Implementação de Logs de Auditoria em um Sistema de Reservas Online Baseado em Microsserviços

Abraão Levi de Oliveira Figueredo[1]*; Carlos Henrique Rodrigues Sarro[2]

[1] Universidade Anhembi Morumbi. Bacharel em Sistemas de Informação. Avenida Raimundo Pereira de Magalhães, 2199 – Jardim Iris; 05145-000 São Paulo, São Paulo, Brasil
[2] Universidade Metodista de Piracicaba. Mestre em Ciências da Computação. Rua Território do Acre, nº 1222, casa – Vila Prudente; 13420-585 Piracicaba, São Paulo, Brasil
*autor correspondente: abraao.of@gmail.com

Implementação de Logs de Auditoria em um Sistema de Reservas Online Baseado em Microsserviços

Resumo

A crescente digitalização do setor de turismo tornou necessária a adoção de mecanismos robustos para a integridade e a confiabilidade das informações em plataformas de reservas online. A ausência de um sistema de logs de auditoria comprometia a rastreabilidade de eventos críticos, dificultando a resolução de problemas e a segurança. Nesse contexto, o objetivo deste trabalho foi desenvolver e implantar um sistema de logs de auditoria para monitorar alterações no banco de dados de um sistema de reservas, visando solucionar dificuldades operacionais e de análise de falhas. Aplicou-se a metodologia de pesquisa-ação para guiar o desenvolvimento de uma solução baseada em microsserviços com comunicação assíncrona via filas de mensagens e banco de dados NoSQL. Avaliou-se a solução por meio de testes de performance e entrevistas qualitativas com as equipes operacionais. Os resultados demonstraram a viabilidade técnica da solução, com impacto médio de 7 milissegundos por requisição. A ferramenta validou seu valor operacional ao permitir o diagnóstico ágil de falhas em produção, e a análise qualitativa confirmou a percepção de melhoria na capacidade de investigação das equipes. Concluiu-se que a implementação foi bem-sucedida, melhorando a rastreabilidade dos dados e gerando aprendizados operacionais e estratégicos sobre a adoção da arquitetura de microsserviços.
Palavras-chave: pesquisa-ação; rastreabilidade de dados; comunicação assíncrona; arquitetura de software; segurança da informação.

Introdução

A crescente digitalização do setor de turismo tem impulsionado uma expansão notável, evidenciada pelo crescimento de 71,5% no número de viagens realizadas no Brasil entre 2021 e 2023, totalizando 21,1 milhões de deslocamentos. No cenário internacional, grandes agências de viagens online, como o Expedia Group, movimentaram aproximadamente 104 bilhões de dólares em um único ano, destacando em seus relatórios a importância da proteção de dados e da segurança cibernética. Esse cenário exige que as plataformas de reservas online adotem medidas para garantir a integridade e a confiabilidade das informações armazenadas (EXPE, 2023; IBGE, 2024).

Para atender às demandas de escalabilidade e flexibilidade desses sistemas complexos, a arquitetura de microsserviços tem se destacado como uma abordagem eficaz, pois permite a implementação de novas funcionalidades de forma independente e facilita a manutenção. Aliada a essa arquitetura, a comunicação assíncrona por meio de filas de mensagens surge como um importante componente para desacoplar processos e minimizar o impacto na latência dos sistemas, garantindo que operações secundárias não interrompam os serviços principais, mesmo durante picos de demanda (Kanizawa e Pinto, 2022; Maharjan et al., 2023).

Contudo, a ausência de um sistema eficiente de logs de auditoria compromete a rastreabilidade de eventos críticos, dificultando a resolução de problemas e a segurança das

informações. A implementação de tal sistema apresenta um desafio técnico central: a necessidade de capturar e registrar dados detalhados pode introduzir sobrecarga e aumento no tempo de resposta do sistema, impactando negativamente a performance da aplicação.

Diante disso, emerge o seguinte problema de pesquisa: "Como desenvolver e implantar um sistema de logs de auditoria em uma plataforma de reservas online de forma desacoplada e assíncrona, a fim de garantir a rastreabilidade de dados sem comprometer a performance do sistema principal?". A relevância de solucionar este problema é fundamental, pois os registros de auditoria, ou trilhas de auditoria, são essenciais não apenas para a segurança e conformidade regulatória, especialmente em setores altamente regulamentados como o setor financeiro, mas também como ferramenta para a detecção de acessos não autorizados, fraudes e a investigação de falhas de software e operacionais (Adkins et al., 2020; Brasil, 2021; Das e Chu, 2023; Gil, 2000).

Diante do cenário exposto, este trabalho tem como objetivo desenvolver e implantar um sistema de logs de auditoria capaz de registrar e monitorar alterações em bancos de dados de reservas e acomodações. A solução busca resolver problemas recorrentes na operação, como a dificuldade na análise de falhas e no atendimento ao cliente, esperando que a rastreabilidade proporcionada pelos logs promova melhorias contínuas nos processos internos e aumente a eficiência na identificação e correção de problemas.

Metodologia

Este estudo foi conduzido na plataforma de reservas online de uma empresa de médio porte do setor de turismo e gestão de imóveis para aluguel de temporada, sediada na Flórida, EUA, com atuação nos mercados norte-americano e latino-americano.

Adotou-se como metodologia a pesquisa-ação, caracterizada por sua abordagem cíclica de planejamento, ação, observação e reflexão. A pesquisa possui um delineamento de métodos qualitativos e quantitativo, integrando dados de desempenho de sistema e percepções dos usuários para avaliar a implantação de uma nova funcionalidade de logs de auditoria (Tripp, 2005).

