globalmente, é. O D1 cria e distribui automaticamente clones somente de leitura dos dados gravados, perto de onde os usuários estão, e os mantém constantemente atualizados com as alterações persistidas, assim garantindo consistência forte (KOZLOV; MADDERN, 2022).

Além de ser um banco de dados distribuído de escala global, o D1 ainda salva automaticamente imagens instantâneas (*snapshots*) dos dados e armazena no R2 Storage em intervalos regulares, permitindo iniciar um processo de restauração a partir de um clique. Tudo isso ocorre com zero configuração. A infraestrutura passa a ser executada na borda da internet, a um custo baixo, sob demanda e sem gerenciamento.

2.5 Bibliotecas

É possível construir um aplicativo de chat em tempo real em qualquer linguagem de programação sem necessidade de instalação de bibliotecas ou *frameworks*. Contudo, nesse caso é necessário escrever a implementação de algum algoritmo para lidar diretamente com a interface de programação do motor de tempo de execução ou da própria linguagem utilizada. No caso do JavaScript, o Node.js fornece uma API de *stream* para criar uma conexão de rede através de *socket*, permitindo um canal de comunicação entre dois pontos, cliente e servidor. Porém,

isso torna o desenvolvimento de uma aplicação completa mais lenta e trabalhosa, exigindo um conhecimento aprofundado por parte do programador do ambiente que se está utilizando.

No desenvolvimento de *software* comercial, agilidade e facilidade de uso e manutenção são pontos-chave. Por isso, é comum que pesquisadores e desenvolvedores busquem ferramentas que tornem o processo mais acessível tanto para iniciantes quanto para profissionais experientes na área. A seguir, para demonstrar dois caminhos distintos possíveis, serão apresentadas algumas bibliotecas que podem ajudar a arquitetar e desenvolver a solução proposta para este trabalho.

2.5.1 Socket.IO

Socket.IO é uma biblioteca JavaScript que permite a comunicação em tempo real entre um servidor e um navegador na web, concentrando-se igualmente na confiabilidade e velocidade (SOCKET.IO, 2017). O Socket.IO permite a comunicação bidirecional entre o servidor e o cliente, o que significa que as mensagens podem ser enviadas e recebidas em tempo real. Isso o torna ideal para aplicativos em que o tempo de resposta é importante, como em uma sala de bate-papo onde as mensagens precisam ser entregues imediatamente, aplicativos colaborativos e jogos.

Embora seja uma biblioteca voltada para Node.js, e não para as plataformas da Cloudflare e Fastly que têm seus próprios *runtimes*, ela é utilizada por muitos projetos de chat em tempo real que são implementados na nuvem tradicional e até mesmo em infraestruturas como a Lambda@Edge, que é *serverless* e atua na borda, mas mantém compatibilidade com Node. A popularidade ocorre devido a sua confiabilidade, permitindo várias novas tentativas de conexão caso perdida, performance através da utilização de WebSockets e escalabilidade de código devido sua simplicidade em abstrair a complexidade da programação.

Algumas implementações *server-side* da biblioteca também estão disponíveis em outras linguagens:

- Java;
- C++;
- Swift;
- Dart;
- Python;
- .NET.

Com ela, cria-se um canal de transmissão onde eventos são emitidos entre servidor e clientes a partir de um tópico que agrega e relaciona usuários para um determinado fim. Implementar um chat em tempo real em memória é relativamente simples, mas pode ser desafiador

escalar um serviço deste tipo ao nível de infraestrutura; principalmente se envolver recursos como persistência de estado, permitir adicionar apelido customizado aos usuários, mostrar quando alguém está digitando, exibir clientes *online* ou adicionar mensagens privadas entre eles.

2.5.2 Wrangler

Wrangler é uma bilbioteca JavaScript no modelo de interface de linha de comando (CLI) para gerenciar o Cloudflare Workers. Com o Wrangler, é possível construir, publicar e atualizar os Workers sem sair do terminal, proporcionando rápido e fácil acesso através de seus comandos utilitários.

O Wrangler também permite criar *pipelines* de integração e entrega contínua (CI/CD) definindo o *token* de segurança da API da Cloudflare e o número de identificação da conta do usuário. Numa perspectiva geral, ele fornece vários recursos para tornar o gerenciamento dos Workers fácil e conveniente, melhorando a experiência do desenvolvedor. Com ele é possível:

- Construir Workers com TypeScript ou JavaScript;
- Visualizar os Workers ativos antes de publicar;
- Minimizar o código automaticamente antes de publicar;
- Atualizar facilmente qualquer Worker com apenas um comando;
- Gerencie vários Workers com facilidade.

2.5.3 Miniflare

O Miniflare é uma biblioteca que simula o ambiente de desenvolvimento e os principais produtos do ecossistema do Cloudflare Workers. Com ele é possível executar um aplicativo voltado ao Workers (sem depender de seu *runtime*) em ambiente local, apresentando uma enorme facilidade para desenvolver e construir testes automatizados, além de ajudar a obter *feedback* instantâneo quando mudanças no código são realizadas.

Essa biblioteca é desenvolvida com Node.js — *runtime* JavaScript construído no mecanismo JavaScript V8 do Chromium como mencionado na Seção 2.2.2. “O V8 é o mesmo mecanismo que alimenta o tempo de execução do Cloudflare Workers, mas o Node e os Workers implementam diferentes APIs de tempo de execução sobre si. Para garantir que as APIs do Node não sejam visíveis para o código do Workers dos usuários e para injetar as APIs dos Workers, o Miniflare usa o módulo Node.js VM. Isso permite que seja executado código arbitrário em um contexto V8 personalizado” (COLL, 2022, tradução nossa).

2.6 Trabalhos Relacionados

No contexto de chat em tempo real, há trabalhos acadêmicos descrevendo diferentes abordagens e com diversos propósitos, tais como sistemas que buscam implementar um design de uma aplicação de chat com suporte multilinguístico (THARANIDHARAN et al., 2022) e (CHAUDHARI; SHINKAR; PAGARE, 2018), ou um sistema de grupos de chat para compartilhamento de arquivos (KUMAR; SINGH, 2019), e outros uma sala de bate-bapo pública (THAKUR; DHIMAN, 2021). Sucede que trabalhos nesta esfera recorrem a arquiteturas tradicionais como a MEAN (POULTER; JOHNSTON; COX, 2015) ou MERN, que envolve MongoDB, Express.js, Angular.js, React e Node.js, ou outros que têm como foco PHP, AJAX ou JAVA com *multi-threading*, *stacks* que, sem severas adaptações, não são adequadas para a computação de borda sem servidor.

Em uma breve análise, a maioria das pilhas de desenvolvimento mencionadas não operam no *back-end* dos principais fornecedores comercias de infraestrutura de borda sem servidor. Node.js, como analisado, não é um *runtime* compatível com todos os fornecedores; Express.js é uma biblioteca de roteamento para Node e tem a mesma limitação. React e Angular.js são, respectivamente, uma biblioteca e um framework voltados ao *front-end* e podem ser renderizados em qualquer uma das soluções, mas não é deles que parte os casos de uso e as regras de negócio da aplicação, portanto ajudam a solucionar problemas apenas no escopo do cliente. MongoDB é um banco de dados não relacional que anunciou compatibilidade com o Cloudflare Workers no final do ano de 2021 (MCKEON, 2021b) e é compatível com a AWS@Lambda, no entanto, utilizando um banco de dados que não foi construído para a borda reintroduzimos o problema de latência, logo, não é recomendado para aplicações que exijam baixo tempo de resposta.

Outras linguagens de programação como PHP e JAVA não estão preparadas para a borda sem servidor comercial — apesar da compatibilidade através do WebAssembly, ainda é provável que a implementação sofra com problemas devido à imaturidade da tecnologia. Por fim, AJAX é uma tecnologia manual e rudimentar para troca de dados em tempo real: funciona, mas gera latência e sobrecarga desnecessária no navegador.

Neste trabalho, em vez de recorrer à nuvem convencional, as tecnologias utilizadas envolvidas serão direcionadas à computação sem servidor na borda. Isso garante que, mesmo que duas salas de conversação sejam criadas, cada uma delas esteja sempre próxima de seus usuários, assegurando desempenho e escalabilidade.

2.7 Considerações do Capítulo

Nesse capítulo foram apresentados áreas de conhecimento, tecnologias e técnicas que possuem relação com o tema abordado neste trabalho. No próximo capítulo serão apresentados os materiais e métodos a serem utilizados para sua realização.

3 METODOLOGIA

Neste capítulo serão delimitados detalhes relativos à metodologia empregada para o desenvolvimento do trabalho, bem como especificidades do ambiente e ferramentas de *software* a serem utilizadas.

3.1 Metodologia Utilizada

Quanto a abordagem teórica da pesquisa, a metodologia definida baseia-se no método dedutivo, utilizando o raciocínio lógico para chegar a conclusões mais particulares, a partir de princípios e preposições gerais.

Adotando os princípios do Manifesto para o Desenvolvimento Ágil de Software (BECK, 2001), a metodologia definida para o desenvolvimento da aplicação foi o Extreme Programming (XP). Existe um conjunto de princípios que deve ser seguido por equipes que forem usar XP em projetos, como o *feedback* rápido, presumir simplicidade, abraçar mudanças, trabalho de alta qualidade, pequenos passos, melhoria, diversidade e reflexão.

O XP é dinâmico e flexível (VALERIANO, 1998), ajudando a ter rapidez, qualidade e flexibilidade no desenvolvimento de um *software*. Portanto, na possibilidade de reajustes necessitarem ser exercidos (conforme as revisões semanais), o cronograma poderá sofrer alterações pontuais.

