Universidade Federal
do Rio de Janeiro
Escola Politécnica

PIPELINE DE IMPLANTAÇÃO CONTÍNUA NO CONTEXTO DE INTERNET DAS COISAS PARA RASPBERRY PI

Andréa Cristina de Souza Doreste

Projeto de Graduação apresentado ao Curso de Engenharia de Computação e Informação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheira.

Orientadores: Guilherme Horta Travassos,
Breno Bernard Nicolau de França

Rio de Janeiro

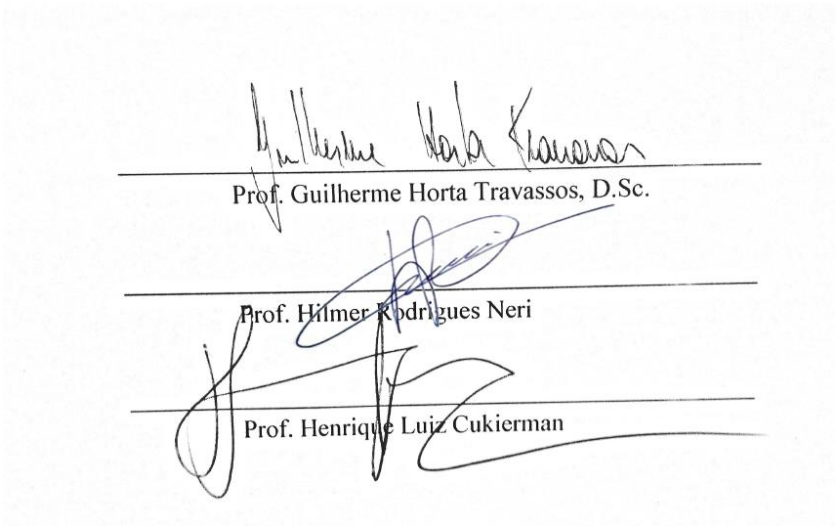
Março de 2018

PIPELINE DE IMPLANTAÇÃO CONTÍNUA NO CONTEXTO DE INTERNET DAS COISAS PARA RASPBERRY PI

Andréa Cristina de Souza Doreste

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO DE ENGENHARIA DE COMPUTAÇÃO E INFORMAÇÃO DA ESCOLA POLITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRA DE COMPUTAÇÃO E INFORMAÇÃO

Examinada por:



Rio de Janeiro – RJ, Brasil

Março de 2018

Doreste, Andréa Cristina de Souza

Pipeline de Implantação Contínua no Contexto de Internet das Coisas para Raspberry Pi/ Andréa Cristina de Souza Doreste - Rio de Janeiro: UFRJ/ Escola Politécnica, 2018

XI, 37 p.: il.; 29,7cm.

Orientadores: Guilherme Horta Travassos

Breno Bernard Nicolau de França

Projeto de Graduação - UFRJ/ POLI/ Engenharia de Computação e Informação, 2018

Bibliografia: p. 47 - 48

I. Travassos, Guilherme Horta et al. II. Universidade Federal do Rio de Janeiro, Escola Politécnica, Curso de Engenharia de Computação e Informação. III. Título

Dedico este trabalho a Márcia, Nilton, Manoel e Maria.

AGRADECIMENTOS

Agradeço, primeiramente, a Deus, por todas as oportunidades e bênçãos ao longo de toda uma vida.

A minha família, por todo o apoio, compreensão e - muita - paciência ao longo de todos esses anos de estudos.

Em especial, agradeço a minha mãe, Márcia, por sempre ter sido a minha base na vida e ao meu tio, Nilton, por ser minha maior fonte de inspiração. Isaac Newton disse uma vez que viu mais longe porque estava “em pé sob os ombros de gigantes”, se eu estou mais perto de realizar um sonho de infância (e ver um pouco mais longe), é graças ao apoio e amor incondicional que eu recebi de vocês ao longo de todos esses anos.

Ao meu orientador, Guilherme Horta Travassos, por ter acreditado em mim e ter me dado essa - e outras oportunidades. Agradeço também pela atenção, paciência e inspiração nesse e em outros trabalhos.

Ao meu co-orientador, Breno França, pela paciência e eficiência, ao longo da realização desse trabalho.

A toda equipe "Parasite” por todo o apoio, especialmente, nessa fase final do projeto.

Ao professor Henrique Luiz Cukierman, por ter enxergado o meu potencial quando eu mesma ainda não conseguia e por me ensinar que todo bom engenheiro precisa saber enxergar a sociedade ao seu redor.

A minha irmã, Fernanda Fernandes, por ser o meu braço direito na vida.

Aos meus amigos e amigas, por fazer cada passo desse caminho valer a pena. Em especial a Alexandre Alves, Ana Beatriz, Andressa Bittencourt, Anna Gabrielle, Brian Confessor, Bruno Dias, Carina Diógenes, Débora Pina, Guilherme Almeida, Gustavo Machado, Igor Amaral, Juliana Nunes, Juliano Marinho, Leonardo Souza, Lucas Santos, Marcelo Santos, Marcos Filho e Victor Maia. Vocês foram, de alguma forma, fundamentais ao longo de todos esses anos.

Por último e não menos importante, agradeço a sociedade brasileira, por arcar com os custos de todos esses anos de estudo. Espero poder contribuir, como profissional e como pessoa, para que, a cada dia, possamos ter uma sociedade melhor.

RESUMO

Com o passar dos anos, os cenários para os quais se desenvolve sistemas de software vêm se modificando, apresentando a desenvolvedores e engenheiros novos desafios, como entregas mais rápidas, frequente mudança de requisitos, menor tolerância a falhas e adequação aos modelos de negócio contemporâneos.

Para lidar com essas novas necessidades, a Engenharia de Software vem, ao longo do tempo, também se modificando e transformando práticas de desenvolvimento discretas em alternativas mais iterativas, flexíveis e contínuas, sem perder o objetivo de construir e disponibilizar produtos com qualidade e dentro dos prazos e orçamentos previstos.

Ao conjunto de práticas e ferramentas que visa assegurar as características de construção acima mencionadas dá-se o nome de Engenharia de Software Contínua, que apoia uma visão holística do processo de desenvolvimento com o objetivo de torná-lo mais rápido, iterativo, integrado e contínuo.

Práticas de desenvolvimento contínuo já são bastante utilizadas em desenvolvimento de aplicações *Web* e *Mobile*, por exemplo, mas o cenário tecnológico está sempre se modificando e uma tendência atualmente em ascendência é de prover soluções de software voltadas a Internet das Coisas (do inglês, *Internet of Things - IoT*). Como todo novo cenário, a *IoT* apresenta seu conjunto de possibilidades e desafios próprios, mas continua inserido num universo que requer agilidade com qualidade.

Levando isto em consideração, foi desenvolvido uma estrutura de apoio ao desenvolvimento contínuo - um pipeline de implantação - para ser utilizada em um nicho específico de sistemas de software *IoT*: aplicações desenvolvidas para o minicomputador *Raspberry Pi*. Através da construção deste pipeline, observou-se que é possível utilizar práticas de desenvolvimento contínuo num cenário *IoT*, embora alguns problemas intrínsecos a este nicho, tais como compatibilidade, limitações do ferramental utilizado e as integrações exigidas por projetos dessa natureza mereçam atenção dos engenheiros de software.

Palavras-Chave: Engenharia de Software Contínua, Desenvolvimento Contínuo, Integração Contínua, Implantação Contínua, Internet das Coisas, *Raspberry*.

ABSTRACT

The set of problems, situations, and scenarios solved by software systems has changed over the years, presenting to developers and engineers new problems every day. Some of these challenges can be described by shorter time-to-market, ever-changing customer requirements, little fault tolerance and fast-changing markets.

To deal with these tasks, Software Engineering has been evolving, turning discrete development practices into more iterative, flexible and continuous alternatives, while still maintaining the goal of building products with quality and making them available within the expected deadlines and budgets.

Continuous Software Engineering is the set of principles and tools with the goal of ensuring the characteristics mentioned above. With its holistic view from the software development process, it tries to make the process faster, more integrated, iterative and continuous.

Although Continuous Software Development practices have frequently been used for Web and Mobile Applications, the prominent paradigm nowadays is the Internet of Things (IoT). Like every new scenario, IoT presents its own set of possibilities and challenges but remains part of a universe that requires an agile process with quality.

Considering all those, a pipeline was created to support continuous development in a specific niche of IoT software systems: applications developed for the Raspberry Pi minicomputer. Through the construction of this pipeline, it was observed that it is possible to use continuous development practices in an IoT scenario, although some problems intrinsic to this niche, such as compatibility, tool's limitations and the integrations required by such projects deserve the attention of the software engineers.

Keywords: Continuous Software Engineering, Continuous Development, Continuous Integration, Continuous Deployment, Internet of Things, *Raspberry*.

Sumário

SIGLAS

CCS - *Cascading Style Sheets*

CI - *Continuous Integration*

GPIO - *General Purpose Input/Output*

HTML - *HyperText Markup Language*

IoT – Internet das Coisas

IP - *Internet Protocol*

NTDs - *Neglected Tropical Diseases*

OMS - Organização Mundial da Saúde

SCP - *Secure copy*

SDHC - *Secure Digital High Capacity*

SSH – *Secure Shell*

SVN - Apache Subversion

UI - *User Interface*

VSC - *Version Control System*

XML - *eXtensible Markup Language*

Xvfb - *X virtual framebuffer*

Capítulo 1

Introdução

1.1 – Motivação

Ao longo da história da computação, o foco do desenvolvimento de sistemas de software vem mudando de acordo com a evolução dos dispositivos computacionais. Em seus tempos primordiais, computadores eram basicamente calculadoras gigantes, concebidas para resolver equações matemáticas complexas e programados através de cartões perfurados [1]. O tempo passou e surgiram os *desktops, notebooks, palmtops, smartphones*, *smartwatches* e uma série de dispositivos que diminuiu de tamanho, mas não de complexidade.

Toda essa evolução que aconteceu na computação não foi limitada somente ao hardware (tamanho físico, capacidade de memória ou de processamento), mas foi acompanhada também pelo software, que é a parte responsável por dar utilidade e fornecer inúmeras possibilidades de funcionamento a um mesmo pedaço de hardware. E, por essas mudanças no cenário para o qual o software é desenvolvido, ocorreram mudanças também na forma como ele é desenvolvido. Modelos de processos de desenvolvimento (inseridos no ciclo de vida do software) que antes eram sequenciais e discretos se tornaram mais iterativos, interativos e incrementais [2]; se apresentando como um processo contínuo, de forma a se adequar aos modelos de negócios e aplicações que surgiam.

No cenário tecnológico atual, um novo paradigma está se consolidando, onde o software não habita somente o hardware, no sentido de conjunto de dispositivos especiais que compõem um computador, mas pode habitar em qualquer “coisa”. Em outras palavras, objetos comuns têm internalizado hardware que, juntamente com software próprios, se tornam minicomputadores, como telefones ou relógios. A esse novo paradigma deu-se o nome de Internet das Coisas (do inglês, *Internet of Things*) que, naturalmente, traz consigo novos desafios e possibilidades para o desenvolvimento de sistemas de software.

Tendo em vista a existência de práticas de desenvolvimento contínuo utilizadas e difundidas atualmente (e que serão explicadas ao longo dos próximos capítulos), surge a necessidade de entender como elas podem funcionar em um cenário de IoT, visto que sistemas de software nesse contexto possuem particularidades, como plataformas específicas, além de requererem a conexão e comunicação de objetos, sensores, transmissores, entre outros [3].

Particularmente, assim como sistemas de software que executam em celulares não são construídos no celular, aplicações IoT não são criadas em dispositivos IoT. É preciso, portanto, garantir a integração, tanto do software desenvolvido com o hardware no qual ele será utilizado, quanto das várias partes do sistema de software que devem se integrar, garantindo o correto funcionamento da aplicação. Portanto, as práticas e conceitos de Engenharia de Software Contínua podem ser úteis para apoiar a construção deste tipo de aplicação, considerando seus ambientes de desenvolvimento, testes e produção.

Desta forma, o presente trabalho tem como motivação principal entender e explorar os desafios apresentados quando se utiliza práticas de desenvolvimento contínuo de software na construção de sistemas de software para IoT. Para tal, utiliza uma estrutura de apoio às práticas de desenvolvimento contínuo - um pipeline – instanciada no projeto *Parasite Watch* – sistema de software para apoio ao diagnóstico de doenças parasitárias tropicais negligenciadas. Este projeto se mostra inserido no contexto de *Internet das Coisas* por utilizar, dentre outros recursos, um minicomputador *Raspberry Pi* (plataforma frequentemente utilizada em experimentos IoT), integrado a seus módulos e dispositivos próprios, como uma câmera específica (PiCamera), que realiza a digitalização e "*iotização*" de microscópio ótico, o qual demanda sistemas de software específicos a esta plataforma computacional. Surge, portanto, como diferencial em relação a sistemas convencionais, a necessidade de avaliar o comportamento dos dispositivos e módulos conectados ao *Raspberry Pi* e considerá-los ao compor o pipeline acima mencionado, a fim de garantir a correta execução do sistema como um todo.