Houve participação direta das equipes de desenvolvimento de software, composta por 12 profissionais, e 3 gestores das equipes operacionais (atendimento, suporte e sucesso do proprietário). A seleção dos participantes se deu por amostragem não-probabilística intencional, incluindo a totalidade da equipe técnica responsável pela implementação e os gestores com conhecimento estratégico sobre os problemas que a solução visava resolver.

As fases de pesquisa, planejamento e implementação iniciaram em maio de 2025. A coleta de dados utilizou múltiplos instrumentos: a coleta quantitativa, realizada em julho de 2025 em ambiente de homologação, empregou a ferramenta Grafana K6 para executar testes

de carga e mensurar o tempo de resposta do sistema. A coleta qualitativa, por sua vez, ocorreu em agosto de 2025 e envolveu entrevistas estruturadas com os gestores para avaliar a percepção de valor da ferramenta em produção, além de reuniões de planejamento via Google Meet com todos os envolvidos.

A análise dos dados quantitativos baseou-se na comparação das métricas de desempenho antes e depois da implementação. Os dados qualitativos, por sua vez, foram avaliados através das entrevistas.

Ciclo 1: Diagnóstico e planejamento da ação

Nesta etapa inicial, focou-se em compreender o contexto a fim de diagnosticar o problema e planejar uma solução técnica viável.

Caracterização do ambiente de pesquisa

A plataforma apresentava crescimento contínuo no número de reservas, aquisição de novos clientes e proprietários de imóveis. Este cenário motivou um planejamento para a expansão das equipes de desenvolvimento e produto, tornando a escalabilidade e a manutenibilidade do software uma prioridade estratégica. Esta plataforma era composta por três módulos principais de um sistema monolítico: o portal do hóspede, a área administrativa e o portal do proprietário.

O portal do hóspede era direcionado a usuários com interesse em buscar acomodações e outros serviços, realizar reservas, efetuar pagamentos e gerenciar suas estadias.

A área administrativa era utilizada pelo time de operações e atendimento, responsável pelo suporte ao cliente, gestão de reservas e resolução de problemas operacionais.

O portal do proprietário oferecia aos donos dos imóveis sob gestão da empresa acesso a relatórios de performance de suas propriedades, além da possibilidade de gerenciar reservas de uso pessoal e bloquear períodos para aluguel ou uso próprio.

A aplicação executava em um ambiente em nuvem utilizando os serviços da Amazon Web Services [AWS] onde o tráfego de conteúdo estático era direcionado para o serviço de Content Delivery Network (CDN) Amazon CloudFront conectado ao serviço de armazenamento de arquivos Amazon S3.

Para o tráfego dinâmico da aplicação as requisições eram direcionadas para o Elastic Load Balancing (ELB), que distribuía a carga de trabalho entre múltiplas instâncias da aplicação. A aplicação em si era orquestrada utilizando-se o AWS Elastic Beanstalk, que gerenciava um grupo de instâncias de máquinas virtuais Amazon Elastic Compute Cloud (EC2) onde o código dos sistemas era executado. Essas instâncias interagiam com dois bancos de dados, o Amazon ElastiCache Redis, usado para cache de dados e o Amazon

Relational Database Service (RDS) PostgresSQL, que armazenava os dados transacionais e persistentes do sistema de reservas. Conforme apresentado na Figura 1.

