Implantação de Backend com Pipeline CI/CD utilizando GitHub Actions e AWS EC2[1]

William Colombo Breda[2], Rafael Vieira Coelho[3]

Instituto Federal do Rio Grande do Sul - *Campus*
Farroupilha Tecnologia em Análise e Desenvolvimento
de Sistemas 95174-274 – Farroupilha – RS – Brasil

williamcolombobreda@gmail.com[4], rafael.coelho@farroupilha.ifrs.edu.br[5]

Resumo. A entrega de *software* com qualidade e robustez está se tornando cada vez mais essencial para empresas de tecnologia que buscam se manter competitivas no mercado. A prática de CI/CD[6] (*Continuous Integration/ Continuous Deploy*) é um processo fundamental que pode ser implementado em aplicações de diferentes escalas. Sem ele, é necessário um esforço maior para definir todas as etapas do ciclo de vida de um *software*. Uma *pipeline* com essa funcionalidade proporciona uma visão clara das etapas necessárias para a entrega do produto em produção. Ao longo desse processo, é possível definir estágios com comportamentos específicos que são essenciais para o sucesso da entrega, tais como: (a) *build* do projeto; (b) análise estática de código; (c) *deploy* da imagem no Docker.io; (d) subida da aplicação na AWS; e (e) testes de integração. O presente trabalho propõe a criação desta *pipeline* utilizando um sistema escolar baseado na arquitetura Spring Boot com funções básicas.

Abstract. *Delivering quality and robust software is becoming increasingly essential for technology companies seeking to remain competitive in the market. The practice of CI/CD is a fundamental process that can be implemented in applications of different scales. Without it, greater effort is required to define all stages of the software life cycle. A pipeline with this functionality provides a clear view of the steps necessary to deliver the product into production. Throughout this process, it is possible to define stages with specific behaviors that are essential for successful delivery, such as: (a) project build; (b) static code analysis; (c) deploy the image to Docker.io; (d) upload of the application to AWS; and (e) integration tests. The present work proposes the creation of this pipeline using a school system based on the Spring Boot architecture with basic functions.*

---

[1] Artigo científico referente ao Trabalho de Conclusão de Curso do curso de Tecnologia em Análise e Desenvolvimento de Sistemas do Instituto Federal do Rio Grande do Sul - *Campus* Farroupilha.
[2] Aluno matriculado no Trabalho de Conclusão de Curso.
[3] Professor orientador do Trabalho de Conclusão de Curso.
[4] Endereço de e-mail do Aluno
[5] Endereço de e-mail do Professor Orientador
[6] Prática que automatiza a construção, teste e implantação de código para melhorar a eficiência e a qualidade do desenvolvimento de software.

1. Introdução

Atualmente, a área de desenvolvimento de *software* tem experimentado uma transformação significativa impulsionada pela demanda crescente por entregas de *software* mais eficientes e robustas. Nesse contexto, a integração contínua (*CI - Continuous Integration*) e a entrega contínua (*CD - Continuous Delivery*) são práticas fundamentais para alcançar esses objetivos. Segundo Fowler (2006), a automação de processos de construção, teste e implantação tem se mostrado crucial para melhorar a eficiência do desenvolvimento de *software* e reduzir o tempo de lançamento de novas funcionalidades.

A ausência de uma abordagem de Integração Contínua e Entrega Contínua (CI/CD) em projetos de desenvolvimento de *software* pode resultar em uma série de problemas significativos. Sem a automação proporcionada pelo CI/CD, os processos de construção, teste e implantação de *software* tendem a ser manuais e propensos a erros humanos, o que pode levar a inconsistências e falhas na integração de código. Além disso, a falta de um *pipeline* automatizado pode causar atrasos na entrega de atualizações e correções, comprometendo a agilidade e a capacidade de resposta às necessidades do mercado e dos usuários. A ausência de CI/CD também dificulta a detecção precoce de defeitos e a validação contínua da qualidade do *software*, aumentando o risco de *bugs* em produção e reduzindo a confiabilidade e a estabilidade do sistema. Portanto, a implementação de CI/CD é essencial para garantir processos de desenvolvimento mais eficientes, com menor risco de falhas e maior capacidade de entrega contínua de valor ao cliente.

O objetivo é explorar e demonstrar a implantação de um *backend*[7] utilizando uma *pipeline*[8] de Integração Contínua e Entrega Contínua (CI/CD) com o auxílio do GitHub Actions e AWS EC2. Através de uma abordagem prática e detalhada, o estudo visa evidenciar as melhores práticas para automatizar o processo de *deploy*, garantindo eficiência, segurança e confiabilidade. A utilização do GitHub Actions para orquestrar as etapas de *build*, teste e *deploy*, juntamente com a robustez da infraestrutura fornecida pela AWS EC2, proporciona um ambiente de desenvolvimento e operação altamente otimizado. O artigo também busca destacar os benefícios de uma *pipeline* CI/CD bem implementada, tais como a redução de erros manuais, aumento da produtividade da equipe e a capacidade de entregar *software* de alta qualidade de forma contínua e ágil.

Na próxima seção, serão abordados os conceitos fundamentais de CI/CD, a arquitetura de *pipelines* com GitHub Actions e o ambiente de implantação para otimização de custos, além de estudos de caso que demonstram a aplicação prática desses conceitos DevOps[9]. Esta pesquisa é valiosa para desenvolvedores, equipes e empresas que desejam adotar práticas modernas de desenvolvimento de *software*, aprimorando sua eficiência e competitividade no mercado.

---

[7] Parte do sistema que gerencia a lógica de negócio e banco de dados.

[8] Fluxo de etapas sequenciais para processar e automatizar rotinas manuais.

[9] DevOps é uma prática que integra desenvolvimento de *software* (Dev) e operações de TI (Ops) para melhorar a colaboração e automação dos processos de desenvolvimento, implantação e gerenciamento de aplicações.

2. Referencial Teórico

Esta seção aborda os conceitos fundamentais para a elaboração deste trabalho, explicando todas ferramentas utilizadas para que a abordagem de CI/CD possa ser realizada com sucesso dentro do GitHub Actions.

2.1. Integração Contínua (CI) e Entrega Contínua (CD)

A integração contínua (CI) e a entrega contínua (CD) são práticas de desenvolvimento de *software* que visam automatizar e melhorar a eficiência do processo de desenvolvimento. CI é a prática de integrar código de forma contínua em um repositório compartilhado, permitindo a detecção precoce de problemas de integração (Fowler, 2006). CD, por sua vez, expande o conceito de CI ao incluir a automação do processo de entrega, permitindo que as alterações de código sejam rapidamente entregues ao ambiente de produção de forma confiável (Humble & Farley, 2010).

As práticas de CI/CD têm se mostrado fundamentais para acelerar o ciclo de desenvolvimento, melhorar a qualidade do *software* e permitir uma resposta ágil às mudanças de mercado. Com CI/CD, as equipes de desenvolvimento conseguem detectar e corrigir erros mais rapidamente, garantindo que o *software* entregue esteja sempre em um estado utilizável (Duvall, Matyas & Glover, 2007).

2.2. GitHub

O GitHub é uma das ferramentas mais populares e confiáveis para gerenciamento de repositórios, oferecendo uma plataforma robusta e intuitiva para colaboração e controle de versões. Utilizamos o GitHub neste trabalho não apenas para gerenciar nosso repositório, mas também para aproveitar suas funcionalidades avançadas, como GitHub Actions e GitHub Pages, que são essenciais para automação de *workflows* e publicação de páginas estáticas. Sua qualidade e importância no mercado são evidentes pela ampla adoção e pela capacidade de facilitar o desenvolvimento colaborativo e a integração contínua.

O GitHub Actions é uma plataforma de integração contínua e entrega contínua (CI/CD) que permite automatizar fluxos de trabalho de desenvolvimento diretamente nos repositórios GitHub. Introduzido em 2019, ele oferece suporte para automação de *pipelines* e integra-se nativamente com GitHub para facilitar a automação de processos de desenvolvimento.

GitHub Actions permite aos desenvolvedores integrar código diretamente a partir do GitHub, automatizando testes, *builds*, *deploys* e outras tarefas rotineiras de desenvolvimento. Estudos mostram que uma grande parte dos repositórios no GitHub já adotaram o GitHub Actions para automatizar seus fluxos de trabalho. Segundo Decan (2022), em um estudo com 68 mil repositórios, 43,9% utilizavam fluxos de trabalho automatizados com GitHub Actions, sendo assim considerada uma ferramenta muito popular. A reutilização de ações é uma prática comum, facilitando a padronização e a manutenção de *pipelines* complexos.

