UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO INFORMÁTICA
GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

DAYANE LIRA DA SILVA

Análise de Desempenho de Soluções Serverless para Jogos Multiplayer

Recife

2025

DAYANE LIRA DA SILVA

Análise de Desempenho de Soluções Serverless para Jogos Multiplayer

Trabalho apresentado ao Programa de Graduação em Engenharia da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

Área de Concentração: Análise de Desempenho, Computação em Nuvem

Orientador: Carlos André Guimarães Ferraz

Recife

2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Silva, Dayane Lira da.
Análise de Desempenho de Soluções Serverless para Jogos Multiplayer / Dayane Lira da Silva. - Recife, 2025.
52 p. : il., tab.

Orientador(a): Carlos André Guimarães Ferraz
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, , 2025.
Inclui referências, apêndices, anexos.

1. Computação em Nuvem. 2. Jogos. 3. Análise de desempenho. I. Ferraz, Carlos André Guimarães. (Orientação). II. Título.

600 CDD (22.ed.)

DAYANE LIRA DA SILVA

ANÁLISE DE DESEMPENHO DE SOLUÇÕES SERVERLESS PARA JOGOS MULTIPLAYER

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Engenharia da Computação.

Aprovado em: 07/08/2025

BANCA EXAMINADORA

_____

Prof. Dr. Carlos André Guimarães Ferraz (Orientador)

Universidade Federal de Pernambuco

_____

Prof. Dr. Kiev Santos da Gama (Examinador Interno)

Universidade Federal de Pernambuco

AGRADECIMENTOS

Agradeço, em primeiro lugar, a Deus, por sempre ter me dado força para seguir meus objetivos e coragem para recomeçar quando necessário, mesmo diante dos desafios que surgiram ao longo do caminho.

À minha mãe, Luciene, que sempre me incentivou a sonhar e a acreditar no meu potencial. Obrigada por cada conselho, por seu amor incondicional e por todos os sacrifícios feitos por mim ao longo dessa caminhada, que foram fundamentais para que eu pudesse chegar até aqui.

Aos meus avós, Maria e Adeildo, que sempre estiveram presentes em minha vida, sendo atenciosos e preocupados quando mais precisei. Muito obrigada pelo carinho e amor.

Ao meu namorado, Leonardo, pela paciência, amor e apoio nos momentos mais conturbados da faculdade. Tudo isso foi essencial para que eu mantivesse o foco e a motivação nos momentos difíceis.

Aos meus colegas de turma, Alice, Ana, Guilherme, Lucca, Mateus, Victoria e Williams, a quem agradeço por toda a companhia, incentivo e pelas risadas. Com vocês, dividi momentos de alegria e frustração, e tudo se tornou mais leve e especial. Minha gratidão por tudo!

Ao Kinho, meu monitor de programação no primeiro período, que se tornou um dos meus melhores amigos. Sua paciência e dedicação foram essenciais para minha adaptação no curso. Agradeço imensamente por toda a ajuda e por me mostrar que a jornada seria mais fácil com amigos ao lado.

Aos professores e professoras que fizeram parte da minha jornada, muito obrigada por compartilharem seus conhecimentos com dedicação e por me inspirarem a buscar sempre mais. Levo comigo cada aprendizado, dentro e fora da sala de aula, que certamente contribuirá para minha vida profissional e pessoal.

E um agradecimento especial à minha calopsita Zequinha, que me acompanhou durante tantas horas de estudo e, com certeza, aprendeu a programar antes de mim. Mesmo sem saber, tornou minhas horas de estudo mais leves, fazendo dos momentos de dedicação algo ainda mais especial.

Where do we go now?

— Guns N' Roses

RESUMO

Este trabalho avalia o desempenho de uma arquitetura *serverless* aplicada ao desenvolvimento de jogos multiplayer online, comparando-a com o modelo tradicional cliente-servidor. Para isso, foram desenvolvidas duas versões funcionais de um jogo da velha: uma baseada em FastAPI e MongoDB executada em uma instância EC2 (cliente-servidor), e outra utilizando serviços gerenciados da AWS como API Gateway, Lambda e DynamoDB (*serverless*). Foram realizados testes de carga simulando até 1000 usuários concorrentes, utilizando a ferramenta Locust. Os resultados mostraram que, embora o modelo cliente-servidor tenha se destacado com baixa carga, apresentou degradação significativa com o aumento da concorrência. Em contraste, a arquitetura *serverless* demonstrou melhor escalabilidade e estabilidade, oferecendo tempos de resposta mais previsíveis em cenários de alta demanda. Conclui-se que a computação *serverless* é uma abordagem promissora no contexto abordado, embora traga desafios como *cold starts* e gerenciamento de estado.

Palavras-chaves: Serverless, Jogos Multiplayer, Desempenho, Arquitetura Cliente-Servidor.

ABSTRACT

This work evaluates the performance of a serverless architecture applied to the development of online multiplayer games, comparing it with the traditional client-server model. To this end, two functional versions of a tic-tac-toe game were developed: one based on FastAPI and MongoDB running on an EC2 instance (client-server), and another using AWS managed services such as API Gateway, Lambda, and DynamoDB (serverless). Load tests simulating up to 1000 concurrent users were conducted using the Locust tool. The results showed that, although the client-server model performed well under low load, it experienced significant degradation as concurrency increased. In contrast, the serverless architecture demonstrated better scalability and stability, offering more predictable response times in high-demand scenarios. It is concluded that serverless computing is a promising approach in the evaluated context, although it presents challenges such as cold starts and state management.

Keywords: Serverless, Multiplayer Games, Performance, Client-Server Architecture.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Com a popularização da internet, jogar videogame se tornou uma atividade cada vez mais social (DUCHENEAUT; MOORE, 2004). Atualmente, milhares de jogadores se conectam simultaneamente em plataformas online para disputar partidas, cooperar em missões ou simplesmente interagir em tempo real. Porém, proporcionar esse tipo de experiência exige que os desenvolvedores enfrentem desafios relacionados à infraestrutura, como garantir que os servidores suportem a alta demanda de conexões, mantenham a estabilidade do sistema e respondam com baixa latência às ações dos usuários.

Tradicionalmente, a maioria dos jogos *multiplayer* online utiliza a arquitetura cliente-servidor, como destacado por (WU et al., 2020). Essa abordagem é adotada principalmente por sua simplicidade de implementação e pelo controle centralizado dos dados e da lógica da aplicação. No entanto, segundo o autor, esse modelo apresenta limitações importantes: além dos altos custos de manutenção associados à operação de servidores dedicados, ele exige que o desenvolvedor tenha conhecimentos específicos de infraestrutura, como provisionamento, monitoramento, balanceamento de carga e escalabilidade manual, o que pode aumentar a complexidade do desenvolvimento. Além disso, essa abordagem também assume um risco estrutural, pois apresenta um ponto único de falha. Ou seja, se o servidor central ficar indisponível, seja por falhas de hardware, sobrecarga, erros de software ou ataques externos, todo o sistema pode se tornar inacessível, afetando negativamente a experiência do jogo.

Com o avanço da computação em nuvem, surgiram novas alternativas arquiteturais, entre as quais está o modelo *serverless* (ADZIC; CHATLEY, 2017). Nessa abordagem, a responsabilidade pela alocação e gerenciamento da infraestrutura é transferida para os provedores de nuvem, permitindo que o desenvolvedor se concentre exclusivamente na lógica da aplicação. A arquitetura *serverless* se baseia em funções pequenas e independentes, que são executadas sob demanda em uma infraestrutura altamente distribuída. Isso elimina a dependência de um único ponto central, tornando o sistema mais resiliente a falhas.

Além disso, provedores *serverless*, como AWS, Azure e Google Cloud, oferecem mecanismos automáticos de balanceamento de carga, replicação geográfica e recuperação de falhas, o que contribui para uma maior disponibilidade e robustez do sistema. A escalabilidade também ocorre de forma automática, com os recursos sendo provisionados de acordo com a demanda, o que pode resultar em redução de custos operacionais, maior elasticidade e baixos níveis de

latência, mesmo em cenários de pico de usuários.

Diante desse cenário, este trabalho propõe uma análise comparativa de desempenho entre as arquiteturas cliente-servidor e *serverless*, aplicadas ao desenvolvimento de um jogo *multiplayer* simples, o jogo da velha. O objetivo principal é investigar a viabilidade de utilizar a arquitetura *serverless* para jogos *multiplayer* online.

A estrutura deste trabalho está dividida em capítulos, conforme descrito a seguir:

- **Capítulo 2 - Trabalhos Relacionados**: Apresenta estudos anteriores que abordam o uso de arquiteturas *serverless* em jogos, servindo de base para a contextualização deste trabalho.
- **Capítulo 3 - Fundamentação Teórica**: Aborda os principais conceitos sobre jogos *multiplayer*, computação em nuvem, arquiteturas cliente-servidor e *serverless*, além de testes de desempenho.
- **Capítulo 4 - Metodologia**: Descreve o tipo de pesquisa, os procedimentos adotados e os critérios definidos para a análise comparativa entre as arquiteturas.
- **Capítulo 5 - Desenvolvimento**: Detalha a implementação das duas versões do jogo da velha e as configurações dos testes de carga, descrevendo as ferramentas utilizadas.
- **Capítulo 6 - Resultados**: Apresenta os resultados obtidos nos teses e realiza a análise comparativa entre as arquiteturas, destacando vantagens e limitações de cada uma.
- **Capítulo 7 - Conclusão**: Sintetiza as principais contribuições do trabalho e propõe possíveis direções para pesquisas futuras.

2 TRABALHOS RELACIONADOS

A adoção de arquiteturas *serverless* tem ganhado espaço nos últimos anos, principalmente entre equipes e desenvolvedores interessados em focar nas funcionalidades de suas aplicações e não na infraestrutura onde ela vai ser executada. Porém, seu uso no contexto de jogos ainda é pouco explorado.

O trabalho de (IANCU, 2021) propõe uma reestruturação de arquitetura usando abordagem *serverless* para lidar com os desafios de escalabilidade em ambientes virtuais modificáveis (MVEs – *Modifiable Virtual Environments*), como o do jogo *Minecraft*. Ambientes virtuais modificáveis são ambientes virtuais onde os usuários podem interagir, modificar, criar novos conteúdos e programar novas interações. A pesquisa parte da afirmação de que servidores tradicionais de jogos como *Minecraft* não escalam bem. Mesmo com múltiplos núcleos e infraestrutura de *datacenters*, eles enfrentam dificuldades para suportar muitos jogadores concorrentes. Essa dificuldade está relacionada diretamente com a geração de conteúdo do mundo virtual.

