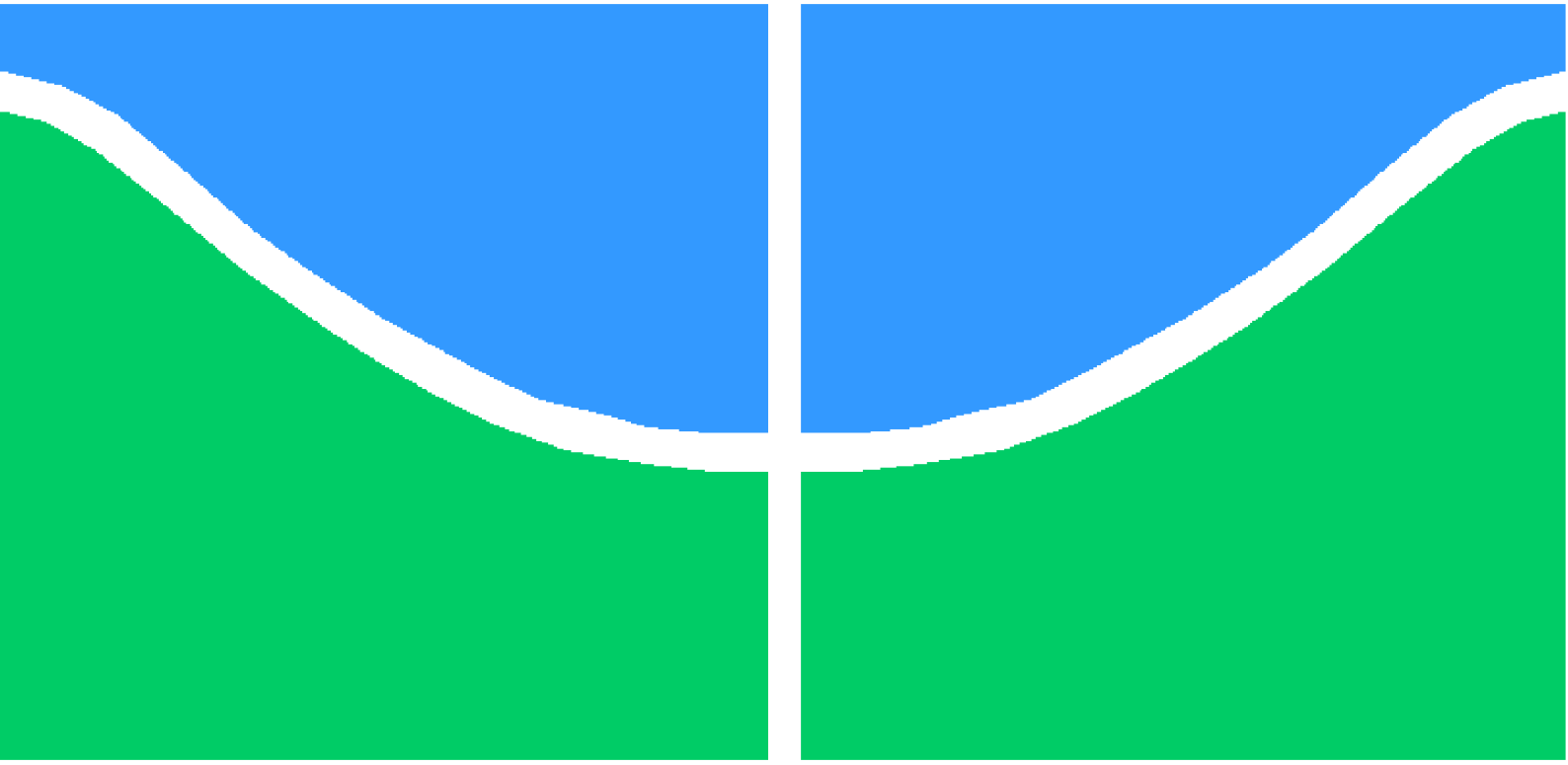

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de *Software*

Automatização da *pipeline* de testes entre um aplicativo móvel e um sistema embarcado com comunicação *Bluetooth*

Autor: Calebe Rios de Sousa Mendes

Orientador: Prof. Dr. Renato Coral Sampaio

Brasília, DF

2021

Calebe Rios de Sousa Mendes

Automatização da *pipeline* de testes entre um aplicativo móvel e um sistema embarcado com comunicação *Bluetooth*

Monografia submetida ao curso de graduação em (Engenharia de *Software*) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de *Software*).

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Renato Coral Sampaio

Brasília, DF

2021

Calebe Rios de Sousa Mendes

Automatização da *pipeline* de testes entre um aplicativo móvel e um sistema embarcado com comunicação *Bluetooth*/ Calebe Rios de Sousa Mendes. – Brasília, DF, 2021-

106 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Renato Coral Sampaio

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2021.

1. DevOps. 2. Bluetooth. I. Prof. Dr. Renato Coral Sampaio. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Automatização da *pipeline* de testes entre um aplicativo móvel e um sistema embarcado com comunicação *Bluetooth*

CDU 02:141:005.6

Calebe Rios de Sousa Mendes

Automatização da *pipeline* de testes entre um aplicativo móvel e um sistema embarcado com comunicação *Bluetooth*

Monografia submetida ao curso de graduação em (Engenharia de *Software*) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de *Software*).

Trabalho aprovado. Brasília, DF, 09 de novembro de 2021.

Prof. Dr. Renato Coral Sampaio
Orientador

Prof. Dr. Tiago Alves da Fonseca
Convidado 1

Prof. Dr. Jones Yudi Mori Alves da Silva
Convidado 2

Brasília, DF
2021

Resumo

O mundo está cada vez mais conectado na rede da Internet, o que demanda *softwares* confiáveis com atualizações rápidas. Com o crescimento de dispositivos conectados com a Internet das Coisas, a utilização do *Bluetooth* para controlar sistemas embarcados torna-se mais comum, só que para entregar um *software* confiável a um usuário final é necessário garantir a integração entre o *software* e o sistema embarcado. Por conseguinte, as empresas têm utilizado o DevOps para conseguir automatizar esses testes de comunicação *Bluetooth*. Inicialmente, foi feita uma revisão bibliográfica para levantar estudos e métodos a fim de desenvolver uma bancada de testes automatizada entre um aplicativo móvel e a fechadura eletrônica da empresa LoopKey. O objetivo da bancada é garantir versões estáveis, do aplicativo móvel, para os usuários finais de acordo com a principal funcionalidade do aplicativo: envio de comando *Bluetooth*. A ferramenta de Integração Contínua utilizada foi o Jenkins, por ser uma ferramenta livre e *open-source*. O projeto conta com testes unitários para garantir a estabilidade.

Palavras-chaves: DevOps. *Bluetooth*. BLE. Testes. Sistema Embarcado. Desenvolvimento *Mobile*. LoopKey.

Abstract

The world is more connected with the internet, and this demands more reliable software with quick updates. With the growth of devices connected to the Internet of Things, the use of Bluetooth to control embedded systems has become more common. To deliver reliable software to end-user it is necessary to guarantee the integration of the software and embedded system. Because of this, companies have used DevOps to be able to automate the integration tests of Bluetooth communication. Was made a bibliographic revision to search studies and methods to develop an automat test stand between a smartphone and an electronic lock of LoopKey's company. The objective of the stand is to guarantee stables versions to end-users and the work of the main functionality of the app: send a Bluetooth command. The tool of Continuous Integration used in this project was Jenkins because it is free and open-source. The project has unit tests.

Key-words: DevOps. Bluetooth. BLE. Testes. Sistema Embarcado. Desenvolvimento Mobile. LoopKey.

Lista de ilustrações

Figura 1 – Arquitetura do BLE. 29
Figura 2 – Exemplos de *pipelines* de DevOps (LWAKATARE et al., 2019, p. 222) 33
Figura 3 – Arquitetura do Android (Android, 2020a) 35
Figura 4 – Arquitetura dos Produtos LoopKey 38
Figura 5 – Exemplo das colunas do *kanban* 45
Figura 6 – A *pipeline*. 54
Figura 7 – A bancada. 57
Figura 8 – Dashboard com apontamento para Gerenciar Jenkins. 61
Figura 9 – Configuração do JDK. 61
Figura 10 – Selecionar tipo de projeto. 62
Figura 11 – Configuração do GitHub. 63
Figura 12 – Configuração a credencial do GitHub. 63
Figura 13 – Tela de *access token* do GitHub. 64
Figura 14 – Tela de gerar uma nova *access token*. 64
Figura 15 – Tela de salvar um *access token*. 65
Figura 16 – Tela de selecionar o modo de interpretar a *pipeline*. 65
Figura 17 – Configuração do ngrok. 67
Figura 18 – Lisa de *webhooks* configurados. 67
Figura 19 – Configuração do *webhook*. 68
Figura 20 – Opção dos gatilhos do *webhook*. 68
Figura 21 – Gatilhos do *webhook*. 69
Figura 22 – *Checks* do *Pull Request* executando. 72
Figura 23 – *Checks* do *Pull Request* completos. 72
Figura 24 – A *pipeline* funcionando com sucesso. 73
Figura 25 – Os *logs* do *build*. 73
Figura 26 – O *log* dos testes. 74
Figura 27 – Erro no Build. 75
Figura 28 – Erro na folha de estilo. 75
Figura 29 – *Log* de erro do *build*. 76
Figura 30 – Erro na etapa do *owner*. 77
Figura 31 – Erro na etapa do *admin*. 78
Figura 32 – Erro na etapa do *guest online*. 78
Figura 33 – Erro na etapa do *guest offline*. 79
Figura 34 – Erro na etapa do *owner final*. 80
Figura 35 – Resumo da *pipeline* do *cronjob*. 80
Figura 36 – *Backlog* do Produto 93

Lista de tabelas

Tabela 1 – Resultados da Medição. 82

Lista de quadros

Quadro 1 – Categorias de Teste de Sistema (MYERS; BADGETT; SANDLER, 2012, p. 122). 26
Quadro 2 – Tipos de Permissão vs Tipos de Usuários. 37
Quadro 3 – Especificações técnicas da LK Bio e da LK Mini. 39
Quadro 4 – Principais Funcionalidades dos Aplicativos 40
Quadro 5 – Classificação da Pesquisa de TCC. 41
Quadro 6 – Quadro de custos. 47
Quadro 7 – Métricas. 48
Quadro 8 – 5W2H. 52
Quadro 9 – Requisitos Funcionais. 53
Quadro 10 – Requisitos Não Funcionais. 53
Quadro 11 – Ordem dos testes. 55
Quadro 12 – As suites de testes. 59
Quadro 13 – Experimentos realizados. 71
Quadro 14 – Principais comandos BLE que não foram testados. 83

Lista de abreviaturas e siglas

adb.	*Android Debug Bridge*
API.	Interface de Programação de Aplicações
APK.	*Android Application Pack*
ART.	Android *Runtime Runtime*
BLE.	*Bluetooth Low Energy*
CD.	*Continuos Delivery*
CI.	*Continuos Integration*
CTO.	*Chief Technical Officer*
DoS.	*Denial of Service*
FGA.	Faculdade do Gama
GAP.	*Generic Access Profile*
GATT.	*Generic Attribute Profile*
GC.	*Garbage Collector*
HAL.	*Hardware Abstraction Layer*
HCI.	*Host Controller Interface*
IaS.	*Infrastructure as Code*
IDE.	Ambiente de Desenvolvimento Integrado
IoT.	Internet das Coisas
JDK.	*Java Development Kit*
JVM.	*Java Virtual Machine*
L2CAP.	*Logical Link Control and Adaptation Protocol*
LK.	LoopKey
OTA.	*Over-the-Air*

PHY.	*Physical Layer*
PO.	*Product Owner*
PR.	*Pull Request*
SM.	*Security Manager*
SO.	Sistema Operacional
TDD.	*Test Driven Development*
UI.	*User Interface*
UnB.	Universidade de Brasília
US.	*User Story*
XP.	*eXtreme Programming*

Sumário

Introdução **21**

1 FUNDAMENTAÇÃO TEÓRICA **23**
1.1 Testes **23**
1.1.1 Sucesso vs Falha 24
1.1.2 Estratégias de Teste 24
1.1.3 Tipos de Testes 25
1.2 *Bluetooth* **27**
1.2.1 *Bluetooth Low Energy* 27
1.2.2 Periférico vs Central 28
1.2.3 Arquitetura do *Bluetooth* 28
1.2.4 Segurança 30
1.3 DevOps **30**
1.3.1 Práticas e Ferramentas do DevOps 32
1.3.1.1 Integração Contínua 32
1.3.1.2 Entrega Contínua e *Deployment* Contínuo 32
1.3.2 *Pipelines* de DevOps 33
1.4 Desenvolvimento *Mobile* **34**
1.4.1 Android 34
1.4.1.1 Arquitetura do Android 34
1.4.1.2 Testes no Android 36
1.5 LoopKey **37**
1.5.1 Arquitetura dos Produtos 37
1.5.1.1 Tipos de permissão 37
1.5.1.2 LK Bio e LK Mini 38
1.5.1.3 Servidor 39
1.5.1.4 *Gateway* 39
1.5.1.5 Sistema *Web* 39
1.5.1.6 Aplicativo 39

2 METODOLOGIA **41**
2.1 A Pesquisa **41**
2.2 Estudo do Problema **42**
2.3 Revisão Bibliográfica **42**
2.4 Metodologia de Desenvolvimento **43**
2.4.1 *Scrum* 43

2.4.2 *eXtreme Programming* . . . 43
2.4.3 *Kanban* . . . 45
2.4.4 Metodologia do Projeto . . . 45
2.5 Ferramentas e Serviços de Desenvolvimento . . . 45
2.5.1 Clubhouse . . . 46
2.5.2 Git . . . 46
2.5.3 Jenkins . . . 46
2.6 *Gitflow* da LoopKey . . . 46
2.7 Custos . . . 47
2.8 Coleta e Análise de Resultados . . . 47
2.8.1 Plano de Medição . . . 48
2.8.1.1 Métrica 1 - Cobertura dos *status* dos testes . . . 48
2.8.1.2 Métrica 2 - Cobertura das principais funcionalidades . . . 48
2.8.1.3 Métrica 3 - Automatização dos Testes ao Abrir PR . . . 49
2.8.1.4 Métrica 4 - Testes de Comando BLE . . . 49

3 PROPOSTA . . . 51
3.1 Requisitos . . . 52
3.2 A *Pipeline* . . . 53
3.2.1 Estrutura dos Testes . . . 55

4 DESENVOLVIMENTO . . . 57
4.1 Testes . . . 58
4.2 Jenkins . . . 60
4.2.1 Configuração de variável de ambiente . . . 60
4.2.2 Configuração do projeto . . . 62
4.2.3 Jenkinsfile . . . 66
4.3 Integração com o GitHub . . . 67

5 EXPERIMENTOS E RESULTADOS . . . 71
5.1 *Pipeline* Completa sem Erro . . . 73
5.2 *Pipeline* Completa com Erro no *Build* . . . 74
5.3 Erro no *Owner* . . . 76
5.4 Erro no *Admin* . . . 77
5.5 Erro no *Guest Online* . . . 78
5.6 Erro no *Guest Offline* . . . 79
5.7 Erro no *Owner* Final . . . 79
5.8 *Cronjob* . . . 80

6 DISCUSSÕES . . . 81
6.1 Experimentos . . . 81

6.2 Métricas 82

7 CONSIDERAÇÕES FINAIS 85
7.1 Trabalhos Futuros 85

REFERÊNCIAS 87

APÊNDICES 91

APÊNDICE A – *BACKLOG* DO PRODUTO 93

APÊNDICE B – ESTRUTURA DA REGRA DE *RETRY* 95

APÊNDICE C – ESTRUTURA DA BASEUNITTEST 97

APÊNDICE D – ESTRUTURA DOS TESTES DE COMANDO . . 99

APÊNDICE E – JENKINSFILE 101

Introdução

A utilização da tecnologia no dia a dia das pessoas tem crescido, principalmente a utilização de *smartphones* e sistemas embarcados, por exemplo: lâmpadas inteligentes, robô aspirador, fechaduras eletrônicas, entre outros. Os produtos que utilizam um sistema embarcados têm uma dificuldade de atualização e de garantia do pleno funcionamento de uma versão.

Muitos sistemas embarcados ainda não permitem a atualização *Over-the-Air* (OTA), em é possível fazer a atualização por meio do *Bluetooth* ou *Wifi*. Portanto são necessários vários testes antes de enviar para o usuário final. Esses sistemas embarcados não têm a mesma facilidade que um aplicativo para dispositivo móvel de receber atualizações e de realizar testes constantes, por isso são necessários testes antes de enviar esse tipo de produto para a venda ou até para atualização.

O emprego de testes é importante, no contexto da Engenharia de *Software*, para achar erros e defeitos de uma aplicação (MYERS; BADGETT; SANDLER, 2012). Os testes feitos nesses sistemas normalmente são manuais e demandam tempo e esforço de uma equipe dedicada. Além dos testes manuais, também são utilizados programas de simulação para testar o produto, mas a simulação não leva em conta o fator mecânico ou humano na hora do teste.

Com vistas a um aproveitamento de tempo e esforço, o emprego das técnicas de DevOps – *mindset* de mudança que utiliza prática com ferramentas automatizadas a fim de garantir a integridade e o lançamento de novas versões – facilita a realização desses testes, de forma automática, para, assim, garantir versões estáveis.

Como os testes são onerosos, demandam tempo e esforço e consomem a maior parte do tempo de uma *release* (WIKLUND et al., 2017), a utilização de práticas de DevOps, como a Integração Contínua (CI), facilita essa entrega do *software* ao automatizar esse processo. Mas a aplicação das técnicas de DevOps, em sistemas embarcados, ainda não é tão fácil, pela dificuldade de simular o sistema ou de instalar novas versões (EBERT et al., 2016).

A LoopKey é uma empresa que trabalha com controle de acesso e possui fechaduras eletrônicas próprias como seu principal serviço. Quando é entregue uma nova versão para o usuário, fica difícil garantir o pleno funcionamento dela já que os testes manuais são os únicos praticados.

Pensando nisso, foi levantada a possibilidade de montar uma bancada física para garantir integridade dos sistemas levando em consideração um ambiente real. Com a

ferramenta de CI, é possível realizar um conjunto de testes, de forma automática, sempre que tiver alteração no código do projeto.

Objetivo

O principal objetivo deste Trabalho de Conclusão de Curso (TCC) é projetar e implementar uma bancada de testes automatizados, utilizando ferramenta de CI, entre um dispositivo móvel e a fechadura eletrônica da LoopKey que se comunicam por *Bluetooth* para garantir o funcionamento do aplicativo. A fim de alcançar esse objetivo é necessário:

- levantamento dos testes necessários;
- levantamento dos requisitos da bancada;
- configuração do Jenkins como ferramenta de CI;
- configuração da integração entre o Jenkins e o GitHub;
- montagem da bancada física com todos os itens necessários.

Este TCC apresenta um referencial teórico com informações detalhadas dos principais assuntos abordados (Testes, *Bluetooh*, DevOps, Desenvolvimento *Mobile* e a empresa LoopKey). A metodologia traz informações de como está sendo desenvolvido o trabalho, bem como métodos, ferramentas e serviços, além do plano de medição do projeto. Já na proposta, há a definição do que pretende este TCC, assim como os requisitos e a *pipeline* utilizados. No desenvolvimento, encontram-se a configuração do ambiente, bem como as etapas necessárias para que funcione o projeto. Os experimentos têm os resultados obtidos a partir de todos os testes realizados. Consta-se ainda uma discussão sobre os principais resultados dos experimentos e as métricas coletadas. Nas considerações finais são apresentados os resultados do TCC, levando em conta todos os objetivos listados.

1 Fundamentação Teórica

Esse capítulo traz uma contextualização necessária para o desenvolvimento de testes automatizados com comunicação *Bluetooth*. Apresenta também um estudo sobre testes de *software*, definições e suas práticas, além de uma explicação sobre a tecnologia *Bluetooth*, desde a sua arquitetura até o funcionamento da comunicação. Em seguida, trata sobre o DevOps, explicando este termo e suas práticas. Faz também um estudo sobre o desenvolvimento *mobile* com explicação da arquitetura, testes e *Bluetooth* do Android. Por fim, finaliza com uma apresentação da empresa LoopKey e seus produtos, principalmente sobre a fechadura eletrônica utilizada para o desenvolvimento deste trabalho.

1.1 Testes

Inicialmente os *softwares* desenvolvidos eram simples e de difícil acesso a todos, já que não existiam tantos computadores pessoais como hoje em dia. Mas com a evolução ao longo dos anos, passaram a ser algo complexo e de fácil acesso. O desenvolvimento de *software* hoje envolve mais do que programação; engloba também o levantamento de requisitos, a documentação, os testes e o levantamento de arquitetura, entre outros pontos.

Nas últimas décadas, os *softwares* passaram de uma ferramenta utilizada somente por máquinas para um programa necessário ao dia a dia das pessoas. Essa normatização do *software* é necessária para seu pleno funcionamento, garantindo que ele irá fazer o que se espera, evitando algo inesperado. O *software* tem que ser previsível e consistente (MYERS; BADGETT; SANDLER, 2012, p. 2).

Quando o *software* faz algo inesperado, pode-se dizer que ocorreu um erro de *software*. Por isso fazer testes antes de mandar o programa para os usuários finais é o recomendado, pois testar é o processo de executar um programa com a intenção de achar erros (MYERS; BADGETT; SANDLER, 2012, p. 6). Mesmo utilizando estratégias conhecidas de *software* e sendo criativo, é impossível ter um programa sem erro, totalmente (MYERS; BADGETT; SANDLER, 2012, p. 41).

Fazer testes de *software* é garantir e controlar a sua qualidade, sendo um trabalho em conjunto com diversas equipes do projeto (EBERT et al., 2016, p. 99). Os testes são definidos pelas organizações de acordo com as especificações do *software* e dos requisitos dos usuários (MATHRANI; MATHRANI, 2013).

1.1.1 Sucesso vs Falha

Os significados de sucesso e de falha dentro de testes de *software* são normalmente mal interpretados pela maioria dos times (MYERS; BADGETT; SANDLER, 2012, p. 7). O sucesso de um teste é quando ele consegue encontrar um erro, independente se o programa passou ou não em um teste. E a falha de um teste significa que ele não conseguiu cumprir o seu papel em achar erros e vulnerabilidade no programa.

Testes recorrentes, muito utilizados em metodologias ágeis, objetivam o sucesso em todos os testes – também conhecido como veredito do teste (AMANNEJAD; BETIN, 2015), a fim de garantir um *software* mais confiável após inserções ou modificações de funcionalidades.

1.1.2 Estratégias de Teste

Os testes de *software* possuem três estratégias definidas para conseguir abordar e cobrir o máximo do escopo do programa. São elas:

1. **Caixa Preta**: a estratégia da caixa preta, também conhecida como *data-driven* ou *input/output-driven* (MYERS; BADGETT; SANDLER, 2012, p. 8), são testes que levam em conta somente as entradas e saídas do *software*, de acordo com as suas especificações, sem se importar com a lógica.

   Para esse tipo de teste é levantado um conjunto de entradas e suas respectivas saídas para certifica-se de que o programa está fazendo o que deveria fazer. A dificuldade desse teste é cobrir a maior possibilidade dos casos de teste possíveis.

2. **Caixa Branca**: outra estratégia conhecida é o da caixa branca, ou *logic-drive*. Diferente da caixa preta, essa estratégia lida como se o *software* fosse uma caixa aberta e faz testes na lógica do *software*.

   Os casos de teste dessa estratégia são feitos pensando nas possibilidades da funcionalidade que está sendo testada. Assim como na estratégia da caixa preta, essa estratégia também tem que se preocupar com a quantidade de testes a ser criados.

3. **Teste Humano**: a última estratégia de testes é o *human test*. Enquanto as estratégias da caixa preta e da caixa branca são testes feitos no computador, essa estratégia é utilizada de forma manual. O *humam test* é basicamente a utilização de grupos de pessoas utilizando o *software* para detectar erros e melhorias. Essa estratégia possui quatro técnicas: Inspeção, Testes por Travessia, Testes de Mesa e Avaliação por Pares.