Já o GitHub Pages é um serviço gratuito de hospedagem de sites estáticos oferecido pelo GitHub, ideal para desenvolvedores que desejam criar e compartilhar rapidamente sites pessoais, documentação de projetos e portfólios. Neste projeto, ele é utilizado para hospedar os *logs* de testes do Allure Reports.

2.3. SonarQube

SonarQube é uma aplicação de código aberto para análise contínua de qualidade de código. Ele permite que as equipes de desenvolvimento avaliem a qualidade de seu código, identificando bugs, vulnerabilidades, dívidas técnicas e outros pontos críticos. A integração do SonarQube com *pipelines* CI/CD permite que essas análises sejam realizadas automaticamente, garantindo a manutenção da qualidade do código ao longo de todo o ciclo de desenvolvimento (Campbell & Papapetrou, 2013).

Possui interface amigável e exibe as informações organizadas em várias seções, com destaque para segurança, confiabilidade e manutenção. Dentro dessas seções, temos acesso aos detalhes das críticas geradas pela ferramenta.

A utilização do SonarQube em *pipelines* CI/CD, como implementado neste trabalho na etapa de homologação, ajuda a garantir que as práticas de desenvolvimento sigam padrões de qualidade. Isso aumenta a confiabilidade e a segurança do *software* entregue, minimizando possíveis imprevistos relacionados a comportamento ou falhas.

2.4. Allure Reports

Allure Reports é uma ferramenta de geração de relatórios para testes automatizados que permite a visualização detalhada dos resultados de testes. Ele organiza os resultados de testes de maneira clara e intuitiva, facilitando a identificação de falhas e a análise do desempenho dos testes (Kochetkov, 2017).

A integração do Allure Reports em um *pipeline* CI/CD fornece uma visão detalhada e acessível dos resultados dos testes, melhorando a transparência e a compreensão sobre a qualidade do código no escopo do time. Isso é particularmente útil para equipes de desenvolvimento que buscam manter um alto padrão de qualidade e eficiência em seus processos de testes de *software*.

A aplicação tem uma interface amigável e intuitiva, de fácil instalação e configuração. Neste trabalho, ele estará integrado com o Github Pages, a cada *push* que acontecer em homologação, a automação irá gerar todos os *logs*[10] dos testes dentro do próprio Github.

2.5. Amazon Web Services (AWS)

AWS é uma das principais plataformas de serviços de computação em nuvem, oferecendo uma ampla gama de serviços que permitem a criação, implantação e gerenciamento de aplicações na nuvem. Entre esses serviços, o Amazon EC2 (Elastic Compute Cloud) permite a execução de instâncias de servidores virtuais, que podem ser usadas para hospedar aplicações de maneira escalável e segura (Amazon Web Services, 2020).

No contexto deste trabalho, a utilização de instâncias EC2 para hospedar os ambientes de homologação e produção oferece flexibilidade e escalabilidade, permitindo que o ambiente de desenvolvimento se ajuste rapidamente às necessidades

[10] Registro sequencial de eventos ou mensagens gerados por um sistema para monitoramento.

do projeto. A segmentação dos ambientes em AWS garante que o processo de desenvolvimento e *deploy* seja gerido de maneira eficiente e segura.

Sua interface possui uma gama de configurações básicas e avançadas, para o desenvolvimento deste trabalho, foi necessário ajustar algumas configurações para o correto funcionamento e sucesso dos fluxos de automação do Github Actions.

2.6. Java Spring Boot

Java Spring Boot é uma extensão do *framework* Spring que torna o desenvolvimento de aplicações Java mais simples e rápido. Ele fornece uma configuração padrão para a criação de aplicativos Spring, reduzindo a necessidade de configurações manuais e complexas (Johnson, 2014).

O Spring Boot oferece suporte integrado para diversas práticas de desenvolvimento modernas, como integração com bancos de dados, segurança, mensageria e APIs RESTful. Além disso, ele é projetado para rodar em containers, o que facilita a implantação em ambientes de nuvem, como o AWS EC2 (Turnquist, 2017).

No contexto de um *pipeline* CI/CD, usar Spring Boot traz várias vantagens. A configuração automática e a fácil integração com ferramentas de teste e análise, como JUnit para testes de integração e SonarQube para análise de qualidade de código, tornam o Spring Boot uma escolha ideal para ambientes de desenvolvimento contínuo. Sua compatibilidade com containers Docker também simplifica a implantação contínua em plataformas de nuvem, garantindo que as aplicações possam ser escaladas e gerenciadas facilmente (Walls, 2016).

2.7. Maven

Maven é uma ferramenta de automação de construção e gerenciamento de projetos desenvolvida pela Apache Software Foundation, que simplifica a configuração e o gerenciamento de projetos Java. Utilizando o modelo de objeto de projeto (POM), Maven permite uma configuração centralizada, facilitando a manutenção e padronização dos processos de construção e gerenciamento de dependências (Apache Maven, 2024).

A principal vantagem do Maven é o gerenciamento eficiente de dependências. Com Maven, desenvolvedores podem declarar as dependências necessárias em um arquivo POM, e essas dependências são automaticamente resolvidas e baixadas de repositórios remotos, evitando conflitos e problemas de compatibilidade (Heffelfinger, 2014).

Além disso, Maven promove uma estrutura de projeto padronizada, facilitando a colaboração e manutenção do código. Ao seguir convenções predefinidas, os desenvolvedores evitam configurações complexas, tornando o processo de construção mais previsível e eficiente (Kousen, 2019).

2.8. Docker

Docker é uma plataforma de código aberto que facilita a implantação de aplicações em contêineres, criando um ambiente isolado e consistente para execução. Os contêineres Docker são leves e eficientes, oferecendo uma alternativa às

tradicionais máquinas virtuais. Isso permite que os desenvolvedores empacotem uma aplicação com todas as suas dependências, garantindo portabilidade entre diferentes ambientes de desenvolvimento, teste e produção, e reduzindo problemas de compatibilidade (Merkel, 2014).

A arquitetura do Docker é composta pelo Docker Engine e o Docker Hub. O Docker Engine é o motor que cria e gerencia os contêineres, enquanto o Docker Hub é um serviço de registro baseado na nuvem que facilita o compartilhamento de imagens de contêineres. Essa estrutura permite o uso de imagens pré-configuradas ou personalizadas, acelerando o desenvolvimento e a implantação de aplicações. A comunidade ativa do Docker contribui constantemente com novas imagens e melhorias, ampliando suas possibilidades de uso (Pahl, 2015).

No contexto de integração contínua (CI) e entrega contínua (CD), Docker se destaca como uma ferramenta essencial. Usar containers Docker em *pipelines* CI/CD traz benefícios como a consistência do ambiente, rapidez na execução de testes e facilidade na implementação de novas versões de *software*. Docker se integra bem com ferramentas como Jenkins, GitLab CI e GitHub Actions, automatizando todo o ciclo de desenvolvimento de *software*, desde o código até a produção, e garantindo entregas mais rápidas e confiáveis (Turnbull, 2014).

2.9. PostgreSQL

PostgreSQL é um sistema de gerenciamento de banco de dados relacional de código aberto que adere estritamente aos padrões SQL. Ele suporta consultas complexas, integridade transacional e funções avançadas, tornando-o ideal para uma ampla variedade de projetos (Postgres, 2024). A escolha pelo PostgreSQL foi baseada em sua robustez e praticidade para o projeto, oferecendo conformidade com as propriedades ACID, alta escalabilidade, extensibilidade através de módulos e suporte a transações complexas. Adicionalmente, PostgreSQL está disponível em uma imagem Docker, facilitando enormemente o processo de configuração. Com um único comando, é possível criar e configurar o banco de dados para integrá-lo à aplicação, simplificando a implantação e manutenção do ambiente de desenvolvimento e produção.

2.10. Flyway

Flyway é uma ferramenta poderosa para a execução e automação de tarefas de banco de dados. Integrada diretamente aos projetos, ela mantém um repositório interno onde registra os *scripts* de migração executados. Cada vez que a aplicação é iniciada, o Flyway verifica a tabela de histórico de migrações e compara com os *scripts* disponíveis. Se houver novos *scripts* ou modificações nos existentes, Flyway executa as ações predefinidas, como aplicar ou reverter as migrações. Suportando uma ampla variedade de sistemas de gerenciamento de banco de dados relacionais, incluindo MySQL, PostgreSQL, Oracle e SQL Server, Flyway oferece grande flexibilidade na administração de múltiplos ambientes de desenvolvimento, teste e produção. Além disso, Flyway permite a execução ordenada de migrações, gerenciamento de conflitos de versão e estratégias de rollback, promovendo práticas de DevOps e integração contínua (CI/CD). Essas funcionalidades tornam o Flyway uma ferramenta muito útil para equipes de desenvolvimento que buscam automatizar e simplificar a gestão de

mudanças em bancos de dados, assegurando consistência e integridade em todos os ambientes.