1.2 – Objetivos

O objetivo principal deste trabalho é, portanto, instanciar um pipeline de implantação contínua que apoie e auxilie o desenvolvimento de aplicações IoT, em especial no contexto do sistema *Parasite Watch*. O pipeline é composto por quatro etapas,

como ilustrado na Figura 1: Controle de Versão, Construção da Aplicação (composta também por testes unitários), Testes de Sistema e Implantação.

Figura 1. Pipeline de Implantação

Como o mesmo está inserido em um cenário de Internet das Coisas, considera-se relevante, como proposta, criar uma estrutura que não apenas automatize o processo, mas que seja o mais fiel possível às necessidades e particularidades que esse tipo de sistema pode apresentar. Isso torna, portanto, a solução proposta inusitada e inovadora, já que IoT ainda é um cenário pouco explorado pelas práticas de engenharia de software contínua.

1.3 – Metodologia

A Figura 2 descreve a metodologia utilizada para desenvolvimento. Ela é composta de quatro etapas:

1. **Revisão da Literatura:** Realização de uma pesquisa na literatura técnica com o objetivo de identificar os principais conceitos, problemas e tecnologias da área;
2. **Concepção e Implementação da Solução:** A partir do que foi identificado na primeira etapa, elaborar e implementar uma solução que atendesse aos objetivos do projeto;
3. **Avaliação da Solução:** A partir do que foi construído, avaliar a eficácia da solução e identificar suas limitações e necessidades de melhorias.

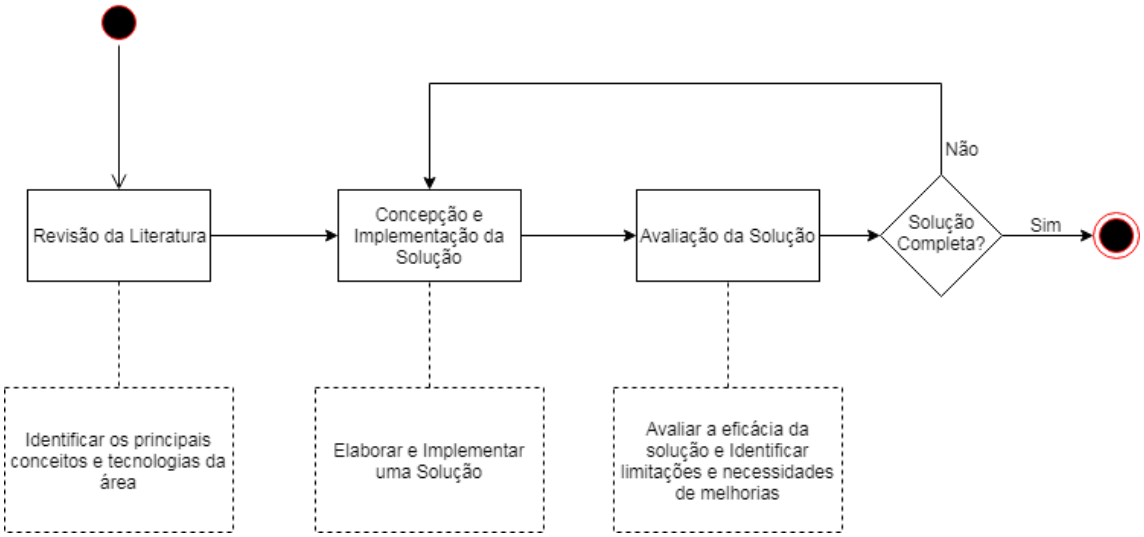


Figura 2. Metodologia adotada

1.4 – Organização do Trabalho

Além desta introdução, os próximos capítulos estão organizados da seguinte forma.

O capítulo 2 apresenta uma revisão da literatura, abordando conceitos fundamentais para desenvolvimento do trabalho, como Engenharia de Software Contínua, Integração Contínua e o desenvolvimento de um Pipeline de Implantação Contínua.

O capítulo 3 apresenta uma breve descrição das tecnologias utilizadas no projeto, como o minicomputador *Raspberry Pi*, as ferramentas de controle de versão *Git* e *GitLab*, o servidor de integração contínua *Jenkins* e seus inúmeros recursos implementados, algumas vezes, na forma de *plugins*.

O capítulo 4 apresenta uma visão geral do projeto *Parasite Watch*, do Pipeline de Implantação desenvolvido e como as características do *Parasite Watch* influenciaram nas decisões do projeto.

O capítulo 5 apresenta as conclusões do projeto: as limitações encontradas e as melhorias a serem implementadas em trabalhos futuros.

Capítulo 2

Revisão da Literatura: Conceitos Gerais

2.1 – Engenharia de Software Contínua

O processo construtivo dos sistemas de software vem sofrendo adaptações e melhorias com o passar dos anos, principalmente devido a evolução e necessidades do mercado. Em geral, conceitos e ferramentas ligadas à Engenharia de Software tradicional apoiam um ciclo de desenvolvimento baseado em atividades discretas, tendendo ao modelo sequencial [4].

Um exemplo clássico e exacerbado dessa forma de desenvolvimento é o tradicional Modelo Cascata (*Waterfall*), o qual é caracterizado pelo desacoplamento entre atividades fundamentais, como planejamento, análise, *design*, codificação, testes, e implantação [4]. Ou seja, nesse tipo de modelo, as etapas do processo de desenvolvimento são definidas com alguma rigidez, ordem específica de execução e sem que haja muita interação entre elas. Desenvolver qualquer sistema de software dessa forma pode se representar um processo lento e sequencial e, uma vez que uma fase é concluída e seus artefatos gerados (seja especificação de requisitos, arquitetura ou código do software), não se costuma revisitar fases anteriores até o fim do ciclo de desenvolvimento.

De acordo com PFLEEGER (2004, p.39), "*o maior problema com o modelo cascata é que ele não reflete, efetivamente, o modo como o código é desenvolvido. Exceto para problemas muito bem entendidos, o software é geralmente desenvolvido com muita iteração*" [2]. Além da iteração natural decorrente do ciclo de desenvolvimento, com o passar dos anos, o ambiente de negócios para o qual o software é produzido também começa a sofrer alterações, e passa a "*não tolerar mais grandes atrasos*" [2].

Desta forma, na década de 80, surgem outros modelos de processo, como o desenvolvimento incremental e iterativo. No desenvolvimento incremental, "*o sistema, como está especificado na documentação dos requisitos, é dividido em subsistemas por funcionalidades*" [2], de maneira que se inicia o processo desenvolvendo um subsistema e novas funcionalidades são adicionadas a cada versão [2]. Já no desenvolvimento

iterativo, "*entrega-se um sistema completo desde o começo e então muda a funcionalidade de cada subsistema a cada nova versão*" [2], aperfeiçoando-as e adicionando novas características.

Em ambos os casos, já é possível perceber um processo mais iterativo, porém etapas importantes como especificação de requisitos e análise de risco, acontecem apenas uma vez, no início do processo. Como consequência, é criado o Modelo Espiral. Este modelo pode ser definido, de forma básica, como um processo iterativo de desenvolvimento acrescido de atividades de gerenciamento de risco.

Como observado em PFLEEGER (2004, p.46),

> "*O modelo em espiral é, de certa maneira, como o modelo iterativo. Começando com os requisitos e um projeto inicial para o desenvolvimento (incluindo um orçamento, restrições e alternativas para o pessoal, o projeto e o ambiente de desenvolvimento), o processo insere uma etapa para avaliar riscos e protótipos alternativos antes de ser produzido um documento de 'concepção das operações', a fim de descrever, em alto nível, como o sistema deverá funcionar. A partir desse documento, um conjunto de requisitos é especificado e detalhado para assegurar que os requisitos estão tão completos e consistentes quanto possível. Consequentemente, o conceito a concepção das operações é o produto da primeira iteração, e os requisitos são o principal produto da segunda iteração. Na terceira iteração, o desenvolvimento do sistema produz o projeto, e na quarta, habilita os testes.*"

Apesar de útil em ambientes de software complexos, essas práticas começaram a se mostrar inadequadas aos novos projetos de software que surgiram. Desenvolvimento de aplicações para celulares, sistemas Web, desenvolvimento de produtos por Startups e, recentemente, até software desenvolvido para objetos e eletrodomésticos (Internet das Coisas) requerem entregas mais rápidas (menor "*time to delivery*"), requisitos sujeitos a alterações frequentes, maior urgência na correção de defeitos e produto sempre sujeito à adequação [5].

Para adaptar-se a essas necessidades, foram criados novos conceitos e formas de organizar o processo de desenvolvimento, a fim de torná-lo mais ágil e adequado ao

mercado e as novas tecnologias. Um exemplo disso são os Métodos Ágeis, que intencionam promover um processo de desenvolvimento mais dinâmico, fluido e iterativo, capaz de atender à necessidade por flexibilidade e rápida adaptação presente nos ambientes de desenvolvimento de *software* dos modelos de negócios atuais [4].

No entanto, exatamente como os Métodos Ágeis procuram agilizar o desenvolvimento de *software* para lidar com mudanças frequentes no ambiente de negócios, a natureza desses negócios também exige que as atividades de planejamento sejam feitas e atualizadas com mais frequência, para garantir o alinhamento entre as necessidades do contexto empresarial e o desenvolvimento do *software*, assim como também requer uma estreita integração entre planejamento, execução e operação [4]. Esta necessidade pela ação continuada entre planejar, construir e operar levou as discussões relacionadas a Engenharia de Software Contínua, que observa o ciclo de vida do software (da concepção até entrega e operação) de forma holística.

É dito por FITZGERALD e STOL [4] que, ao invés de uma sequência de atividades discretas, realizadas por equipes ou departamentos claramente distintos, o objetivo da engenharia de software contínua é estabelecer um fluxo contínuo, que deve levar em consideração todo o ciclo de vida do software. Ainda segundo FITZGERALD e STOL [4], esse ciclo é composto por três áreas principais: Negócios, Desenvolvimento e Operações.

Práticas como planejamento e orçamento contínuo fazem parte da primeira área: Negócios. A terceira área, Operações, é composta pela manutenção contínua do software e do usuário, gerando preocupações com a utilização e confiança contínua, bem como o monitoramento contínuo e em tempo real. A área intermediária, Desenvolvimento (foco deste trabalho), compreende as principais práticas de desenvolvimento de software: análise, *design*, codificação e verificação/teste, e é composta, principalmente, pelas práticas de Integração e Entrega Contínua, envolvendo também as práticas de evolução, conformidade e segurança contínua [4].

É importante ressaltar que a Engenharia de Software Contínua defende que as três áreas principais se integrem de forma que o processo seja contínuo do início ao fim. Como dito por FITZGERALD e STOL [4], uma vez que uma nova função do produto for identificada, deve ser inicialmente planejada, implementada, integrada, testada e entregue ao usuário. Dessa forma, aquilo que é observado numa das áreas se refletirá em todas as

outras e é possível garantir a fluidez do processo, facilitando a detecção e correção de defeitos, proporcionando o rápido *feedback* dos usuários, evitando o desperdício de tempo e esforço dos desenvolvedores e entregando um produto que esteja de acordo com o que é desejado e de interesse dos usuários.

2.2 – Implantação Contínua

Integração Contínua é um conjunto de princípios que deve ser aplicado ao fluxo de trabalho dos times de desenvolvimento [6], que visa integrar pedaços novos ou modificados de código a um software já existente, enquanto garante a integridade do software como um todo [7].

Conforme FITZGERALD e STOL [4], há uma variabilidade considerável na forma como a prática é definida e quais atividades podem ser consideradas parte do conjunto de Integração Contínua. No entanto, em essência, essa prática pode ser definida como um processo que é ativado automaticamente e compreende etapas interligadas, como compilação/análise do código, verificação de conformidade com os padrões de codificação, construção de pacotes da aplicação, execução de testes (unitários, de integração, de aceitação, entre outros), e análise de cobertura de testes.

No entanto, vale ressaltar que, apesar da automação do processo ser importante, a frequência de integração também é fundamental, pois é o que garante o *feedback* regular aos desenvolvedores [4]. Ainda citando FITZGERALD e STOL [4], com a integração contínua, podem ser identificados vários outros modos de atividades contínuas, como Entrega e Implantação Contínua.

A Entrega Contínua estende a Integração Contínua no sentido de proporcionar a habilidade de implantar o software em algum ambiente, mas não necessariamente disponibilizá-lo aos usuários finais (ambiente de produção).

