UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia

Departamento de Computação

Trabalho de Conclusão de Curso - TCC

João Vitor Fidelis Cardozo

Um estudo sobre diferentes arquiteturas para aplicações mobile na nuvem

São Carlos

2024

João Vitor Fidelis Cardozo

Um estudo sobre diferentes arquiteturas para aplicações mobile na nuvem

Monografia apresentada ao Departamento de Computação da Universidade Federal de São Carlos como requisito parcial para obtenção do título de Bacharel em Engenharia de Computação.

Orientação Prof. Dr. Daniel Lucrédio

São Carlos

2024

Este trabalho é dedicado aos meus avós que mesmo distantes, me incentivaram e me forneceram todo o suporte para eu estar aqui hoje.

Agradecimentos

Agradecimentos a Daniel Lucrédio, professor Doutor do Departamento de Computação (DC) da Universidade Federal de São Carlos – UFSCar que forneceu todo o suporte para o desenvolvimento do aplicativo mobile Caronas Universitárias. Agradecimentos também a Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), por financiar parcialmente este trabalho, visto que a ferramenta que é o foco de estudo, foi desenvolvida em um período de iniciação científica. Agradeço à Universidade Federal de São Carlos – UFSCar e ao Departamento de Computação (DC) por todas as disciplinas ministradas e conhecimento proporcionado.

Agradecimentos também a todos os meus colegas e amigos de universidade, com os quais tive a oportunidade de compartilhar experiências que me trouxeram crescimento pessoal e profissional. Por fim, agradecimentos para todos os membros da minha família que sempre incentivaram os meus estudos e acreditaram que seria possível alcançar esses resultados, me fornecendo toda compreensão e suporte necessário para isso.

"O software pode ser feito do mesmo material que os sonhos, mas funciona no mundo físico."

(Robert C. Martin)

Resumo

No atual cenário tecnológico, observa-se uma grande demanda no mercado de software para desenvolvimento de aplicações, sejam elas web ou mobile. Com o surgimento da computação em nuvem, mostraram-se também diversas possibilidades de arquiteturas com propostas e tecnologias distintas. Este panorama oferece aos desenvolvedores de sistemas o acesso a diversas combinações tecnológicas, gerando questionamentos sobre a escolha do melhor conjunto e consequentemente, a construção da melhor arquitetura.

O objetivo desta pesquisa foi analisar e comparar três diferentes arquiteturas de software para aplicações móveis preparadas para serem hospedadas na nuvem, com foco em critérios específicos de avaliação. Inicialmente, a avaliação foi realizada no quesito modificabilidade, ou seja, foi mensurada a capacidade do software de ser modificado ou adaptado para situações específicas, por exemplo, a necessidade de inclusão de um campo inexistente ao banco de dados relacional e não relacional. Seguidamente, a confiabilidade do software foi avaliada com ênfase no aspecto de tolerância a falhas, analisando como o produto consegue manter a consistência de informações na presença de eventuais defeitos. Por fim, a segurança, focada em confidencialidade, foi investigada também para ser compreendido o grau que sistema assegura que os dados sejam exclusivamente acessíveis apenas aos indivíduos devidamente autorizados. Este estudo, também considerou as vantagens de possuir um back-end próprio e a utilização de um serviço de mensageria, embora esses critérios não tenham sido analisados com a mesma rigidez sistemática dos demais.

Dessa forma, após realizadas as comparações com base nos critérios estabelecidos, foram obtidos resultados que descrevem de maneira empírica os aspectos positivos e negativos de cada arquitetura proposta, de modo a proporcionar escolhas pautadas nesses aspectos, auxiliando assim, na decisão sobre qual arquitetura utilizar no processo desenvolvimento de software voltado para dispositivos móveis com tecnologias em nuvem.

De maneira geral, os resultados da comparação indicaram que o acesso direto a um serviço gerenciado de armazenamento, como o Firestore do Google Firebase, aprimora a modificabilidade e oferece uma gestão mais eficiente para o tratamento de falhas de requisição, além de assegurar a confidencialidade das informações dos usuários. Por outro lado, a utilização de um back-end próprio fornece o controle total sobre a infraestrutura e o código, além de otimizar os custos da aplicação a longo prazo. Por fim, destaca-se que o emprego de um serviço de mensageria, pode proporcionar maior escalabilidade e alta disponibilidade de dados.

Palavras-chave: Web, Móvel, Arquitetura Móvel, Mensageria, NoSQL, SQL

Abstract

In the current technological scenario, there is a significant demand in the software market for the development of applications, whether they are web or mobile. With the emergence of cloud computing, a variety of architectures with different proposals and technologies have also been revealed. This panorama offers system developers access to various technological combinations, raising questions about the choice of the best set and consequently, the construction of the best architecture.

The objective of this research was to analyze and compare three different software architectures for mobile applications prepared to be hosted in the cloud, focusing on specific evaluation criteria. Initially, the evaluation was carried out in terms of modifiability, that is, the software's capacity to be modified or adapted to specific situations was measured, for example, the need to include a non-existent field in the relational and non-relational database. Subsequently, the software's reliability was evaluated with an emphasis on fault tolerance, analyzing how the product maintains the consistency of information in the presence of potential issues. Finally, security, focused on confidentiality, was also investigated to understand the degree to which the system ensures that data is exclusively accessible only to properly authorized individuals. This study also considered the advantages of having one's own back-end and the use of a messaging service, although these criteria were not analyzed with the same systematic rigor as the others.

Thus, after making comparisons based on the established criteria, results were obtained that empirically describe the positive and negative aspects of each proposed architecture, in order to provide choices based on these aspects, thus assisting in the decision on which architecture to use in the software development process for mobile devices with cloud technologies.

Overall, the results of the comparison indicated that direct access to a managed storage service, such as Google Firebase's Firestore, enhances modifiability and offers more efficient management for handling request failures, in addition to ensuring the confidentiality of user information. On the other hand, the use of one's own back-end provides total control over the infrastructure and code, as well as optimizing the application's long-term costs. Finally, it is highlighted that the use of a messaging service can provide greater scalability and high data availability.

Keywords: Web, Mobile, Mobile Architecture, Messaging, NoSQL, SQL

Lista de ilustrações

Lista de siglas

BaaS *Back-end as a Service*

DaaS *Database as a Service*

EDA *Event Driven Architecture*

MQTT *MQ Telemetry Transport*

AMQP *Advanced Message Queuing Protocol*

RDBMS *Relational Database Management System*

FCM *Firebase Cloud Messaging*

ORM *Object-Relational Mapping*

ACK *Acknowledgement*

Sumário

1 Introdução

Atualmente o mercado de software possui uma grande demanda de desenvolvimento de aplicações, sejam elas *web* ou *mobile*. Com o surgimento da computação em nuvem e a ascensão dos microsserviços, torna-se possível construir arquiteturas diferentes da abordagem monolítica tradicional, na qual os componentes são dependentes um dos outros, visto que todas as funções de uma aplicação estão centralizadas em um único código base (LI, 2017). Sendo assim, as plataformas de nuvem, e os serviços fornecidos por essas, disponibilizam aos desenvolvedores de software a opção de escolha por esses recursos, sem a necessidade de prover e realizar manutenção nos servidores. Além disso, é necessário ressaltar que a capacidade computacional e a infraestrutura oferecida por esses serviços possuem a habilidade de escalar horizontal ou verticalmente, conforme a necessidade de leitura, escrita ou atualização dos dados.

Uma das principais vantagens de desfrutar de uma diversidade de serviços de nuvem no âmbito do desenvolvimento de software é a capacidade de integrar diferentes serviços e tecnologias em um único sistema. Esta multiplicidade de opções tecnológicas, embora benéfica, introduz dúvidas e complexidades durante a seleção da arquitetura considerada ótima para uma dada aplicação. Dessa forma, torna-se nítido o desafio de escolha e identificação de combinações de recursos e tecnologias que sejam eficazes para um projeto em questão, considerando seu escopo e documento de requisitos.

Entre as possibilidades de serviços de nuvem, está o *Google Firebase*. Esse serviço proporciona ao desenvolvedor de software ferramentas de extrema importância que auxiliam na construção de um projeto. Por se tratar de um *Back-end as a Service* (BaaS) e *Database as a Service* (DaaS), o *Firebase* fornece um conjunto de instrumentos que permitem o gerenciamento do lado do servidor (*back-end*) e a administração de informações em um banco de dados de forma on-line, sem haver a necessidade de infraestrutura própria para isso. Além disso, é preciso ressaltar que esse tipo de serviço pode ser combinado com outras tecnologias, por exemplo, serviço de mensageria, como o *RabbitMQ*, ou até um banco de dados relacional, como o *MySQL*. Nesse último contexto, considera-se o cenário no qual um desenvolvedor pode utilizar o banco de dados do *Firebase*, por exemplo, o *Realtime Database* e, o *MySQL*, para propósitos diferentes, mas, em conjunto para um único sistema.

1.1 Objetivos

No presente estudo, foram avaliadas as combinações de tecnologias diferentes com o serviço de nuvem do *Google Firebase*, adotando uma abordagem comparativa. Para isso,

foi utilizada como ferramenta principal uma aplicação móvel denominada por *Caronas Universitárias*, construída durante um período de iniciação científica. Esse sistema possui três arquiteturas distintas e o mesmo propósito, a de gerenciar caronas na mesma cidade para universitários. Sendo assim, para as três arquiteturas foram utilizados alguns dos serviços do *Google Firebase*, sendo esses, autenticação, *Storage*, *Realtime Database*, além do *Firestore* para a primeira arquitetura. Para a segunda e a terceira arquitetura, o *Firestore* foi substituído por um banco de dados relacional, sendo esse o *MySQL*, no qual as ações de CRUD (*Create, Read, Update e Delete*), foram feitas por meio de requisições HTTP com um servidor construído em *Node.js* e *Express*. Para a terceira arquitetura, foi explorado o serviço de mensageria *RabbitMQ*, o qual armazena as requisições em filas específicas e encaminha para o banco de dados *MySQL* ou para os usuários da aplicação que recebem esses dados.

1.2 Metodologia

Como base de comparação das três arquiteturas supracitadas, foi utilizada a versão mais recente da ISO/IEC 25010 (ISO/IEC 25010, 2011), a qual se trata de uma norma internacional para avaliar a qualidade de um software. Essa ISO possui vários critérios e subcritérios, porém, para este trabalho, os subcritérios escolhidos para comparar foram manutenibilidade, o qual é subcritério de modificabilidade, confiabilidade, subcritério de tolerância a falhas e confidencialidade, subcritério de segurança. Adicionalmente, é importante enfatizar que, embora esses critérios foram utilizados para avaliar cada arquitetura com maior profundidade, também foram considerados outros critérios na avaliação dessas construções do sistema, os quais se referem as vantagens de ter um *back-end* próprio e a utilização de um serviço de mensageria. É importante ressaltar, que todos os critérios listados não foram avaliados em sua plenitude e sim, para casos específicos dentro de inúmeras possibilidades. A escolha desses critérios se deu pelo fato de que esses fatores são comumente considerados durante um projeto arquitetural, além do fato de que foram observadas diferenças significativas em relação a eles nas três arquiteturas estudadas, o suficiente para motivar este estudo.

Para as três arquiteturas, o processo de análise foi o mesmo, ou seja, o código foi explorado com base nas seguintes operações:

- Para a análise de modificabilidade, foram feitas requisições no formato HTTP e inserido um novo campo no banco de dados;
- Para a análise de tolerância a falhas, a internet do usuário autor deste trabalho foi desconectada e realizada uma requisição HTTP visando cadastrar um usuário. Após isso, foi observado se os dados da requisição foram inseridos no banco de dados

(para a primeira e segunda arquitetura) ou na fila do *RabbitMQ* (para a terceira arquitetura), quando retomada a conexão;

- Para a análise de confidencialidade, foi realizada a tentativa de obter dados de outros usuários por meio de requisições HTTP, via *API do Cloud Firestore* ou servidor.

Como comentado anteriormente, essa pesquisa visa avaliar de forma empírica três arquiteturas de software que possuem o mesmo propósito, contribuindo para o estado da arte com as análises e resultados obtidos referente a esse sistema, para cada conjunto de decisões e serviços utilizados.

1.3 Contribuições

Como contribuições, para esta pesquisa, foram alcançados resultados obtidos de forma empírica, identificando vantagens e desvantagens para cada subcaracterística no contexto dos critérios selecionados da ISO/IEC 25010. Por exemplo, para a primeira arquitetura de software, foram encontradas vantagens nos critérios de modificabilidade, tolerância a falhas e confidencialidade em relação às outras arquiteturas. O *Google Firebase* fornece um conjunto de ferramentas que lida com inserção de dados de forma simplificada e tolerância a falhas, ou seja, em caso de falha durante uma requisição para esse serviço (por exemplo, o *Firestore*), essa é armazenada em cache local e enviada quando o cenário de falha não estiver mais presente, o que garante a consistência dos dados. Além disso, no quesito de confidencialidade, as regras de segurança podem ser configuradas diretamente pela aba de Regras do *Cloud Firestore* e isso, garante que um usuário possa ou não ter acesso aos dados de outros usuários, tudo isso, com uma sintaxe simples e poucas operações necessárias.

Por outro lado, para as outras duas arquiteturas, também foram notadas vantagens nos quesitos de "Outros critérios". Entre esses, destacaram-se os benefícios de ter um *back-end* próprio e utilizar serviço de mensageria. Para o *back-end* próprio, o controle total sobre a infraestrutura e o código, respostas personalizadas do servidor para o *front-end* além de otimização de custos a longo prazo, foram características essenciais observadas durante a manutenção ou evolução de sistemas de software. Além disso, a utilização de um serviço de mensageria, por exemplo, o *RabbitMQ*, traz diversos benefícios, entre esses o balanceamento de carga, a resiliência e alta disponibilidade e a escalabilidade de software.

2 Trabalhos Relacionados

Outros trabalhos que implementaram arquiteturas semelhantes às arquiteturas propostas neste estudo já foram realizados, mas como cada uma possui suas especificidades, foram encontrados estudos na literatura que se aproximam de uma das arquiteturas, mas que não possuem as mesmas combinações de tecnologias aqui propostas.

Pratama, Prihatmanto e Sukoco (2020) fizeram um estudo tratando de um problema semelhante ao proposto neste trabalho, ou seja, a comunicação entre dois dispositivos (ou sistemas) com a troca de informações constante. Nesse caso, foi recomendada uma arquitetura para um sistema de monitoramento no âmbito do transporte público, o qual precisa lidar com o gerenciamento de grande volume de dados no quesito de troca de informações, especificamente entre elas, estão as coordenadas relacionadas ao Sistema de Posicionamento Global (GPS). Neste artigo, são tratadas as informações de comunicação síncrona e assíncrona e isso é definido com base na arquitetura utilizada, por exemplo, quando uma parte do sistema depende da outra, geralmente a comunicação realizada é síncrona e isso ocasiona problemas durante a troca de informações, pois as duas partes precisam estar online e disponíveis para a sincronização desses dados. Por outro lado, na comunicação assíncrona isso não acontece, visto que o sistema pode continuar as suas funções sem depender de uma resposta para isso. Sendo assim, a solução para esse problema, que foi sugerida no artigo citado, foi implementar um tipo de arquitetura orientada a eventos, dirigida principalmente, por um serviço de mensageria, especificamente o *rabbitMQ*. Quando esse tipo de arquitetura é implementada, os problemas de troca de informação que dependem de outros módulos do sistema são solucionados, visto que no cenário de indisponibilidade de um dos módulos, o *rabbitMQ* guarda as informações em uma fila e tudo isso consegue ser processado em um momento futuro, assim que houver a disponibilidade do módulo em questão.

No estudo conduzido por Sjodin e Lotfy (2019), é destacada a relevância emergente da computação em nuvem, especificamente no contexto de aplicativos móveis. O artigo aborda os modelos de serviços conhecidos por BaaS e DaaS, enfatizando como essas duas modalidades conseguem auxiliar e tornarem-se facilitadores no desenvolvimento de aplicativos móveis. *Back-end as a Service* e *Database as a Service*, se encaixam muito no contexto do *Google Firebase*, visto que o primeiro serviço (*BaaS*), é projetado para prover aos desenvolvedores um conjunto integrado de ferramentas destinadas ao gerenciamento do *back-end*, sem haver a necessidade de infraestrutura para isso, ou seja, isso permite que os desenvolvedores se concentrem primariamente no *front-end*, enquanto a complexidade do lado do servidor é fornecida pelo provedor do serviço, nesse caso, o *Firebase*. Similarmente ao modelo *BaaS*, é possível afirmar que *Database as a Service* segue uma filosofia análoga,

permitindo aos desenvolvedores de software o armazenamento e gerenciamento de suas informações em um banco de dados, sem a necessidade de configurar e manter uma infraestrutura física. Como exemplos de *BaaS* e *DaaS*, podem ser citados os serviços de autenticação e de armazenamento do *Firebase* (*Realtime Database, Firestore*, entre outros). Ademais, é preciso dizer que este artigo levanta a importância da computação em nuvem e diz ser necessário capacitar alunos para conseguirem atuar em um mercado com esse modelo de computação, haja vista a alta demanda pela computação em nuvem atualmente.

Na pesquisa realizada por Prellwitz, Parzyjegla e Mühl (2023), são apresentadas abstrações de programação para protocolos de mensagens com arquiteturas orientadas a eventos, ou seja, *Event Driven Architecture* (EDA). Além disso, neste artigo, são ressaltadas características dos sistemas baseados em mensageria, por exemplo, baixo acoplamento e flexibilidade, as quais também são consideradas características de sistemas construídos com arquitetura de microsserviços. O estudo referencia o cenário de que as características podem se tornar completamente desconhecidas, visto que sempre tem que ser considerado o fator de escalabilidade e sendo assim, é proposto um *middleware* com abstrações independentes de plataforma e protocolo para vinculação de objetos remotos e interações baseadas em *push* e *pull*. O estudo introduz um esquema de endereçamento expansível baseado em comunicação *publish/subscribe* e discute abstrações para lidar com a incerteza dos protocolos de transporte, utilizando *MQ Telemetry Transport* (MQTT) e *Advanced Message Queuing Protocol* (AMQP) como exemplos, os quais são suportados pelo *rabbitMQ* também.

Na literatura também existem estudos que tratam do gerenciamento de dados para aplicativos móveis que dependem de informações geolocalizadas. Em outro estudo (SANTOS et al., 2023), essa problemática é abordada de modo a ressaltar a utilização de bancos de dados NoSQL para atuar com o gerenciamento dessas informações. No artigo, o problema é direcionado ao armazenamento dessas informações no banco de dados, ou seja, é afirmado que os sistemas de armazenamento atuais não levam em consideração a localização geográfica dos dados e acabam impondo modelos de consistência uniformes, os quais tornam o armazenamento e as operações com essas informações não tão confiáveis. Dessa forma, o artigo descreve a arquitetura, o modelo de dados e as características do *FocusDB*, o qual é tratado como um novo sistema de gerenciamento de dados que leva em consideração a localização do usuário também. Além disso, os autores apresentam um caso de uso aplicado a um aplicativo de localização de vagas de estacionamento, demonstrando a eficácia do *FocusDB* em fornecer informações atualizadas e detalhadas com base na localização do usuário. A avaliação experimental do *FocusDB* mostrou melhorias significativas no desempenho em comparação com alternativas clássicas, validando sua abordagem de adaptar a entrega de dados à localização do objeto e do usuário. O artigo conclui destacando o potencial do *FocusDB* para aplicações móveis que dependem de dados geolocalizados e delineia planos para futuras pesquisas visando aprimorar modelos de consistência e protocolos de replicação baseados em localização.