3. Metodologia

A implementação envolve dois ambientes distintos na AWS, sendo um para homologação e outro para produção, ambos hospedados em instâncias EC2 (Amazon Web Services, 2020). Os *deploys* serão automaticamente acionados através de *triggers* em *branches*[11] específicas, garantindo que as atualizações passem por um rigoroso processo de validação antes de serem disponibilizadas em produção. A utilização de AWS para hospedagem oferece escalabilidade e robustez, enquanto a segmentação dos ambientes permite uma gestão mais eficiente e segura do ciclo de desenvolvimento.

Nas Figuras 1, 2, 3 e 4, é apresentado o fluxo de trabalho de maneira visual, destacando todas as ações necessárias e as ferramentas utilizadas para garantir a execução bem-sucedida das tarefas. Conforme ilustrado na Figura 1, o processo se inicia quando o desenvolvedor trabalha em seu *branch* específico. Ao concluir a atividade, ele submete um *pull request*[12] para o *branch* de homologação. Após atender aos requisitos de aprovação e fornecer as evidências de testes necessárias, o *merge*[13] é habilitado, permitindo a continuação do fluxo de integração.

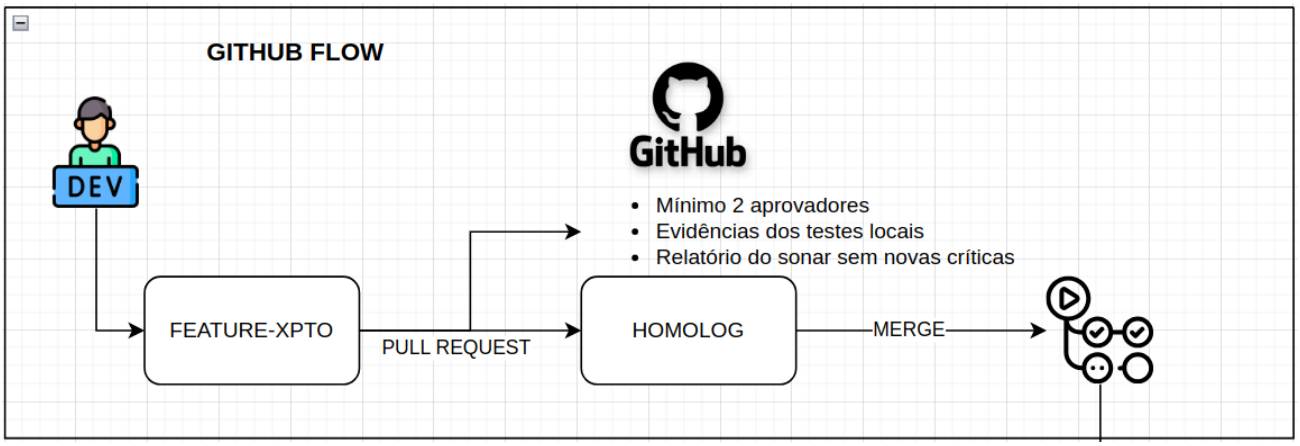

Figura 1: Imagem mostrando fluxo inicial da *pipeline*
Fonte: Autoria própria retirada do repositório do projeto

Na Figura 2, ilustramos a etapa inicial da execução da *pipeline*, que é o processo de *build*. Nesta fase, o Maven é utilizado para compilar e gerar o artefato executável da aplicação. Após a conclusão da construção, o artefato é utilizado para

[11] Ramificação do código principal que permite desenvolver novas funcionalidades sem afetar o código.
[12] Solicitação para revisar mudanças de código de um *branch* para outro em um repositório.
[13] Processo de integrar mudanças de código de um *branch* para outro.

criar uma imagem Docker. Esta imagem é então enviada para o Docker Hub, que atua como o repositório central para o gerenciamento e armazenamento das imagens Docker produzidas.

Subsequentemente, a imagem Docker é implantada na instância de homologação da AWS. No processo, o Docker realiza o *pull*[14] da imagem previamente gerada, substitui a imagem existente que está em execução na instância e, em seguida, inicia o contêiner com a nova imagem. Esse processo garante que a aplicação em homologação esteja atualizada com a versão mais recente da imagem.

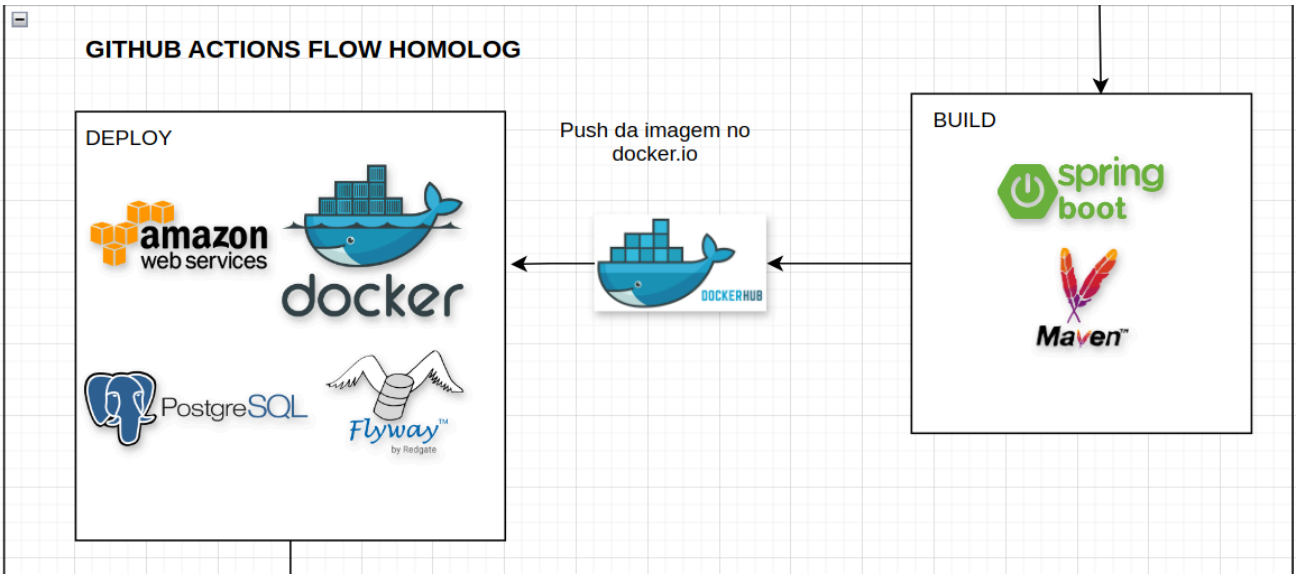

Figura 2: Imagem mostrando as etapas que são executadas em homologação
Fonte: Autoria própria retirada do repositório do projeto

Na Figura 3, detalha-se o fluxo subsequente à conclusão bem-sucedida do *deploy*. Nessa fase, a aplicação é submetida a uma série de testes de integração para validar seu comportamento e a integração com outros componentes. Simultaneamente, a análise estática do código é realizada pelo SonarQube, e os relatórios resultantes são gerados e disponibilizados.

Os *logs* dos testes de integração são registrados e armazenados no Allure Reports, cuja visualização é facilitada por meio do GitHub Pages, que serve como o repositório de documentação e relatórios. Durante esta etapa, é crucial validar os logs dos testes para identificar qualquer falha ocorrida e revisar as críticas e novas vulnerabilidades detectadas pelo SonarQube. A análise minuciosa desses *logs* e relatórios permite assegurar que a aplicação não só funcione conforme o esperado, mas também esteja em conformidade com os padrões de qualidade e segurança estabelecidos.

[14] Processo de baixar uma imagem Docker de um repositório, como o Docker Hub, para uso local.

Figura 3: Imagem ilustrando o fluxo da parte de testes após *deploy* em homologação
Fonte: Autoria própria retirada do repositório do projeto

Após a análise e validação da nova versão em homologação, que pode ser realizada tanto pela equipe de desenvolvimento quanto pela equipe que requisitou a modificação, a etapa seguinte é o *deploy* em produção. Nesta fase, a imagem Docker previamente validada e considerada estável na instância de homologação é preparada para produção.