De acordo com Myers, Badgett e Sandler (2012, p. 42), "O procedimento padrão é utilizar primeiro os testes de caixa preta, e depois, se necessário, complementar com os testes de caixa branca".

1.1.3 Tipos de Testes

Os testes de *software* podem ser divididos em três categorias, e cada uma é utilizada para testar partes diferentes do *software*:

1. **Testes Unitários**: os testes unitários, conhecidos também por modulares, são utilizados para testar um subprograma, uma sub-rotina, uma classe ou um procedimento de um programa (MYERS; BADGETT; SANDLER, 2012, p. 85). Mais especificamente, os testes são utilizados para identificar erros do *software* testando módulo por módulo. Os testes modulares são amplamente orientados pela estratégia da caixa branca.

   Para os testes unitários, são necessárias duas informações: as especificações do módulo (entrada e saída) e o código fonte do programa (MYERS; BADGETT; SANDLER, 2012, p. 86).

   Os testes modulares podem ser divididos em duas categorias:

   - Testes Incrementais: fazem os testes dos módulos de forma incremental, seja de forma *top-down* ou *bottom-up*
   - Testes Não Incrementais: são realizados de forma individual, para isso são utilizados *stubs* para simular o resultado de outros módulos.

   De acordo com Myers, Badgett e Sandler (2012, p. 101), "Os testes incrementais são superiores aos não incrementais.", pois os testes incrementais levam em consideração como outro módulo vai retornar uma resposta. Em contrapartida, nos testes não incrementais, quando ocorre uma alteração em algum módulo, é necessário alterar o seu *stub* para que funcione como esperado.

2. ***High Order Testing***: os *high order* testes são estruturados para se moldar ao ciclo de desenvolvimento; sendo assim, cada etapa do desenvolvimento de uma funcionalidade/produto pode ser testado.

   Os testes de *high-order* são descritos a seguir – os testes de integração estão implícitos nos testes modulares incrementais:

   - Teste de Função: os testes de função servem para mostrar que o programa não funciona de acordo com as especificação externas (MYERS; BADGETT; SANDLER, 2012, p. 116). Os testes de função cobrem as funções que os testes modulares não cobrem. Eles são testes de caixa preta.

Quadro 1 – Categorias de Teste de Sistema (MYERS; BADGETT; SANDLER, 2012, p. 122).

Categoria	**Descrição**
Facilidade	Garante que os objetivos foram implementados em uma funcionalidade
Volume	Submete o programa a grandes volumes de entradas para processar
Stress	Submete o programa a grandes cargas e/ou muitas funcionalidades seguidas
Segurança	Tenta quebrar a segurança do *software*
Desempenho	Determina a performance das funcionalidades
Armazenamento	Garante que o *software* gerencia o armazenamento de forma correta
Configuração	Verifica se o programa performa de forma adequada com as configurações recomendadas
Compatibilidade	Determina quando uma nova versão é compatível com a antiga
Instalação	Garante que a nova versão consegue ser instalada em todas as plataformas
Confiabilidade	Determina se o programa segue as especificações de confiabilidade
Recuperação	Testa se o *software* consegue se recuperar após erros
Manutenibilidade	Determina se a aplicação provê mecanismos para lidar com suporte técnico
Documentação	Valida a acurácia de todos os documentos de usuário
Procedimento	Determina a acurácia dos procedimentos para uso da manutenção do *software*

- Testes de Sistema: os testes de sistema servem para mostrar que o produto não está consistente com os objetivos principais (MYERS; BADGETT; SANDLER, 2012, p. 116). Esses testes não podem ser feitos se o produto não tiver objetivos definidos e claros. No Quadro 1 é possível observar os tipos de teste de sistema.
- Testes de Aceitação: esse tipo de teste é utilizado para comparar o programa com os requisitos iniciais e as necessidades atuais do usuário final (MYERS; BADGETT; SANDLER, 2012, p. 131). Os testes de aceitação não são testes de computador.

3. **Testes de Usabilidade**: por muitas décadas os *softwares* foram desenvolvidos pensando no uso das máquinas, num pensamento industrial. Mas de algumas décadas para cá, o uso de *software* por humanos tem aumentado bastante, e a preocupa-

ção com a usabilidade também. Os testes de usabilidade são tentativas de achar problemas no fator humano ou na usabilidade (MYERS; BADGETT; SANDLER, 2012, p. 143). Eles são testes de caixa preta, já que eles estão preocupados com o funcionamento de acordo com as especificações.

Os testes de usabilidade podem ser feitos por meio de testes de computador ou com pessoas. Os testes automatizados são testes para garantir que os textos, botões e outros elementos estão corretos e que a ação deles funcionam como deveriam. Os testes com pessoas são feitos nos encontros com grupos de grandes pessoas para que elas possam testar o *software* e, assim, responder a um questionário sobre o *software*.

1.2 *Bluetooth*

O *Bluetooth* é uma tecnologia que surgiu no final da década de 1990, como uma forma de conectar dispositivos e periféricos. A comunicação entre dispositivos é feita pelas frequências de rádio, que variam de 2.4GHz a 2.4835GHz (JEONG et al., 2015, p. 155). Após mais de 20 anos, consideramos que o *Bluetooth* é uma tecnologia pervasiva, ou seja, está no nosso dia a dia de uma forma natural. Atualmente está na versão 5, mas a versão 4 ainda é bastante utilizada.

Essa tecnologia foi desenvolvida para a troca de informações de curta distância. Ela é dividida entre três grupos: Classe 1 (100m), Classe 2 (10m) e Classe 3 (1m) (JEONG et al., 2015, p.156). De acordo com a IETF RFC 1925 "The Twelve Networking Truths", "É sempre algo: Bom, Rápido, Barato. Você pode escolher (você não pode ter o três) [1]". O *Bluetooth* é uma tecnologia boa e barata, mas ela não é rápida.

Bluetooth já foi considerada uma tecnologia morta em 2003 (MATHIAS, 2003), mas hoje ela é uma das tecnologias mais utilizadas e em crescimento quando falamos em Internet das Coisas (IoT) (TAYLOR, 2019). De acordo com a pesquisa do Gartner citado por Taylor (2019), teremos 43 bilhões de dispositivos de IoT no mundo em 2023, prevendo um crescimento exponencial se comparado com os 20 bilhões de dispositivos em 2020 e os números de anos anteriores.

1.2.1 *Bluetooth Low Energy*

O *Bluetooth Low Energy* (BLE) é um protocolo derivado do *Bluetooth* para comunicações com poucos dados e que tenha baixo consumo de energia.

"Por sua característica principal ser a economia de energia, um dispositivo BLE permanece em modo IDLE (inativo) durante a maior parte do tempo.

[1] It is always something: Good, Fast, Cheap: Pick any two (you can't have all three) - (Tradução livre)

> Esta tecnologia foi desenvolvida para aplicações que apenas precisam enviar poucas informações, esporadicamente saindo do modo "inativo" apenas para realizar conexões que duram apenas milissegundos. Dessa forma, tem-se um consumo energético com picos de 6 mA, mas com uma média de 1 uA apenas."(PESSOA, 2016, pg.3).

O BLE foi inserido na versão 4.0 do *Bluetooth* (Bluetooth Special Interest Group (SIG), 2010) que saiu no ano de 2010. Desde então ele tem sido utilizado para comunicações de periféricos que não necessitam de grandes quantidades de dados de forma contínua.

1.2.2 Periférico vs Central

O BLE pode ser dividido em duas categorias: Periférico e Central. Em uma comunicação entre dois dispositivos *Bluetooth*, sempre um vai ter o papel de periférico enquanto o outro vai ser o central.

1. **Periférico**: periférico que sinaliza disponibilidade ao enviar o *advertising* para que a central reconheça e comece uma conexão. O *advertising* são dados úteis para que a central saiba que tem um periférico perto e saiba como se conectar a ele (AFANEH, 2016, p. 6). Um papel similar ao do periférico é do *broadcaster*, porque ele somente fica enviando o *advertising* e não aceita conexões.

2. **Central**: dispositivo que fica procurando o *advertising* perto e se conectando a ele para enviar algum dado/comando. Uma central consegue observar e se conectar com mais de um periférico, podendo chegar até sete conexões diferentes (JEONG et al., 2015, p. 155). A central também pode ser um *observer*, um dispositivo que somente observa *advertising* perto, mas que não consegue se conectar a ele. Os dispositivos utilizados para central normalmente é o celular, *tablet* e computador (AFANEH, 2018).

1.2.3 Arquitetura do *Bluetooth*

A arquitetura do BLE, descrita na Figura 1, difere um pouco do *Bluetooth* Clássico, mas os chips encontrados na maioria do aparelhos telefônicos hoje são uma mistura dos dois tipos de *Bluetooth* para conseguir utilizar ambos.

Podemos definir cada camada, assim:

- *Physical Layer (PHY)*: essa é a camada de mais baixo nível do *Bluetooth*, e nela se encontra o módulo de comunicação de rádio. Ela trabalha na banda de 2.4GHz e, no BLE, é segmentada em 40 canais.

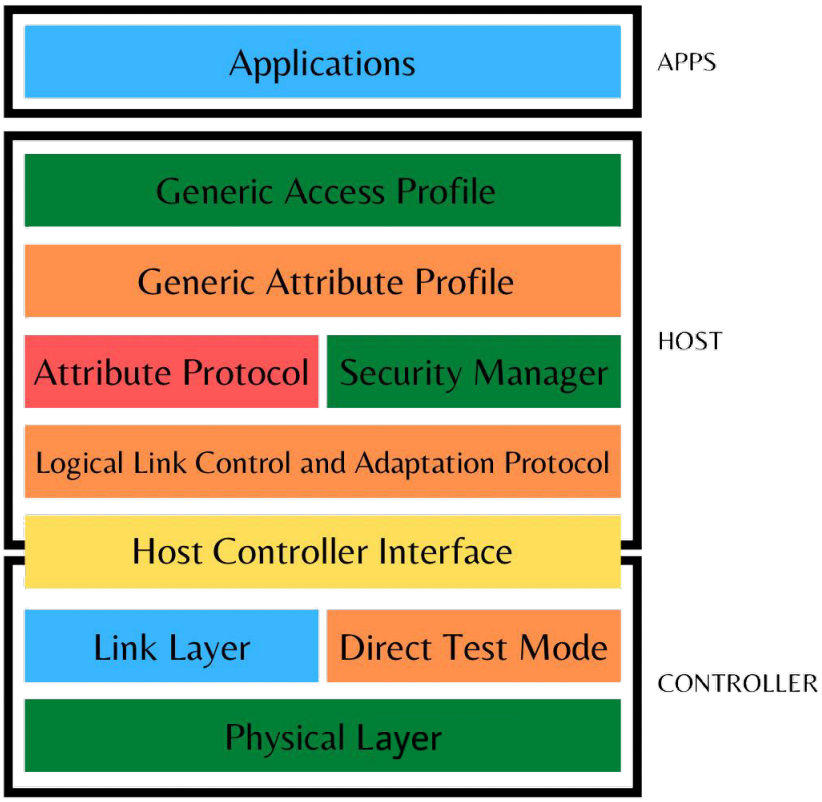

Figura 1 – Arquitetura do BLE.

- *Link Layer*: é responsável por abstrair a camada PHY para que o *host* consiga interagir com a camada. A *Link Layer* também controla o estado do dispositivo BLE, podendo ser: *advertising*, *scanning* e *connected*.
- *Direct Test Mode*: utlizado somente em testes da fabricante ou de certificação.
- *Host Controller Interface (HCI)*: o HCI é uma ponte entre as camadas superiores e as camadas inferiores. Ele é responsável por pegar comandos do *host* e enviar para a *controller* assim como pegar os dados que vêm da *controller* e mandar para o *host*.
- *Logical Link Control and Adaptation Protocol (L2CAP)*: o L2CAP tem dois principais papéis: conseguir passar múltiplos protocolos para as camadas inferiores; e fragmentar ou combinar dados. Quando um dado é muito grande para ser enviado pelas ondas de rádio do BLE, o L2CAP fragmenta esse dado em *chunks* para conseguir enviar um *chunk* por vez. O oposto também acontece: se ele recebe vários *chunks* de uma mesma mensagem, ele consegue combinar para enviar o dado completo para as camadas superiores.
- *Attribute Protocol*: é um protocolo de como o BLE vai manter alguns dados e enviar para o cliente quando ele requisitar, bem como um servidor. Utilizado para armazenar *services* e *characteristics*.

- *Generic Access Profile (GAP)*: o GAP é um *framework*, que é mandatório da implementação; é ele quem estabelece como dois dispositivos *BLE* se comunicam. A comunicação inclui os seguintes aspectos: modos e papéis, *advertisements*, *connection* e segurança (AFANEH, 2018).

- *Generic Attribute Profile (GATT)*: essa camada é utilizada assim que o BLE estabelece uma conexão com outro dispositivo. Ela é responsável pela transmissão de dados entre dispositivos por meio de *characteristics*.

- *Applications*: nessa camada encontra-se a lógica de envio e recebimento dos dados, tratando-os, para serem utilizados da melhor forma.

1.2.4 Segurança

A utilização do *Bluetooth* como um dos principais meios de controle do IoT fez com que a segurança na comunicação desse protocolo aumentasse, principalmente no BLE – o módulo de segurança do BLE só foi inserido na versão 4.2. O *Bluetooth* utiliza ondas de rádio para fazer a comunicação entre dispositivos, então é possível conseguir interceptar essa transferência de dados com um dispositivo malicioso.

As principais preocupações da segurança no BLE são: autenticação, integridade, confidencialidade e privacidade – a garantia desses requisitos fica por responsabilidade de cada aplicação e produto (AFANEH, 2018). A autenticação é utilizada para garantir que o dispositivo que quer se conectar com um módulo BLE é realmente quem diz ser, e não um dispositivo malicioso. A integridade é para garantir que os dados recebidos não estão corrompidos. A confidencialidade é a garantia de que nenhum outro dispositivo conseguirá ler os dados que estão sendo enviados. Já a privacidade está relacionada com a comunicação entre dois dispositivos, para garantir que um terceiro dispositivo não vai conseguir rastrear os dispositivos, durante a comunicação.

No BLE a segurança é de responsabilidade da camada *Security Manager* (SM), também responsável por algoritmos que geram chaves privadas para utilização na criptografia. Por padrão, o BLE utiliza a 128-bit *AES Encryption* para a criptografia (AFANEH, 2018).

1.3 DevOps

Até o início dos anos 2000, o gerenciamento de projetos de *software* era dominado por modelos do estilo cascata (BASSIL, 2012) ou similares. Esse estilo é conhecido, principalmente, por dividir o processo de concepção do *software* em etapas bem definidas e sequenciais. Mas com a evolução da tecnologia, o mercado foi exigindo *softwares* que

necessitavam de entregas constantes para se adequar ao momento (LWAKATARE et al., 2019, p. 217).

O Manifesto Ágil nasceu a partir dessa necessidade do mercado de fazer entregas rápidas e focadas nas necessidades dos usuários. O Ágil é um conjunto de princípios (BECK et al., 2001), que, quando bem aplicados, conseguem entregar para o usuário as alterações necessárias.

Assim como o Ágil veio para melhorar o processo de desenvolvimento de *software*, o DevOps foi criado para conseguir entregar valor para o usuário de forma íntegra e rápida. O DevOps é comumente definido como: movimento, prática, cultura, ferramenta, filosofia, metodologia, neologismo, paradigma, entre outros. Para Lwakatare et al. (2019, p. 228) o conceito de DevOps é:

> "DevOps é um *mindset* de mudança baseado em um conjunto de práticas automatizadas que encoraja colaboração entre equipes multifuncionais, especialmente entre o desenvolvimento e operações de TI – dentro de uma organização de desenvolvimento de software, a fim de operar sistemas resilientes e acelerar a entrega de mudanças.[2]"

O termo DevOps apareceu na literatura somente em 2009 (LWAKATARE et al., 2019, 217), e ele é a junção de duas áreas: desenvolvimento e operações. O desenvolvimento é responsável por, em razão das necessidades do usuário, desenvolver ou melhorar funcionalidades do *software*. Enquanto operações é responsável por entregar esse *software* para todos os usuários e monitorar essas entregas. O DevOps não entrega valor somente para essas duas áreas, podendo entregar valor tanto para a área de qualidade quanto a de suporte (EBERT et al., 2016, p. 94).

Para entregar valor de forma íntegra e rápida para o usuário final, o DevOps possui algumas práticas que utilizam ferramentas para automatizar o processo. Utilizando esse *mindset*, é possível inovar o produto e melhorar as *features* de forma rápida para o mercado (EBERT et al., 2016, p. 95). Lwakatare et al. (2019) cita os benefícios do DevOps como: diminuição do tempo de *realease* – ação de disponibilizar uma nova versão para o usuário –, melhoria na qualidade e diminuição na tensão entre o time de desenvolvedores e de operações. A Amazon consegue entregar, utilizando o conceito de DevOps, uma nova versão do sistema *web* deles a cada 11.6 segundos em média (JENKINS, 2011).

[2] DevOps is a mindset change substantiated with a set of automated practices to encourage cross-functional collaboration between teams–especially development and IT operations–within a software development organisation, in order to operate resilient systems and accelerate the delivery of changes. - (Tradução Livre)

1.3.1 Práticas e Ferramentas do DevOps

O DevOps é implementado dependendo da interpretação do conceito de cada time (LWAKATARE et al., 2019, p. 218). Independente da interpretação, o DevOps possui práticas bem definidas, mas que podem variar de acordo com cada *pipeline* de DevOps – o processo desde o desenvolvimento até chegar no usuário final.

As práticas de DevOps são:

1.3.1.1 Integração Contínua

O *build* é a compilação do projeto, por isso essa etapa depende da linguagem utilizada no projeto, pois dependendo da linguagem essa etapa pode ser descartada (MEYER, 2014, p. 15). Por mais que linguagens interpretadas não necessitem do *build*, alguns *frameworks* de linguagens interpretadas se utilizam dessa etapa. Ao compilar o projeto, garantimos a integridade da versão mesmo após as mudanças no código (EBERT et al., 2016, p. 95). Em alguns projetos, essa etapa pode incluir outras ações além da compilação.

A Integração Contínua (CI) é uma prática do DevOps que garante o funcionamento correto da nova versão que está passando pela *pipeline*. Ela é utilizada principalmente para garantir que a versão está criando o *build* corretamente, e que os vereditos dos testes sejam de sucesso. Com essa prática, é possível garantir que não será enviada uma versão defeituosa para o usuário final (MEYER, 2014, p. 14).

Uma vez com o *script* – conjunto de instruções para realizar uma determinada ação – de CI e os testes escritos, a ferramenta consegue executar a *pipeline* de forma rápida e repetidamente (AMANNEJAD; BETIN, 2015), além de reduzir o risco de erro humano (WIKLUND et al., 2017). A manutenção e melhoramento da ferramenta de CI é importante tanto quanto o desenvolvimento de novas funcionalidades no *software*, pois caso o CI pare de funcionar o desenvolvimento fica comprometido também.

O CI também é responsável por informar a ferramenta de *Deployment* Contínua (CD), a qual é necessária fazer o *deploy* daquela versão, seja em produção ou em *staging* – ambiente próprio para testes.

1.3.1.2 Entrega Contínua e *Deployment* Contínuo

A Entrega Contínua e o *Deployment* Contínuo, conhecidos como CD, são duas práticas do DevOps que são facilmente confundidas, por serem parecidas.

Ambos os conceitos podem ser definidos como uma automação do processo de *deploy* – ação de subir uma nova versão para o usuário, podendo ser em produção ou em teste. A principal diferença entre os dois é o nível de automação de cada um: enquanto a entrega contínua precisa de uma interação humana para ocorrer um *deploy*, o *deployment* contínuo faz o *deploy* a cada mudança no repositório.

O CD faz uso de ferramentas que utilizam *Infrastructure as Code* (IaS) tornando possível compartilhar, testar e controlar cada versão (EBERT et al., 2016, p. 97). A utilização do CD no projeto é benéfico principalmente pelo tempo de entrega de uma versão para o mercado (EBERT et al., 2016, p. 97).

Essa prática de DevOps ainda não é tão trivial quando se trata de sistemas embarcados. Para fazer um *deploy* de um sistema embarcado, é necessário um dispositivo que se comunique ao sistema utilizando a arquitetura de Internet das Coisas (IoT) (EBERT et al., 2016, p. 97).

1.3.2 *Pipelines* de DevOps

O DevOps possui práticas bem definidas, mas cada projeto implementa a sua *pipeline* da forma que mais se adequa à sua realidade.

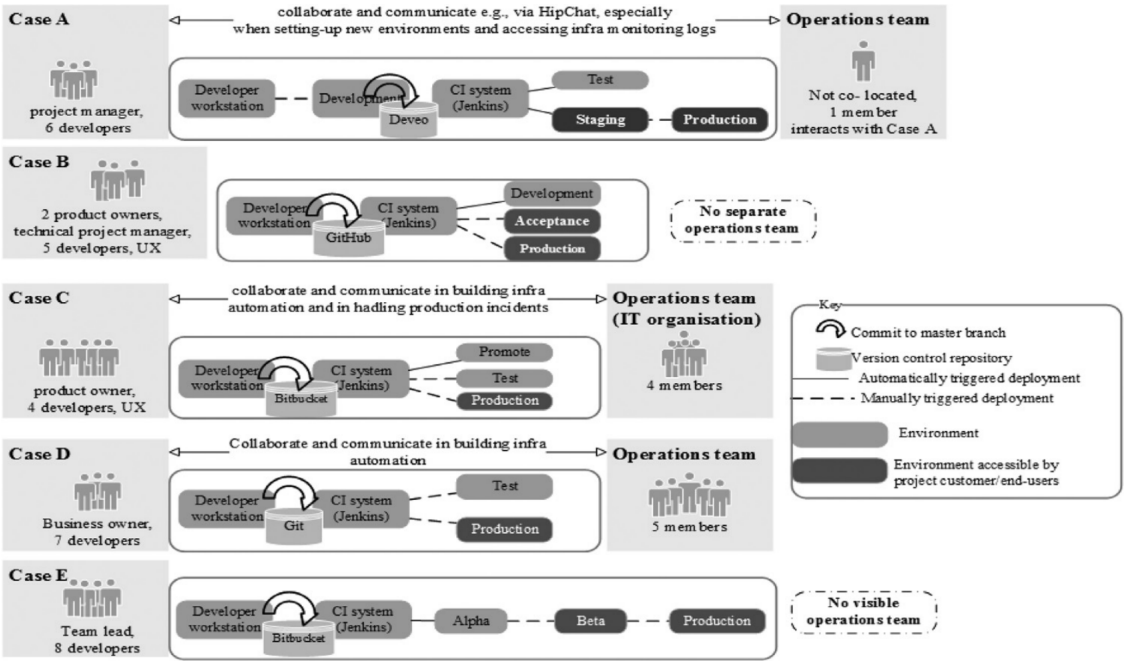

Figura 2 – Exemplos de *pipelines* de DevOps (LWAKATARE et al., 2019, p. 222)

Na Figura 2, é possível ver 5 casos de *pipelines* de empresas reais. Todas as etapas de desenvolvimento e de CI são similares, só alterando as ferramentas utilizadas, no entanto, na etapa de CD, cada empresa realiza etapas diferentes. É possível perceber também que todos os casos possuem ambientes diferentes, sejam de teste interno ou de acesso ao usuário final, e que, em alguns ambientes, o *deploy* é feito ora de forma automática, ora de forma manual.

1.4 Desenvolvimento *Mobile*

O mundo da tecnologia, principalmente de celulares, tem crescido de forma exponencial (GARG; BALIYAN, 2021). Hoje é possível que cada pessoa tenha acesso à informação e à tecnologia na palma de sua mão, através dos dispositivos móveis. Com essa facilidade de acesso a um dispositivo móvel, muitas empresas têm focado no desenvolvimento de aplicativos. Em uma pesquisa feita pela Flurry, a população tem utilizado o celular aproximadamente 2 horas e 42 minutos por dia, sendo que, desse tempo, somente 22 minutos são utilizados em sites da *web* (SPENCE, 2014).