Para enfrentar esse desafio, os autores propuseram um protótipo híbrido *serverless*, no qual o componente de geração de conteúdo do mundo virtual é reestruturado para funcionar de maneira sob demanda com uso de funções *serverless*. O estudo conclui que, embora existam vantagens relacionadas à elasticidade, essa abordagem possui limitações como *cold start* e dificuldade de coordenação de múltiplas funções em tempo real.

Outro estudo relevante é o trabalho de (OSMAN, 2025), que propõe e implementa um sistema de placar para jogos utilizando abordagem *serverless* e Microsoft Azure. A solução utiliza Azure Functions para processar em tempo real os dados de pontuação dos jogadores e armazenar em um banco de dados (MongoDB Atlas). A motivação de (OSMAN, 2025) é a ausência de preocupação relacionada à infraestrutura que a abordagem *serverless* provê. O autor concluiu que a abordagem *serverless* diminuiu a complexidade relacionada à infraestrutura, demonstrou boa escalabilidade e manutenção simplificada, apesar de ter enfrentado desafios relacionados a picos de latência introduzidos por *cold start* e complexidade em manter a comunicação em tempo real.

Embora os trabalhos de (IANCU, 2021) e (OSMAN, 2025) se aproximem diretamente da proposta deste estudo ao aplicar *serverless* em jogos e avaliar seus impactos, os autores optaram por migrar apenas um componente específico do jogo. Em contraste, este estudo propõe a

migração completa da lógica do jogo para uma arquitetura *serverless*, permitindo uma análise mais abrangente, envolvendo todo o ciclo de execução da aplicação.

3 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, são apresentados os conceitos teóricos que são necessários para a análise realizada ao longo do trabalho. O objetivo é contextualizar as tecnologias e abordagens utilizadas. Primeiro, são discutidas as principais características relacionadas a jogos *multiplayer*; em seguida, são abordados os fundamentos da computação em nuvem. Na sequência, são colocadas em pauta as arquiteturas cliente-servidor e *serverless*, destacando suas características, vantagens e limitações. Por fim, o capítulo trata sobre testes de desempenho de software.

3.1 JOGOS MULTIPLAYER ONLINE

Jogos *multiplayer* online são um tipo específico de videogame caracterizados pela presença de múltiplos usuários e pela inclusão de recursos de interação social online (L BIGNILL J; L, 2021). Esses jogos possibilitam um ambiente virtual onde jogadores de todos os lugares do mundo podem interagir, colaborar e competir uns com os outros, dentro das capacidades do jogo (COLE S. H., 2013).

Dada a natureza dos jogos *multiplayer* online, eles demandam uma infraestrutura capaz de suportar múltiplos jogadores de forma sincronizada e em tempo real. Para isso, é necessário lidar com desafios como latência, consistência de estado, sincronização de eventos e suporte a conexões simultâneas.

3.2 COMPUTAÇÃO EM NUVEM

Computação em nuvem é atualmente um mecanismo para permitir, de forma facilitada, conveniente e sob demanda, o acesso a vários recursos como servidores, redes, dispositivos de armazenamento, aplicações e serviços (PAUL; GHOSE, 2012).

De acordo com (ZHANG Q., 2010), a principal característica da computação em nuvem é a virtualização dos recursos, que possibilita sua alocação dinâmica conforme a necessidade da aplicação. Com isso, sistemas podem ser escalados vertical ou horizontalmente, de forma automática ou programada, sem a necessidade de intervenções físicas ou reconfigurações complexas. Essa elasticidade permite que soluções em nuvem se ajustem rapidamente a variações de carga, mantendo o desempenho e a disponibilidade dos serviços.

Outro aspecto central abordado por Zhang é o modelo de cobrança *pay-as-you-go*, no qual as organizações pagam apenas pelos recursos efetivamente utilizados. Isso representa uma vantagem significativa em comparação a modelos tradicionais, nos quais é necessário adquirir e manter infraestrutura ociosa para suportar picos de demanda. Como resultado, empresas podem reduzir seus custos operacionais e de manutenção, além de aumentar a eficiência no uso dos recursos computacionais.

Além disso, o autor destaca que a nuvem elimina a dependência de data centers físicos próprios, transferindo a responsabilidade de gerenciamento, segurança e escalabilidade para provedores especializados. Essa abordagem facilita a adoção de soluções mais ágeis, seguras e resilientes, características fundamentais para aplicações modernas, como sistemas distribuídos e jogos online com alta demanda de conectividade.

O National Institute of Standards and Technology (NIST) (MELL; GRANCE, 2011) define que a computação em nuvem é composta por cinco características essenciais, três modelos de serviço e quatro modelos de implantação.

3.2.1 Características essenciais

- **Autoatendimento sob demanda**: Um consumir pode de maneira unilateral provisionar recursos computacionais tais como tempo de servidor e armazenamento, de forma automática, sem a necessidade de interações humanas com cada provedor de serviço.
- **Amplo acesso à rede**: Os recursos estão disponíveis por meio da rede e podem ser acessados por mecanismos padronizados, como celulares, tablets, laptops e estações de trabalho.
- **Agrupamento de recursos**: Os recursos computacionais do provedor são agrupados para atender a múltiplos consumidores usando um modelo multi-inquilino (*multi-tenant*), com recursos físicos e virtuais sendo atribuídos e realocados dinamicamente de acordo com a demanda do consumidor.
- **Elasticidade rápida**: Os recursos podem ser provisionados e liberados de forma elástica, e, em alguns casos, automaticamente.
- **Serviço mensurado**: Os sistemas em nuvem controlam e otimizam automaticamente o uso dos recursos por meio de mecanismos de medição em algum nível de abstração

apropriado ao tipo de serviço (por exemplo, armazenamento, processamento, largura de banda e contas de usuários ativas). O uso dos recursos pode ser monitorado, controlado e relatado, oferecendo transparência tanto para o provedor quanto para o consumidor do serviço utilizado.

3.2.2 Modelos de serviço

- **Software como serviço (SaaS – *Software as a Service*)**: Refere-se à configuração de uma nuvem que permite o compartilhamento de uma aplicação. Essa aplicação é hospedada em um servidor e acessada pelo usuário por meio de diferentes dispositivos conectados à rede. Embora o usuário possa em alguns casos modificar algumas opções menores da aplicação, ela é executada e configurada a partir de um servidor que não é acessível ao usuário. Um exemplo de SaaS é o Gmail ou outro serviço de e-mail acessível por meio de um navegador da internet.
- **Plataforma como serviço (PaaS – *Platform as a Service*)**: O PaaS fornece o ambiente computacional adequado para que o cliente possa desenvolver, instalar e implantar aplicações ou outros serviços. Trata-se de disponibilizar uma máquina já configurada (com hardware previamente definido, sistema operacional e softwares necessários).
- **Instraestrutura como serviço (IaaS – *Infrastructure as a Service*)**: O objetivo aqui é alugar toda a insfraestrutura de hardware de um terceiro (servidores, rede e outros recursos). O usuário tem a liberdade de implantar qualquer tipo de software, incluindo sistemas operacionais.

3.2.3 Modelos de implantação

- **Nuvem privada**: A infraestrutura de nuvem é provisionada para uso exclusivo de uma única organização. Essa nuvem pode ser gerenciada e operada pela própria organização, por um terceiro, ou por uma combinação de ambos, e pode estar localizada dentro ou fora das instalações da organização.
- **Nuvem comunitária**: A infraestrutura de nuvem é provisionada para uso exclusivo de uma comunidade específica de consumidores, composta por diferentes organizações que

compartilham interesses ou requisitos em comum (por exemplo, missão, exigências de segurança, políticas ou conformidade regulatória).

- **Nuvem pública**: A infraestrutura de nuvem é disponibilizada para uso aberto pelo público em geral.
- **Nuvem híbrida**: A infraestrutura de nuvem é composta por duas ou mais infraestruturas de nuvem distintas (privada, comunitária ou pública) que permanecem como entidades separadas, mas são conectadas por tecnologias padronizadas ou proprietárias que permitem a portabilidade de dados e aplicações entre elas (por exemplo, cloud bursting para balanceamento de carga entre nuvens).

3.3 ARQUITETURA CLIENTE-SERVIDOR

Segundo (MBUGUAH; MONY; NYABUTO, 2024), a arquitetura cliente-servidor descreve um modelo computacional onde um computador, chamado cliente, faz requisições a outro computador, chamado servidor, através de uma conexão com a internet. O servidor recebe a requisição, processa e responde ao cliente. Nesse modelo, podem existir um ou mais clientes fazendo requisições a um ou mais servidores que trabalham em conjunto.

3.3.1 Principais características do modelo cliente-servidor

- Tanto o computador cliente quanto o servidor precisam de protocolos para que a comunicação entre eles seja possível. Essa comunicação é feita diretamente com o protocolo da camada de transporte, que, por sua vez, utiliza as camadas inferiores para enviar e receber mensagens.
- Um único servidor pode atender a várias requisições simultaneamente. Porém, cada serviço exigirá uma *thread* separada para processar as solicitações.
- A arquitetura apresenta possibilidade de escalabilidade tanto horizontal quanto vertical. É possível adicionar mais servidores à arquitetura para lidar com aumento de carga de trabalho (escalabilidade horizontal), ao mesmo tempo em que se pode aumentar a capacidade de cada servidor, como memória RAM e CPU (escalabilidade vertical).

- Na arquitetura cliente-servidor, os computadores cliente e servidor podem operar em recursos de hardware e software heterogêneos, ou seja, diferentes entre si.

3.3.2 Tipos de implementação

Existem diferentes tipos de implementação da arquitetura cliente-servidor, baseada no número de servidores envolvidos. Abaixo estão os principais tipos de arquitetura:

- **Arquitetura de duas camadas (*two-tier*)**: É a mais simples e consiste apenas do cliente e do servidor. O cliente é responsável pela interface com o usuário e o servidor se responsabiliza pelo gerenciamento de dados, armazenando e manipulando as informações do sistema. Um exemplo desse tipo de arquitetura é um sistema de e-mail, onde o cliente é um programa que permite ao usuário enviar e receber mensagens, enquanto o servidor é um programa que armazena e gerencia essas mensagens.
- **Arquitetura de três camadas (*three-tier*)**: Esse modelo é mais complexo e adiciona uma nova camada à arquitetura, que passa a ter os seguintes componentes: o cliente, o servidor de aplicação e o servidor de dados. Nessa implementação, a interface com o usuário, o acesso à informação e o armazenamento de dados são hospedados de maneira separada. A camada adicional é responsável por hospedar o sistema de aplicação, enquanto outra camada hospeda o banco de dados (Figura 1).