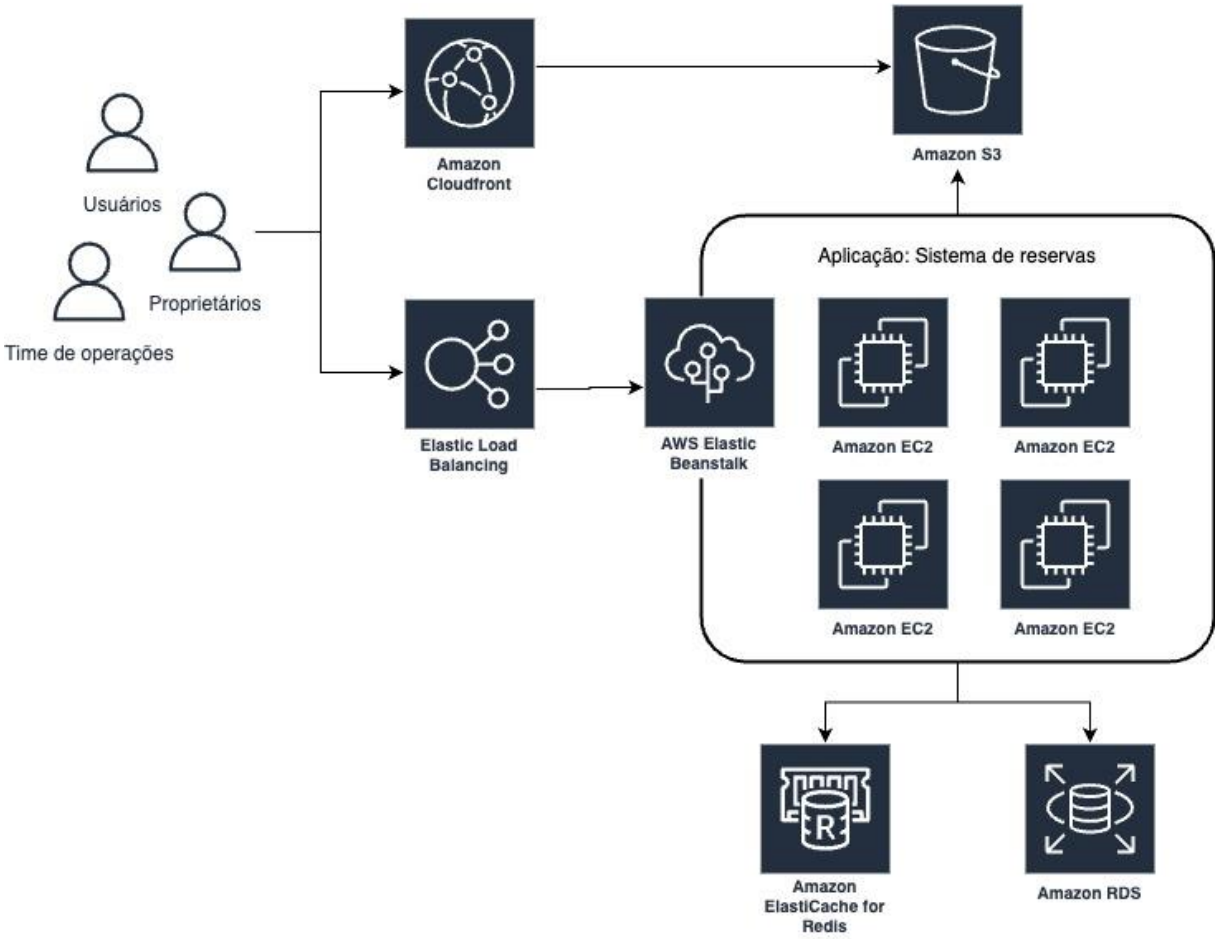

Figura 1. Diagrama da arquitetura da aplicação antes da implantação do sistema de logs de auditoria
Fonte: Dados originais da pesquisa

O sistema foi desenvolvido utilizando a linguagem de programação Typescript, com o framework NestJS na versão 10, executado no ambiente de tempo de execução Node.js, na versão 22, juntamente com o Prisma Object-Relational Mapper (ORM) na versão 5 que realizava as interações com o banco de dados RDS PostgresSQL na versão 18.

Diagnóstico do problema

A partir das interações com as equipes envolvidas, identificou-se uma dificuldade relacionada à rastreabilidade de modificações em dados do sistema, especialmente aqueles vinculados a reservas de clientes e às configurações internas das acomodações registradas no sistema. Essa limitação tornava-se evidente em situações em que clientes entravam em contato com o suporte relatando inconsistências em suas reservas ou problemas ocorridos durante a estadia. Nessas circunstâncias, surgia a necessidade de compreender a origem das

alterações realizadas, identificar o agente responsável, verificar a possível participação de integrações externas, como "webhooks", bem como recuperar o estado anterior dos dados alterados. A ausência de mecanismos que permitissem responder a essas demandas dificultava a resolução assertiva de incidentes e comprometia a qualidade do atendimento prestado.

A partir dessas observações permitiu-se identificar a necessidade da implantação de um sistema de logs de auditoria. Tal sistema deveria centralizar, em um único repositório, os registros de alterações nas entidades de reservas e acomodações, viabilizando o rastreamento da origem e autoria das modificações. Além disso, identificou-se que esses registros poderiam também auxiliar a equipe de desenvolvimento na identificação e reprodução de erros, oferecendo dados relevantes para correções e melhorias contínuas nos sistemas existentes.

Levantamento de requisitos

A fase seguinte, referente ao planejamento técnico, envolveu o levantamento dos requisitos funcionais e não funcionais da solução proposta. Definiu-se como eventos relevantes a serem registrados a criação, edição e exclusão de registros nas entidades mencionadas. Estabeleceu-se, ainda, que os metadados associados a cada evento deveriam incluir a data e hora da modificação, os dados originais, as alterações realizadas, o identificador do agente responsável (usuário interno, sistema ou integração), bem como informações contextuais da requisição, como endereço IP e cabeçalhos HTTP.

Outro requisito identificado foi a necessidade de que a solução desenvolvida evitasse o aprisionamento tecnológico, também conhecido como "vendor lock-in", visto que a equipe de desenvolvimento avaliava a possibilidade de migração da infraestrutura para outro provedor de serviços em nuvem.

Definição do plano de ação e arquitetura da solução

A partir da definição dos dados que deveriam ser coletados, iniciou-se uma pesquisa sobre as possíveis abordagens técnicas para viabilizar a implementação do sistema de auditoria. Nesse processo, a equipe de desenvolvimento identificou três alternativas principais: a utilização de "triggers" (gatilhos) no banco de dados, a aplicação da técnica de Change Data Capture (CDC) por meio das ferramentas Debezium e Apache Kafka, e a implementação da auditoria diretamente na camada da aplicação.

A opção baseada em gatilhos foi descartada após avaliação técnica, devido a experiências anteriores da equipe com essa abordagem em outras funcionalidades do sistema. Em projetos passados, o uso de gatilhos gerou dificuldades de manutenção, uma vez que parte da lógica de negócios ficava descentralizada, estando distribuída entre a

aplicação e o próprio banco de dados. Essa separação exigia que os desenvolvedores compreendessem e gerenciassem duas camadas lógicas distintas. Como resultado, houve um esforço passado para remover gatilhos preexistentes e concentrar toda a lógica de negócios exclusivamente na aplicação.

A abordagem de CDC com Debezium e Kafka foi descartada devido à maior complexidade arquitetural, necessidade extensiva de manutenção de componentes distribuídos e limitações inerentes na captura de metadados específicos da aplicação, uma vez que CDC opera no nível do log de transações do banco de dados (Kodakandla, 2023; Nõu, 2025).

Com base na pesquisa e nas discussões realizadas, concluiu-se que a implementação do sistema de auditoria no nível da aplicação ofereceria maior flexibilidade, além de permitir o registro de um contexto de negócios mais completo. Essa abordagem possibilitaria capturar informações como a intenção do usuário e a operação lógica que originou a modificação, incluindo eventos que não se limitam a comandos Data Manipulation Language (DML), mas abrangem também ações específicas do domínio da aplicação.