Os dispositivos *mobile* já tiveram grandes plataformas como sistema operacional, mas hoje contam com duas: Android e iOS. De acordo com Garg e Baliyan (2021), em 2020 cerca de 79.95% dos celulares do mundo rodam Android, enquanto 26.27% rodam o iOS, representando 99% do mercado de dispositivos *mobile*. O mundo da tecnologia vem evoluindo rapidamente, e ao mesmo tempo, diminuindo o tempo de lançamento de novidades, por isso tecnologias para desenvolver aplicativos para ambos os sistemas operacionais com um código só têm aumentado. Mas a empresa da LoopKey optou por utilizar tecnologias nativas (Android e iOS) por questões de atuação do *Bluetooth*. Neste TCC só serão abordados testes com a plataforma Android.

1.4.1 Android

O Android é um sistema operacional (SO) mantido e desenvolvido pela Google. Ele é baseado no sistema Linux, por isso é um projeto *Open Source* – projeto que tem o código aberto e permite a alteração e a distribuição de acordo com a licença do dono do projeto, que no caso é a licença Apache 2.0.

O Android é o sistema operacional mais comum entre os brasileiros, pois cerca de 91% dos celulares do território brasileiro rodam uma versão do Android (RIBEIRO, 2019). Essa grande abrangência do mercado do Android se dá principalmente pelo fato de o sistema operacional estar disponível em celulares de diferentes fabricantes.

1.4.1.1 Arquitetura do Android

De acordo com o site Android (2020a), "O Android é uma pilha de *software* com base em Linux de código aberto, criada para diversos dispositivos e fatores de forma."

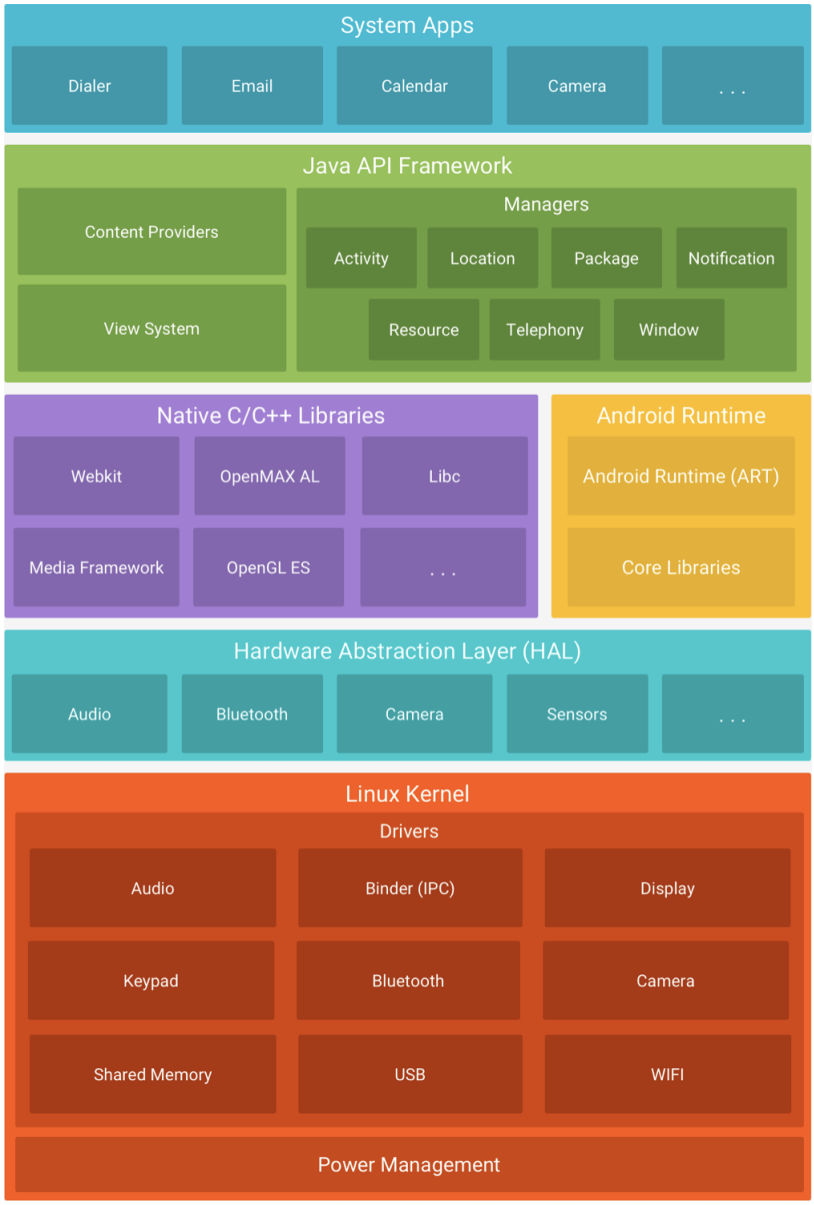

Figura 3 – Arquitetura do Android (Android, 2020a)

Na Figura 3, é possível ver a pilha de *software* que compõe o Android. Podemos definir cada camada como:

- Linux Kernel: a base do Android é o *Kernel* do Linux, pois já é uma tecnologia utilizada em outros dispositivos eletrônicos que facilita a criação de *drivers* do *hardware* para um *kernel* conhecido. O *kernel* é responsável por pegar informações que vêm da camada superior e enviar comandos para o *hardware* do dispositivo. É responsável também por gerenciar serviços como memória, rede, processos, entre outros (GARG; BALIYAN, 2021).

- Camada de abstração de hardware: a camada de abstração de *hardware* (HAL) fornece um meio de comunicação entre as camadas superiores e o *kernel*, para controlar

as capacidades do *hardware* (Android, 2020a). O HAL é composto por bibliotecas especificas para cada módulo do *hardware*, por exemplo: câmera e *Bluetooth*.

- Android *Runtime*: nas versões 5.0 (API 21) ou superior do Android, cada aplicativo roda uma instância do Android *Runtime* (ART). O ART executa várias máquinas virtuais para fazer a gerência de memória. Os principais recursos do ART são: compilação *ahead-of-time* e *just-in-time*; coleta de lixo (GC) otimizada; e melhor compatibilidade de depuração.

- Bibliotecas C/C++ Nativas: alguns serviços do Android, como o ART e o HAL, demandam bibliotecas da linguagem C e C++. Além de serviços principais do Android, é possível acessar essas bibliotecas nos aplicativos pelo Java Interface de Programação de Aplicações (API).

- Java API *Framework*: essa camada possui todas as bibliotecas necessárias para desenvolver um aplicativo no Android. Além de bibliotecas para controlar o *hardware*, por meio do HAL, é possível encontrar bibliotecas para interface do usuário (UI), gerenciador de recursos, gerenciador de notificações, entre outros.

- Aplicativos: os aplicativos para Android são desenvolvidos tanto em Java quanto em Kotlin, ficando a cargo do time de desenvolvimento escolher a melhor tecnologia.

1.4.1.2 Testes no Android

No Android, a biblioteca mais utilizada é a **androidx.test** do Java API para todos os tipos de testes. O teste unitário não necessita de um emulador ou um dispositivo físico para conseguir executar, mas pode ser utilizado quando necessário. Já para os outros testes, é necessária a utilização de um emulador ou um dispositivo físico.

O ciclo de vida dos testes no Android é dividido em cinco etapas: ***BeforeAll***, ***Before***, ***Test***, ***After*** e ***AfterAll***. O *BeforeAll* e o *AfterAll* são funções para preparar e destruir, respectivamente, algumas configurações necessárias para um conjunto de teste. As funções de *Before* e o *After* são similares às funções *All*, mas eles rodam antes e depois de cada caso de teste. E o *Test* são os próprios testes.

Os testes são divididos em duas pastas em projetos Android: ***androidTest*** e ***test***. Na pasta do *androidTest* ficam os testes que necessitam da *Java Virtual Machine* (JVM) para executar, ou seja, necessitam de um dispositivo móvel ou físico. Já na pasta *test*, encontram-se os testes que necessitam somente da máquina local. A documentação do Android recomenda a seguinte divisão: 70% de teste unitário, 20% de teste de sistema e 10% de teste de UI (Android, 2020b).

1.5 LoopKey

A LoopKey, antiga Loop Engenharia, é uma *startup*[3] que nasceu em 2012, idealizada por Pedro Salum e Daniel Assad, na época alunos da Universidade de Brasília (UnB). Ela atuava como uma fábrica de *software* desenvolvendo projetos variados sob contratos. Foi acelerada[4] pelo PCTec da UnB no ano de 2015 (PCTec-UnB, 2021).

A partir de 2015, a LoopKey mudou o foco para um produto no mercado de controle de acesso por meio de fechaduras eletrônicas. A empresa vende e desenvolve uma fechadura, com *firmware* e *software* próprios, com foco na facilitação do compartilhamento de acesso com outras pessoas. É possível compartilhar acesso e gerenciar fechaduras tanto por um sistema *web* quanto pelos aplicativos Android e iOS.

1.5.1 Arquitetura dos Produtos

A LoopKey possui alguns produtos que se comunicam entre si, como podemos ver na Figura 4. Todos os produtos e *softwares* são proprietários desenvolvidos pela própria empresa.

1.5.1.1 Tipos de permissão

Dentro do sistema LoopKey, têm-se quatro tipos diferentes de permissão: *Owner*, *Admin*, *Guest Online* e *Guest Offline*. Já para a fechadura ou controladora só se têm três tipos de usuários: usuários que possuem a chave de *admin* e de usuário, usuários que possuem somente a chave de usuário e os usuários que não possuem nenhuma das chaves.

Quadro 2 – Tipos de Permissão vs Tipos de Usuários.

Tipo de Permissão	**Chave de *Admin***	**Chave de usuário**
Owner	X	X
Admin	X	X
Guest Offline		X
Guest Online		

No Quadro 2, é possível observar como cada tipo de permissão se relaciona com os tipos de usuários para o *firmware*. A chave de *admin* é utilizada para realizar ações de gerenciamento, já a chave de usuário é utilizada para ações comuns. Quando o usuário não tem nenhuma das chaves de usuário, é o servidor quem autentica a permissão do usuário e monta a mensagem para a fechadura.

[3] Uma *Startup* é uma empresa que traz inovação de mercado em meio a incertezas e que possua um modelo de negócios repetível e escalável.

[4] Acelerar uma empresa é um modelo para ajudar uma *startup* a crescer, ajudando com mentorias diversas e injetando capital também.

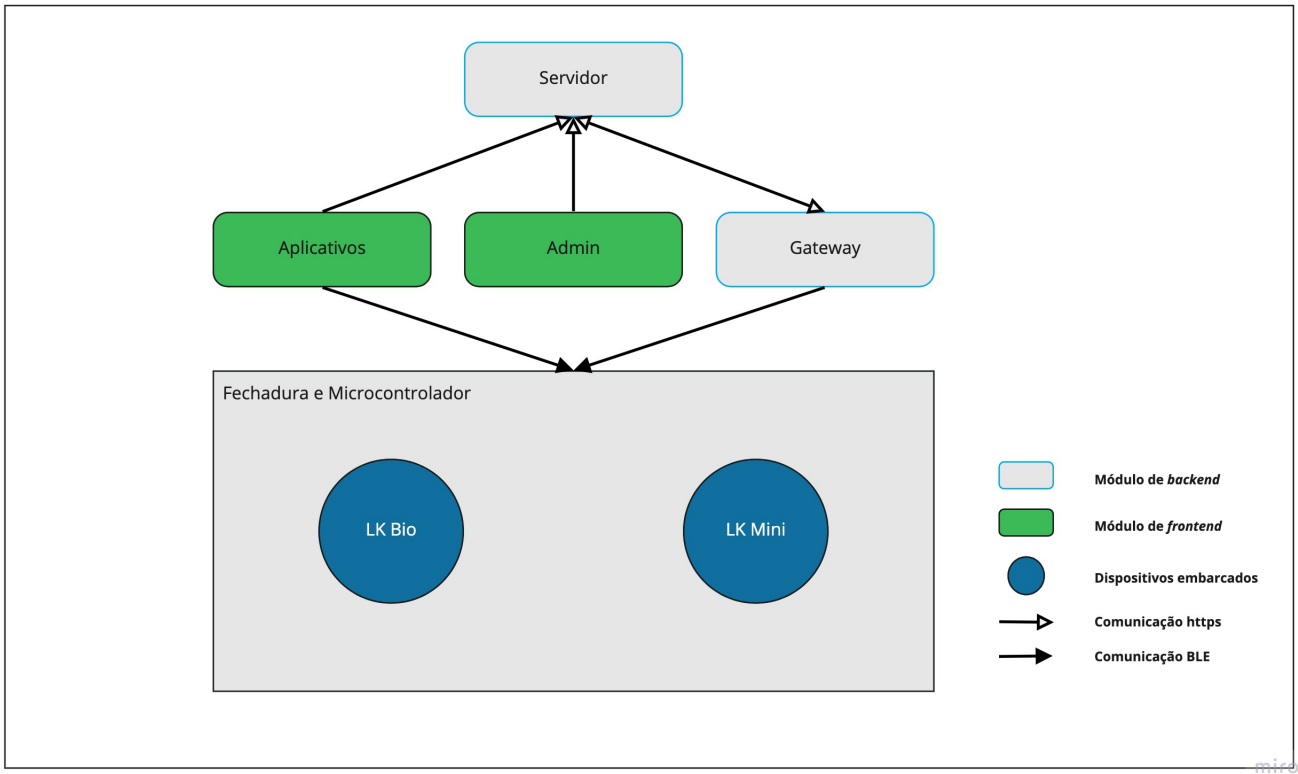

Figura 4 – Arquitetura dos Produtos LoopKey

1.5.1.2 LK Bio e LK Mini

A LoopKey hoje conta com dois produtos principais para abertura da porta: LK Bio (LoopKey, 2021a) e a LK Mini (LoopKey, 2021b). Ambas possuem a comunicação BLE e se comunicam com os mesmos produtos, além de assumirem o papel de periférico, utilizando o BLE de Classe 2. As especificações de cada uma estão demonstradas no Quadro 3.

Ambos os produtos também possuem proteção digital contra ataques de força bruta, e a LK Mini também possui proteção contra *Denial of Service* (DoS), além de sensor de porta fechada e botoeira externa.

Tanto a LK Bio quanto a LK Mini possuem um *sock* da Nordic, o NRF51822. Ele é um chip que já possui a tecnologia BLE e tem um microcontrolador ARM Cortex M0. Em ambos, o *firmware* é desenvolvido na linguagem C e recebe comandos pelo BLE para que possa realizar alguma atividade. Esse recebimento de comando pode ser visto como uma tentativa de alguma ação, pois a fechadura ou controladora, após receber esse comando, além de retornar sucesso também pode retornar alguns erros, como: fora de alcance, falha de assinatura, falha de descriptografia, desconhecido, entre outros.

Quadro 3 – Especificações técnicas da LK Bio e da LK Mini.

Especificações técnicas	**LK Bio**	**LK Mini**
Comunicação com fio	-	Wiegand 26/34 bits e Serial (RS-232)
Formas de Abertura	Smartphone (iOS e Android), Apple Watch, Biometria, Senha, Tags e Cartões 13.56 MHz e Chave Mecânica Pantográfica	Smartphone (iOS e Android), Apple Watch, QRCodes, Senha, Tags e Cartões 13.56 MHz e/ou 125KHz
Alimentação	4 pilhas AAA (aprox. 1 ano de duração)	Fonte de 12-18VDC
Protocolos Criptográficos	HMAC-SHA256 e Speck 128/256 CBC	HMAC-SHA256 e Speck 128/256 CBC

1.5.1.3 Servidor

O servidor da LoopKey é uma API Rest desenvolvida em *Java Spring MVC*. Ele é utilizado para manter a integridade das informações das fechaduras ou controladoras com o banco de dados.

Todos os tipos de permissão possuem acesso ao servidor, mas com níveis diferentes de acesso.

1.5.1.4 *Gateway*

O *gateway* é um dispositivo com comunicação BLE e *wifi*, o qual possui um *firmware* desenvolvido em Typescript. Este dispositivo é utilizado para fazer comunicação com a LK Bio e a LK Mini de forma remota, sem a necessidade de um *smartphone*. Ele recebe informações do servidor para realizar comandos na fechadura ou controladora. O BLE utilizado é da Classe 2.

1.5.1.5 Sistema *Web*

A LoopKey possui um sistema de gestão *web* chamado de *admin*. Ele está disponível para acesso somente para *owners* e *admins*. É um sistema *web* desenvolvido no *framework* Angular JS. Por meio dele, é possível compartilhar/editar acesso, editar uma porta e abrir uma porta de forma remota.

1.5.1.6 Aplicativo

O aplicativo da LoopKey pode ser encontrado tanto para o Android quanto para o iOS, e foi desenvolvido em linguagem nativa Kotlin e Swift, respectivamente. No Android ele está disponível para aparelhos a partir da versão 6.0 (API 23), já no iOS está disponível

para aparelhos a partir do iOS 12. Ambos os projetos, Android e iOS, são divididos entre o aplicativo e uma biblioteca, chamada de LKLib.

A LKLib é desenvolvida na mesma linguagem de cada projeto. Ela possui todas as configurações do BLE como central, ou seja, é ela quem faz a descoberta dos dispositivos em alcance, conecta e envia os comandos. A *lib* reconhece uma fechadura ou controladora pelas informações presentes no *service*.

A biblioteca também é responsável por uma parte da segurança da comunicação. Além da segurança desenvolvida pelo próprio BLE, ela também encripta e decripta as mensagens de acordo com as chaves de *admin* e de usuário.

No Quadro 4 é possível observar as principais funcionalidades dos aplicativos:

Quadro 4 – Principais Funcionalidades dos Aplicativos

1	Configurar uma porta
2	Editar informações/configurações da porta
3	Abrir um porta
4	Compartilhar acesso da porta com um telefone
5	Compartilhar acesso da porta com um *e-mail*;
6	Compartilhar acesso da porta com um cartão
7	Compartilhar acesso da porta com um senha
8	Compartilhar acesso da porta com um biometria
9	Deixar a porta aberta por um tempo (modo *timer*)
10	Cancelar um *timer*
11	Abrir uma porta de forma remota
12	Ler a bateria da fechadura
13	Atualizar o horário e o *timezone*
14	Realizar a atualização do *firmware*
15	Desconfigurar uma porta

2 Metodologia

Este capítulo apresenta a metodologia utilizada para o desenvolvimento deste TCC, para que se possa replicar ou verificar os resultados apresentados. Primeiramente é apresentado o tipo de pesquisa. Logo após, trabalhe-se o problema existente para o qual a pesquisa propõe uma solução. Também explica como foi feito o estudo do problema, explicitando a revisão bibliográfica. São apresentadas ainda as metodologias da Engenharia de *Software* utilizadas para o desenvolvimento da pesquisa, além das ferramentas e dos serviços necessários. Por fim, é apresentado o plano de métricas, bem como será feita a coleta e a análise dos resultados.

2.1 A Pesquisa

Uma pesquisa é um conjunto de ações e propostas que têm como objetivo resolver um problema de forma sistemática. Ela pode ser classificada por objetivo, natureza, abordagem e procedimentos.

Este TCC tem como principal objetivo criar uma bancada de testes com uma *pipeline* automatizada entre o aplicativo móvel e a fechadura eletrônica da LoopKey. A classificação da pesquisa pode ser vista no Quadro 5.

Quadro 5 – Classificação da Pesquisa de TCC.

Objetivo	**Natureza**	**Abordagem**	**Procedimentos**
Pesquisa Exploratória	Pesquisa Aplicada	Pesquisa Quantitativa e Pesquisa Qualitativa	Pesquisa Bibliográfica, Pesquisa Documental e Pesquisa-ação

Uma pesquisa exploratória tem como o objetivo explorar um determinado assunto para torná-lo mais visível ou criar hipóteses. De acordo com o Gil (2002, p. 42), esse tipo de pesquisa é utilizado principalmente para "o aprimoramento de ideias ou a descoberta de intuições".

A classificação pesquisa aplicada pressupõe a aplicação prática dos estudos.

A abordagem do problema é como serão analisados os resultados. No caso dessa pesquisa, será utilizada a análise quantitativa, ou seja, os resultados serão apresentados em números. Com a pesquisa quantitativa é possível apresentar os resultados utilizando recursos e técnicas estatísticas (porcentagem, média, moda, mediana, entre outros). Tam-

bém será utilizada a pesquisa qualitativa, um método para analisar os resultados de forma subjetiva.

Já os procedimentos utilizados no TCC, são três: pesquisa bibliográfica, pesquisa documental e pesquisa-ação. A pesquisa bibliográfica é a busca por referencial com base em material publicado, como livros e artigos científicos. Já na pesquisa documental, a busca é baseada em documentos que não receberam tratamento analítico (GIL, 2002). O conteúdo deste trabalho é algo novo e pouco estudado, por isso foi necessária a utilização desses dois tipos de pesquisa. Por fim, a pesquisa-ação é a resolução de um problema de forma cooperativa entre o(s) pesquisador(es) e o(s) participante(s) da situação.

2.2 Estudo do Problema

As informações contidas nessa seção são frutos de conversas informais e experiência empírica do pesquisador no dia a dia, trabalhando para a empresa em questão.

Na rotina da equipe de *software* da LoopKey é normal a inserção e alteração de funcionalidades que se utilizam da comunicação BLE com o sistema embarcado. Com essas alterações acontecendo semanalmente, ou até mesmo diariamente, a preocupação com o pleno funcionamento da comunicação tem que ser redobrado. Com uma equipe reduzida, duas pessoas desenvolvem novas funcionalidades e mantêm o aplicativo Android e iOS, nem sempre é possível garantir o pleno funcionamento de todas as funcionalidades.

Com isso em mente, o time da LoopKey já vem pensando em formas de fazer testes, principalmente automatizados, para sempre entregar um *software* confiável e diminuir os *tickets* urgentes que o time de suporte recebe após uma atualização mal testada.

2.3 Revisão Bibliográfica

Para o estudo teórico deste TCC, foi realizada uma revisão bibliográfica em artigos acadêmicos e alguns artigos da literatura cinzenta – material publicado em formato eletrônico que não é controlado por editoras comerciais. Um dos filtros do material pesquisado foi a data da publicação. Como este trabalho envolve o estudo do BLE, uma tecnologia relativamente nova, o filtro foi para selecionar artigos a partir de 2014, com algumas exceções.

Foram utilizadas as seguintes plataformas para obtenção dos artigos: Portal de Periódicos CAPES/MEC, Google Acadêmico, documentação oficial do iOS e do Android, e alguns artigos da literatura cinzenta.

2.4 Metodologia de Desenvolvimento

Na Engenharia de *Software* é possível encontrar diversas metodologias para o desenvolvimento de um projeto, tais como: cascata, *scrum*, *lean*, *kanban*, entre outros. Nesta pesquisa, serão utilizadas as seguintes metodologias: ***scrum***, b ***eXtreme Programming* (XP)** e ***kanban***.

2.4.1 *Scrum*

O *scrum* é uma metodologia baseada nos princípios do manifesto ágil (BECK et al., 2001). Dividida entre ritos e artefatos, tem como principal pilar a adaptabilidade a mudanças. Ela funciona por meio de *sprints*, que é um período de tempo definido para entregar um incremento do *software*.

A metodologia consiste em um time multidisciplinar com papéis e rotina bem definidos. O time é formado principalmente por um *Scrum Master*, um Product Owner (PO) e o time de desenvolvimento. O *Scrum Master* é o gerente do projeto, quem comanda os ritos e é um facilitador para o resto do time. O PO é a ligação entre o projeto e os interessados externos. Já o time de desenvolvimento é quem desenvolve o produto. Dependendo do contexto, é possível criar outros papéis.

Os ritos são: *sprint planning*, *daily meeting*, *sprint review* e *sprint retrospective*. A *sprint planning* é um encontro que ocorre no início de um *sprint* e é utilizada para planejar o que será entregue no próximo incremento. A *daily meeting* é uma reunião diária com curta duração para que cada pessoa no time possa falar sobre o que está realizando, além dos impedimentos, fazendo com que o time sempre esteja alinhado e se ajudando. A *sprint review* é uma reunião que ocorre no final da *sprint* para que o time apresente o que foi entregue e levante as melhorias e pendências para a próxima *sprint*. Por último, a *sprint retrospective* pode ocorrer na mesma reunião que a *review* e é utilizada para levantar melhorias no processo no que tange a pessoas, ritos, ferramentas, entre outros.