O desenvolvimento será separado em módulos, os quais foram definidos no escopo deste trabalho, e foram apresentadas as necessidades e requisitos do projeto. Posteriormente, será traçado estratégias de como a construção deve ocorrer na prática, acordando com a metodologia empregada.

Todo o projeto terá controle de versionamento através do GIT e armazenado remoto no GitHub. *Wireframes* serão criados para guiar a elaboração da aplicação. Ainda, alguns componentes construídos com tecnologia web serão utilizados para a estruturação do *layout*, mas com flexibilidade de resolução e tamanho, por se tratar de uma interface amigável e que se adapte bem a uma variedade de telas de dispositivos diferentes.

Desse ponto em diante, o projeto será desenvolvido usando o Visual Studio Code. Diferente de outros ambientes de desenvolvimento, o Visual Studio Code é gratuito e funciona de forma mais parecida com um IDE convencional, mesmo se considerando apenas um editor de texto. Esse ambiente permite que o código seja organizado de forma mais concisa e clara, o que se torna importante à medida que o projeto avança e o número de linhas cresce.

3.2 Técnicas de Pesquisa

Visto que um dos objetivos do estudo é entender como a computação de borda sem servidor se compara à computação em nuvem tradicional em termos de desempenho, custo e escalabilidade, esse tema se enquadra na classe de estudo comparativo, ou seja, o método comparativo se consiste em investigar coisas ou fatos e explicá-los segundo suas semelhanças e suas diferenças. O método permite a análise de dados concretos e a dedução de semelhanças e divergências entre elementos constantes, abstratos e gerais, propiciando investigação de caráter indireto.

Contudo, o cerne do trabalho é a verificação da viabilidade do uso de *serverless edge computing* para a construção de uma aplicação de bate-papo, portanto, para o êxito do projeto, aplica-se o conhecimento e os métodos de abordagem a fim de construir um projeto prático que traga uma solução que ajude empresas e contribua para a melhora de vida de pessoas. Para o trabalho prático, a técnica utilizada para colete de dados e análise é a revisão bibliográfica e de documentação das tecnologias especificadas neste trabalho.

3.3 Ambiente de Teste

Para desenvolvimento e prototipação do projeto foi utilizado um computador *desktop* de propriedade do autor. Para criar um ambiente de testes que simula a infraestrutura do provedor, foi utilizado Node.js e a biblioteca Miniflare em conjunto com a CLI Wrangler. A execução dos métodos de avaliação e posterior implementação foram realizadas na infraestrutura da Cloudflare. As especificações de *hardware* e *software* desses ambientes estão nas Tabelas 1 e 2.

Tabela 1 – Especificação do ambiente de prototipação

Recurso	**Especificação**
CPU	Intel Core i5-7600K @ 3,80 GHz, 4 núcleos
GPU	NVIDIA GeForce GTX 1060 @ 1708 MHz, 1152 Cuda® Cores
RAM	32 GB
Armazenamento	Samsung M.2 NVMe 250 GB, Seagate 2.5" 7200RPM 1 TB
Sistema Operacional	Windows 11 Pro, WSL 2.0 Ubuntu 20.04.4 LTS
Versão do Node.js	16.7.0
Versão do JavaScript	ES2020

Fonte: Autor

3.4 Considerações do Capítulo

Neste capítulo foram apresentadas as metodologias e as especificidades do ambiente de desenvolvimento e das ferramentas utilizadas no trabalho, tanto no que diz respeito ao *software*

Tabela 2 – Especificação do ambiente de produção por instância do Worker

Recurso	**Especificação**
CPU	50 Milissegundos por Requisição HTTP
GPU	-
RAM	128 MB
Armazenamento	Ilimitado
Sistema Operacional	Linux (Não Especificado)
Versão do Node.js	-
Versão do JavaScript	ES2020

Fonte: Cloudflare (2022e)

quanto ao *hardware*. Também foram descritas as técnicas de pesquisas para a parte teórica e, ainda, como elas foram realizadas.

4 DESENVOLVIMENTO

Nesta Seção serão descritas, de forma mais detalhada, as etapas de análise dos requisitos e os processos realizados em cada etapa, visando elucidar como a construção foi realizada através do uso das ferramentas e tecnologias relatadas nas Seções anteriores.

4.1 Requisitos da Aplicação

Os requisitos de um aplicativo são o conjunto de tarefas que o aplicativo deve ser capaz de executar. Esses requisitos devem ser claramente definidos antes do início do processo de construção do software. Uma vez que os requisitos são estabelecidos, o desenvolvedor pode começar a trabalhar no projeto de maneira que fique consistente com o design e escopo de implementação, construindo e testando o aplicativo até sua entrega.

4.1.1 Requisitos Funcionais

Requisitos funcionais são aqueles que definem características diretas do sistema, como funções a cumprir ou detalhamento técnico que o desenvolvedor deve estar atento no momento da construção do *software*. Abaixo os itens e subitens definidos.

- Definir apelido:
  - Deve ser obrigatório a definição de um apelido aos usuários;
  - A definição do apelido deve ser feita antes de entrar na sala;
- Criar sala:
  - Deve ser possível criar uma sala de bate-papo;
  - O usuário poderá definir o nome da sala somente após informar seu apelido;
  - Não deve ser possível criar uma sala com nome repetido, nesse caso o usuário deve ingressar na sala já existente;
  - Após criada, deve ser gerado um endereço compartilhável para que outros usuários possam ingressar diretamente na sala;
- Trocar mensagens:
  - Deve ser possível enviar mensagens a outros usuários;
  - Deve ser possível receber mensagens de outros usuários;
  - Todas as mensagens devem ser públicas e compartilhadas com todos os membros da sala;

  - As mensagens devem ser armazenadas para gerar um histórico de conversas;
  - O histórico das últimas 100 mensagens deve ser enviado aos usuários quando ingressarem em uma sala;

- Visualizar usuários:
  - Deve ser possível ver quais outros usuários estão atualmente na mesma sala através de seus apelidos;
  - Não deve ser exibindo o apelido de um usuário caso ele saia da sala.

4.1.2 Requisitos Não Funcionais

Requisitos não funcionais são conhecidos como os que controlam a qualidade do produto; normalmente são eles que impõem restrições aos requisitos funcionais, seja no projeto ou em sua execução. Alguns pontos para essa categoria foram levantados:

- Tecnologia: o aplicativo deve ser desenvolvido na linguagem JavaScript devido sua versatilidade na web, como visto na Seção 2.2.2.
- Comunicação em tempo real: o programa deve conseguir lidar com comunicação em tempo real entre dois ou mais usuários;
- Latência: o sistema deve ter baixa latência para proporcionar uma experiência de usuário tranquila;
- Escalabilidade: a aplicação deve ser capaz de aumentar ou diminuir conforme necessário para acomodar um número maior ou menor de usuários;
- Disponibilidade: o *software* deverá ter alta disponibilidade, proporcionando segurança e confiança a seus usuários — 99,9% de *uptime*.
- Custo: o sistema deve ser econômico para que não aumente desnecessariamente o custo total durante sua operação;
- Compatibilidade: a aplicação deve ser compatível com as versões modernas dos navegadores mais utilizados;
- Acessibilidade: o sistema precisa ser acessível por múltiplos dispositivos de diferentes tamanhos de tela.

4.2 Wireframes

Para cumprir com os requisitos definidos, os *wireframes* do aplicativo de chat devem conter três páginas centrais: a tela de apelido, a de entrada e a de sala. A tela de apelido é a

primeira página que o usuário verá e será usada para autenticar e definir seu nome. A segunda, de entrada, será usada para permitir que ele crie ou entre na sala de bate-papo desejada. Por fim, a última, será a tela principal onde o usuário poderá enviar e receber mensagens em tempo real.

4.2.1 Tela de Apelido

Esse componente tem como objetivo permitir que o usuário defina um apelido para sua identificação na sala de bate-papo. A tela possui um campo de texto para inserção do apelido e um botão para confirmar a ação.

Figura 9 – Wireframe da tela inicial da aplicação

Fonte: Autor

4.2.2 Tela de Entrada

Esse componente autoriza o usuário a selecionar uma sala de bate-papo para participar. A tela não possui uma lista de salas de bate-papo disponíveis, mas permite a inserção do nome da sala que se deseja entrar ou criar.

4.2.3 Tela de Sala de Bate-papo

Essa página permite que o usuário participe do bate-papo em tempo real, trazendo elementos de maneira clara e intuitiva. Nela está incluído a lista de usuários disponíveis, um campo de texto para inserção de mensagens, uma lista de mensagens enviadas e um botão para enviar a mensagem.

Figura 10 – Wireframe da segunda tela da aplicação

Fonte: Autor

Figura 11 – Wireframe da terceira e principal tela da aplicação

Fonte: Autor

4.3 Escolha do Provedor

Como foi demonstrado na Seção 2.4, os principais provedores de infraestrutura são a AWS Lambda@Edge, a Fastly Compute@Edge e a Cloudflare Workers. Embora as 3 companhias possibilitem atuar na borda, a Cloudflare se mostrou a solução mais madura para os casos de uso e escopo deste trabalho.

A empresa criou um rico ecossistema de produtos que se complementam e permitem atuar na borda sem exigir conhecimento aprofundado da complexa tarefa de gerenciar a infraestrutura. Nela não é necessário configurar balanceadores de carga para escalar qualquer aplicação implementada em sua rede de pontos de presença, o que é importante para garantir que o aplicativo de bate-papo seja capaz de lidar com grandes volumes de tráfego sem problemas. Ainda, ao implantar um serviço, ele é automaticamente distribuído para todo o planeta, próximo de cada usuário, em poucos segundos.