Khawas e Shah (2018) fizeram um estudo que traz informações sobre a aplicação do *Firebase* no âmbito do desenvolvimento Android. A motivação inerente à pesquisa parte do pressuposto de que existe a dificuldade de tratar dados não estruturados, no contexto dos Sistemas de Gerenciamento de Banco de Dados Relacionais (RDBMS) e sendo assim, o *Firebase*, como tecnologia emergente, se destaca nos quesitos de eficiência e rapidez quando comparado aos RDBMS tradicionais. O artigo oferece uma análise detalhada dos diversos serviços fornecidos pelo *Firebase*, incluindo *Firebase Cloud Messaging* (FCM), *Firebase Auth, Real-time Database, Firebase Storage, Firebase Test Lab* para Android e *Firebase Crash Reporting*, evidenciando, para cada serviço, suas funcionalidades e aplicações no contexto de desenvolvimento de software. Além disso, no estudo realizado, o *Firebase* é comparado com outros sistemas de gerenciamento de banco de dados, ressaltando suas diferenças em termos de armazenamento de informações (no *Realtime Database* do *Firebase*, os dados são armazenados no formato JSON (Google Firebase, 2024a) e no *Cloud Firestore*, é possível utilizar matrizes, objetos aninhados (Google Firebase, 2024b), entre outras estruturas e valores, enquanto os outros bancos de dados SQL armazenam em linhas e colunas de uma tabela); flexibilidade de esquema, ou seja, o *Firebase* possui um esquema dinâmico de dados e portanto, podem ser adicionados, alterados ou excluídos, enquanto os RDBMS possuem um esquema com flexibilidade restrita; tipagem ou especificidade, ou seja, no *Firebase* os dados não possuem um tipo ou estruturas definidas, enquanto nos outros bancos de dados SQL existe a tipagem dos dados. Outras comparações além do modelo de dados também foram realizadas, sendo essas relacionadas à classificação de banco de dados, desenvolvedores e linguagens suportadas. Em outra seção do artigo, é feito um passo a passo destinado à implementação do *Firebase* no desenvolvimento de aplicativos Android, abrangendo desde a criação de um projeto no console do *Firebase* até a implementação de várias de suas funcionalidades, como autenticação e armazenamento de dados em tempo real. Concluindo, o estudo referenciado ressalta como o *Firebase* facilita o desenvolvimento de aplicativos Android, tornando-os mais rápidos e eficientes, eliminando a necessidade de linguagens de terceiros. Por fim, também é destacado o potencial de expansão e exploração de novas funcionalidades no desenvolvimento de aplicativos Android utilizando o *Firebase*, indicando um caminho promissor para futuras inovações na área.

Emmadi e Potluri (2019) abordaram em outro estudo, o desenvolvimento de uma aplicação de mensagens instantâneas para Android utilizando o *Firebase*. No artigo é ressaltada a importância da comunicação entre as pessoas por meio da internet com pouca ou nenhuma latência. Dessa forma, o *Firebase* é destacado mais uma vez como uma das plataformas que fornece serviços em nuvem e de banco de dados em tempo real, por exemplo, por meio do *Realtime Database*, o que possibilita o auxílio aos desenvolvedores de software para a implementação dessas aplicações. O artigo também detalha algumas das funcionalidades e serviços do *Firebase*, como o armazenamento em nuvem, recursos para relatórios de falhas e banco dados em tempo real, mas, o foco do estudo é apresentar

uma aplicação para a comunicação em tempo real entre usuários. O sistema é desenvolvido especificamente para a plataforma Android e possibilita a comunicação de texto entre usuários por meio da Internet e é baseado no modelo *BaaS*, fornecido pelo *Firebase*.

Em outro artigo (SHARVARI; NAG, 2019), são investigadas as capacidades e diferenças entre três sistemas modernos de mensageria, sendo esses, *Apache Kafka*, *rabbitMQ* e *NATS Streaming*. O estudo inicia falando da necessidade de serviços de mensageria que sejam escaláveis, tolerantes a falhas e com baixa latência, haja vista o consumo exponencial de dados e a carência por comunicação entre plataformas. É necessário dizer que essas plataformas de mensagens podem ser aplicadas em *big data*, *cloud native* e principalmente, na construção de arquitetura de microsserviços. As aplicações dos dias atuais possuem natureza dinâmica, ou seja, pode mudar a qualquer momento o volume de informações e dados a serem tratados. Consequentemente, a comunicação síncrona, grande parte das vezes, não é suficiente para suprir todas as possíveis necessidades. Sendo assim, na pesquisa referenciada, o propósito é avaliar a adequação dos serviços de mensageria previamente citados, para aplicações críticas em tempo real, considerando os parâmetros de entrega de mensagens, persistência das mensagens, ordenação das mensagens, taxa de transferência, latência, disponibilidade e escalabilidade. Em primeiro lugar, é introduzido o paradigma de produtor, consumidor e filas de mensagens, especificando como esses serviços já referenciados conseguem lidar com o processamento das informações. Em uma parte do estudo, é citado o *rabbitMQ*, o qual é referenciado como um sistema de corretagem de mensagens versátil e agnóstico a linguagem. Esse serviço de mensageria é reconhecido por sua capacidade de comunicação assíncrona e desacoplada, além de ser adequado para roteamento complexo em aplicações de Internet das Coisas (IoT).

No âmbito da pesquisa atual, destaca-se a relevância de estudos anteriores focados na integração da tecnologia NoSQL baseada em nuvem com bancos de dados relacionais de dispositivos móveis, especialmente o *SQLite*, o qual também foi empregado neste estudo. Sjodin, Mason e Lotfy (2020) investigaram como o *Firebase Realtime Database*, um banco de dados NoSQL, pode servir como a principal fonte de dados para usuários, enquanto o *SQLite* desempenha o papel de um cache local, mantendo as informações quando o acesso ao *Realtime Database* está indisponível. Dessa forma, utilizando para esse propósito o *SQLite*, assim que houver novamente o acesso ao banco de dados, as informações são escritas e a consistência dessas são mantidas. O estudo referenciado utiliza-se do *Google Firebase* como serviço de nuvem e a estrutura de configuração para aplicativos móveis Android fornecida por esse serviço, para ser possível aplicar a proposta de salvamento das informações após *backup* local. Sendo assim, é necessário ressaltar que o objetivo é demonstrar como essa abordagem de armazenamento de dados pode ser aplicada em diferentes provedores de nuvem e plataformas móveis, além de tornar evidentes as vantagens de combinar o armazenamento de dados em nuvem e local (no próprio dispositivo).

Em (MADAMINOV; ALLABERGANOVA, 2023) foi feito um estudo, no qual é explorada a eficiência do *Firebase* em relação a outros sistemas de bancos de dados, em especial, o *MySQL*. Para isso, primeiramente, o artigo aborda as várias ferramentas que o *Google Firebase* proporciona aos desenvolvedores, sendo essas, o tratamento dos dados em relação à segurança das informações, operações de alteração e criação de dados, armazenamento de arquivos e uso de funções na própria plataforma. Para a comparação, nos exemplos mostrados na pesquisa, foram utilizados os serviços de autenticação, *Realtime Database, Storage e Function*, dentro do próprio console desse serviço de nuvem. Primeiramente, foram exibidas algumas etapas para realizar a autenticação na plataforma. Em segundo lugar, foram mostrados os dados populados no *Realtime Database* e, comentado sobre o tipo de estrutura na qual esses dados são armazenados, a qual se refere ao formato JSON. Por fim, para realizar uma comparação entre o *Firebase* e o *MySQL* de forma quantitativa, foram feitas operações de criação, leitura, atualização e remoção (*CRUD*), executando essas em ambos bancos de dados (*MySQL* e *Realtime Database*). Essas operações, foram realizadas 50 vezes (em cada banco de dados) e refletem, funções básicas para inserir e atualizar as informações em ambos bancos de dados. Como resultados, foi obtido que o *Firebase Realtime Database* foi mais eficiente nas operações de criar e atualizar informações, em relação ao *MySQL*.

Acevedo, Jorge e Patiño (2017) realizaram um estudo que apresenta uma metodologia para a transição de sistemas monolíticos para arquiteturas baseadas em microsserviços, no contexto de aplicações desenvolvidas para *web*. No início do artigo, são discutidos os desafios associados às aplicações construídas com base em arquiteturas monolíticas. Dessa forma, os autores enfatizam os problemas de disponibilidade, de manutenção e também de escalabilidade desse tipo de arquitetura. Após exposto isso, são mostradas as vantagens e os motivos de utilizar arquiteturas baseadas em microsserviços, as quais solucionam os problemas previamente citados, além de fornecer a capacidade de autocontrole de falhas no sistema. A metodologia proposta para a transição de sistemas é descrita em detalhes, abrangendo desde a análise do modelo de negócios de uma aplicação monolítica até a definição de subdomínios funcionais. Esses subdomínios servem como base para a fragmentação do monólito e o protótipo dos microsserviços. O processo é dividido em várias etapas, incluindo análise e *design*, implementação, teste e integração contínua, em um ciclo de vida evolutivo que permite uma transformação escalonada e controlada.

Conforme a literatura revisada, arquiteturas ou tecnologias semelhantes às propostas neste estudo foram encontradas, mas, cada uma com características distintas. Alguns destes trabalhos, destacaram as diferenças entre comunicação síncrona e assíncrona, especialmente no quesito de troca de informação nesses dois tipos de comunicação, para grandes volumes de dados. Nesse contexto, outras pesquisas relatam sobre a utilização de serviços de mensageria, comparando esses com os mais utilizados no mercado de software. Além disso, foram ressaltados os benefícios desse tipo de serviço quando utilizado nas aplicações, os

quais se referem a escalabilidade e capacidade de re-entrega de dados em caso de falhas, isso após a recepção da solicitação pelo servidor. Portanto, no contexto de troca de informações para grandes volumes de dados, a utilização de um serviço de mensageria, por exemplo, o *rabbitMQ*, soluciona problemas para o cenário que um módulo do sistema, depende do outro.

Outros artigos, abordaram os modelos de BaaS e DaaS, como os fornecidos pelo *Google Firebase*. Nestes modelos, permite que o desenvolvedor concentre o desenvolvimento do software no *front-end* da aplicação, haja vista que as complexidades relacionadas a servidores e infraestrutura são providas e gerenciadas pelo serviço de nuvem, nesse caso, o *Google Firebase*. Além disso, dentro dessas complexidades, estão um conjunto de ferramentas que também são providas para gerenciamento de erros e configuração das regras de segurança, tudo isso, de forma simplificada. Esses erros, se referem a possíveis falhas durante as operações de leitura, escrita, atualização ou remoção de informações nos bancos de dados hospedados na nuvem (*Firestore*, por exemplo). Os mecanismos automáticos providos para tratamento dessas requisições, incluem o armazenamento em cache local, no próprio dispositivo, e na tentativa de reenvio sob condições apropriadas.

Já em outros estudos, foi explorado o uso de bancos de dados não relacionais (NoSQL) para o armazenamento de informações geolocalizadas, utilizando, por exemplo, o *FocusDB*. Este tipo de banco de dados é utilizado em aplicações projetadas para gerenciar grandes volumes de informações, buscando por escalabilidade. Adicionalmente, em outro artigo, uma comparação foi realizada entre o armazenamento e gerenciamento de dados em Sistemas de Gerenciamento de Banco de Dados Relacionais - *Relational Database Management System* (RDBMS) e o esquema dinâmico de dados do *Firebase*. Nesta comparação, é destacada a facilidade de alteração e exclusão de dados no *Firebase*, em contraste com as restrições mais significativas encontradas nos RDBMS. A Tabela 1 apresenta um sumário dos estudos revisados, facilitando a análise das contribuições da literatura existente.

Tabela 1 – Sumarização dos estudos

Referência	Assunto
Pratama, Prihatmanto e Sukoco (2020)	Estudo sobre a comunicação entre dispositivos em sistemas de monitoramento de transporte público, recomendando uma arquitetura orientada a eventos com RabbitMQ para gerenciamento de grandes volumes de dados e troca de informações GPS de maneira assíncrona.
Sjodin e Lotfy (2019)	Discussão sobre a importância da computação em nuvem e da capacitação de alunos para atuar nesse contexto. É destacada a importância dos modelos de *Back-end as a Service* e *Database as a Service* no contexto do desenvolvimento de aplicativos móveis com ênfase no Google Firebase.
Prellwitz, Parzyjegla e Mühl (2023)	Apresentação de abstrações de programação para arquiteturas orientadas a eventos (EDA), destacando características como baixo acoplamento e flexibilidade, com menção a MQTT e AMQP por meio do RabbitMQ.
Santos et al. (2023)	Estudo sobre o uso de bancos de dados NoSQL para gerenciamento de dados geolocalizados. Aplicação do FocusDB em um aplicativo de localização de vagas de estacionamento, apontando melhorias para modelos de consistência.
Khawas e Shah (2018)	Análise do uso do Firebase no desenvolvimento Android, comparando com RDBMS tradicionais e destacando a eficiência e rapidez do Firebase em tratar dados não estruturados. Além disso, são destacados alguns serviços do Google Firebase.
Emmadi e Potluri (2019)	Desenvolvimento de uma aplicação de mensagens em tempo real para Android utilizando Firebase Realtime Database, enfatizando a importância da comunicação em tempo real com baixa latência. Além disso, é ressaltada a importância do modelo BaaS.
Sharvari e Nag (2019)	Comparação dos sistemas de mensageria Apache Kafka, RabbitMQ e NATS Streaming, considerando os critérios de escalabilidade, tolerância a falhas e baixa latência.
Sjodin, Mason e Lotfy (2020)	Análise da integração do Firebase Realtime Database com SQLite para armazenamento de dados em nuvem e cache local em dispositivos móveis.
Madaminov e Allaberganova (2023)	Exploração da eficiência do Firebase em comparação com MySQL, abordando ferramentas de segurança, operações CRUD e eficiência em operações de banco de dados.
Acevedo, Jorge e Patiño (2017)	Metodologia para transição de sistemas monolíticos para arquiteturas baseadas em microsserviços, discutindo vantagens como escalabilidade e autocontrole de falhas.

3 Abordagem Metodológica

No presente estudo, para assegurar a obtenção de resultados significativos e relevantes, adotou-se a seguinte abordagem metodológica: inicialmente, recorreu-se à revisão da literatura, a qual incluiu as buscas em mecanismos automáticos. Por conseguinte, foram empregados também, os métodos de *Snowballing* (bola de neve) e a realização de um estudo de caso. Para a coleta de artigos científicos que serviram como base de pesquisa, utilizou-se do motor de busca Google Acadêmico. Adicionalmente, foram exploradas as bases de dados ACM Digital Library e IEEE como fontes de informação. Com essa abordagem, foi possível realizar uma análise assertiva e fundamentada para o desenvolvimento do estudo.

A revisão da litetura desempenhou um papel fundamental nesta pesquisa, visando analisar criticamente e sistematizar as informações dos trabalhos relevantes que já foram publicados. O método inicialmente empregado, conhecido como *Snowballing*, foi utilizado com o intuito de encontrar por meio de trabalhos já existentes na literatura outros trabalhos que foram referências nesses já construídos. Por meio dessa técnica, foi possível realizar um levantamento bibliográfico de estudos com relação ao trabalho proposto.

Além da revisão na literatura, foram feitas análises que consistiram na execução do sistema em máquina local, mudando as condições para cada uma das três arquiteturas propostas conforme o teste a ser realizado. Dessa forma, para cada uma dessas três, foi iniciada a instalação do aplicativo em um dispositivo móvel com Android 13, configurada via cabo USB. Para a segunda e a terceira arquitetura, apenas a instalação da aplicação não foi suficiente, portanto, foi executado um servidor externo desenvolvido com *Node.js* e *Express* para suprir essa necessidade. Para a terceira arquitetura, além da configuração citada, para ser possível a visualização das informações no painel do serviço de mensageria, foi necessário configurar esse serviço de forma local. Após instalada a aplicação, executado o servidor e o serviço de mensageria (quando necessário), foram feitos os testes conforme os critérios de comparação selecionados pela ISO/IEC 25010.

As arquiteturas foram denominadas conforme as tecnologias empregadas: *FIREBASE* para a primeira, *FIREBASE-MYSQL* para a segunda e *FIREBASE-MSG* para a terceira. Em relação aos testes, segundo os critérios de comparação selecionados, foram feitos para as três arquiteturas de forma equivalente, com base nas seguintes operações:

- Para a análise de modificabilidade, foram feitas requisições no formato HTTP e inserido um novo campo no banco de dados. Por conseguinte, foi analisado o código-fonte para compreender os passos necessários para essa modificação;
- Para a análise de tolerância a falhas, a internet do usuário autor deste trabalho

foi desconectada e realizada uma requisição HTTP visando cadastrar um usuário. Após isso, foi observado se os dados da requisição foram inseridos no banco de dados (para a primeira e segunda arquitetura) ou na fila do *RabbitMQ* (para a terceira arquitetura), quando retomada a conexão;

- Para a análise de confidencialidade, foi realizada a tentativa de obter dados de outros usuários por meio de requisições HTTP, via API do *Cloud Firestore* ou servidor.

4 Aplicação Desenvolvida

4.1 Interface de usuário

Utilizando o *framework* de desenvolvimento de aplicativos móveis *React Native*, juntamente com a linguagem de programação *JavaScript*, foi desenvolvida em um período de iniciação científica uma aplicação para dispositivos móveis, a priori denominada por *Caronas Universitárias* (FIDELIS, 2023). Essa aplicação, possui o objetivo de gerenciar caronas na mesma cidade para universitários e sendo assim, foram criadas duas categorias de usuários, os motoristas (os quais oferecem caronas) e caronistas (os quais solicitam carona). As principais telas da aplicação desenvolvida, são as de buscar e oferecer carona, as quais podem ser visualizadas pelas Figuras 1 e 2, respectivamente.

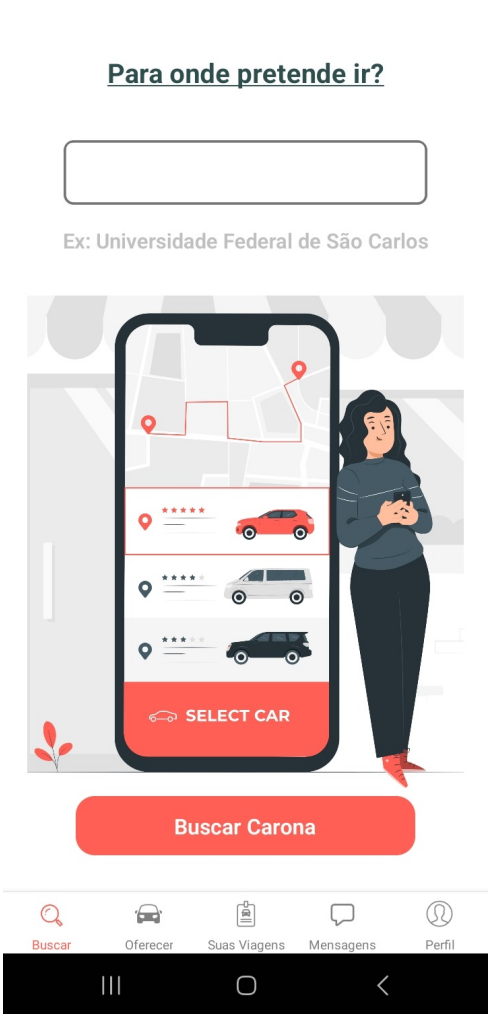
Na tela de buscar carona, exibida pela Figura 1, o fluxo de ações se inicia com o caronista preenchendo o seu destino, por exemplo, "Universidade Federal de São Carlos (UFSCar)" e acionando o botão "Buscar". Logo após isso, esse é redirecionado para outra tela, onde é exibida uma imagem animada enquanto a busca por carona é realizada. Por outro lado, por parte do motorista, a tela inicial para oferecer carona, deve ser preenchida com o destino do motorista e o número de vagas disponíveis em seu veículo, visualizada pela Figura 2.