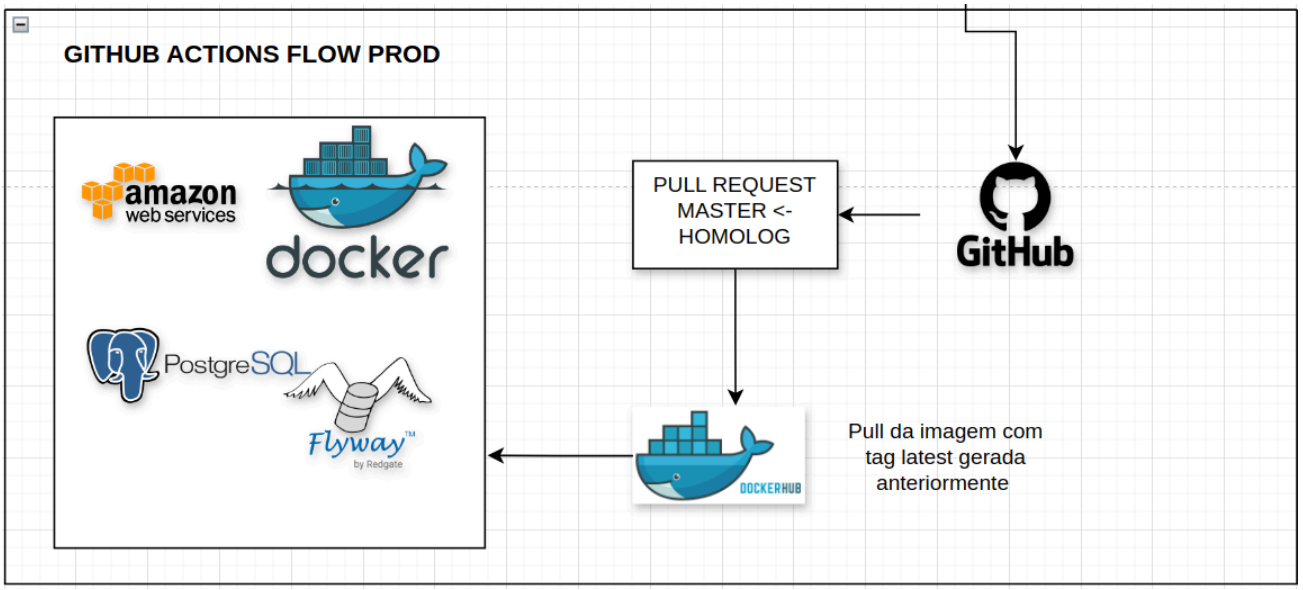

Figura 4: Imagem mostrando o fluxo para o *deploy* em produção
Fonte: Autoria própria retirada do repositório do projeto

3.1. Gerenciando Repositório com Github

Neste trabalho utilizaremos o Github para fazer o gerenciamento de todo código armazenado no repositório, também usaremos suas extensões como o Github Actions e o Github Pages.

Na Figura 5 temos um exemplo do *flow* realizado com o Github Actions para o ambiente de homologação, cada quadrado desses indica uma ação, que dentro delas possuem uma série de comandos definidos no nosso arquivo da automação da *pipeline.*

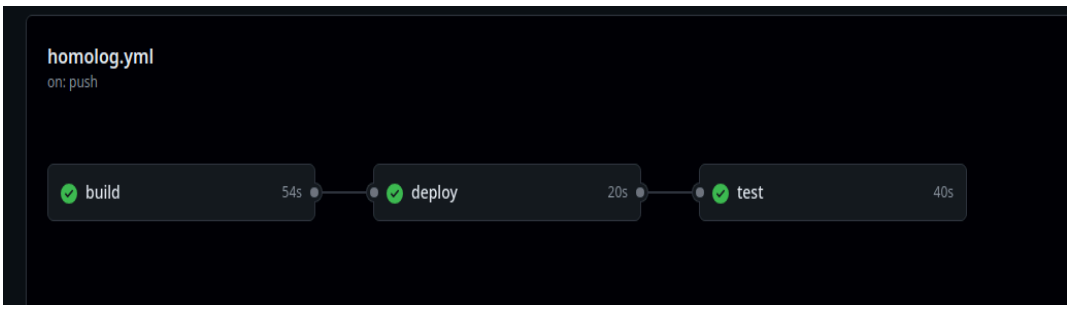

Figura 5: Imagem mostrando os passos da *pipeline* CI/CD do Github Actions
Fonte: Autoria própria retirada do repositório do projeto

Para tornar o processo mais fluido e seguro, todas as variáveis que englobam o processo estão armazenadas dentro do Github, elas são acessadas somente em tempo de execução e não temos como saber o valor delas. Dentro da *pipeline*, quando elas são invocadas o próprio Github se encarrega de mascarar elas para a segurança da execução, como observado na Figura 6:

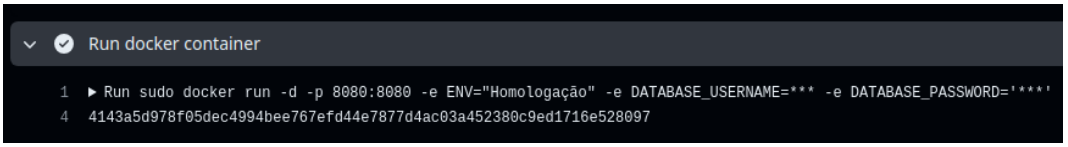

Figura 6: Imagem mostrando variáveis mascaradas em tempo de execução
Fonte: Autoria própria retirada do log de execução da *pipeline*

Na seção de configurações do repositório, temos as variáveis disponíveis que a *pipeline* utiliza, conforme observado na Figura 7, todas elas em algum momento serão acessadas.

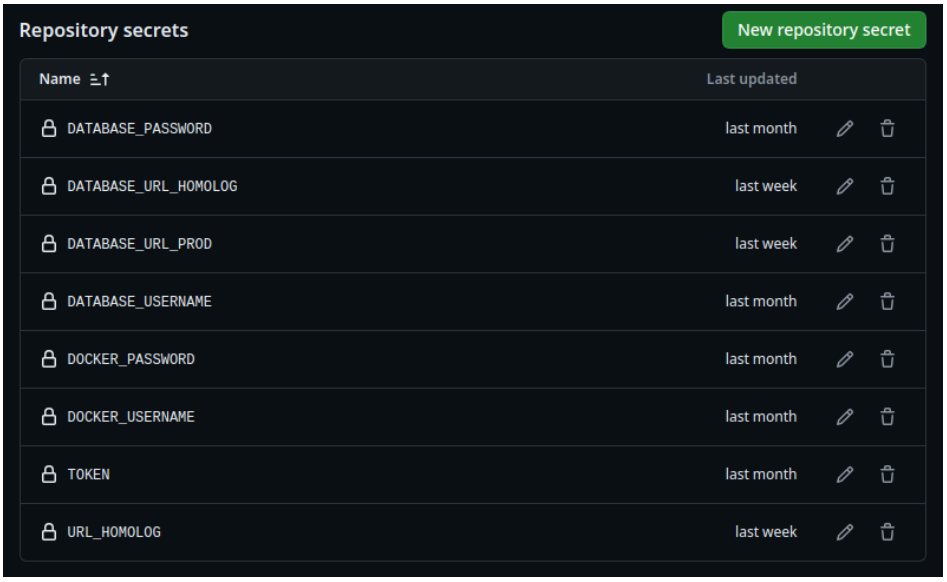

Figura 7: Imagem mostrando variáveis dentro do Github
Fonte: Autoria própria retirada do repositório do projeto

3.2. Banco Relacional PostgreSQL

Postgres é o responsável por armazenar todas informações que chegam na aplicação via *endpoints*[15] que estão expostos para consumo, tanto no ambiente de homologação quanto no de produção.

Ele possui uma versão encapsulada em uma imagem Docker, o que torna todo trabalho de configuração muito fácil, pois com apenas um comando você pode ter sua base de dados criada e já configura para conectar com a aplicação.

Como observado na Figura 8, temos o banco rodando dentro da instância na AWS e acessível externamente, isso é importante para podermos realizar validações na base diretamente da máquina local, não sendo necessário fazer via AWS.

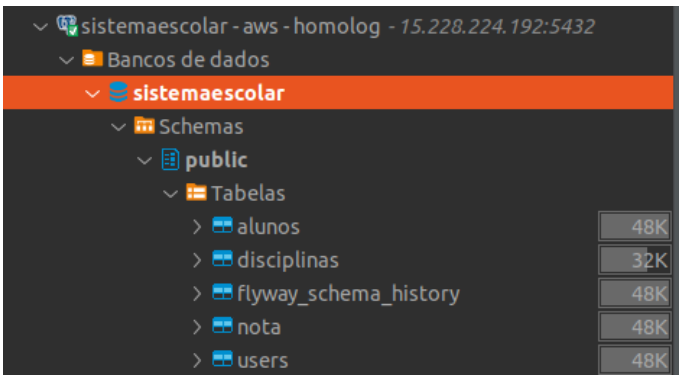

Figura 8: Imagem do banco postgre rodando na AWS e acessando via client
Fonte: Autoria própria retirada da tela do Dbeaver[16]

[15] Ponto de comunicação em uma rede ou API que permite que diferentes sistemas troquem informações.
[16] Ferramenta gerenciadora de banco de dados

Toda vez que criamos uma instância nova e subimos o banco juntamente com a aplicação, o Flyway irá ler o caminho destacado na Figura 9 e executará todos *scripts*. Caso seja um *deploy* da aplicação em uma base já existente, ele faz a comparação deste diretório com o que já existe na tabela *flyway_schema_history* e caso hajam novos *scripts*, os mesmos serão executados automaticamente, populando a tabela de controle do versionamento do Flyway, como destacado na Figura 10.

