UNIVERSIDADE DO VALE DO TAQUARI - UNIVATES
CURSO DE ENGENHARIA DE SOFTWARE

UMA PLATAFORMA WEB PARA SUPORTE AO DESENVOLVIMENTO DE APLICAÇÕES COM ÊNFASE NO PROCESSO DE PROVISIONAMENTO E DEPLOY

Artur Comunello

Lajeado, novembro de 2018.

Artur Comunello

UMA PLATAFORMA WEB PARA SUPORTE AO DESENVOLVIMENTO DE APLICAÇÕES COM ÊNFASE NO PROCESSO DE PROVISIONAMENTO E DEPLOY

Monografia apresentada ao Centro de Ciências Exatas e Tecnológicas da Universidade do Vale do Taquari - UNIVATES, como parte dos requisitos para a obtenção do título de Bacharel em Engenharia de Software.

Orientador: Prof. Me. Fabrício Pretto

Lajeado, novembro de 2018.

Artur Comunello

UMA PLATAFORMA WEB PARA SUPORTE AO DESENVOLVIMENTO DE APLICAÇÕES COM ÊNFASE NO PROCESSO DE PROVISIONAMENTO E DEPLOY

A Banca examinadora abaixo, aprova a Monografia apresentada na disciplina Trabalho de Conclusão de Curso I, da Universidade do Vale do Taquari - UNIVATES, como exigência para obtenção do grau de Bacharel em Engenharia de Software:

Prof. Me. Fabrício Pretto – orientador

Universidade do Vale do Taquari - UNIVATES

Lajeado, novembro de 2018.

RESUMO

A computação em nuvem está cada vez mais presente no desenvolvimento de todos os tipos de sistemas e ela proporciona um rápido crescimento da estrutura de hardware das aplicações. As oportunidades disponíveis no mercado mudam constantemente, o que força as equipes de desenvolvimento responderem rapidamente a essas mudanças. Para que as necessidades do mercado sejam atendidas o quanto antes, as mudanças devem ser realizadas no menor tempo possível. Diante disso, é essencial a existência de ferramentas que auxiliem as equipes de desenvolvimento e operações no gerenciamento de instalações e atualizações de aplicações em ambientes em nuvem. Tendo em vista esse cenário, o objetivo do presente trabalho foi desenvolver uma ferramenta que permita realizar provisionamento de servidores e *deploy* aplicações. A ferramenta foi desenvolvida e validada junto com profissionais da área de TI, demonstrando resultados positivos e um grande potencial para virar um produto de mercado.

Palavras-chave: Engenharia de Software. Gerenciamento de Configuração. Provisionamento. Deploy. DevOps. Integração Contínua. Entrega Contínua.

ABSTRACT

Cloud computing is increasingly present in the development of all types of systems and it provides a rapid growth in the hardware structure of the applications. The opportunities available in the market are constantly changing, which forces development teams to respond quickly to these changes. In order for market needs to be met quickly, changes must be made in the shortest time possible. Faced with this, it is essential to have tools that assist development teams and operations in managing installations and upgrading applications in the cloud environment. Considering this scenario, the objective of the present work is to develop a tool that performs provisioning and deploying. The tool was developed and validated along with IT professionals, showing positive results and great potential to become a market product.

Keywords: Software Engineering. Configuration Management. Provisioning. Deploy. DevOps. Continuous Integration. Continuous Delivery.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS

CCB:	Change Control Board
GCS:	*Gerência de Configuração de Software*
IC:	Itens de Configuração
IEEE:	Institute of Electrical and Eletronics Engineers
SWEBOK:	*Software Engineering Body of Knowledge*
TI:	*Tecnologia da Informação*
VCS:	*Version Control System*

SUMÁRIO

1 INTRODUÇÃO

A sociedade contemporânea não poderia desempenhar suas atividades da mesma maneira se softwares não existissem. A produção e distribuição industrial, segurança, saúde, educação, entretenimento e outros setores da economia possuem processos parcialmente ou totalmente informatizados. Todos esses processos incluem equipamentos elétricos com computador e um software que o controla. Portanto, para o atual funcionamento das sociedades a Engenharia de Software é completamente essencial (SOMMERVILLE, 2012).