No *scrum* têm-se três artefatos (SCHWABER; SUTHERLAND, 2015, p. 10). O *product backlog* é uma lista de atividades que serão realizadas para que o *software* seja considerado pronto. Ele é sempre atualizado e priorizado pelo PO. O *sprint backlog* é a lista de atividades a serem feitas na *sprint*. Quando uma *sprint* começa, o *backlog* dela não pode ser alterado. Já o incremento é a nova versão após o trabalho de uma *sprint*, sendo a definição de pronto.

2.4.2 *eXtreme Programming*

O *extreme programming* também é uma metodologia baseada nos princípios ágeis. O XP consiste em práticas de engenharia, baseadas em valores, que são complementares

com a metodologia *scrum*. Os valores do XP são: comunicação, simplicidade, *feedback*, respeito e coragem. Esses valores são a base para um entrega de qualidade do *software*. A metodologia acredita que tem que ter entregas rápidas, com teste, e que tenham refatoração e padronização no código. A seguir temos uma lista das práticas do XP (MEDEIROS, 2006):

- Disponibilidade do cliente: o cliente sempre tem que estar disponível para demandas do time.
- Metáforas: utilizar metáforas dentro do projeto para facilitar a comunicação.
- Planejamento: fazer o planejamento junto ao cliente para as próximas entregas.
- Pequenas versões: fazer entregas pequenas em um curto período de tempo é mais benéfico e consegue *feedbacks* mais rápidos.
- Testes de aceitação: uma métrica desenvolvida no início do projeto para representar o que o *software* necessita para ser aceito. Também pode ser utilizado nas tarefas da *sprint backlog*.
- *Test Driven Development* (TDD): prática que prevê o desenvolvimento de testes antes de desenvolver uma funcionalidade, para garantir a fidelidade do que foi acordado previamente com o cliente.
- Integração Contínua: utilização de *software* para automatizar os testes e o *build* do projeto.
- Simplicidade do projeto: o código tem que ser escrito e documentado da melhor forma, para que qualquer pessoa possa entender.
- Refatoração: forma de melhoria constante do código para manter a simplicidade.
- Propriedade coletiva: garantia de que todos se sintam donos do projeto e com coragem para acrescentar/refatorar um código desenvolvido por outra pessoa.
- *Pair Programming*: desenvolvimento de funcionalidade em dupla. Isso facilita o compartilhamento de conhecimento e fortalece a ideia de propriedade coletiva.
- Rodízio de pessoas: quando utilizado o *pair programming*, é necessário ter um rodízio entre as duplas.
- Padronização de código: utilização de uma folha de estilo para garantir um código conciso.
- 40 horas na semana: o XP defende que cada pessoa tem que trabalhar somente o combinado, sem hora extra, para que não afete a produtividade.

2.4.3 *Kanban*

O *kanban* é um termo japonês que significa "cartão"(DINIZ, 2020). É uma metodologia ágil e visual que facilita a gestão de tarefas. Está dividido em colunas e cartões, onde cada coluna é uma etapa do processo e os cartões são tarefas do projeto. As colunas básicas são: a fazer, fazendo e feito; mas pode-se ter mais divisões dependendo do contexto.

As vantagens dessa metodologia são:

- Autonomia: cada membro do time consegue atualizar as suas próprias tarefas e pegar outras se necessário.
- Priorização: as tarefas podem ser priorizadas de acordo com qualquer critério, e serem facilmente visualizadas por qualquer um.
- Colaboração: como as tarefas são visualizadas por todos do time, é possível que alguém possa colaborar com alguma tarefa de outra pessoa.

2.4.4 Metodologia do Projeto

A metodologia do *scrum* será utilizada somente no inicio do projeto, onde serão desenvolvidos os testes de comando BLE. Nesse momento será utilizado o formato adotado pela empresa. Após esse primeiro momento, o gerenciamento do projeto se dará de forma livre utilizando somente as práticas do XP e o *kanban*.

Para complementar o processo do *scrum*, vão ser utilizadas algumas práticas do XP, sendo elas: disponibilidade do cliente, planejamento, pequenas versões, testes de aceitação, integração contínua, simplicidade do projeto, refatoração e padronização de código.

O *kanban* do projeto será dividido em 5 colunas: pronto para desenvolvimento, em desenvolvimento, pronto para revisão, pronto para *deploy* e finalizado; como observado na Figura 5.

Figura 5 – Exemplo das colunas do *kanban*

2.5 Ferramentas e Serviços de Desenvolvimento

Para o desenvolvimento deste TCC, serão utilizadas algumas ferramentas e serviços da Engenharia de *Software* para auxiliar em todas as etapas.

2.5.1 Clubhouse

O Clubhouse é uma ferramenta de gerenciamento de projetos que utiliza a metodologia *kanban* (Clubhouse, 2020). Essa ferramenta permite a criação de várias camadas de gerenciamento de projetos, como *roadmap*, *milestones*, épicos e iterações. Essa ferramenta já é utilizada dentro da LoopKey para o gerenciamento de todos os projetos.

2.5.2 Git

O Git é uma ferramenta *open source* de controle de versão. Ela é bastante utilizada em desenvolvimento de *software* por permitir o trabalho de diferentes pessoas no projeto de forma simultânea.

As plataformas de colaboração, também conhecidas como *forge*, são utilizadas para manter projetos que utilizam o Git. O *forge* também facilita a utilização das práticas de DevOps. A plataforma que será utilizada no projeto é o Github (Github, 2021), pois é onde estão hospedados os projetos da LoopKey.

2.5.3 Jenkins

O Jenkins (Jenkins, 2021a) é uma ferramenta livre e *open source* de integração contínua. Ela é desenvolvida em Java e necessita de um *web server* para funcionar. Para utilizar o Jenkins, é necessária a instalação dele e um servidor, seja na nuvem ou local, diferente de outras ferramentas que têm uma instância na nuvem já disponível para que outros projetos possam utilizar.

A escolha dessa ferramenta se deu principalmente pela liberdade em que ela proporciona, pois a proposta é montar uma bancada de testes física, é necessário uma instância do CI executando localmente.

2.6 *Gitflow* da LoopKey

O *Gitflow* é um fluxo de trabalho utilizado na ferramenta do *git*. Antes de definir o fluxo do *git*, é importante definir alguns termos:

- *Branch*: é uma ramificação da versão do projeto. Ela é utilizada para dividir os diversos trabalhos feitos em cima de um código, facilitando assim a codificação simultânea de um trecho do código.

- *Commit*: É um registro de modificações do código. Nele são apresentados arquivos alterados, data, mensagem, autor, entre outros.

- *Pull Request* (PR): é a intenção de unir uma *branch* em outra. O PR é uma etapa crucial para a revisão de outras pessoas do time sobre a alteração proposta.
- *Rebase*: é uma das formas com que o *git* consegue fazer a união entre duas *branchs*.

A *branch* principal dos projetos *mobile* dentro da LoopKey tem a nomenclatura de ***main***, já a *branch* de desenvolvimento é a ***dev*** – essa *branch* é utilizada para agrupar todo o desenvolvimento antes de lançar uma nova versão. A partir da *branch main*, é gerada uma versão para os usuário finais. O nome das outras *branches* é definido a partir do *ClubHouse*, que ao criar uma tarefa já sugere um nome para a *branch*, além de definir um identificador único. O *ClubHouse* cria o nome no seguinte formato: **nome-desenvolvedor/id/descrição_da_funcionalidade**.

Ao criar um PR para a *dev* ou *main*, o *GitHub Actions* é acionado para verificar as regras de *lint* definidas para o projeto. Se a etapa de *lint* tiver sucesso e for aprovada por outro integrante da equipe, o autor da funcionalidade pode fazer o *rebase* do seu trabalho na *dev* ou *main*.

2.7 Custos

A criação da bancada necessita de alguns equipamentos para seu pleno funcionamento. No Quadro 6 podemos observar os itens e seus respectivos valores encontrados na internet durante a escrita deste TCC.

Quadro 6 – Quadro de custos.

#	**Item**	**Valor**
1	Fechadura eletrônica LK Bio	R$ 1.600,00
2	Computador MacBook Pro 13"2017	R$ 6.000,00
3	Celular LG K11+	R$ 649,00
4	Bancada	R$ 300,00

A soma de todos os itens totaliza um valor de R$ 8.549,00. Para a replicação desse projeto, não é necessário a utilização do mesmo computador ou celular, é possível utilizar o que tiver disponível.

2.8 Coleta e Análise de Resultados

A coleta dos dados será feita durante todo o desenvolvimento, enquanto a análise será realizada ao final. A coleta e análise dos resultados são importantes para validar e garantir a conclusão da pesquisa, seja ela como esperada ou não. Para fazer a análise dos resultados, será levado em conta o plano de medição.

2.8.1 Plano de Medição

Com o plano de medição fica demonstrado quais métricas serão coletadas e analisadas para concluir os resultados da pesquisa. Na Quadro 7 é possível observar as métricas utilizadas na pesquisa.

Quadro 7 – Métricas.

#	**Métrica**
1	Cobertura dos *status* dos testes
2	Cobertura das principais funcionalidades
3	Automatização dos testes ao abrir um *Pull Request* (PR)
4	Testes de comando BLE

2.8.1.1 Métrica 1 - Cobertura dos *status* dos testes

Tem como objetivo monitorar os *status* dos testes que foram executados.

- **Descrição:** verificar se todos os testes estão funcionando de acordo.
- **Formula:** $CST = \frac{TotalDeTestesComSucesso}{TotalDeTestes} * 100$
  - CST = Cobertura dos *Status* dos Testes
  - TotalDeTestesComSucesso = Total de testes que obtiveram sucesso.
  - TotalDeTestes = Total de testes.
- **Forma de coleta:** a coleta será feita pelas informações obtidas da ferramentas de testes dos aplicativos.
- **Escala de medição:** proporcional.
- **Indicador:**
  - CST = 100%: Ótimo
  - CST < 100%: Ruim

2.8.1.2 Métrica 2 - Cobertura das principais funcionalidades

Tem como objetivo verificar a porcentagem das principais funcionalidades testadas.

- **Descrição:** verificar a cobertura das funcionalidades.
- **Formula:** $CF = \frac{TotalDeFuncionalidadesTestadas}{TotalDeFuncionalidades} * 100$
  - CF = Cobertura das Funcionalidades

  - TotalDeFuncionalidadesTestadas = Total das funcionalidades que possuem testes.
  - TotalDeFuncionalidades = Total das principais funcionalidades.

- **Forma de coleta:** a coleta será feita de forma manual utilizando um *checklist* apresentado no Quadro 4.
- **Escala de medição:** proporcional.
- **Indicador:**
  - CF = 100%: Ótimo
  - 80% >= CF < 100%: Bom
  - 50% >= CF < 80%: Regular
  - 50% < CF: Ruim

2.8.1.3 Métrica 3 - Automatização dos Testes ao Abrir PR

Tem como objetivo garantir que a *pipeline* irá executar sempre que um PR for aberto.

- **Descrição:** verificar a automatização da *pipeline* dentro do CI.
- **Forma de coleta:** a coleta será feita pela observação do comportamento do CI ao abrir um PR.
- **Escala de medição:** subjetiva.

2.8.1.4 Métrica 4 - Testes de Comando BLE

Tem como objetivo garantir que os testes estão conseguindo enviar os comandos BLE na bancada.

- **Descrição:** verificar o funcionamento do BLE na bancada de testes.
- **Forma de coleta:** a coleta será feita pela observação dos testes na bancada.
- **Escala de medição:** subjetiva.

3 Proposta

A proposta do TCC é a criação de uma bancada física para testes automatizados entre o aplicativo móvel Android e a fechadura eletrônica da LoopKey, para garantir o funcionamento da versão do aplicativo da LoopKey. Para essa bancada, foi necessário desenvolver todos os testes de comando BLE da plataforma Android, configurar o Jenkins como ferramenta de Integração Contínua e integrar o GitHub com o Jenkins para a comunicação de mão dupla da *pipeline*.

Os testes do aplicativo móvel são pensados para garantir o funcionamento de uma funcionalidade dependendo do tipo de usuário utilizando o aplicativo. Quando um teste tem um veredito de falha, ele pode estar relacionado tanto ao código do próprio comando em si, que estaria na LKLib, ou estar relacionado ao tipo de usuário, nesse caso pode ser um problema na API, no aplicativo ou até em como foi compartilhado aquela permissão com o usuário. Por mais que os testes não se proponham a testar a fechadura em si ou a LKLib, isso acaba ocorrendo, já que a bancada simula um ambiente real, mas o testes não irão identificar onde está problema caso seja na fechadura ou na LKLib.

A bancada de testes contará com um computador pessoal, um dispositivo móvel Android e uma fechadura LK Bio. A ferramenta do Jenkins será instalada no computador pessoal para conseguir executar a *pipeline*. O dispositivo móvel será conectado ao computador por um cabo USB para que o Integração Contínua (CI) consiga executar os testes. A fechadura deverá estar a menos de 1 metro dos dispositivos, evitando que fiquem foram de alcance do BLE. Vale ressaltar que a fechadura não poderá ser utilizada para outra finalidade, além dos testes.

O gatilho da *pipeline* se dará por duas situações distintas:

- **Abertura de um *Pull Request* para a *branch main***: a principal *branch* do projeto é a *main*, ou seja, ela representa o que está em produção. Por conta disso, é necessário garantir o funcionamento dos principais comandos do aplicativo antes de enviar uma nova versão para a *main*.
- **Periodicamente**: durante os dias da semana, o CI irá executar a *pipeline* na *branch dev* em um horário predeterminado.

Como no dia a dia são abertos diversos *pull requests*, foi decidido por executar o CI todo dia com as alterações da *dev*, para evitar um congestionamento de *builds*.

A substituição do dispositivo móvel não acarretará em problemas, já que o CI roda os testes baseados no dispositivo encontrado. Mas a substituição da fechadura pode ser

danosa, e necessitará de alteração no *script* da *pipeline*.

Neste TCC serão contemplados somente os testes unitários, deixando os testes de usabilidade e os testes de *high order* como possíveis evoluções.

3.1 Requisitos

Os requisitos do projeto são as atividades necessárias para entregar a bancada de testes. Para conseguir descrever os requisitos, além da pré-rastreabilidade, técnica para entender o contexto e o plano de ação para um problema, foi feita também a elicitação com técnicas da Engenharia de *Software*.

Para fazer a pré-rastreabilidade dos requisitos da bancada de testes automatizados, foi aplicada a metodologia 5W2H, utilizada para levantar um plano de ação para um determinado problema. No Quadro 8, é possível observar as perguntas e respostas que norteiam a pesquisa. Ela também dá base para a elicitação dos requisitos.

Quadro 8 – 5W2H.

Pergunta	**Resposta**
O que?	Garantir a utilização das principais funcionalidades dos aplicativos móveis da LoopKey
Por quê?	Aumentar a confiabilidade das versões enviadas a produção
Onde?	Na casa do pesquisador
Quem?	Pesquisador
Quando?	Durante o ano de 2021
Como?	Utilizando uma bancada física para realizar testes automatizados entre os aplicativos e a fechadura da LoopKey
Quanto?	Total de R$ 8.549,00 de acordo com a Seção 2.7

A elicitação dos requisitos foi feita utilizando técnicas de observação participativa, introspecção e *brainstorming*. Ela é dividida entre requisitos funcionais – tarefas a serem executadas para gerar o produto final – e não funcionais – são requisitos relacionados à qualidade do software. Eles são descritos, respectivamente, nos Quadros 9 e 10.

Quadro 9 – Requisitos Funcionais.

Número	Requisito	Técnica
1	Escrever os testes das funcionalidades na plataforma Android	Observação Participativa
2	Montar uma bancada com o computador, um celular Android e uma fechadura LK Bio	*Brainstorming*
3	Configurar o Jenkins no computador	*Brainstorming*
4	O Jenkins deve ser capaz de iniciar uma *pipeline* sempre que um PR para a *main* for aberto	*Brainstorming*
5	O Jenkins deve ser capaz de iniciar uma *pipeline* periodicamente	*Brainstorming*
6	O Jenkins deve ser capaz de se recuperar e retornar a fechadura para o estado inicial, caso haja erros na *pipeline*	Introspecção
7	O Jenkins deve ser capaz de informar o *status* da *pipeline* para o GitHub	Introspecção

Quadro 10 – Requisitos Não Funcionais.

Número	Requisito	Categoria	Técnica
1	Evitar acesso de pessoas de fora da empresa	Segurança	Introspeção
2	Os testes precisam dar *retry* em caso de erro de comunicação (*Door Not Reachable*)	Confiabilidade	Introspeção
3	O Jenkins precisa conseguir executar a *pipeline* do Android	Compatibilidade	Introspeção

A partir dos requisitos apresentados acima, foi levantado um *backlog* do projeto utilizando histórias de usuários. É possível encontrar o *backlog* no Apêndice A.

3.2 A *Pipeline*

A *pipeline* utilizada na ferramenta de CI está dividida conforme as etapas descritas na Seção 1.3.1.1. Na Figura 6, são apresentadas todas as etapas e subetapas da *pipeline*. Percebe-se na figura que o *admin* irá compartilhar tanto com o *guest online* quanto com o *guest offline*, mas os testes de ambos não serão feitos de forma simultânea, e sim seguindo a ordem presente na figura.

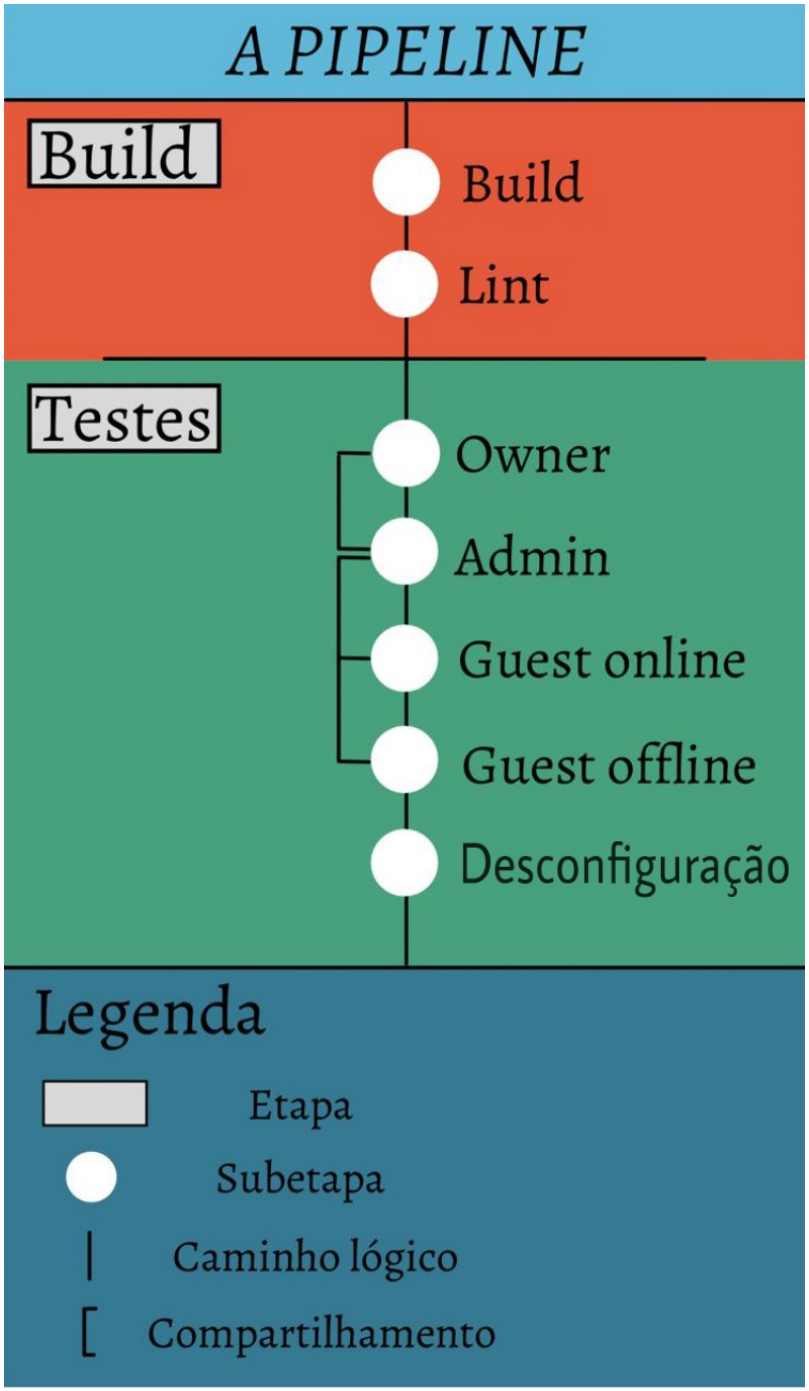

Figura 6 – A *pipeline*.

No Quadro 11, é possível observar a descrição e a ordem dos testes.

Quadro 11 – Ordem dos testes.

Tipo de Usuário	**Comando**	**Código**
Owner	Configurar uma fechadura	owner_1
	Editar uma fechadura	owner_2
	Realizar abertura	owner_3
	Verificar se a fechadura está em modo *timer*	owner_4
	Configurar o modo *timer* na fechadura	owner_5
	Verificar se a fechadura está em modo *timer*	owner_6
	Cancelar o modo *timer*	owner_7
	Compartilhar uma permissão de *admin* por *e-mail*	owner_8
	Compartilhar uma permissão de *admin* por telefone	owner_9
	Compartilhar uma permissão de cartão	owner_10
	Compartilhar uma permissão de senha	owner_11
Admin	Realizar abertura	admin_1
	Compartilhar uma permissão de *guest online* por *e-mail*	admin_2
	Compartilhar uma permissão de *guest offline* por *e-mail*	admin_3
Guest Online	Realizar abertura	guestOnline_1
	Verificar se a fechadura está em modo *timer*	guestOnline_2
	Configurar o modo *timer* na fechadura	guestOnline_3
	Verificar se a fechadura está em modo *timer*	guestOnline_4
	Cancelar o modo *timer*	guestOnline_5
Guest Offline	Realizar abertura	guestOffline_1
	Verificar se a fechadura está em modo *timer*	guestOffline_2
	Configurar o modo *timer* na fechadura	guestOffline_3
	Verificar se a fechadura está em modo *timer*	guestOffline_4
	Cancelar o modo *timer*	guestOffline_5
Owner	Remover permissão de *admin* por *e-mail*	ownerFinal_1
	Remover permissão de *admin* por telefone	ownerFinal_2
	Remover permissão de *guest online*	ownerFinal_3
	Remover permissão de *guest offline*	ownerFinal_4
	Remover permissão de cartão	ownerFinal_5
	Remover permissão de senha	ownerFinal_6
	Desconfigurar a fechadura	ownerFinal_7

3.2.1 Estrutura dos Testes

A maioria dos testes da *pipeline* descrita são testes unitários, com exceção do comando de configuração e desconfiguração, que são testes de integração. Os testes unitário são relacionado a um pequeno módulo do projeto, normalmente um método. Já os testes de integração envolvem 2 ou mais módulos, como é o caso do comando de configuração e desconfiguração que envolve mais de dois métodos.

Como os testes envolvem uma comunicação BLE, e esse tipo de comunicação é passível de erros inesperados, os testes terão a possibilidade de *retry* caso retorne falha no comando. Os casos de testes serão programados para tentar 3 vezes um comando. Se retornar sucesso em alguma das 3 tentativas, o veredito do caso de teste será de sucesso; caso retorne erro nas 3 tentativas, o veredito será de falha.

Os comandos BLE são assíncronos, então os testes devem estar preparados para esperar a resposta da fechadura ou retornar erro em caso de *timeout*, que será de 10 segundos.