O Cloudflare Workers fornece um grau mais alto de controle sobre como as solicitações são processadas ou roteadas e como os próprios Workers são configurados. Ela permite criar uma camada de *cache* estático gratuitamente apenas informando o *endpoint*, o tempo de expiração e onde deve buscar a resposta caso expire. A política de preços da Cloudflare também é bastante generosa, oferecendo um nível gratuito para aplicativos que estão começando e não têm financiamento; seu modelo de precificação de pagamento é conforme o uso e nenhum recurso computacional é desperdiçado.

Além disso, o Workers oferece melhor suporte do que seus concorrentes para WebSockets através da API implementada por seu *runtime* (KUPERMAN, 2021). Sua documentação é clara e detalhada a ponto de que desenvolvedores iniciantes também consigam construir aplicativos nela sem experiência aprofundada com a plataforma.

Por fim, o Workers é um serviço completo que abrange diversos casos de uso como, por exemplo: retornar páginas HTML, atuar como um *middleware*, realizar *proxy* com o servidor de origem, redirecionar solicitações de um URL a outro, efetuar testes A/B controlando qual versão de recurso o cliente verá, checar e autenticar usuários, servir como um serviço de geolocalização, atuar como uma camada de *cache*, entre inúmeros outros. Todos os motivos citados levaram a optar pela Cloudflare como provedor final de infraestrutura.

4.4 Implementação

A principal parte da implementação deste sistema se concentrou em uma classe que representa uma sala de troca de mensagens através de um Durable Object — a chamamos de ChatRoom. Essa classe é responsável por criar e controlar a sala de bate-papo e é instanciada por uma função principal que é análoga ao método *main* de outras linguagens de programação; é essa função que cria ou recupera salas existentes — ela é o próprio Worker em seu funcionamento.

Tabela 3 – Comparação de recursos entre os 3 provedores

	Lambda@Edge	**Compute@Edge**	**Cloudflare Workers**
Localizações	13 cidades, 10 países	81 cidades	270 cidades, 100+ países
Linguagens	JavaScript (Node.js), Python	JavaScript, WebAssembly	JavaScript, WebAssembly
Runtime	Virtual Machine (VM)	Lucet	V8 Process Isolates
Acesso ao Sistema de Arquivos	Sim	Não	Não
Variáveis de Ambiente	Não	Sim	Sim
Tempo de CPU	5 seg	50 ms	10 ms (grátis) 50 ms (*bundled*) 30 seg (*unbundled*)
Memória RAM	128 MB	128 MB	128 MB
Tamanho Máximo de Código	1 MB	50 MB	1 MB
Quantidade de Funções	100	-	30 (grátis) 100 (pago)
Requisições	10.000/seg/região	-	1.000/min (grátis) Ilimitado (pago)
Requisições Concorrentes	1.000/região	-	Ilimitado
Nível Gratuito	Nenhum	US$ 50,00	100.000 requisições/dia (∼3.000.000/mês)
Precificação	US$ 0,60/mi de invocações + US$ 0,00005001 por GB/seg	US$ 0,50/mi de requisições + US$ 0,000035 por GB/seg	*Bundled*: US$ 5,00/10 mi de requisições por mês + US$ 0,50/milhão de requisições adicionais

Fonte: AWS (2022b), Fastly (2022a) e Cloudflare (2022e)

Neste trabalho, uma sala de bate-papo é uma implementação direta de um Durable Object, ou seja, é um objeto cuja estrutura e comportamento são preservados e mantidos pelo sistema entre execuções. Isso é feito através do armazenamento do objeto na infraestrutura da Cloudflare, o objeto é carregado em memória quando uma nova execução é iniciada, as alterações que ocorreram durante a execução anterior são refletidas na nova execução e novas alterações são salvas nesse objeto quando a execução é encerrada.

Entretanto, apenas implementar um Durable Object não é suficiente. Para isso, é necessário um Worker: somente através dele que é possível acessar o DO.

4.4.1 Construindo a Sala de Bate-papo

No construtor da classe ChatRoom temos parâmetros para lidar com as variáveis de ambiente e o estado da aplicação, onde todas as mensagens são armazenadas conjuntamente com o nome do remetente e a hora enviada (Figura 12). Nela também criamos um vetor para armazenar todas as sessões WebSocket criadas internamente, assim é possível saber quais os clientes conectados e redirecionar novas mensagens a eles posteriormente.

Figura 12 – Classe do Durable Object ClassRoom

```
export class ChatRoom {
  constructor(state, env) {
    this.storage = state.storage;
    this.env = env;
    this.sessions = [];
  }

  // more code...
}
```

Fonte: Autor

4.4.2 Comunicação Entre Múltiplos Clientes

Todo Objeto Durável precisa ter, além do construtor, o método *fetch*. Ele é o responsável pela execução e orquestração das regras de negócio da classe. É no *fetch* (Figura 13) que os pares da conexão WebSocket são criados e repassados, então, um para o método *handleSession*, responsável por lidar no *back-end* com os eventos gerados, e outro para o navegador do usuário a fim de receber e enviar eventos ao servidor.

É o método *handleSession* (Figura 14) que efetivamente faz o gerenciamento da sala de bate-papo. Ele salva as sessões de comunicação, informa os demais usuários conectados, recupera as mensagens armazenadas para exibir um histórico recente e define dois eventos de escuta: *message* e *close*.

O evento de escuta *message*, definido na função *handleMessage*, é executado sempre que uma mensagem é enviada por um dos usuários. O *handleMessage* agrupa os agregados (remetente, mensagem e data/hora), dispara isso aos outros clientes através do método *boardcast* e salva a mensagem recebida pelo evento no *storage* instanciado no construtor.

Figura 13 – Método assíncrono *fetch*, onde o par de sessões WebSocket são transformados em vetor e desestruturados nas variáveis *client* e *server*

```
export class ChatRoom {
  // some code...

  async fetch(request) {
    const url = new URL(request.url);

    if (url.pathname === "/websocket") {
      const [client, server] = Object.values(new WebSocketPair());
      await handleSession(server);
      return new Response(null, { status: 101, webSocket: client });
    }

    return new Response("Not Found", { status: 404 });
  }

  // more code...
}
```

Fonte: Autor

Por fim, temos o método *broadcast*, que busca no vetor de sessões todas as conexões com clientes criadas e retransmite qualquer mensagem recebida em seu parâmetro. As mensagens transmitidas por ele seguem um protocolo interno para que o *front-end* saiba lidar com cada tipo de mensagem, são eles: *joined*, *message*, *quit* e *error*. Com esse conjunto de protocolos conseguimos dizer para o navegador quando um usuário entrou, quando uma mensagem foi recebida, em que momento um usuário saiu e se ocorreu algum erro.

4.4.3 Concepção do Cliente

Diferente do *back-end*, e apesar do *front-end* não conter nada de único em relação aos trabalhos aqui citados, ainda vale a sua menção. A sua construção foi feita em uma única página e foi elaborada com JavaScript através do uso de HTML, CSS e auxílio da biblioteca de UI TailwindCSS.

Ele é composto por 3 formulários manipulados por JavaScript usando a DOM (Document Object Model). O primeiro serve para informar um apelido antes de escolher a sala de bate-papo. No segundo deve-se informar o nome da sala que se deseja entrar. Finalmente, o terceiro, é a própria sala de conversação.

Figura 14 – Método assíncrono *handleSession*, responsáveis por definir o evento de escuta para a troca de mensagens

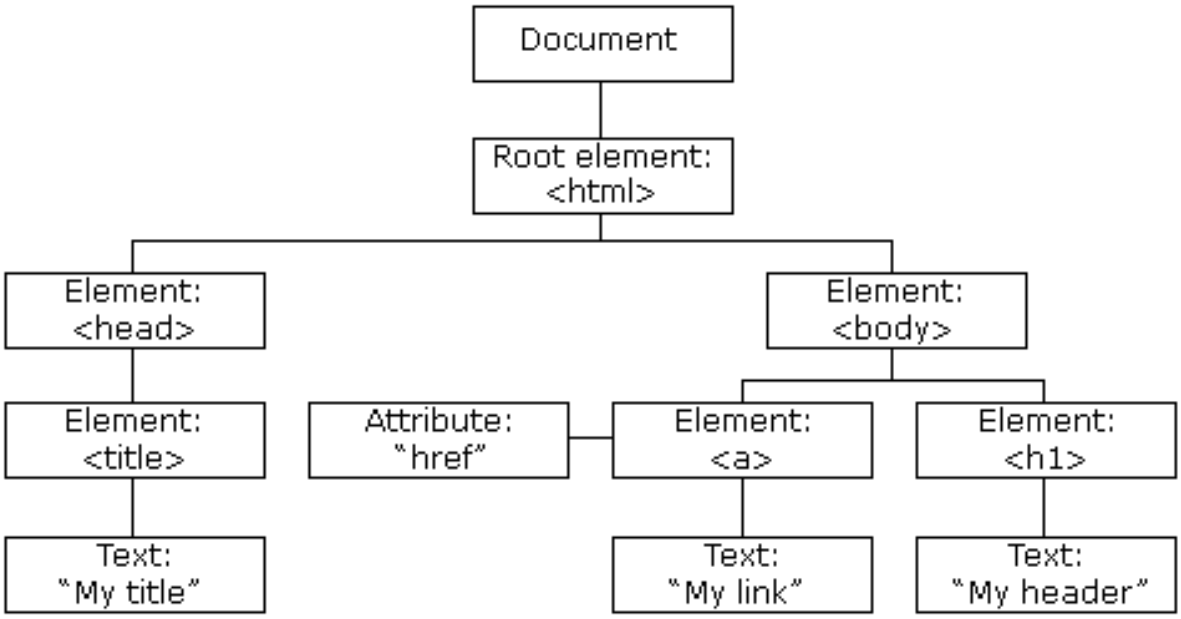
```
export class ChatRoom {
  // some code...

  async handleSession(webSocket) {
    webSocket.accept();

    const session = { webSocket, recoverMessages: [] };
    this.sessions.push(session);

    this.sessions.forEach((otherSession) => {
      if (otherSession.name) {
        session.recoverMessages.push(JSON.stringify({ joined: otherSession.name }));
      }
    });

    const storage = await this.storage.list({ reverse: true, limit: 100 });
    const recoveredMessages = [...storage.values()].reverse();
    recoveredMessages.forEach((value) => {
      session.recoverMessages.push(value);
    });

    let messagesAlreadyRecovered = false; // only allow one time to recover messages

    webSocket.addEventListener("message", messageHandler);
    webSocket.addEventListener("close", closeHandler);

    // messageHandler and closeHandler code here...
  }

  // more code...
}
```