Sommerville (2012) define a Engenharia de Software como a disciplina que se preocupa com todos os pontos importantes de um software, e todos os diferentes tipos de aplicações que precisam dela. Diferentes programas podem usar diferentes técnicas, sendo que os fundamentos básicos da Engenharia de Software são aplicáveis a qualquer sistema. Esses fundamentos incluem gerência de processo, confiança, desempenho de software, reúso de software, Engenharia de Requisitos e Gerência de Configuração, pro exemplo.

O sucesso dos softwares que são utilizados por empresas como viabilizadores do seu negócio, está em parte na capacidade que eles têm de apresentar novas e interessantes funcionalidades antes que seus concorrentes. Para isso acontecer, as equipes de desenvolvimento estão sempre buscando melhorar o produto e corrigir erros. No momento em que as melhorias são implantadas no ambiente de produção, outras preocupações passam a existir, como suporte, monitoramento, segurança, usabilidade, entre outras.

Essa dinâmica de inserir novidades nos softwares e a prioridade de manter o ambiente de produção estável, fez com que alguns departamentos de tecnologia da informação (TI) criassem setores com divisões claras de responsabilidades. As equipes de desenvolvimento ficaram responsáveis pela produção e manutenção do código-fonte, enquanto as equipes de

operações ficaram responsáveis por cuidar desse código-fonte e prezar pela estabilidade dos sistemas (SATO, 2014).

Embora seja uma divisão lógica, essa abordagem torna lenta a entrada de novas funcionalidades para os produtos, pois a equipe de desenvolvimento produz mudanças ao passo que a equipe de operações as evita. Essa aversão a mudanças, da equipe de operações, se dá porque ela deve reagir imediatamente a qualquer sinal de problemas buscando uma solução. Quanto maior o número de mudanças que entram em produção, maiores são as chances de algo sair do controle. Mudanças em geral trazem riscos à estabilidade do sistema (SATO, 2014).

Para controlar esse conflito criado entre a equipe de desenvolvimento e a de operações, são criados processos que definem o modo de trabalho de cada equipe. Quando a equipe de desenvolvimento empacota as novas modificações e passa a responsabilidade para a equipe de operações disponibilizá-las no ambiente de produção, parece que o “código-fonte foi arremessado sobre um muro”, sem a comunicação devida. Em muitos casos, para contornar o problema na troca de informações entre as equipes, existe um processo de comunicação controlado por um sistema de *tickets*, o que diminui a frequência de *deploys*, pois torna burocrático o processo como um todo.

A Gerência de Configuração de Software (GCS) surgiu para resolver os problemas do desenvolvimento e fornecimento de software, controlar e acompanhar mudanças, registrar a evolução de um projeto e estabelecer a integridade do sistema, que também são conhecidos como controle de mudança e controle de versão e integração contínua (CMMI, 2010). A maior parte das empresas que desenvolvem software mantém equipes que trabalham simultaneamente nos mesmos produtos, corrigindo problemas, adicionando funcionalidades, atualizando tecnologias, etc. Para que isso aconteça de forma natural e organizada, elas precisam usar as metodologias da Gerência de Configuração (ANDRADE, 2015).

Uma abordagem moderna fortemente baseada nos princípios da GCS é o conceito DevOps. Esse modelo de trabalho busca fazer com que as equipes de desenvolvimento e operações possam trabalhar melhor juntas, pensar de modo semelhante e dividir responsabilidades. Uma empresa direcionada para o conceito de DevOps tem suas equipes integradas para aumentar a produtividade, melhorar a colaboração e manter os fluxos de trabalho automatizados.

A IBM (2017) define o DevOps como uma habilidade essencial para empresas manterem a Entrega Contínua e que possibilita aproveitar melhor as oportunidades do mercado, reduzir o tempo de resposta para clientes e acelerar a inovação. Isso tudo sem nenhum comprometimento em termos de custo, qualidade e riscos. A integração contínua, do inglês *continuous delivery*, é o estado futuro alcançado com a aplicação de uma abordagem DevOps. Ela é definida por Sato (2014) como o modelo de trabalho que tenta reduzir o tempo entre o surgimento de uma ideia e sua implementação.