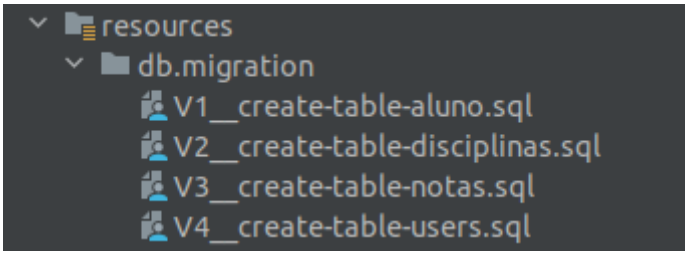

Figura 9: Imagem do repositório de *scripts* dentro do projeto Spring Boot
Fonte: Autoria própria retirada da tela da IDE IntelliJ

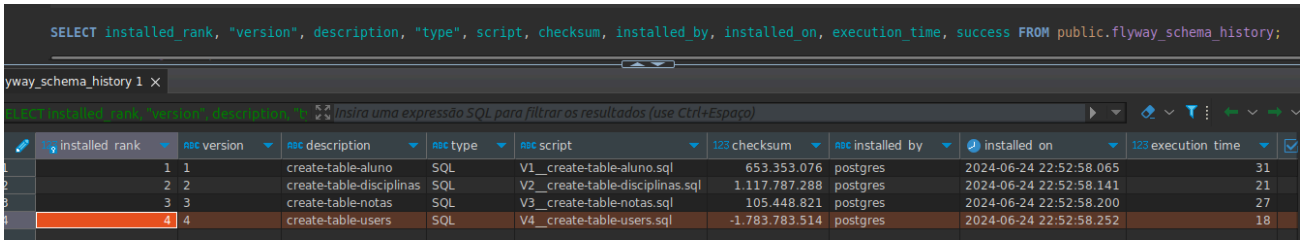

Figura 10: Imagem do banco postgres rodando na AWS e acessando via Dbeaver
Fonte: Autoria própria retirada da tela do Dbeaver

3.3. Análise Estática SonarQube

O SonarQube é amplamente reconhecido por sua capacidade robusta de análise estática de código, oferecendo uma interface *web* altamente organizada que categoriza os problemas detectados em diferentes seções. Embora a ferramenta seja gratuita, ela possui limitações significativas, como a restrição de realizar análise de código apenas em uma *branch* por vez, impedindo o escaneamento simultâneo de múltiplas *branches*. Portanto, para este trabalho, o escaneamento será restrito à nossa *branch* de homologação.

Na Figura 11, é apresentado o *dashboard* inicial do SonarQube, que exibe as críticas encontradas organizadas em seções distintas. Este *dashboard* proporciona uma visão clara das áreas do código que necessitam de ajustes ou remoções, facilitando a identificação e a resolução de problemas de qualidade e segurança no código.

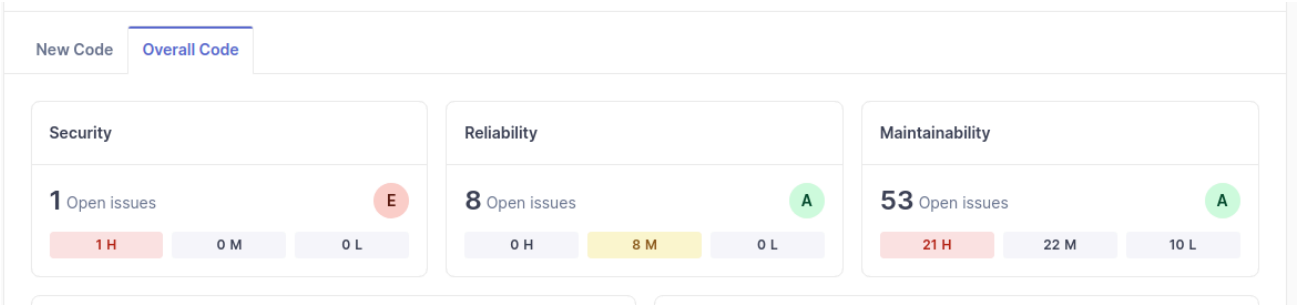

Figura 11: Imagem do *dashboard* inicial no SonarQube
Fonte: Autoria própria retirada da tela inicial do SonarQube

3.4. Aplicação Java Spring Boot e Maven

A combinação de Spring Boot e Maven nos proporciona uma série de possibilidades quando o assunto é desenvolver um sistema backend, o Spring Boot é um *framework* que já vem com um ferramental incluso na sua estrutura de dependências, fazendo assim muito simples a construção de API 's. O uso de anotações em seu código é muito evidente, pois a partir delas transformamos um método em um Controller por exemplo, que servirá como meio de entrada para as requisições armazenadas e consultadas no banco de dados.

O Maven é nosso gerenciador de pacotes, será responsável por controlar todas dependências e realizar o *build* do nosso executável, onde tem tudo que precisa para rodar com perfeição no ambiente da AWS juntamente com o Docker.

Também tem como responsabilidade a manipulação de variáveis de ambiente, na Figura 12, temos as variáveis necessárias para que o processo flua conforme o esperado, a nível de exemplo, a *base.url* vai ser populada com o endereço de onde a aplicação está rodando para que os testes batam nos serviços de forma correta, mostrado na Figura 13 a passagem dos parâmetros.

```
<properties>
    <java.version>17</java.version>
    <base.url>http://localhost</base.url>
    <allure.version>2.15.0</allure.version>
    <sonar.host.url>http://localhost:9000</sonar.host.url>
    <sonar.token>sqa_d84a6f28fd61c26f3c84f2e4e8985dcf01bbaa27</sonar.token>
</properties>
```

Figura 12: Imagem das variáveis definidas no Maven
Fonte: Autoria própria retirada da tela de configuração do Maven

```
- name: Run tests with Maven
  run: mvn clean test -Dgroups=integracao -Dbase.url=${{secrets.URL_HOMOLOG}}:8080
```

Figura 13: Imagem das variáveis passadas na execução do Maven
Fonte: Autoria própria retirada da tela do projeto

A aplicação desenvolvida para este trabalho com o Spring Boot é um sistema escolar com funcionalidades básicas. Temos autenticação via Bearer Token[17], a cada autenticação de um usuário específico, ele gera um *token* exclusivo para o uso na sessão até que expire o seu tempo, definido no projeto por 60 minutos. Também contamos com os *endpoints* de alunos, disciplinas, notas e histórico, para fazer operações de cadastros, consultas e deleções conforme necessário.

Na Figura 14 temos o Swagger da aplicação, ele permite que desenvolvedores definam a estrutura de suas API's[18] (*Application Programming Interface*) de forma clara e compreensível, tanto para humanos quanto para máquinas. A principal ferramenta do conjunto é o *Swagger UI*, uma interface *web* interativa que gera automaticamente a documentação da API e permite a execução de chamadas diretamente na interface, facilitando o teste e a compreensão dos endpoints disponíveis. Com o Swagger, é possível garantir que todos os aspectos de uma API sejam bem documentados e acessíveis, promovendo melhores práticas de desenvolvimento e integração (Swagger, 2024).