Fonte: Autor

4.5 Implantação na Borda

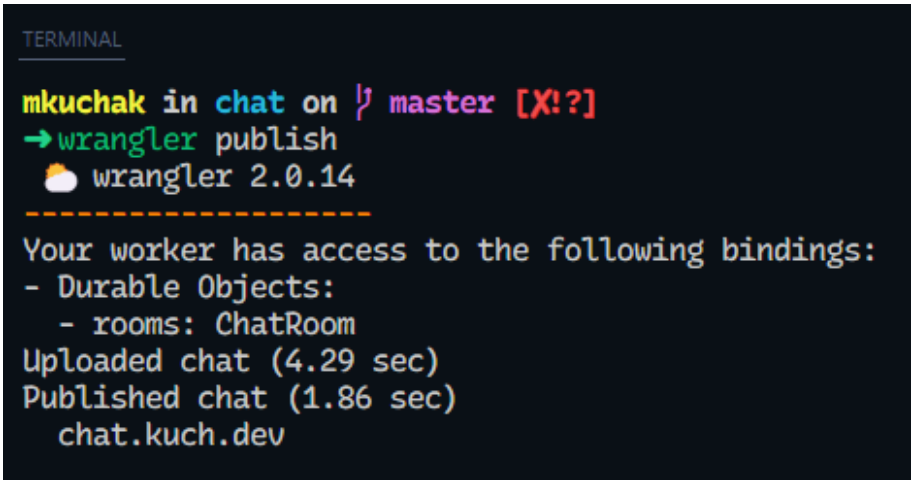
Por último, a implantação da aplicação é tão simples quanto a codificação. Para esta finalidade foi utilizado a biblioteca CLI Wrangler. Nela é possível configurar as credenciais para a Cloudflare a partir do comando *wrangler login* e definir as configurações do Workers para a aplicação através do arquivo *wrangler.toml*. Com tudo configurado, basta executar o comando *wrangler publish* e em alguns segundos o código é implantado na infraestrutura *serverless* em todo o mundo.

Figura 15 – Funcionamento da DOM, que representa os objetos de uma página HTML em forma de uma árvore



Fonte: Okere (2021)

Figura 16 – Com o Wrangler, apenas um comando é executado para publicar o aplicativo na borda



Fonte: Autor

4.6 Considerações do Capítulo

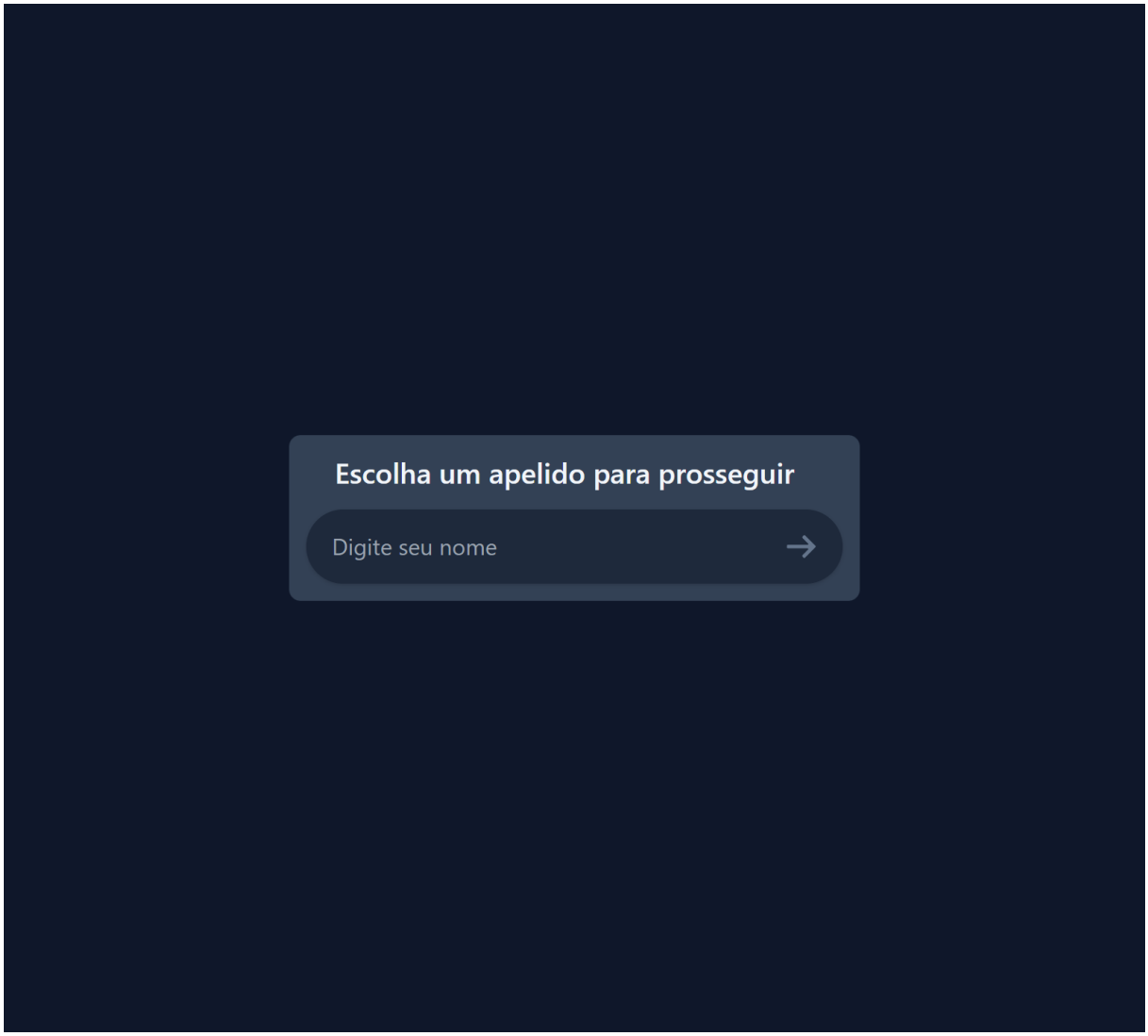
Neste capítulo foi descrito as circunstâncias que levaram à escolha do provedor e a construção do *back-end* e *front-end*, detalhando os requisitos da aplicação e as decisões tomadas durante seu desenvolvimento. O capítulo discorreu também sobre os motivos pelos quais as metodologias foram escolhidas, e como elas se aplicam no contexto da implementação da aplicação de chat.

Ainda, é importante destacar que o *front-end* da aplicação levou mais tempo para ser construído do que o *back-end*, devido a quantidade de detalhes envolvidos. A facilidade de uso do Cloudflare Workers e seu ecossistema simplificam a codificação e o gerenciamento de infraestrutura, o que também explica o tempo relativamente mais curto para sua implementação.

5 RESULTADOS

Neste capítulo serão apresentadas análises dos resultados da aplicação em relação ao uso da infraestrutura na computação de borda sem servidor. Essa análise é baseada na pesquisa e experiência adquirida na construção deste trabalho.