Figura 1 – Arquitetura Cliente-Servidor de Três Camadas

Fonte: (QUEIROZ; CONCEIçãO; BARRETO, 2021)

3.4 ARQUITETURA SERVERLESS

A computação *serverless* é um paradigma para o desenvolvimento, implantação e operação de aplicações em nuvem. Ela representa uma evolução em direção a uma maior abstração na

tecnologia de virtualização. A computação *serverless* simplifica a experiência do usuário ao delegar ao provedor de serviço a responsabilidade pela infraestrutura e pelas tarefas operacionais, oferecendo alta agilidade e redução de custos no desenvolvimento de aplicações, sem exigir muita expertise operacional por parte dos desenvolvedores (TOOSI et al., 2025).

Em outras palavras, no modelo *serverless*, o desenvolvedor não precisa se preocupar com a infraestrutura que executa seu código; ela é responsabilidade do provedor de nuvem. O provedor faz a gestão desses servidores, incluindo provisionamento, escalonamento, manutenção e alocação de recursos. Apesar do termo *serverless* induzir a uma ideia de computação sem servidor, o servidor ainda existe, mas você simplesmente não precisa comprá-lo, gerenciá-lo ou mantê-lo.

Ainda segundo (TOOSI et al., 2025), a computação *serverless* é tipicamente baseada em eventos, com funções que são acionadas por diferentes tipos de gatilhos, como alteração em uma fonte de dados, requisições de usuários, mensagens adicionadas a uma fila ou inserções em bancos de dados. Com o modelo *serverless*, os usuários pagam apenas pelos recursos efetivamente consumidos, eliminando a necessidade de provisionar e pagar antecipadamente por infraestrutura.

3.4.1 Vantagens da arquitetura serverless

- **Custo**: O usuário paga apenas pelos recursos efetivamente consumidos durante a execução das funções. Em outros modelos baseados em nuvem, o usuário reserva previamente uma instância ou recursos computacionais e paga por eles mesmo que não os utilize.
- **Flexibilidade**: O modelo *serverless* escala de forma automática, sem que o desenvolvedor precise realizar configurações manuais para lidar com picos de acesso.
- **Velocidade**: Os desenvolvedores podem focar no desenvolvimento de codigo sem perder tempo se preocupando com configurações do servidor. Isso acelera o processo de desenvolvimento, favorecendo ciclos de entrega mais curtos e com maior foco na lógica de negócio.

3.4.2 Desafios da arquitetura serverless

- **Cold start**: Um dos principais desafios da computação *serverless* é o *cold start.* Ele ocorre quando uma função precisa ser executada após um período de inatividade e o provedor precisa inicializar um novo ambiente de execução. Esse processo pode causar um aumento no tempo de resposta da primeira requisição.
- **Imprevisibilidade de custo**: Apesar do modelo de cobrança baseado no uso ser uma vantagem, ele também pode se tornar uma fonte de imprevisibilidade financeira. Em aplicações com picos súbitos de uso, ou com eventos inesperados que disparam muitas funções simultaneamente, os custos podem aumentar de forma rápida e difícil de controlar.
- **Gerenciamento de estado**: A arquitetura *serverless* é, por natureza, *stateless*, ou seja, cada função é executada de forma independente e não mantém estado entre execuções. Isso gera um desafio no desenvolvimento de aplicações que exigem persistência de contexto, como sessões de usuário, transações longas ou fluxos com múltiplas etapas. Para contornar esse desafio, é necessário utilizar serviços externos de armazenamento, como bancos de dados ou sistemas de cache, o que pode aumentar a complexidade da aplicação.

3.5 COMPARATIVO ENTRE A ARQUITETURA CLIENTE-SERVIDOR E SERVERLESS

As arquiteturas cliente-servidor e *serverless* representam abordagens distintas para o desenvolvimento de sistemas. Para melhor visualização dessas diferenças, na Tabela 1 é possível ver uma comparação entre essas duas abordagens com base em características fundamentais.

3.6 TESTES DE DESEMPENHO

Teste de desempenho é um processo primordial no desenvolvimento de software, que avalia o quão bem um sistema se comporta sob diferentes condições. Ele faz mais do que avaliar se uma aplicação funciona; ele avalia o quão eficientemente ela se comporta às interações do usuário, lida com cargas variáveis e mantém seu desempenho ao longo do tempo. Existem vários tipos de teste de desempenho, alguns deles são:

- **Teste de carga (*Load testing*)**: Avalia como o sistema se comporta sob cargas de trabalho esperadas e de pico.
- **Teste de estresse (*Stress testing*)**: Ao forçar o sistema além de seus limites normais, esse teste verifica como ele lida com condições extremas e se consegue se recuperar de falhas.
- **Teste de resistência (*Soak testing*)**: Consiste em executar o sistema sob carga constante por um longo período, com objetivo de identificar vazamentos de memória, esgotamentos de recursos e degradação de desempenho.
- **Teste de pico (*Spike testing*)**: Simula aumentos súbitos e extremos no tráfego para observar como o sistema reage e se é capaz de lidar com picos abruptos de demanda.

Tabela 1 – Comparativo entre as arquiteturas Cliente-Servidor e Serverless. Baseado em (KELLTON, 2023).

Característica	**Cliente-Servidor**	**Serverless**
Gerenciamento de servidores	Requer configuração, provisionamento e manutenção manual dos servidores.	Totalmente gerenciado pelo provedor de nuvem; o desenvolvedor não se envolve com a infraestrutura.
Escalabilidade	Manual ou semi-automática, limitada à configuração prévia de instâncias.	Automática, granular e baseada em requisições, com escalabilidade sob demanda.
Modelo de cobrança	Baseado em uso contínuo dos recursos, com custos fixos mesmo quando ociosos.	Baseado em uso real; paga-se apenas pelo tempo de execução e recursos consumidos.
Tempo de resposta	Estável, já que os servidores estão sempre ativos.	Pode sofrer atrasos iniciais devido ao *cold start*.
Gerenciamento de estado	Pode manter estado na aplicação ou no servidor.	Necessita de soluções externas para persistência de estado.
Manutenção e atualização	Requer intervenção manual e pode causar indisponibilidade.	Facilita deploys contínuos com impacto mínimo.
Aderência ao provedor	Menor dependência de plataformas específicas.	Maior dependencia de integração com serviços do provedor.
Monitoramento e depuração	Mais simples, com ferramentas tradicionais e ambiente centralizado.	Exige ferramentas especializadas para observabilidade distribuída.

Neste trabalho, os testes de carga foram utilizados como método de avaliação de desempenho das arquiteturas analisadas. Essa escolha se deu por dois fatores principais: a capacidade

dos testes de carga em simular condições reais de uso, representando cenários com múltiplos usuários simultâneos, e a sua ampla aplicação como técnica para análise de escalabilidade e eficiência em ambientes distribuídos.

4 METODOLOGIA

A metodologia utilizada foi uma pesquisa aplicada, com abordagem quantitativa e experimental, voltada para a comparação de desempenho de duas arquiteturas distintas, no contexto de jogos *multiplayer* online.

4.1 TIPO DE PESQUISA

A pesquisa é do tipo aplicada, pois visa identificar se a arquitetura *serverless* é uma solução promissora no desenvolvimento de jogos online. A abordagem é quantitativa, uma vez que os dados analisados são mensuráveis e obtidos a partir de testes automatizados. Por fim, trata-se ainda de uma pesquisa experimental, uma vez que envolve a implementação de duas aplicações distintas, seguida pela execução de testes e a análise dos resultados obtidos.

4.2 ESTRATÉGIA METODOLÓGICA

O trabalho foi conduzido por meio da implementação de duas aplicações funcionalmente iguais, mas com arquiteturas diferentes. Essa igualdade funcional entre as versões foi importante para garantir que a comparação entre elas seja justa e focada na arquitetura. Após as arquiteturas serem implementadas, foram realizados testes de carga para simular cenários reais de uso, com múltiplos usuários interagindo de maneira concorrente.

Os testes foram realizados de forma controlada, utilizando uma ferramenta específica de testes, com monitoramento de tempo de resposta, seus percentis e vazão. Foram definidos cinco cenários de carga, com 20, 100, 500, 750 e 1000 usuários concorrentes, todos eles executando o fluxo completo da partida (criação de partida, entrada na partida, realização de jogadas e finalização). Os resultados obtidos foram utilizados como ferramenta comparativa na análise presente nos capítulos seguintes.

5 DESENVOLVIMENTO

Este capítulo apresenta o desenvolvimento das soluções utilizadas para analisar o desempenho da arquitetura *serverless* no contexto de jogos *multiplayer* online, bem como o desenvolvimento dos testes de carga que tiveram papel fundamental nesta análise, pois foram utilizados para estressar o ambiente, simulando picos de jogadores simultâneos.

5.1 DESENVOLVIMENTO DAS APLICAÇÕES

A etapa inicial do trabalho foi a definição da aplicação a ser implementada e utilizada na comparação das arquiteturas cliente-servidor e *serverless*. A escolhida foi o popular jogo da velha, por apresentar regras simples e fáceis de codificar, mas que ainda assim dispõe das principais características atribuídas aos jogos online, como controle de sessão, persistência de dados, comunicação bidirecional e atualização síncrona de estado. Essas características tornam o jogo um cenário experimental adequado para simular interações entre múltiplos usuários em tempo real, exigindo resposta rápida e eficiente por parte da infraestrutura.

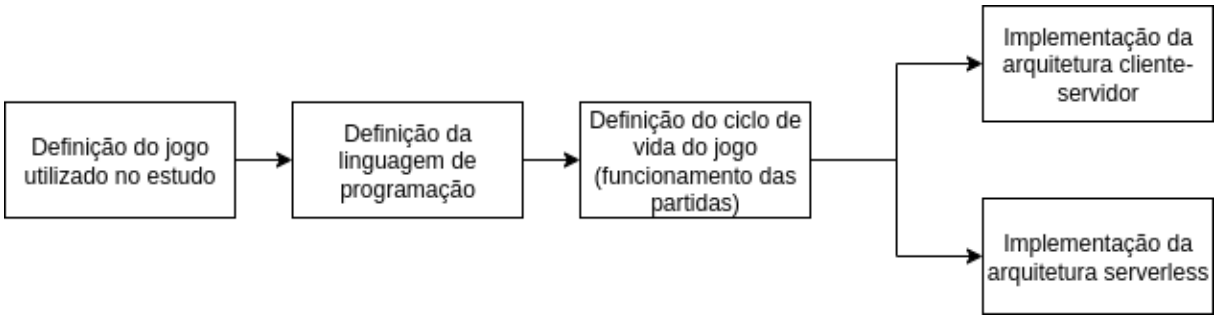
A definição do jogo foi uma escolha estratégica, pois permitiu que o foco do trabalho fosse a análise de desempenho e escalabilidade das arquiteturas, sem que a complexidade do jogo em si interferisse no processo de desenvolvimento.