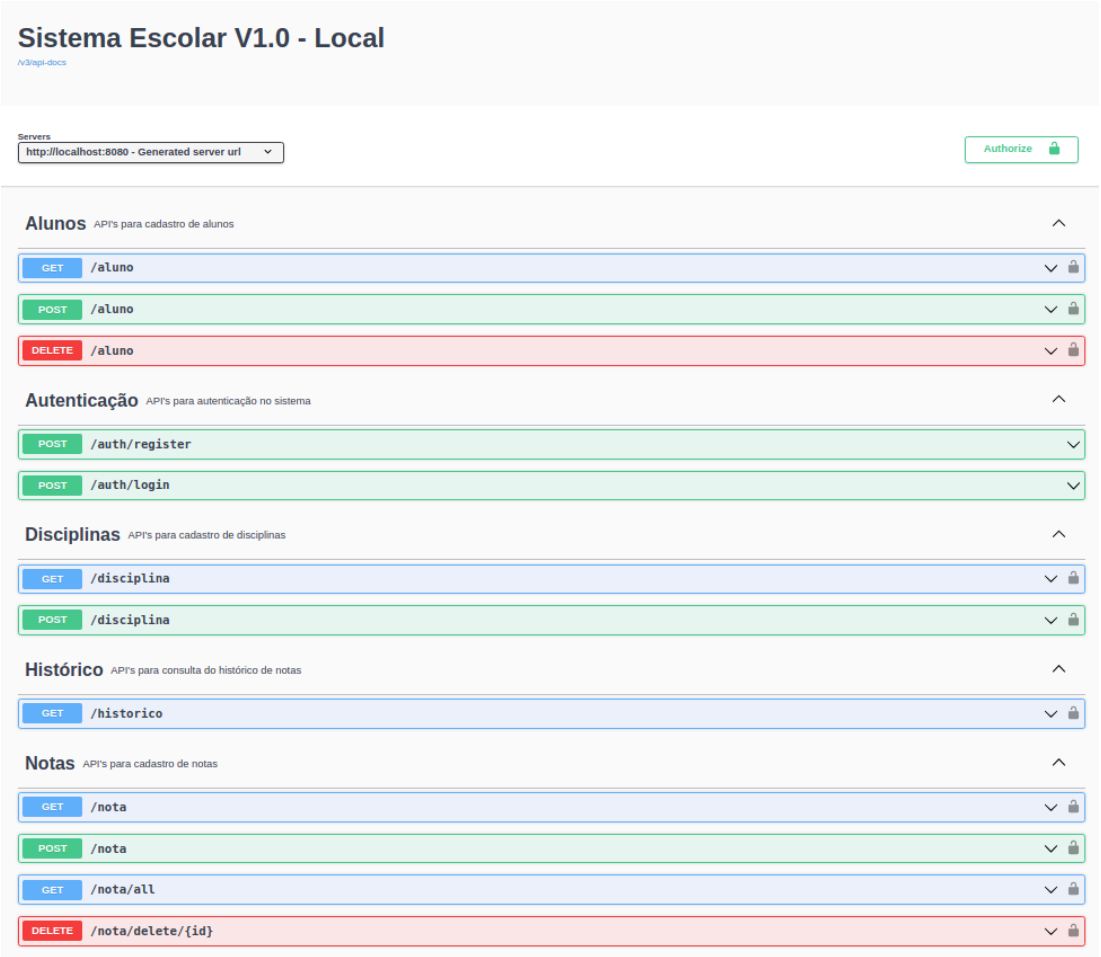

Figura 14: Imagem do Swagger da aplicação
Fonte: Autoria própria retirada da tela do projeto

[17] Autenticação utilizada para conceder acesso a recursos protegidos em *endpoints*, onde o portador do *token* é considerado autorizado.
[18] Conjunto de regras e protocolos que permite a comunicação entre diferentes *softwares*.

3.5. Deploy na EC2 da Amazon Web Services

O primeiro passo é a configuração dos ambientes de desenvolvimento e produção na AWS. Serão utilizadas instâncias EC2 para hospedar ambos os ambientes, sendo uma destinada a homologação e outra à produção. Essa estrutura permite testes rigorosos antes de qualquer atualização ser promovida para o ambiente de produção, assegurando maior segurança e confiabilidade (Amazon Web Services, 2023).

Como observado na Figura 15, temos os dois ambientes rodando separadamente, cada um em uma instância diferente para isolar os fluxos. As instâncias são do tipo *t2.micro,* o que significa que possuem um recurso discreto de 1CPU e 1GB de memória em seu *hardware.*

- **Ambiente de Homologação**: Utilizado para realizar testes e validações antes de enviar as mudanças para produção. Será configurado com os mesmos parâmetros do ambiente de produção para garantir que os testes sejam o mais próximo possível do real.

- **Ambiente de Produção**: Destinado a hospedar a aplicação final acessível aos usuários. A configuração será otimizada para performance e segurança, utilizando boas práticas recomendadas pela AWS.

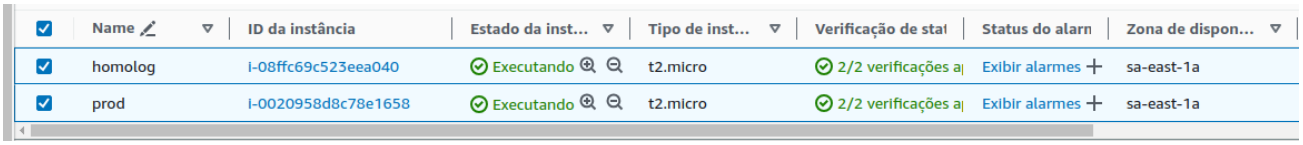

Figura 15: Imagem do dashboard com as instâncias rodando na AWS.
Fonte: Autoria própria retirada da tela inicial de instâncias EC2.

Precisamos também configurar nos servidores a liberação de portas, para que os acessos sejam liberados.

Como observado na Figura 16, está liberado exatamente o que a aplicação e o Github Actions precisa para interagir com a instância:

- **Porta 22 SSH**: porta utilizada para o gerenciamento do servidor, utilizada para instalar algum programa, por exemplo, ela está liberada somente com acesso via certificado.

- **Porta 80 HTTP**: porta usada para o acesso da aplicação via web, ela está liberada por padrão dentro das regras.

- **Porta 443 HTTPS**: porta usada para acesso seguro via web, é usada quando acessamos a aplicação via https na url.

- **Porta 5432 Postgres**: essa porta é utilizada para acessar o banco de dados, sendo que o PostgreSQL opera, por padrão, na porta 5432. É essencial liberar essa porta para possibilitar a comunicação entre o banco de dados e a aplicação que está sendo executada dentro do Docker.

- **Porta 8080 Aplicação**: a aplicação roda neste porta, então para que qualquer acesso ou chamada para os serviços expostos seja possível, é necessária a liberação desta porta.

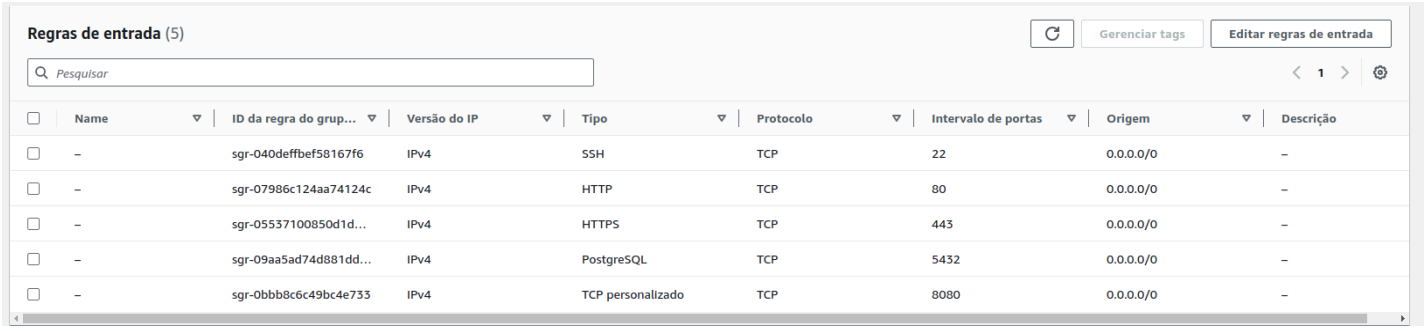

Figura 16: Imagem da relação de liberação de portas para tráfego interno.
Fonte: Autoria própria retirada da tela de gerenciamento da AWS

3.6. Evidências/Logs com Allure Reports

O Allure é responsável pela geração detalhada dos relatórios de execução dos testes. O JUnit, ao receber parâmetros apropriados, identifica quais testes devem ser executados. Após a execução, o GitHub Actions publica os arquivos gerados pelo Allure, proporcionando uma visão consolidada dos relatórios.

O Allure Reports é integrado ao GitHub Pages, permitindo a configuração e visualização dos relatórios de teste de maneira estruturada e acessível. Com a colaboração do GitHub Actions, a organização dos logs de teste é otimizada; cada execução do *pipeline* organiza de forma isolada os arquivos XML gerados, garantindo que a página inicial do Allure Reports exiba dados completos e atualizados.

Conforme ilustrado na Figura 17, a interface do Allure Reports organiza os testes em suítes, apresentando os testes executados de forma clara.