Figura 17 – Componente de definição de apelido

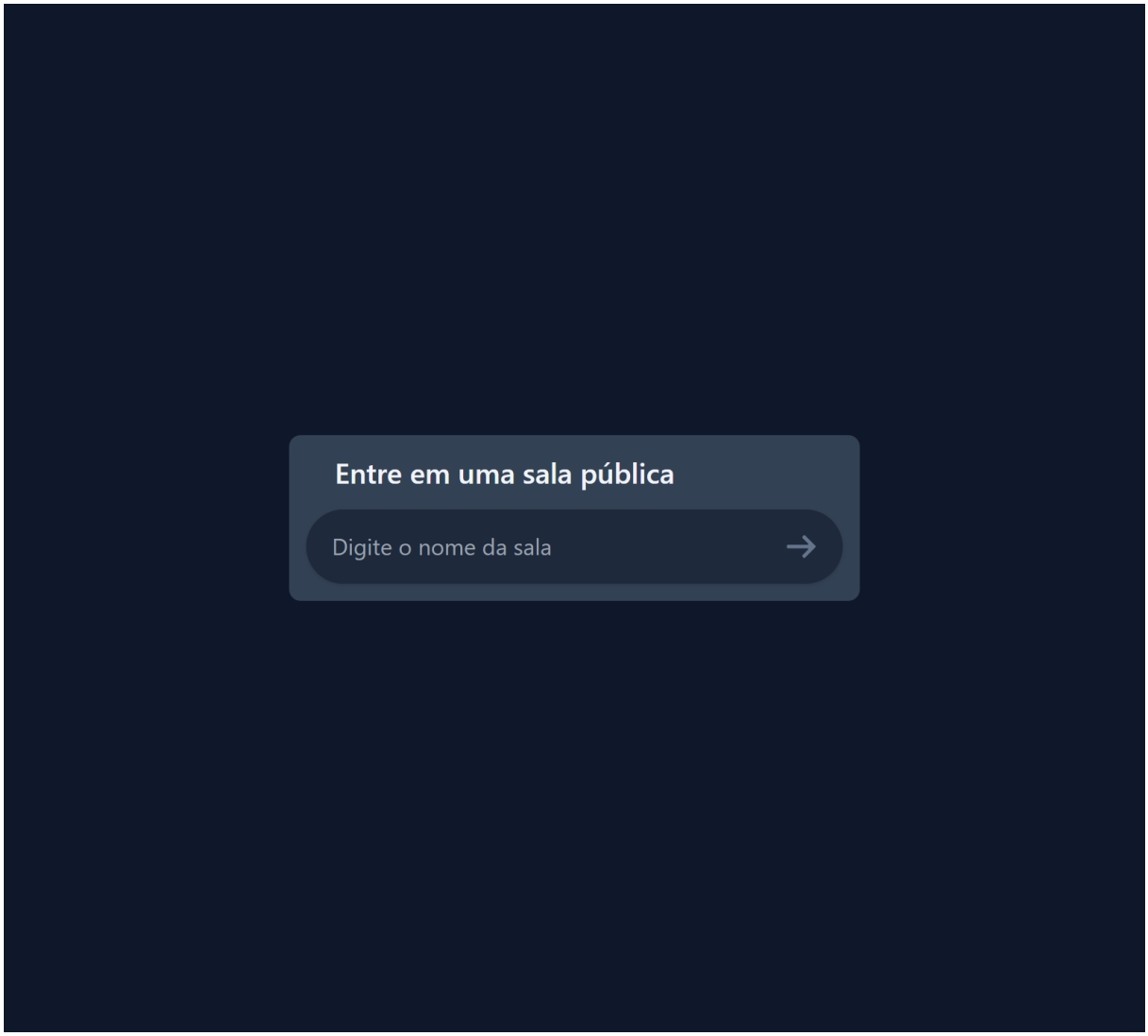


Fonte: Autor

5.1 Análise de Viabilidade

Em comparação com os trabalhos dos autores Tharanidharan et al. (2022), Chaudhari, Shinkar e Pagare (2018) e Kumar e Singh (2019), que usam arquiteturas tradicionais em infraestrutura na nuvem, ficou claro que é perfeitamente possível criar uma aplicação de chat em tempo real na *serverless edge computing*.

Figura 18 – Componente de ingresso à sala de bate-papo



Fonte: Autor

Os 3 principais serviços de infraestruturas analisadas neste trabalho são capazes de operar uma aplicação deste tipo, já que todas elas tem suporte a WebSocket. As principais diferenças entre elas é a maneira de criar e configurar os serviços, a dificuldade no manuseio das interfaces de gerenciamento, a complexidade de suas APIs de programação de seus *runtimes* e a oferta de serviços (qualidade e quantidade). Entretanto, o provedor que mais se destaca, é o que tem maior preparo na área de foco deste trabalho: a Cloudflare, uma empresa que nasceu na borda.

5.2 Análise de Performance

Um dos elementos mais importantes para que haja desempenho na borda é relativo à fonte de dados da aplicação. Embora seja possível executar uma aplicação como esta no AWS Lambda@Edge e Fastly Compute@Edge, ainda enfrentaríamos problemas de performance na persistência de estado. Como possível solução, a AWS oferta o DynamoDB, um serviço de banco

Figura 19 – Página principal da sala de bate-papo

Fonte: Autor

de dados não relacional com distribuição em múltiplas regiões. Também temos o MongoDB Atlas, com sua oferta multirregional, e o FaunaDB, que congrega *serverless* multi-região ao seu banco de dados. Ainda assim, existe um abismo entre ser multi-região e atuar na borda.

Nenhuma dessas ferramentas realmente chegou à borda. E a primeira razão é bastante simples: distribuir um banco de dados não é barato. O armazenamento, sim, é barato, mas são centenas de cópias distribuídas de todo o banco de dados. A outra restrição é o consenso. É difícil criar consenso sobre o estado de um banco de dados inteiro, com latência de mais de 100 milissegundos e centenas de réplicas distribuídas em múltiplos pontos de presença.

A Cloudflare cumpre esse objetivo com o Durable Objects através da fragmentação. Assim, a fonte da verdade é unitária, fica distribuída e se move entre diversos pontos de presença, conforme a necessidade. O fato mais importante é que ele cria o objeto o mais próximo possível do seu usuário. Esse é o principal motivo de sua eficiência e rapidez. O poder de escalar

exageradamente sem preocupação com a infraestrutura é fascinante.

Toda essa velocidade importa quando estamos falando de aplicações que precisam ser entregues rápido, de qualquer lugar do mundo. Por exemplo, para aplicações de jogos, aplicativos de mensagens e aplicativos de *streaming*. A Internet das Coisas é outro exemplo de onde essa tecnologia pode ser aplicada sem a necessidade de gerenciar servidores locais para operar na borda. Essa proximidade com os dados em sua fonte pode gerar grandes benefícios de negócios: tempos de resposta melhores, disponibilidade de largura de banda aprimorada e *insights* mais rápidos.

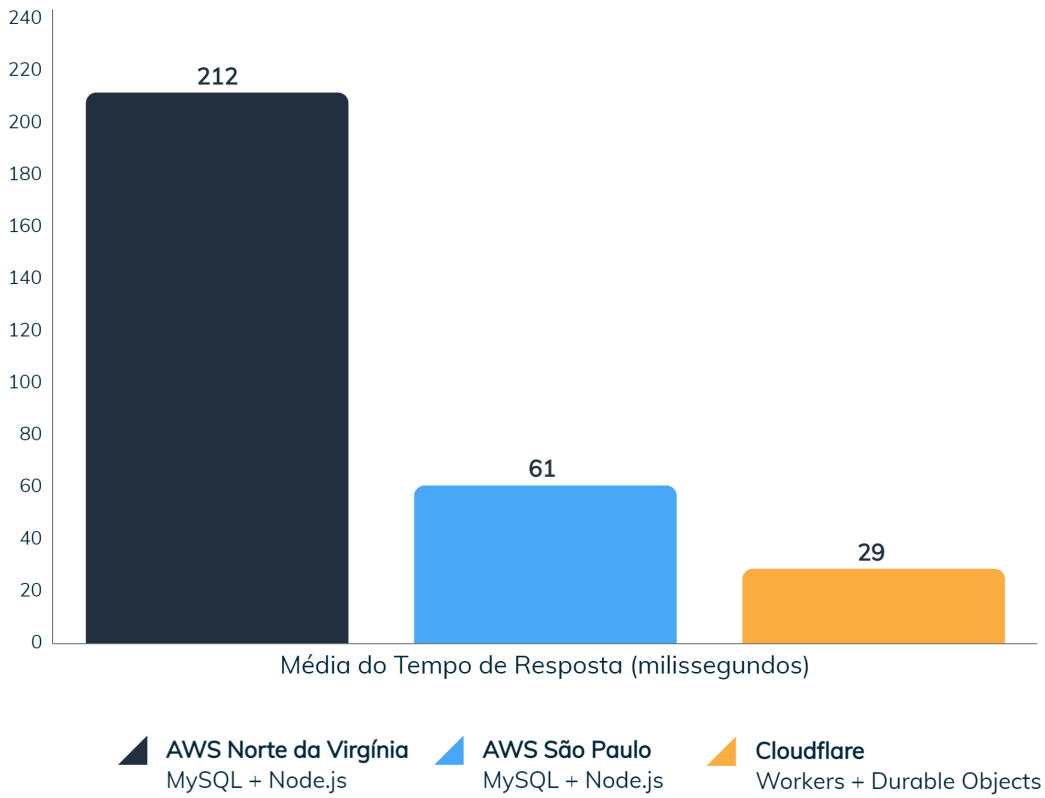
Um aplicativo de bate-papo construído em uma infraestrutura tradicional, com um banco de dados tradicional, terá, na melhor das hipóteses, segundos de latência, pois as mensagens devem ser armazenadas e depois pesquisadas por outros usuários. A proposta deste trabalho, que visa uma abordagem na computação em borda, pode ajudar a mitigar essas adversidades fornecendo um ponto de coordenação em tempo real no qual as mensagens podem ser retransmitidas sem passar por uma camada dedicada unicamente ao armazenamento e, mesmo assim, tendo seu estado persistido com integridade consistente. Isso resulta em latência muito menor e uma experiência de usuário muito mais suave.

Aplicações triviais também se beneficiam em estar na borda quando se trata de diminuir o tempo de resposta: um carrinho de compras de um *e-commerce*, por exemplo, ao responder 50 ms mais rápido quando alguém clicar em "adicionar ao carrinho", certamente terá um benefício mensurável na receita. Uma experiência muito melhor ocorre quando um botão é clicado e a ação ocorre instantaneamente, independente da qualidade da conexão de rede do usuário. Esperar entre meio segundo a vários segundos para cada clique é uma UX ruim, mesmo que os utilizadores da aplicação venham a se acostumar com isso.