Com o jogo definido, os passos seguintes foram definir a linguagem de programação, as ferramentas e tecnologias mais adequadas a cada abordagem e a montagem do ciclo de vida da aplicação, conforme a figura 2. Essas decisões resultaram na construção das duas versões da aplicação, cada uma representando uma arquitetura distinta – cliente-servidor e *serverless*. Mas, ainda garantindo paridade funcional entre ambas as versões, de modo que os testes de carga e desempenho pudessem ser conduzidos sob condições similares e, assim, fornecer dados comparativos confiáveis.

5.1.1 Linguagem de programação

Para ambas as arquiteturas, foi adotado o Python, devido à sua popularidade e facilidade de integração com diversos frameworks e bibliotecas. Além da popularidade, o Python também se destaca devido à sua facilidade de leitura e escrita, o que acelera o desenvolvimento e reduz

Figura 2 – Etapas de desenvolvimento da solução proposta



Fonte: Elaboração da autora (2025).

a complexidade do código-fonte.

Outro ponto relevante é o fato de que o Python oferece uma excelente integração com ferramentas voltadas para computação em nuvem, o que o torna adequado para a implementação de soluções baseadas em arquitetura serverless.

5.1.2 Ferramentas utilizadas

5.1.2.1 Arquitetura Tradicional Cliente-Servidor

A versão tradicional (Figura 3) utiliza um servidor escrito em Python 3.11 baseado em FastAPI, um framework que se destaca por seu alto desempenho, simplicidade de uso e suporte nativo a WebSockets — protocolo que foi usado para garantir a comunicação bidirecional entre os clientes e o servidor. Essa solução adota uma arquitetura do tipo N:1, em que múltiplos clientes estabelecem conexões simultâneas com uma única instância de servidor. Cabe destacar que, embora a replicação do servidor com balanceamento de carga seja uma abordagem tecnicamente possível para promover maior escalabilidade e resiliência, essa alternativa não foi considerada neste estudo, que se restringe à análise de uma topologia composta por apenas uma instância de backend.

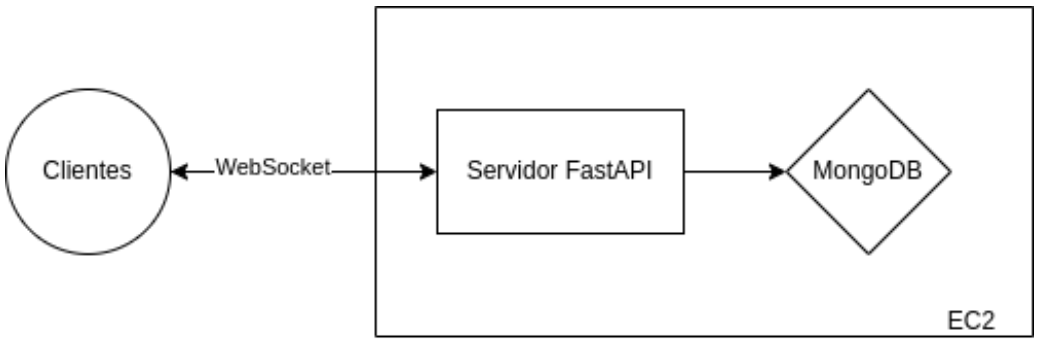
Para persistência de dados, tais como estado da partida e sessão, foi utilizado o banco de dados MongoDB, que é um banco não relacional e orientado a documentos, cuja flexibilidade na modelagem permite que cada partida possa ser armazenada como um documento JSON, facilitando a dinâmica de manipulação de dados da partida.

De modo a garantir uma comparação justa com a arquitetura *serverless* no momento de execução dos testes de carga, a aplicação cliente-servidor foi hospedada em uma instância

EC2 (*Amazon Elastic Compute Cloud*) da AWS, do tipo t2.micro — inclusa na camada gratuita (*free-tier*). A aplicação foi containerizada utilizando Docker, assegurando portabilidade e consistência no ambiente de execução.

A instância EC2 foi provisionada na região norte da Virgínia (us-east-1), reconhecida por ser a zona de disponibilidade mais econômica da AWS e também onde os serviços utilizados na arquitetura serverless foram implantados. Essa escolha teve como objetivo garantir que ambas as arquiteturas fossem executadas sob as mesmas condições geográficas e de infraestrutura. Ao manter todos os recursos na mesma região, foi evitada a introdução de variações no tempo de resposta por conta da distância física entre os componentes, assegurando assim um ambiente de testes mais controlado e justo.

Figura 3 – Arquitetura cliente-servidor do jogo da velha.



Fonte: Elaboração da autora (2025).

5.1.2.2 Arquitetura Serverless

A versão *serverless* (Figura 4) foi implementada utilizando exclusivamente os recursos gerenciados pela AWS. Nessa abordagem, toda a infraestrutura é abstraída do desenvolvedor, permitindo foco total na lógica da aplicação. A comunicação entre os clientes e o backend é feita através do Amazon API Gateway, que oferece suporte a WebSocket, viabilizando a troca contínua de mensagens em tempo real. Cada ação realizada pelo cliente — como realizar uma jogada ou iniciar uma partida — é tratada de forma isolada por funções AWS Lambda, que são executadas sob demanda.

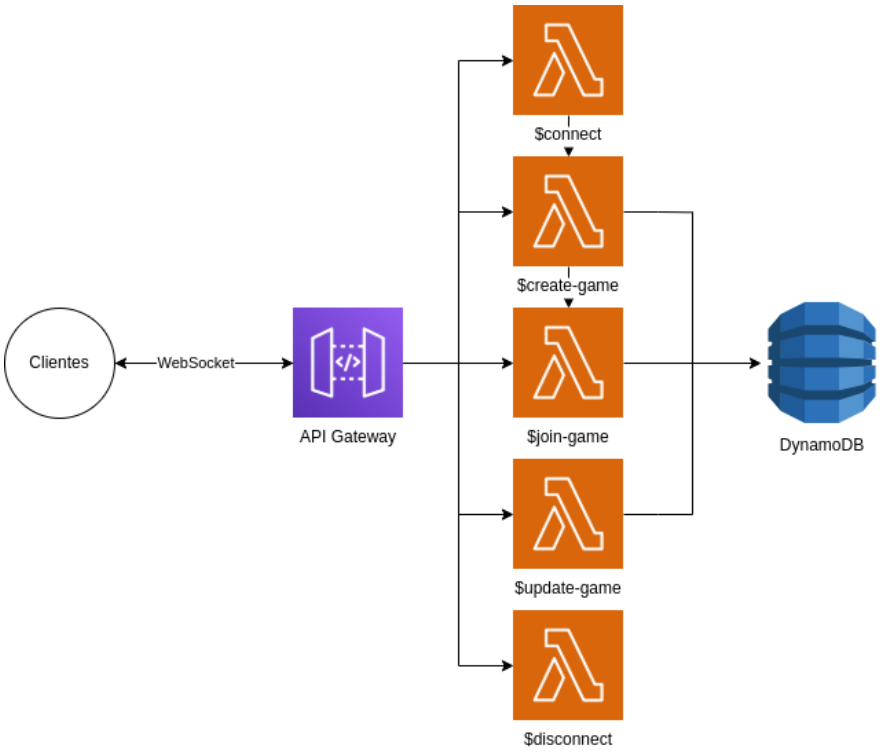
O estado da partida é armazenado e recuperado através do banco de dados Amazon DynamoDB, que também é um banco não relacional gerenciado e orientado a documentos.

A arquitetura foi composta pelos seguintes serviços:

- **Amazon API Gateway:** Responsável por gerenciar as entradas e saídas da aplicação, com suporte a WebSockets, permitindo uma comunicação contínua entre clientes.
- **AWS Lambda:** Responsável por executar a lógica de cada funcionalidade do jogo sob demanda, sem necessidade de provisionar ou manter servidores ativos. Cada função é executada apenas quando requisitada.
- **Amazon DynamoDB:** Banco de dados não relacional e totalmente gerenciado, com latência baixa e alta escalabilidade, utilizado para armazenar o estado das partidas.

A AWS foi escolhida como ferramenta de desenvolvimento neste caso devido à sua popularidade na indústria, sendo considerada uma das ferramentas de computação em nuvem mais utilizadas.

Figura 4 – Arquitetura serverless do jogo da velha.



Fonte: Elaboração da autora (2025).

5.1.3 Ciclo de vida do jogo

O ciclo de vida do jogo da velha *multiplayer* implementado neste trabalho é composto por três etapas principais: criação da partida, entrada de jogadores e realização de jogadas.

Cada uma dessas etapas é representada por um evento específico na aplicação, responsável por tratar as ações dos usuários e atualizar o estado do jogo, conforme descrito a seguir:

create-game: Este evento é responsável por criar uma partida gerando um identificador único (ID) que será retornado para o cliente/usuário. Esse ID pode ser compartilhado com outro jogador, permitindo que ambos participem da mesma partida. Além disso, ao criar a partida, os dados são armazenados no banco de dados (Figura 5), garantindo a persistência das informações. A estrutura da partida inclui os seguintes campos:

- **id**: Identificador único da partida — gerado utilizando UUID (*Universally Unique Identifier*);
- **board**: Vetor representando o tabuleiro do jogo no estado atual, sendo 9 posições preenchidas com X, O ou string vazia;
- **player-x e player-o**: Identificadores para os usuários (correspondentes ao ID da conexão WebSocket);
- **turn**: Indica qual jogador possui a vez de jogar;
- **winner**: Informa o vencedor da partida, caso já tenha sido definido.

Figura 5 – Documento JSON de uma partida armazenada no DynamoDB.

Fonte: Elaboração da autora (2025).

join-game: Este evento permite que um segundo jogador envie o ID de uma partida existente para se conectar a ela. Caso o ID não exista, a partida já tenha dois jogadores ou já tenha sido finalizada, ele retorna uma mensagem de alerta ao cliente avisando que não é possível entrar na partida.

update-game: Este evento permite que o cliente realize uma jogada em alguma posição do tabuleiro, desde que seja a vez dele de jogar e a posição seja válida. Caso não se encaixe em alguma das condições, ele retorna uma mensagem alertando que não é a vez daquele jogador de fazer o movimento ou que a posição é inválida.

5.2 DESENVOLVIMENTO DOS TESTES DE CARGA

Para avaliar o desempenho das duas arquiteturas já implementadas — cliente-servidor e *serverless* — foi necessário estabelecer uma metodologia que pudesse simular um comportamento de pico de usuários interagindo com o sistema. Para isso, foram realizados testes de carga.

A realização de testes de carga é essencial em aplicações que envolvem múltiplos usuários simultaneamente. Por meio deles, é possível simular situações de uso intensivo e observar como o sistema se comporta quando exposto à alta demanda, identificando possíveis gargalos, falhas ou limitações de desempenho. Essa prática permite não apenas garantir a robustez e a estabilidade da aplicação, mas também comparar, com base em dados concretos, a eficiência de diferentes soluções. O foco dos testes foi analisar o comportamento de cada arquitetura sob diferentes níveis de carga, levando em consideração principalmente o tempo de resposta, fator extremamente importante na experiência do usuário.