Na era pré-internet, as empresas tinham processos de *deploy* manuais, nada semelhante com o modelo DevOps, grandes blocos de código-fonte eram produzidos e inseridos no ambiente de produção de uma só vez. Essa prática demandava um grande esforço da equipe de operações para que nenhum serviço parasse.

Com o advento da Internet e a Computação em Nuvem, possibilitou empresas crescerem em escala. O modo de trabalho DevOps acelera ainda mais esse crescimento, pois ele tenta automatizar todas as tarefas, testes, fluxos de trabalho, criação de infraestrutura. Na atualidade, é impensável uma grande empresa realizar *deploys* manuais ou sem o auxílio de ferramentas de Gerência de Configuração.

1.1 Motivação

A competição no mercado para oferecer os melhores serviços pelos melhores preços está presente em todas as áreas de negócio e não é diferente com software. Como observado, é essencial ter ferramentas de apoio a etapas como a de *build* e de *deploy* automatizado de software.

Empresas que insistem em desenvolver e manter softwares sem auxílios na Gerência de Configuração, inevitavelmente ficam atrás de competidores que estão atentos para as novas tendências e ferramentas que possam trazer velocidade na entrega de seus produtos.

Existem ferramentas, inclusive *open source*, disponíveis para etapas como *build* (Ant, Jenkins, Gitlab), e *deploy* (Deployer, Go, Travis), porém existe um espaço para ser explorado na construção de uma plataforma que integre de maneira simples, visual e automatizada várias etapas de um processo de Entrega Contínua automatizado como conceituado em DevOps,

desde o provisionamento, até o *deploy* automatizado para múltiplos servidores, com verificação de qualidade e *rollback*.

É impensável hoje, implementar uma funcionalidade de um produto em 1000 servidores diferentes sem algum tipo de ferramenta de suporte. Quando problemas acontecem e um *rollback* é necessário, esse torna-se momento mais crítico no qual uma ferramenta deve voltar a versão em todos os clientes. Essa lacuna de ferramentas que dão esse poder de provisionar e entregar software continuamente a múltiplos servidores que motiva o desenvolvimento deste trabalho.

Considerando a dificuldade de gerenciar uma grande quantidade de servidores, que uma empresa de software possa ter, o objetivo do presente trabalho é melhorar o processo de Entrega Contínua automatizado, desde o provisionamento até o *deploy* automatizado para múltiplos servidores, por meio da criação e validação de uma ferramenta web de suporte a esse processo.

A ferramenta desenvolvida busca oferecer funcionalidades que facilitem o provisionamento de servidores e *deploy* contínuo de *releases* de software. Para tal, ela integra-se com ferramentas de controle de versão de código-fonte, permitindo configurar *builds* automatizados, e realizar *deploy* em servidores provisionados automaticamente em *providers* que disponibilizem API's de controle, tais como: Digital Ocean, Linode, Vultr, entre outros.

1.2 Objetivos

Nesta seção são apresentados os objetivos gerais e específicos do presente trabalho.

1.2.1 Objetivo geral

O objetivo geral do presente trabalho é, apresentar o desenvolvimento de uma ferramenta que torne o processo de provisionar servidores e realizar *deploys* uma tarefa de alto nível.

Quando este trabalho menciona a expressão *alto* nível, significa que a ferramenta desenvolvida procura tornar tarefas complexas mais simples. Desta maneira, possibilitando que usuários com menos experiência consigam colocar seus trabalhos em produção.

1.2.2 Objetivos específicos

Os objetivos específicos são:

a) Levantar os requisitos de uma ferramenta para modelagem de software.

b) Modelar a arquitetura do software.

c) Implementar uma ferramenta de provisionamento e *deploy* para vários servidores.

d) Avaliar tanto qualitativa quanto quantitativamente a ferramenta desenvolvida com profissionais da área de TI.

e) Analisar os resultados obtidos.