Após devidamente preenchida a tela de oferecer carona, o motorista consegue visualizar em tempo real um mapa com os possíveis caronistas, esboçados por meio de um marcador clicável na cor vermelha e branca, exibido na Figura 3. Ao pressionar esse marcador, é mostrado um modal com o nome do usuário que está buscando por carona, a avaliação desse (de 0 a 5 estrelas), seu local de destino e a opção de oferecer carona ou não, por meio de um botão, visualizado na Figura 4.

Uma vez que a oferta de carona foi feita por parte do motorista, no *layout* do caronista, é possível fazer a escolha de um motorista, visto que o usuário caronista pode ter mais de uma oferta de carona. Nesse caso, o caronista teve apenas uma oferta de carona, visualizada pela Figura 5. Uma vez que a oferta foi selecionada, basta aceitar a carona e aguardar a viagem começar, exibida na Figura 6. Durante esse processo, as coordenadas do caronista e do motorista são escritas e atualizadas no banco de dados em tempo real (*Realtime Database*).

Após um caronista aceitar a proposta de carona de um motorista, o vermelho de seu marcador (do caronista) é atualizado para verde, indicando assim que um passageiro aceitou a sua carona. Logo após isso, há duas opções para o motorista, sendo essas a de oferecer mais caronas até completar o número de vagas ofertadas, ou buscar esse caronista

Figura 1 – Busca de carona



Fonte: Elaborada pelo autor

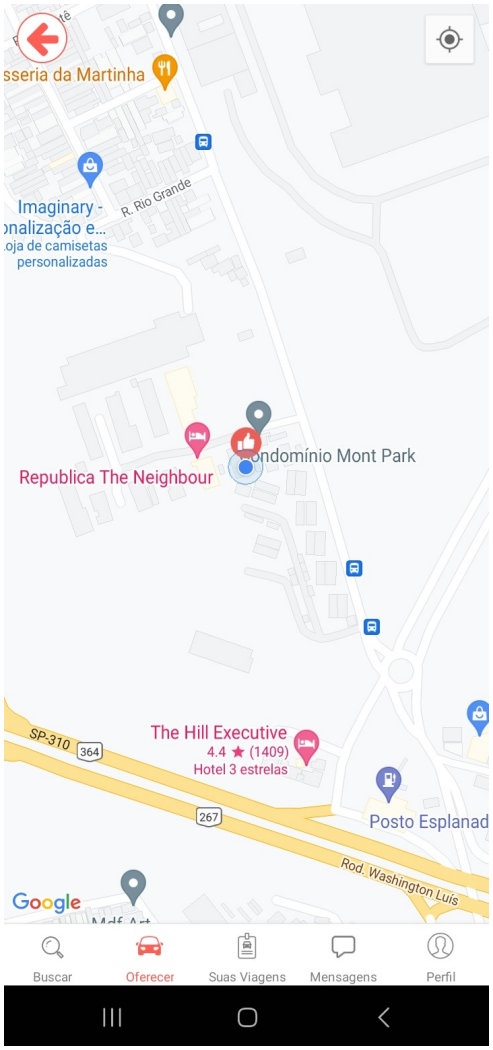
Figura 2 – Oferecer carona

Fonte: Elaborada pelo autor

que já aceitou a proposta de carona, pressionando o marcador já exibido no mapa. Uma vez que a busca por um caronista é iniciada, é possível se locomover até ele por meio de aplicativos de sistema de posicionamento global (GPS) externos (por exemplo, *Waze, Google Maps*, entre outros), visualizada pela Figura 7. Sendo assim, quando o motorista estiver próximo do passageiro, aparecerá o botão de "Embarcar Passageiro", o qual quando pressionado, faz de fato a viagem de ambos (motorista e caronista) iniciar, visualizada pela Figura 8. É necessário ressaltar que durante todo esse processo as coordenadas do motorista também são atualizadas em tempo real no banco de dados do *Firebase* (*Realtime Database*), para que o caronista consiga acompanhar a chegada do seu motorista até ele.

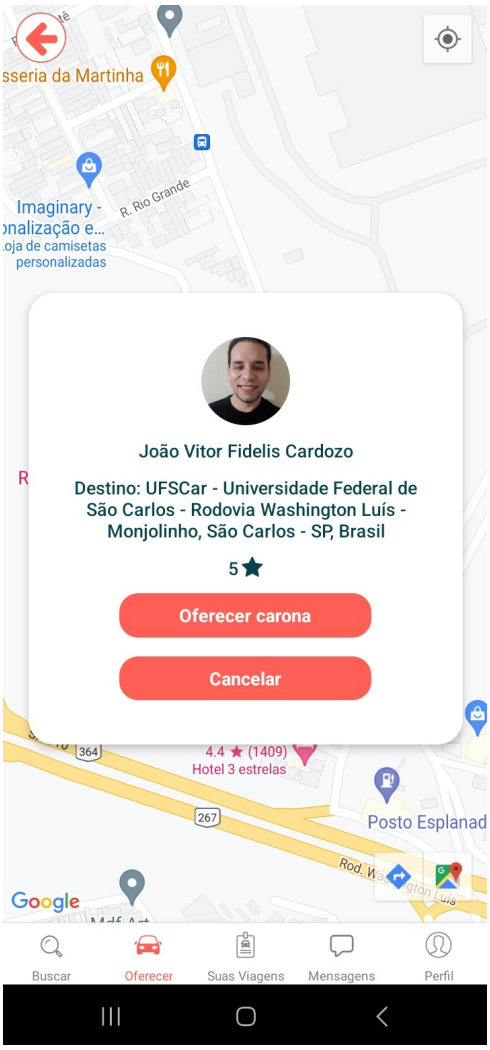
Para a implementação dessas funcionalidades, foi necessário utilizar os recursos de mapa, localização em tempo real e geolocalização reversa. Para isso, foram empregadas algumas bibliotecas do *React Native*, sendo essas "*react-native-maps*" e "*react-native-*

Figura 3 – Caronistas disponíveis



Fonte: Elaborada pelo autor

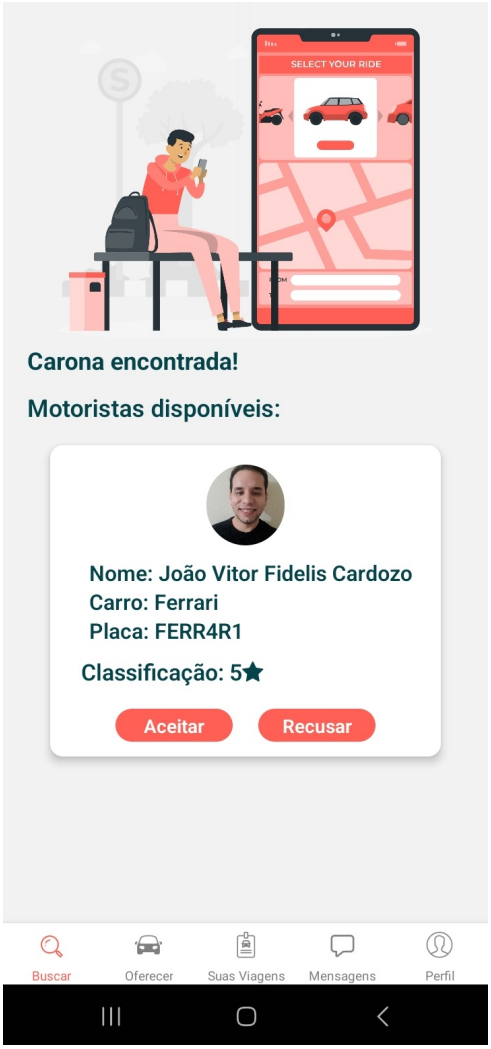
Figura 4 – Marcador pressionado



Fonte: Elaborada pelo autor

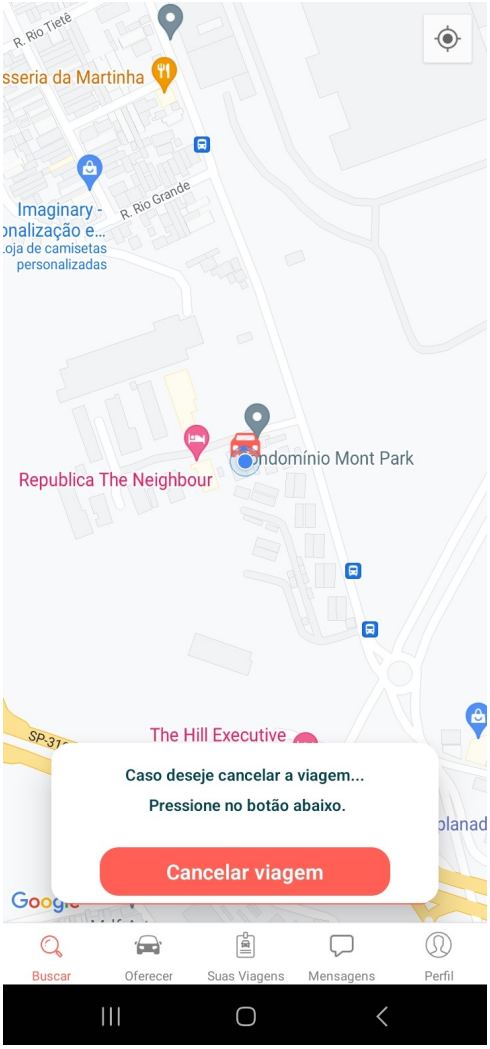
community/geolocation", havendo a necessidade de realizar o cadastro no *Google Cloud Plataform (GCP)* com o propósito de obter uma chave de API e conseguir usufruir desses serviços. Além disso, para armazenar o estado atual da aplicação, diferentemente do banco de dados do *Firebase*, foi utilizado um banco de dados relacional que se chama *SQLite database*, o qual armazena em cache local as informações necessárias. Ademais, para ambos usuários, foram armazenadas no banco de dados as informações pessoais de cadastro e de localização, sendo esta última, atualizada em tempo real. Além desses dados, foi projetado para que os usuários consigam interagir uns com os outros via *chatrooms*, nos quais as informações também foram salvas no banco de dados.

Figura 5 – Motoristas Disponíveis



Fonte: Elaborada pelo autor

Figura 6 – Aguardando motorista

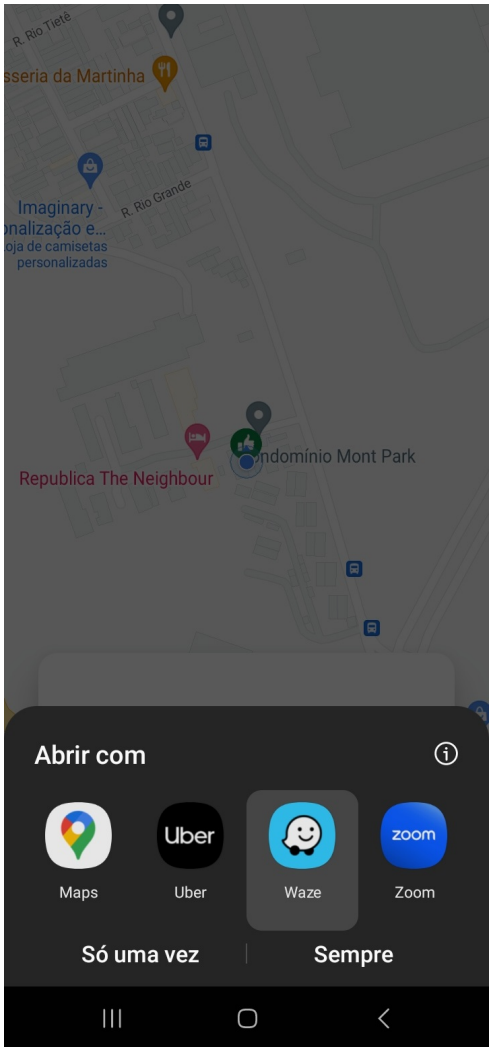


Fonte: Elaborada pelo autor

4.2 Estrutura geral

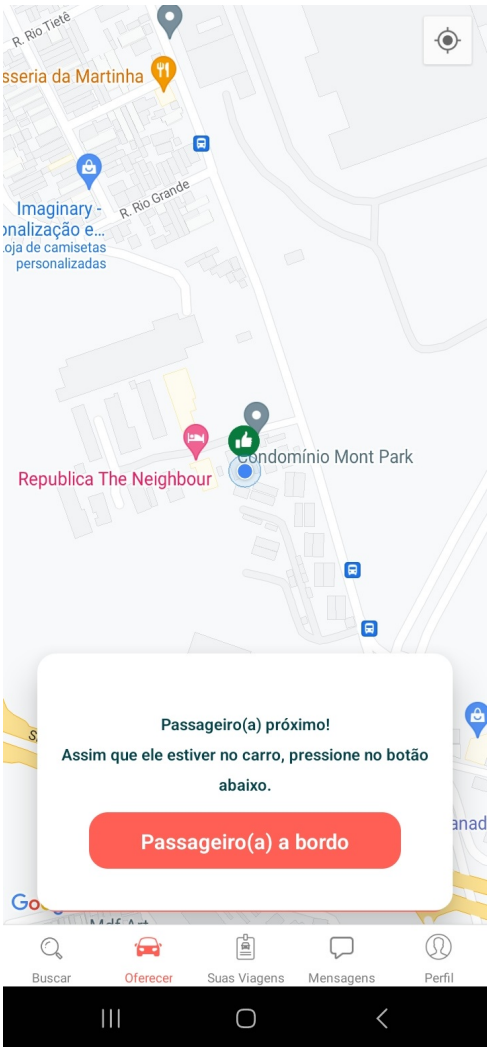
Nesta aplicação, foram construídas três versões equivalentes com arquiteturas de software distintas, as quais são detalhadas aqui neste estudo. O produto resultante para a primeira versão do aplicativo, foi a primeira arquitetura de software (denominada por *FIREBASE*) baseada em serviços de nuvem, com a utilização do *Google Firebase*. Para a escolha dessa arquitetura, uma etapa que envolve um conjunto de tecnologias e decisões fundamentais, foi decidido o emprego do *React Native* como framework de desenvolvimento e a linguagem de programação *JavaScript*, os quais foram os componentes centrais. Além disso, optou-se por utilizar um sistema de banco de dados não-relacional (NoSQL), dada a natureza da aplicação, a qual foi projetada para lidar uma vasta quantidade de informações. A solução selecionada para auxiliar nesta etapa, foi o *Google Firebase*, a qual se trata de uma plataforma de desenvolvimento de aplicativos oferecida pelo Google. Ela inclui serviços,

Figura 7 – Buscando caronista



Fonte: Elaborada pelo autor

Figura 8 – Caronista próximo



Fonte: Elaborada pelo autor

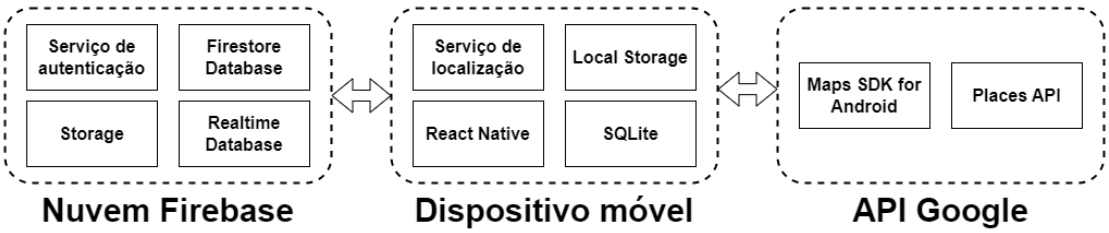
como autenticação de usuários, banco de dados e mensagens em tempo real, armazenamento em nuvem e hospedagem na *web*. Esses recursos permitem que os desenvolvedores criem aplicativos móveis e *web* escaláveis e eficientes, com autenticação segura, armazenamento de dados e notificações em tempo real. O *Firebase* é amplamente utilizado devido à sua facilidade de uso e integração com outras ferramentas do Google, tornando-o uma escolha popular para desenvolvedores que desejam criar aplicativos de alta qualidade de forma eficaz. Os serviços utilizados para a aplicação foram os de autenticação (por e-mail e via Google), *Firestore* (um banco de dados não-relacional estático), *Storage* (para o armazenamento de imagens) e *Realtime Database* (um banco de dados em tempo real).

Vale a pena ressaltar, que cada um dos serviços do *Firebase* desempenhou um papel único e fundamental na aplicação. O *Firestore Database*, por exemplo, foi usado para armazenar as coleções de usuários, contendo informações pessoais e histórico de viagens,

atualizando a coleção a cada viagem finalizada. O serviço de *Storage*, foi empregado para o armazenamento de imagens de perfil e veículos, ambos com identificadores únicos (*UIDs*). Por fim, o *Realtime Database* foi aplicado para armazenar dados relacionados ao estado, cidade e localização dos usuários em geral, incluindo coordenadas geográficas. Para construir esta última etapa, foi utilizado o conceito de geolocalização reversa, ou seja, a partir de uma coordenada base, encontrar o endereço da cidade do usuário ativo e logo após, o estado. É preciso dizer que, além dessas informações serem armazenadas no *Realtime Database*, os dados relacionados aos *chatrooms* também foram mantidos nesse banco de dados.

A construção do software para esta primeira arquitetura, foi feita diretamente no *front-end*, sem a necessidade de criação de servidores externos ou rotas, haja vista que o *Firebase* já proporciona todos os serviços necessários. Além disso, também foram essencias os recursos oferecidos pelo *Google Cloud Platform (GCP)*, especificamente suas Interfaces de Programação de Aplicações (APIs). Nesse contexto, foram utilizadas as APIs necessárias para localização do usuário no Android (*Maps SDK for Android*) e para listagem de locais próximos com base nos parâmetros enviados (*Places API*). Na Figura 9, é possível visualizar uma representação esquemática desta arquitetura.