4 Desenvolvimento

Para alcançar o objetivo de testar o aplicativo móvel de forma automatizada, foi montada uma bancada com todos os itens para se comunicar entre si. A disposição da bancada pode ser observada na Figura 7; já a lista de dispositivos pode ser encontrada a seguir:

- Computador MacBook Pro 13"Modelo 2017 com macOS Big Sur 11.6;
- Celular LG K11+ com Android 7.1.2;
- Fechadura LK Bio com versão 6.4.0.

Figura 7 – A bancada.

O desenvolvimento divide-se em: Escrita dos Testes, Configuração do Jenkins e Integração com o GitHub. A etapa de escrita dos testes foi realizada junto da configuração do Jenkins, já que um não necessitava do outro totalmente pronto. Já para a integração com o GitHub, foi necessária esperar as duas primeiras etapas serem concluídas.

O código dos testes e outras configurações sensíveis foram omitidos do trabalho por questão de *compliance*.

4.1 Testes

Essa etapa é a mais importante do trabalho, pois é nela que garantimos que uma funcionalidade está sendo executada corretamente ou não, por isso essa etapa foi a mais longa para deixar os testes completos e confiáveis. Durante os testes, também foram levantados pontos de melhorias por todo o trabalho, principalmente sobre a *pipeline* proposta inicialmente.

Para o desenvolvimento dos testes da plataforma Android foram utilizados a IDE Android Studio na versão 4.2, com o *gradle* na versão 4.2.1, o Kotlin na versão 1.5.0 e Java na versão 1.8.0_281.

Os testes foram escritos em sua totalidade na linguagem Kotlin, utilizando duas bibliotecas principais: JUnit na versão 4.12 e Awaitility na versão 3.1.6. A biblioteca JUnit, uma das mais utilizadas no desenvolvimento Android, foi utilizada para realizar testes unitários. Já a biblioteca Awaitility é utilizada para esperar as respostas das chamadas assíncronas. Ela foi necessária para esperar as respostas do comando BLE, já que o JUnit não sabe lidar com esse tipo de chamada.

Foram escritos no total 16 testes unitários, conforme descritos acima no Quadro 11. Para executar os testes de acordo com cada tipo de usuário, foram criados também SuiteClasses. Os SuiteClasses, que serão chamados de suites de testes, é uma interface do JUnit que permite agrupar um conjunto de testes na ordem desejada. As suites de testes são explicadas no Quadro 12, abaixo.

Todas as suites seguem uma ordem semelhante: *login* no usuário selecionado, download das portas do usuário, os comandos a serem testados para aquele usuário e o *logout* do usuário. A única exceção a essa ordem é a do ErrorUnitTestSuite, que é chamada quando acontece um erro na *pipeline* para desconfigurar uma porta e deixar tudo preparado para uma nova *pipeline* executar.

É possível executar os testes de três formas distintas: pelo Android Studio, pelo Android Debug Bridge (adb) e pelo gradlew. O adb é uma ferramenta de linha de comando que realiza comunicações com o dispositivo conectado ou emulador. Já o gradlew também é uma ferramenta de linha de comando que utiliza o gradle para empacotar e executar versões/testes em um dispositivo. À primeira vista essas duas soluções parecem semelhantes, mas têm uma diferença impactante para a escolha da ferramenta utilizada: o gradlew sempre apaga a versão do aplicativo do dispositivo ao terminar de executar os testes. Por esse motivo foi escolhido utilizar o adb como ferramenta, já que é necessário que o aplicativo continue instalado por conta das permissões exigidas. O Android Studio não foi cogitado já que ele é uma ferramenta visual e seria necessária uma ferramenta de linha de comando.

Quadro 12 – As suites de testes.

Suite de Teste	**O que é**
OwnerUnitTestSuite	Os testes são realizados para o usuário do tipo *owner*. Nessa etapa é feita toda a configuração da fechadura e também testados os principais comandos do *owner*, principalmente os de compartilhamento de permissão com cartão e senha.
AdminUnitTestSuite	Os testes são realizados para o usuário do tipo *admin*. Nessa etapa são feitos os testes dos principais comandos do *admin*, principalmente os de compartilhamento de permissão com os dois tipos de *guest*.
GuestOnlineUnitTestSuite	Os testes são realizados para o usuário do tipo *guest online*. Nessa etapa são feitos somente os testes de abertura da fechadura e colocar/verificar/cancelar modo *timer*.
GuestOfflineUnitTestSuite	Os testes são realizados para o usuário do tipo *guest offline*. Nessa etapa são feitos somente os testes de abertura da fechadura e colocar/verificar/cancelar modo *timer*.
OwnerFinalUnitTestSuite	Os testes são realizados para o usuário do tipo *owner*. Nessa etapa é feita toda a desconfiguração da fechadura, além de remover todas as permissões criadas ao longo da *pipeline*.
ErrorUnitTestSuite	Essa suite é realizada com o usuário *owner* da fechadura e é disparada quando acontece algum erro em qualquer etapa da *pipeline*. Nessa etapa é feita a desconfiguração da fechadura.

Para conseguir utilizar tanto o aplicativo quanto executar os testes, é necessário liberar duas permissões: *bluetooth* e localização (o Android utiliza a localização para a descoberta de novos dispositivos em alcance). Não foram encontradas formas de liberar essas permissões por linha de comando, logo a solução foi instalar a versão no celular e liberar essas permissões de forma manual. Caso o aplicativo seja desinstalado do celular, será necessário fazer essa etapa manual de novo.

Para realizar novamente os testes, foi utilizada uma interface do JUnit, a TestRule, que define uma regra que o teste tem que seguir. Após uma quantidade de tentativas e observada alguma falha, ela verifica se pode tentar novamente ou realmente retornar error ao usuário. A implementação dessa classe pode ser encontrado no Apêndice B.

Os testes de comando BLE seguem uma estrutura parecida em todos os casos. Eles herdam de uma classe abstrata BaseUnitTest que já implementa algumas regras necessárias para todos os comandos, como: pegar o usuário logado, pegar a porta a ser testada, iniciar a busca por portas em alcance, parar a busca por portas em alcance e

esperar uma porta ficar em alcance. Uma pseudoimplementação dessa classe pode ser encontrada no Apêndice C.

No teste em si, ele só chama uma função para executar o teste e espera uma resposta de sucesso com o Awaitility. A pseudoimplementação de um teste de comando pode ser encontrada no Apêndice D.

4.2 Jenkins

O Jenkins é uma ferramenta de automação, que ajuda na implementação da Integração Contínua e Entrega Contínua. Ela foi iniciada em 2004 por Kohsuke Kawaguchi, mas hoje é mantida pela comunidade e apoiada por grandes empresas (Red Hat, Amazon, GitHub, entre outras). Ela é uma ferramenta *open source* e que é possível a instalação gratuita de forma fácil e rápida. Ela foi escolhida pela facilidade de instalação em uma máquina para realizar os testes localmente.

Ele foi instalado seguindo a documentação oficial (Jenkins, 2021b). Foi instalada a versão 2.277.3 utilizando o *homebrew*, o gerenciador de pacotes do macOS, utilizando o comando brew install jenkins-lts. Como esse TCC utiliza a versão 2.277.3, caso deseja instalar essa versão em especifico, pode ser utilizado o comando brew install jenkins-lts@2.277.3.

Para iniciar o serviço do Jenkins foi utilizado o seguinte comando brew services start jenkins-lts. Com esse comando, ele inicia o Jenkins na porta 8080, então é possível acessar ele pelo caminho http://localhost:8080.

4.2.1 Configuração de variável de ambiente

Com o Jenkins instalado, foi necessário fazer algumas configurações para que o ambiente Java funcione como esperado. Para isso é necessário ir em Gerenciar Jenkins, como demonstrado na Figura 8. Já na Figura 9, é possível observar as duas variáveis de ambiente configuradas.

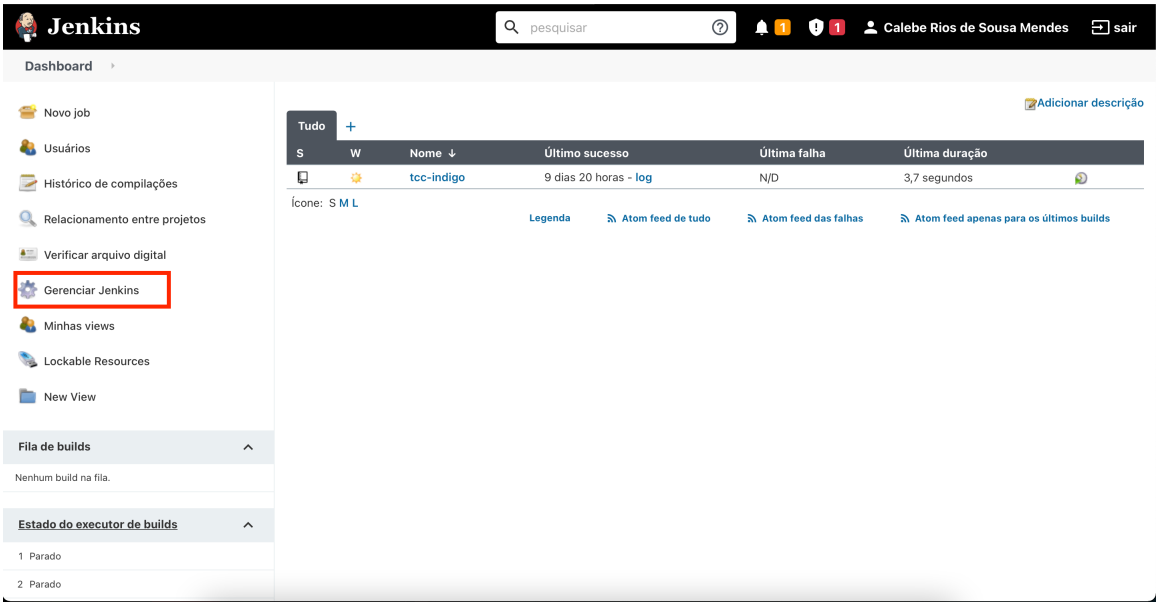

Figura 8 – Dashboard com apontamento para Gerenciar Jenkins.

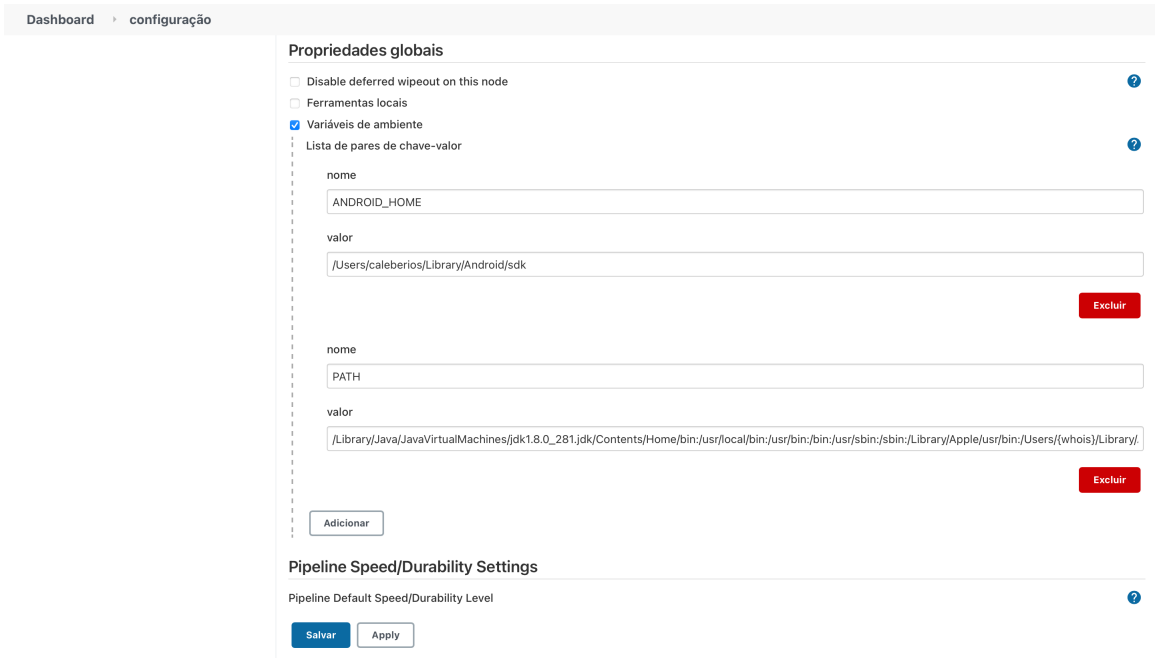

Figura 9 – Configuração do JDK.

A primeira variável é necessária para dizer onde está instalado o sdk do Android. Ela utiliza o caminho completo, então é necessário alterar pelo menos o nome do usuário. Já a segunda variável é utilizada para dizer o caminho do Java instalado. As variáveis são:

- ANDROID_HOME: /Users/caleberios/Library/Android/sdk
- PATH : /Library/Java/JavaVirtualMachines/jdk1.8.0_281.jdk/Contents/Home /bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/Apple/usr/bin:/Users /whois/Library/Android/sdk/platform-tools:/Users/whois/Library/Android/sdk/tools

4.2.2 Configuração do projeto

Em seguida, deve-se configurar um novo projeto. Para criar isso, é necessário ir em Novo Job ta tela de Dashboard. O projeto criado foi do tipo Multibranch Pipeline, como podemos observar na Figura 10. O Multibranch Pipeline é utilizado quando se tem um conjunto de *pipelines*, que nesse caso, ele vai criar uma para cada *pull request* aberto.

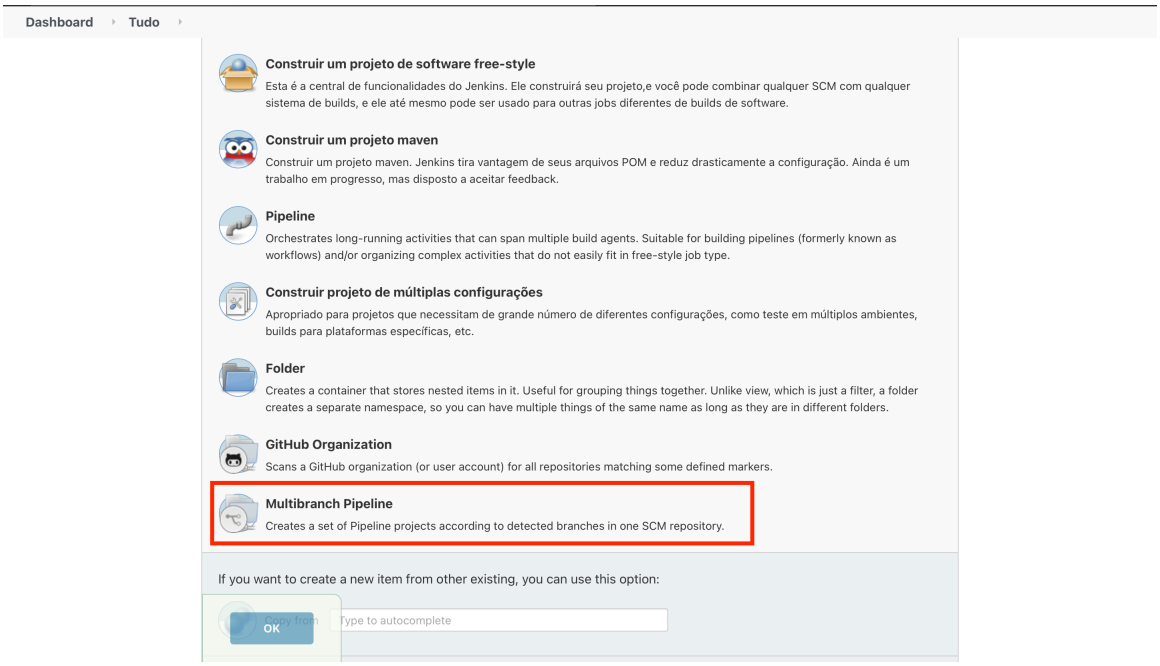

Figura 10 – Selecionar tipo de projeto.

Na configuração do projeto é necessário adicionar algumas configurações, como, por exemplo, a URL do GitHub utilizado. Na Figura 11 é possível ver como foi configurado o repositório para esse projeto.

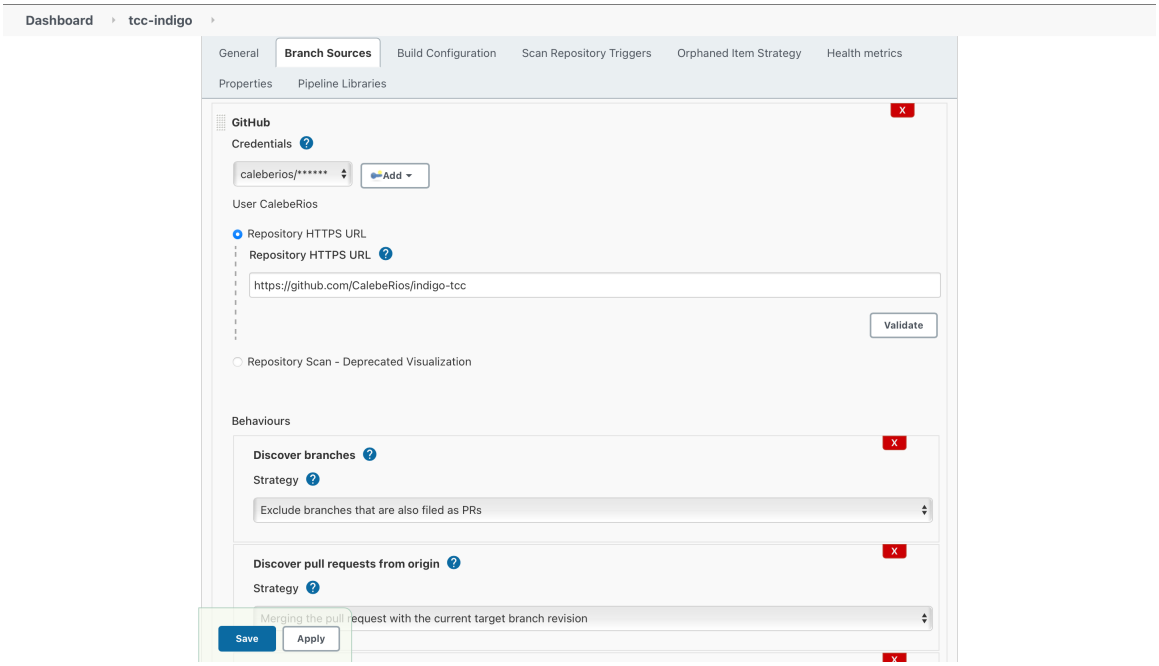

Figura 11 – Configuração do GitHub.

Como o repositório é privado, é necessário configurar a credencial do GitHub, como demonstrado na Figura 12.

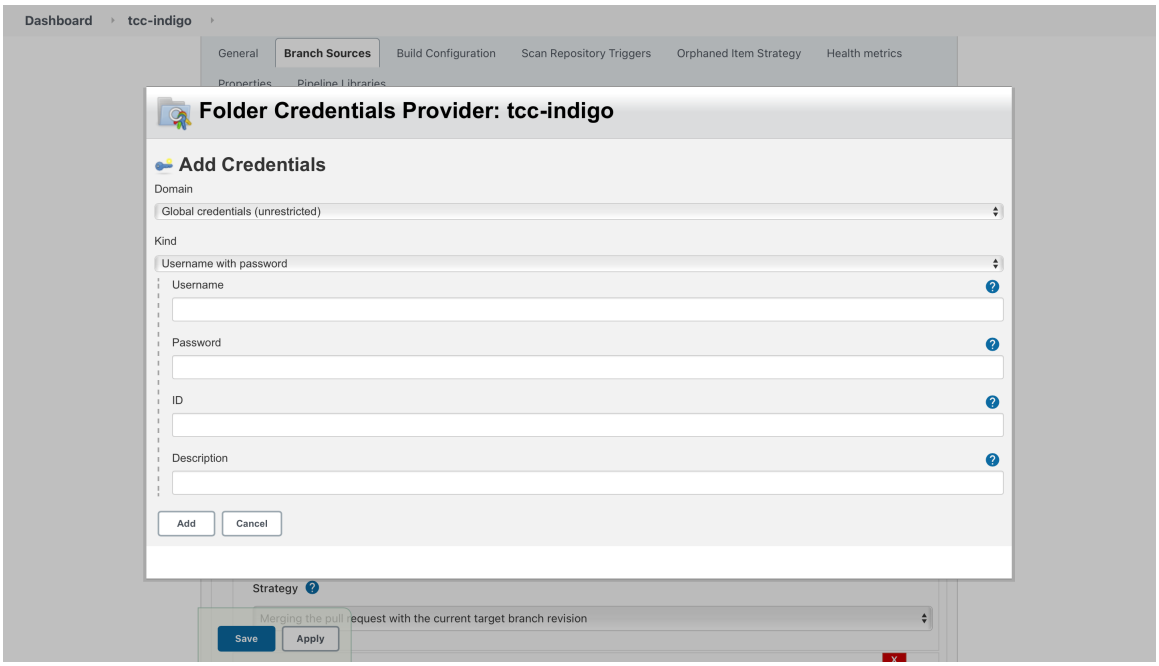

Figura 12 – Configuração a credencial do GitHub.

Na versão atual do GitHub é necessário criar um *access token* que vai servir como uma senha. Para criar um *access token* é necessário ir em *settings* do GitHub, depois em *developer settings* e por último em *personal access tokens*, como demonstrado na Figura 13.

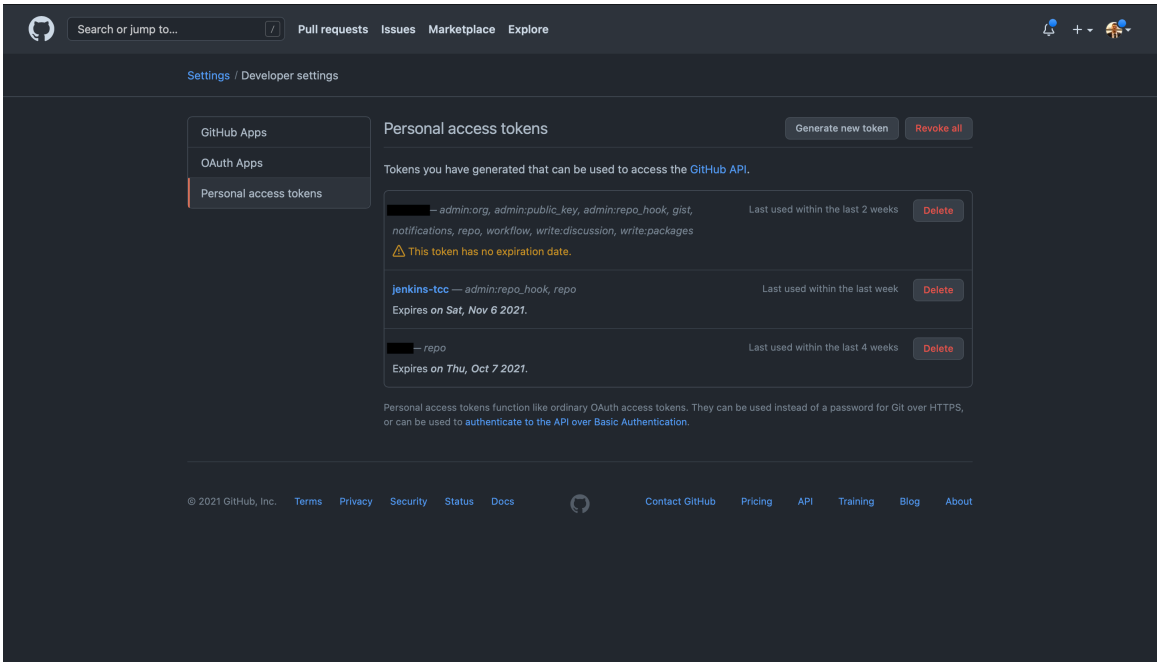

Figura 13 – Tela de *access token* do GitHub.

Para criar um novo *token*, é necessário clicar em *Generate new token*, escolher um nome, selecionar a opção *repo* e *admin:repo_hook*, como também observado nas Figuras 14 e 15.