Após a decisão da alternativa a ser seguida, identificou-se uma possível fonte de aumento do tempo de resposta das requisições. Para essa abordagem, seria necessária uma consulta adicional ao banco de dados nas operações de criação, edição e exclusão para capturar o estado anteiro e posterior da operação, executar um algoritmo para extrair as alterações entre o estado anterior e posterior, e registrar a alteração. Essa execução adicional poderia resultar em um aumento no tempo de resposta das requisições ao sistema.

Com base em discussões internas com os membros mais experientes da equipe de desenvolvimento, estabeleceu-se como requisito não funcional que o tempo adicional causado por esse processo não ultrapassasse 150 milissegundos por requisição. Esse valor foi considerado um limite razoável para não comprometer a experiência dos usuários finais nem a performance geral da aplicação.

Durante o processo de definição da arquitetura da solução, considerou-se a possibilidade de implementar o sistema de auditoria como um novo módulo dentro da aplicação monolítica existente ou como um serviço separado. Por se tratar de um projeto secundário para as operações da empresa, optou-se por desenvolver a solução como um microsserviço independente, com o objetivo de experimentar esse modelo arquitetural e avaliar sua viabilidade técnica e operacional no contexto da organização.

Essa decisão visava proporcionar aprendizado prático sobre implantação, comunicação entre serviços e manutenção de aplicações distribuídas, contribuindo para decisões futuras sobre a adoção de microsserviços em outras frentes do sistema, em conformidade com a recomendação de não reescrever um sistema monolítico inteiro de uma

só vez, mas sim extrair gradualmente funcionalidades do monolito para microsserviços de forma segura e incremental (Newman, 2021; Richardson, 2018).

Após a decisão de adotar a abordagem baseada em microsserviços, foram realizadas pesquisas nas documentações oficiais de sistemas de mensageria com o objetivo de definir a tecnologia mais adequada ao projeto. Essa investigação visou evitar o vínculo tecnológico restritivo com um único provedor de serviços, de modo a preservar a portabilidade da solução e facilitar uma eventual migração da infraestrutura para outros ambientes de computação em nuvem.

Elaborou-se uma tabela comparativa entre serviços de mensageria gerenciados disponíveis na AWS e Google Cloud Platform [GCP], nos quais a instalação, configuração e manutenção é de responsabilidade dos provedores e sistemas de filas não gerenciado, onde essas responsabilidades são da equipe técnica da empresa. Levou-se em consideração a interface de programação de aplicações (API) principal, suporte à protocolos abertos, nível de aprisionamento tecnológico e facilidade de migração do sistema como um todo em caso de mudança de infraestrutura e provedor de nuvem conforme é apresentado na Tabela 1.

Tabela 1. Comparação entre serviços de mensageria

Sistema	API Principal	Protocolo aberto	Aprisionamento	Facilidade de migração
AWS SQS	AWS SDK	Não	Alto	Baixa
AWS SNS	AWS SDK	Não	Alto	Baixa
AWS Kinsesis Data Streams	AWS SDK	Não	Alto	Baixa
Amazon MQ (RabbitMQ)	AMQP, MQTT, STOMP	Sim	Médio	Alta
Amazon MQ (ActiveMQ)	AMQP, MQTT, OpenWire	Sim	Médio	Alta
CGP Pub/Sub	gRPC/REST (Proprietário)	Não	Alto	Baixa
RabbitMQ (Autogerenciado)	AMQP, MQTT, STOMP	Sim	Baixo	Alta
Apache Kafka (Autogerenciado)	Protocolo Kafka	Sim	Baixo	Alta

Fonte: Dados originais da pesquisa

Para a classificação do nível de aprisionamento (alto, médio ou baixo), considerou-se o grau de dependência de APIs proprietárias dos provedores de nuvem e a complexidade envolvida na substituição do serviço. Soluções com APIs exclusivas e ausência de suporte a protocolos abertos foram classificadas como de alto aprisionamento, uma vez que dificultam a portabilidade entre diferentes infraestruturas. Segundo Opara-Martins et al. (2017), o uso de interfaces e formatos de dados específicos do fornecedor é uma causa primária de "lock-in",

pois as aplicações tornam-se intrinsecamente ligadas ao ecossistema desse provedor, dificultando a migração ou integração com outros sistemas.

Serviços gerenciados com suporte a padrões abertos foram classificados como de médio aprisionamento, enquanto os autogerenciados, baseados em protocolos amplamente adotados, receberam classificação de baixo aprisionamento por permitirem maior liberdade na migração entre plataformas. Opara-Martins et al. (2017) apontam que APIs proprietárias criam barreiras significativas à portabilidade de aplicações, reforçando a necessidade de adoção de padrões abertos para reduzir riscos de dependência.

Quanto à facilidade de migração, considerou-se como alta a possibilidade de migração com baixo esforço de modificações nos códigos já existentes, especialmente quando a aplicação pode ser transferida para outra instância do mesmo sistema, por exemplo, de RabbitMQ gerenciado para RabbitMQ autogerenciado. Classificações baixas indicam que a migração exigiria reimplementação de parte do sistema por depender de APIs específicas do provedor e não adotar padrões abertos.

Após discussões com a equipe de tecnologia, optou-se pelo Amazon MQ RabbitMQ para a comunicação assíncrona entre os sistemas, por ser um serviço gerenciado e devido à sua capacidade de portabilidade de código por ser compatível com protocolos abertos.

O serviço de mensageria permitiu desacoplar o processamento das operações principais do registro de logs, minimizando o impacto na latência dos sistemas. A implementação das filas seguiu os padrões de integração de sistemas propostos por Hohpe e Woolf (2003).