1.3 Organização do trabalho

Para auxiliar na compreensão total do presente trabalho, a ordem de apresentação de seus capítulos foi dividida da seguinte maneira:

O capítulo 2 contém o referencial teórico, que foi utilizado como base para o desenvolvimento do trabalho, que possui os seguintes subcapítulos: Engenharia de Software, Gerenciamento de Configuração de Software, Ferramentas de Controle de Versão e Mudança, DevOps, Integração Contínua e Provisionamento.

No capítulo 3 é apresentada a metodologia que foi utilizada pra realizar o presente trabalho e o método científico que ele se enquadra. Por fim, o capítulo ainda apresenta as etapas para o desenvolvimento da ferramenta.

O capítulo 4 apresenta uma visão geral da ferramenta desenvolvida, também são detalhados os requisitos funcionais e não funcionais, tecnologias utilizadas, interface, arquitetura e artefatos da aplicação.

O capítulo 5, apresenta a avaliação dos resultados obtidos aplicando um questionário para um grupo de profissionais da área de TI.

O capítulo 6 contém as considerações finais que foram tiradas com a realização deste presente trabalho.

2 REFERENCIAL TEÓRICO

2.1 Engenharia de Software

O significado do termo engenharia de software já foi determinado e atualizado por centenas de autores nas últimas décadas, cada um deles levando em consideração suas definições pessoais para tal (PRESSMAN, 2011). Anterior a todas essas definições, em 1969 Friedrich Ludwig Bauer definiu pela primeira vez o que é Engenharia de Software, em uma conferência sobre o tema, para solucionar a chamada crise do software (HIRAMA, 2012).

O aumento da complexidade e das demandas no desenvolvimento de software, aliado à inexistência de técnicas para solucionar esses problemas, deram origem a crise do software que aconteceu próximo ao início da década de 70 (ENGHOLM JR., 2011).

Friedrich definiu a Engenharia de Software como a maneira de conseguir software confiável e de modo econômico, através do emprego de princípios sólidos de Engenharia (NATO, 1969).

Pressman (2011) apontou a superficialidade dessa primeira definição e fez alguns questionamentos sobre pontos ainda não explicados por Friedrich. Mesmo servindo de base, essa definição ainda levantava importantes questionamentos que os engenheiros de software procuravam solucionar. Por exemplo, Bauer não definiu o que nem quais eram os princípios tradicionais da engenharia, tampouco tratou sobre pontos importantes como qualidade de software, satisfação do cliente ou prazos de entrega.

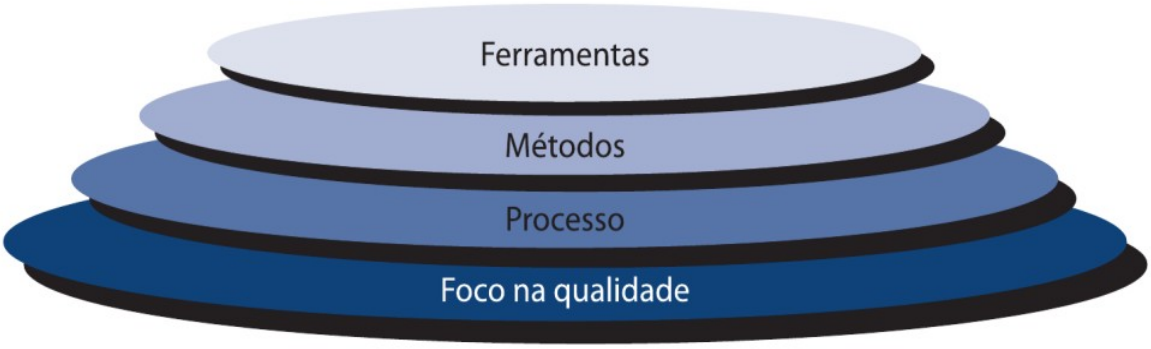
A definição de Engenharia de Software continuou a ser atualizada com o passar dos anos e em 1990 o Institute of Electrical and Eletronic Engineers (IEEE) definiu Engenharia de

Software como, a aplicação disciplinada, sistemática e que possa ser medida na produção e manutenção de software (IEEE Std 828-1990, 1990).

Engholm Jr. (2011) diz que o surgimento da Engenharia de Software tem o objetivo de utilizar processos repetíveis e eficazes nos ciclos de desenvolvimento e manutenção de software, de maneira que os custos e riscos sejam diminuídos e que a qualidade seja aumentada.