Na Figura 20 pode-se observar o resultado de um experimento realizado entre dois diferentes serviços de infraestrutura: AWS e Cloudflare. Para isto foi desenvolvido uma aplicação simplificada, que faz um pequeno processamento, gera um dado aleatório, o persiste em algum serviço compatível e o recupera, retornando-o como resposta. No experimento na AWS foi utilizado uma instância EC2 modelo *t2.nano* e o Amazon RDS com banco de dados MySQL. O teste foi executado em duas regiões distintas, uma no Norte da Virgínia e outra em São Paulo. No teste na Cloudflare uma aplicação semelhante, mas adaptada, foi utilizada na plataforma Workers em conjunto com os Durable Objects para persistência de estado.

Os resultados obtidos (tendo como origem Ijuí, Rio Grande do Sul), nos mostram que a borda pode trazer vantagens mensuráveis na latência, obtendo uma diferença de 212 ms para 29 ms entre um servidor localizado nos Estados Unidos e um ponto de presença da Cloudflare no Brasil. A diferença é enorme considerando que a aplicação é implantada em todos os pontos de presença da Cloudflare, então se o acesso ocorrer dentro do Estados Unidos, ainda assim a latência pode ser menor do que na AWS. Essa exata diferença, mas no Brasil, pode ser constatada no teste realizado na AWS em São Paulo, que atingiu uma média de 61 ms e 29 ms na Cloudflare.

Figura 20 – Comparação do tempo de resposta entre a nuvem tradicional e a borda



Fonte: Autor

Um detalhe importante no estudo e análise de performance deste trabalho é que a aplicação na Cloudflare, quando testada, foi roteada para o ponto de presença localizado em São Paulo. Para fins de averiguação, foi utilizado o comando *tracert* no Windows para rastreamento de rota. Isso pode ter ocorrido por algum desvio temporário de rota do *backbone* ou por estado de manutenção do ponto de presença mais próximo na Cloudflare, que fica em Porto Alegre. Caso a rota alcançasse o PoP mais próximo, é possível afirmar com segurança que haveria uma redução ainda maior no tempo de resposta da requisição.

Outro ponto a considerar, é o motivo da discrepância entre a latência dos serviços, já que ambos redirecionaram as solicitações para São Paulo. O principal motivo para isso deve-se a tecnologia utilizada para persistir os dados. Enquanto na AWS foi utilizado um segundo servidor para executar o banco de dados (através do RDS), na Cloudflare foi utilizado o Durable Objects que funciona em conjunto com o Workers. Isso faz com que, no caso da AWS, haja um atraso ainda maior, pois é necessário trafegar pela rede interna antes de persistir e recuperar uma informação, além das próprias particularidades de implementação do MySQL.

5.3 Análise de Custos

Como visto na Seção 2.1.4, utilizando a infraestrutura *serverless*, os custos tendem a ser mais baixos do que a computação em nuvem tradicional. Isso se deve ao fato de que, ao criarmos uma máquina virtualizada, pré-definimos seus recursos computacionais. Dessa forma, quando ela está ociosa ou poucos recursos estão sendo ocupados, essa capacidade computacional está sendo desperdiçada, e ainda assim paga. A Figura 3 ilustra claramente o problema.

Ao contrário, em um ambiente *serverless*, a capacidade computacional é otimizada, pois o serviço é cobrado somente quando está sendo executado, gerando uma economia significativa. O modelo *serverless* usa apenas recursos computacionais no exato momento em que as requisições ocorrem, portanto, não há recurso computacional desperdiçado.

Tabela 4 – Tabela de precificação do Workers

	Plano Grátis	**Plano Pago (Bundled)**	**Plano Pago (Unbounded)**
Custo de ingresso	Grátis	US$ 5,00	US$ 5,00
Requisições	100.000/dia	10 milhões/mês, + US$ 0,50/milhão	1 milhões/mês, + US$ 0,15/milhão
Duração	10 ms	50 ms	400.000 GB-s, + US$ 12,50/milhão GB-s Até 30 segundos

Fonte: Cloudflare (2022e)

O nível gratuito para implantar um serviço na Cloudflare usando a plataforma Workers é generoso e oferece cerca de 3 milhões de requisições mensais em uma execução de CPU de até 10 milissegundos. Essa capacidade é suficiente para a maioria dos casos de uso simples, mas pode ser incrementada para até 30 segundos no plano pago.

Tabela 5 – Tabela de precificação do Durable Objects

	Plano Pago
Requisições	1 milhão, + US$ 0,15/milhão
Duração	400.000 GB-s, + US$ 12,50/milhão GB-s

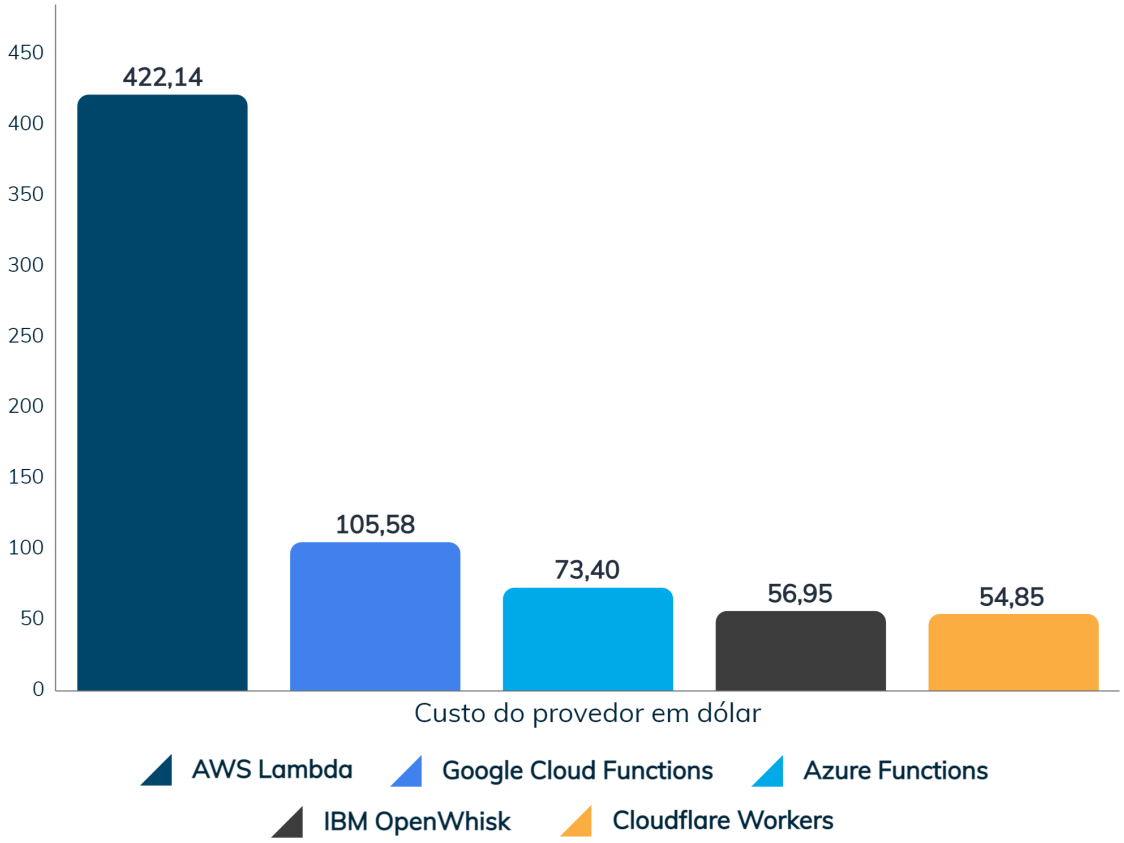
Fonte: Cloudflare (2022e)

Entretanto, a aplicação desenvolvida neste trabalho depende também de um serviço de coordenação e armazenamento, o Durable Objects, que tem uma precificação diferente do Workers. Pagando a taxa de ingresso de US$ 5,00 temos direito a 10 milhões de requisições mensais com tempo de CPU de 50 ms e mais 1 milhão de requisições para o DO, em um tempo total de execução de 400.000 GB-s.

Por exemplo, se um único Objeto Durável fosse chamado por um Worker 1.500.000 vezes e estivesse ativo por 1.000.000 de segundos no mês, o custo estimado em um mês seria de US$ 0,08 + US$ 5,00 de ingresso. Enquanto isso, a menor máquina disponibilizada na nuvem da AWS, modelo *t2.nano*, com 1 vCPUs e 0,5 GB de memória RAM, tem o custo de US$ 7,23/mês (AWS, 2022a). Assim, é fácil de imaginar que, quanto mais requisições *serverless* fizermos, mais barato será o custo total do serviço na Cloudflare, visto que a taxa de inscrição é fixa.

O modelo *t2.nano* é capaz de executar esta aplicação perfeitamente em baixa escala. Todavia, no momento em que recebermos um pico de acessos, ou apenas ocorra o aumento de tráfego gradual, torna-se inevitável adicionar horizontalmente mais servidores e configurar um balanceador de carga na frente da aplicação. Todo esse processo aumenta ainda mais os custos do modelo de nuvem tradicional, enquanto na borda da Cloudflare tudo isso ocorre automaticamente com distribuição global e sem gastos extras.