Figura 9 – Primeira arquitetura de software (*FIREBASE*)

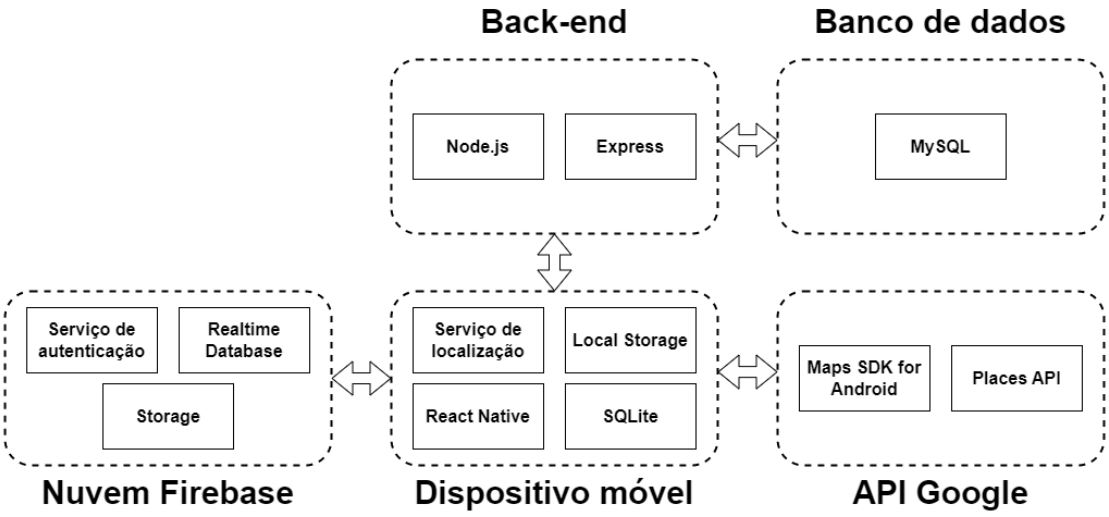


Fonte: Elaborada pelo autor

Para a segunda arquitetura de software (*FIREBASE-MYSQL*), houve a adaptação da estrutura inicial (primeira arquitetura desenvolvida), incorporando um *back-end* próprio implementado em *Node.js* e *Express*, mantendo para o *front-end*, o *Firebase Realtime Database*. Esta segunda abordagem, buscou uma alternativa ao serviço de banco de dados estático do *Google Firebase*, o *Firestore*. Além disso, como substituição para o *Firestore*, foi utilizado um banco de dados relacional, o *MySQL*, em conjunto com o *Sequelize*, o qual é um *Object-Relational Mapping* (ORM) para o *Node.js*. O *Sequelize* foi empregado nesta aplicação com o propósito de obter um nível maior de abstração durante as operações de banco de dados, além de facilitar a portabilidade. Além disso, é importante lembrar que o

MySQL foi acessado por meio de requisições no formato HTTP, utilizando rotas definidas no servidor. Na Figura 10 é possível visualizar a segunda arquitetura de software proposta.

Figura 10 – Segunda arquitetura de software (*FIREBASE-MYSQL*)

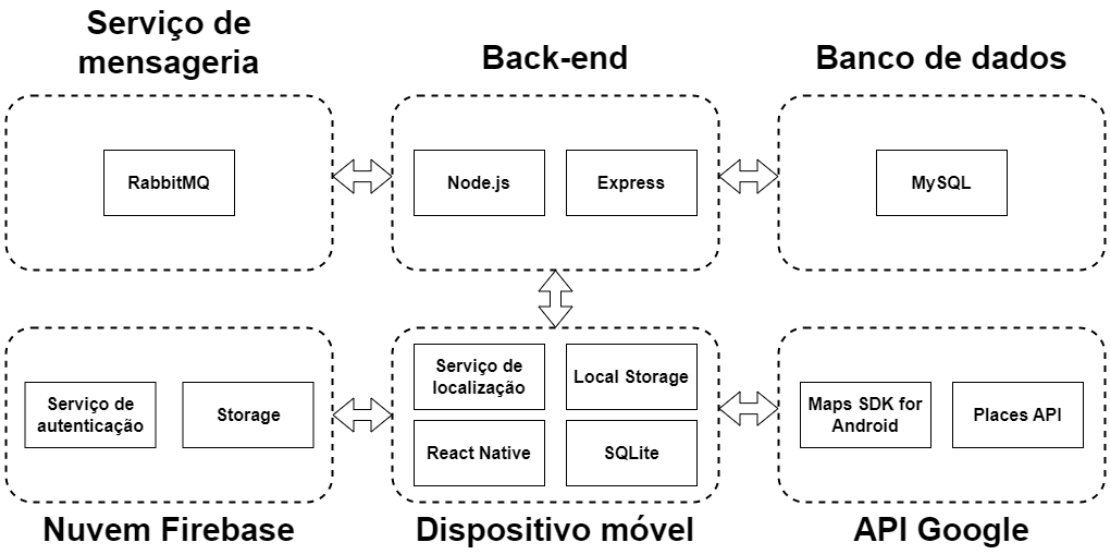


Fonte: Elaborada pelo autor

Para a terceira arquitetura de software (*FIREBASE-MSG*), foi mantida como base a segunda estrutura (segunda arquitetura desenvolvida), com o diferencial de ter incorporado ao sistema o *RabbitMQ*. Essa tecnologia, se trata de um serviço de mensageria que utiliza dos conceitos de produtores, consumidores, filas de mensagem e *exchange*. Os produtores ou consumidores, podem ser tanto os usuários motoristas, quanto caronistas, os quais podem executar as ações de enviar ou consumir das filas de mensagem, objetos com as informações necessárias para que a carona ou a oferta de carona aconteça, por exemplo, dados como cidade, estado, latitude, longitude, informações de destino, são essenciais para essas operações. Em acréscimo a isso, a *exchange* escolhe para qual fila de mensagem o dado será enviado, com base em uma chave de roteamento específica. Para isso, foram acrescentadas nesta arquitetura, rotas próprias que disparam essas ações previamente citadas. Além disso, esse serviço de mensageria, foi empregado em conjunto com um tipo de requisição diferente da HTTP, o *Server Sent Events (SSE)*, a qual foi utilizada para fornecer uma resposta contínua do *back-end* ao *front-end*, sem encerrar o canal de comunicação, com o propósito de substituir as funções nativas do *Firebase Realtime Database* utilizadas no sistema. Em contextos onde requisições HTTP são usadas, a resposta do *back-end* para o *front-end* é singular, no entanto, para o sistema objeto de estudo desta pesquisa, exigiu-se leitura e escrita constantes de coordenadas para todas as arquiteturas, implicando na necessidade de um canal de comunicação contínuo, assim como

ocorre quando há a utilização do *Firebase Realtime Database*. Além disto, é necessário dizer que requisições no formato HTTP também foram utilizadas nesta arquitetura em partes específicas do sistema, por exemplo, durante o cadastro de um usuário na aplicação. Para isso, por meio de uma requisição HTTP, é produzida uma mensagem em um produtor do *RabbitMQ*, a qual é consumida em outro momento durante o fluxo da aplicação. Após implementadas as rotas e devidamente configuradas, foi possível realizar a troca das funções do *Firebase Realtime Database* para o código implementado. É necessário ressaltar que foram substituídas todas as funções que envolvem o *Firebase Realtime Database*, por funções que utilizam apenas o *RabbitMQ* em conjunto com o *Node.js* e *Express*. Na Figura 11 é possível visualizar a terceira arquitetura de software proposta.

Figura 11 – Terceira arquitetura de software (*FIREBASE-MSG*)



Fonte: Elaborada pelo autor

5 Critérios de Comparação

A ISO/IEC 25010 (ISO/IEC 25010, 2011) é uma norma internacional que define parâmetros para se avaliar a qualidade de um software. Primeiramente, a qualidade de um software, é definida por essa norma como o grau em que o sistema satisfaz as necessidades declaradas e implícitas por todas as partes interessadas, de modo a proporcionar valor. Sendo assim, essas necessidades, são explícitas em totalidade pelas oito características exibidas na ISO. Essas características, se referem aos critérios de adequação funcional, eficiência e desempenho, compatibilidade, usabilidade, confiabilidade, segurança, manutenibilidade e portabilidade, cada uma, contendo uma definição diferente. Dentre esses critérios citados, para análise do produto desenvolvido, foram escolhidos os de manutenibilidade, confiabilidade e segurança, especificamente nos quesitos de modificabilidade, tolerância a falhas e confidencialidade, respectivamente. Na Figura 12, são mostrados todos os critérios da ISO/IEC 25010.

Figura 12 – Parâmetros para qualidade do produto de software

Fonte: ISO/IEC 25010 (2011)

No contexto da avaliação da qualidade de produtos de software, conforme exibido na Figura 12, é possível destacar que a manutenibilidade (*maintainability*) é uma característica principal de comparação para qualidade de um produto de software, mas, possui subcritérios de classificação. Nesse cenário, o subcritério escolhido para este trabalho foi o de modificabilidade, que conforme a ISO/IEC 25010, consegue ser definido como o grau em que um produto ou sistema pode ser modificado de forma eficaz e eficiente sem introduzir defeitos ou degradar a qualidade do produto existente. Para ser possível avaliar o critério de modificabilidade, para as três arquiteturas foi considerado o cenário de que houve a necessidade de manutenção na função de cadastrar novos usuários (*insertDataNewUser*), ou seja, além das informações já informadas, foi introduzido um campo de texto para que o usuário digite, durante o seu cadastro, o nome do curso no qual está matriculado.

Ademais, também é necessário ressaltar que confiabilidade (*reliability*) é definida como um critério principal. Entretanto, é imperativo considerar que, no mesmo escopo deste critério, existem diversos subcritérios que requerem uma análise minuciosa, sendo escolhido para este trabalho o subcritério de tolerância a falhas. Segundo a ISO/IEC 25010, o subcritério de tolerância a falhas é definido como o grau em que um sistema, produto ou componente opera conforme pretendido, apesar da presença de falhas de hardware ou software. Dessa forma, para tornar-se possível a simulação de uma falha, foi desconectada a Internet do usuário autor deste trabalho durante o procedimento de cadastro de passageiro e, analisado o comportamento do software enquanto ocorria a escrita das informações no banco de dados.

Finalmente, é imperativo evidenciar que a segurança (*security*) também constitui uma característica fundamental na avaliação da qualidade de produtos de software. De forma similar aos critérios principais anteriormente mencionados, a segurança também possui subcritérios de comparação. Neste estudo, o subcritério selecionado para análise foi a confidencialidade, o qual, segundo a definição da ISO/IEC 25010, pode ser escrito como o grau em que um produto ou sistema garante que os dados sejam acessíveis apenas àqueles autorizados a ter acesso. Portanto, para ser possível avaliar esse critério, foram considerados dois cenários, sendo esses:

- Na arquitetura *FIREBASE*, a tentativa de acesso a dados de outros usuários por meio da API do *Google Cloud Firestore*;
- Nas arquiteturas *FIREBASE-MYSQL* e *FIREBASE-MSG*, a tentativa de obter dados de outros usuários por meio de requisições HTTP, via servidor.

5.1 Modificabilidade

Manutenibilidade de software, exclusivamente no contexto de modificabilidade, ressalta uma característica considerada crítica que deve ser levada em consideração durante todo o processo de desenvolvimento de aplicações de software. Essa característica, como já evidenciado anteriormente, diz respeito à facilidade com que o software em questão pode ser modificado ou adaptado para situações específicas. No presente estudo, o contexto de análise para um subcritério característico foi estabelecido, visando comparar diversas arquiteturas de software implementadas no sistema em questão. Para tal, selecionou-se uma função comum às três arquiteturas analisadas, a qual foi fundamental para a análise de cada arquitetura nesse critério. No código exibido na Listagem 5.1, é possível observar um comentário formalizado por "//bloco try-catch", o qual resume operações equivalentes para as três arquiteturas. No bloco citado, para as três implementações do sistema, as operações equivalentes se resumem ao ato de cadastrar um usuário. Na primeira arquitetura, a implementação do bloco try-catch, possui o objetivo de inserir um usuário no banco de

dados *Firestore*. Na segunda arquitetura, o foco é inserir esse mesmo usuário no *MySQL*. Por fim, na terceira arquitetura, a qual possui o *RabbitMQ*, o objetivo é enviar para uma fila, os dados do usuário a ser cadastrado, os quais são consumidos em um próximo passo no fluxo na aplicação, cadastrando esse, no banco de dados *MySQL*.

Listing 5.1 – Código equivalente para as três arquiteturas

```
1   const insertDataNewUser = async () => {
2     if (
3       nome == '' ||
4       CPF == '' ||
5       data_nasc == '' ||
6       num_cel == '' ||
7       universidade == ''
8     ) {
9       setWarning('Preencha todos os campos!');
10      setModalVisible(true);
11    } else if (CPF.length != 14) {
12      setWarning('CPF incorreto.');
13      setModalVisible(true);
14    } else if (data_nasc.length != 10) {
15      setWarning('Data de nascimento incorreta.');
16      setModalVisible(true);
17    } else if (num_cel.length != 14) {
18      setWarning('Numero de celular incorreto.');
19      setModalVisible(true);
20    } else
21      //bloco try-catch
22  }
```

A função "*insertDataNewUser*" é uma função assíncrona escrita na linguagem de programação *JavaScript* e, em relação ao seu propósito, foi projetada para inserir dados de um novo usuário, seja motorista ou passageiro, em um banco de dados, por exemplo, o *Firestore* ou o *MySQL*. Para a arquitetura *FIREBASE*, os dados são inseridos no *Firestore*, o qual se trata de um banco de dados NoSQL hospedado na nuvem, dentro do serviço do *Google Firebase*. Além disso, é necessário dizer que, para as arquiteturas *FIREBASE-MYSQL* e *FIREBASE-MSG*, os dados são populados no banco de dados relacional *MySQL*. Portanto, uma vez que os usuários são cadastrados no sistema, sejam motoristas ou passageiros, eles obrigatoriamente irão preencher os dados essenciais da aplicação, que dizem respeito aos campos de nome, CPF, data de nascimento e universidade, dados comuns ao motorista e ao passageiro.

Para avaliar a primeira arquitetura de software desenvolvida, após introduzido o contexto, é exibida na Listagem 5.2 a implementação do bloco try-catch dentro da função "*insertDataNewUser*". O código dessa listagem, faz a inserção de um usuário no serviço *Firestore*. Entre as linhas 4 e 10, ocorre a inserção dos dados privados de um usuário e, entre as linhas 12 e 24, é feita a inserção dos dados públicos de um usuário. Finalmente, nas linhas 25 e 26, quando essas operações prévias já ocorreram, o usuário é redirecionado para a tela de buscar carona, com o modo passageiro. Um exemplo dessas informações populadas no *Firestore*, pode ser observada pela Figura 13.

Para avaliar a modificabilidade, conforme descrito na metodologia, analisa-se agora

Listing 5.2 – Cadastrar novo usuário (arquitetura 1)

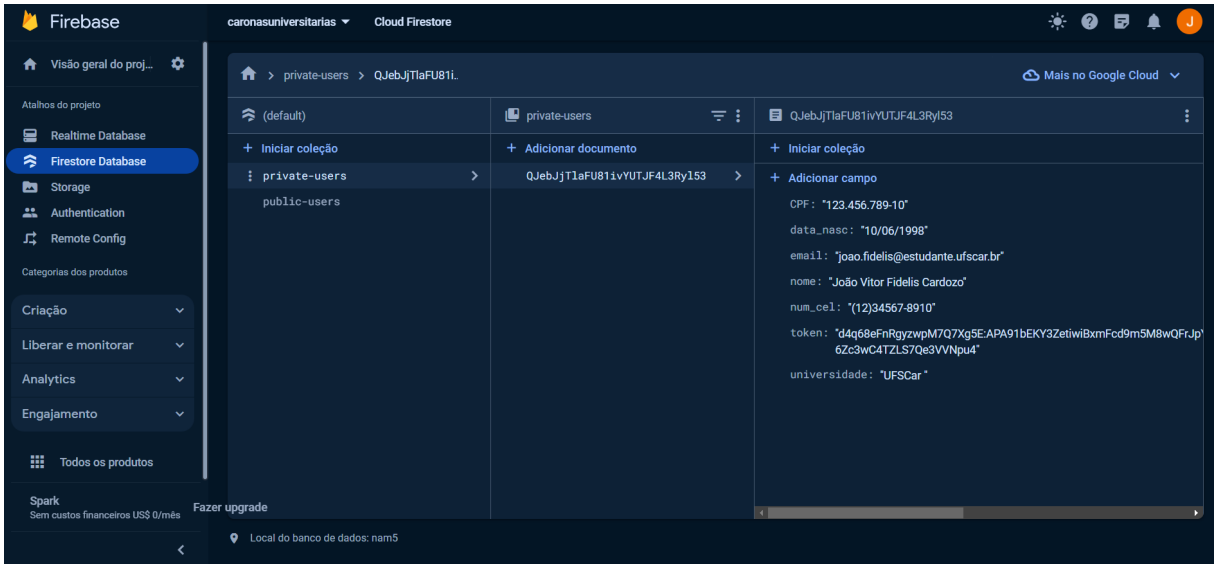
```
const insertDataNewUser = async () => {
    //codigo equivalente...
    try {
      await firestore().collection('private-users').doc(userID).set({
        nome: nome,
        CPF: CPF,
        data_nasc: data_nasc,
        num_cel: num_cel,
        universidade: universidade,
        email: route.params?.email,
      });
      await firestore()
        .collection('public-users')
        .doc(userID)
        .set({
          nome: nome,
          universidade: universidade,
          email: route.params?.email,
          placa_veiculo: '',
          ano_veiculo: '',
          cor_veiculo: '',
          nome_veiculo: '',
          motorista: false,
        })
        .then(() => {
          navigation.navigate('ModoPassageiro', {userID: userID});
        });
    } catch (error) {
      console.error('Erro inesperado:', error);
    }
  }
};
```

a inserção de um novo campo no cadastro: o curso em que o usuário está matriculado. Sendo assim, nesse caso, para adaptar a função "*insertDataNewUser*" a essa necessidade, é preciso apenas incluir novos pares chave-valor ao objeto enviado por parâmetro do método ".set". Na Listagem 5.3, é possível observar a função atualizada para inserir o campo "curso" no sistema.

Dessa forma, é concluído que para a primeira arquitetura, as vantagens de se utilizar os serviços do *Firebase*, em específico o *Firestore* (diretamente pelo *front-end* da aplicação), é a sintaxe, a qual em caso de necessidade de manutenção ou adição de novas informações, essas tarefas podem ser cumpridas apenas enviando a chave e o valor do novo dado a ser adicionado. Além disso, quando se utiliza o *Firebase*, o acesso ao banco de dados fica centralizado em um único lugar, nesse caso, diretamente no *front-end*, o que facilita a manutenção por parte do desenvolvedor.