A afirmação de Engholm fica próxima da definição de Sommerville (2012) em que o resultado da preocupação com todos os aspectos do desenvolvimento de software forma a Engenharia de Software.

Figura 1 - Camadas da engenharia de software



Fonte: Pressman(2011).

Conforme visto na Figura 1, Pressman (2011) divide a Engenharia de Software em quatro camadas: foco na qualidade, processo, métodos e ferramentas. Sendo que o foco na qualidade é o alicerce que sustenta a Engenharia de Software, pois abordagens mais assertivas no desenvolvimento de software são resultados da cultura do aperfeiçoamento que a busca pela qualidade traz.

Buscando criar uma forma de consenso sobre todas as áreas de conhecimento da Engenharia de Software, foi criado o guia *Software Engineering Body of Knowledge* (SWEBOK) que é uma iniciativa do IEEE. Ele é um guia com as melhores práticas da Engenharia de Software. O SWEBOK é um compêndio com conhecimentos acumulados em 4 décadas por profissionais do mundo todo envolvidos com a Engenharia de Software (SWEBOK 2014).

O SWEBOK busca promover a Engenharia de Software através de uma visão consciente mundial, para que seja definido claramente o limite da Engenharia de Software em

relação com outras disciplinas. Além disso, também são objetivos do SWEBOK fornecer uma base para o desenvolvimento curricular, certificação individual e caracterizar claramente quais são os conteúdos das disciplinas da Engenharia de Software (SWEBOK, 2014).

O SWEBOK (2014) define as áreas da Engenharia de Software da seguinte maneira:

a) **Requisitos de Software:** São as restrições e necessidades de um software que juntos formam o produto e resolvem problemas do mundo real.

b) **Design de Software:** Componentes, interfaces, arquiteturas e outras características do sistema são definidas. Pressman (2011) define design de software como a área que traduz os requisitos em um documento para o desenvolvimento do software.

c) **Construção de Software:** O desenvolvimento propriamente dito do software, junto com testes de unidade e integração. Essa área se preocupa com a minimização da complexidade, antecipação de mudanças e implementação seguindo padrões de construção.

d) **Testes de Software:** Avaliação da qualidade software procurando identificar problemas e defeitos. Esta etapa revela erros que são corrigidos antes da entrega para o cliente.

e) **Manutenção de Software:** Todas as mudanças e modificações que um software sofre durante o completo ciclo de utilização. Sejam correções de problemas, melhorias ou novas funcionalidades.

f) **Gerenciamento de Configuração de Software:** Procura manter a integridade de todos os produtos de software durante seu ciclo de utilização. Mantém o controle sistemático de todas as mudanças de configuração e versão do software.

g) **Gerenciamento de Engenharia de Software:** Aglomerado de diversas áreas de gerenciamento: medição, documentação, monitoramento, coordenação e planejamento.

h) **Engenharia de Processo de Software:** Tem como objetivo implementar novos processos e melhorar os existentes, seja no escopo ou organização do projeto.

i) **Ferramentas e Métodos de Software:** Por meio dos métodos é possível pesquisar e encontrar maneiras que aumentem a produtividade da equipe. E ferramentas são as que foram criadas para tornar o desenvolvimento do software mais fácil. Essas ferramentas normalmente automatizam processos ou testes.

j) **Qualidade de Software:** A qualidade de software está intimamente ligada com a qualidade que os processos são desempenhados e através dos quais o software é desenvolvido. Portanto, para que um projeto tenha qualidade é necessário que todo o processo de desenvolvimento seja cuidadosamente definido através de acompanhamento e documentação.

Pressman (2011) afirma que o desenvolvimento de softwares dentro do prazo e de forma racional são possibilitados pela camada de processos. Além disso, ela também é encarregada por definir uma metodologia responsável por entregar a tecnologia de Engenharia de Software.

Tarefas como suporte, construção de programa, testes, modelagem de projeto, análise de requisitos e comunicação são englobadas pela camada de métodos. As técnicas e informações necessárias para desenvolver um programa são fornecidas pelos Métodos de Software (PRESSMAN, 2011).