Implantação Contínua refere-se a prática de liberar versões de software válidas para os usuários de forma automática ou semiautomática. Ou seja, uma vez que o processo é disparado e passa, com sucesso, por uma série de etapas interligadas como as citadas anteriormente, a última etapa do processo é a disponibilização direta ao usuário (modo automático) ou a disponibilização ao usuário com o simples apertar de um botão [8], ou alguma ação que traduz a autorização por parte do gerente de projetos (modo semiautomático). Como pode ser observado, esses conceitos estão relacionados, visto que

a entrega contínua é um pré-requisito para a Implementação Contínua, mas o oposto não é verdade [4].

Além dessas duas práticas, Verificação e Testes Contínuos integram o conjunto de atividades do processo de Integração Contínua. Como dito em FITZGERALD e STOL [4], Testes Contínuos procuram aproximar o máximo possível as etapas de teste com a codificação em si. Ou seja, de forma contrária ao que acontecia em modelos mais antigos, onde os testes só eram executados ao final do projeto ou de uma longa iteração, a cada etapa e mudança de código, também há a execução de testes de forma automatizada.

Esse tipo de abordagem gera uma série de benefícios, pois falhas se tornam eventos importantes, que costumam gerar uma série de alertas, garantindo que serão priorizadas pelo time de desenvolvimento e solucionadas o mais breve possível [4]. Além disso, pelo período entre a codificação e a descoberta da falha ser menor, assim como o pedaço do software a ser inspecionado, os defeitos são corrigidos rapidamente, enquanto o contexto do problema está fresco nas mentes dos desenvolvedores, evitando também a aglutinação de defeitos.

Outros benefícios intencionados com a utilização da Integração Contínua incluem: desperdício de recursos são evitados [6][5][7], assim como erros de configuração manual [8]; a estabilidade do software é garantida [7] e novas funções são desenvolvidas com mais rapidez e flexibilidade.

Porém, nenhuma solução é universal e, com a Integração Contínua, não poderia ser diferente. As maiores dificuldades encontradas fazem referência ao grande esforço inicial de configuração exigido por práticas de Integração Contínua [5] e a adaptação das etapas que fazem parte do processo de desenvolvimento.

Esse conjunto de etapas que acontecem de forma ordenada e iterativa pode ser descrito também como um *pipeline* de desenvolvimento e, é importante ressaltar, que cada *pipeline* deve ser adequado ao produto ao qual está vinculado, de forma a atender as necessidades daquele produto em particular [8].

2.3 – Pipeline de Implantação Contínua

A ideia de se produzir software de maneira mais frequente e contínua, em pequenos ciclos que se complementam, é possível apenas porque existe todo um conjunto de práticas e princípios de engenharia que garantem a qualidade e a compatibilidade entre

as partes de software que são produzidas. Essas práticas, como já foi mencionado, fazem parte do que é conhecido como Implantação Contínua e são aplicadas ao fluxo de desenvolvimento de forma a facilitar o processo e garantir software coerente ao final de cada entrega.

Para que isto possa funcionar, é necessário que, além de haver desenvolvedores codificando frequentemente, haja um pipeline de apoio ao desenvolvimento e que a execução de atividades que costumam ser complexas e trabalhosas, como a implantação da aplicação no ambiente de produção, seja feita de forma automatizada. Vale ressaltar que, um ambiente de produção é caracterizado pelo ambiente onde os usuários finais tem acesso ao software e podem utilizá-lo.

Como mencionado na seção anterior, todo pipeline deve ser adequado a aplicação a qual se destina. No entanto, é possível destacar algumas etapas essenciais a um *pipeline* de implantação contínua.

A primeira dessas etapas é materializada por um sistema de controle de versão. Sistemas como *Git* e *Subversion* (SVN) apresentam inúmeras características que beneficiam o processo de desenvolvimento, como a organização do código e da documentação associada e a integração de partes de código que são escritos por profissionais distintos em momentos diferentes do processo. Porém, o maior benefício que um sistema de controle de versão pode prover, é a possibilidade de se reverter qualquer ambiente de desenvolvimento a um estado anterior sem grandes prejuízos ou dificuldades. Ou seja, em caso de falhas da aplicação, reverter o estado da aplicação para um estado de funcionamento anterior é trivial.

O próximo passo é automatizar a fase de construção (*build*) do software. Essa etapa específica varia de acordo com a linguagem de programação utilizada. Construir uma aplicação em linguagens como C ou C++ inclui a compilação do código, já em linguagens como Python, essa fase seria substituída pela simples junção dos arquivos que compõem a aplicação. Em ambos os casos, além da compilação/execução do código, um conjunto de testes (geralmente unitários e de integração [8]) devem ser executados nessa etapa, garantindo assim, o funcionamento do código a cada mudança.

Em uma etapa posterior, outros tipos de testes automatizados podem ser adicionados, como de desempenho, de aceitação ou qualquer outro tipo de teste necessário para a aplicação [7]. A etapa final seria a disponibilização do software em

alguma plataforma, seja para direto uso do cliente (como na Implantação Contínua) ou em um ambiente de homologação, por exemplo (como na Entrega Contínua).

Caso alguma falha aconteça ao longo desse processo automatizado, o mesmo é interrompido, a entrega não é realizada e os respectivos responsáveis são avisados instantaneamente do ocorrido, seja através de e-mail ou mensagens instantâneas (como *Slack*, *Telegram* ou *SMS*) ou de forma gráfica (algumas equipes e interfaces gráficas utilizam cores para sinalizar o estado do projeto).

Além disso, alguns artefatos podem ser gerados para facilitar o acompanhamento do projeto e a identificação de defeitos, como relatórios de falhas ou de cobertura de código.

Para auxiliar na tarefa de construção, teste e entrega (*deploy*) do software de forma automatizada, um Servidor de Integração Contínua é utilizado. Como dito por MEYER [6], um servidor de integração contínua é uma das mais importantes ferramentas que um time de desenvolvimento pode utilizar. Seu único objetivo é monitorar o código no controle de versão e, se alguma mudança for detectada, uma lista de comandos é executada com o objetivo de desencadear a construção e testes do software.

As três etapas acima citadas podem ser descritas como as fases fundamentais de um *pipeline* de implantação contínua e podem ser adaptadas e modificadas conforme a necessidade. Diferentes ambientes de projeto, como *Mobile* ou *Web*, terão diferentes necessidades e cabe aos envolvidos no projeto adequar o *pipeline* e as ferramentas como convém.

Por último, é importante lembrar que o ambiente de testes deve ser o mais semelhante possível ao ambiente de produção, de forma a garantir uma similaridade entre os problemas encontrados no ambiente de testes e o do usuário. Isto se mostra extremamente relevante em sistemas IoT, onde o sistema a ser testado envolve sensores, atuadores e dispositivos característicos em plataformas específicas. Nesse tipo de cenário, a simples execução de testes de sistemas num Servidor de Integração Contínua ou qualquer ambiente diferente do de produção não garante a qualidade do software e, portanto, nesse tipo de caso, uma solução específica deve ser criada.

Capítulo 3

Tecnologias Utilizadas para Construção do Pipeline de Implantação Contínua

3.1 – Raspberry Pi

Raspberry Pi é uma linha de computadores de baixo custo e alto desempenho desenvolvido pela Raspberry Pi Foundation[1] para fins educacionais e recreativos. Contando hoje com sete modelos, esses dispositivos integram todo o hardware necessário em uma única placa que, ao ser conectado a outros periféricos, pode funcionar perfeitamente como um desktop ou, usado de forma isolada, pode desempenhar o papel de um servidor de baixa capacidade. No entanto, todos os seus atributos (como pequeno tamanho e baixo custo), associado a facilidade em utilizá-lo e a infinidade de recursos disponíveis, tornam esses minicomputadores perfeitos para o uso em um vasto espectro de projetos que vão de invenções caseiras a sistemas embarcados e Internet das Coisas.

Essa linha de produtos tem como objetivo principal seu uso para o aprendizado infantil sobre computadores e programação. Desta forma, a plataforma vem equipada com uma série de ferramentas visando a simplificar e facilitar sua utilização.

O sistema operacional oficial do *Raspberry Pi* é o Raspbian[2], um sistema Debian otimizado e adaptado para este perfil de hardware, que conta com um grande número de pacotes e utilitários.

A linguagem padrão para desenvolvimento de aplicações na plataforma é o *Python* que, além de já vir com um ambiente de desenvolvimento configurado na instalação padrão do sistema operacional, também pode ser utilizado para configurar e controlar dispositivos adicionais conectados ao *Raspberry Pi*, tais como a *PiCamera*, câmera padrão do sistema. Além da *PiCamera*, existe uma variedade de sensores e

[1] https://www.raspberrypi.org/
[2] www.raspbian.org

dispositivos que podem ser conectados ao sistema através dos pinos de *GPIO* (*general purpose input/output*) ou por hardwares auxiliares como o Arduíno.

A versão atual e mais recente da família de produtos é o *Raspberry Pi* 3 Modelo B, com processador *QuadCore* de 1.2GHz de frequência, 1024 MB de memória RAM, quatro portas USB, entrada para cabo Ethernet, além de suportar as tecnologias WiFi e Bluetooth. É importante lembrar que a placa de hardware da linha *Raspberry Pi* não inclui nenhum tipo de memória não-volátil, mas possui entrada para cartão de memória do tipo Micro SDHC, de forma que todos os arquivos e dados, assim como o sistema operacional, ficam armazenados num micro cartão.

Devido a todas essas características, os computadores deste tipo ganharam seu espaço não só como ambiente de aprendizagem, mas também como instrumento usado por uma grande comunidade que aproveita os benefícios advindos dele para criar projetos inusitados e inovadores.

3.2 – GitLab

Como já foi mencionado anteriormente, uma parte fundamental de um *pipeline* de implantação contínua é um sistema de controle de versão. Um dos mais conhecidos e estabelecidos atualmente é o *Git*. De acordo com o site oficial[3], o *Git* é um sistema de controle de versões distribuído, livre e de código aberto, projetado para lidar com projetos de diferentes magnitudes, com velocidade e eficiência.

O que faz o *Git* diferente de outros sistemas de controle de versões, como o SVN, é o modelo de ramificações (*branching model*), que são completamente independentes entre si, tornando as tarefas de criação, fusão e exclusão de linhas de desenvolvimento muito mais velozes.

O *Git* também é um sistema de controle de versões descentralizado, ou seja, todo diretório *Git* existente em um computador é um repositório completo com capacidade de rastreamento de versão completa, independentemente do acesso à rede ou de um servidor central. Dessa forma, cada desenvolvedor tem uma cópia completa e separada do repositório que, de acordo com a necessidade do projeto ou decisão do desenvolvedor, pode ser sincronizado com um repositório remoto.

[3] "Git." https://git-scm.com/. Accessed 17 Feb. 2018.

Repositórios remotos[4] são versões de um projeto armazenadas em repositórios na Internet ou em outro lugar da rede. Isso permite o compartilhamento de código e possibilita que muitos desenvolvedores possam trabalhar de forma organizada e eficiente em um mesmo projeto de *software*. O gerenciamento desses repositórios remotos, assim como o de usuários, de aspectos de segurança, estrutura e outros aspectos mais "burocráticos" de implantação, podem ficar a cargo de serviços e ferramentas de terceiros, como o *GitLab* ou *GitHub*.

O *GitLab*[5] é um gerenciador de repositório *Git* online que oferece ao usuário controle total sobre repositórios e projetos e permite definir se eles serão públicos ou privados sem custo adicional. O *GitLab* é oferecido ao usuário de três formas diferentes.

O *GitLab.com* é um serviço gratuito no qual não é necessário executar qualquer tipo de instalação local, apenas realizar o cadastro no *site* e hospedar os projetos através de repositórios públicos ou privados sem custo adicional. O *GitLab Community Edition* (CE) também é um serviço gratuito, de código aberto, que é instalado em um servidor privado e garante acesso aos mesmos benefícios dos usuários do *GitLab.com*, além de ter o suporte da Comunidade do *GitLab*. Por ser instalado em servidor próprio, a infraestrutura necessária, a instalação e configuração do serviço e os dados do projeto ficam inteiramente sob a responsabilidade do usuário.

Por último, existe o *GitLab Enterprise Edition* (EE) que, assim como *GitLab CE*, é disponibilizado para ser instalado em servidor privado, mas não é um serviço gratuito e possui suporte adicional e um conjunto maior de funcionalidades disponíveis.

3.3 – Jenkins

O *Jenkins* é um dos Servidores de Integração Contínua mais populares, desenvolvido em Java e de código aberto [9]. Ele foi desenvolvido inicialmente por Kohsuke Kawaguchi que trabalhava na *Sun* e batizado, como "Hudson".