Figura 21 – Custo simulado de 100 milhões de requisições HTTP *serverless* com 250 ms de tempo de execução



Fonte: Sbarski (2022) e Cerulean (2022)

Para compreender a diferença de custo entre os provedores de infraestrutura *serverless* na nuvem e a Cloudflare, que atua na borda, foram simuladas 100 milhões de requisições HTTP, com um tempo de execução total de 250 milissegundos e 128 MB de memória RAM. O resultado,

apresentado na Figura 21, mostra a enorme diferença entre os provedores. A IBM é o fornecedor que se aproxima mais da Cloudflare em termos de preço, no entanto, quanto mais recursos são adicionados nos simuladores, maior é a diferença entre os dois. O AWS Lambda (não confundir com Lambda@Edge) é o fornecedor mais caro, chegando a 4 vezes o valor do segundo, o Google Cloud Functions. Por fim, o serviço Azure Functions da Microsoft ficou no meio-termo nesta análise. No entanto, é importante salientar que o Cloudflare Workers distribui a aplicação mundialmente na borda, diferentemente dos outros 4 serviços apresentados.

5.4 Limitações da Aplicação

Enquanto a aplicação suporta um número muito grande de clientes conectados, pode-se considerar que um ser humano provavelmente só pode acompanhar, no máximo, algumas mensagens de bate-papo por segundo. Mas a infraestrutura da Cloudflare pode lidar com ordens de magnitude muito maiores do que isso. Portanto, o gargalo de escalabilidade para o caso de uso de uma sala de bate-papo é, na verdade, humano, e não limitante à tecnologia utilizada. Muitos casos de uso acabam sendo assim.

Se de alguma maneira a aplicação precisasse oferecer suporte a uma sala de conversação com mais tráfego do que uma classe do Durable Objects pode manipular (serviço utilizado na concepção deste trabalho), então a melhor maneira para corrigir isso é alterando a arquitetura da aplicação. Para tanto, é preciso fragmentar o objeto, forçando com que cada cliente se conecte a todos os fragmentos e, então, escolher aleatoriamente um desses objetos para lidar com cada mensagem enviada.

Se o problema for o número de clientes conectados — e não na frequência das mensagens enviadas —, outra abordagem seria conectar cada cliente a um único objeto, e então forçar com que cada objeto tenha as mensagens retransmitidas entre si. Desta forma podemos aumentar os recursos computacionais, já que cada instância (e cada cliente) teria 128 MB de memória RAM e processamento de CPU exclusivo.

Também é importante observar que, caso o serviço implantado na borda exija um banco de dados relacional, somente o Lambda@Edge poderá suprir tal necessidade. Isso ocorre porque os mecanismos de tempo de execução da Cloudflare e Fastly na *edge* não suportam conexões TCP. Para resolver esse problema, a Cloudflare anunciou seu novo produto, D1, que deve entrar em testes (fase beta) nos próximos meses. No entanto, para desenvolver um aplicativo completo na borda e independente de serviços externos, o Cloudflare Workers é o único concorrente que traz as ferramentas necessárias para isso.

5.5 Considerações do Capítulo

Neste capítulo foram descritos os resultados obtidos a partir da análise das avaliações de três pontos fundamentais: viabilidade, performance e custo. Além da exposição destes pontos,

também foi traçado um paralelo entre a nuvem tradicional e a borda em aplicações semelhantes a deste trabalho. Por fim, foram exploradas algumas possíveis limitações para a aplicação construída.

6 CONCLUSÃO E TRABALHOS FUTUROS

Neste capítulo serão apresentadas as conclusões obtidas no desenvolvimento do presente trabalho, bem como possibilidades de expansão do estudo em trabalhos futuros.

6.1 Conclusão

Este trabalho apresentou o desenvolvimento de um sistema de bate-papo em tempo real utilizando infraestrutura na Serverless Edge Computing. O sistema descrito permite criar salas públicas, onde cada sala criada é uma nova instância construída por um Durable Object. Cada sala criada se comporta como uma aplicação independente, com seus próprios recursos computacionais e armazenamento. Ela também é gerada a partir do ponto de presença mais próximo do usuário que a originou.

O trabalho alcançou o objetivo de criar um sistema de chat em tempo real cuja proposta era ser escalável para atender demandas infinitamente crescentes, sem a necessidade de grandes investimentos e sem gerenciamento de infraestrutura. Além disso, o sistema oferece baixa latência e persistência dos dados gerados próximos ao usuário, o que torna a solução mais atrativa em termos de experiência de uso.

Apesar dos bons resultados alcançados, este trabalho deve ser considerado um protótipo inicial, que pode ser melhorado em muitos aspectos. Além da necessidade de uma interface gráfica mais completa, o sistema pode ser otimizado a fim de melhorar a experiência do usuário, como trazer a listagem de salas já criadas, exibir quando alguém está digitando, salvar os apelidos já definidos e exibir a data do envio das mensagens. Outro ponto que deve ser considerado é a segurança, já que o sistema atual não implementa salas privadas e, tampouco, algum mecanismo para garantir que as mensagens trocadas sejam criptografadas quando persistidas.

6.2 Trabalhos Futuros

Em trabalhos futuros, um conjunto de possibilidades surge no contexto deste trabalho. Para a aplicação construída neste trabalho, alguns recursos relevantes poderiam ser adicionados, como inserir um limite através de uma taxa de envios de mensagens por minuto, evitando *spam* (*flood*).

Outra possível extensão para este trabalho seria a implementação de comunicação em tempo real por vídeo: há poucos meses a Cloudflare introduziu seu novo serviço de stream usando computação *serverless*. Anexar funções semelhantes ao Zoom ou Google Meet na borda se tornam viáveis com esse serviço, aumentando ainda mais os possíveis caso de uso para esta tecnologia.

Por último, uma possível linha de estudo no emprego da tecnologia de borda, é o desenvolvimento de uma aplicação de edição colaborativa de documentos em tempo real — assim como um bate-papo, ela também exige coordenação, e o Workers se encaixa bem neste cenário. Ou, ainda, aplicações como um carrinho de compras, *feed* de redes sociais, servidor de jogos e até mesmo coordenação de múltiplos dispositivos IoT.

REFERÊNCIAS

AHMED, M. **Kubernetes Native Edge Computing Framework, kubeEdge**. 2021. Disponível em: <https://dev.to/ahmedmansoor012/kubernetes-native-edge-computing-framework-kubeedge-6e8>. Acesso em: 30 mai 2022.

AWS. **AWS Pricing Calculator**. 2022. Disponível em: <https://calculator.aws/#/createCalculator/EC2>. Acesso em: 12 jun 2022.

AWS. **Preço do Lambda@Edge**. 2022. Disponível em: <https://aws.amazon.com/pt/lambda/pricing/>. Acesso em: 05 jun 2022.

AWS. **Using AWS Lambda with CloudFront Lambda@Edge**. 2022. Disponível em: <https://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html>. Acesso em: 12 mai 2022.

BANDAKKANAVAR, R. **What is Serverless Edge Computing? Features and Benefits**. 2022. Disponível em: <https://krazytech.com/technical-papers/serverless-edge-computing>. Acesso em: 8 jun 2022.

BECK, K. **Manifesto for Agile Software Development**. 2001. Disponível em: <http://agilemanifesto.org>. Acesso em: 16 out 2021.

BESWICK, J. **Operating Lambda**. 2021. Disponível em: <https://aws.amazon.com/blogs/compute/operating-lambda-performance-optimization-part-1>. Acesso em: 12 mai 2022.

BHOSALE, S. **Research Paper On Cloud Computing**. [S.l.]: Contemporary Research In India, 2021.

BOGU, B. **Chat online: um aliado indispensável nas vendas internas**. 2021. Disponível em: <https://www.estudevendas.com.br/chat-online-um-aliado-indispensavel-nas-vendas-internas>. Acesso em: 04 mai 2022.

CERULEAN. **Unofficial Cloudflare Workers® pricing calculator**. 2022. Disponível em: <https://pricing.ceru.dev/?rpm=100000000&spr=0.25>. Acesso em: 18 jun 2022.

CHAN, A. **What is an Edge Location in AWS? A Simple Explanation**. 2021. Disponível em: <https://www.lastweekinaws.com/blog/what-is-an-edge-location-in-aws-a-simple-explanation/>. Acesso em: 18 mar 2022.

CHAUDHARI, N.; SHINKAR, S.; PAGARE, P. **Chatting Application with Real Time Translation**. [S.l.]: Sanghavi college of Engineering, 2018.

CLOUDFLARE. **The Cloudflare global network**. 2022. Disponível em: <https://www.cloudflare.com/en-gb/network/>. Acesso em: 01 jun 2022.

CLOUDFLARE. **Deploy serverless code instantly**. 2022. Disponível em: <https://workers.cloudflare.com/>. Acesso em: 02 jun 2022.

CLOUDFLARE. **How Workers works**. 2022. Disponível em: <https://developers.cloudflare.com/workers/learning/how-workers-works/>. Acesso em: 01 jun 2022.

CLOUDFLARE. **What is serverless computing?** 2022. Disponível em: <https://www.cloudflare.com/learning/serverless/what-is-serverless/>. Acesso em: 28 mar 2022.

CLOUDFLARE. **Workers Pricing**. 2022. Disponível em: <https://developers.cloudflare.com/workers/platform/pricing/>. Acesso em: 12 jun 2022.

COLL, B. **Miniflare: fully-local development and testing for Workers**. 2022. Disponível em: <https://blog.cloudflare.com/miniflare/>. Acesso em: 04 jun 2022.

COSTA, R. **Mark Zuckerberg compartilha fotos do data center mais frio do Facebook**. 2016. Disponível em: <https://www.tecmundo.com.br/facebook/110005-mark-zuckerberg-compartilha-fotos-data-center-frio-do-facebook.htm>. Acesso em: 16 mar 2022.