Para avaliar a segunda arquitetura de software desenvolvida, é necessária uma melhor contextualização sobre o *back-end* desta arquitetura. Analisando-se a modificação prevista (inserção de um novo campo), nota-se que não basta apenas incluir novos pares chave-valor ao objeto a ser enviado, haja vista que esse segundo sistema é um pouco mais complexo. As mudanças do primeiro para o segundo sistema, implicam na adição de um *back-end*, ou seja, um servidor que foi desenvolvido com *Node.js* e *Express*, com rotas seguindo o padrão de arquitetura *Model, View* e *Controller* (MVC). Na parte do

Figura 13 – Coleção de usuários no Firestore



Fonte: ISO/IEC 25010 (2011)

Listing 5.3 – Cadastrar novo usuário (arquitetura 1) - Campo curso inserido

```
const insertDataNewUser = async () => {
    //codigo equivalente...
    try {
      await firestore().collection('private-users').doc(userID).set({
        nome: nome,
        CPF: CPF,
        data_nasc: data_nasc,
        num_cel: num_cel,
        universidade: universidade,
        curso: curso, //Linha adicionada
        email: route.params?.email,
      });
      await firestore()
        .collection('public-users')
        .doc(userID)
        .set({
          nome: nome,
          universidade: universidade,
          curso: curso, //Linha adicionada
          email: route.params?.email,
          placa_veiculo: '',
          ano_veiculo: '',
          cor_veiculo: '',
          nome_veiculo: '',
          motorista: false,
        })
        .then(() => {
          navigation.navigate('ModoPassageiro', {userID: userID});
        });
    } catch (error) {
      console.error('Erro inesperado:', error);
    }
  }
};
```

controller, tem-se várias rotas definidas, e entre elas, as rotas de "cadastrarUsuarioPublico" e "cadastrarUsuarioPrivado", as quais são utilizadas para solicitar as informações para

o *model*. Já no *model*, tem-se a definição das funções de "cadastrarUsuárioPublico" e "cadastrarUsuarioPrivado", as quais de fato possuem o acesso às tabelas do banco de dados que armazenam essas informações, em específico, as tabelas "*public_users*" e "*private_users*". As tabelas citadas podem ser visualizadas pelas Figuras 14 e 15 na descrição feita pelo console.

Figura 14 – Tabela de dados públicos de usuários - Arquitetura 2

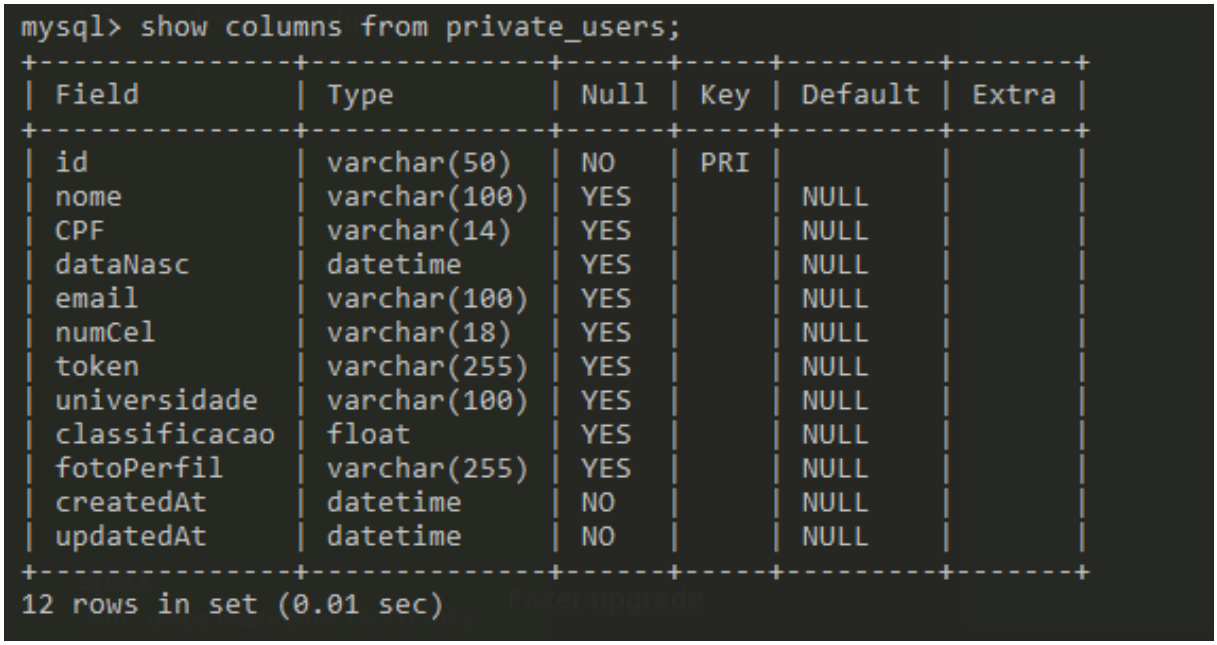
```
mysql> show columns from public_users;
+---------------+-------------+------+-----+---------+-------+
| Field         | Type        | Null | Key | Default | Extra |
+---------------+-------------+------+-----+---------+-------+
| id            | varchar(50) | NO   | PRI |         |       |
| nome          | varchar(100)| YES  |     | NULL    |       |
| email         | varchar(100)| YES  |     | NULL    |       |
| universidade  | varchar(100)| YES  |     | NULL    |       |
| classificacao | float       | YES  |     | NULL    |       |
| fotoPerfil    | varchar(255)| YES  |     | NULL    |       |
| motorista     | tinyint(1)  | YES  |     | NULL    |       |
| createdAt     | datetime    | NO   |     | NULL    |       |
| updatedAt     | datetime    | NO   |     | NULL    |       |
+---------------+-------------+------+-----+---------+-------+
9 rows in set (0.00 sec)
```

Fonte: Elaborada pelo autor

Após introduzido com mais detalhes o funcionamento da arquitetura *FIREBASE-MYSQL*, é necessário dizer que a modificação da função "*insertDataNewUser*" deve-se iniciar com um *migrate* da tabela, pois, como se pode notar pelas Figuras 14 e 15, o campo "curso" não está incluso nessas. Essa operação citada, consiste na adição do campo "curso" nas tabelas de usuário dentro do *MySQL*. Como o tratamento dos dados do banco de dados relacional *MySQL* foi feito utilizando o *Sequelize*, o procedimento considerado correto é realizar esse *migrate* seguindo a documentação dessa biblioteca. Após realizado o *migrate*, os dados referentes ao curso dos usuários já populados nas tabelas previamente comentadas, serão preenchidos com o valor "*NULL*", haja vista que essa informação ainda não foi atualizada. Para solucionar esse problema, o novo dado necessário pode ser solicitado em algum momento após o login na aplicação.

Uma vez que o *migrate* foi feito e o campo foi acrescentado nas tabelas, a manutenção na função "*insertDataNewUser*" pode ser de fato iniciada. Para ser possível adicionar o campo "curso" e persistir essa informação no banco de dados, existem dois passos que devem ser seguidos. O primeiro deles é a adição desse campo nas funções do *model* que solicitam e inserem esses dados no banco de dados, ou seja, as funções assíncronas "cadastrarUsuarioPublico" e "cadastrarUsuarioPrivado", ambas definidas no *back-end* do

Figura 15 – Tabela de dados privados de usuários - Arquitetura 2



```
mysql> show columns from private_users;
+---------------+--------------+------+-----+---------+-------+
| Field         | Type         | Null | Key | Default | Extra |
+---------------+--------------+------+-----+---------+-------+
| id            | varchar(50)  | NO   | PRI |         |       |
| nome          | varchar(100) | YES  |     | NULL    |       |
| CPF           | varchar(14)  | YES  |     | NULL    |       |
| dataNasc      | datetime     | YES  |     | NULL    |       |
| email         | varchar(100) | YES  |     | NULL    |       |
| numCel        | varchar(18)  | YES  |     | NULL    |       |
| token         | varchar(255) | YES  |     | NULL    |       |
| universidade  | varchar(100) | YES  |     | NULL    |       |
| classificacao | float        | YES  |     | NULL    |       |
| fotoPerfil    | varchar(255) | YES  |     | NULL    |       |
| createdAt     | datetime     | NO   |     | NULL    |       |
| updatedAt     | datetime     | NO   |     | NULL    |       |
+---------------+--------------+------+-----+---------+-------+
12 rows in set (0.01 sec)
```

Fonte: Elaborada pelo autor

sistema. Essas funções podem ser visualizadas pelas Listagens 5.4 e 5.5.

Listing 5.4 – Cadastrar usuário público - back-end (arquitetura 2)

```
1  async function cadastrarUsuarioPublico(req, res) {
2    try {
3      let reqs = await model.PublicUser.create({
4        id: req.body.id,
5        nome: req.body.nome,
6        email: req.body.email,
7        universidade: req.body.universidade,
8        curso: req.body.curso, //Linha adicionada
9        classificacao: req.body.classificacao,
10       fotoPerfil: req.body.fotoPerfil,
11       motorista: req.body.motorista,
12       createdAt: new Date(),
13       updatedAt: new Date(),
14     });
15     if (reqs) {
16       res.status(200).send(JSON.stringify('Usuario cadastrado com sucesso!'));
17     } else {
18       res.status(500).send(JSON.stringify('Falha ao cadastrar o usuario.'));
19     }
20   } catch (error) {
21     console.error('Erro ao cadastrar usuario:', error);
22     res.status(500).send(JSON.stringify('Erro interno ao cadastrar o usuario.'));
23   }
24 }
```

Na Listagem 5.4, na linha 3 é utilizado o modelo "*PublicUser*" criado pelo *Sequelize*, o qual faz referência a tabela "*public_users*" do *MySQL*. O método "*.create*" é empregado nesta etapa, para preparar uma *query* de inserção (*INSERT INTO*) no banco de dados, na qual os dados inseridos se referem às linhas 4 até 12. No intervalo das linhas 14 até 22,

Listing 5.5 – Cadastrar usuário privado - back-end (arquitetura 2)

```
1  async function cadastrarUsuarioPrivado(req, res) {
2    try {
3      let reqs = await model.PrivateUser.create({
4        id: req.body.id,
5        nome: req.body.nome,
6        CPF: req.body.CPF,
7        dataNasc: req.body.dataNasc,
8        email: req.body.email,
9        numCel: req.body.numCel,
10       token: req.body.token,
11       universidade: req.body.universidade,
12       curso: req.body.curso, //Linha adicionada
13       classificacao: req.body.classificacao,
14       fotoPerfil: req.body.fotoPerfil,
15       createdAt: new Date(),
16       updatedAt: new Date(),
17     });
18     if (reqs) {
19       res.status(200).send(JSON.stringify('Usuario cadastrado com sucesso!'));
20     } else {
21       res.status(500).send(JSON.stringify('Falha ao cadastrar o usuario.'));
22     }
23   } catch (error) {
24     console.error('Erro ao cadastrar usuario:', error);
25     res.status(500).send(JSON.stringify('Erro interno ao cadastrar o usuario.'));
26   }
27 }
```

é verificado se a inserção na tabela foi bem sucedida ou se houve algum problema. Além disso, na Listagem 5.5, a operação aplicada é a mesma, com a diferença de ser na tabela "*private_users*" e do modelo utilizado ser o "*PrivateUser*". Os dados inseridos nesta etapa, são descritos pelas linhas 4 até 15.

Sendo assim, o campo "curso", pode ser inserido em ambas funções embaixo do campo "universidade", mantendo o padrão "curso: req.body.curso", para evidenciar que esse dado será lido do corpo da solicitação HTTP e recebido diretamente pelo *front-end* via API. Uma vez que a mudança no *back-end* foi realizada, ou seja, o campo curso foi inserido na rota necessária, a requisição pode ser feita diretamente pelo *front-end*, utilizando para isso, o método "*fetch*", nativo do *JavaScript*. Além disso, na Listagem 5.6, é possível notar que as rotas são descritas por "cadastrarUsuarioPublico" e "cadastrarUsuarioPrivado", as quais quando acionadas, conseguem fazer referência às funções já comentadas. Sendo assim, o tipo de requisição feita nesta etapa, é do tipo "*POST*", a qual por padrão, recebe um objeto *JavaScript* como corpo da requisição, convertido para JSON previamente e essa. Os dados desse objeto, dependendo da função invocada, refletem o id do usuário, nome, data de nascimento, número de celular, universidade, email e todas as outras informações também necessárias e importantes já comentadas na arquitetura *FIREBASE*. Além disso, é necessário ressaltar que ao final dessa função, é verificada se a resposta da requisição foi bem-sucedida, ou seja, se retornou um *status* com código no formato "2xx", validado por "if (responsePublicUser.ok && responsePrivateUser.ok)". Caso o formato seja equivalente ao esperado, o usuário também é redirecionado para a tela de buscar carona, no modo passageiro, e caso contrário, é mostrada uma mensagem de erro para controle interno da

aplicação e possíveis tratamentos futuros.

Listing 5.6 – Cadastrar novo usuário (arquitetura 2)

```
const insertDataNewUser = async () => {
    //codigo equivalente...
    try {
      const responsePublicUser = await fetch(
        serverConfig.urlRootNode + 'cadastrarUsuarioPublico',
        {
          method: 'POST',
          headers: {
            Accept: 'application/json',
            'Content-type': 'application/json',
          },
          body: JSON.stringify({
            id: userID,
            nome: nome,
            universidade: universidade,
            curso: curso, //Linha adicionada
            email: route.params?.email,
            motorista: false,
          }),
        },
      );

      const responsePrivateUser = await fetch(
        serverConfig.urlRootNode + 'cadastrarUsuarioPrivado',
        {
          method: 'POST',
          headers: {
            Accept: 'application/json',
            'Content-type': 'application/json',
          },
          body: JSON.stringify({
            id: userID,
            nome: nome,
            CPF: CPF,
            data_nasc: data_nasc,
            num_cel: num_cel,
            universidade: universidade,
            curso: curso, //Linha adicionada
            email: route.params?.email,
          }),
        },
      );

      if (responsePublicUser.ok && responsePrivateUser.ok) {
        navigation.navigate('ModoPassageiro', {userID: userID});
      } else {
        console.error('Erro ao inserir:', response.statusText);
      }
    } catch (error) {
      console.error('Erro inesperado:', error);
    }
  }
};
```

Portanto, após comentados os detalhes de manutenção para a adição desse campo, na segunda arquitetura, notam-se algumas diferenças e necessidades. Primeiramente, a adição de um *back-end*, ou seja, neste caso, um servidor com *Node.js* e *Express*, cria-se uma complexidade maior para acrescentar esse campo, haja vista que essa solicitação não é resolvida apenas acrescentando uma linha de código no *front-end*. Sendo assim, as etapas envolvem realizar um *migrate* nas tabelas de usuário (*public_users* e *private_users*) do *MySQL*, para de fato, atualizá-las e ser possível incluir o campo "curso". Uma vez que

o campo curso foi inserido, foi preciso introduzir esse campo tanto na rota definida no *model* (que se comunica com o banco de dados), quanto na função "*insertDataNewUser*", no *front-end* da aplicação. Sendo assim, a manutenção nesse caso, torna-se um pouco mais complexa.

Para a arquitetura *FIREBASE-MSG*, também foi avaliada a necessidade de inclusão do campo "curso" no sistema. Porém, assim como foram introduzidos para a segunda arquitetura de software conceitos mais detalhados referentes à implementação do sistema, nesta arquitetura em questão, também é necessário esse estudo. Na Listagem 5.7, nas linhas 4 e 5, é feita uma requisição HTTP para a rota "api/rabbit/enviarInfo/cadastroUsuario". Nas linhas 12 até 24, é enviado no corpo da requisição os dados solicitados para o tratamento dessa no servidor. Nas linhas 29 até 44, é verificado se a requisição ocorreu com sucesso e caso sim, o usuário é levado para a tela de buscar carona, no modo passageiro. Caso não, é disparada uma mensagem de erro inesperado. Além disso, ressalta-se que quando essa requisição é feita, ocorre o envio dessa mensagem para uma fila do *RabbitMQ*, por meio de um produtor.

A requisição feita na Listagem 5.7, tem como ponto de referência do lado do servidor, o *endpoint* da Listagem 5.8, ou seja, é acessada a função da rota "api/rabbit/enviarInfo/cadastroUsuario", a qual aciona um produtor do *RabbitMQ*. Nesta última, nas linhas 2 a 5, é primeiro verificado se não existe um canal ou uma conexão aberta para o *RabbitMQ* atuar, retornando uma mensagem de erro no caso de afirmação dessa condição. Nas linhas 9 até 20, são obtidos os valores da requisição HTTP do tipo "*POST*". Nas linhas 22 até 35, é construído o objeto de informação que será enviado posteriormente para uma fila específica do *RabbitMQ*. Nas linhas 37 até 39, é definido a *exchange* para a comunicação das mensagens, chave de roteamento e nome da fila, respectivamente. A *exchange*, nesse caso, escolhe para qual fila enviar a mensagem com base em uma chave de roteamento específica, nessa ocasião, os dados são roteados para cadastrar usuários, pois a *exchange* foi definida como "cadastroUsuarios". Além disso, para escolher a fila em questão que essa informação será enviada, foi utilizado o valor booleano "motorista", que diz se um usuário naquele momento em específico está atuando como motorista ou como passageiro. Portanto, para este exemplo, o usuário está atuando como passageiro, haja vista que a chave motorista é definida como false no parâmetro da requisição "*POST*". Portanto, nas linhas 41 até 43, são utilizadas funções do *RabbitMQ* que garantem a existência da *exchange* e da fila "passageiros", vinculando a fila a *exchange*, utilizando para isso a chave de roteamento. Por fim, na linha 44, a mensagem com o objeto é publicada na fila.

Por outro lado, para o consumidor, na Listagem 5.9, as mensagens são consumidas da fila previamente citada (passageiros), buscando para isso, uma informação em específico. Analisando a Listagem que exibe o código do consumidor (5.9), afirma-se que até a linha 6, as operações são equivalentes as do produtor. Nas linhas 12 até 26, para cada mensagem

Listing 5.7 – Cadastrar novo usuário (arquitetura 3)

```
const insertDataNewUser = async () => {
    //codigo equivalente...
    try {
      let reqs = await fetch(
      `${serverConfig.urlRootNode}api/rabbit/enviarInfo/cadastroUsuario`,
        {
          method: 'POST',
          headers: {
            Accept: 'application/json',
            'Content-type': 'application/json',
          },
          body: JSON.stringify({
            id: userID,
            nome: nome,
            CPF: CPF,
            data_nasc: data_nasc,
            num_cel: num_cel,
            universidade: universidade,
            curso: curso, //Linha adicionada
            email: route.params?.email,
            placa_veiculo: '',
            ano_veiculo: '',
            cor_veiculo: '',
            nome_veiculo: '',
            motorista: false,
          }),
        },
      );

      if (reqs.ok) {
        await reqs.json().then(response => {
          navigation.navigate('ModoPassageiro', {userID: userID});
        });
      } else {
        console.error(
          'Erro ao enviar informacoes de cadastro:',
          reqs.statusText,
        );
      }
    } catch (error) {
      console.error(
        'Erro inesperado ao enviar informacoes de cadastro:',
        error,
      );
    }
  }
};
```

recebida na fila "passageiros", é testado se o id dessa mensagem corresponde ao id do usuário que está sendo cadastrado na aplicação. Caso seja, a mensagem é reconhecida por meio da tag "ack", além de ser devolvida ao *front-end*. Caso contrário, a mensagem é rejeitada por meio da tag "nack" e é devolvida ao *front-end* uma informação de mensagem não encontrada, permanacendo na fila, para que outro usuário consiga consumi-la.

Dessa forma, é necessário dizer que para incluir o campo "curso", é preciso passar pelas mesmas modificações da segunda arquitetura, com o acréscimo de ser necessário também, incluir esse campo no *back-end*, em específico nas rotas e na requisição HTTP do *RabbitMQ*. Sendo assim, para realizar a manutenção desta arquitetura, sempre é exigido um passo a mais do que a segunda arquitetura, haja vista a necessidade de manutenção do serviço de mensageria, além das etapas necessárias comentadas na arquitetura 2.