Em 2009, quando a *Oracle* adquiriu a *Sun*, herdou também o código base do projeto Hudson, mas em 2011, conflitos entre a comunidade de código aberto (incluindo seus desenvolvedores) e a *Oracle* levaram à uma ruptura que dividiu o projeto em dois: o

[4] "Working with Remotes - Git SCM." https://git-scm.com/book/en/Git-Basics-Working-with-Remotes. Accessed 17 Feb. 2018.
[5] "GitLab." https://gitlab.com/. Accessed 17 Feb. 2018.

projeto Hudson, então, permaneceu sob o controle da empresa, enquanto seus desenvolvedores originais continuaram o projeto com o nome de *Jenkins* [9].

A popularidade do *Jenkins* pode ser justificada pela sua interface simples, intuitiva, com apelo visual que proporciona um fácil uso e uma curva de aprendizado baixa [9]. Além disso, o *Jenkins* dá suporte a uma variedade de linguagens e tecnologias, incluindo .NET, Ruby, Groovy, Grails, PHP, Java [9], Python, entre outras.

Outra característica que torna o *Jenkins* relevante é a sua capacidade de integração com várias ferramentas distintas e úteis no contexto de Integração Contínua através de *plugins* de fácil instalação. Esses *plugins* cobrem uma ampla variedade de *features*, desde sistemas de controle de versões, ferramentas de compilação, métricas de qualidade de código, notificadores de construção de código, integração com sistemas externos, até personalização de UI, jogos e muito mais [9]. Alguns desses *plugins* serão tratados nas próximas seções, assim como outros recursos importante do *Jenkins*.

3.3.1 – *GitLab Plugin*

De acordo com o que é encontrado na *wiki* de *plugins* do *Jenkins*[6], o *GitLab plugin* é um gatilho de compilação que permite que o *GitLab* dispare as compilações do *Jenkins* quando o código é alterado no repositório ou uma solicitação de mesclagem é criada. Ou seja, ele é a interface entre o servidor de controle de versão do *GitLab* e o *Jenkins*; e, através dele, inicia-se o *pipeline* de implantação contínua. Mais informações a respeito deste *plugin* podem ser encontradas no seu repositório[7].

3.3.2 – *Shell-Script Interface*

O *Jenkins* disponibiliza, em cada projeto, um interpretador de comandos que possibilita a utilização da linguagem *Shell script* e pode ser utilizada, por exemplo, para a construção do código fonte do projeto. Esse recurso, no entanto, é apenas uma interface, uma ponte entre o *Jenkins* e o sistema operacional do servidor e, portanto, esse recurso só pode ser utilizado quando o *Jenkins* é instalado em sistemas operacionais Unix ou Linux [9].

---

[6] "GitLab Plugin - Jenkins - Jenkins Wiki." 22 Apr. 2017, https://wiki.jenkins.io/display/JENKINS/GitLab+Plugin. Accessed 17 Feb. 2018.
[7] "GitHub - jenkinsci/gitlab-plugin: A Jenkins plugin for interfacing with ...." https://github.com/jenkinsci/gitlab-plugin. Accessed 17 Feb. 2018.

3.3.3 – *JUnit Plugin*

O *JUnit plugin* [8] oferece uma funcionalidade que tem como entrada relatórios de testes no formato *XML* gerados durante a execução dos testes e fornece uma visualização gráfica dos resultados, assim como uma interface de *web* para visualizar relatórios de teste, rastros de falhas, entre outras funcionalidades.

3.3.4 – *Email Notification*

Email é a forma de notificação fundamental do *Jenkins* [9] pois, quando uma compilação falha, uma mensagem de correio eletrônico é enviada para a equipe de desenvolvimento do projeto com a finalidade de alertar sobre o ocorrido.

Esse recurso do *Jenkins* tem, portanto, duas funções principais: integrar um servidor de *e-mail* ao *Jenkins* (através do qual as mensagens serão enviadas) e gerenciar o envio da notificação de cada falha para os endereços de *email* configurados.

3.3.5 – Xvfb

Xvfb ou *X virtual framebuffer*[9] é um servidor de display que implementa o protocolo de serviço X11. Em suma, *Xvfb* executa todas as operações gráficas necessárias em uma memória virtual sem mostrar nenhuma saída na tela.

O *plugin* Xvfb do *Jenkins*[10] permite a execução de operações gráficas que, de outra forma, não poderiam ser executadas. Ele é configurado a cada projeto, de forma que é iniciado no início do projeto e é finalizado ao fim da construção. É útil em contextos de testes que requerem interação (disparo de eventos) com uma interface gráfica de usuário para serem executados.

---

[8] "JUnit Plugin - Jenkins - Jenkins Wiki." https://wiki.jenkins.io/display/JENKINS/JUnit+Plugin. Accessed 17 Feb. 2018.
[9] "XVFB - X.Org." https://www.x.org/archive/X11R7.6/doc/man/man1/Xvfb.1.xhtml. Accessed 17 Feb. 2018.
[10] "Xvfb Plugin - Jenkins - Jenkins Wiki." 26 Apr. 2016, https://wiki.jenkins.io/display/JENKINS/Xvfb+Plugin. Accessed 17 Feb. 2018.

3.4 – SSH e SCP

O protocolo SSH[11] (também conhecido como *Secure Shell*) é um método para autenticação remota e segura de um computador para outro. Ele fornece várias opções alternativas para autenticação e protege a segurança e integridade das comunicações com criptografia forte, garantindo a segurança da comunicação entre dois *hosts* não confiáveis em uma rede insegura. Seu funcionamento é baseado em um modelo cliente-servidor, de forma que o cliente inicia a conexão fazendo uso de criptografia de chave pública para verificar a identidade do servidor SSH. Após a fase de configuração e autenticação, é possível realizar operações diretamente no computador que está se comportando como servidor, através da rede.

Já o SCP (do inglês, *Secure copy*) é um protocolo de rede usado para transferir arquivos entre dois *hosts* numa rede. Esse protocolo é baseado no protocolo SSH, utilizando os mesmos mecanismos de autenticação e garantindo a autenticidade e a confidencialidade do dado transferido.

Em sistemas Unix, eles são implementados através da ferramenta *OpenSSH*[12]. Além da autenticação por meio de chave pública que, uma vez configurada, acontece de forma automática, é possível realizar a conexão através do par *login* e senha realizada de forma manual.

Essa forma de autenticação, no entanto, não se mostra muito útil para processos automatizados, já que as informações de *login* e senha devem ficar armazenadas em algum local, podendo comprometer a segurança. Dessa forma, para processos automatizados, uma melhor solução é adotar o primeiro método citado, a criptografia de chave pública, com o cliente e o servidor sendo configurado previamente e a conexão entre máquinas (e, consequentemente, transferência de arquivos) acontecendo sem a necessidade de se inserir outra informação durante a execução da ação.

[11] "SSH Protocol – Secure Remote Login and File Transfer | SSH.COM." 29 Aug. 2017, https://www.ssh.com/ssh/protocol/. Accessed 26 Feb. 2018.
[12] "OpenSSH." https://www.openssh.com/. Accessed 26 Feb. 2018.

3.5 – Selenium

O *Selenium*[13] é uma ferramenta de automação de *browsers*, que permite a execução e repetição de tarefas que, normalmente, são executadas de forma manual. Para tal, ele faz uso de *scripts*, onde as ações são previamente descritas, armazenadas e posteriormente executadas quantas vezes for necessário.

Os *scripts* utilizados pelo *Selenium* podem ser gerados de duas formas: através do *Selenium* IDE[14] ou do Selenium *WebDriver*[15]. No primeiro caso, um *plugin* é adicionado ao *browser* Firefox[16] e os passos que caracterizam a interação do usuário com o *browser* são gravados e transformados num script.

No segundo caso, um conjunto de funcionalidades é disponibilizado através de pacotes e *bindings* de uma linguagem de programação específica, como Java ou Python. Nesse caso, as ações são descritas no *script* na linguagem utilizada, fazendo referência, por exemplo, aos elementos HTML da página a ser testada.

Uma das principais vantagens de se utilizar o *Selenium* é que ele está disponível em diversas plataformas, podendo ser usado em diferentes browsers e sistemas operacionais. Além disso, as ações executadas através dele podem acontecer de forma física, onde os passos descritos no *script* são de fato executados num *display* físico, ou de forma virtual (também chamada de *headless*), com o auxílio de ambientes do tipo *Xvfb*, onde as ações acontecem de forma simulada, em *display* virtual criado em memória.

Todo o conjunto de funcionalidades apresentado anteriormente torna a ferramenta muito útil na criação de testes automatizados para aplicações Web, permitindo, portanto, a simulação da interação cliente-aplicação e automatizando a execução de testes funcionais de sistemas. Apesar de não estar apenas limitado a automatização de testes, pode-se dizer que essa é a principal funcionalidade do *Selenium* nos projetos desenvolvidos atualmente.

---

[13] "Selenium - Web Browser Automation." https://www.seleniumhq.org/. Accessed 7 Mar. 2018.
[14] "Selenium IDE Plugins." https://www.seleniumhq.org/projects/ide/. Accessed 7 Mar. 2018.
[15] "Selenium WebDriver." https://www.seleniumhq.org/projects/webdriver/. Accessed 7 Mar. 2018.
[16] "The new, fast browser for Mac, PC and Linux | Firefox - Mozilla." https://www.mozilla.org/en-US/firefox/. Accessed 7 Mar. 2018.

Capítulo 4

Desenvolvimento do Pipeline de Implantação Contínua em *Raspberry Pi*

4.1 – Motivação

4.1.1 – Contexto: O projeto *Parasite Watch*

Segundo a Organização Mundial de Saúde (OMS), as doenças tropicais negligenciadas (NTDs) representam um grupo de agravos transmissíveis prevalentes em 149 países em desenvolvimento e afetam mais de um bilhão de pessoas [10]. Entre essas doenças destacam-se as infecções parasitárias, cujo diagnóstico, apesar de toda a evolução tecnológica, ainda é realizado como descrito originalmente há aproximadamente 100 anos [11].

Para o diagnóstico desse tipo de doença é necessária a visualização, através de microscópio óptico, de formas parasitárias em amostras biológicas obtidas de pacientes, que é um procedimento laborioso, lento e que requer técnicos treinados e experientes, caracterizando um tipo de mão-de-obra especializada cada vez mais escassa. Além disso, o diagnóstico de infecções parasitárias requer infraestrutura de apoio indisponível (eventualmente de alto custo) na maioria das regiões em desenvolvimento [10].

Em busca de uma solução para esse problema surgiu o projeto *Parasite Watch*. Um sistema de software que tem como objetivo principal auxiliar o diagnóstico de doenças tropicais negligenciadas através do reconhecimento de imagens parasitárias submetidas pelos seus usuários, fazendo uso, para tanto, de um algoritmo de inteligência artificial, de forma a diminuir o tempo de diagnóstico despendido e demanda de mão-de-obra especializada.

O sistema de software *Parasite Watch* é composto por três subsistemas principais: Coleta, Identificação, e Confirmação e Aprendizado [12]. O Subsistema de Coleta é responsável por realizar a captura de informações (imagens digitalizadas via microscópio

com software embarcado) e é composto, atualmente, por um computador *Raspberry Pi* com uma *PiCamera*, integrado a um microscópio óptico não digital [12]. Uma vez realizada a captura das imagens, é preciso analisá-las. Essa é a tarefa do Subsistema de Identificação, que contém o componente para reconhecimento de parasitas em imagens geradas pelo Subsistema de Coleta. Ele inclui, portanto, um algoritmo de localização de imagens que faz uso de redes neurais sem peso *WiSARD* e deve estar disponível tanto através da Internet, quanto localmente, para que o sistema possa funcionar em regiões remotas. O último subsistema é o de Confirmação e Aprendizado, que engloba um repositório de imagens parasitárias reconhecidas pelo Subsistema de Identificação. Esse componente também permite que usuários especialistas em reconhecimento de parasitas façam verificações a respeito da correção dos diagnósticos feitos pelo subsistema de Identificação, permitindo uma análise a respeito do funcionamento do algoritmo de identificação e provendo mais uma fonte de aprendizado.

Além desses três componentes, o *Parasite Watch* inclui também um Repositório de Diagnósticos de NTD, que é uma base de dados, externa ao sistema, a qual recebe os resultados dos diagnósticos realizados nas diferentes regiões. As informações básicas desse repositório devem incluir dados de geolocalização, data, hora, resultado do exame e outras informações relevantes para apoiar, dentre outras ações, a elaboração de políticas de saúde pública.

O desenvolvimento do *Parasite Watch* iniciou-se com um conjunto básico de ideias e requisitos iniciais que deveriam ser validados através da criação de um protótipo, uma solução inicial desenvolvida de forma experimental, cíclica e incremental, com foco no desenvolvimento de funcionalidades específicas de um subsistema e que permitisse a avaliação e, principalmente, a aprendizagem.