A escolha por um banco de dados do tipo NoSQL justificou-se pela sua maior adequação ao armazenamento de dados semiestruturados e às cargas de trabalho com elevada intensidade de operações de escrita. A heterogeneidade dos registros de auditoria, nos quais diferentes entidades monitoradas podem gerar estruturas distintas de dados, exigiu flexibilidade de esquema que é característica uma característica dos bancos orientados a documentos. Este modelo se alinha ao conceito de dados orientados a agregados, onde cada evento de auditoria é tratado como uma unidade completa e autocontida (Sadalage e Fowler, 2012).

Elaborou-se uma pesquisa com base nas documentações oficiais dos bancos de dados NoSQL. Foram analisadas o Amazon DynamoDB, Google Cloud Firestore, MongoDB e MongoDB Atlas Database. Considerou aspectos como o modelo adotado, gerenciamento da solução, potencial de dependência tecnológica e facilidade de migração entre plataformas, conforme apresentado na Tabela 2. Optou-se pela adoção do MongoDB Atlas Database por se tratar de uma solução gerenciada e por não estar vinculada exclusivamente a um único provedor de serviços em nuvem.

Tabela 2. Comparação entre banco de dados NoSQL

Sistema	Modelo	Gerenciamento	Aprisionamento	Facilidade de migração
DynamoDB	Chave-valor	Gerenciado	Alto	Baixa
Firestore	Documento	Gerenciado	Alto	Baixa
MongoDB	Documento	Autogerenciado	Baixo	Alta
Atlas Database	Documento	Gerenciado	Baixo	Alta

Fonte: Dados originais da pesquisa

A empresa fazia parte de programas de aceleração para startups da AWS e do GCP, dispondo de créditos para a utilização dos serviços de infraestrutura em nuvem. Por essa razão, a análise de custos dos serviços adotados não fez parte do escopo deste trabalho, que se concentrou na viabilidade técnica, no impacto operacional e nos aprendizados arquiteturais da solução implementada.

Ciclo 2: implementação da ação

Com as ferramentas para o projeto definidas, o ciclo seguinte focou no desenvolvimento e implementação técnica da solução.

No sistema monolítico de reservas, implementou-se uma extensão do Prisma ORM na camada de acesso a dados, com o objetivo de interceptar as operações de criação, edição e remoção das entidades selecionadas. Durante essas operações, os dados originais foram capturados e submetidos a um algoritmo responsável por identificar as diferenças entre os estados anterior e posterior do registro.

As informações obtidas, incluindo data e hora da modificação, conteúdo alterado e identificação do usuário responsável pela ação e do contexto da requisição, foram organizadas em um formato estruturado e enviadas para a fila de mensagens, possibilitando o posterior processamento pelo microsserviço de logs.

Em seguida, implementou-se um microsserviço utilizando a linguagem Typescript e a biblioteca Express, na versão 5.1.0. Ele teve como responsabilidade recepcionar os eventos de log recebidos por meio da fila de mensagens, registrá-los no banco de dados NoSQL e, quando solicitado por meio de requisições HTTP, realizar buscas dos registros de auditoria. Também foi desenvolvido um componente consumidor que processava continuamente os eventos provenientes da fila. Após implantação do sistema de logs de auditoria a nova arquitetura foi documentada, como apresentado na Figura 2.

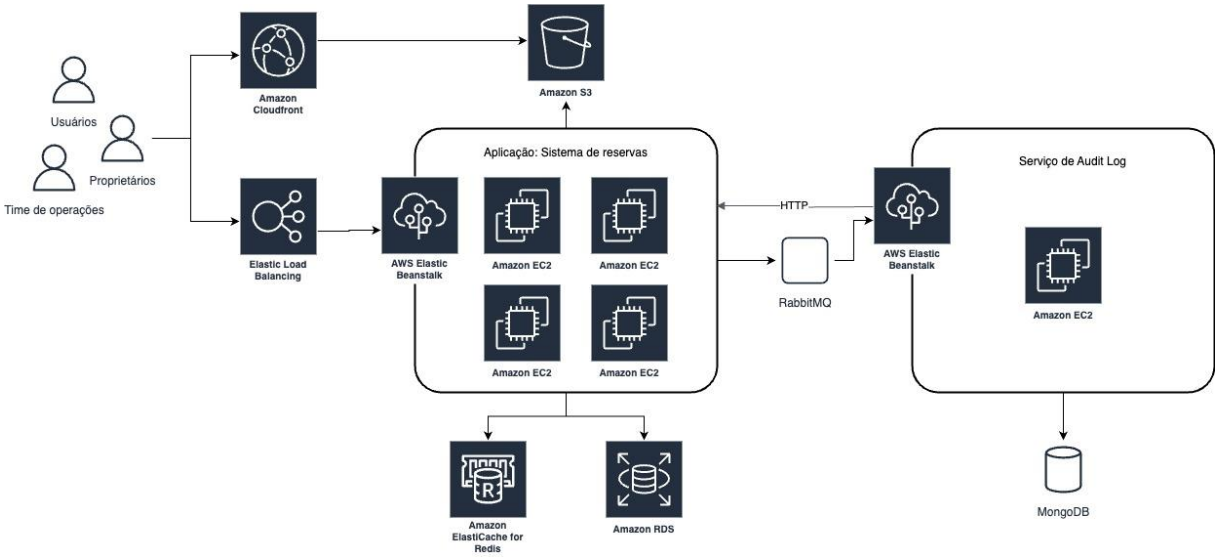

Figura 2: Diagrama da arquitetura da aplicação após da implantação do sistema de logs de auditoria
Fonte: Dados originais da pesquisa

Coleta de dados do impacto no desempenho do sistema principal

Utilizou-se a ferramenta Grafana K6 para executar esses testes de performance antes e depois da implantação do novo sistema. O objetivo principal da avaliação foi verificar qual o impacto causado na latência dos sistemas principais após a introdução do sistema de registros de logs. Compararam-se indicadores de tempo de resposta das operações principais dos módulos já existentes para avaliar se o novo sistema atendeu aos requisitos de desempenho estabelecidos. Para essa análise, fez-se uso da métrica de tempo de duração das requisições HTTP, identificado pela variável "http_req_duration", disponibilizada pela ferramenta, conforme apresentado na Tabela 3.