Suporte semiautomatizado ou até mesmo automatizado para as camadas de processo e de métodos é dado através da última camada definida por Pressman (2011) a de ferramentas.

Sommerville (2012) resume a Engenharia de Software como a área que cuida de todos os pontos do desenvolvimento de um software. Sendo que todas as atividades propostas na produção do software fazem parte do processo de software.

A definição de processo de software é fortemente relacionada com a definição geral de processo. Conceitualmente, ações sequenciais com a meta de realizar uma tarefa é um processo. Em grande parte as atividades realizadas no cotidiano podem ser vistas como

processos, eles são usados para desenvolver, manusear, modificar, criar e manter, sistemas ou produtos (ENGHOLM JR., 2011).

Seguindo a definição de geral de processo, Sommerville (2012) afirma que um conjunto de ações relacionadas que levam a produção de um programa são um processo de software.

Dois grandes autores da área de Engenharia de Software tem ideias levemente diferentes quanto ao modelo básico de processo de software. Sommerville (2012) diz que entre os mais diversos processos de software todos devem incluir quatro itens essenciais:

a) **Especificação de Software:** Definir o que o software deve ou não atender.

b) **Projeto e Implementação de Software:** Produzir o software conforme as especificações previamente definidas.

c) **Validação de Software:** O Software deve passar por validação garantindo que ele atende às especificações.

d) **Evolução de Software:** Para atender às novas necessidades do cliente o software deve estar pronto para mudanças.

Outra abordagem é a seguida por Pressman (2011) na qual o autor afirma que uma metodologia de processo de software geral deve abranger cinco itens principais:

a) **Comunicação:** Antes de iniciar qualquer trabalho é de suma importância que os objetivos de todos os interessados, também conhecidos como *stakeholders*, sejam apresentados.

b) **Planejamento:** Criar um cronograma com as tarefas que deverão ser realizadas e em quanto tempo. Um levantamento de possíveis riscos e recursos necessários.

c) **Modelagem:** Criar modelos que melhorem o entendimento do problema a ser resolvido e quais as necessidades devem ser solucionadas.

d) **Construção:** Geração do código-fonte do software junto com testes para revelar possíveis falhas do desenvolvimento.

e) **Emprego:** O software é entregue para o cliente que deve avaliar se a ferramenta atende suas necessidades e deve fornecer um feedback conforme avaliado.

As metodologias apresentadas pelos autores são muito semelhantes e se aplicam perfeitamente no desenvolvimento de programas simples, para aplicações maiores e mais complexas outros pontos devem ser levados em consideração. Conhecidos como atividades de apoio, elas são aplicadas durante todo o processo de desenvolvimento e ajudam a controlar os riscos, as mudanças e a qualidade. (PRESSMAN, 2011).

Segundo Pressman (2011) as típicas atividades de apoio são:

a) **Controle e acompanhamento do projeto:** Acompanhamento do progresso do projeto pela equipe a fim de tomar medidas necessárias para que o cronograma seja cumprido.

b) **Administração de riscos:** Todos os riscos que tenham possibilidade de atrapalhar os resultados do projeto devem ser avaliados.

c) **Garantia da qualidade de software:** Atividades que garantam a qualidade do produto devem ser definidas.

d) **Revisões técnicas:** Erros que possam afetar o produto devem ser identificados e corrigidos antes que eles afetem outras áreas do sistema, esse processo ocorre através da revisão dos artefatos da Engenharia de Software.

e) **Medição:** Ajuda na entrega do sistema conforme o levantamento de requisitos.

f) **Gerenciamento da configuração de software:** Controla os impactos das mudanças durante o processo de desenvolvimento.

g) **Gerenciamento da reusabilidade:** Estabelece como os artefatos de software são reutilizados.

A qualidade do software está fortemente ligada ao Gerenciamento de Configuração, sendo ele uma das mais importantes atividades de apoio. A Gerência de Configuração é uma disciplina que busca tornar o desenvolvimento de software mais fácil mantendo-o íntegro,

monitorando, administrando e controlando todas as mudanças que possam ocorrer ao longo do projeto.