Após a validação da ideia, o projeto entrou em uma nova fase, onde o objetivo passou a ser o aperfeiçoamento da solução inicial, a transformação do protótipo em um produto. O desenvolvimento em ciclos de curta duração continuou, mas, se antes o foco estava nas funcionalidades, nessa nova fase do projeto era necessário aperfeiçoar continuamente a arquitetura e integrar as diferentes partes do sistema, assim como as tecnologias envolvidas.

4.1.2 – Tecnologias Envolvidas no *Parasite Watch*

O Projeto *Parasite Watch* está sendo desenvolvido em *Python*[17], linguagem de programação interpretada, orientada a objetos e de alto nível. Em sua versão atual, adota uma arquitetura *Model-View-Control*, que está sendo implementada através do *framework* Flask[18], utilizando tecnologias para desenvolvimento *Web*, como a linguagem de formatação HTML (*HyperText Markup Language*) e a linguagem de estilo[19] CSS (*Cascading Style Sheets)* que geram uma interface com o usuário através de um navegador Web.

No entanto, devido aos ambientes externos nos quais o *Parasite Watch* deve ser utilizado (regiões remotas, sem acesso à Internet), todas as facilidades do sistema devem estar contidas no minicomputador *Raspberry Pi* de forma a serem usadas localmente.

4.2 – Os ambientes de desenvolvimento

Um dos desafios apresentados pelo *Parasite Watch* é o fato dos ambientes de desenvolvimento e produção serem, por natureza, diferentes. Apesar de haver a possibilidade de que a atividade de codificação seja executada diretamente em um computador *Raspberry Pi* conectado a *mouse*, teclado e monitor, é mais esperado que a codificação da aplicação seja realizada em computadores utilizados com frequência pela equipe de desenvolvimento. Além disso, o projeto *Parasite Watch* dispõe apenas de dois *Raspberry Pi* que podem ser alocados em tarefas mais específicas, portanto, usá-los para codificação seria um desperdício de recurso.

Dessa forma, como é ilustrado na Figura 3, o ambiente de desenvolvimento é caracterizado por computadores comuns (*notebook* ou *desktop*) que, em alguns casos, utilizam o sistema operacional Windows e, em outros, Ubuntu.

Para garantir o correto funcionamento da aplicação mesmo com as diferenças de características entre os dois ambientes acima citados (desenvolvimento e homologação),

[17] "What is Python? Executive Summary | Python.org." https://www.python.org/doc/essays/blurb/. Accessed 21 Feb. 2018.
[18] "Welcome | Flask (A Python Microframework)." http://flask.pocoo.org/. Accessed 21 Feb. 2018.
[19] *Style Sheet Language*

os testes são executados em um *Raspberry Pi*, que hospeda o servidor de integração contínua, caracterizando o ambiente de homologação.

Por fim, o segundo *Raspberry Pi* é destinado a utilização da aplicação em campo e captura de imagens do Subsistema de Coleta. Ele caracteriza, portanto, o ambiente de produção que é onde a entrega, ao fim do processo, deve ser realizada.

Mais informações a respeito dos ambientes de homologação e produção (assim como a respeito da entrega do software) serão fornecidas e explicadas nos próximos tópicos.

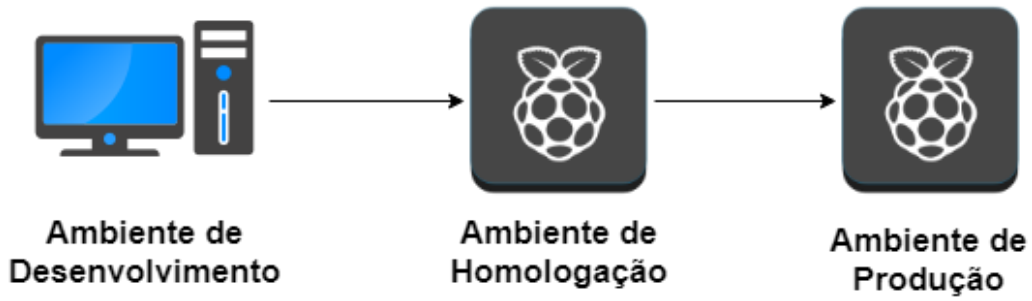


Figura 3. Ilustração dos ambientes de construção do *Parasite Watch*.

4.3 – O Pipeline de Implantação Contínua

Como explicado na seção 2.3, um pipeline de Implantação Contínua possui etapas essenciais (apresentadas na Figura 4) que se iniciam, geralmente, com a submissão (*push)* do código ao Sistema de Controle de Versões.

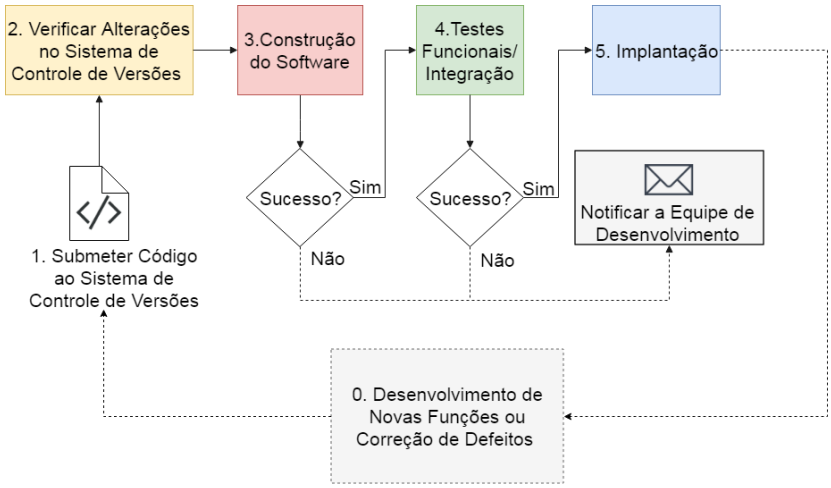


Figura 4. Pipeline de Implantação Contínua do Parasite Watch.

Em cada uma das etapas, existe um vasto número de ferramentas que podem ser utilizadas para a composição do *pipeline* e que variam de acordo com as decisões dos desenvolvedores e as configurações do projeto ao qual ele está vinculado. Com o Pipeline de Implantação Contínua do *Parasite Watch* não poderia ser diferente:

- O Sistema de Controle de Versões escolhido foi o *GitLab*, já utilizado como repositório pela equipe de desenvolvimento;
- A etapa de Construção do Código é caracterizada pela instalação dos pacotes necessários a execução da aplicação através do gerenciador de pacotes do Python Pip[20] e a realização de um conjunto de testes unitários que são executados por um interpretador acionado através do *Shell-Script Interface*, já que Python não é uma linguagem compilada;
- Os Testes de Sistema são realizados através da ferramenta *Selenium*, visto que a interface com o usuário ocorre através de um navegador e fazendo uso da tecnologia Xvfb. Isso permite que as ações do *Selenium* sejam executadas a partir de um *display* virtual (*headless*);
- Os Relatórios de Testes são gerados através do *JUnit*, que consome os arquivos XML gerados durante a execução dos testes e apresenta os resultados de forma gráfica;
- A notificação, em caso de falha, ocorre através do envio de e-mails para a equipe;
- E a Implantação consiste na entrega, através dos protocolos SSH e SCP aos dispositivos *Raspberry Pi* previamente configurados.

Dessa forma, levando em consideração as ferramentas utilizadas, o Pipeline de Implantação Contínua desenvolvido neste trabalho é ilustrado através da Figura 5.

[20] "pip 9.0.1 : Python Package Index." https://pypi.python.org/pypi/pip. Accessed 5 Mar. 2018.

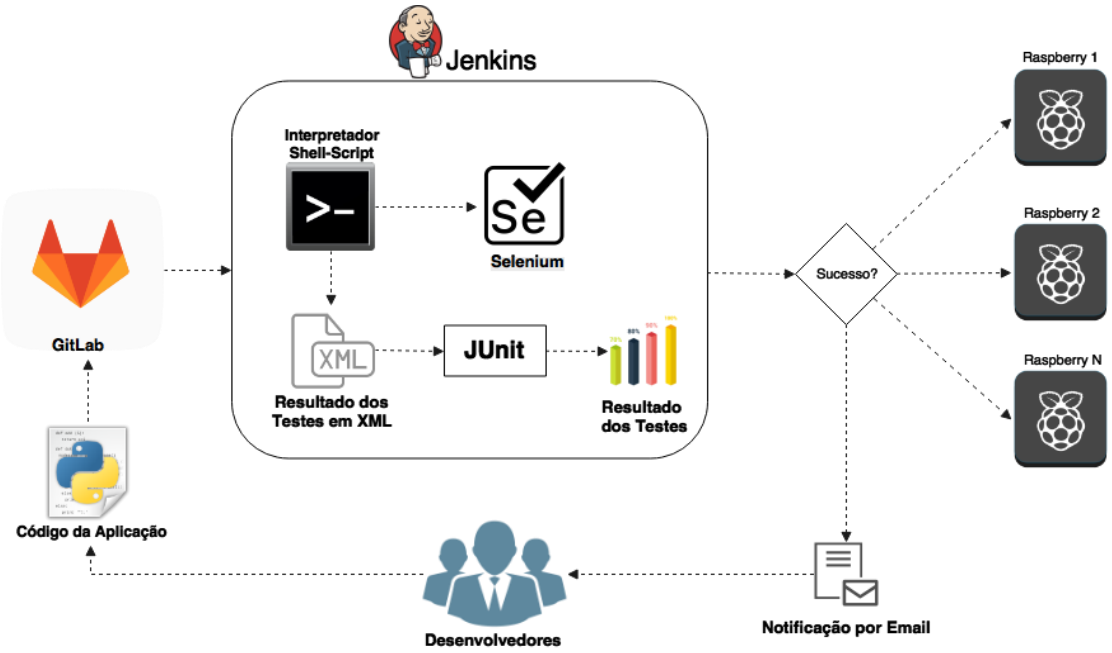


Figura 5. Ilustração do Pipeline de Implantação Contínua para o projeto Parasite Watch

O código é submetido ao *GitLab* pela equipe de desenvolvimento e dispara o processo do pipeline. O *GitLab* envia uma notificação ao *Jenkins* que cria um *workspace* e faz uma cópia dos arquivos. Após essa etapa, os testes unitários são executados através de um Interpretador Python invocado pela *Shell-Script Interface* que gera os resultados dos testes em arquivos XML. Os arquivos XML são interpretados pelo *JUnit*, gerando descrições detalhadas e visuais a respeito dos testes. Uma vez que haja sucesso, o *Selenium* é invocado para realizar os testes funcionais, em nível de sistema, e dar prosseguimento ao pipeline. Se os testes realizados pelo *Selenium* também forem bem-sucedidos, inicia-se a etapa de *deploy* (implantação).

Se qualquer uma dessas etapas falhar, o processo é interrompido e uma notificação é enviada para a equipe de desenvolvimento com o problema descrito de forma detalhada. Assim, a equipe poderá identificar o defeito e corrigi-lo, submetendo, novamente, o código ao Sistema de Controle de Versões.

As etapas para o desenvolvimento da solução, a criação do pipeline e a configuração das ferramentas serão descritas, passo a passo, ao longo das próximas seção.

4.4 – Implementação da Solução

Por apresentar proposta aparentemente inédita na literatura, foram necessárias, ao longo do projeto, duas tentativas de configuração para que a solução pudesse ser, de fato, implementada. De início, começou-se a desenvolver uma estrutura até que problemas foram encontrados e decidiu-se mudar de abordagem. As duas experiências estão descritas abaixo, assim como os motivos que acarretaram a mudança de abordagem construtiva.

4.4.1 – Alternativa de Configuração 1

A primeira tentativa de implantação do pipeline utilizava o *Jenkins* instalado em um servidor (externo) com o sistema operacional Debian e arquitetura AMD64.

Nessa alternativa, um projeto (ou *job*) foi criado e integrado ao sistema de controle de versão, mas, ao executar o script em Python para construir a aplicação, surgiram alguns problemas de compatibilidade de plataforma.

A aplicação *Parasite Watch* é executada em um *Raspberry Pi* 3 Model B[21]. Esse modelo utiliza um chip BCM2837[22], com quatro núcleos de processamento de arquitetura ARM Cortex A53 (ARMv8).

Alguns pacotes utilizados na aplicação são próprios do *Raspberry*, compilados para sua arquitetura e funcionam como a interface entre o computador e os módulos que podem ser acoplados a ele, como o pacote de utilização da *PiCamera*. Ao tentar instalá-los em uma arquitetura diferente, foi descoberto que não existiam versões compatíveis para esta nova arquitetura.