Listing 5.8 – Produtor do RabbitMQ (arquitetura 3)

```
router.post('/enviarInfo/cadastroUsuario', async (req, res) => {
  if (!conn || !ch) {
    return res
      .status(500)
      .send({status: 'Conexao com AMQP nao esta disponivel.'});
  }
  try {
    console.log('enviarInfo/cadastroUsuario');
    const id = req.body.id;
    const nome = req.body.nome;
    const CPF = req.body.CPF;
    const data_nasc = req.body.data_nasc;
    const num_cel = req.body.num_cel;
    const universidade = req.body.universidade;
    const curso = req.body.curso, //Linha adicionada
    const email = req.body.email;
    const placa_veiculo = req.body.placa_veiculo;
    const ano_veiculo = req.body.ano_veiculo;
    const cor_veiculo = req.body.cor_veiculo;
    const nome_veiculo = req.body.nome_veiculo;
    const motorista = req.body.motorista;

    const objInfo = {
      id: id,
      nome: nome,
      CPF: CPF,
      data_nasc: data_nasc,
      num_cel: num_cel,
      universidade: universidade,
      curso: curso, //Linha adicionada
      email: email,
      placa_veiculo: placa_veiculo,
      ano_veiculo: ano_veiculo,
      cor_veiculo: cor_veiculo,
      nome_veiculo: nome_veiculo,
      motorista: motorista,
    };

    const exchangeName = 'cadastroUsuarios';
    const routingKey = motorista ? 'motoristas' : 'passageiros';
    const queueName = routingKey;

    ch.assertExchange(exchangeName, 'direct', {durable: false});
    ch.assertQueue(queueName, {durable: false});
    ch.bindQueue(queueName, exchangeName, routingKey); //vincula uma fila (queue) a um
        exchange especifico, usando uma routing key
    ch.publish(exchangeName, routingKey, Buffer.from(JSON.stringify(objInfo)));
    res.status(200).send({status: 'Informacoes do usuario enviadas!'}); //envia ao
        servidor
  } catch (error) {
    console.error('Erro ao enviar informacoes do usuario:', error);
    res.status(500).send({status: 'Erro ao enviar informacoes do usuario.'});
  }
});
```

Por fim, é preciso dizer que para todas as arquiteturas, seria preciso, em uma atualização do sistema solicitar aos usuários que já estavam cadastrados preencherem o campo de "curso", para que esse possa ser atualizado nos bancos de dados, mantendo assim a consistência das informações.

Listing 5.9 – Consumidor do RabbitMQ (arquitetura 3)

```
router.get('/consumirInfo/cadastroUsuario/:id', async (req, res) => {
  if (!conn || !ch) {
    return res
      .status(500)
      .send({status: 'Conexao com AMQP nao esta disponivel.'});
  }
  try {
    console.log('consumirInfo/cadastroUsuario/:id');
    const idUsuario = req.params.id;
    let mensagemEncontrada = false;

    const onMessage = msg => {
      if (msg) {
        const mensagem = JSON.parse(msg.content.toString());
        if (mensagem.id === idUsuario) {
          mensagemEncontrada = true;
          ch.ack(msg);
          res.status(200).json({
            status: 'Mensagem consumida com sucesso!',
            data: mensagem,
          });
        } else {
          ch.nack(msg);
        }
      }
    };

    ch.consume('passageiros', onMessage, {noAck: false});

    setTimeout(() => {
      if (!mensagemEncontrada) {
        res.status(404).send({status: 'Mensagem nao encontrada.'});
      }
    }, 5000);
  } catch (error) {
    console.error('Erro ao consumir mensagem:', error);
    if (!res.headersSent) {
      res.status(500).json({status: 'Erro ao consumir mensagem.'});
    }
  }
});
```

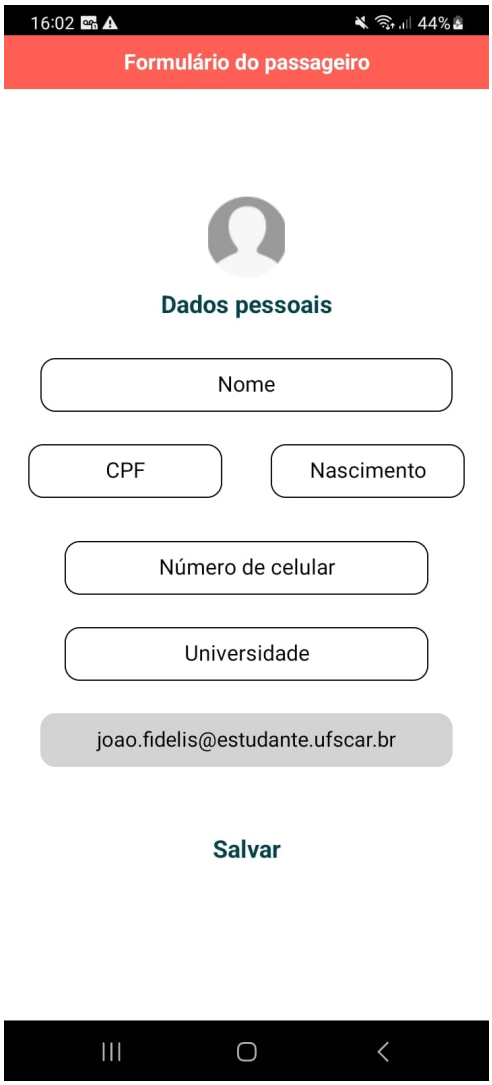
5.2 Tolerância a Falhas

Existem várias falhas, de fato inesperadas, que podem ocorrer durante o funcionamento de um software, isto é, a conexão com a internet pode falhar, a capacidade de processamento do hardware do dispositivo móvel onde o aplicativo está instalado pode apresentar falhas, entre outros possíveis imprevistos. O comportamento esperado citado na definição desse critério na ISO/IEC 25010, pode ser interpretado para o contexto desta pesquisa, como manter a consistência das informações, especificamente, no banco de dados.

Dado o contexto, para ser possível comparar as três arquiteturas nesse quesito, foi simulada uma única falha (dentre todas as possíveis). Foi forçado um problema de falha de conexão com a internet e observado se após retomada essa conexão, os dados que tentaram ser inseridos no banco de dados ou produzidos por um produtor no *RabbitMQ* e depois populados (no caso da terceira arquitetura), conseguem ser obtidos pelos usuários que supostamente deveriam receber essas informações. Dessa forma, foi escolhida uma função específica do sistema que cumpre o mesmo propósito, mas foi modificada para as três

diferentes arquiteturas. Sendo assim, a função escolhida a critério de comparação, também foi a função "*insertDataNewUser*". Essa função é invocada quando o botão "Salvar" da tela de "Formulário do passageiro" é pressionado, a qual pode ser visualizada pela Figura 16.

Figura 16 – Formulário do passageiro

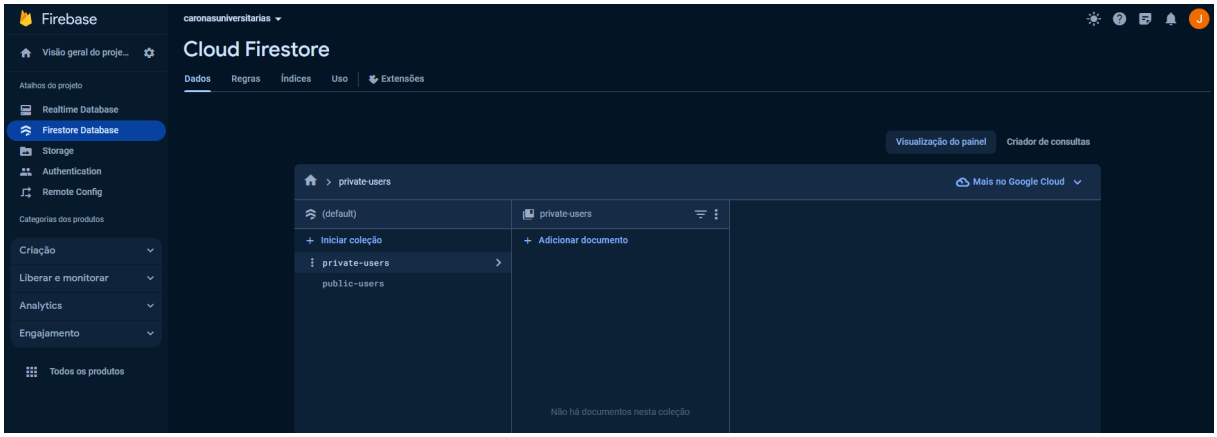


Fonte: Elaborada pelo autor

Para tornar-se viável realizar essa comparação, primeiramente, toda a base de dados relacionada ao usuário de e-mail "joao.fidelis@estudante.ufscar.br" no *Firestore* foi apagada, resultando na base de dados visualizada pela Figura 17.

Feito isso, a internet do dispositivo móvel do autor deste trabalho cujo aplicativo está instalado foi desabilitada, ou seja, foi removida a conexão com os dados móveis e com o Wi-Fi. Sendo assim, a tela de "Formulário do passageiro" foi preenchida com as informações do usuário, conforme se pode notar na Figura 18.

Figura 17 – Base de dados do Firestore limpa



Fonte: Elaborada pelo autor

Após o preenchimento das informações do usuário, o botão “Salvar” foi pressionado e a aplicação foi encerrada. O próximo passo foi restaurar a conexão com a internet e abrir novamente a aplicação e uma vez que isso foi feito, o usuário foi redirecionado diretamente para a tela principal do aplicativo, a qual foi exibida na Figura 1. A base de dados do *Firestore* foi consultada novamente, e é possível notar pela Figura 19, que os dados da requisição foram inseridos, mesmo havendo a falha de conexão com a internet previamente a essa requisição.

Para essa primeira arquitetura, como o *Firestore* está sendo utilizado diretamente do *front-end* da aplicação, após o experimento, é possível afirmar que o *Firestore* consegue lidar com a permanência de dados off-line. Segundo a documentação oficial do *Firebase* (Google Cloud, 2023) para esse banco de dados, para que esse recurso seja funcional, é armazenado em cache local uma cópia dos dados que o aplicativo estiver usando, nesse caso, os dados da requisição preenchidos na tela de formulário. Uma vez que a conexão é retomada, essas informações são enviadas ao *Firestore*, tornando novamente, consistente o estado da aplicação.

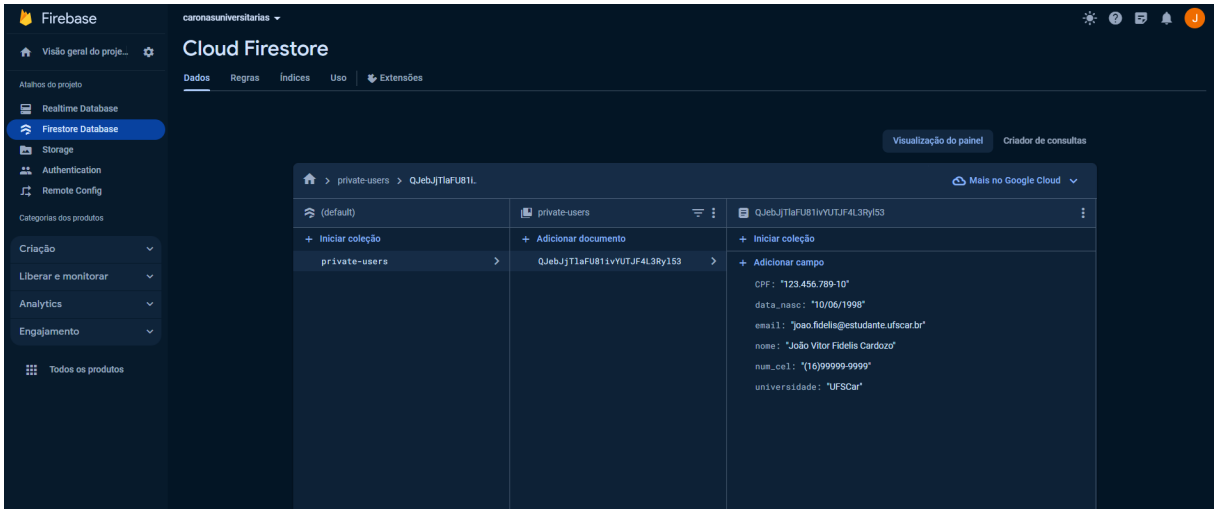
Para a análise da segunda arquitetura, é necessário lembrar que o *Firestore* não está sendo utilizado, mas sim o *MySQL*, acessado por meio de requisições HTTP de um servidor construído com *Node.js* e *Express*. Sendo assim, a base de dados relacionada ao usuário de e-mail “joao.fidelis@estudante.ufscar.br” também foi removida das tabelas “*public_users*” e “*private_users*”. Feito isso, foi seguido o mesmo procedimento para a arquitetura 1, ou seja, a internet foi desconectada, a tela de “Formulário do passageiro” foi preenchida com as mesmas informações, o botão “Salvar” foi pressionado e a aplicação foi encerrada. Após reestabelecida a conexão com a internet e reaberta a aplicação, o usuário não foi redirecionado para a tela principal do aplicativo.

Figura 18 – Formulário do passageiro preenchido

Fonte: Elaborada pelo autor

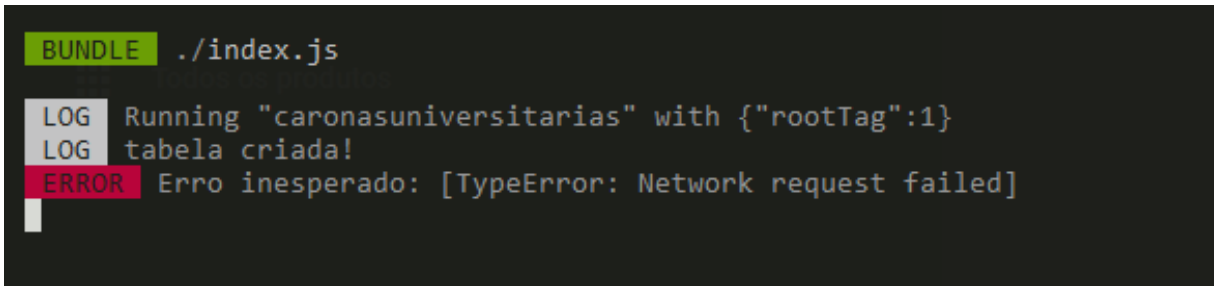
Depois de realizada uma análise minuciosa dos motivos que podem ter levado a esse comportamento, pode-se afirmar que há uma limitação nesta segunda arquitetura para trabalhar com um cenário offline, haja vista que as solicitações do cliente não conseguem chegar até o servidor, o que também aconteceu na primeira arquitetura antes da requisição (com a internet desconectada) e após a requisição (com a conexão reestabelecida). Dado esse contexto, como as solicitações não chegaram até o servidor após reestabelecida a conexão com a internet, seria necessário por parte do programador implementar um tratamento para essas requisições quando o cenário offline estiver presente, ou seja, armazenar essas requisições em cache local (por exemplo, com *AsyncStorage*) e quando o dispositivo tiver novamente a conexão com a internet, enviá-las ao servidor e, consequentemente, popular as informações no banco de dados. Para melhor análise das requisições, pode-se observar os logs no console para o *front-end* e *back-end* da aplicação, respectivamente.

Figura 19 – Dados inseridos no Firestore - Arquitetura 1



Fonte: Elaborada pelo autor

Figura 20 – Erro na requisição - Arquitetura 2



Fonte: Elaborada pelo autor

Analisando a Figura 20, é possível observar o log "Erro inesperado: [TypeError: Network request failed]", o qual indica que houve alguma falha relacionada a internet, nesse caso, forçada durante uma requisição HTTP. Por outro lado, na Figura 21, pode-se analisar que a única requisição que o servidor recebeu foi a de um "*select*" na tabela "*public_users*", realizada ao iniciar a aplicação para checar se o usuário em questão possui ou não cadastro no sistema. Nesse caso, o usuário não possui, por isso foi preenchida a tela de "Formulário do passageiro". Dessa forma, para essa segunda arquitetura, pode-se concluir que não há um tratamento automático por parte do servidor e do banco de dados utilizado para cenários offline, ou seja, as requisições feitas não são armazenadas, ficando a cargo do programador implementá-las em código diretamente.

Por fim, para a terceira arquitetura, ressalta-se novamente que se mantiveram as funcionalidades da segunda arquitetura como base, acrescidas do serviço de mensageria *RabbitMQ*. Para ser possível realizar a comparação, foram repetidas as mesmas etapas

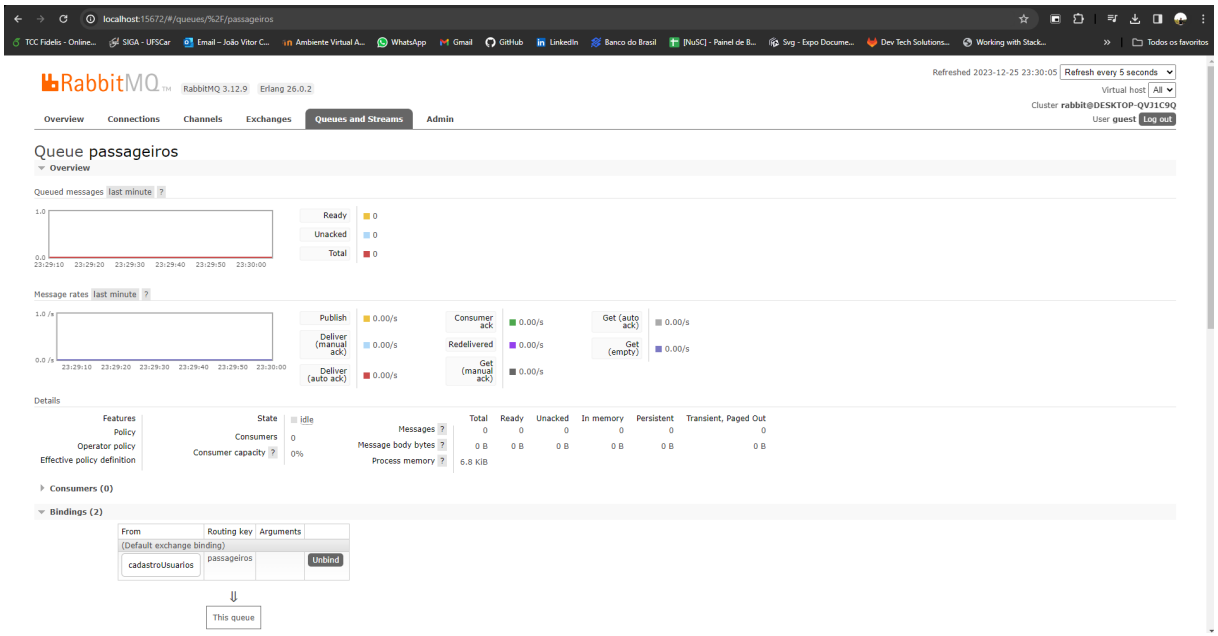
Figura 21 – Logs do servidor - Arquitetura 2

```
C:\Users\Cardozo\Desktop\React Native\caronasuniversitarias\src\services\node-server (firebase-nodejs_TCC -> origin)
λ nodemon server.js
[nodemon] 2.0.22
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Servidor Rodando
PORT: 8000
Executing (default): SELECT 1+1 AS result
Executing (default): SELECT `id`, `nome`, `email`, `universidade`, `classificacao`, `fotoPerfil`, `motorista`, `creat
edAt`, `updatedAt` FROM `public_users` AS `PublicUser` WHERE `PublicUser`.`email` = 'joao.fidelis@estudante.ufscar.br
';
```

Fonte: Elaborada pelo autor

da segunda arquitetura. Sendo assim, após reestabelecida a conexão com a internet e reaberta novamente a aplicação, notou-se que o usuário também não foi redirecionado para a tela principal do aplicativo. O primeiro passo foi checar o painel do *RabbitMQ*, para ver se havia alguma mensagem em espera para ser consumida na fila denominada por "passageiros". O painel desse serviço pode ser observado pela Figura 22.