2.2 Gerenciamento de Configuração de Software

É normal que durante o desenvolvimento de software mudanças aconteçam, no entanto elas podem criar confusão à medida que seu número aumenta. Quando modificações são realizadas sem uma análise ou um registro devido, a confusão nasce dentro da equipe de desenvolvimento (PRESSMAN, 2011).

A GCS se preocupa com toda a gestão de qualidade de um produto de software. Em 1986 Babich definiu a atuação da área:

> A arte de coordenar desenvolvimento de software para minimizar a confusão é chamada de gestão de configuração. A gestão de configuração é a arte de identificar, organizar e controlar modificações no software que está sendo criado por uma equipe de programação. O objetivo é maximizar a produtividade minimizando os erros (PRESSMAN, 2011, p. 514).

A GCS fornece as ferramentas e os métodos para controlar e identificar todo o desenvolvimento do software. As atividades da GCS incluem a revisão, a aprovação e o controle de mudanças, o monitoramento e a notificação das mudanças, o controle de versões, as auditorias, as revisões do produto de software em evolução e o controle de documentação de interface e fornecedor do projeto (IEEE Std 828-1990, 1990).

A GCS está ligada com as ferramentas, os processos e as políticas para gerenciamento de mudanças dos sistemas de software. Com a evolução de projetos de software é fácil perder o controle de quais versões de componentes e mudanças foram adicionadas em cada nova versão do sistema. Se não forem implementados procedimentos de GCS adequados, é possível que inúmeros retrabalhos sejam realizados, pois pode-se usar versões errôneas do projeto (SOMMERVILLE 2012).

> O gerenciamento de configuração é útil para projetos individuais, pois é fácil uma pessoa esquecer quais mudanças foram feitas. É essencial para projetos em equipe em que vários desenvolvedores trabalham ao mesmo tempo em um sistema de software. Às vezes, esses desenvolvedores trabalham no mesmo local, mas, cada vez

> mais, as equipes de desenvolvimento são distribuídas, com membros em diferentes locais pelo mundo (SOMMERVILLE, 2012, p. 475).

Através da fala de Sommerville (2012) anterior, é possível verificar o quão vital é a GCS dentro de uma empresa onde todos os processos devem ser organizados e meticulosos. E o uso de GCS possibilita que todos os participantes do desenvolvimento de um projeto tenham acesso a todas as informações necessárias sem que um interfira no trabalho de outro.

A finalidade da GCS é fazer com que o desenvolvimento de software seja facilitado, mantendo e estabelecendo a integridade dos artefatos do projeto por todo seu o tempo de vida. A integridade acontece através do monitoramento, da administração e do controle de todas as mudanças relacionadas ao produto de software (ENGHOLM JR., 2011).

Para Sommerville (2012), a GCS envolve quatro atividades interligadas:

a) **Gerenciamento de Mudanças:** Acompanhar todas as solicitações feitas por clientes e usuários do produto de software, realizar as mudanças, definir os custos, impactos e quando elas devem ser implementadas.

b) **Gerenciamento de Versões:** Todas modificações realizadas por diferentes desenvolvedores devem ser acompanhadas e controladas de maneira que uma não interfira nas outras.

c) **Construção do Sistema:** Elaboração do código-fonte, dos componentes, das bibliotecas e dos dados que darão origem ao sistema.

d) **Gerenciamento de Releases:** Controle de todas as versões do sistema que foram disponibilizadas para uso do cliente.

A GCS está relacionada às características e às funcionalidades físicas do produto, documentação seja de software ou hardware usadas em qualquer parte do projeto além de banco de dados, framework, versão de aplicativos entre outros. O gerenciamento de configuração é necessário para que seja possível controlar os itens de configuração (IC) do projeto e seu versionamento para que se possa controlar todas as mudanças que venham a ser realizadas nessas características (ENGHOLM JR., 2011).

2.2.1 Gerenciamento de mudanças

Para Bach (1998), o controle de mudanças é algo vital dentro de um projeto de software. Modificações dentro de um software são comuns, no entanto pequenas mudanças podem causar enormes problemas em um sistema. Mas elas também podem corrigir falhas grotescas ou disponibilizar recursos importantes há muito tempo aguardados.