ELLERBY, B. **Why Serverless will enable the Edge Computing Revolution**. 2021. Disponível em: <https://medium.com/serverless-transformation/why-serverless-will-enable-the-edge-computing-revolution-4f52f3f8a7b0>. Acesso em: 15 mar 2022.

FASTLY. **Fastly Compute@Edge pricing**. 2022. Disponível em: <https://www.fastly.com/pricing/>. Acesso em: 12 jun 2022.

FASTLY. **Learn to build on the Fastly platform**. 2022. Disponível em: <https://developer.fastly.com/learning/compute>. Acesso em: 16 mai 2022.

FEE, N. **What Is Serverless Architecture? Key Benefits and Limitations**. 2020. Disponível em: <https://newrelic.com/blog/best-practices/what-is-serverless-architecture>. Acesso em: 15 mar 2022.

GUPTA, M. **Content Delivery Network Approach to Improve Web Performance: A Review**. [S.l.]: Maharishi Markandeshwar University, 2015.

JETBRAINS. **The smartest JavaScript IDE**. 2022. Disponível em: <https://www.jetbrains.com/webstorm>. Acesso em: 04 mai 2022.

KEMPF, R. **Pushing Computation to the Edge With Serverless Compute**. 2021. Disponível em: <https://www.azion.com/en/blog/pushing-computation-to-edge-serverless-compute>. Acesso em: 16 mar 2022.

KOZLOV, R. **The next chapter for Cloudflare Workers: open source**. 2022. Disponível em: <https://blog.cloudflare.com/workers-open-source-announcement/>. Acesso em: 25 mai 2022.

KOZLOV, R.; MADDERN, G. **Announcing D1: our first SQL database**. 2022. Disponível em: <https://blog.cloudflare.com/introducing-d1/>. Acesso em: 04 jun 2022.

KUMAR, A.; SINGH, A. **Group Chatting Application**. [S.l.]: Galgotias University, 2019.

KUPERMAN, J. **WebSockets**. 2021. Disponível em: <https://developers.cloudflare.com/workers/runtime-apis/websockets/>. Acesso em: 05 jun 2022.

LINDSAY, G. **Visão geral do suporte para WebSocket no Gateway de Aplicativo**. 2022. Disponível em: <https://docs.microsoft.com/pt-br/azure/application-gateway/application-gateway-websocket>. Acesso em: 04 jun 2022.

LUTKEVICH, B. **SearchData Management**. 2021. Disponível em: <https://www.techtarget.com/searchdatamanagement/definition/database>. Acesso em: 01 mai 2022.

MCKEON, G. **R2 Storage: Rapid and Reliable Object Storage, minus the egress fees**. 2021. Disponível em: <https://blog.cloudflare.com/introducing-r2-object-storage/>. Acesso em: 02 jun 2022.

MCKEON, G. **Workers adds support for two modern data platforms: MongoDB Atlas and Prisma**. 2021. Disponível em: <https://blog.cloudflare.com/workers-adds-support-for-two-modern-data-platforms-mongodb-atlas-and-prisma/>. Acesso em: 05 jun 2022.

MCKEON, G. **Workers KV - free to try, with increased limits**. 2022. Disponível em: <https://blog.cloudflare.com/workers-kv-free-tier/>. Acesso em: 02 jun 2022.

MEULEN, R. van der. **What Edge Computing Means for Infrastructure and Operations Leaders**. 2018. Disponível em: <https://www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders>. Acesso em: 18 jun 2022.

MICROSOFT. **Why did we build Visual Studio Code?** 2021. Disponível em: <https://code.visualstudio.com/docs/editor/whyvscode>. Acesso em: 05 mai 2021.

NANWANI, R. **What is a Content Delivery Network (CDN)**. 2021. Disponível em: <https://imagekit.io/blog/what-is-content-delivery-network-cdn-guide>. Acesso em: 30 mar 2021.

NEOVIM. **Hyperextensible Vim-based text editor**. 2022. Disponível em: <https://neovim.io/charter>. Acesso em: 04 mai 2022.

OBADJERE, E. N. **Design and Implementation of a Real Time Chat Application**. [S.l.]: Baze University, 2020.

OKERE, C. **How to Manipulate the DOM**. 2021. Disponível em: <https://www.freecodecamp.org/news/how-to-manipulate-the-dom-beginners-guide>. Acesso em: 12 jun 2022.

OKTA. **Serverless Computing: Uses, Advantages, and Disadvantages**. 2021. Disponível em: <https://www.okta.com/identity-101/serverless-computing/>. Acesso em: 15 mar 2022.

PALLIS, G. **Edge Computing**. [S.l.]: University of Cyprus: Nicosia, CY, 2019.

POULTER, A. J.; JOHNSTON, S. J.; COX, S. J. **Using the MEAN stack to implement a RESTful service for an Internet of Things application**. [S.l.]: IEEE, 2015.

PRATT, M. K. **Top 5 benefits of edge computing for businesses**. 2021. Disponível em: <https://www.techtarget.com/iotagenda/tip/Top-5-benefits-of-edge-computing-for-businesses>. Acesso em: 15 mar 2022.

SBARSKI, P. **Serverless Cost Calculator**. 2022. Disponível em: <http://serverlesscalc.com>. Acesso em: 18 jun 2022.

SILVA, D. da. **Como alavancar as vendas na internet via chat em tempo real?** 2021. Disponível em: <https://www.zendesk.com.br/blog/como-alavancar-vendas/>. Acesso em: 04 mai 2022.

SOCKET.IO. **Bidirectional and low-latency communication for every platform**. 2017. Disponível em: <https://socket.io/>. Acesso em: 02 jun 2022.

SOOKOCHEFF, K. **How Do Websockets Work?** 2019. Disponível em: <https://sookocheff.com/post/networking/how-do-websockets-work>. Acesso em: 12 jun 2022.

STACKOVERFLOW. **Insights of most popular technologies**. 2021. Disponível em: <https://insights.stackoverflow.com/survey/2021#most-popular-technologies-new-collab-tools>. Acesso em: 12 mai 2022.

SUBLIME HQ. **Text Editing, Done Right**. 2022. Disponível em: <https://www.sublimetext.com/docs/index.html>. Acesso em: 04 mai 2022.

SULIEMAN, N. A. **Edge-Oriented Computing: A Survey on Research and Use Cases**. [S.l.]: MDPI journals, 2022.

THAKUR, A.; DHIMAN, K. **Chat Room Using HTML, PHP, CSS, JS, AJAX**. [S.l.]: University of Mumbai, 2021.

THARANIDHARAN, S. K. et al. **Real Time Web Based Multilingual Chat Application**. [S.l.]: University Punjab, 2022.

VALERIANO, D. **Gerência em Projetos: Pesquisa, Desenvolvimento e Engenharia**. [S.l.]: Pearson Universidades, 1998.

VARDA, K. **Run JavaScript Service Workers at the Edge**. 2017. Disponível em: <https://blog.cloudflare.com/introducing-cloudflare-workers/>. Acesso em: 16 mai 2022.

VARDA, K. **Workers Durable Objects Beta: A New Approach to Stateful Serverless**. 2020. Disponível em: <https://blog.cloudflare.com/introducing-workers-durable-objects/>. Acesso em: 02 jun 2022.

Apêndices

APÊNDICE A – DURABLE OBJECT

```
export class ChatRoom {
  constructor(state, env) {
    this.storage = state.storage;
    this.env = env;
    this.sessions = [];
  }

  async fetch(request) {
    const url = new URL(request.url);

    if (url.pathname === "/websocket") {
      const [client, server] = Object.values(new WebSocketPair());
      await handleSession(server);
      return new Response(null, { status: 101, webSocket: client });
    }

    return new Response("Not Found", { status: 404 });
  }

  async handleSession(webSocket) {
    webSocket.accept();

    const session = { webSocket, recoverMessages: [] };
    this.sessions.push(session);

    this.sessions.forEach((otherSession) => {
      if (otherSession.name) {
        session.recoverMessages.push(JSON.stringify({ joined: otherSession.name }))
      }
    });

    const storage = await this.storage.list({ reverse: true, limit: 100 });
    const recoveredMessages = [...storage.values()].reverse();
    recoveredMessages.forEach((value) => {
      session.recoverMessages.push(value);
    });
```

```
37
38      let messagesAlreadyRecovered = false; // only allow one time to recover message
39
40      webSocket.addEventListener("message", messageHandler);
41      webSocket.addEventListener("close", closeHandler);
42
43      const messageHandler = async (msg) => {
44        const sentMessage = JSON.parse(msg.data);
45
46        if (!messagesAlreadyRecovered) {
47          session.name = String(data.name || "anonymous");
48
49          session.recoverMessages.forEach((queued) => {
50            webSocket.send(queued);
51          });
52          delete session.recoverMessages; // recove messages and delete queue
53
54          this.broadcast({ joined: session.name });
55          webSocket.send(JSON.stringify({ ready: true }));
56
57          messagesAlreadyRecovered = true;
58          return;
59        }
60
61        const data = JSON.stringify({
62          name: session.name,
63          message: String(sentMessage.message),
64          timestamp: Date.now(),
65        });
66
67        this.broadcast(data);
68
69        const key = new Date(data.timestamp).toISOString();
70        await this.storage.put(key, data);
71      };
72
73      const closeHandler = () => {
74        this.sessions = this.sessions.filter((member) => member !== session);
75        if (session.name) {
76          this.broadcast({ quit: session.name });
77        }
```

```
78      };
79    }
80
81    broadcast(message) {
82      this.sessions = this.sessions.filter((session) => {
83        if (session.name) {
84          session.webSocket.send(message);
85          return true;
86        }
87      });
88    }
89  }
```