Os testes foram estruturados de forma a representar situações reais de uso, em que vários jogadores realizam partidas em paralelo. Para isso, foi adotado um fluxo completo de interação com o jogo, desde a criação da partida até a sua finalização.

5.2.1 Ferramenta utilizada

A ferramenta escolhida para a realização dos testes de carga foi a Locust, uma solução de código aberto, amplamente utilizada por grandes empresas para testes de performance. Sua escolha se deu por diversos fatores:

- Suporte nativo à linguagem Python, que também foi utilizado no desenvolvimento das aplicações;
- Permite a criação de scripts altamente personalizáveis, simulando cenários completos de partidas entre usuários;

- Permite monitoramento em tempo real das métricas;

5.2.2 Cenários de teste

Para garantir uma comparação justa, foram definidos cenários de teste iguais para ambas as arquiteturas, com base no ciclo de vida completo de uma partida de jogo da velha. Cada usuário simulado executou uma sequência padronizada de eventos:

- Criação ou entrada em uma partida (*create-game* ou *join-game*);
- Realização de jogadas alternadas (*update-game*);
- Finalização da partida após a vitória ou empate.

Esse fluxo garante que todas as funcionalidades da aplicação são acionadas durante o teste. As simulações foram configuradas para simular cargas progressivas, com os seguintes volumes de usuários simultâneos: 20, 100, 500, 750 e 1000. Cada teste foi executado por 1 minuto, permitindo observar gargalos e medir o tempo médio de resposta dos sistemas.

5.2.3 Ambiente de execução

Todos os testes foram realizados sob condições padronizadas e controladas:

- **Cliente-servidor:** A aplicação foi executada em uma instância EC2 modelo *t2.micro*, com backend em FastAPI utilizando uma instância local do MongoDB para armazenamento dos dados.
- **Serverless:** A arquitetura *serverless* foi implementada com os serviços AWS API Gateway, AWS Lambda e AWS DynamoDB, aproveitando a escalabilidade automática oferecida pela nuvem.
- **Locust:** Os scripts de teste foram executados em uma segunda instância EC2 t2.micro, dedicada exclusivamente à execução dos testes. Isso garantiu isolamento dos testes, evitando interferências do ambiente da aplicação testada.

5.2.4 Métricas coletadas

Durante os testes, foram observadas as seguintes métricas fundamentais para a análise de desempenho das arquiteturas:

- **Tempo médio de resposta por requisição**: Representa o tempo médio que o sistema leva para responder a uma requisição após seu recebimento. Essa métrica é essencial para avaliar a eficiência da aplicação sob diferentes níveis de carga e para mensurar o tempo de espera do usuário final.
- **Requisições por segundo (RPS)**: Representa o número de requisições processadas por segundo.
- **Tempo de resposta percentil (P95 e P99)**: Medição do tempo máximo em que 95% e 99% das requisições foram respondidas. Esses percentis são utilizados para entender como a aplicação se comporta em cenários de pico, identificando gargalos ou inconsistências no tempo de resposta e ajudam a mensurar a confiabilidade do sistema sob carga intensa.

Essas métricas foram analisadas com o propósito de entender como cada arquitetura se comporta ao aumento de demanda, considerando o impacto direto na experiência do usuário.

6 RESULTADOS

Nesta seção, são apresentados os resultados obtidos após a execução dos testes de carga nas duas arquiteturas propostas.

6.1 CENÁRIOS DE TESTE

6.1.1 Cenário 1 - 20 Usuários

Neste cenário foi possível entender como os sistemas se comportam com baixa carga. Como visto nas Tabelas 2 e 3, a arquitetura *serverless* apresentou um tempo de resposta inicial mais alto em comparação à cliente-servidor. Este comportamento na versão *serverless* pode ser explicado por dois fatores principais:

- **Cold Start das Funções Lambda:** No modelo *serverless*, as funções Lambda são executadas sob demanda. Quando uma função é invocada pela primeira vez, ou após um período de inatividade, a AWS precisa alocar recursos computacionais, carregar o código da função e inicializar as dependências. Esse processo é conhecido como *cold start*, e pode adicionar até centenas de milissegundos ao tempo de resposta na primeira execução. Após esse primeiro uso, as funções permanecem quentes, ou seja, com tudo carregado e pronto para rodar novamente, tornando as próximas chamadas bem mais rápidas.
- **Overhead do API Gateway:** A arquitetura *serverless* também depende do Amazon API Gateway para lidar com os eventos e roteá-los para as funções Lambda corretas. Esse estágio introduz um *overhead* adicional, que embora pequeno, também contribui para o aumento do tempo de resposta.

Tabela 2 – Resultados do teste de carga na arquitetura cliente-servidor com 20 usuários simultâneos (tempos de resposta médios em milissegundos).

Evento	Média	Mínimo	Máximo	Mediana	RPS
create-game	1	1	4	2	3.89
join-game	2	2	6	2	3.89
update-game	1	0	34	2	27.62

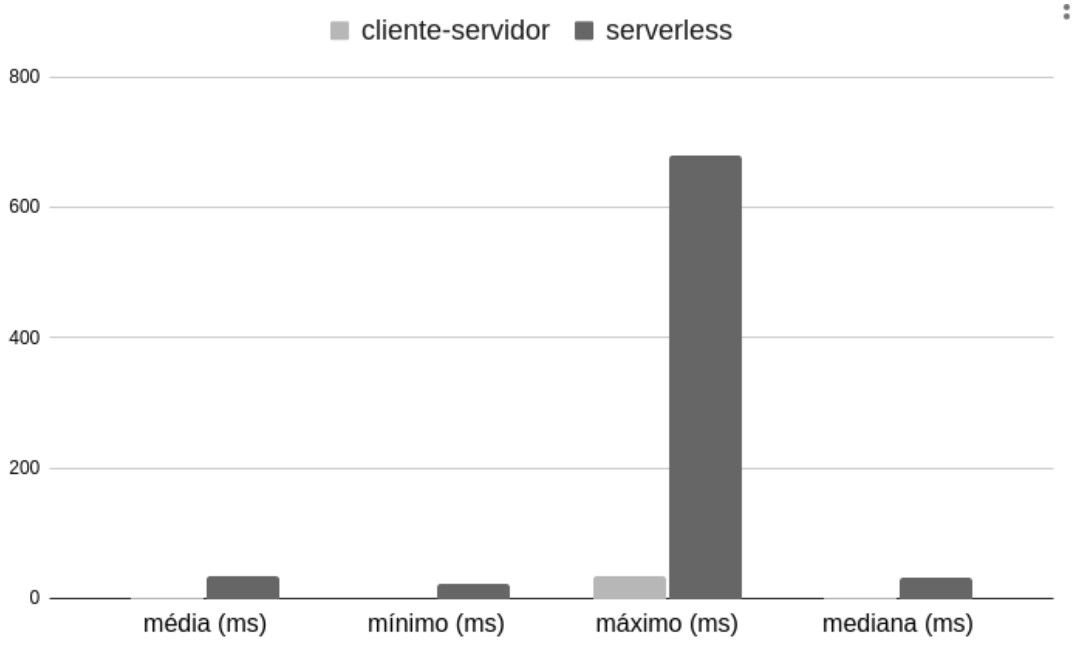
Fonte: Testes da carga com Locust (2025)

Tabela 3 – Resultados do teste de carga na arquitetura serverless com 20 usuários (tempos de resposta médios em milissegundos).

Evento	Média	Mínimo	Máximo	Mediana	RPS
create-game	38	24	679	28	1.64
join-game	37	26	102	34	1.64
update-game	34	22	150	22	11.15

Fonte: Testes da carga com Locust (2025)

Figura 6 – Comparativo de tempo de resposta serverless vs. cliente-servidor com 20 usuários.



Fonte: Elaboração da autora (2025).

6.1.2 Cenário 2 - 100 Usuários

Após adicionar mais 80 usuários ao teste, ainda foi obtida uma situação similar à anterior, com a cliente-servidor ligeiramente mais eficiente, porém, começando a apresentar um aumento no tempo máximo de resposta, como é possível ver nas tabelas 4 e 5.

Tabela 4 – Resultados do teste de carga na arquitetura cliente-servidor com 100 usuários (tempos de resposta médios em milissegundos).

Evento	Média	Mínimo	Máximo	Mediana	RPS
create-game	2	1	212	2	15.14
join-game	3	1	24	3	15.14
update-game	2	0	36	2	109.14

Fonte: Testes da carga com Locust (2025)

Tabela 5 – Resultados do teste de carga na arquitetura serverless com 100 usuários (tempos de resposta médios em milissegundos).

Evento	Média	Mínimo	Máximo	Mediana	RPS
create-game	30	22	122	28	5.34
join-game	35	26	152	33	5.34
update-game	34	26	154	32	35.57

Fonte: Testes da carga com Locust (2025)

6.1.3 Cenário 3 - 500 Usuários

Neste cenário, a diferença entre as arquiteturas começou a se tornar mais expressiva. O tempo de resposta da versão cliente-servidor (Tabela 6), que era inicialmente muito baixo, passa a crescer ao aumento de carga e, em alguns momentos, se torna mais lento que a versão *serverless* (Tabela 7).

Tabela 6 – Resultados do teste de carga na arquitetura cliente-servidor com 500 usuários (tempos de resposta médios em milissegundos).

Evento	Média	Mínimo	Máximo	Mediana	RPS
create-game	74	1	978	56	49.61
join-game	100	1	1012	77	49.61
update-game	78	0	1134	41	364.36

Fonte: Testes da carga com Locust (2025)

Tabela 7 – Resultados do teste de carga na arquitetura serverless com 500 usuários (tempos de resposta médios em milissegundos).