Figura 22 – Painel RabbitMQ - Arquitetura 3

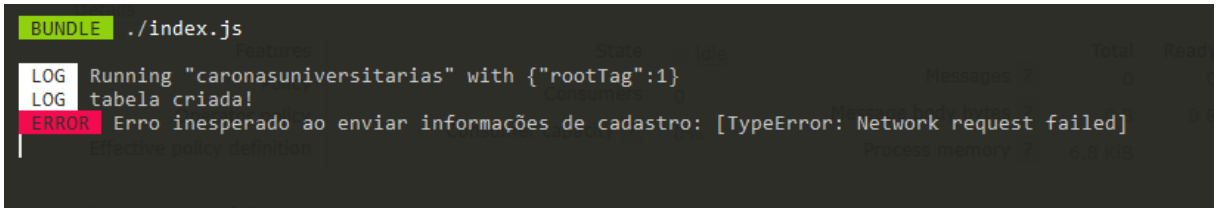


Fonte: Elaborada pelo autor

Analisando o painel exposto na Figura 22, nota-se que não existem mensagens pendentes para serem consumidas na fila "passageiros" do *RabbitMQ* e isso indica novamente, que a solicitação do cliente não chegou ao servidor e consequentemente, não passou pela rota do *RabbitMQ*, pelos mesmos motivos já expostos na arquitetura *FIREBASE-MYSQL*.

As Figuras 23 e 24, também mostram os logs do *front-end* e *back-end* para a terceira arquitetura, respectivamente.

Figura 23 – Erro na requisição - Arquitetura 3



Fonte: Elaborada pelo autor

Figura 24 – Logs do servidor - Arquitetura 3

Fonte: Elaborada pelo autor

Dessa forma, a conclusão é similar à da segunda arquitetura, ou seja, a terceira arquitetura não possui mecanismos automáticos para lidar com solicitações offline e trabalhar nesse cenário. Fica a cargo do programador mais uma vez, implementar esse tratamento em código, armazenando as requisições em cache local e observando quando a conexão com a internet é reestabelecida. Uma vez que há conexão novamente, as solicitações podem ser enviadas ao servidor.

Por fim, é necessário ressaltar que para as arquiteturas *FIREBASE-MYSQL* e *FIREBASE-MSG*, se a conexão fosse interrompida após a solicitação do cliente ter sido enviada ao servidor, as informações seriam populadas no banco de dados *MySQL*. Para a terceira arquitetura, em específico, a mensagem com os dados do usuário a ser cadastrado, ficaria pendente na fila "passageiros" do *RabbitMQ*, e seria consumida e populada no banco de dados quando o usuário acessasse novamente a aplicação, dessa vez, com conexão à internet.

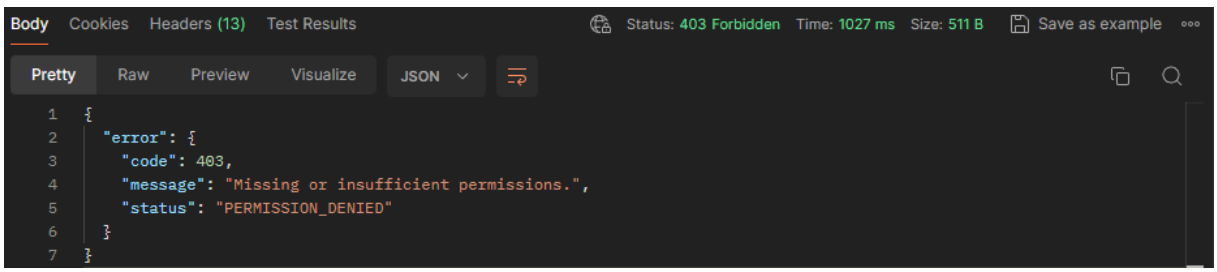
5.3 Confidencialidade

No âmbito da norma ISO/IEC 25010, o conceito de segurança de software é diversificado em várias subcaracterísticas, sendo uma delas, a confidencialidade. Conforme a ISO referenciada, essa subcaracterística se refere ao grau em que um produto ou sistema garante que os dados sejam acessíveis apenas aos usuários autorizados. Este aspecto deve ser considerado fundamental nos sistemas de software, pois garante que o acesso à informação seja restrito somente aos indivíduos que possuem essa devida permissão, mantendo assim, a privacidade e a integridade dos dados.

Após a introdução do contexto, iniciou-se uma análise comparativa entre as três arquiteturas de software distintas. Para isso, foram implementadas simulações de tentativas de acesso a dados de usuários por outros usuários anônimos ou não autorizados. O método empregado, conforme mencionado anteriormente, consistiu na realização de requisições HTTP direcionadas ao banco de dados por meio de rotas do servidor, com o propósito de obter de forma não autorizada, essas informações. Esse procedimento foi adotado, para mensurar para cada arquitetura a eficácia das medidas de segurança, em específico, a subcaracterística confidencialidade.

Para a primeira arquitetura, o teste necessário para avaliação foi feito sob condições mais complexas. Como o *Firebase* fornece um *back-end* e banco de dados como serviço, as requisições ao banco de dados são realizadas via HTTP, mas não com o *Node.js* atuando como uma camada de *back-end* com rotas definidas, e sim por funções nativas do *Firebase*, por meio de bibliotecas do *React Native*. Sendo assim, para simular requisições HTTP como nas demais arquiteturas, empregou-se a API REST do *Cloud Firestore* (Google Firebase Team, 2024a), com auxílio do *Postman*, uma plataforma de teste de APIs. Sendo assim, primeiramente, por meio do *Postman*, foi configurado para a primeira arquitetura uma requisição HTTP do tipo "*GET*" para obter os dados no *Firestore* de todos os usuários sem realizar autenticação. O resultado obtido é apresentado na Figura 25.

Figura 25 – Requisição GET sem autenticação - Arquitetura 1

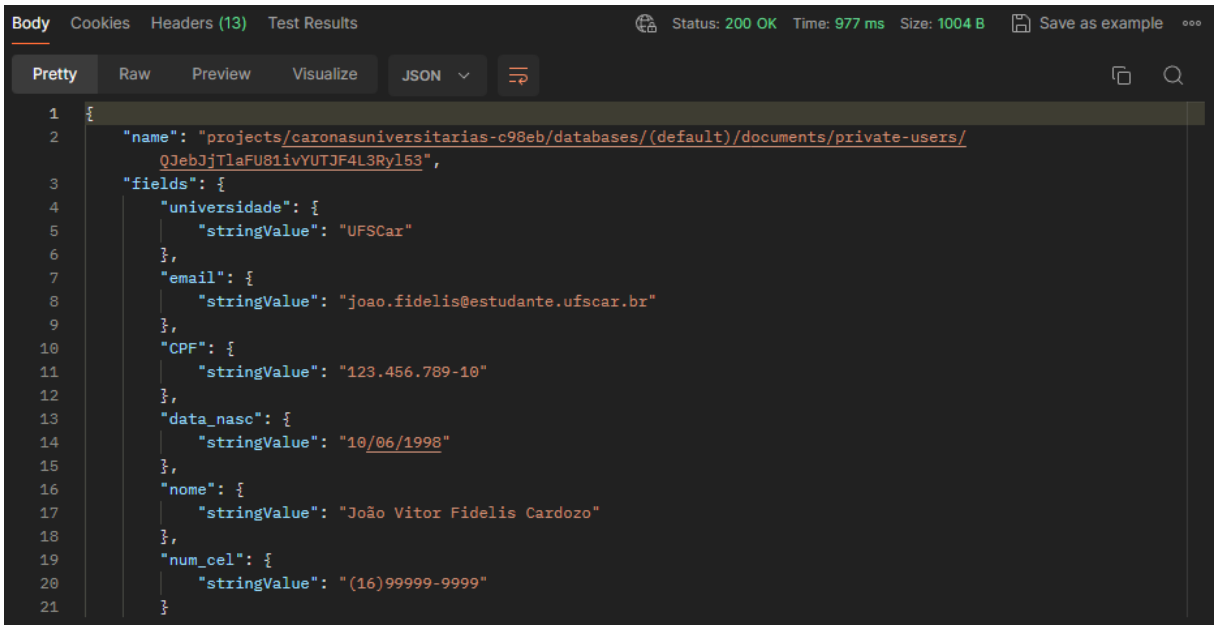


Fonte: Elaborada pelo autor

Após a análise da figura em questão, é possível notar que a resposta obtida para o

teste foi "*Missing or insufficient permissions*" e "*PERMISSION_DENIED*", ou seja, revela que a tentativa de acesso aos dados dos usuários sem autenticação não foi possível. Sendo assim, uma vez que esse teste foi finalizado, foi realizado um novo na mesma arquitetura, só que com um usuário autenticado. Portanto, o segundo teste realizado consistiu na tentativa de obter os dados do mesmo usuário que está fazendo a requisição. A autenticação foi realizada utilizando um token de acesso gerado no código-fonte da aplicação, o qual foi inserido no *Postman* como "*Bearer token*". O resultado obtido pode ser observado na Figura 26.

Figura 26 – Requisição GET com autenticação - Dados do meu usuário - Arquitetura 1

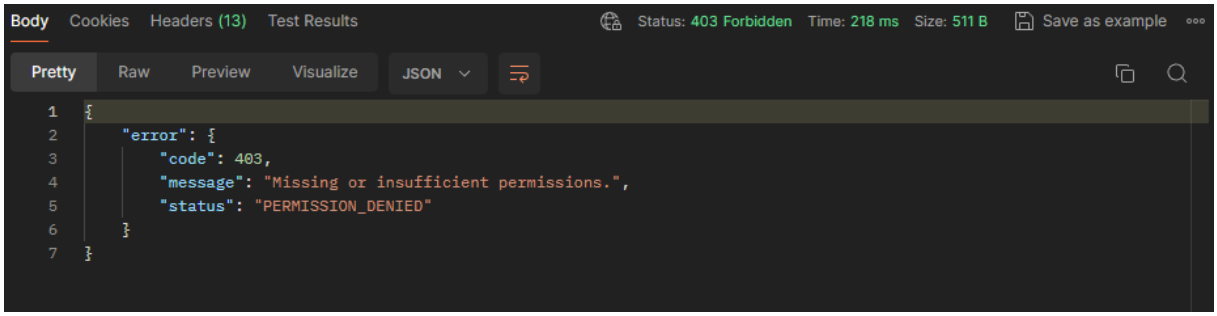


Fonte: Elaborada pelo autor

Com o resultado obtido na Figura 26, conclui-se que os dados foram recebidos, assim como esperado. Por fim, o último teste realizado para a arquitetura *FIREBASE*, corresponde à tentativa de obtenção de dados de um usuário diferente do usuário que está autenticado. O resultado da requisição é expresso na Figura 27.

Dessa forma, após análise da resposta obtida, é possível notar que essa foi similar à do primeiro teste, com o resultado de "*Missing or insufficient permissions*" e "*PERMISSION_DENIED*". Isso comprova a segurança e confidencialidade para essa arquitetura que possui o acesso ao banco de dados implementado diretamente pelo *front-end* com as bibliotecas do *Firebase*. É necessário ressaltar que para ser viável realizar esses testes, foi fundamental a chave de API do sistema desenvolvido, obtida por meio do console do *Firebase*. Além disso, para autenticação, é preciso possuir um token de autenticação para cada usuário, o qual muda a cada vez que a aplicação é instalada. Sendo assim, o acesso às informações torna-se mais difícil, haja vista que esses dados só podem ser

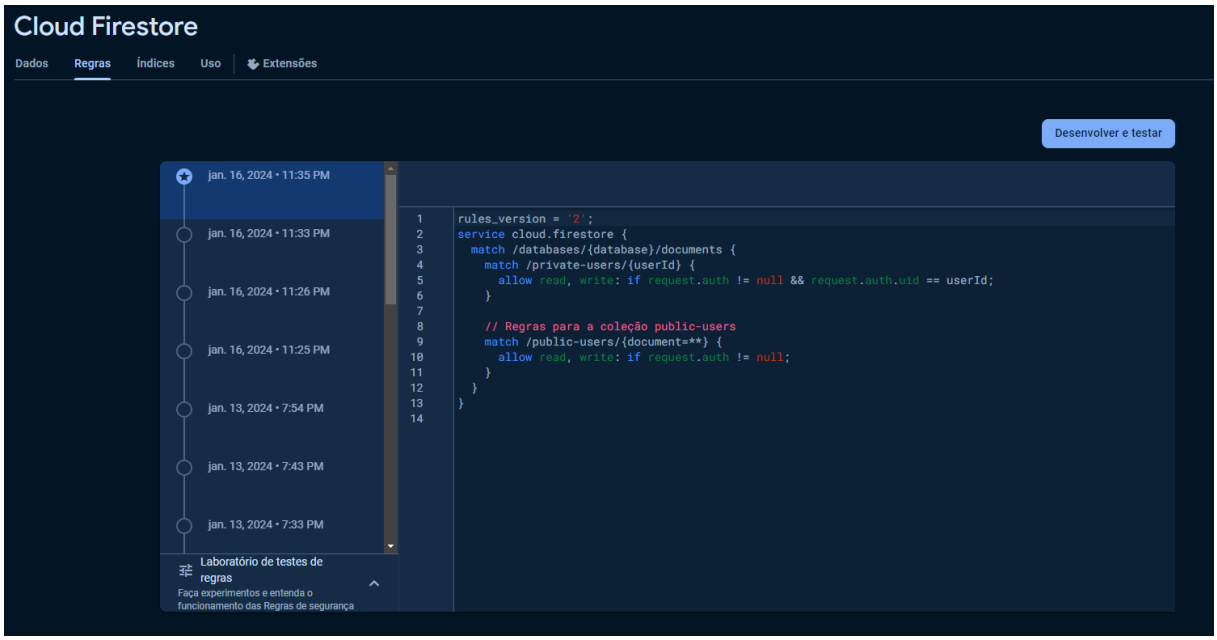
Figura 27 – Requisição GET com autenticação - Dados de outro usuário - Arquitetura 1



Fonte: Elaborada pelo autor

alcançados por meio dos proprietários do sistema. Caso seja possível a obtenção desses dados, ainda existe o fator de que um usuário não consegue acessar as informações de outros usuários, dentro dessa coleção do banco de dados. Essas regras de segurança que garantem a confidencialidade dos dados, foram implementadas diretamente na aba de regras do *Cloud Firestore*, de forma simplificada por meio do console. As regras são exibidas na Figura 28.

Figura 28 – Requisição GET com autenticação - Dados de outro usuário - Arquitetura 1

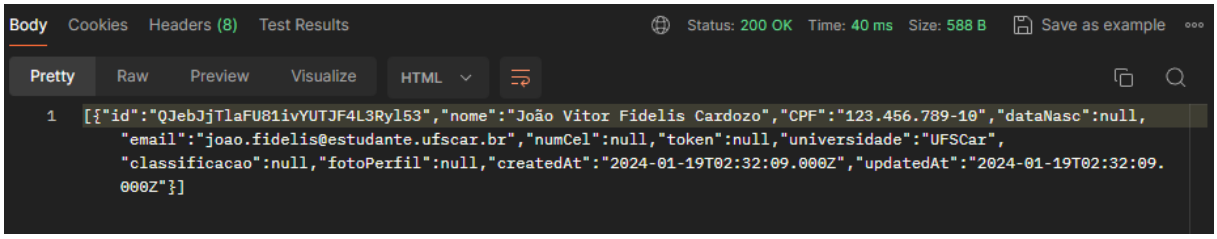


Fonte: Elaborada pelo autor

Nos testes feitos para as arquiteturas *FIREBASE-MYSQL* e *FIREBASE-MSG*, empregou-se uma metodologia semelhante à da primeira. Contudo, é importante destacar que, nestas arquiteturas, o acesso ao banco de dados é realizado por meio de rotas definidas,

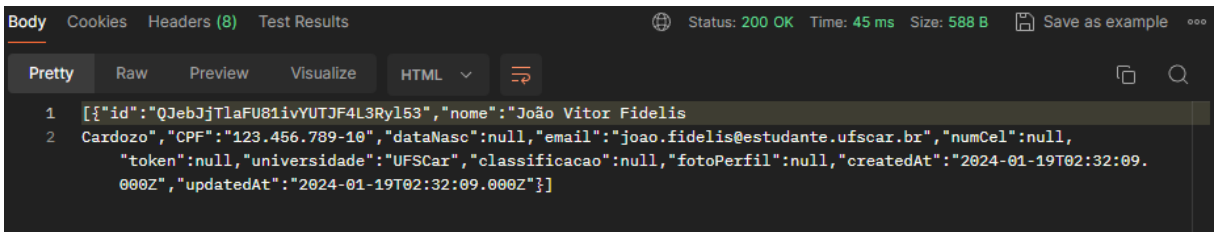
utilizando um servidor baseado em *Node.js* e *Express*. Sendo assim, as requisições foram feitas à rota "http://localhost:8000/buscarPorEmailUsuarioPrivado/:email", a qual retorna todos os dados de um usuário a partir do seu endereço de e-mail. É importante ressaltar que os resultados obtidos por meio dessa requisição estão populados na tabela "*private_users*" do *MySQL*. As respostas dessas requisições são exibidas pelas Figuras 29 e 30.

Figura 29 – Requisição GET - Dados de qualquer usuário - Arquitetura 2



Fonte: Elaborada pelo autor

Figura 30 – Requisição GET - Dados de qualquer usuário - Arquitetura 3



Fonte: Elaborada pelo autor

Após análise dos resultados obtidos, pode-se concluir que foi possível, tanto para a segunda quanto para a terceira arquitetura, a obtenção dos dados dos usuários com base em seu endereço de e-mail, sem ser necessária a autenticação. Sendo assim, pode-se afirmar que quando o acesso ao banco de dados é realizado por meio de rotas definidas, utilizando servidores construídos, vale a preocupação de garantir que um usuário esteja autenticado antes de fazer essa requisição, além de garantir a confidencialidade dos dados, verificando se as informações que esse usuário está tentando buscar pertencem a ele mesmo. Mas, assim como na primeira arquitetura, para que esse teste seja viável, o indivíduo que o está fazendo teria que ter o conhecimento sobre o endereço da rota, assim como os parâmetros necessários para a requisição.

5.4 Outros critérios

No decorrer deste estudo, a análise da aplicação construída foi realizada com base em três critérios centrais, definidos na ISO/IEC 25010. Entretanto, é crucial reconhecer a

existência de outros fatores que, embora não tenham sido analisados com a mesma rigidez sistemática, foram percebidos durante os estudos das três arquiteturas. Esses critérios, se referem aos benefícios da utilização de um *back-end* próprio e, do uso de um serviço de mensageria em um sistema de software.