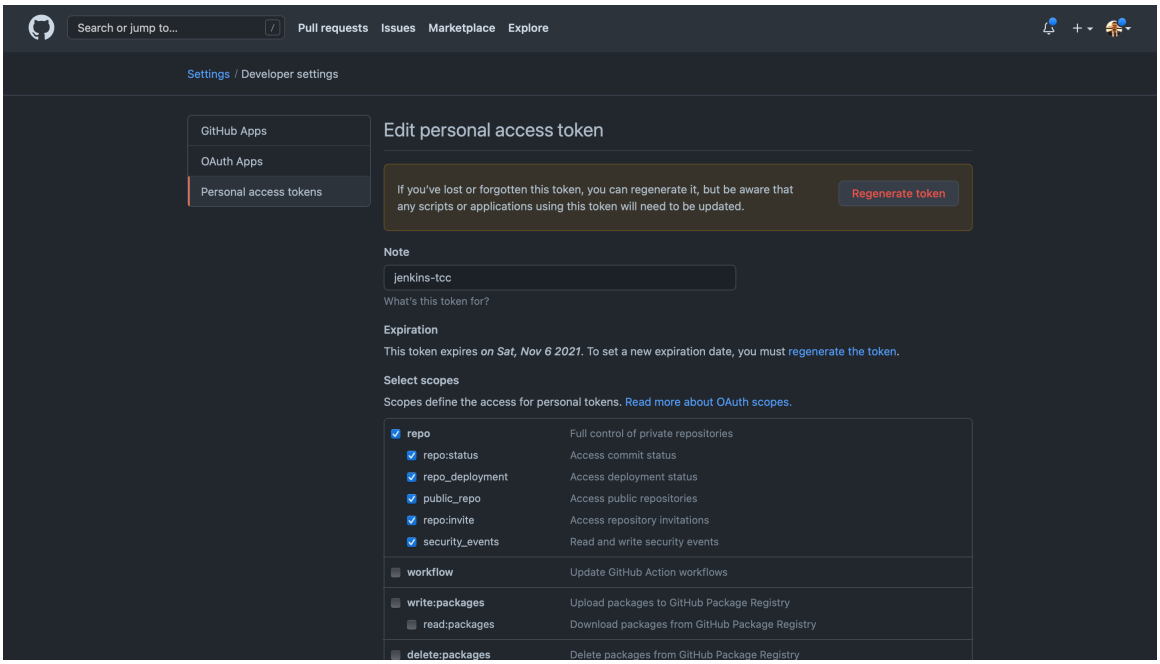

Figura 14 – Tela de gerar uma nova *access token*.

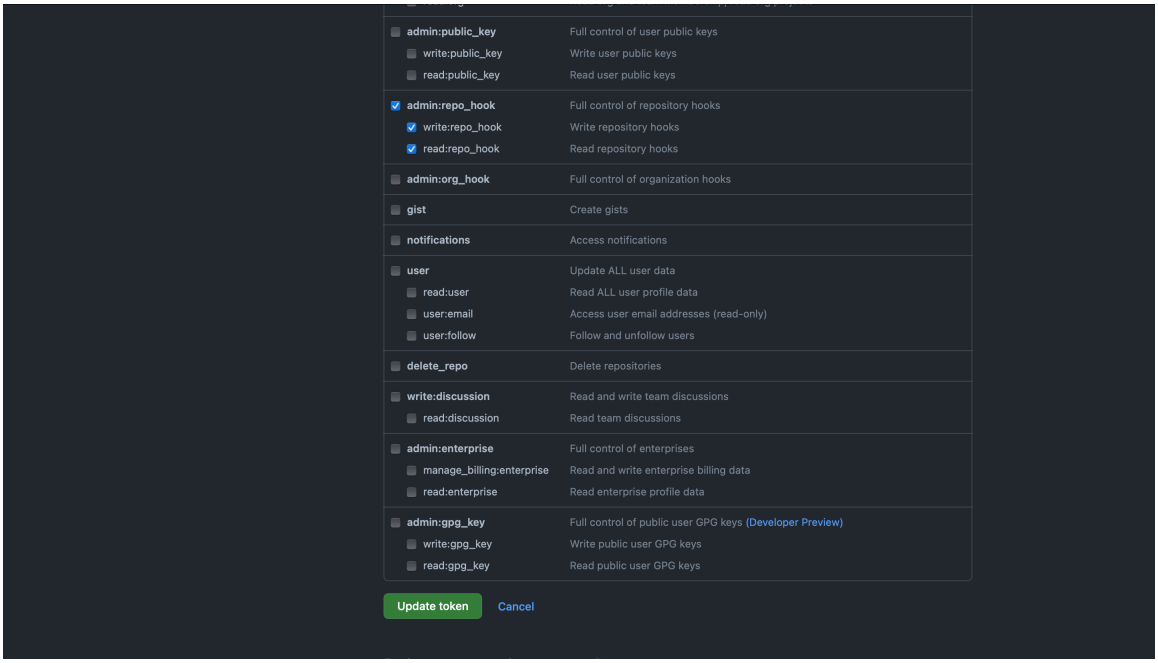

Figura 15 – Tela de salvar um *access token*.

Por último, é necessário selecionar qual o modo que o Jenkins irá interpretar a *pipeline*. No caso desse trabalho, será utilizado um Jenkinsfile e pode ser configurado na etapa de *Build Configurations*, como na Figura 16.

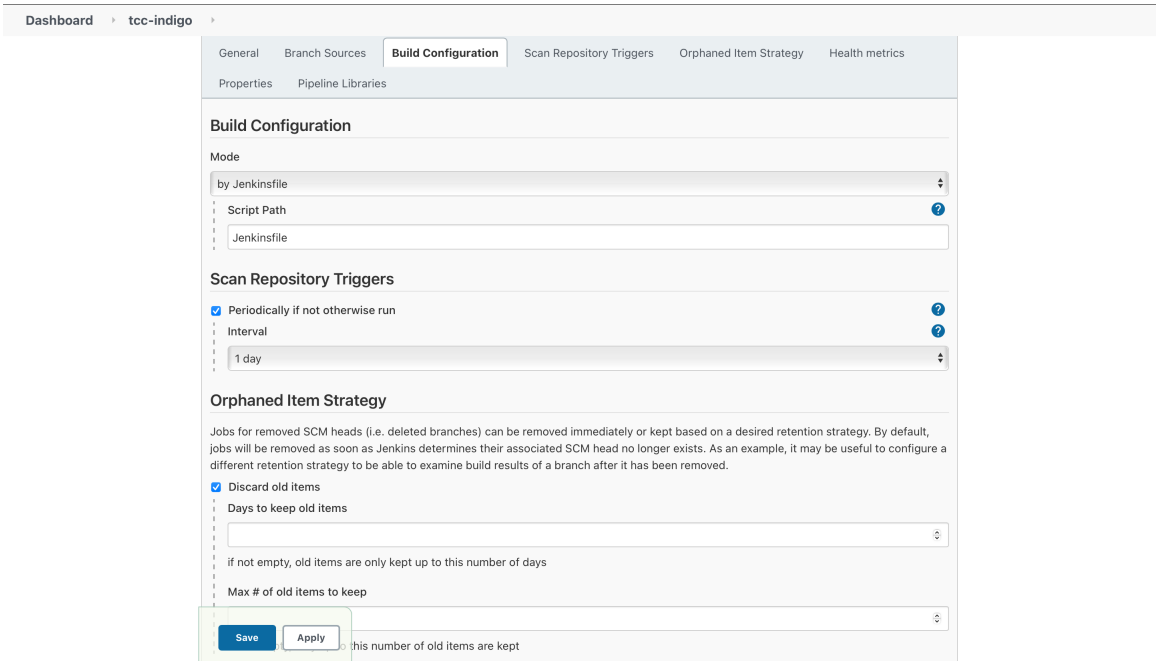

Figura 16 – Tela de selecionar o modo de interpretar a *pipeline*.

4.2.3 Jenkinsfile

Para definir a *pipeline* com suas etapas e configurações, é utilizado um arquivo de configuração chamado de Jenkinsfile. O arquivo na íntegra pode ser encontrado no Apêndice E.

No Jenkinsfile é possível encontrar um *cronjob*, um temporizador que irá disparar a *pipeline* de tempos em tempos, configurado para executar as 7 da manhã de todos os dias da semana.

Os estágios da *pipeline* são configurados para executar com dois gatilhos: um *pull request* para a *main* e um *cronjob* na *dev*.

A *pipeline* é dividida em 8 estágios. Sendo eles:

1. *Pre-Build*: Estágio para configurar a variável de ambiente ANDROID_SERIAL com o dispositivo Android conectado no computador.

2. *Build*: Etapa onde é gerada a *build* do projeto, ou seja, é verificado a folha de estilo e é gerado o Android Application Package (apk) – arquivo de pacotes do aplicativo para instalação.

3. *Install*: Aqui é instalado no celular de teste tanto a versão de *debug* do aplicativo, quanto a versão de *debug* dos testes.

4. *Owner Test*: Nesse estágio é onde rodam os principais testes com a suite OwnerUnitTestSuite.

5. *Admin Test*: Nessa etapa são testadas as funcionalidades com a suite AdminUnitTestSuite.

6. *Guest Online Test*: Aqui são verificadas as funcionalidades com a suite GuestOnlineUnitTestSuite.

7. *Guest Offline Test*: Nesse estágio é onde rodam os testes com a suite GuestOfflineUnitTestSuite.

8. *Owner Final Test*: A última etapa é onde as permissões são apagadas e a fechadura desconfigurada para deixar pronta para a próxima *pipeline*.

A *pipeline* tem também um *fallback* para quando um dos estágios falhar. É um *script* que pega as últimas 100 linhas do *logs* para verificar se teve falhas ou não. Caso ache uma falha, o *script* irá disparar a ErrorUnitTestSuite e um erro fatal do Jenkins para interromper a *pipeline* onde estiver.

4.3 Integração com o GitHub

Ao abrir um *pull request* no GitHub, é necessário informar ao Jenkins que uma nova *pipeline* precisa ser disparada. O caminho inverso também tem que ser possível, ou seja, o Jenkins precisa ficar informando o GitHub sobre o *status* da *pipeline*.

Para fazer essa comunicação entre os dois serviços, é necessário configurar um *webhook*, que é uma API baseada em eventos. Como o Jenkins está executando localmente no computador, foi necessário expor a porta 8080 para um DNS disponível utilizando o ngrok – ferramenta para expor serviços que estão sendo executados localmente para a internet (ROBERTO, 2021) –, como podemos observar na Figura 17.

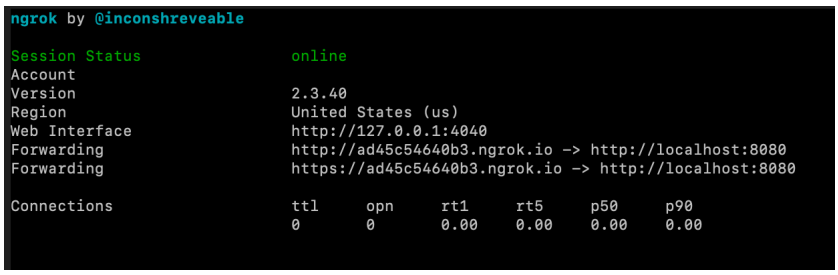

Figura 17 – Configuração do ngrok.

Agora que temos o Jenkins disponível na internet, podemos continuar configurando o *webhook* no GitHub. Para isso vai ser necessário ir em *settings* do projeto, selecionar *webhooks* e assim criarmos um, como podemos observar na Figura 18.

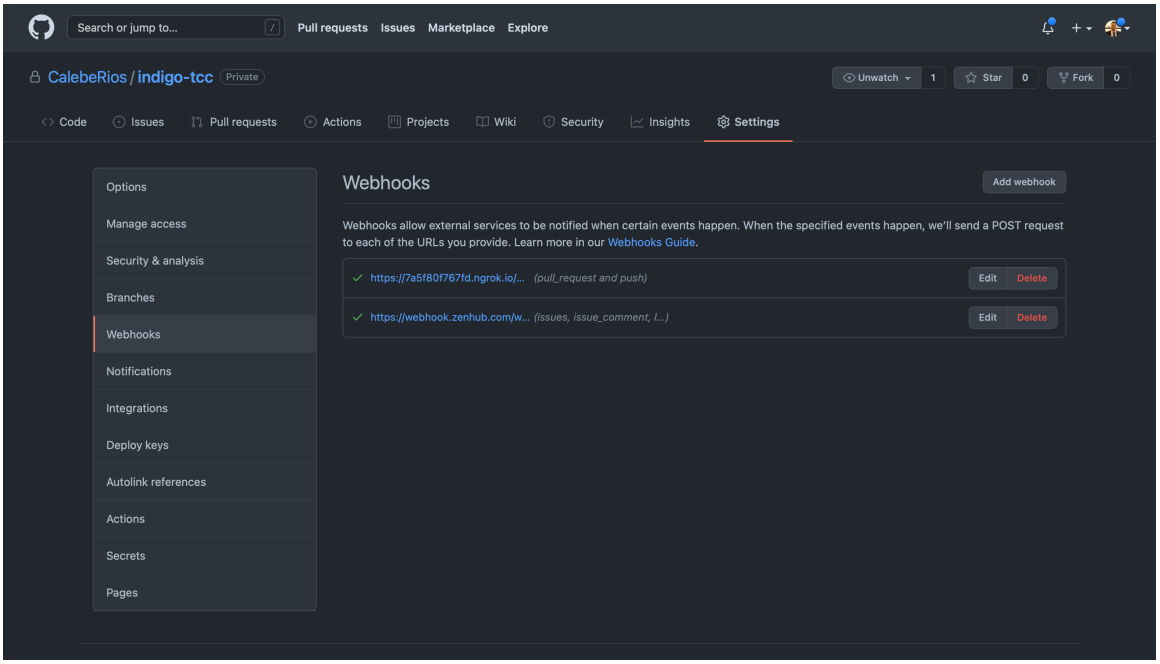

Figura 18 – Lisa de *webhooks* configurados.

Agora é só colocar a URL que o **ngrok** disponibilizou, como na Figura 19.

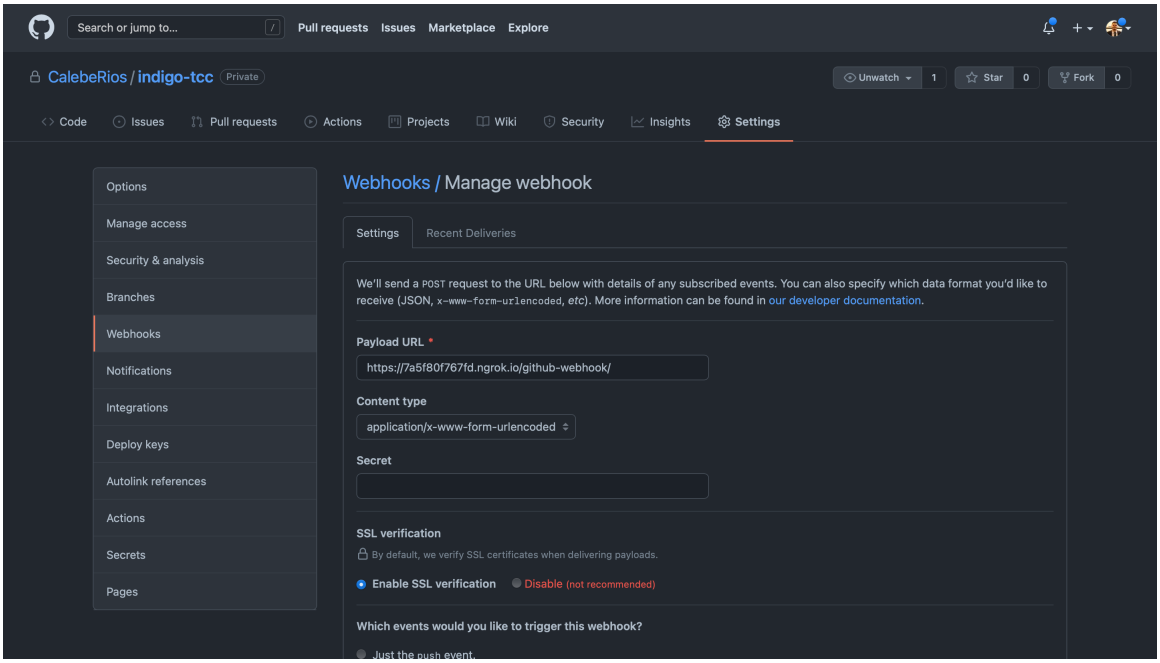

Figura 19 – Configuração do *webhook*.

Para continuar a configuração, é necessário selecionar a opção *Let me select individual events*, como apresentado na Figura 20.

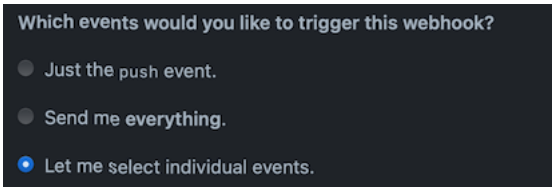

Figura 20 – Opção dos gatilhos do *webhook*.

Ao selecionar a opção para selecionar eventos individuais, irão aparecer várias opções de gatilhos para ser selecionado, os únicos utilizados no projeto são as opções de *pull request* e *pushes*, podemos observar essas opções na Figura 21.

Figura 21 – Gatilhos do *webhook*.

5 Experimentos e Resultados

Foram realizados 8 experimentos com a *pipeline* proposta para demonstrar o pleno funcionamento da bancada e dos testes. Os experimentos, junto com a explicação de cada um, podem ser observados no Quadro 13.

Quadro 13 – Experimentos realizados.

Número	**Experimento**	**O que é**
1	*Pipeline* completa sem erro	Teste completo, com disparo por PR, com todas as etapas funcionando como esperado
2	*Pipeline* completa com erro no *build*	Teste completo, com disparo por PR, com erro na etapa de *build*, ao alterar a folha de estilo
3	*Pipeline* completa com erro no *owner*	Teste completo, com disparo por PR, com erro na etapa de *owner test*, ao alterar uma configuração no comando de configuração da fechadura
4	*Pipeline* completa com erro no *admin*	Teste completo, com disparo por PR, com erro na etapa de *admin test*, ao alterar uma configuração no comando de compartilhamento de permissão
5	*Pipeline* completa com erro no *guest online*	Teste completo, com disparo por PR, com erro na etapa de *guest online test*, ao alterar uma configuração na montagem do comando
6	*Pipeline* completa com erro no *guest offline*	Teste completo, com disparo por PR, com erro na etapa de *guest offline test*, ao alterar uma configuração específica do comando de abertura
7	*Pipeline* completa com erro no *owner* final	Teste completo, com disparo por PR, com erro na etapa de *owner final test*, ao alterar uma configuração no comando de *reset* da fechadura
8	*Pipeline* completa sem erro no *cronjob*	Teste completo, com disparo por temporizador, com todas as etapas funcionando como esperado

Todos os experimentos foram realizados com a mesma configuração do Jenkisfile, sendo alteradas somente partes do código referentes a cada tipo de usuário para forçar um erro nos testes. Eles foram realizados abrindo um *pull request* da *branch dev* para a *main*, com exceção do *cronjob*, em que o experimento aconteceu em um dia diferente e é sempre realizado na *branch dev*.

Ao abrir um PR para a *branch main*, o *merge* é bloqueado enquanto os *checks* não obtiverem sucesso. Na Figura 22 é possível observar tanto o *check* do *review dog*, utilizado para análise de qualidade estática do código, quanto da *pipeline* executando no Jenkins.

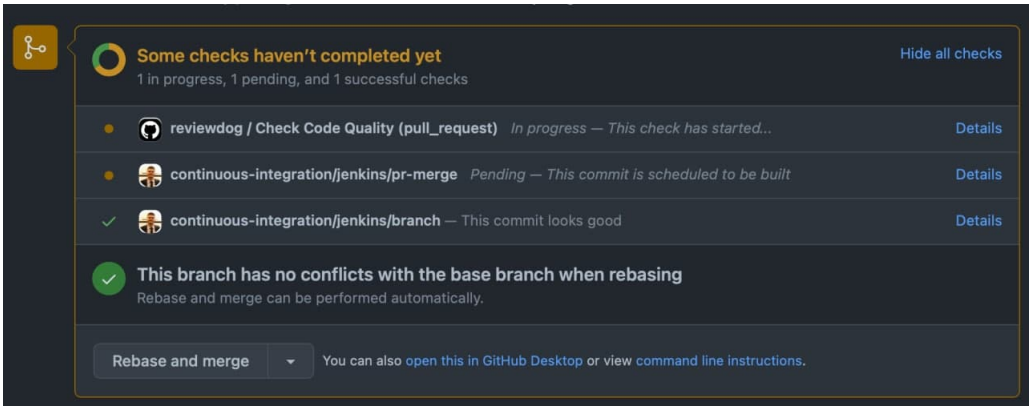

Figura 22 – *Checks* do *Pull Request* executando.

Quando a *pipeline* do Jenkins finalizar, será possível observar tanto no GitHub, Figura 23, quanto no Jenkins, Figura 24. No GitHub, você consegue ser direcionado para a tela do Jenkins onde se encontra o resumo das etapas. Já no Jenkins, além de ver o tempo médio de cada etapa, é possível ler os *logs* detalhados.

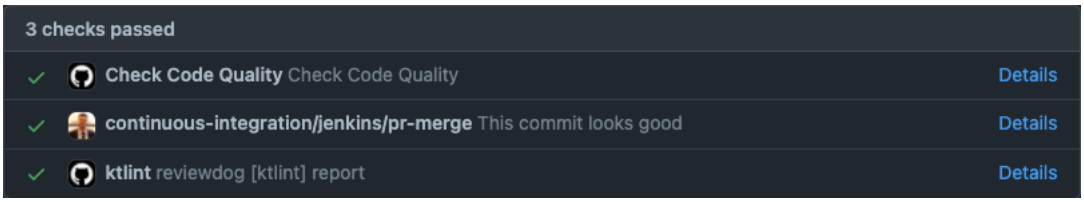

Figura 23 – *Checks* do *Pull Request* completos.

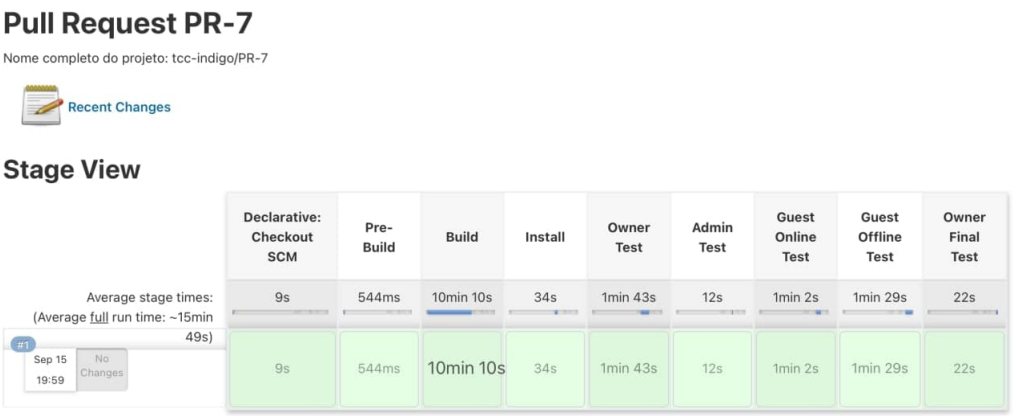

Figura 24 – A *pipeline* funcionando com sucesso.

5.1 *Pipeline* Completa sem Erro

A *pipeline* funcionando, passa por todas as etapas com sucesso. Na Figura 24 acima, é possível observar que todas as etapas passaram com sucesso assim como seus respectivos tempos de execução. A média final de uma *pipeline* funcionando por completo do projeto é de **15 minutos**. Caso uma *pipeline* seja disparada sem alteração no *build*, o Jenkins consegue pegar o *build* do *cache* e reduzir esse tempo total da *pipeline*.