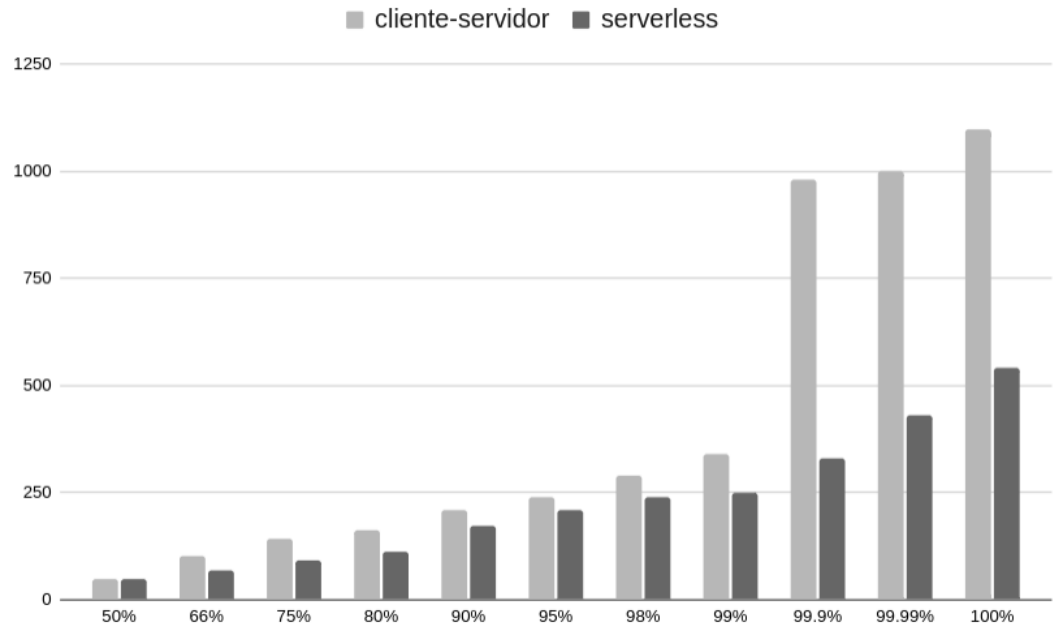
Evento	Média	Mínimo	Máximo	Mediana	RPS
create-game	80	22	538	45	21.40
join-game	65	26	433	48	21.28
update-game	74	25	433	48	139;77

Fonte: Testes da carga com Locust (2025)

Outro resultado relevante observado neste estágio foram os percentis dos tempos de resposta das duas arquiteturas (Figura 7). A análise do gráfico mostrou uma diferença acentuada na inclinação das curvas, indicando que a arquitetura cliente-servidor possui maior dispersão nos tempos de resposta, principalmente nos percentis mais elevados. Isso indica que, à medida que a carga aumenta, há usuários experimentando situações de lentidão na aplicação.

Por outro lado, a arquitetura *serverless* manteve os tempos de resposta mais concentrados ao longo dos percentis, demonstrando um comportamento mais previsível e uniforme.

Figura 7 – Comparativo de percentis de tempo de resposta serverless vs. cliente-servidor com 500 usuários.



Fonte: Elaboração da autora (2025).

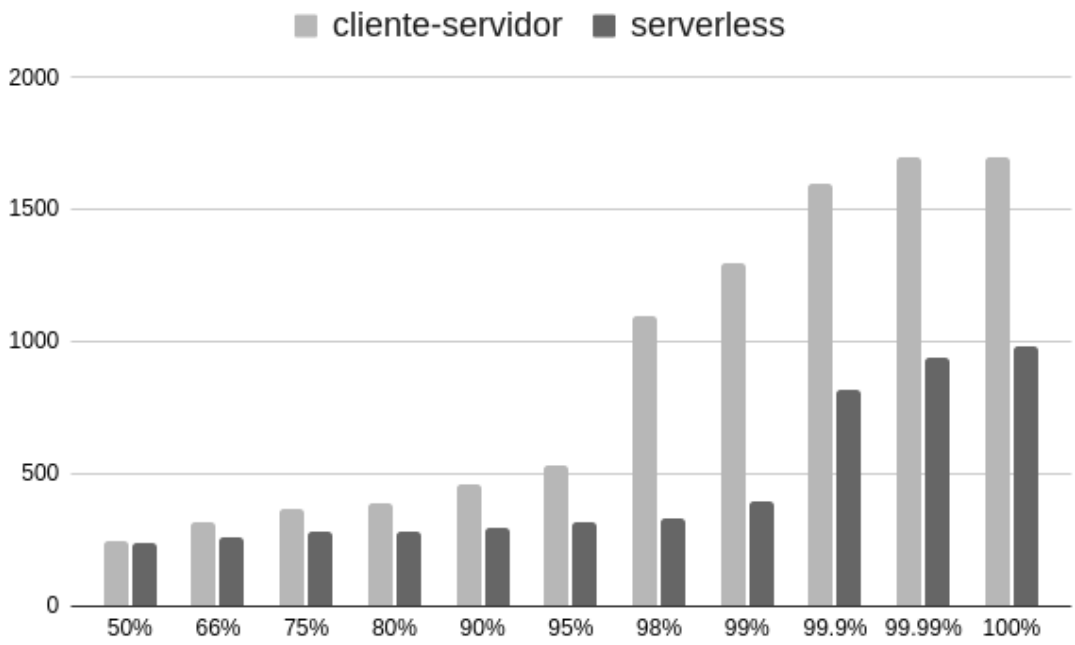
6.1.4 Cenário 4 - 750 Usuários

Neste cenário, foi confirmada a capacidade de escalabilidade automática da arquitetura *serverless*, que continuou apresentando mais estabilidade, enquanto a cliente-servidor passou a apresentar uma curva de percentis de tempo de resposta cada vez mais acentuada e com crescimento rápido, conforme ilustrado na Figura 8.

6.1.5 Cenário 5 - 1000 Usuários

Este teste representou o ponto máximo de estresse do sistema. Conforme os resultados encontrados (Tabelas 8 e 9), a arquitetura cliente-servidor demonstrou sinais de saturação: o tempo de resposta médio aumentou significativamente, e os percentis rapidamente ultrapassaram a marca de 1000ms (Figura 9), revelando uma grande variação na experiência dos usuários. Apesar de ter apresentado uma taxa de requisições por segundo (RPS) mais elevada, o sistema é muito imprevisível.

Figura 8 – Comparativo de percentis de tempo de resposta serverless vs. cliente-servidor com 750 usuários.



Fonte: Elaboração da autora (2025).

Tabela 8 – Resultados do teste de carga na arquitetura cliente-servidor com 1000 usuários (tempos de resposta médios em milissegundos).

Evento	Média	Mínimo	Máximo	Mediana	RPS
create-game	371	1	1697	370	63.54
join-game	485	1	1840	480	62.96
update-game	342	0	1844	310	596.22

Fonte: Testes da carga com Locust (2025)

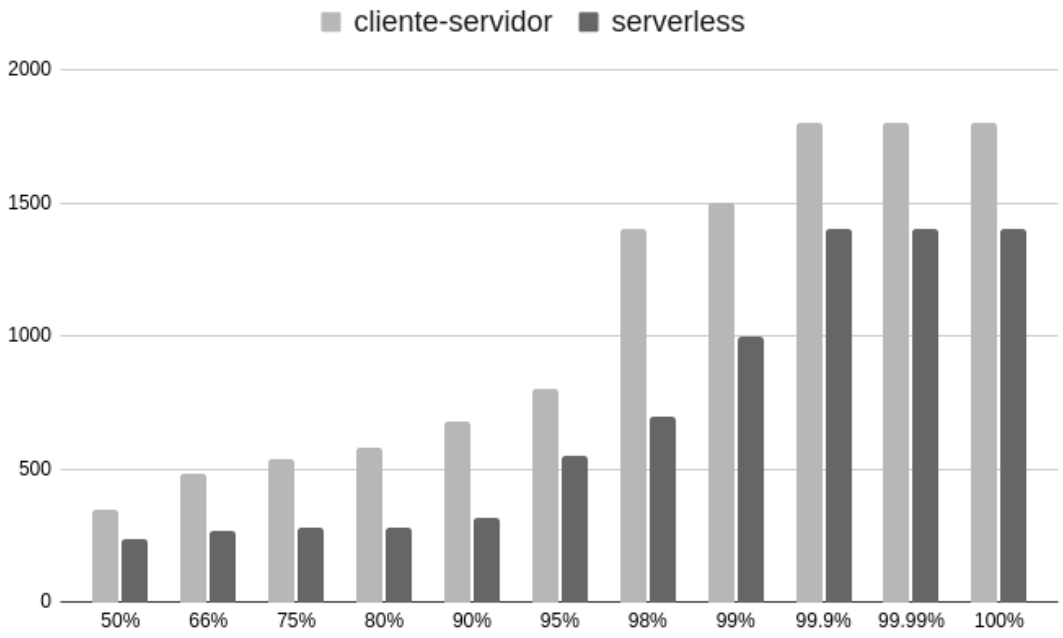
Tabela 9 – Resultados do teste de carga na arquitetura serverless com 1000 usuários (tempos de resposta médios em milissegundos).

Evento	Média	Mínimo	Máximo	Mediana	RPS
create-game	256	24	1446	270	25.87
join-game	164	26	1438	160	25.87
update-game	265	26	1432	250	177.30

Fonte: Testes da carga com Locust (2025)

Em contrapartida, a arquitetura *serverless* manteve um comportamento estável e previsível ao longo de todo o teste. Embora sua RPS média tenha sido inferior à da cliente-servidor, o sistema é mais coerente. O tempo de resposta permaneceu dentro de uma faixa controlada, mesmo nos percentis mais altos. Essa consistência está diretamente relacionada à capacidade de escalabilidade automática da AWS, que distribui a carga entre diversas instâncias das

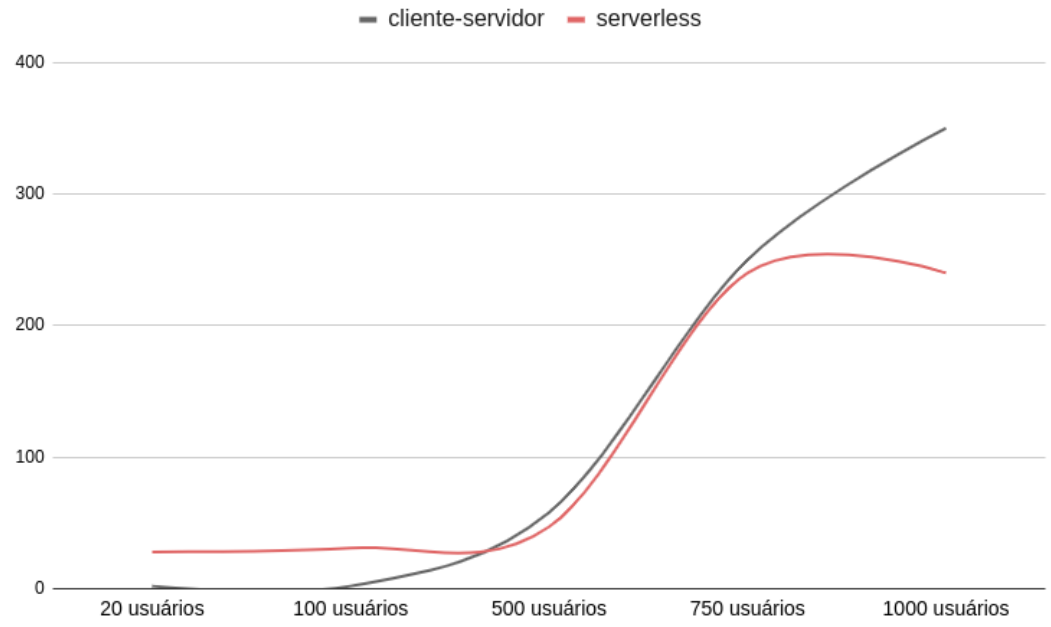
Figura 9 – Comparativo de percentis de tempo de resposta serverless vs. cliente-servidor com 1000 usuários.