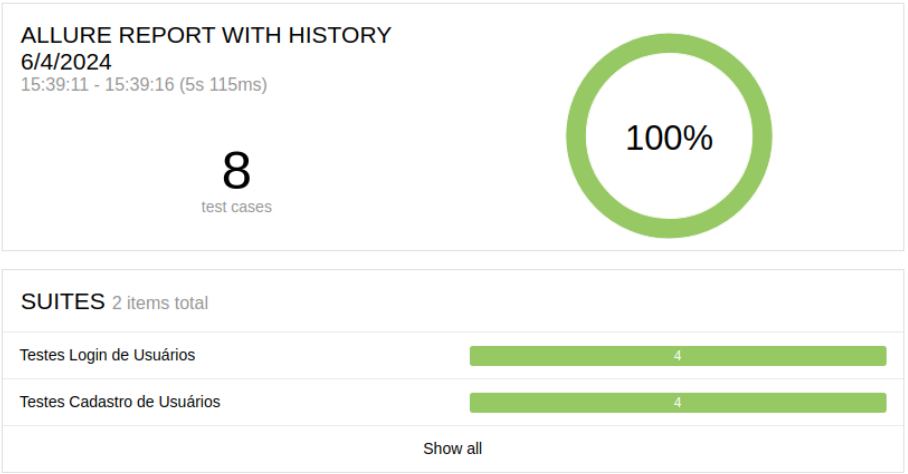

Figura 17: Imagem de parte do *dashboard* inicial do Allure Reports
Fonte: Autoria própria retirada da tela inicial da ferramenta

3.7. Triggers e Automação de Deploy

Foram configurados triggers nos arquivos .yml[19] para automatizar a execução dos *scripts* de *deploy* sempre que um *merge* é realizado nas *branches* de homologação e *main* (produção).

No caso da homologação, o *script* de automação inclui um conjunto mais extenso de comandos, dado que essa fase envolve um roteiro de validações mais detalhadas antes de promover as alterações para o ambiente de produção. Após a validação bem-sucedida das modificações pela equipe, é necessário abrir um *pull request* de homologação para a *branch main*. Esta ação aciona a automação de *deploy*, que então realiza a atualização no ambiente de produção.

A Figura 18 ilustra um exemplo da sintaxe utilizada para a configuração desses *triggers*, demonstrando como garantir que o processo de automação seja executado com sucesso.

```yaml
name: Deploy Application Homolog AWS EC2

on:
  push:
    branches: [homolog]
```

Figura 18: Imagem mostrando *trigger* da *branch* de homolog
Fonte: Autoria própria retirada da tela da IDE IntelliJ

3.8. Implementação do Pipeline com GitHub Actions

O *pipeline* é estruturado utilizando GitHub Actions, aproveitando sua integração nativa com os repositórios do GitHub. As etapas principais do *pipeline* incluirão:

- **Build**: Automatização da construção do código a partir dos commits realizados nos *branches* principais. Utilizamos o Docker para garantir que a construção do código seja consistente e reproduzível em diferentes ambientes. Nesta etapa também estamos fazendo *push* da imagem gerada para o Docker, na Figura 18 visualizamos a utilização do Spring Boot e Maven para o *build*.

[19] Arquivo de configuração que define os workflows de automação, incluindo etapas e ações para construir, testar e implantar o código no GitHub.

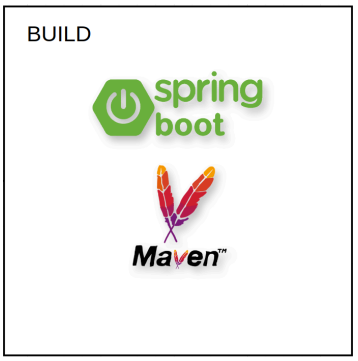

Figura 19: Imagem mostrando tecnologias usadas no *build*
Fonte: Autoria própria utilizado draw.io

Na Figura 20 detalhamos melhor cada etapa do processo de *build*, onde temos vários comandos executados em cadeia, cada um deles para um propósito específico, desde configurar o *node* de onde irá rodar, até o *push* da imagem para o Docker.

```yaml
build:
  runs-on: ubuntu-latest
  steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Setup Java
      uses: actions/setup-java@v3
      with:
        distribution: 'temurin'
        java-version: '17'
    - name: Build project
      run: mvn clean install -DskipTests
    - name: Login Docker Hub
      run: docker login -u ${{secrets.DOCKER_USERNAME}} -p ${{secrets.DOCKER_PASSWORD}}
    - name: Build docker image
      run: docker build -t williambreda/sistema-escolar .
    - name: Push image docker
      run: docker push williambreda/sistema-escolar
```

Figura 20: Imagem mostrando etapas do processo de *build*
Fonte: Autoria própria retirado da IDE IntelliJ

- **Deploy**: Esta etapa é responsável pela entrega da imagem gerada anteriormente, onde a nossa instância já preparada com todos os requisitos para a aplicação rodar, recebe os comandos para *pull* da imagem e subida da mesma no ambiente Docker. Na Figura 21, visualizamos a utilização do Docker, AWS, Postgres e Flyway para o processo de *deploy* na instância.

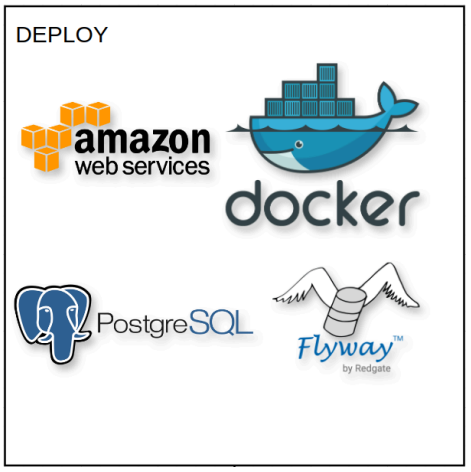

Figura 21: Imagem mostrando tecnologias usadas no *deploy*
Fonte: Autoria própria utilizado draw.io

Na Figura 22, são apresentados os comandos executados em sequência para garantir a realização bem-sucedida do *deploy* em uma instância EC2 na AWS. Esses comandos são executados diretamente na nossa instância, conforme especificado pela *tag* homolog em *runs-on*. No GitHub, associamos o executor deste ambiente com a mesma *tag*, garantindo que os comandos sejam direcionados precisamente para a instância de homologação designada.

```
deploy:
  needs: build
  runs-on: homolog
  steps:
    - name: Pull image from docker hub
      run: sudo docker pull williambreda/sistema-escolar:latest
    - name: Remove docker container
      run: sudo docker rm -f /sistema-escolar
    - name: Run docker container
      run: sudo docker run -d -p 8080:8080 -e ENV="Homologação" -e DATABASE_USERNAME=${{secrets.DATABASE_USERNAME}}
      -e DATABASE_PASSWORD='${{secrets.DATABASE_PASSWORD}}' -e DATABASE_URL=${{secrets.DATABASE_URL_HOMOLOG}}
      --name sistema-escolar williambreda/sistema-escolar
```

Figura 22: Imagem mostrando etapas do processo de *deploy*
Fonte: Autoria própria retirado da IDE IntelliJ

- **Testes Automatizados e Geração de Relatórios**: A execução de uma suíte de testes integração é realizada para garantir a integridade do código. O JUnit, comandado pelo gerenciador de pacotes Maven, é responsável pela execução dos testes. Paralelamente, o SonarQube realiza a análise estática do código. Os resultados dos testes são exibidos e armazenados pelo Allure Reports e GitHub Pages. A Figura 23 ilustra a integração entre JUnit, Allure Reports, SonarQube e GitHub Pages, destacando o fluxo de testes e a geração de análises estáticas.

Figura 23: Imagem mostrando tecnologias usadas nos testes
Fonte: Autoria própria utilizado draw.io

4. Resultados

Esta seção do artigo apresenta todos os aspectos que obtivemos como resultado, tendo métricas do custo e desempenho de nossa aplicação que está rodando na AWS.

4.1. Performance da Aplicação

O JMeter foi empregado para avaliar a performance do sistema por meio de testes de carga em serviços expostos. No nosso caso, o JMeter foi configurado para enviar requisições simultâneas para os três principais endpoints do sistema, responsáveis por retornar as listas de alunos, notas e disciplinas. Esta ferramenta permite a medição detalhada do *throughput*[20] e da carga de trabalho nos *endpoints* testados (Apache JMeter, 2024).

Os resultados obtidos foram excelentes, conforme evidenciado na Figura 24, onde cada serviço alcançou um *throughput* de quase 200 requisições por segundo, uma carga muito elevada para a finalidade do sistema. Apesar dessa alta volumetria, o aumento no uso da CPU da instância de homologação foi de apenas 50%, conforme mostrado na Figura 25. A taxa de erro de 0,05% corresponde a falhas em cerca de 1000 requisições, um número relativamente baixo considerando o estresse significativo da aplicação.

[20] Quantidade de requisições que a API pode processar por segundo.

Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename Browse... Log/Display Only: Errors Successes Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
getAlunos	67056	172	0	14721	199.86	0.05%	192.9/sec	99.42	68.55	527.7
getNotas	67024	174	0	15229	214.13	0.05%	192.9/sec	95.26	69.09	505.7
getDisciplinas	66992	171	0	7546	187.15	0.05%	192.8/sec	101.84	69.44	540.9
TOTAL	201072	172	0	15229	200.69	0.05%	578.5/sec	296.47	207.04	524.8

Figura 24: Imagem mostrando tabela de informações do teste de carga
Fonte: Autoria própria retirada da tela de sumário do Jmeter

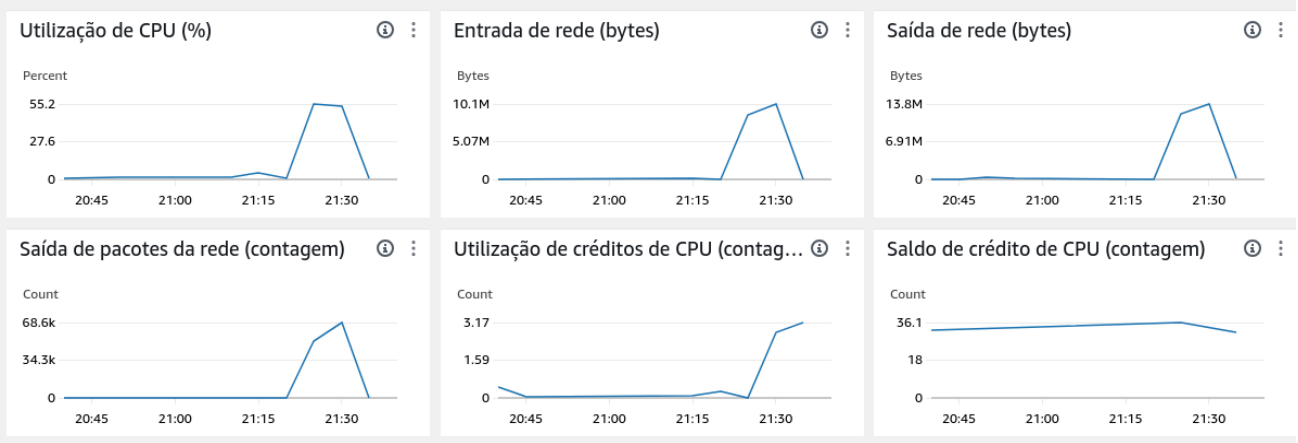

Figura 25: Imagem mostrando recursos da instância na AWS
Fonte: Autoria própria retirado da tela gerenciador de instâncias da AWS

4.2. Custos do Ferramental Utilizado

Após a conclusão das etapas e ferramentas descritas neste trabalho, passamos a considerar os aspectos relacionados aos custos, que são de particular interesse para os leitores. O único fator gerador de custos foi a utilização da AWS, uma vez que a instância está hospedada em uma infraestrutura robusta e confiável. A AWS oferece uma ampla gama de serviços que podem ser acionados conforme necessário, o que contribui para o custo associado à execução e manutenção da instância.

O restante das ferramentas, todas são *open-source* ou possuem uma versão gratuita limitada, como exemplo o SonarQube, na sua versão gratuita, conseguimos analisar e reter o histórico somente de uma *branch*, na sua versão paga, consegue analisar várias *branchs* e repositórios.

Na Figura 26 temos o custo de AWS desde o início deste trabalho, os primeiros testes foram implementados em fevereiro, porém como temos uma cota de 700 horas de EC2 gratuito, não gera cobrança nos primeiros meses (AWS, 2024).

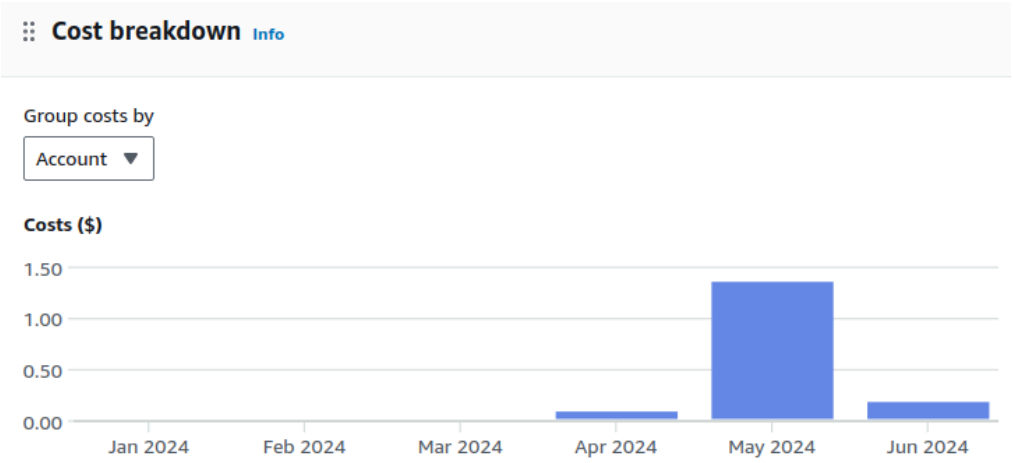

Figura 26: Imagem mostrando o custo que tivemos com EC2 da AWS
Fonte: Autoria própria retirado da tela de faturamento da AWS

5. Conclusão

Este artigo ilustra a eficácia de uma *pipeline* CI/CD ao demonstrar como o processo pode ser automatizado para permitir a homologação e a entrega de aplicações em ambiente produtivo com apenas alguns cliques, por meio de um fluxo robusto e bem definido. A integração entre Spring Boot, GitHub Actions e AWS proporcionou uma solução abrangente e eficiente para o *deploy* contínuo de aplicações, alinhando-se às melhores práticas de DevOps. Esse processo não apenas otimiza o ciclo de desenvolvimento, mas também aprimora a qualidade e a confiabilidade do *software* entregue.

Pesquisas futuras podem investigar a integração de outras ferramentas e serviços, bem como estratégias para a otimização de recursos em projetos de maior escala. Além disso, o custo associado ao uso dessas tecnologias é relativamente baixo, aproveitando ferramentas open source e uma infraestrutura robusta na AWS, garantindo a hospedagem da aplicação de maneira segura e eficiente.

Referências

DECAN, Alexandre et al. **On the use of github actions in software development repositories**. In: 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2022. p. 235-245.

Amazon Web Services. (2020). Amazon EC2 Documentation. Available at: https://docs.aws.amazon.com/ec2/

Amazon Web Services (2023). AWS Documentation. Available at: https://aws.amazon.com/documentation/.

Campbell, P., & Papapetrou, P. (2013). SonarQube in Action. Manning Publications.
Decan, A. (2022). Empirical Study on the Use of GitHub Actions in Software Development. Journal of Empirical Software Engineering.

Duvall, P. M., Matyas, S., & Glover, A. (2007). Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley Professional.

Fowler, M. (2006). Continuous Integration. Available
Fowler, M. (2006). Continuous Integration. Available at: https://martinfowler.com/articles/continuousIntegration.html

GitHub. (2020). GitHub Actions Documentation. Available at: https://docs.github.com/en/actions

Postgres. (2024). PostgreSQL Documentation. Available at: https://www.postgresql.org/docs/

Humble, J., & Farley, D. (2010). Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional.

Johnson, R. (2014). Spring Framework Documentation. Available at: https://spring.io/projects/spring-boot
Kochetkov, S. (2017). Allure Report Documentation. Available at: http://allure.qatools.ru

Turnquist, J. (2017). Learning Spring Boot 2.0: Simplify the Development of Lightning-Fast Applications Based on Microservices and Reactive Programming. Packt Publishing.

Walls, C. (2016). Spring Boot in Action. Manning Publications.

Merkel, D. (2014). Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux Journal. Recuperado de Linux Journal.

Pahl, C. (2015). Containers and Clusters for Edge Cloud Architectures – A Technology Review. Springer. Recuperado de Springer.

Turnbull, J. (2014). The Docker Book: Containerization is the New Virtualization. Recuperado de Docker Book.

Docker Documentation. (2024). Disponível em: https://docs.Docker.com/. Acesso em: 20 de junho de 2024.

Allure Framework. (2024). Allure Report. Recuperado de https://docs.qameta.io/allure/.

Amazon Web Services. (2024). Free Tier FAQs. Recuperado de

https://aws.amazon.com/pt/free/faqs/.

SonarQube Documentation. (2024). Disponível em https://docs.sonarsource.com/sonarqube/latest/.

Apache Maven. (2024). Introduction to the POM. Retrieved from https://maven.apache.org/guides/introduction/introduction-to-the-pom.html

Heffelfinger, D. J. (2014). Java EE 7 Development with WildFly. Packt Publishing Ltd.

Kousen, K. (2019). Modern Java Recipes: Simple Solutions to Difficult Problems in Java 8 and 9. O'Reilly Media, Inc.

Apache JMeter. (2024). *Apache JMeter*. Retrieved from https://jmeter.apache.org/

Swagger. (2024). Swagger. Retrieved from https://swagger.io/