5.4.1 Utilização de back-end próprio

A utilização de um *back-end* próprio, ao invés de serviços de nuvem, por exemplo, o *Google Firebase*, pode oferecer várias vantagens conforme as necessidades específicas do projeto. Para o projeto objeto de estudo desta pesquisa, o *back-end* foi construído com *Node.js* e *Express*, e por ser próprio, entre todos os fatores de benefícios, destacam-se os de controle total sobre a infraestrutura e o código, além de fornecer uma otimização de custos a longo prazo.

Com um *back-end* próprio, existe a possibilidade de melhor tratamento de erros durante a requisição, além de lidar com as respostas do servidor para o *front-end* de forma personalizada. Portanto, é necessário destacar que existe o controle completo sobre a arquitetura, as tecnologias utilizadas e a lógica de negócios. Além disso, pelas regras de segurança do *Firestore* do *Google Firebase*, não é possível que um usuário tenha acesso apenas algumas informações de uma coleção do banco de dados, por exemplo, e-mail, nome e classificação, o que deixa evidente a necessidade de um banco de dados público com informações não sensíveis e outro privado, com informações sensíveis dos usuários, por exemplo, o CPF. Na aplicação desenvolvida, um usuário deve ter acesso às informações não sensíveis de outros usuários, como o nome e classificação mas, como comentado, as regras de segurança não proporcionam essa restrição, ou seja, um usuário caso tenha acesso às chaves de API e de autenticação (*Bearer token*), conseguirá acessar por meio da API do *Firestore*, todos ou nenhum dado de uma coleção, caso a aplicação seja projetada para essa questão. Para que isso não aconteça, foi necessária a criação de duas coleções de dados no *Firestore*, sendo essas, "*public_users*" e "*private_users*". Por outro lado, quando há o *back-end* próprio, é possível tratar essas informações de retorno sem a necessidade de dividir o banco de dados, mas, sendo necessária a implementação desses recursos já providos pelo *Google Firebase*, por parte do desenvolvedor.

Outro fator decisivo para ter o próprio *back-end*, é a otimização de custos a longo prazo. Geralmente, serviços de nuvem no início das aplicações podem ser vantajosos, mas são capazes de se tornarem mais custosos à medida que uma aplicação escala, por exemplo, no caso de aplicações que utilizam o serviço de SMS para autenticação, o *Google Firebase* fornece até 10 autenticações por SMS gratuitas diariamente, mas após isso, é cobrada uma taxa por SMS enviado (Google Firebase Team, 2024b). Sendo assim, vale ressaltar que também existem custos por armazenamento no *Cloud Firestore* caso exceda o total de 1GiB de dados populados, assim como nos outros bancos de dados disponibilizados por

esse serviço de nuvem. Portanto, ter um *back-end* próprio, por mais que seja mais custoso inicialmente, a longo prazo a otimização de custos torna-se um fator decisivo.

5.4.2 Utilização de serviço de mensageria

A utilização de um serviço de mensageria, por exemplo, o *RabbitMQ*, proporciona outro paradigma de arquitetura. Entre todos os benefícios possíveis, destacam-se os fatores de balanceamento de carga automática, resiliência e alta disponibilidade, além de escalabilidade.

O balanceamento de carga automática se refere à capacidade de distribuir de forma equilibrada as mensagens para os consumidores nas filas. Ou seja, se para uma fila em questão, por exemplo, a de cadastro de usuários tiver mais de um consumidor e receber muitas requisições ao mesmo tempo, essas mensagens serão distribuídas entre esses consumidores para não sobrecarregar o sistema.

Resiliência e alta disponibilidade, se referem à capacidade do *RabbitMQ* de fornecer mecanismos que garantem a entrega e o processamento das mensagens. No contexto desses mecanismos, existe a comprovação de *Acknowledgement* (ACK), de forma manual ou automática. Ambas formas possuem o propósito de tratar o recebimento da mensagem, por exemplo, se um consumidor consome uma mensagem, mas acontece uma falha durante o tratamento desse dado, existe a possibilidade de não enviar a confirmação de recebimento dessa para o *RabbitMQ*. Nesses casos, o armazenamento persistente é útil, conforme a política de retenção de dados, ou seja, o consumidor poderá processar novamente essa informação e quando houver a confirmação essa mensagem é removida da fila.

A escalabilidade é outro fator importante quando se utiliza serviços de mensageria. Nesse contexto, pode-se imaginar um cenário que o sistema está recebendo muitas requisições ao mesmo tempo, e sendo assim, o usuário consegue adicionar mais consumidores para dividir o consumo e o processamento dessas informações.

5.4.3 Lições aprendidas

A avaliação do software analisado neste estudo, com base nos critérios selecionados da ISO/IEC 25010, trouxe resultados variados. Para facilitar a interpretação dessas informações, foi elaborada a Tabela 2. Esta tabela, apresenta uma análise comparativa entre as diferentes arquiteturas estudadas, destacando para cada uma os principais aspectos de aprendizado em relação aos critérios correspondentes, assim como seus prós e contras.

Tabela 2 – Comparações dos critérios definidos para as arquiteturas

Critérios	Arquitetura 1	Arquitetura 2	Arquitetura 3
Modificabilidade	O Firestore, quando utilizado diretamente pelo front-end da aplicação, fornece uma simples sintaxe para alterações de dados, além da facilidade para inclusão de novos campos com poucas etapas.	A utilização de um servidor próprio, traz maior complexidade durante a manutenção de uma função que se comunica com o banco de dados, por meio de rotas definidas nesse servidor.	A utilização de um servidor próprio, traz maior complexidade durante a manutenção de uma função que se comunica com o banco de dados, por meio de rotas definidas nesse servidor.
Tolerância a Falhas	Existe um tratamento automático de erros durante possíveis falhas de requisição. As requisições são armazenadas em cache local e enviadas de forma automática, quando não há mais falhas.	É necessária a implementação do tratamento de possíveis falhas de requisição por parte do programador, garantindo que a requisição chegue ao servidor.	É necessária a implementação do tratamento de possíveis falhas de requisição por parte do programador, garantindo que a requisição chegue ao servidor.
Confidencialidade	O tratamento das solicitações e a devolução das informações necessárias, pode ser facilmente configurado pelas regras de segurança, acessadas por meio do painel do Google Firebase.	É necessária a implementação de verificação da solicitação antes de retornar os dados requisitados. Tem maior flexibilidade no tratamento dessas solicitações, por serem executadas em um servidor próprio.	É necessária a implementação de verificação da solicitação antes de retornar os dados requisitados. Tem maior flexibilidade no tratamento dessas solicitações, por serem executadas em um servidor próprio.
Utilização de back-end próprio	Não possui back-end próprio. Portanto, pode se tornar custoso, a longo prazo, manter o sistema em funcionamento conforme o número de requisições diárias recebidas.	Menos custos a longo prazo. Controle completo sobre a arquitetura, tecnologias utilizadas e lógica de negócios.	Menos custos a longo prazo. Controle completo sobre a arquitetura, tecnologias utilizadas e lógica de negócios.
Utilização de serviço de mensageria	Não possui serviço de mensageria. Para ser escalável, necessita de mais recursos providos pela nuvem, o que pode ser custoso.	Não possui serviço de mensageria. Para ser escalável, necessita de mais recursos providos pela nuvem, o que pode ser custoso	Sistema escalável. Alta capacidade de tratamento de dados e entrega das mensagens. Balanceamento de carga automática e alta disponibilidade.

6 Conclusão

No contexto contemporâneo do desenvolvimento de software, profissionais e organizações encontram uma grande variedade de arquiteturas e tecnologias disponíveis para a elaboração de sistemas computacionais. Com inúmeras possibilidades de decisões tecnológicas, surge um desafio significativo: a escolha da arquitetura mais adequada para um projeto em questão. Para essa decisão, deve ser levada em consideração a identificação de um conjunto de recursos e tecnologias que se enquadram no escopo da proposta do software e é necessário ressaltar que os serviços em nuvem contribuem muito para essa diversidade de opções. O *Google Firebase*, por exemplo, proporciona ao desenvolvedor ferramentas de extrema importância, uma vez, que se trata de um BaaS (*Back-end as a Service*) e DaaS (*Database as a Service*), isto é, fornece um conjunto de ferramentas para o gerenciamento do back-end, sem a haver a necessidade de implementação de uma estrutura para isso, além de permitir a administração de informações em um banco de dados, de forma on-line, por exemplo, com o *Firestore*, *Realtime Database* ou o sistema de armazenamento de arquivos *Storage*.

A implementação do sistema objeto de estudo desta pesquisa, teve como foco a utilização dos serviços do *Google Firebase*, em partes, o *Firestore* e o *Realtime Database*, além do serviço de autenticação. Para a comparação das três arquiteturas desenvolvidas, foi utilizada a versão mais recente da ISO/IEC 25010, a qual se trata de uma norma internacional que define os parâmetros para avaliar a qualidade de um sistema de software. Com base nessa ISO, foram escolhidos os critérios de comparação de manutenibilidade, subcritério de modificabilidade, confiabilidade, o qual é um subcritério de tolerância a falhas e confidencialidade, subcritério de segurança. Também foram consideradas as vantagens de possuir um *back-end* próprio e a utilização de um serviço de mensageria, embora esses critérios não tenham sido analisados com a mesma rigidez sistemática. Além disso, ressalta-se, que para as três arquiteturas foi selecionada e comparada uma função em comum, sendo essa, a função denominada por "*insertDataNewUser*", a qual tem o mesmo propósito para as três versões do sistema.

Após finalizadas as análises, notou-se que para a primeira arquitetura, a qual foi implementada diretamente pelo *front-end* utilizando apenas o *Google Firebase*, para os critérios de manutenibilidade, tolerância a falhas e confidencialidade, as respostas encontradas foram mais positivas do que para as outras duas arquiteturas implementadas. Esse resultado é conclusivo, haja vista que para o critério de manutenibilidade, para acrescentar um novo campo ao banco de dados, foi necessário apenas informar o nome desse e o seu valor, em formato de objeto em *JavaScript* (em cada chamada do método ".set"). Além disso, durante a comparação de outro critério escolhido, o de tolerância a

falhas, foi possível armazenar em cache local uma requisição do sistema feita completamente de forma off-line e, uma vez que a conexão com a internet foi reestabelecida e a aplicação foi reaberta, essa requisição foi enviada ao *Firestore* de forma automática, populando, as coleções de usuários denominadas por "*public-users*" e "*private-users*". Por fim, para o critério de confidencialidade, concluiu-se que não foi possível obter os dados dos outros usuários (diferentemente das arquiteturas 2 e 3), configurando o acesso a esses por meio do painel de segurança do *Cloud Firestore*. Sendo assim, para os três critérios, notou-se eficiência nas questões de lidar com tolerância a falhas, maior facilidade para a manutenção da função "*insertDataNewUser*", no caso da necessidade de acrescentar um novo campo as coleções de usuários e na segurança dos dados dos usuários cadastrados, o que representou uma mudança de paradigma no desenvolvimento do software, priorizando a eficiência e o foco na experiência do usuário final.

Para a segunda e a terceira arquitetura, os resultados obtidos foram semelhantes entre essas. No quesito de manutenibilidade, para a segunda arquitetura, foram necessárias alterações no *front-end* durante a requisição e também, do lado do servidor, durante o tratamento dos dados. Para a terceira arquitetura, foi preciso fazer os mesmos procedimentos da segunda, porém com uma etapa a mais necessária, sendo essa, a atualização dessas informações nas rotas do *RabbitMQ* para o dado a ser enviado e recebido. Dessa forma, tornou-se mais complexa a inclusão de um único campo para essas versões do sistema. No critério de tolerância a falhas, essas duas arquiteturas (*FIREBASE-MYSQL* e *FIREBASE-MSG*), não tiveram a implementação por parte do desenvolvedor de um mecanismo que armazena as requisições em cache local e as reenvia quando há conexão com a internet. Portanto, para ambas, a resposta da análise nesta etapa, não foi positiva quanto a do primeiro teste realizado, ou seja, pode-se afirmar que a requisição não chegou ao *endpoint* construído e, consequentemente, foi perdida após a reinicialização da aplicação. Para que essas arquiteturas se equiparassem a primeira (*FIREBASE*) nesse teste, seria necessária a implementação desse mecanismo de armazenamento de requisições em cache local previamente citado, o que pode ser feito em trabalhos futuros. Por fim, para a terceira análise, a qual envolve o critério de confidencialidade, notou-se que as duas últimas arquiteturas devolveram para as requisições realizadas os dados solicitados, não havendo bloqueio desses por falta de autenticação ou por não pertencerem ao usuário que as está solicitando. Por outro lado, essas duas arquiteturas possuem maior flexibilidade no tratamento dessas informações, haja vista que a segurança pode ser desenvolvida pelo programador, diferentemente da arquitetura 1, na qual essa questão é provida pelos serviços do *Google Firebase*.

Por fim, é necessário ressaltar que a primeira arquitetura foi a que apresentou melhores resultados em todos os quesitos de comparação (exceto nos critérios definidos em outros critérios), ou seja, para o código implementado, forneceu o melhor conjunto de decisões tecnológicas para esse propósito. O tratamento automático de tolerância

a falhas, a simples sintaxe, os passos necessários durante a inclusão de um campo em uma coleção do banco de dados, além da modificação das regras de segurança para as operações com esses dados populados, são benefícios que a utilização do *Google Firebase* proporciona a um programador, por oferecer um serviço de *BaaS* e *DaaS*. Ademais, para as outras duas arquiteturas, também notam-se vantagens, ou seja, entre elas, quando há a criação de um servidor próprio, existe possibilidade de melhor tratamento de erros durante a requisição, além de lidar com as respostas do servidor para o *front-end* de forma personalizada. Além disso, a utilização de um serviço de mensageria, por exemplo, o *RabbitMQ*, proporciona outro paradigma de arquitetura, podendo armazenar as requisições realizadas ao sistema em filas específicas, as quais são lidas e consumidas pelos usuários que precisam receber essas informações, o que garante a replicação de estado do sistema conforme a política de retenção de dados, caso haja problemas. Sendo assim, conclui-se que cada arquitetura construída tem os seus prós e contras, mas também, com propósitos e ferramentas diferentes, deixando ao desenvolvedor a escolha entre desenvolver sistemas com mais liberdade de tratamento de erros de forma personalizada ou, atuar com um conjunto de ferramentas e estruturas previamente fornecidas por algum serviço de nuvem.

Referências

ACEVEDO, C. A. J.; JORGE, J. P. Gómez y; PATIñO, I. R. Methodology to transform a monolithic software into a microservice architecture. In: *2017 6th International Conference on Software Process Improvement (CIMPS)*. [S.l.: s.n.], 2017. p. 1–6. Citado 2 vezes nas páginas 17 e 19.

EMMADI, S. S. R.; POTLURI, S. Android based instant messaging application using firebase. *International Journal Recent Technology and Engneering*, v. 7, n. 5, p. 352–355, 2019. Citado 2 vezes nas páginas 15 e 19.

FIDELIS, J. *Caronas Universitárias*. 2023. <https://github.com/joaofidelisc/caronasuniversitarias>. Acessado em 1 de janeiro de 2024. Citado na página 22.

Google Cloud. *Gerenciar dados do Firestore em modo offline*. 2023. <https://cloud.google.com/firestore/docs/manage-data/enable-offline?hl=pt-br>. Acessado em: 25 de dezembro de 2023. Citado na página 44.

Google Firebase. *Firebase Realtime Database*. 2024. <https://firebase.google.com/docs/database?hl=pt-br>. Acessado em 21 de Janeiro de 2024. Citado na página 15.

Google Firebase. *Modelo de dados do Cloud Firestore*. 2024. Disponível em: <https://firebase.google.com/docs/firestore/data-model?hl=pt-br>. Acessado em 21 de Janeiro de 2024. Citado na página 15.

Google Firebase Team. *Firebase Firestore REST API Documentation*. 2024. Acessado em 13 de janeiro de 2024. Disponível em: <https://firebase.google.com/docs/firestore/use-rest-api>. Citado na página 49.

Google Firebase Team. *Firebase Pricing*. Google Firebase, 2024. <https://firebase.google.com/pricing?hl=pt-br>. Acessado em 14 de janeiro de 2024. Disponível em: <https://firebase.google.com/pricing?hl=pt-br>. Citado na página 53.

ISO/IEC 25010. *ISO/IEC 25010:2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. 2011. <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010>. Acessado em: 10 de dezembro de 2023. Citado 2 vezes nas páginas 11 e 30.

KHAWAS, C.; SHAH, P. Application of firebase in android app development-a study. *International Journal of Computer Applications*, v. 179, n. 46, p. 49–53, 2018. Citado 2 vezes nas páginas 15 e 19.

LI, S. Understanding quality attributes in microservice architecture. In: *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*. [S.l.: s.n.], 2017. p. 9–10. Citado na página 10.

MADAMINOV, U. A.; ALLABERGANOVA, M. R. Firebase database usage and application technology in modern mobile applications. In: *2023 IEEE XVI International Scientific and Technical Conference Actual Problems of Electronic Instrument Engineering (APEIE)*. [S.l.: s.n.], 2023. p. 1690–1694. Citado 2 vezes nas páginas 17 e 19.

PRATAMA, H. P.; PRIHATMANTO, A. S.; SUKOCO, A. Implementation messaging broker middleware for architecture of public transportation monitoring system. In: *2020 6th International Conference on Interactive Digital Media (ICIDM)*. [S.l.: s.n.], 2020. p. 1–5. Citado 2 vezes nas páginas 13 e 19.

PRELLWITZ, M.; PARZYJEGLA, H.; MüHL, G. Programming abstractions for messaging protocols in event-based systems. In: *Proceedings of the 17th ACM International Conference on Distributed and Event-Based Systems*. New York, NY, USA: Association for Computing Machinery, 2023. (DEBS '23), p. 157–167. ISBN 9798400701221. Disponível em: <https://doi.org/10.1145/3583678.3596896>. Citado 2 vezes nas páginas 14 e 19.

SANTOS, N. M. et al. Data management for mobile applications dependent on geo-located data. In: *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data*. New York, NY, USA: Association for Computing Machinery, 2023. (PaPoC '23), p. 70–76. ISBN 9798400700866. Disponível em: <https://doi.org/10.1145/3578358.3591334>. Citado 2 vezes nas páginas 14 e 19.

SHARVARI, T.; NAG, K. S. A study on modern messaging systems-kafka, rabbitmq and nats streaming. *CoRR abs/1912.03715*, 2019. Citado 2 vezes nas páginas 16 e 19.

SJODIN, R.; LOTFY, M. Developing cloud-based mobile apps using google firebase. *J. Comput. Sci. Coll.*, Consortium for Computing Sciences in Colleges, Evansville, IN, USA, v. 35, n. 2, p. 16–18, oct 2019. ISSN 1937-4771. Citado 2 vezes nas páginas 13 e 19.

SJODIN, R.; MASON, R.; LOTFY, M. Integrating cloud-based nosql technology and mobile phone relational databases. *The Journal of Computing Sciences in Colleges*, p. 24, 2020. Citado 2 vezes nas páginas 16 e 19.