Tabela 3. Comparação dos tempos de respostas do teste de carga em milissegundos

Teste	Média	Máximo	Mediana	Mínimo	p90	p95	p99
Pré-implantação	238	585	238	208	254	263	303
Pós-implantação	245	653	243	210	262	279	326
Latência adicionada	7	68	5	2	8	16	23

Fonte: Dados originais da pesquisa

Para a elaboração do script de testes de carga, utilizaram-se como referência os dados de acesso obtidos por meio da ferramenta Google Analytics, considerando períodos de baixa e alta demanda. A simulação reproduziu o uso cotidiano dos principais pontos de entrada da aplicação, com variação de carga entre 300 e 1500 usuários simultâneos, mantendo o pico por cerca de quatro minutos, em um teste com duração total de 16 minutos.

Os testes foram conduzidos em ambiente de homologação. A infraestrutura foi implantada na região Leste dos Estados Unidos, Norte da Virgínia (US-East-1) da AWS. A

aplicação principal foi executada em duas instâncias de servidores do tipo t3.medium, gerenciadas pelo serviço AWS Elastic Beanstalk. O banco de dados relacional utilizado foi o Amazon RDS PostgreSQL, com uma instância do tipo db.t4g.large. Para a fila de mensagens, responsável por intermediar o envio dos dados de auditoria, foi utilizada uma instância do serviço Amazon MQ do tipo mq.m5.large. Já os registros de auditoria foram armazenados em um banco de dados MongoDB Atlas, em um cluster com execução sob demanda serverless.

Ciclo 3: Avaliação da ação

Para complementar a análise de desempenho e avaliar a percepção de valor da ferramenta no contexto operacional, realizou-se a coleta de uma avaliação qualitativo após um período de duas semanas de utilização do sistema em ambiente de produção. Foram conduzidas entrevistas estruturadas com os gestores operacionais, previamente selecionados. O objetivo foi capturar as impressões iniciais sobre a utilidade da ferramenta, a facilidade de uso para a investigação de incidentes e o impacto percebido na resolução de chamados. Os depoimentos coletados foram transcritos e categorizados para posterior análise na seção de Resultados e Discussão.

Resultados e Discussão

A implantação do sistema de auditoria em ambiente de homologação validou sua viabilidade técnica. Os testes confirmaram o correto funcionamento de todas as etapas do fluxo de auditoria: a interceptação das operações via Prisma ORM, a comparação entre os dados antigos e novos, o envio dos eventos à fila de mensagens, o consumo assíncrono pelo microsserviço e a persistência final dos registros no banco de dados NoSQL. A escolha metodológica pela implementação da auditoria na camada da aplicação mostrou-se acertada, proporcionando maior flexibilidade e a captura de metadados do contexto de negócios, algo que não se limita apenas a comandos DML.

Quanto aos aspectos de performance, os testes indicaram um impacto médio de 7 milissegundos na latência do sistema principal, com um pico de 68 milissegundos. Esse acréscimo, resultante das operações adicionais de consulta, diferenciação de dados e publicação na fila, manteve-se confortavelmente dentro do limite de 150 milissegundos estabelecido como requisito não funcional. Dessa forma, a decisão de utilizar comunicação assíncrona foi crucial para mitigar o impacto no desempenho, garantindo o desacoplamento do registro de logs das operações principais, conforme apontado na literatura sobre padrões de integração.

O valor prático da ferramenta foi demonstrado durante as duas primeiras semanas de operação. O sistema foi utilizado pela equipe de atendimento para investigar dois chamados

de hóspedes sobre incorreções no número de pessoas em suas reservas, o que levou à identificação de um bug em um fluxo específico. A análise dos depoimentos coletados com os gestores operacionais corroborou essa percepção de valor. Um dos supervisores relatou que a equipe tinha um ponto de partida para investigações, superando a dificuldade anterior de lidar com situações nas quais a equipe não tinha meios para investigar a origem das alterações. Outro destacou que a ferramenta evidenciou que muitas inconsistências eram, na verdade, ações de usuários que antes não eram rastreadas, agregando valor para além da simples correção de falhas técnicas. Essas observações confirmam as premissas teóricas de que logs de auditoria são fundamentais para a investigação de falhas e a segurança dos sistemas (Das e Chu, 2023; Adkins et al., 2020).

Entretanto, a aplicação prática revelou que a adoção de microsserviços elevou a complexidade do sistema, exigindo maior atenção à comunicação e ao gerenciamento dos serviços. A equipe de 12 desenvolvedores enfrentou elevada sobrecarga cognitiva e operacional, de modo que os ganhos em agilidade não se materializaram. Dessa forma, verifica-se que, para obter os benefícios prometidos pelos microsserviços, é importante alinhar decisões arquiteturais ao dimensionamento das equipes, conforme destacado por Skelton e Pais (2019), que enfatizam a necessidade de organizar equipes de negócios e tecnologia para facilitar o fluxo rápido de entrega. Foi identificado que uma possível futura divisão em microsserviços seriam os 3 principais módulos já existentes no sistema, ao invés de uma funcionalidade isolada, reassentando melhor a estrutura da empresa. Essa análise reforçou a necessidade de que futuras decisões arquiteturais avaliem não apenas os aspectos técnicos, mas também a estrutura e a capacidade da equipe de desenvolvimento.