Dentro da ferramenta do Jenkins é possível também observar o *log* gerado pelos testes da *pipeline*. Na Figura 25 é possível observar que o *log* da etapa de *build* foi concluída com sucesso.

```
> Task :app:mergeReleaseGeneratedProguardFiles
> Task :app:compileReleaseSources
> Task :app:mergeReleaseJavaResource
> Task :app:minifyReleaseWithR8
> Task :app:packageRelease

> Task :app:bugsnagReleaseReleaseTask
Bugsnag: Uploading to Releases API

> Task :app:generateBugsnagReleaseMapping

> Task :app:uploadBugsnagReleaseMapping
Bugsnag: Uploading JVM mapping file from: /Users/caleberios/.jenkins/workspace/tcc-indigo_PR-
7/app/build/intermediates/bugsnag/jvmMappings/mappingForRelease.gz

> Task :app:assembleRelease
> Task :app:assemble

Deprecated Gradle features were used in this build, making it incompatible with Gradle 8.0.
Use '--warning-mode all' to show the individual deprecation warnings.
See https://docs.gradle.org/7.0/userguide/command_line_interface.html#sec:command_line_warnings

BUILD SUCCESSFUL in 10m 7s
117 actionable tasks: 116 executed, 1 up-to-date
```

Figura 25 – Os *logs* do *build*.

Já na Figura 26 temos um exemplo do *log* dos testes funcionando corretamente. Nesse *log* temos algumas informação de qual classe e método estão sendo testados naquele momento e, no final, um tempo total da etapa e o número de testes com sucesso.

```
INSTRUMENTATION_STATUS: id=AndroidJUnitRunner
INSTRUMENTATION_STATUS: test=resetDoorTest
INSTRUMENTATION_STATUS: class=br.com.loopkey.indigo.ResetDoorUnitTest
INSTRUMENTATION_STATUS: current=9
INSTRUMENTATION_STATUS_CODE: 1
INSTRUMENTATION_STATUS: numtests=10
INSTRUMENTATION_STATUS: stream=.
INSTRUMENTATION_STATUS: id=AndroidJUnitRunner
INSTRUMENTATION_STATUS: test=resetDoorTest
INSTRUMENTATION_STATUS: class=br.com.loopkey.indigo.ResetDoorUnitTest
INSTRUMENTATION_STATUS: current=9
INSTRUMENTATION_STATUS_CODE: 0
INSTRUMENTATION_STATUS: numtests=10
INSTRUMENTATION_STATUS: stream=
br.com.loopkey.indigo.LogoutUnitTest:
INSTRUMENTATION_STATUS: id=AndroidJUnitRunner
INSTRUMENTATION_STATUS: test=logout
INSTRUMENTATION_STATUS: class=br.com.loopkey.indigo.LogoutUnitTest
INSTRUMENTATION_STATUS: current=10
INSTRUMENTATION_STATUS_CODE: 1
INSTRUMENTATION_STATUS: numtests=10
INSTRUMENTATION_STATUS: stream=.
INSTRUMENTATION_STATUS: id=AndroidJUnitRunner
INSTRUMENTATION_STATUS: test=logout
INSTRUMENTATION_STATUS: class=br.com.loopkey.indigo.LogoutUnitTest
INSTRUMENTATION_STATUS: current=10
INSTRUMENTATION_STATUS_CODE: 0
INSTRUMENTATION_RESULT: stream=

Time: 19,608

OK (10 tests)
```

Figura 26 – O *log* dos testes.

Essa etapa da *pipeline* foi testada 30 vezes para garantir a confiabilidade da *pipeline* durante o dia a dia da equipe. Após os testes foi percebido que a taxa de falha era baixa (por se tratar de comunicação BLE, tinha vezes que alguns testes falhavam mesmo sem alteração no código), e mesmo nas falhas, após fazer um *retry* manual, a *pipeline* funcionava como esperado.

5.2 *Pipeline* Completa com Erro no *Build*

Para conseguir quebrar a *build* do projeto, foi alterada uma parte do código que não estivesse de acordo com a folha de estilo do *ktlint*. Nesse caso, o *build* retorna um erro e não completa o restante da *pipeline*, como podemos observar na Figura 27.

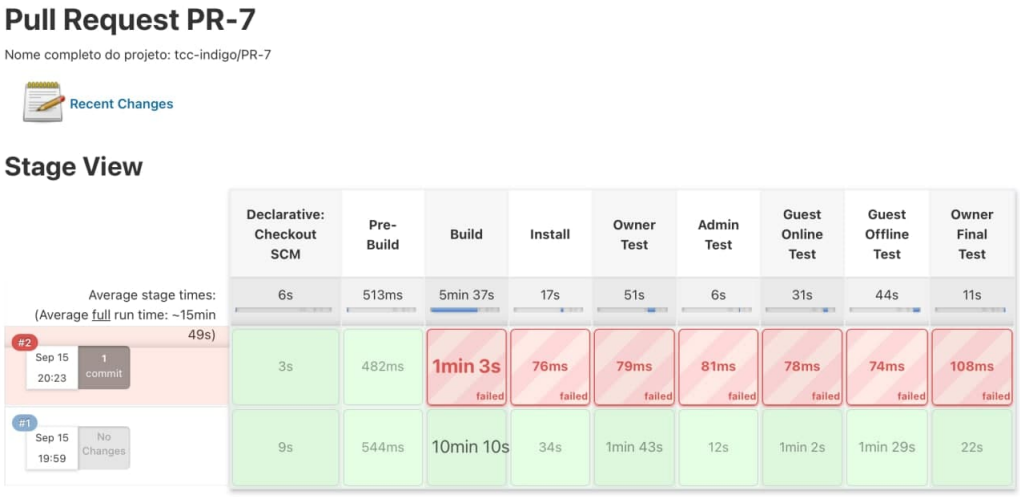

Figura 27 – Erro no Build.

No caso do *build*, não vai ter tratamento de *fallback* caso essa etapa retorne erro, já que ela não tem ligação com os testes de BLE.

Como foi alterado algo da folha de estilo, podemos observar na Figura 28 que o *check* do reviewdog também falhou, como esperado.

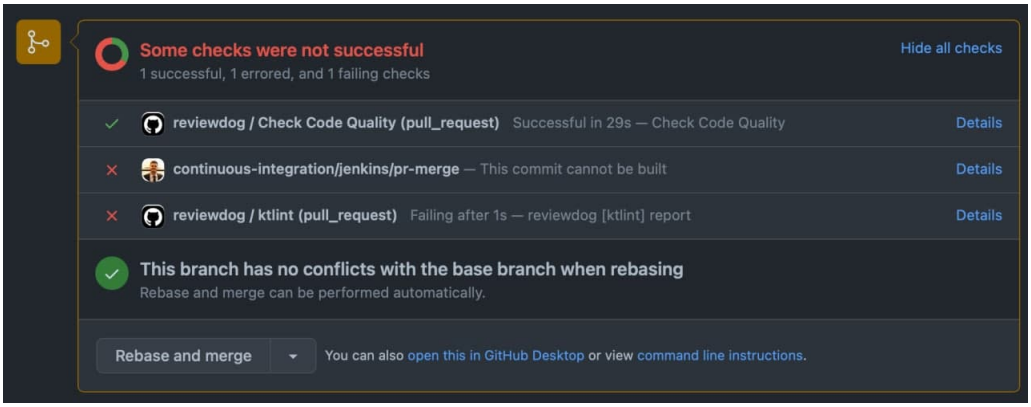

Figura 28 – Erro na folha de estilo.

No *log* da Figura 29, é apresentado o arquivo junto da linha e da coluna onde se encontra o erro que resultou na falha.

```
WARNING:: Please remove usages of `jcenter()` Maven repository from your build scripts and migrate your build to other Maven
repositories.
This repository is deprecated and it will be shut down in the future.
See http://developer.android.com/r/tools/jcenter-end-of-service for more information.
Currently detected usages in: root project 'tcc-indigo_PR-7', project ':app'
> Task :clean
> Task :app:clean

> Task :app:ktlint
/Users/caleberios/.jenkins/workspace/tcc-indigo_PR-7/app/src/androidTest/java/br/com/loopkey/indigo/suite/AdminUnitTestSuite.kt:5:1:
Unused import

> Task :app:ktlint FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':app:ktlint'.
> Process 'command '/Library/Java/JavaVirtualMachines/jdk1.8.0_281.jdk/Contents/Home/bin/java'' finished with non-zero exit value 1

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output. Run with --scan to get
full insights.

* Get more help at https://help.gradle.org

Deprecated Gradle features were used in this build, making it incompatible with Gradle 8.0.
Use '--warning-mode all' to show the individual deprecation warnings.
See https://docs.gradle.org/7.0/userguide/command_line_interface.html#sec:command_line_warnings

BUILD FAILED in 1m 1s
3 actionable tasks: 3 executed
```

Figura 29 – *Log* de erro do *build*.

5.3 Erro no *Owner*

Na etapa de *owner*, além dos testes normais do modo *timer* e de abertura da fechadura, é realizada também a configuração da fechadura e o compartilhamento do acesso com um *admin*, cartão e senha. A configuração da fechadura é quem define o *owner*, ou seja, quem configurou se torna o dono daquela fechadura.

O teste de configuração demanda dois comandos BLE e uma requisição para a API, por ser uma configuração mais extensa; foi utilizado um *timeout* de 30 segundos nela.

Ao remover uma variável importante para a configuração da porta, a etapa do *owner* falha e não completa o restante da *pipeline*. Na Figura 30, podemos observar essa falha junto do tempo que durou aquela etapa. A duração ficou com um pouco mais de 5 minutos pois mesmo ocorrendo erro na configuração da fechadura, ele ainda tenta realizar os próximos testes dessa suite de teste, pois o *fallback* só é chamado no final da etapa.

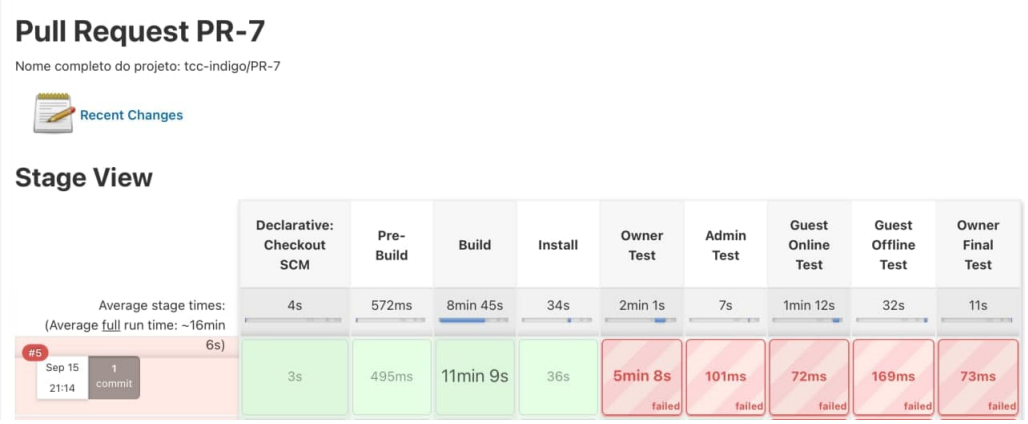

Figura 30 – Erro na etapa do *owner*.

A etapa de *owner* tem o *fallback* para restaurar a fechadura para as configurações de fábrica.

5.4 Erro no *Admin*

A etapa de *admin* possui somente um teste BLE, que é o teste de abertura. Isso se dá porque, para a fechadura LoopKey, o *owner* e o *admin* não têm nenhuma diferença, pois os dois têm as mesmas chaves, ou seja, possuem as mesmas permissões. A diferença dos dois se dá pelo lado do aplicativo, sendo o único diferencial que o *admin* não pode desconfigurar ou editar uma porta.

O outro teste realizado nessa etapa é o compartilhamento com *guest online* e *offline*.

Como para a fechadura o *owner* e o *admin* têm as mesmas chaves, não foi possível alterar algo para falhar os testes de *admin* no único teste de comando BLE. Nesse caso, foi realizada uma alteração no compartilhamento com um *guest* para forçar a falha. Na Figura 31, podemos ver a falha dessa etapa.

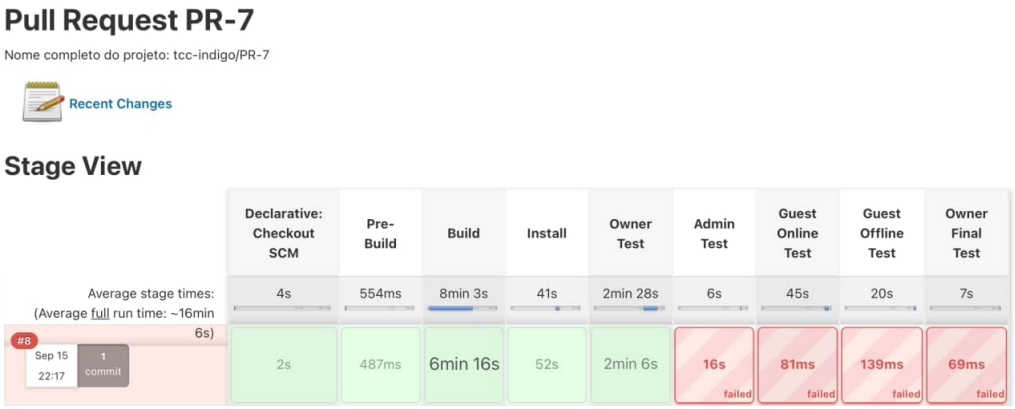

Figura 31 – Erro na etapa do *admin*.

Essa etapa também possui um *fallback* para restaurar as configurações de fábrica da fechadura.

5.5 Erro no *Guest Online*

O *guest online* é o único tipo de usuário que não possui nenhuma chave da fechadura, então para realizar os comandos BLE, ele precisa solicitar a mensagem do comando já autenticada para o servidor. Para esse tipo de usuário é obrigatória a conexão com a internet. Os testes realizados nessa etapa são o de abertura e os relacionados ao modo *timer*.

Já que a mensagem do comando BLE já vem montado pelo servidor, foi alterada uma variável que é enviada para o servidor, para assim forçar a falha.

A Figura 32 apresenta a falha que ocorre nos testes de *guest online*.

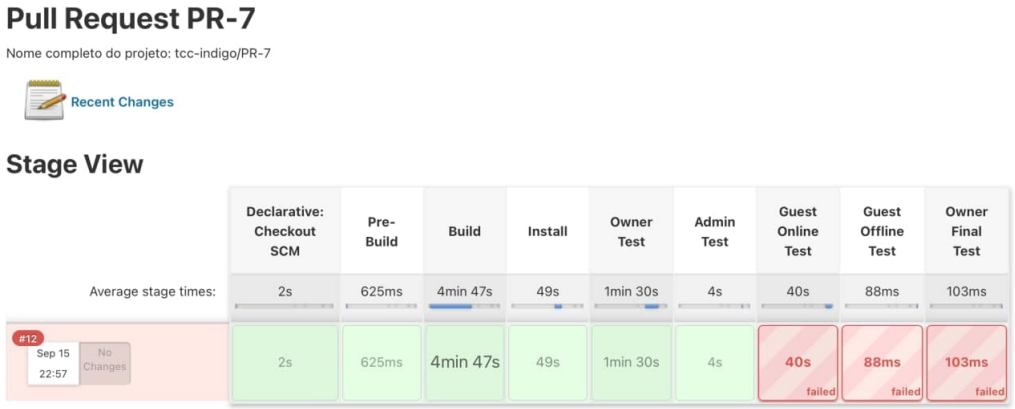

Figura 32 – Erro na etapa do *guest online*.

Essa etapa também possui um *fallback* para as restaurar as configurações de fábrica da fechadura.

5.6 Erro no *Guest Offline*

O *guest offline* possui somente uma das chaves da fechadura, então, diferente do *guest online*, ela não precisa de uma conexão com a internet para montar e enviar os comandos BLE. Por ter somente uma das chaves, são realizados somente os testes de abertura e os relacionados ao modo *timer*.

Para forçar a falha no comando de abertura nessa etapa, foi necessário alterar uma variável específica do *guest offline*, já que se alterasse uma configuração do comando, os outros tipos de usuários não iriam conseguir realizar tal comando.

A Figura 33 apresenta o erro nessa etapa, como esperado.

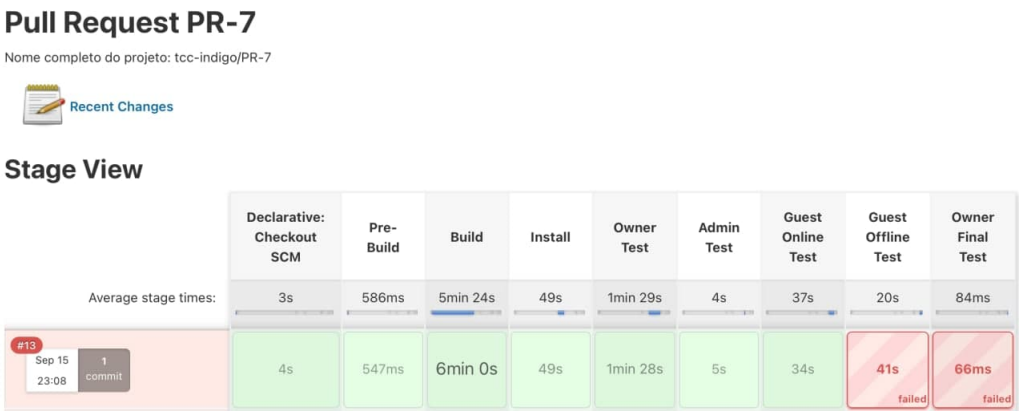

Figura 33 – Erro na etapa do *guest offline*.

Essa etapa também possui um *fallback* para as restaurar as configurações de fábrica da fechadura.

5.7 Erro no *Owner* Final

A etapa final é realizada novamente com o *owner*, pois nessa etapa é feita a restauração de fábrica da fechadura. Além da redefinição, também é testada a remoção dos compartilhamentos de cartão, senha e de todos os tipos de usuário, mas esse último não é um comando BLE como os outros.

Foi alterado o comando de *reset* da fechadura para conseguir forçar o erro dessa etapa com podemos ver na Figura 34.

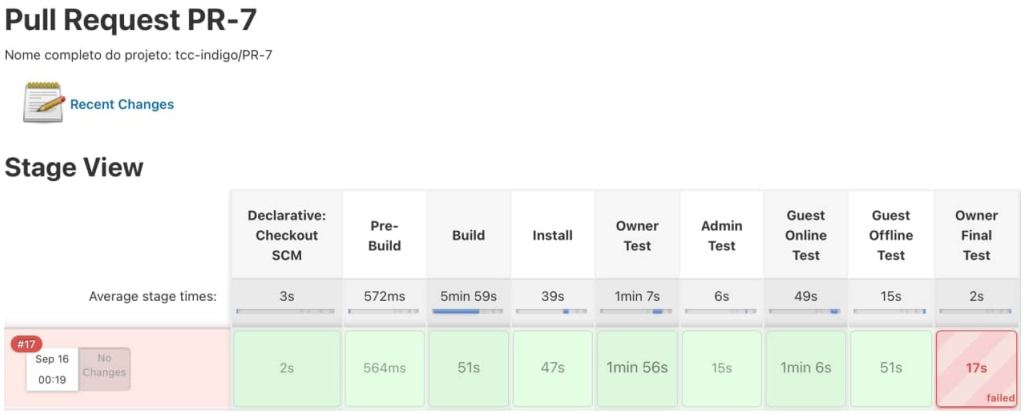

Figura 34 – Erro na etapa do *owner final*.

Essa etapa também possui um *fallback* para as restaurar as configurações de fábrica da fechadura.

5.8 *Cronjob*

O único experimento realizado diferente dos demais foi o do *cronjob*. A *pipeline* foi programada para executar as 7 da manhã em todos os dias da semana. Como podemos observar na Figura 35, a tarefa está sendo executada exatamente como programado e todas as etapas da *pipeline* passaram como esperado.

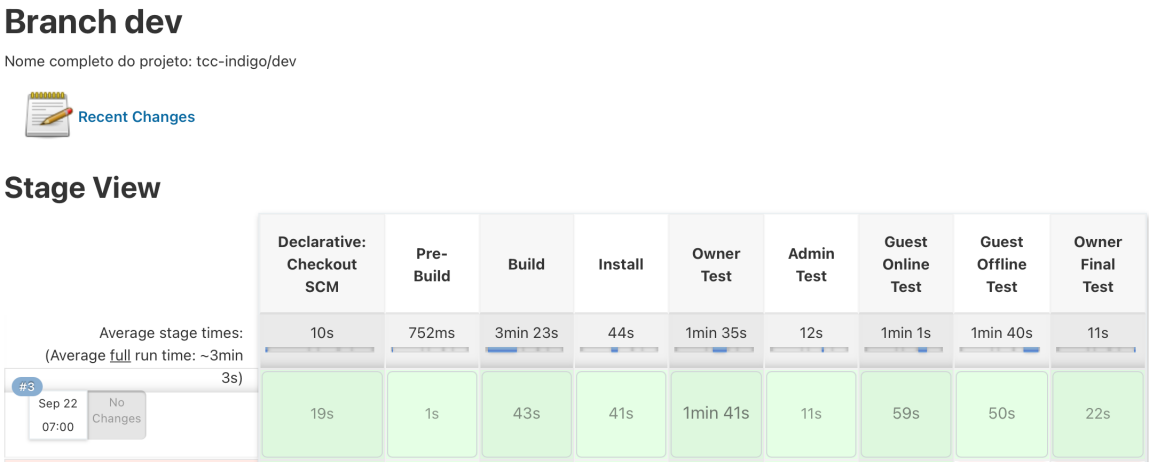

Figura 35 – Resumo da *pipeline* do *cronjob*.

6 Discussões

Após todas as configurações e os experimentos concluídos, foram analisados os resultados e coletadas as métricas. Neste capítulo, serão explicados os principais resultados coletados dos experimentos, além das métricas.

6.1 Experimentos

Foram realizados os experimentos pensando em todos as possibilidades que podem ocorrer no dia a dia do desenvolvimento. Além do caminho feliz, que seria a *pipeline* funcionando sem problemas, foi testado também o erro em cada etapa para garantir que os erros são mostrados para o usuário corretamente.

No caso do caminho feliz, tudo ocorreu como esperado, mas foi percebido que, em algumas vezes, ocorriam erros inesperados por conta da comunicação BLE dos comandos. Mesmo com esses erros, foi observado que eles ocorriam em uma taxa pequena (de 30 testes, somente 2 tiveram erros inesperados). Mas, mesmo com a taxa pequena, seria interessante a investigação desses erros e até uma possível melhora nos testes ou comandos para evitar esses erros inesperados.

Outro ponto de atenção é quanto aos tempos totais da *pipeline* completa, os quais ficaram com uma média de 15 minutos, como observado na Figura 24 acima. É possível observar que 2/3 desse tempo, ou seja, 10 minutos, são utilizados para o *build* do projeto. Esse tempo pode ser reduzido se forem investigadas as possíveis configurações desnecessárias do projeto. Outra possível solução seria a utilização de uma ferramenta de *cache* de *build*, que são utilizados para diminuir o tempo.

Os outros 5 minutos da *pipeline* são utilizados pelos comandos, e isso é um tempo considerável bom, já que tem que esperar a resposta do comando BLE e em algumas vezes ocorre erro de *timeout*.

Os experimentos com falha em alguma etapa da *pipeline* também ocorreram como esperado, já que, ao realizar uma alteração no código dedicado a cada etapa, ele falhava e retornava para o usuário onde ocorreu a falha.

Além dos experimentos demonstrados acima, foi testada também a situação onde o serviço do Jenkins está fora do ar e é aberto um *pull request* para a *branch main*. Como esperado, o *pull request* dá erro e não é possível verificar a *pipeline*, nesse caso, o PR fica preso até o serviço do Jenkins voltar. Nessa situação, se o um PR for urgente e necessário o *merge* imediato, um administrador do repositório pode fazer um *force merge* e ignorar o erro do Jenkins.

6.2 Métricas

Levando em consideração as métricas definidas na Seção 2.8.1, temos o resultado na Tabela 1.

Tabela 1 – Resultados da Medição.