Fonte: Elaboração da autora (2025).

funções Lambda, evitando sobrecarga em pontos específicos da aplicação e garantindo uma resposta equilibrada.

Figura 10 – Comparativo da mediana em milissegundos dos tempos de resposta das arquiteturas serverless e cliente-servidor para cada um dos cenários abordados.



Fonte: Elaboração da autora (2025).

6.2 DISCUSSÃO DOS RESULTADOS

A partir da execução dos testes de carga e análise dos resultados, foi possível identificar diferenças significativas no comportamento das duas arquiteturas. Embora a arquitetura cliente-servidor tenha começado com um desempenho mais satisfatório, ela perdeu estabilidade quando foi exposta a um grande volume de usuários (Figura 10). Em contrapartida, a arquitetura *serverless* manteve um desempenho mais consistente e confiável ao longo dos cenários testados.

Os tempos médios de resposta obtidos foram base para essa observação. No cenário de 1000 usuários simultâneos, a versão *serverless* foi aproximadamente 66% mais rápida que a cliente-servidor no evento **join-game**. Nos eventos **create-game** e **update-game**, observou-se uma melhoria no desempenho de cerca de 31% e 22,5%, respectivamente.

7 CONCLUSÃO

Este trabalho teve como objetivo fazer uma análise de desempenho de soluções *serverless* em comparação com soluções tradicionais cliente-servidor no contexto de jogos *multiplayer* online, tomando como estudo de caso a implementação de um jogo da velha online. A comparação das arquiteturas permitiu identificar diferenças significativas nas arquiteturas quando expostas à carga.

A arquitetura cliente-servidor apresentou um desempenho estável em cenários com baixa concorrência, mas sua incapacidade em escalar horizontalmente de forma automática resultou em degradação de performance à medida que o número de usuários aumentou. Esse fator compromete diretamente a experiência do jogador, uma vez que a dinâmica dos jogos online demanda respostas rápidas para manter a fluidez e a competitividade da partida. Em contrapartida, a arquitetura *serverless* se mostrou mais adequada ao cenário proposto. Baseada em serviços gerenciados da AWS, como Lambda, API Gateway e DynamoDB, essa abordagem permitiu que a aplicação escalasse de maneira automática, adaptando-se dinamicamente à carga imposta pelos testes. No cenário com 1000 usuários simultâneos, os tempos de resposta médios foram significativamente inferiores aos da arquitetura cliente-servidor, com destaque para uma melhora de aproximadamente 66% no evento **join-game**. Essa superioridade evidencia a capacidade do modelo *serverless* de entregar desempenho consistente mesmo em ambientes de alta concorrência, o que é fundamental para jogos *multiplayer* em tempo real.

Diante dos resultados, conclui-se que a adoção da arquitetura *serverless* representa uma alternativa promissora para o desenvolvimento de jogos online que visam alta escalabilidade, resiliência e desempenho sob demanda sem a necessidade de configurar servidores. Entretanto, sua adoção impõe novos desafios, como a necessidade de adaptação da lógica da aplicação a um modelo orientado a eventos, o gerenciamento dos tempos de inicialização das funções (*cold starts*), e a complexidade no controle dos custos, que variam de acordo com o volume de uso e podem impactar no orçamento de aplicações com alta frequência de chamadas.

Para trabalhos futuros, seria interessante ampliar o escopo de análise para outros tipos de aplicação ou até mesmo para jogos com maior complexidade lógica e interação em tempo real mais intensa. Outra possibilidade que pode ser levada em consideração é a exploração de arquiteturas híbridas, que combinem o modelo cliente-servidor tradicional com a flexibilidade e a elasticidade automática da computação *serverless*.

REFERÊNCIAS

ADZIC, G.; CHATLEY, R. Serverless computing: economic and architectural impact. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2017. (ESEC/FSE 2017), p. 884–889. ISBN 9781450351058. Disponível em: <https://doi.org/10.1145/3106237.3117767>.

COLE S. H., . H. J. M. Clinical and personality correlates of mmo gaming. *Social Science Computer Review*, p. 31, 2013. Disponível em: <https://doi.org/10.1177/0894439312475280>.

DUCHENEAUT, N.; MOORE, R. J. The social side of gaming: a study of interaction patterns in a massively multiplayer online game. In: *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*. New York, NY, USA: Association for Computing Machinery, 2004. (CSCW '04), p. 360–369. ISBN 1581138105. Disponível em: <https://doi.org/10.1145/1031607.1031667>.

IANCU, T. *A Case Study in Scaling Minecraft Using Serverless Computing*. Amsterdam, Países Baixos, 2021. Disponível em: <https://jdonkervliet.com/assets/pdf/students/202112-hp-report-tiberiu-iancu.pdf>.

KELLTON. *A Beginner's Guide to Serverless Computing*. 2023. Accessed: 2025-07-30. Disponível em: <https://www.kellton.com/kellton-tech-blog/beginners-guide-to-serverless-computing>.

L BIGNILL J, S. V. M. P. A. A. S. H. M. J. D. R. T. W. A. R.; L, K.-D. Massively multiplayer online games and well-being: A systematic literature review. *Frontiers in Psychology*, p. 12, 2021. Disponível em: <https://doi.org/10.3389/fpsyg.2021.698799>.

MBUGUAH, S.; MONY, V.; NYABUTO, G. Architectural review of client-server models. *International Journal of Scientific Research and Engineering Trends*, v. 10, p. 139–143, 01 2024.

MELL, P.; GRANCE, T. *The NIST Definition of Cloud Computing*. [S.l.], 2011. Disponível em: <https://doi.org/10.6028/NIST.SP.800-145>.

OSMAN, B. H. Bachelor's thesis, *Serverless Game Leaderboard System: Unreal & Web Integration*. 2025. Acesso em: 9 jul. 2025. Disponível em: <https://www.theseus.fi/bitstream/handle/10024/894055/Osman_Bilal.pdf?sequence=2>.

PAUL, P. K.; GHOSE, M. K. Cloud computing: Possibilities, challenges and opportunities with special reference to its emerging need in the academic and working area of information science. *Procedia Engineering*, v. 38, p. 2222–2227, 2012. ISSN 1877-7058. INTERNATIONAL CONFERENCE ON MODELLING OPTIMIZATION AND COMPUTING. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877705812021807>.

QUEIROZ, C.; CONCEIçãO, L.; BARRETO, M. Estudo dos aspectos para a disponibilizaÇÃo de dados locais na nuvem: Estudo de caso netdoctor. *RECIMA21 - Revista Científica Multidisciplinar - ISSN 2675-6218*, v. 2, p. e26474, 07 2021.

TOOSI, A. N.; JAVADI, B.; IOSUP, A.; SMIRNI, E.; DUSTDAR, S. Serverless computing for next-generation application development. *Future Generation Computer Systems*, v. 164, p. 107573, 2025. ISSN 0167-739X. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0167739X24005375>.

WU, F.; YUEN, H. Y.; CHAN, H. C.; LEUNG, V. C.; CAI, W. Infinity battle: A glance at how blockchain techniques serve in a serverless gaming system. In: *Proceedings of the 28th ACM International Conference on Multimedia*. New York, NY, USA: Association for Computing Machinery, 2020. (MM '20), p. 4559–4561. ISBN 9781450379885. Disponível em: <https://doi.org/10.1145/3394171.3414458>.

ZHANG Q., C. L. . B. R. Cloud computing: state-of-the-art and research challenges. *Procedia Computer Science*, p. 1, 2010. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1877705812021807>.

APÊNDICE A — TRECHOS DE CÓDIGO UTILIZADOS

Código Fonte 1 – Código main da versão cliente-servidor

```
from fastapi import FastAPI, WebSocket, WebSocketDisconnect
from game_store import create_game, get_game, join_game, update_game
from game_logic import check_winner
import json

app = FastAPI()
clients = {}

@app.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    await websocket.accept()
    player_id = str(id(websocket))
    clients[player_id] = websocket

    try:
        while True:
            msg = await websocket.receive_text()
            data = json.loads(msg)
            action = data.get("action")

            if action == "create":
                game_id = await create_game(player_id)
                await websocket.send_json({
                    "status": "created",
                    "game_id": game_id
                })

            elif action == "join":
                game_id = data["game_id"]
                game = await get_game(game_id)
                if not game or game["player_o"]:
                    await websocket.send_json({"error": "Game not available"})
                    continue

                await join_game(game_id, player_id)
                await websocket.send_json({
                    "status": "joined",
                    "game_id": game_id
                })

                player_x = game["player_x"]
                if player_x in clients:
                    await clients[player_x].send_json({
```

```
                        "status": "start",
                        "message": "Opponent joined.",
                        "game_id": game_id
                    })

            elif action == "move":
                game_id = data["game_id"]
                index = data["index"]
                game = await get_game(game_id)

                if not game or game["winner"]:
                    await websocket.send_json({"error": "Game not found or
                        finished"})
                    continue

                board = game["board"]
                turn = game["turn"]

                if board[index] != "":
                    await websocket.send_json({"error": "Invalid move"})
                    continue

                expected_player = game["player_x"] if turn == "X" else game["
                    player_o"]
                if player_id != expected_player:
                    await websocket.send_json({"error": "Not your turn"})
                    continue

                board[index] = turn
                winner = check_winner(board)
                next_turn = "O" if turn == "X" else "X"

                await update_game(game_id, {
                    "board": board,
                    "turn": next_turn if not winner else turn,
                    "winner": winner
                })

                for pid in [game["player_x"], game["player_o"]]:
                    if pid in clients:
                        await clients[pid].send_json({
                            "status": "update",
                            "board": board,
                            "turn": next_turn,
                            "winner": winner
                        })
```

```
89      except WebSocketDisconnect:
            del clients[player_id]
```

Código Fonte 2 – Código de manipulação do banco de dados da versão cliente-servidor

```
1   from database import db
    from bson import ObjectId
3
    async def create_game(player_id):
5       game = {
            "player_x": player_id,
7           "player_o": None,
            "board": [""] * 9,
9           "turn": "X",
            "winner": None
11      }
        result = await db.games.insert_one(game)
13      return str(result.inserted_id)

15  async def get_game(game_id):
        return await db.games.find_one({"_id": ObjectId(game_id)})
17
    async def join_game(game_id, player_id):
19      await db.games.update_one(
            {"_id": ObjectId(game_id)},
21          {"$set": {"player_o": player_id}}
        )
23
    async def update_game(game_id, data):
25      await db.games.update_one(
            {"_id": ObjectId(game_id)},
27          {"$set": data}
        )
```