Conclusão

O objetivo de implantar um sistema de logs de auditoria para melhorar a rastreabilidade e auxiliar na resolução de problemas em uma plataforma de reservas online foi alcançado. A solução desenvolvida demonstrou ser uma ferramenta eficaz, fornecendo os mecanismos necessários para o monitoramento de alterações de dados em um ambiente produtivo.

Concluiu-se que a implementação gerou um valor operacional tangível, permitindo que as equipes de atendimento e desenvolvimento analisassem o histórico de modificações para diagnosticar falhas com agilidade, conforme evidenciado pela rápida identificação de um bug. Adicionalmente, o uso da arquitetura de microsserviços, embora tenha aumentado a complexidade para a equipe atual, proporcionou um aprendizado organizacional estratégico sobre os pré-requisitos de estrutura de equipe necessários para obter os benefícios de agilidade prometidos por esse modelo.

Portanto, a solução atendeu ao seu propósito principal, sendo validada como uma ferramenta útil para a equipe de atendimento. Como limitação do estudo, aponta-se a não inclusão de uma análise de custos de infraestrutura, justificada pela utilização de créditos de programas de aceleração. Para trabalhos futuros, recomenda-se aprofundar a análise de custo-benefício da arquitetura de microsserviços em equipes de menor porte e, em uma evolução deste projeto, investigar a aplicação de algoritmos de machine learning sobre os logs de auditoria para a detecção proativa de anomalias, tanto nos processos de desenvolvimento de software quanto em relação à segurança dos sistemas.

Referências

Adkins, H.; Beyer, B.; Blankinship, P.; Lewandowski, P.; Oprea, A.; Stubblefield, A. 2020. Building Secure and Reliable Systems: Best Practices for Designing, Implementing, and Maintaining Systems. 1ed. O'Reilly Media, Sebastopol, CA, USA.

Brasil. 2021. Resolução CMN nº 4.893, de 26 de fevereiro de 2021. Dispõe sobre a política de segurança cibernética e sobre os requisitos para a contratação de serviços de processamento e armazenamento de dados e de computação em nuvem a serem observados pelas instituições financeiras e demais instituições autorizadas a funcionar pelo Banco Central do Brasil. Diário Oficial da União, Brasília, 01 mar. 2021. Seção 1, p. 82-83.

Das, B.K.S.; Chu, V. 2023. Security as Code: DevSecOps Patterns with AWS. 1ed. O'Reilly Media, Sebastopol, CA, USA.

Expedia Group, Inc. [EXPE]. 2023. 2023 Annual report. Disponível em: <https://s202.q4cdn.com/757635260/files/doc_financials/2023/ar/expe-2023-annual-report.pdf>. Acesso em: 19 fev. 2025.

Gil, A.L. 2000. Auditoria de Computadores. 5ed. Atlas, São Paulo, SP, Brasil.

Hohpe, G.; Woolf, B. 2003. Enterprise Integration Patterns. Addison-Wesley, Boston, MA, USA.

Instituto Brasileiro de Geografia e Estatística [IBGE]. 2024. Turismo 2023. Disponível em: <https://biblioteca.ibge.gov.br/visualizacao/livros/liv102116_informativo.pdf>. Acesso em: 08 mai. 2025.

Kanizawa, D.T.; Pinto, G.S. 2022. Arquitetura de microsserviços. Revista Interface Tecnológica 19(2): 308-318.

Kodakandla, P. 2023. Real-time data pipeline modernization: a comparative study of latency, scalability, and cost trade-offs in kafka-spark-bigquery architectures. International Research Journal of Modernization in Engineering Technology and Science 5(10): 3340-3349.

Maharjan, R.; Chy, M.S.H.; Arju, M.A.; Cerny, T. 2023. Benchmarking message queues. Telecom 4(2): 298-312.

Newman, S. 2021. Building Microservices: Designing Fine-grained Systems. 2ed. O'Reilly Media, Sebastopol, CA, USA.

Nõu, A. 2025. Investigating performance overhead of distributed tracing in microservices and serverless systems. Dissertação de Mestrado em Ciência da Computação. Vrije Universiteit Amsterdam, Amsterdam, Holanda.

Opara-Martins, J.; Sahandi, R.; Tian, F. 2017. A Holistic Decision Framework to Avoid Vendor Lock-in for Cloud SaaS Migration. Computer and Information Science 10(3): 29-47.

Sadalage, P.J.; Fowler, M. 2012. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional, Upper Saddle River, NJ, USA.

Skelton, M.; Pais, M. 2019. Team Topologies: Organizing business and technology teams for fast flow. IT Revolution, Portland, OR, USA.

Richardson, C. 2018. Microservices Patterns: With Examples in Java. Manning Publications, Shelter Island, NY, USA.

Tripp, D. 2005. Pesquisa-ação: uma introdução metodológica. Educação e Pesquisa 31(3): 443-466.