Métrica	**Indicador**
Métrica 1 - Cobertura dos *status* dos testes	100%
Métrica 2 - Cobertura das principais funcionalidades	66,7%
Métrica 3 - Automatização dos Testes ao Abrir PR	Sim
Métrica 4 - Testes de Comando BLE	Sim

A métrica de cobertura dos *status* dos testes foi coletada a partir da ferramenta do Jenkins, e de forma implícita, ou seja, como todos os testes passaram na *pipeline* completa, foi entendido que todos os testes retornaram sucesso. Como todos os testes passaram, temos uma cobertura de 100%, então esse indicador está ótimo.

A métrica de cobertura das principais funcionalidades é de 66,7%; nesse caso, o indicador está regular, por estar entre 50% e 80%. O resultado dessa métrica se deu por não realizar alguns testes considerados principais. No Quadro 14, é possível ver os comandos não testados e os motivos de não terem sido incluídos na *pipeline*.

Quadro 14 – Principais comandos BLE que não foram testados.

Número	**Caso de teste**	**Motivo**
1	Compartilhar acesso da fechadura com uma biometria	Este comando é necessário a presença de uma pessoa para adicionar manualmente a digital
2	Abrir uma fechadura de forma remota	Para realizar esse comando é necessário o *gateway*; e como não está no escopo desse trabalho, ele não está sendo testado
3	Ler a bateria da fechadura	Esse comando é realizado sempre ao realizar uma abertura ou comandos relacionados ao modo *timer*, e por isso não foi considerado esse comando
4	Atualizar o horário e o *timezone*	Esse comando está incluído dentro da edição da fechadura, por isso não foi considerado esse comando
5	Realizar a atualização do *firmware*	Esse comando estava em desenvolvimento e testes; por isso não foi incluído no escopo do trabalho

A métrica de automatização dos testes, ao abrir um PR, é medida de forma subjetiva a partir da observação. Durante os experimentos, foi observado que, ao abrir um PR para a *main*, acionava a ferramenta de CI para executar a *pipeline* corretamente.

Por último, a métrica de testes de comando BLE também é medida de forma subjetiva, a partir da observação. Foi observado que os testes automatizados, executados pela ferramenta de integração contínua, foram capazes de enviar comandos BLE para a fechadura de testes.

7 Considerações Finais

A proposta deste trabalho foi construir uma bancada de testes automatizados para comandos BLE entre um aplicativo móvel e a fechadura eletrônica da LoopKey, a fim de garantir o pleno funcionamento do aplicativo num ambiente real do dia a dia.

Após estudos e definição do escopo para o desenvolvimento da bancada, foram desenvolvidos todos os testes do aplicativo que utilizam comandos BLE, que totalizam 16 testes. A etapa de teste é a principal etapa, pois é nela que é garantido que o aplicativo está funcionando como esperado.

A montagem da bancada foi realizada logo depois do desenvolvimento dos testes e a configuração da ferramenta Jenkins. A bancada foi importante para realizar todos os experimentos e coletar os resultados gerados por ela. A bancada que pode ser utilizada em produção, não tem diferença da bancada utilizada para os experimentos deste TCC.

Depois de realizar os experimentos, foi constatado que tudo está funcionando como esperado. A *pipeline* consegue capturar algum erro nos testes e avisar aos desenvolvedores onde o problema se encontra. Com eles, foi possível demonstrar o funcionamento completo da *pipeline* e também das falhas que possam ocorrer em cada etapa.

Recomenda-se a utilização da bancada de testes em um ambiente isolado para que os equipamentos não sejam alterados, além de ter uma boa conexão, sem interrupção, com a internet. A bancada já pode ser utilizada em produção e é recomendado o seu uso para garantir o funcionamento das novas versões.

7.1 Trabalhos Futuros

Para o futuro deste TCC, foram observados alguns pontos de melhoria. O primeiro é em relação à melhoria do tempo da *pipeline* completa. A etapa de *build* consome 2/3 do tempo total, sendo que é uma configuração para gerar o apk da aplicação. É recomendada a utilização de um equipamento com maior poder de processamento para hospedar o serviço do Jenkins ou a utilização de uma ferramenta de *cache* para evitar algumas configurações desnecessárias da etapa do *build*.

Foi percebida também a necessidade de uma melhoria nos testes desenvolvidos, já que os testes retornam erros inesperados uma vez ou outra. O teste pode ser melhorado para ter uma tratativa de *retry* melhor, ou serem revisados os comandos na LKLib ou no aplicativo para garantir que foi desenvolvido da melhor forma.

A grande dependência da *pipeline* com o teste de desconfiguração é muito grande,

pensando nisso é importante que também se tenha uma *fallback* para essa situação, para que a *pipeline* não fique bloqueada. Junto com esse problema, também se vê a utilidade de configurar o slack, uma plataforma de comunicação, para enviar mensagens para a equipe com os sucessos e falhas da *pipeline*. Sendo uma dessas mensagens uma destinada para a falha da desconfiguração, assim fica fácil a necessidade de um *reset* manual.

Os testes realizados durante o TCC podem não ser o suficiente a longo prazo, para isso é recomendado também a criação de testes da biblioteca da LoopKey e também testes de *firmware* através de um simulador. Os testes da biblioteca são importantes já que a LoopKey disponibiliza ela para parceiros, já os testes de *firmware* é uma garantia do funcionamento da fechadura em si.

Referências

AFANEH, M. *The Basics of Bluetooth Low Energy (BLE)*. 2016. Disponível em: <https://www.novelbits.io/basics-bluetooth-low-energy/>. Acesso em: 07 mar 2021. Citado na página 28.

AFANEH, M. *Intro to Bluetooth Low Energy v1.1*. 1.1. ed. [S.l.]: Novel Bits, 2018. 110 p. Citado 2 vezes nas páginas 28 e 30.

AMANNEJAD, Y.; BETIN, A. Software test-code engineering : A systematic mapping. v. 58, p. 123–147, 2015. Citado 2 vezes nas páginas 24 e 32.

Android. *Arquitetura da plataforma*. 2020. Disponível em: <https://developer.android.com/guide/platform?hl=pt-br>. Acesso em: 27 mar 2021. Citado 4 vezes nas páginas 9, 34, 35 e 36.

Android. *Conceitos básicos de testes*. 2020. Disponível em: <https://developer.android.com/training/testing/fundamentals?hl=pt-br>. Acesso em: 30 mar 2021. Citado na página 36.

BASSIL, Y. A Simulation Model for the Waterfall Software Development Life Cycle. v. 2, n. 5, 2012. Disponível em: <http://arxiv.org/abs/1205.6904>. Citado na página 30.

BECK, K. et al. *Manifesto for Agile Software Development*. 2001. Citado 2 vezes nas páginas 31 e 43.

Bluetooth Special Interest Group (SIG). Bluetooth Core Specification Version 4.0 Master Table of Contents & Compliance Requirements. *Bluetooth SIG*, v. 0, n. June, p. 2302, 2010. Disponível em: <https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=456433>. Citado na página 28.

Clubhouse. *About Clubhouse*. 2020. Disponível em: <https://clubhouse.io/about>. Acesso em: 09 abr 2021. Citado na página 46.

DINIZ, A. B. *Debian Cloud Images Finder*. [S.l.], 2020. Citado na página 45.

EBERT, C. et al. DevOps. *IEEE Software*, v. 33, n. 3, 2016. ISSN 19374194. Citado 5 vezes nas páginas 21, 23, 31, 32 e 33.

GARG, S.; BALIYAN, N. Comparative analysis of Android and iOS from security viewpoint. *Computer Science Review*, v. 40, 2021. ISSN 15740137. Citado 2 vezes nas páginas 34 e 35.

GIL, A. C. *Como elaborar projetos de pesquisa*. 4.ed.. ed. São Paulo: Editora Atlas, 2002. 57 p. ISBN 85-224-3169-8. Citado 2 vezes nas páginas 41 e 42.

Github. *Github*. 2021. Disponível em: <https://github.com>. Acesso em: 09 abr 2021. Citado na página 46.

Jenkins. *Jenkins*. 2021. Disponível em: <https://www.jenkins.io>. Acesso em: 09 abr 2021. Citado na página 46.

Jenkins. *macOS Installers for Jenkins LTS*. 2021. Disponível em: <https://www.jenkins.io/download/lts/macos/>. Acesso em: 20 abr 2021. Citado na página 60.

JENKINS, J. *Velocity 2011: Jon Jenkins, "Velocity Culture*

. 2011. Disponível em: <https://www.youtube.com/watch?v=dxk8b9rSKOo>. Acesso em: 11 mar 2021. Citado na página 31.

JEONG, H. D. J. et al. Utilizing a Bluetooth remote lock system for a smartphone. *Pervasive and Mobile Computing*, v. 24, 2015. ISSN 15741192. Citado 2 vezes nas páginas 27 e 28.

LoopKey. *LK Bio Specification*. 2021. Disponível em: <https://www.loopkey.com.br/loopkey-bio/>. Acesso em: 04 abr 2021. Citado na página 38.

LoopKey. *LK Mini Specification*. 2021. Disponível em: <https://www.loopkey.com.br/mini/>. Acesso em: 04 abr 2021. Citado na página 38.

LWAKATARE, L. E. et al. DevOps in practice: A multiple case study of five companies. *Information and Software Technology*, v. 114, n. March 2017, p. 217–230, 2019. ISSN 09505849. Citado 4 vezes nas páginas 9, 31, 32 e 33.

MATHIAS, C. Bluetooth is dead. *EETimes*, 2003. Disponível em: <https://www.eetimes.com/bluetooth-is-dead/#>. Citado na página 27.

MATHRANI, A.; MATHRANI, S. Test strategies in distributed software development environments. *Computers in Industry*, v. 64, n. 1, p. 1–9, 2013. ISSN 01663615. Citado na página 23.

MEDEIROS, M. P. *Extreme Programming – Conceitos e Práticas*. 2006. Disponível em: <https://www.devmedia.com.br/extreme-programming-conceitos-e-praticas/1498>. Acesso em: 09 abr 2021. Citado na página 44.

MEYER, M. Continuous integration and its tools. *IEEE Software*, v. 31, n. 3, p. 14–16, 2014. ISSN 07407459. Citado na página 32.

MYERS, G. J.; BADGETT, T.; SANDLER, C. *The Art of Software Testing*. Third edit. [S.l.: s.n.], 2012. 240 p. ISSN 1098-6596. ISBN 9788578110796. Citado 7 vezes nas páginas 13, 21, 23, 24, 25, 26 e 27.

PCTec-UnB. *Histórico*. 2021. Disponível em: <http://www.pctec.unb.br/institucional/historico>. Acesso em: 05 abr 2021. Citado na página 37.

PESSOA, L. *Introdução ao Bluetooth Smart (BLE)*. 2016. Disponível em: <https://www.embarcados.com.br/bluetooth-smart-ble/>. Acesso em: 07 mar 2021. Citado na página 28.

RIBEIRO, D. *Número de celulares conectados é maior que a população mundial*. 2019. Disponível em: <https://www.consumidormoderno.com.br/2019/06/14/celulares-populacao-mundial/>. Acesso em: 27 mar 2021. Citado na página 34.

ROBERTO, J. *Ngrok: do Localhost para o Mundo*. 2021. Disponível em: <https://medium.com/desenvolvendo-com-paixao/ngrok-do-localhost-para-o-mundo-5445ad08419>. Citado na página 67.

SCHWABER, K.; SUTHERLAND, J. The Scrum Guide. *Software in 30 Days*, n. November, p. 133–152, 2015. Citado na página 43.

SPENCE, E. *The Mobile Browser Is Dead, Long Live The App*. 2014. 4–7 p. Disponível em: <http://www.forbes.com/sites/ewanspence/2014/04/02/the-mobile-browser-is-dead-long-live-the-app/>. Acesso em: 25 mar 2021. Citado na página 34.

TAYLOR, T. *6 MOST COMMONLY USED IOT COMMUNICATION PROTOCOLS*. 2019. Disponível em: <http://techgenix.com/iot-communication-protocols/>. Acesso em: 06 mar 2021. Citado na página 27.

WIKLUND, K. et al. *Impediments for software test automation: A systematic literature review*. 2017. Citado 2 vezes nas páginas 21 e 32.

Apêndices

APÊNDICE A – *Backlog* do Produto

Requisitos	Histórias de Usuário
R1 - Escrever os testes das funcionalidades nas duas plataformas	US1 - Escrever teste do comando de *unlock*
	US2 - Escrever teste do comando de configuração
	US3 - Escrever teste do comando de edição
	US4 - Escrever teste do comando de pegar o nível da bateria
	US5 - Escrever teste do comando de verificar o *timer*
	US6 - Escrever teste do comando de configurar *timer*
	US7 - Escrever teste do comando de cancelar um *timer*
	US8 - Escrever teste de compartilhar permissão por email
	US9 - Escrever teste do compartilhar permissão por telefone
	US10 - Escrever teste do comando de criar permissão de cartão
	US11 - Escrever teste do comando de criar permissão de senha
	US12 - Escrever teste de remover permissão por email
	US13 - Escrever teste de remover permissão por telefone
	US14- Escrever teste do comando de remover permissão de cartão
	US15- Escrever teste do comando de remover permissão de senha
	US16- Escrever teste do comando de desconfigurar uma porta
R2 - Montar uma bancada com o servidor, um iPhone, um Android, uma LK Bio e uma LK Mini	US17 - Montar a bancada com o servidor, um iPhone, um Android, uma LK Bio e uma LK Mini
R3 - Configurar o Jenkins no servidor	US18 - Configurar o Jenkins para testes
	US19- Configurar o Jenkins no servidor
	US20 - Configurar a integração com o GitHub
R4 - O Jenkins deve ser capaz de iniciar uma pipeline sempre que um PR para a *main* for aberto	US21 - Rodar a *pipeline* sempre que abrir um PR para a *main*
R5 - O Jenkins deve ser capaz de iniciar uma pipeline periodicamente	US22 - Configurar o agendamento da *pipeline durante a semana*
	US23 - Configurar o agendamento da *pipeline durante o final de semana*
R6 - O Jenkins deve ser capaz de se recuperar e retornar a fechadura/controladora para o estado inicial caso haja erros na *pipeline*	US24 - Recuperar o estado inicial da fechadura e do app quando ocorrer um erro na *pipeline*
	US# - Recuperar o estado inicial da fechadura e do app através de um *trigger externo*
R7 - O Jenkins deve ser capaz de informar o *status* da *pipeline* para o GitHub	US25 - Mostrar o *status* da pipeline no GitHub
	US26 - Bloquear/Aprovar um PR de acordo com o resultado da *pipeline*

Figura 36 – *Backlog* do Produto

APÊNDICE B – Estrutura da regra de *retry*

```
1  class RetryRule(private val retryCount: Int) : TestRule
2  {
3      override fun apply(base: Statement,
4                         description: Description): Statement
5      {
6          return statement(base, description)
7      }
8
9      private fun statement(base: Statement,
10                           description: Description): Statement
11     {
12         return object : Statement() {
13             @Throws(Throwable::class)
14             override fun evaluate()
15             {
16                 var caughtThrowable: Throwable? = null
17                 // implement retry logic here
18                 for (i in 0 until retryCount) {
19                     try {
20                         base.evaluate()
21                         return
22                     } catch (t: Throwable) {
23                         caughtThrowable = t
24                     }
25                 }
26                 throw Objects.requireNonNull(caughtThrowable)!!
27             }
28         }
29     }
30 }
```

APÊNDICE C – Estrutura da BaseUnitTest

```
1  abstract class BaseUnitTest : KoinComponent, CoroutineScope,
       ReachableDevices.Listener
2  {
3      //Declaração das variáveis necessárias para os testes.
4      @get:Rule
5      var retryRule: TestRule? = RetryRule(4)
6
7      @Before
8      open fun tearUp()
9      {
10         /**
11         * Método para iniciar configurações e pegar informações
       importantes, como:
12         *  - Pega o usuário logado no banco de dados;
13         *  - Pega a lista de portas do banco de dados;
14         *  - Pega uma porta, a partir da lista, de acordo com o
       serial informado anteriormente;
15         *  - Da um subscribe no ReachableDevices.
16         */
17     }
18
19     @After
20     open fun tearDown()
21     {
22         // Da um unsubscribe no ReachableDevices
23     }
24
25     override fun onRechableDevice()
26     {
27         // Verificar se o dispositivo está em alcance
28     }
29
30     private suspend fun _waitReachableDevice(): LKCommDevice?
31     {
32         // Função com retry para retornar o dispositivo selecionado,
       caso ele esteja em alcance.
33     }
34 }
```

APÊNDICE D – Estrutura dos Testes de Comando

```
1  class CommandTest : BaseUniteTest()
2  {
3      // Faz o inject do interactor necessário para realizar o teste.
4
5      @Test
6      fun test()
7      {
8          /**
9          * Chama a função do comando da interactor desejada e
10         * utiliza a biblioteca Awaitility para esperar um sucesso.
11         */
12     }
13 }
```

APÊNDICE E – Jenkinsfile

```
pipeline {
    agent any

    triggers { cron("0 7 * * 1-5") }

    stages {
        stage('Pre-Build') {
            when {
                anyOf {
                    environment name: 'CHANGE_TARGET', value: 'main'
                    allOf {
                        branch 'dev'
                        triggeredBy "TimerTrigger"
                    }
                }
            }
            steps {
                sh 'export ANDROID_SERIAL=$(adb devices | grep -w "device" | cut -f 1)'
            }
        }
        stage('Build') {
            when {
                anyOf {
                    environment name: 'CHANGE_TARGET', value: 'main'
                    allOf {
                        branch 'dev'
                        triggeredBy "TimerTrigger"
                    }
                }
            }
            steps {
                sh './gradlew assemble'
```

```
33              }
34          }
35          stage('Install') {
36              when {
37                  anyOf {
38                      environment name: 'CHANGE_TARGET', value: '
    main'
39                      allOf {
40                          branch 'dev'
41                          triggeredBy "TimerTrigger"
42                      }
43                  }
44              }
45              steps {
46                  sh './gradlew installDebug'
47                  sh './gradlew installDebugAndroidTest'
48              }
49          }
50          stage('Owner Test') {
51              when {
52                  anyOf {
53                      environment name: 'CHANGE_TARGET', value: '
    main'
54                      allOf {
55                          branch 'dev'
56                          triggeredBy "TimerTrigger"
57                      }
58                  }
59              }
60              steps {
61                  sh 'adb shell am instrument -w -r -e type phone
    -e value +5561998170441 -e pass PASSWORD -e serial
    LLKFAABAAACADWCF -e debug false -e class "br.com.loopkey.
    indigo.suite.OwnerUnitTestSuite" br.com.loopkey.indigo.test/
    androidx.test.runner.AndroidJUnitRunner'
62
63                  script {
64                      def logz = currentBuild.rawBuild.getLog(100)
    ;
```

```
65                      def result = logz.find { it.contains('
    FAILURES!!!') }
66                      if (result) {
67                          sh 'adb shell am instrument -w -r -e
    type phone -e value +5561998170441 -e pass PASSWORD -e serial
     LLKFAABAAACADWCF -e debug false -e class "br.com.loopkey.
    indigo.suite.ErrorUnitTestSuite" br.com.loopkey.indigo.test/
    androidx.test.runner.AndroidJUnitRunner'
68                          error ('Failing due to ' + result)
69                      }
70                  }
71              }
72          }
73          stage('Admin Test') {
74              when {
75                  anyOf {
76                      environment name: 'CHANGE_TARGET', value: '
    main'
77                      allOf {
78                          branch 'dev'
79                          triggeredBy "TimerTrigger"
80                      }
81                  }
82              }
83              steps {
84                  sh 'adb shell am instrument -w -r -e type email
    -e value calebe@loopce.com -e pass PASSWORD -e serial
    LLKFAABAAACADWCF -e debug false -e class "br.com.loopkey.
    indigo.suite.AdminUnitTestSuite" br.com.loopkey.indigo.test/
    androidx.test.runner.AndroidJUnitRunner'
85
86                  script {
87                      def logz = currentBuild.rawBuild.getLog(100)
    ;
88                      def result = logz.find { it.contains('
    FAILURES!!!') }
89                      if (result) {
90                          sh 'adb shell am instrument -w -r -e
    type phone -e value +5561998170441 -e pass PASSWORD -e serial
```

```
    LLKFAABAAACADWCF -e debug false -e class "br.com.loopkey.
    indigo.suite.ErrorUnitTestSuite" br.com.loopkey.indigo.test/
    androidx.test.runner.AndroidJUnitRunner'
91                      error ('Failing due to ' + result)
92                  }
93              }
94          }
95      }
96      stage('Guest Online Test') {
97          when {
98              anyOf {
99                  environment name: 'CHANGE_TARGET', value: '
    main'
100                 allOf {
101                     branch 'dev'
102                     triggeredBy "TimerTrigger"
103                 }
104             }
105         }
106         steps {
107             sh 'adb shell am instrument -w -r -e type email
    -e value calebe+1@loopce.com -e pass PASSWORD -e serial
    LLKFAABAAACADWCF -e debug false -e class "br.com.loopkey.
    indigo.suite.GuestOnlineUnitTestSuite" br.com.loopkey.indigo.
    test/androidx.test.runner.AndroidJUnitRunner'
108
109             script {
110                 def logz = currentBuild.rawBuild.getLog(100)
    ;
111                 def result = logz.find { it.contains('
    FAILURES!!!') }
112                 if (result) {
113                     sh 'adb shell am instrument -w -r -e
    type phone -e value +5561998170441 -e pass PASSWORD -e serial
    LLKFAABAAACADWCF -e debug false -e class "br.com.loopkey.
    indigo.suite.ErrorUnitTestSuite" br.com.loopkey.indigo.test/
    androidx.test.runner.AndroidJUnitRunner'
114                     error ('Failing due to ' + result)
115                 }
```

```
116                 }
117             }
118         }
119         stage('Guest Offline Test') {
120             when {
121                 anyOf {
122                     environment name: 'CHANGE_TARGET', value: '
    main'
123                     allOf {
124                         branch 'dev'
125                         triggeredBy "TimerTrigger"
126                     }
127                 }
128             }
129             steps {
130                 sh 'adb shell am instrument -w -r -e type phone
    -e value +5561992534172 -e pass PASSWORD -e serial
    LLKFAABAAACADWCF -e debug false -e class "br.com.loopkey.
    indigo.suite.GuestOfflineUnitTestSuite" br.com.loopkey.indigo
    .test/androidx.test.runner.AndroidJUnitRunner'
131 
132                 script {
133                     def logz = currentBuild.rawBuild.getLog(100)
    ;
134                     def result = logz.find { it.contains('
    FAILURES!!!') }
135                     if (result) {
136                         sh 'adb shell am instrument -w -r -e
    type phone -e value +5561998170441 -e pass PASSWORD -e serial
     LLKFAABAAACADWCF -e debug false -e class "br.com.loopkey.
    indigo.suite.ErrorUnitTestSuite" br.com.loopkey.indigo.test/
    androidx.test.runner.AndroidJUnitRunner'
137                         error ('Failing due to ' + result)
138                     }
139                 }
140             }
141         }
142         stage('Owner Final Test') {
143             when {
```

```
144                 anyOf {
145                     environment name: 'CHANGE_TARGET', value: '
    main'
146                     allOf {
147                         branch 'dev'
148                         triggeredBy "TimerTrigger"
149                     }
150                 }
151             }
152             steps {
153                 sh 'adb shell am instrument -w -r -e type phone
    -e value +5561998170441 -e pass PASSWORD -e serial
    LLKFAABAAACADWCF -e debug false -e class "br.com.loopkey.
    indigo.suite.OwnerFinalUnitTestSuite" br.com.loopkey.indigo.
    test/androidx.test.runner.AndroidJUnitRunner'
154
155                 script {
156                     def logz = currentBuild.rawBuild.getLog(100)
    ;
157                     def result = logz.find { it.contains('
    FAILURES!!!') }
158                     if (result) {
159                         sh 'adb shell am instrument -w -r -e
    type phone -e value +5561998170441 -e pass PASSWORD -e serial
     LLKFAABAAACADWCF -e debug false -e class "br.com.loopkey.
    indigo.suite.ErrorUnitTestSuite" br.com.loopkey.indigo.test/
    androidx.test.runner.AndroidJUnitRunner'
160                         error ('Failing due to ' + result)
161                     }
162                 }
163             }
164         }
165     }
166 }
```