Outro problema, além da incompatibilidade, foi percebido com a experiência: para garantir a qualidade do software que está sendo entregue é preciso realizar testes, e como esses módulos seriam testados se um servidor *Jenkins* comum não teria acesso aos módulos, mas, principalmente, não teria acesso aos dispositivos? No caso de testes unitários, ainda seria possível criar *'mockups'* para substituir esses dispositivos, mas os

[21] "Raspberry Pi Hardware - Raspberry Pi Documentation." https://www.raspberrypi.org/documentation/hardware/raspberrypi/README.md. Accessed 21 Feb. 2018.
[22] "BCM2837 - Raspberry Pi Documentation." https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2837/README.md. Accessed 21 Feb. 2018.

testes de sistemas precisavam ser realizados num *Raspberry pi*. Não é possível, portanto, fazer testes de sistemas e testar esses dispositivos que são característicos de cenários IoT. Deu-se início então a outra alternativa de solução que não apresentasse estas limitações.

4.4.2 – Alternativa de Configuração 2

Na segunda alternativa de desenvolvimento do pipeline resolveu-se instalar o *Jenkins* no minicomputador *Raspberry Pi*. As etapas do processo de instanciação do pipeline (Figura 6) são divididas em cinco categorias: instalação (laranja), configuração (verde), criação (lilás), integração (azul) e verificação (vermelho).

Foi necessário, inicialmente, configurar o ambiente de desenvolvimento em três etapas: instalação do sistema operacional Raspbian num cartão do tipo Micro SDHC classe 10 (que foi escolhido por ser o mais rápido entre os disponíveis), configuração de um endereço IP estático para permitir o acesso ao servidor através da rede local e Internet; e instalação dos pré-requisitos necessários para a correta instalação do Servidor de CI *Jenkins*.

Após essas três etapas, o *Jenkins* foi instalado através do gerenciador de pacotes apt-get, seguindo as instruções que podem ser encontradas em sua *wiki*[23], e configurado utilizando a porta padrão 8080. Nessa parte de configuração também foram definidos um usuário administrador e uma senha de acesso.

Após a instalação e configuração, um *job* foi criado, onde os passos do pipeline foram parcialmente implementados.

Com o Servidor de CI sendo executado corretamente e o código da aplicação já no *GitLab*, o passo seguinte consistiu na primeira parte da integração entre essas duas aplicações. Foi instalado o *GitLab plugin* no *Jenkins*, a conexão com o *GitLab* foi configurada e o repositório do projeto *Parasite Watch* vinculado ao projeto do *Jenkins* foi utilizado.

Como a aplicação *Parasite Watch* possui uma interface gráfica (que necessita ser testada), foi realizada a instalação e configuração do *Xvfb*, tanto em nível de sistema operacional (através de um gerenciador de pacotes), como sua utilização pelo *Jenkins* através do *Xvfb plugin*.

[23] "Installing Jenkins on Ubuntu - Jenkins - Jenkins Wiki." https://wiki.jenkins.io/display/JENKINS/Installing+Jenkins+on+Ubuntu. Accessed 21 Feb. 2018.

E, após isso, um script inicial de construção do sistema foi criado. Esse script instala automaticamente os módulos utilizados pela aplicação, configura as variáveis de ambiente que são usadas e executa os testes unitários.

Precisou-se, então, começar a configurar as etapas de testes automatizados. Na etapa seguinte foi, portanto, criado um conjunto inicial de testes unitários para o sistema *Parasite Watch* e o *JUnit plugin* foi adicionado ao projeto. Além disso, o serviço de notificações de falhas na construção do sistema foi configurado.

Para finalizar a fase de testes, foi necessário configurar o *Jenkins* para os testes de sistema, que utilizaram a ferramenta *Selenium* de forma *headless* (sem que as ações fossem executadas em uma interface com display gráfico) e, logo após essa etapa, os scripts de testes de sistema foram criados.

Com o pipeline quase pronto, foi implementado um script de entrega dos arquivos que compõem o *Parasite Watch* através do protocolo SCP, com chave pública e privada previamente configurada, para que não seja necessário o armazenamento de senhas no script armazenado pelo *Jenkins*.

A última fase do projeto foi configurar o *GitLab* para notificar o *Jenkins* toda vez que um *push* de código fosse realizado no repositório. Isso foi feito através da criação de um *webhook* no *GitLab* e autenticação realizada através de uma chave gerada no *Jenkins* e inserida no *webhook* criado. O *GitLab* permite que diversos eventos sejam utilizados como notificação para o *Jenkins* e, consequentemente, como forma de disparo para o pipeline. O evento 'push' foi escolhido por ser um evento citado na literatura técnica como forma prática de verificação do funcionamento do pipeline. No entanto, outros eventos poderiam servir como *trigger*, como requisições para realizar uma integração (*merge*) de galhos (*branchs*).

Ao longo de todo o projeto, a cada nova etapa implementada, era realizada a verificação manual da integração de ferramentas (representadas na Figura 6 com numeração x.5). Apesar de não estar diretamente relacionada a alguma etapa do pipeline, essas atividades tinham o objetivo de garantir que o próximo passo só seria iniciado após garantir que tudo criado e configurado anteriormente funcionava corretamente e era, portanto, relevante ao processo.

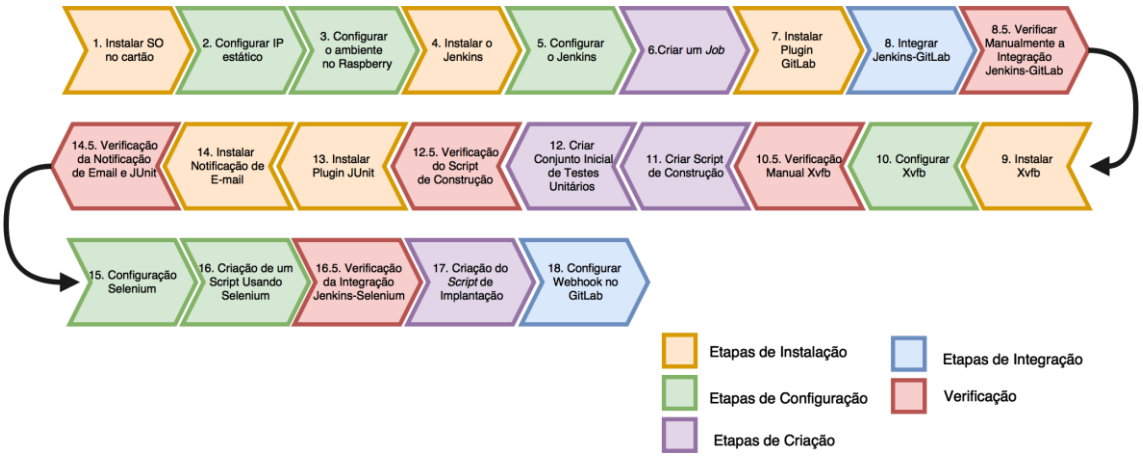


Figura 6. Processo de instanciação do pipeline de Implantação Contínua do Parasite Watch.

4.4.3 – Configuração dos Projetos no *Jenkins*

Um projeto no *Jenkins* é configurado através da opção de configuração disponibilizada pela interface *web*, na página do projeto escolhido. As configurações do projeto são divididas em seis áreas principais: *General*, Gerenciamento de código fonte, *Trigger* de *builds*, Ambiente de *build*, *Build* e Ações de pós-*build*, de acordo com a nomenclatura utilizada pelo *Jenkins* quando se faz uso da interface em língua portuguesa.

Em *General*, o nome do projeto e uma descrição geral são inseridos; a parte de Gerenciamento do código fonte armazena as configurações do sistema de controle de versão utilizado; em *Trigger* de *builds*, são selecionados as ações e os eventos que irão disparar o início do projeto, bem como sua frequência; em Ambiente de *build*, o *Jenkins* apresenta algumas configurações de ambiente, como a opção de iniciar o recurso *Xvfb*, que foi anteriormente explicado e utilizado neste projeto; em *build*, os *scripts* para construção foram descritos no formato *Shell Script Interface*; e em Ações de pós-*build* é dada a opção de, ao finalizar a fase de *build*, iniciar-se outras ações, como o envio de e-mails e a inicialização de outros projetos.

Além das configurações já citadas ao longo do trabalho, três *scripts* foram descritos para construir os projetos utilizados para a composição do pipeline, onde as três principais tarefas são realizadas: construção e execução de testes unitários, execução de um conjunto de testes de sistemas e implantação dos arquivos.

O *script* utilizado para construção e configuração de testes unitários é exibido na Figura 7, onde as duas primeiras linhas são responsáveis por criar um arquivo e armazenar

o *Workspace* utilizado no primeiro projeto e que deverá ser acessado pelos outros projetos que irão sucedê-lo.

O comando da linha seis, instala o módulo em *Python* que armazena os resultados dos testes unitários no formato XML.

Na linha 10, os módulos necessários à execução da aplicação e documentado no arquivo "requirements.txt" são instalados.

As linhas 12 e 14 configuram as variáveis de ambiente necessárias à execução da aplicação *Flask* e a linha 16 executa os testes unitários.

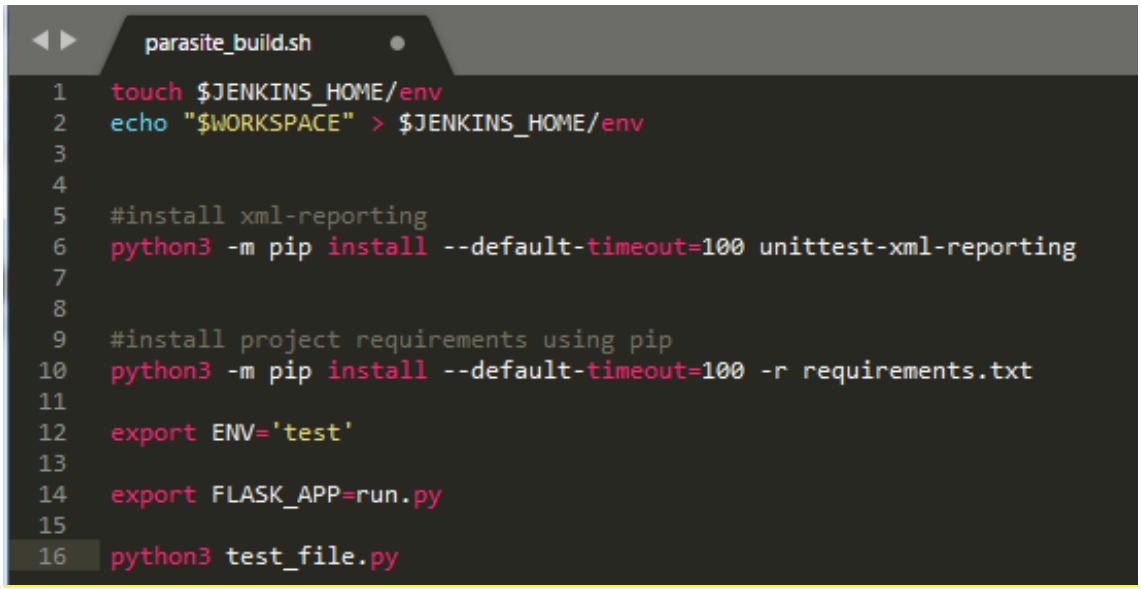


Figura 7.Script de construção da aplicação e execução de testes unitários

A execução dos testes de sistema é mostrada na Figura 8, onde as linhas 2 e 4 recuperam a informação do *Workspace* no qual os arquivos estão sendo armazenados. Além disso, acontece a instalação, através do comando "*python3 –m pip install nome-do-modulo*", dos módulos "*selenium*" e "*pyvirtualdisplay*" (linha 8), necessários para a execução de testes utilizando a ferramenta *Selenium* e é finalizado com a execução dos arquivos referentes aos testes de sistemas na linha 10.

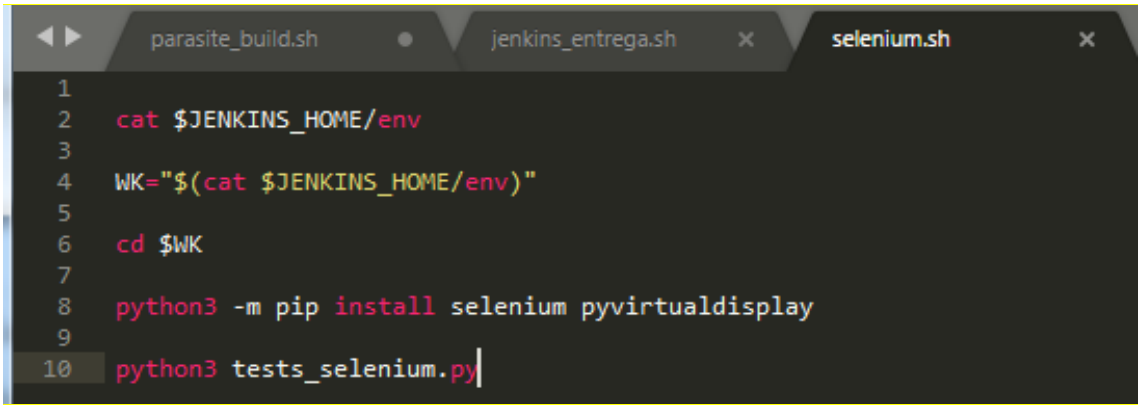


Figura 8.Script de execução de testes de sistema

Por último, a implantação do software utiliza o script mostrado na Figura 9, onde as linhas 1 e 3 recuperam, novamente, a informação a respeito do *Workspace* armazenada no primeiro script. Uma conexão *SSH* é aberta com a finalidade de verificar a existência do diretório utilizado (como indicado nas linhas de 7 a 11), os arquivos são transferidos através do comando *SCP* nas linhas 13 e 14 e, por fim, o arquivo de texto que armazena as informações do *workspace* é removido.