Apêndice

A. Códigos fonte da solução para coleta das alterações

```
25  private _buildAuditLogExtension(prisma: PrismaClient) {
26    const rabbitMQService = this.rabbitMQService
27    const requestContextService = this.requestContextService
28    const targetModels = ['Reservation', 'Accommodation']
29    return Prisma.defineExtension({
30      query: {
31        $allModels: {
32          async create({ model, args, query }) {
33            if (!targetModels.includes(model)) return query(args)
34
35            const oldData = {}
36            const newData = await query(args)
37            const auditLogMessage: AuditLogMessage = {
38              timestamp: new Date().toISOString(),
39              action: 'create',
40              resource: model,
41              resourceId: newData.id || 0,
42              data: deepDiff(oldData, newData),
43              context: requestContextService.getContext(),
44            }
45            await rabbitMQService.sendAuditLog(auditLogMessage)
46
47            return newData
48          },
49
50          async update({ model, args, query }) {
51            if (!targetModels.includes(model)) return query(args)
52
53            const oldData = await prisma[model].findUnique({
54              where: { id: args.where.id },
55            })
56            const newData = await query(args)
57            const auditLogMessage: AuditLogMessage = {
58              timestamp: new Date().toISOString(),
59              action: 'update',
60              resource: model,
61              resourceId: newData.id || 0,
62              data: deepDiff(oldData, newData),
63              context: requestContextService.getContext(),
64            }
65            await rabbitMQService.sendAuditLog(auditLogMessage)
66
67            return newData
68          },
69
70          async delete({ model, args, query }) {
71            if (!targetModels.includes(model)) return query(args)
72
73            const oldData = await prisma[model].findUnique({
74              where: { id: args.where.id },
75            })
76            const newData = await query(args)
77            const auditLogMessage: AuditLogMessage = {
78              timestamp: new Date().toISOString(),
79              action: 'delete',
80              resource: model,
81              resourceId: oldData.id || 0,
82              data: deepDiff(oldData, newData),
83              context: requestContextService.getContext(),
84            }
85            await rabbitMQService.sendAuditLog(auditLogMessage)
86
87            return newData
88          },
89        },
90      },
91    })
92  }
```

Figura 1: Código da implementação da extensão do Prisma ORM para interceptar as mudanças nos dados de reservas e acomodações
Fonte: Dados originais da pesquisa

```typescript
export function deepDiff(obj1: any, obj2: any): any {
  // O objeto resultado onde armazenaremos as diferenças
  const differences: any = {}

  // Percorre as chaves do primeiro objeto
  for (const key in obj1) {
    // Verifica se a chave existe em ambos os objetos
    if (obj1.hasOwnProperty(key)) {
      // Se ambos são objetos, encontra diferenças recursivamente
      if (typeof obj1[key] === 'object' && typeof obj2[key] === 'object') {
        const nestedDiff = deepDiff(obj1[key], obj2[key])

        if (Object.keys(nestedDiff).length > 0) {
          differences[key] = nestedDiff // Armazena as diferenças aninhadas
        }
      } else if (obj1[key] !== obj2[key]) {
        // Se os valores são diferentes, armazena a diferença
        differences[key] = {
          oldValue: obj1[key],
          newValue: obj2[key],
        }
      }
    } else {
      // Se a chave não está presente no segundo objeto, registra como diferença
      differences[key] = {
        oldValue: obj1[key],
        newValue: undefined,
      }
    }
  }

  // Percorre as chaves do segundo objeto para encontrar chaves que não estão no primeiro
  for (const key in obj2) {
    if (!obj1.hasOwnProperty(key)) {
      differences[key] = {
        oldValue: undefined,
        newValue: obj2[key],
      }
    }
  }

  return differences
}
```

Figura 2: Código do algoritmo para identificar as alterações dos dados
Fonte: Dados originais da pesquisa

B. Roteiro da entrevista estruturada

Pergunta 1

Qual foi sua primeira impressão sobre a finalidade e a utilidade da nova ferramenta de logs de auditoria quando ela foi apresentada? E essa impressão mudou após essas duas semanas de uso?

Reposta 1: Há muito tempo sentíamos falta de algo assim. Antes, quando um cliente ou proprietário ligava dizendo que "algo mudou sozinho na reserva", ficávamos de mãos atadas. Nessas duas semanas, mesmo usando poucas vezes, a impressão se confirmou. Agora temos um ponto de partida para investigar.

Resposta 2: Inicialmente, pareceu mais uma ferramenta técnica, algo mais para o time de desenvolvimento do que para nós da operação. Após usar para um caso específico, comecei a ver mais sentido, mas ainda não está totalmente claro como isso vai se integrar no nosso dia a dia, se tivesse como ter esses dados Zendesk, também seria interessante.

Resposta 3: A impressão não mudou: a ferramenta é útil. O que me surpreendeu foi a quantidade de alterações que ocorrem sem que a gente perceba. Já vimos que muitas inconsistências não são bugs, mas sim ações de diferentes usuários que não estavam sendo rastreadas.

Pergunta 2:

Em sua percepção, qual foi o impacto da ferramenta na capacidade da sua equipe de resolver os chamados dos clientes/proprietários nessas duas semanas? Você acredita que o tempo de resolução ou a qualidade da resposta mudou?

Resposta 1: Creio que o impacto foi positivo. No caso que mencionei, levamos 15 minutos para ter uma resposta definitiva para o cliente. Antes, isso se tornaria um chamado para a equipe de tecnologia que poderia levar horas para ser respondido. A qualidade da resposta também melhorou, pois passamos de "vamos verificar o que pode ter acontecido" para "verificamos que a alteração foi feita em tal dia e hora".

Resposta 2: Ainda é cedo para dizer que mudou drasticamente, pois foram poucos os casos em que precisamos usar. Mas o potencial é enorme. Também sabemos que temos uma base de dados para consultar, o que nos dá mais segurança para lidar com reclamações mais complexas.

Resposta 3: O impacto na operação? Ainda não sei te dizer, eu tenho que reforçar com o time a usar isso antes de chamar a ajuda de vocês (time de desenvolvimento). Mas acho sim que vai ajudar.

Pergunta 3:

Que melhorias ou funcionalidades adicionais você gostaria de ver nesta ferramenta?

Reposta 1: Poderia ter alguma coisa de AI, já que está tão na moda. Talvez algum alerta de que alguma coisa está errada antes da gente ter que lidar com o problema.

Reposta 2: Seria muito bom ter o histórico de alterações direto na tela de detalhes da reserva no Admin (sistema administrativo) e colocar esses dados dentro do Zendesk, iria facilitar usar isso.

Reposta 3: Não sei dizer o que poderia melhorar. Talvez colocar mais informações? Como por exemplo, saber quando o e-mail que enviamos foi aberto pode ser útil também.