Código Fonte 3 – Código de checagem de ganhador da versão cliente-servidor

```
1   def check_winner(board):
        wins = [
3           [0, 1, 2], [3, 4, 5], [6, 7, 8],
            [0, 3, 6], [1, 4, 7], [2, 5, 8],
5           [0, 4, 8], [2, 4, 6]
        ]
7       for a, b, c in wins:
            if board[a] and board[a] == board[b] == board[c]:
9               return board[a]
        if "" not in board:
11          return "draw"
        return None
```

Código Fonte 4 – Configuração de banco de dados da versão cliente-servidor

```
from motor.motor_asyncio import AsyncIOMotorClient

MONGO_URI = "mongodb://XXX.XX.XX.X:XXXXX"

client = AsyncIOMotorClient(MONGO_URI, serverSelectionTimeoutMS=50000)

db = client.tic_tac_toe
```

Código Fonte 5 – Dockerfile da versão cliente-servidor

```
FROM ubuntu:latest

WORKDIR /app
COPY requirements.txt /app/
RUN apt update && apt install -y python3 python3-pip python3-venv
RUN python3 -m venv /opt/venv
RUN /opt/venv/bin/pip install --no-cache-dir -r requirements.txt
COPY . /app/
EXPOSE 8000

CMD ["/opt/venv/bin/uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000",
     "--workers", "4"]
```

Código Fonte 6 – Função create-game da versão serverless

```
import json
import boto3, uuid
from botocore.exceptions import BotoCoreError, ClientError

dynamodb = boto3.resource('dynamodb')
TABLE_NAME = "game"

def create_game_session(connection_id):
    game_id = str(uuid.uuid4())

    item = {
            'id': game_id,
            'player_x': connection_id,
            'player_o': None,
            'board': [""] * 9,
            'turn': "X",
            'winner': None
        }

    try:
        table = dynamodb.Table(TABLE_NAME)
```

```
        response = table.put_item(Item=item)
        return {
            "statusCode": 200,
            "body": json.dumps({"status": "created", "game_id": game_id})
        }
    except (BotoCoreError, ClientError) as error:
        print(f"Erro ao criar item: {error}")
        return None

def lambda_handler(event, context):
    connection_id = event['requestContext']['connectionId']

    return create_game_session(connection_id)
```

Código Fonte 7 – Função join-game da versão serverless

```
import json
import boto3, uuid
from botocore.exceptions import BotoCoreError, ClientError

dynamodb = boto3.resource('dynamodb')
TABLE_NAME = "game"

def join_game_session(game_id, connection_id):
    try:
        table = dynamodb.Table(TABLE_NAME)

        response = table.get_item(Key={'id': game_id})
        game = response.get('Item')

        if not game or game['player_o']:
            return {
                'statusCode': 400,
                'body': json.dumps({'error': 'Game not available'})
            }

        table.update_item(
            Key={'id': game_id},
            UpdateExpression="SET player_o = :o",
            ExpressionAttributeValues={':o': connection_id}
        )

        return {
            'statusCode': 200,
            'body': json.dumps({'status': 'joined'})
        }
    except (BotoCoreError, ClientError) as error:
        print(f"Error: {error}")
```

```
        return None

def lambda_handler(event, context):
    body = json.loads(event["body"])
    data = body["data"]
    connection_id = event['requestContext']['connectionId']

    return join_game_session(data["game_id"], connection_id)
```

Código Fonte 8 – Função update-game da versão serverless

```
import json
import boto3, uuid
from botocore.exceptions import BotoCoreError, ClientError

dynamodb = boto3.resource('dynamodb')

def check_winner(board):
    wins = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],
        [0, 3, 6], [1, 4, 7], [2, 5, 8],
        [0, 4, 8], [2, 4, 6]
    ]
    for a, b, c in wins:
        if board[a] and board[a] == board[b] == board[c]:
            return board[a]
    if "" not in board:
        return "draw"
    return None

def update_status(game_id, index, player_id):
    table = dynamodb.Table("game")
    response = table.get_item(Key={"id": game_id})
    game = response.get("Item")

    if not game or game['winner']:
        return {'statusCode': 400, 'body': json.dumps({'error': 'Game not found
            or finished'})}

    board = game['board']
    turn = game['turn']

    if board[index] != "":
        return {'statusCode': 400, 'body': json.dumps({'error': 'Invalid move'})}

    expected_player = game['player_x'] if turn == "X" else game['player_o']
    if player_id != expected_player:
```

```
        return {'statusCode': 403, 'body': json.dumps({'error': 'Not your turn'})
            }

    board[index] = turn
    winner = check_winner(board)
    next_turn = "O" if turn == "X" else "X"

    table.update_item(
        Key={'id': game_id},
        UpdateExpression="SET board = :b, turn = :t, winner = :w",
        ExpressionAttributeValues={
            ':b': board,
            ':t': next_turn if not winner else turn,
            ':w': winner
        }
    )
    return {
        "statusCode": 200,
        "body": json.dumps({
            'status': 'move registered',
            'board': board,
            'turn': next_turn,
            'winner': winner
        })
    }

def lambda_handler(event, context):
    body = json.loads(event["body"])
    data = body["data"]
    connection_id = event['requestContext']['connectionId']
    return update_status(data["game_id"], data["index"], connection_id)
```

Código Fonte 9 – Script de teste da versão cliente-servidor

```
import json
import random
import time
from locust import User, task, between
import websocket

class TicTacToeUser(User):
    wait_time = between(1, 3)

    @task
    def play_game(self):
        self.simulate_match()


```

```
    def simulate_match(self):
        ws_a = websocket.create_connection("ws://XX.XX.XXX.XX:XXXX/ws")
        ws_b = websocket.create_connection("ws://XX.XX.XXX.XX:XXXX/ws")

        ws_a.settimeout(10)
        ws_b.settimeout(10)

        try:
            start_time = time.time()

            # Jogador A cria partida
            ws_a.send(json.dumps({"action": "create"}))
            response = ws_a.recv()

            total_time = int((time.time() - start_time) * 1000)

            self.environment.events.request.fire(
                request_type="WS", name="create", response_time=total_time,
                response_length=len(response), exception=None
            )

            print(f"Player A: {response}")
            game_id = json.loads(response)["game_id"]

            # Jogador B entra na partida
            start_time = time.time()

            ws_b.send(json.dumps({"action": "join", "game_id": game_id}))
            response = ws_b.recv()

            total_time = int((time.time() - start_time) * 1000)

            self.environment.events.request.fire(
                request_type="WS", name="join", response_time=total_time,
                response_length=len(response), exception=None
            )

            print(f"Player B: {response}")

            winner = None
            moves = [0, 1, 2, 3, 4, 5, 6, 7, 8]
            for move in moves:
                print(f"Player move => Position: {move}")

                start_time = time.time()

                ws_a.send(json.dumps({"action": "move", "game_id": game_id, "
```

```
                    index": move}))
62                response = ws_a.recv()

64                total_time = int((time.time() - start_time) * 1000)

66                self.environment.events.request.fire(
                      request_type="WS", name="move", response_time=total_time,
68                    response_length=len(response), exception=None
                  )
70
                  print(f"Player move => Response: {response}")
72
                  update = json.loads(response)
74                winner = update.get("winner")

76                if winner:
                      print(f"Winner: {winner}")
78                    break

80                ws_a, ws_b = ws_b, ws_a

82        except websocket.WebSocketTimeoutException as e:
              print(f"Timeout: {e}")
84            self.environment.events.request.fire(
                  request_type="WS", name="timeout", response_time=0,
86                response_length=0, exception=e
              )
88        except Exception as e:
              print(f"Error: {e}")
90            self.environment.events.request.fire(
                  request_type="WS", name="error", response_time=0,
92                response_length=0, exception=e
              )
94            raise
          finally:
96            ws_a.close()
              ws_b.close()
```

Código Fonte 10 – Script de teste da versão serverless

```
  import json
2 import random
  import time, ssl
4 from locust import User, task, between, events
  import websocket
6 from statistics import median, stdev

8 class TicTacToeUser(User):
```

```
    wait_time = between(1, 3)

    @task
    def play_game(self):
        self.simulate_match()


    def simulate_match(self):
        ws_a = websocket.create_connection("wss://XXXXXXXXX.execute-api.us-east-1.amazonaws.com/dev/")
        ws_b = websocket.create_connection("wss://XXXXXXXXX.execute-api.us-east-1.amazonaws.com/dev/")

        ws_a.settimeout(100)
        ws_b.settimeout(100)

        try:
            start_time = time.time()

            # Jogador A cria partida
            ws_a.send(json.dumps({"action": "create_session"}))

            response = ws_a.recv()

            total_time = int((time.time() - start_time) * 1000)

            self.environment.events.request.fire(
                request_type="WS", name="create", response_time=total_time,
                response_length=len(response), exception=None
            )

            print(f"Player A: {response}")
            game_id = json.loads(response)["game_id"]

            # Jogador B entra na partida
            start_time = time.time()

            ws_b.send(json.dumps({"action": "join_session", "data": {"game_id": game_id}}))
            response = ws_b.recv()

            total_time = int((time.time() - start_time) * 1000)

            self.environment.events.request.fire(
                request_type="WS", name="join", response_time=total_time,
                response_length=len(response), exception=None
            )
```

```
            print(f"Player B: {response}")

            winner = None
            moves = [0, 1, 2, 3, 4, 5, 6, 7, 8]

            for move in moves:
                print(f"Player move => Position: {move}")

                start_time = time.time()

                ws_a.send(json.dumps({"action": "update_status", "data": {"
                    game_id": game_id, "index": move}}))
                response = ws_a.recv()

                total_time = int((time.time() - start_time) * 1000)

                self.environment.events.request.fire(
                    request_type="WS", name="move", response_time=total_time,
                    response_length=len(response), exception=None
                )

                print(f"Player move => Response: {response}")

                update = json.loads(response)
                winner = update.get("winner")

                if winner:
                    print(f"Winner: {winner}")
                    break


                ws_a, ws_b = ws_b, ws_a

        except websocket.WebSocketTimeoutException as e:
            print(f"Timeout: {e}")
            self.environment.events.request.fire(
                request_type="WS", name="timeout", response_time=0,
                response_length=0, exception=e
            )
        except Exception as e:
            print(f"Error: {e}")
            self.environment.events.request.fire(
                request_type="WS", name="error", response_time=0,
                response_length=0, exception=e
            )
            raise
```

```
        finally:
            ws_a.close()
            ws_b.close()
```