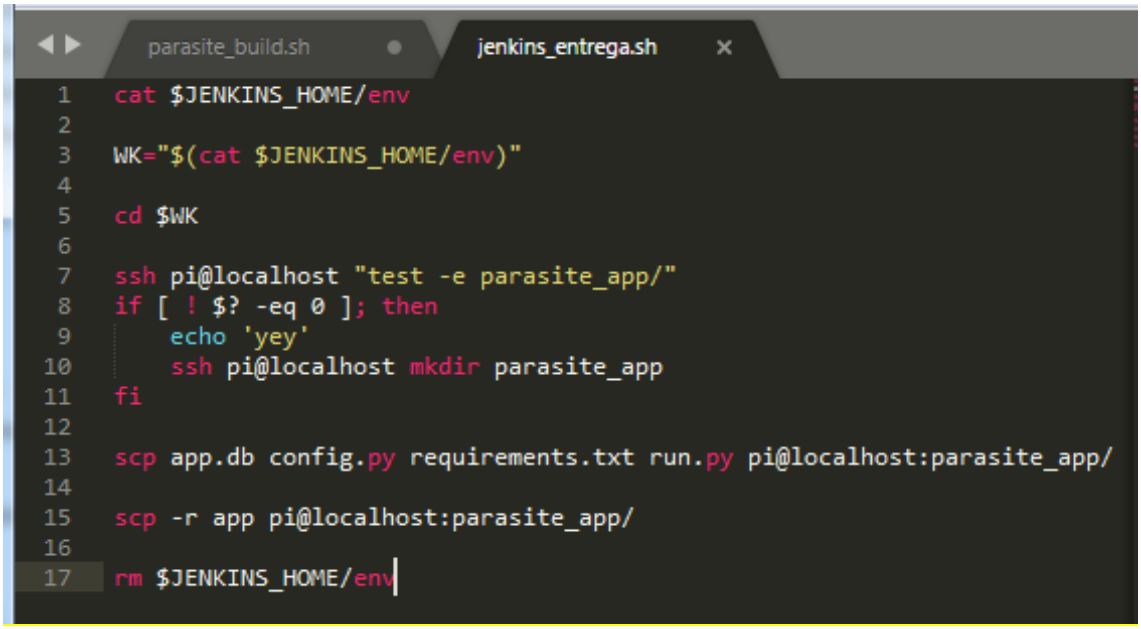


Figura 9.Script de Implantação dos arquivos

Para finalizar, utiliza-se o comando "*python3*" porque as duas versões do Python estão instaladas na mesma máquina, faz-se necessário, portanto, diferenciá-las de alguma forma. Muitos dos comandos utilizados nos *scripts* mostrados acima fazem uso do nome dos arquivos utilizados diretamente (como a execução de testes e a transferência de arquivos) porque foram capturados numa versão inicial do desenvolvimento do projeto *Parasite Watch* com o objetivo de testar o fluxo do *pipeline* como um todo. No entanto, eles podem ser usados de forma genérica, o que não compromete a escalabilidade de aplicação.

4.5 – Execução do Pipeline de Implantação Contínua

As etapas do pipeline foram divididas em três *jobs* no *Jenkins*: parasite_build, parasite_selenium, parasite_deploy , conforme pode ser visto na Figura 10:

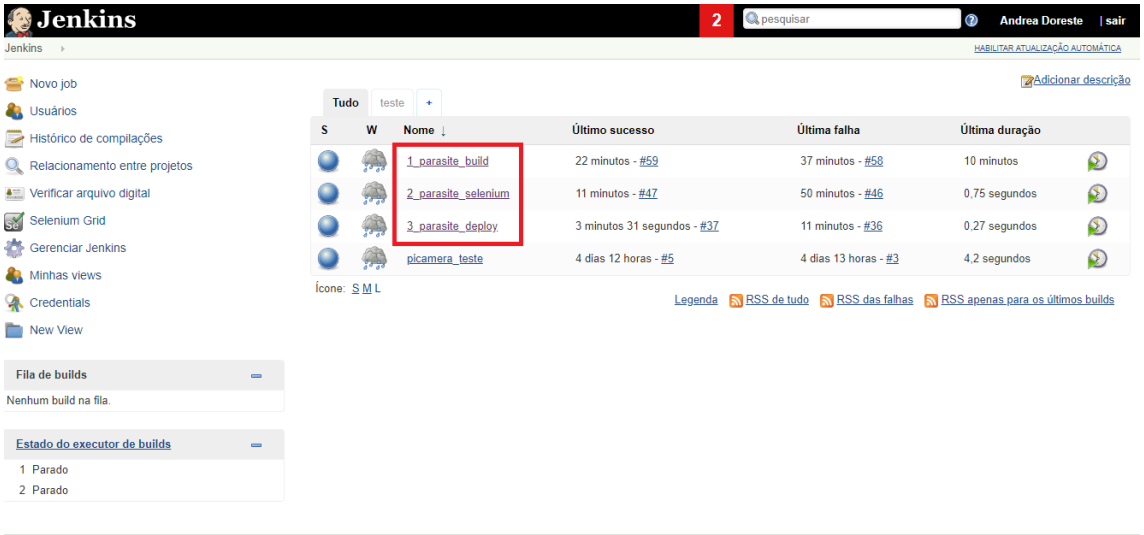


Figura 10. Representação de Jobs no Jenkins

Toda vez que uma modificação é enviada para o *GitLab*, é disparado um processo para o *job* "*parasite_build"* como indicado pelo retângulo de cor vermelha na Figura 11.

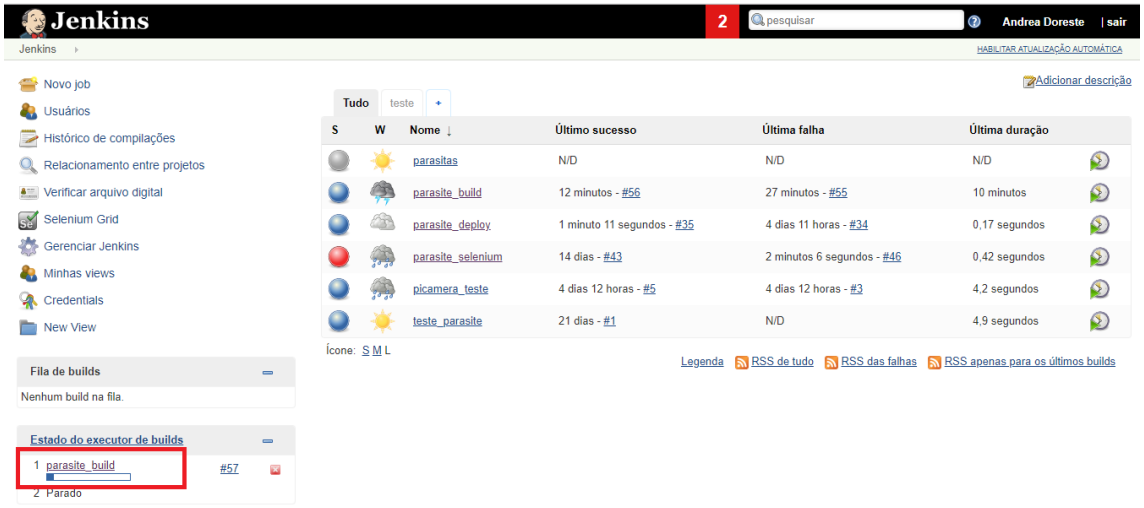


Figura 11. Execução do Job parasite_build no Jenkins

Cada *job* tem uma página principal através da qual suas informações podem ser facilmente acessadas, como o histórico de execuções e os resultados de testes anteriores. A página principal do *job parasite_build* é mostrada na Figura 12.

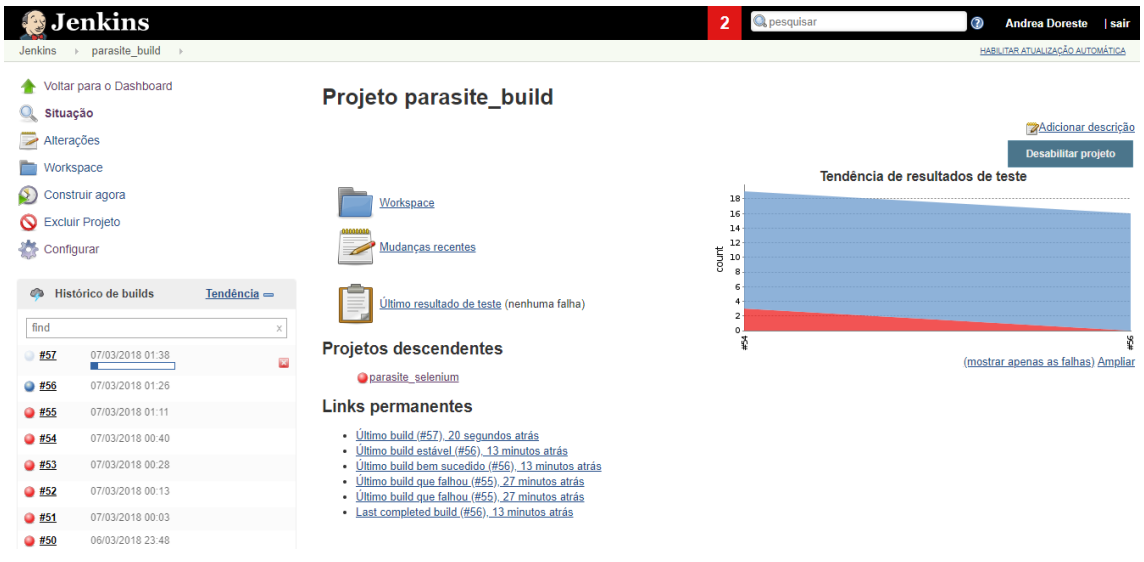


Figura 12. Página Principal do Job parasite_build no Jenkins

Se o projeto falhar, uma notificação é enviada para o e-mail com toda a informação da execução, como mostra a Figura 13 e, conforme mencionado anteriormente, o fluxo de ações é interrompido.

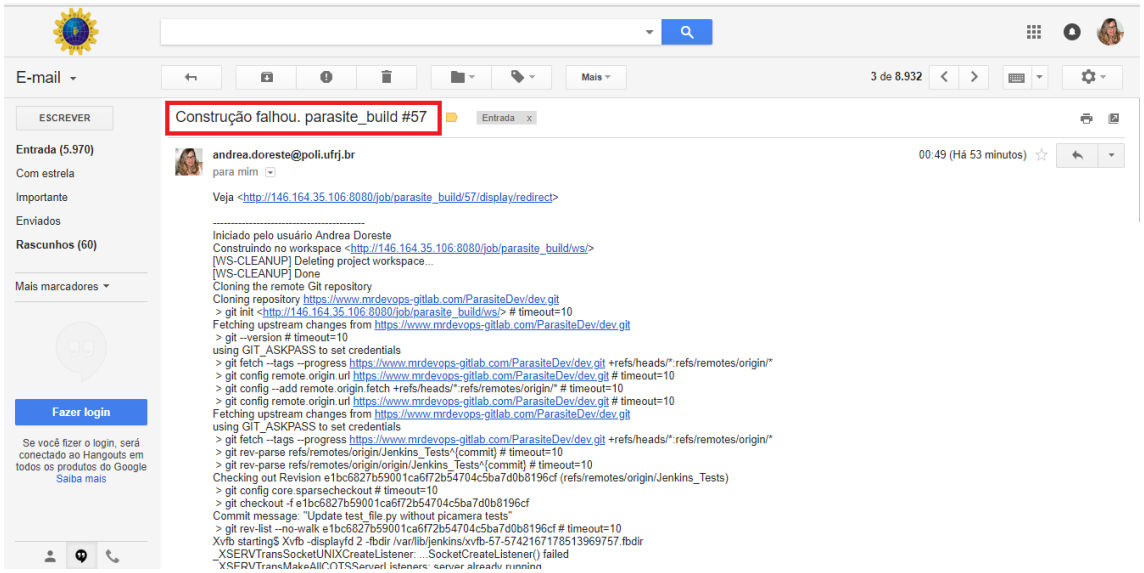


Figura 13. Notificação de Falha enviada pelo Jenkins

Caso a construção ocorra com sucesso, o segundo *job* é disparado como mostra a Figura 14.

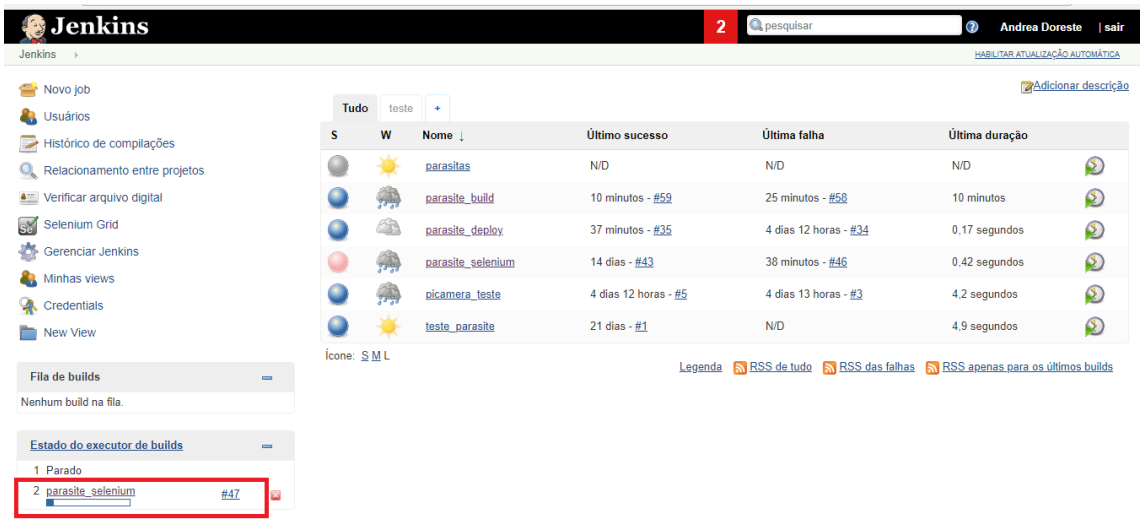


Figura 14. Execução do Job parasite_selenium no Jenkins

Novamente, se não houver falhas durante a execução desse *job*, o último *job* é executado. O resultado do estado final dos três projetos é apresentado na Figura 15.

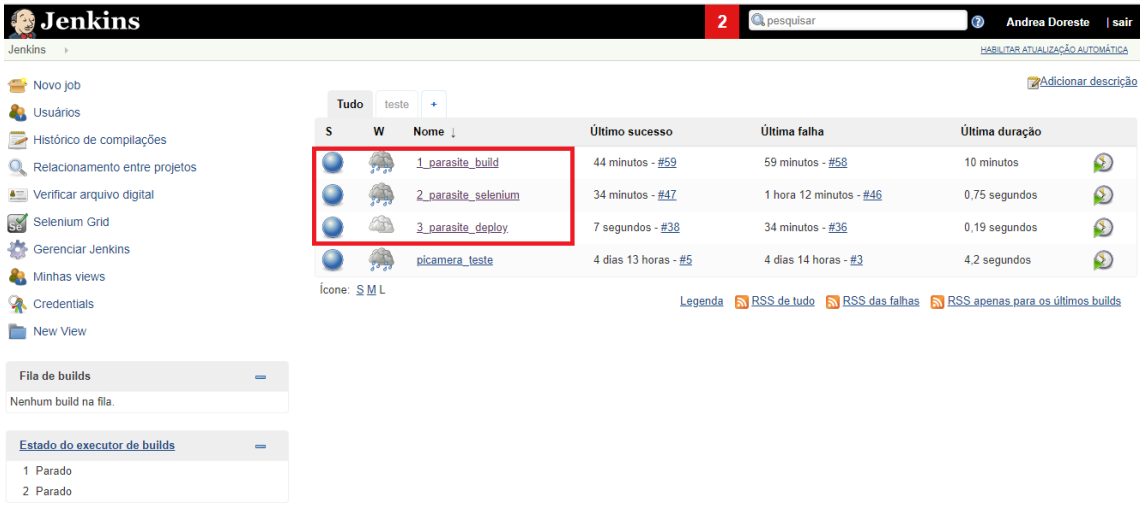


Figura 15. Estado Final dos Jobs no Jenkins

Um detalhe importante que pode ser percebido nas Figuras 10, 11, 14 e 15 é que o *Jenkins* utiliza artefatos gráficos para representar o sucesso de execução dos *jobs*. A cor azul representa uma execução de sucesso, enquanto a cor vermelha representa falha. Além disso, o *Jenkins* representa a estabilidade dos *jobs* com figuras que representam o "clima", quanto melhor o "clima" representado pela figura (sol, nuvem, chuva ou tempestade), mais estável é o projeto.

É possível perceber pela execução do pipeline e pelas imagens mostradas que o *Jenkins* é uma ferramenta de fácil uso e que apresenta diversas funcionalidades.

Todo o processo poderia ter sido implementado com a utilização de apenas um *job* mas, por questão de organização, foi dividido em três *jobs* distintos. A escolha por essa forma de implementação foi feita para que, em caso de falha, se torne mais fácil visualizar e identificar em qual fase de execução do pipeline ela ocorreu, tornando, portanto, a procura pelo problema mais direta e mais prática e, consequentemente, agilizando a correção.

Capítulo 5

Conclusão

5.1 – Resultados e Contribuições

A primeira descoberta do presente trabalho ocorreu ainda em fase de desenvolvimento do pipeline e pode ser descrita como a identificação das peculiaridades e necessidades intrínsecas de sistemas IoT (como a utilização de módulos integrados), que os tornam diferentes de aplicações convencionais do ponto de vista de práticas de desenvolvimento contínuo e deixando evidente a necessidade de configuração de um ambiente específico para aplicações IoT.

Levando em consideração essas características, o principal resultado foi a criação de um ambiente para desenvolvimento contínuo de software em Raspberry *Pi*, especificamente, para o sistema *Parasite Watch*, tornando possível que práticas de Implantação Contínua (com evidencia de viabilidade apresentadas na literatura técnica) sejam implantadas nos processos de software do sistema.

Adicionalmente, minicomputadores da linha *Raspberry Pi* são utilizados em um espectro muito amplo de projetos por uma comunidade grande e ativa. No entanto, durante a execução deste trabalho não foram encontrados relatos de uma configuração de solução similar à que foi criada, tornando-a inédita, inovadora e desafiadora do ponto de vista de tecnologias de desenvolvimento e suas limitações.

Além disso, pôde-se perceber, durante a criação da solução, configuração e integração de ambientes que algumas ferramentas utilizadas, como o *Jenkins* e o *Selenium*, apesar de amplamente conhecidas e utilizadas, apresentam limitações de configuração quando utilizadas em ambientes alternativos, como o utilizado nesta solução. Por exemplo, o *Selenium* utiliza *WebDrivers* para interagir com navegadores como Chrome e Firefox. No entanto, a comunidade responsável por desenvolver o *WebDriver* do Chrome não disponibiliza um driver para o navegador Chromium, utilizado pelo sistema Raspbian no *Raspberry Pi* e o WebDriver do Firefox, apesar de disponível, apresentou problemas de desempenho que o inviabilizaram como solução. Já

no caso do *Jenkins*, alguns *plugins* (que não foram utilizados neste projeto) apresentaram mau funcionamento durante a execução.

Percebe-se, portanto, que as soluções atualmente mais utilizadas são limitadas a ambiente de software específicos, mais "convencionais". O questionamento que fica é se isso pode ser considerado uma limitação imposta pela solução proposta ou representam oportunidades de melhoria das ferramentas utilizadas.

5.2 - Limitações

As principais limitações encontradas para o pipeline desenvolvido são:

- O *Raspberry Pi* tem apenas um gigabyte (1Gb) de memória interna, o que faz com que as etapas do pipeline demorem para serem executadas;
- O *Raspberry Pi* possui uma arquitetura diferente do padrão (ARMv8), então alguns *plugins* do *Jenkins* (criadas para sistemas Unix "convencionais") podem não funcionar corretamente.
- Para que o pipeline funcione corretamente, é preciso o apoio e conscientização da equipe de desenvolvimento. Afinal, uma estrutura que apoie o desenvolvimento contínuo não funciona se as práticas de desenvolvimento não forem contínuas.
- Devido a necessidade de efetuar o download dos pacotes necessários a aplicação, para o correto funcionamento do pipeline é preciso garantir o acesso à Internet, mesmo que todos os outros dispositivos com os quais ele interage (o servidor de integração *GitLab* e os dispositivos *Raspberry* nos quais a entrega será feita) estejam conectados na mesma rede.

5.2 – Trabalhos Futuros

Tendo em vista as limitações apontadas, as melhorias a serem implementadas estão listadas abaixo:

- A forma como a etapa de implantação acontece deve ser aperfeiçoada. Com a entrega feita não só para os *Raspberry Pi* cujos endereços IPs são conhecidos, mas também para um servidor, de forma que computadores

não conectados à Internet no momento da execução do pipeline possam ter acesso as melhorias da aplicação e fazer o download e atualização da aplicação.

- Deve ser desenvolvida uma funcionalidade onde o sistema cliente detecte quando uma nova versão está disponível e realize a atualização (download e instalação) a partir de um repositório no qual a aplicação ficará disponível.

5.3 – Considerações Finais

Nos dias atuais, muito é falado sobre técnicas de desenvolvimento contínuo, mas a maioria dos cenários em que essas técnicas são aplicadas podem ser consideradas "convencionais", sobretudo em aplicações Web. É preciso entender os desafios que essas práticas e ferramentas encontrariam em nichos mais específicos, como o desenvolvimento para aplicações em Internet das Coisas que utilizam plataformas diferentes e conexão com sensores e dispositivos.

A partir do trabalho desenvolvido, alguns problemas intrínsecos de um nicho específico (aplicações desenvolvidas para *Raspberry Pi*), como problemas de compatibilidade e limitações do ferramental utilizado e das integrações exigidas, puderam ser identificados e tratados.

O mundo está entrando em um novo paradigma de sistemas de software, mas não se pode afirmar que as soluções atuais são suficientes para apoiá-lo. Portanto, esse trabalho apresenta valor, não só por propor e desenvolver práticas de desenvolvimento contínuo para um novo cenário, mas também por apontar que as soluções atuais não são ainda suficientes. Se Internet das Coisas for mesmo o futuro (ou um presente próximo), é preciso começar a se pensar e desenvolver soluções que apoiem a engenharia do software contemporâneo.

Bibliografia

[1] CERUZZI, P. E. *A History of Modern Computing*, 2 ed., Cambridge, Massachusetts, MIT Press Edition, 2003

[2] PFLEEGER, S. L. *Engenharia de Software: Teoria e Prática*. 2. ed., Capítulo 1, São Paulo, Pearson Education do Brasil, 2004.

[3] ATZORI, L., IERA, A., MORABITO, G. The Internet of Things: A survey. *Computer Networks*, [s.l.], v. 54, n. 15, p.2787-2805, out. 2010. Elsevier BV.

[4] FITZGERALD, B., STOL, K. "Continuous software engineering: A roadmap and agenda". Journal of Systems and Software, [s.l.], v. 123, p.176-189, jul. 2015. Elsevier BV.

[5] KLEPPER, S. et al. "Introducing Continuous Delivery of Mobile Apps in a Corporate Environment: A Case Study". *IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering*, 2015.

[6] MEYER, M. "Continuous Integration and Its Tools". IEEE Software, v. 31, n. 3, p. 14-16, 2014.

[7] VENCESLAU, D. K., FRANÇA, B. B. N. *Reporting Deployment Pipeline Studies*. Coppe/UFRJ, Rio de Janeiro, 2016.

[8] CHEN, L., "Continuous Delivery: Huge Benefits, but Challenges Too". IEEE Software, v. 32, n. 2, p. 50-54, 2015.

[9] SMART, J., Jenkins: The definitive guide. 1 ed., New Zealand, O'Reilly Media, Inc, 2011.

[10] WORLD HEALTH ORGANIZATION et al. First WHO report on neglected tropical diseases: working to overcome the global impact of neglected tropical diseases. First WHO report on neglected tropical diseases: Working to overcome the global impact of neglected tropical diseases. 2010.

[11] RICCIARDI, A.; NDAO, M. Diagnosis of parasitic infections: what's going on?. Journal of biomolecular screening, v. 20, n. 1, p. 6-21, 2015.

[12] DORESTE, A., TRAVASSOS, G. H., Subsistema de Coleta de Imagens do Parasite Watch, Caderno de Resumos: Centro de Tecnologia, 8a Semana de Integração Acadêmica, out. 2017