Um equilíbrio no gerenciamento de mudanças é necessário, pois um controle extremamente pesado pode desencorajar os desenvolvedores em seu trabalho criativo. Por outro lado, a não existência de nenhum tipo de controle, pode acarretar onde mudanças de um único desenvolvedor poderiam afundar o projeto (BACH, 1998).

As modificações de software dentro de empresas são acontecimentos comuns, os requisitos e os problemas a serem resolvidos se alteram durante o ciclo de vida do sistema. E para que as atualizações do sistema sejam implementadas de forma controlada é de vital importância a existência de um conjunto de processos que gerenciarão as mudanças (SOMMERVILLE, 2012).

> O gerenciamento de mudanças destina-se a garantir que a evolução do sistema seja um processo gerenciado e que seja dada prioridade às mudanças mais urgentes e efetivas (SOMMERVILLE, 2012, p. 478).

O gerenciamento de mudanças não tem como finalidade prevenir as mudanças, mas na realidade, identificar e gerenciar toda e qualquer possível mudança. São analisados todos os impactos que as mudanças possam trazer para o projeto que podem afetar a qualidade, escopo, cronograma ou orçamento. O gerenciamento de mudanças deve se assegurar que antes de implementadas as mudanças devem ser aprovadas por todos os envolvidos (ENGHOLM JR., 2011).

O gerenciamento de mudanças segue alguns princípios (AIELLO; SACHS, 2010):

a) Mudanças devem sempre ser planejadas e não deixadas para serem executadas às pressas.

b) Mudanças devem ser compreendidas inclusive os pontos negativos que elas podem trazer.

c) Todas as mudanças devem ser aprovadas seguindo a hierarquia pré-definida.

d) Procedimentos para mudanças de emergência devem ser estabelecidos para cobrir incidentes imprevistos.

e) O controle de mudanças deve se assegurar de que todos os processos de Gerência de Configuração estão sendo aplicados.

As mudanças podem ter inúmeras origens e um projeto sempre deve estar preparado para absorvê-las. Por exemplo, um projeto pode ter mudanças de requisitos para que seja entregue um produto ainda melhor para o cliente. Todas as mudanças trazem consigo riscos ao projeto, por isso durante a análise da solicitação de mudanças, é necessário que se estude os riscos envolvidos em cada pedido (ENGHOLM JR., 2011).

A maioria das empresas começa com a gerência de mudanças com um *change control board* (CCB) ou quadro de controle de mudanças, em tradução livre. Nele todas as mudanças são revisadas, analisadas e aprovadas antes de irem para a produção (AIELLO; SACHS, 2010).

> Um ponto que deve estar na mente de todos os envolvidos em projetos é que não é simplesmente pelo fato de uma mudança ter sido solicitada que ela deve ser implementada (ENGHOLM JR., 2011 p. 248).

Para Aiello e Sachs (2010), a melhor maneira de iniciar um CCB é começar de maneira simples e ao longo dos ciclos de desenvolvimento novos controles devem ser adicionados, conforme as necessidades da empresa.

O processo de gerência de mudanças começa quando um usuário ou cliente solicita uma modificação pertinente para o sistema. Essa solicitação é colocada em um formulário eletrônico de solicitação de mudanças, comumente conhecidos como sistemas de *tickets*, A partir do preenchimento da solicitação e aprovação, o desenvolvedor pode registrar como e quais foram as alterações implementadas (SOMERVILLE, 2012).

A partir do momento que as mudanças são implementadas pela equipe de desenvolvimento deve-se manter um controle contendo todas as mudanças realizadas. E para que exista um acompanhamento adequado de todas as diferentes versões dos componentes do projeto existe a gerência de versões.

2.2.2 Gerenciamento de versões

Conforme um projeto avança novas funções são adicionadas e problemas são corrigidos, essas alterações criam diferentes versões do mesmo programa. Todas essas versões devem ser armazenadas em um local que permita uma gerência eficaz e viabilize aos desenvolvedores alterar o projeto para qualquer versão armazenada. Esse local é conhecido como repositório de versões (PRESSMAN, 2011).

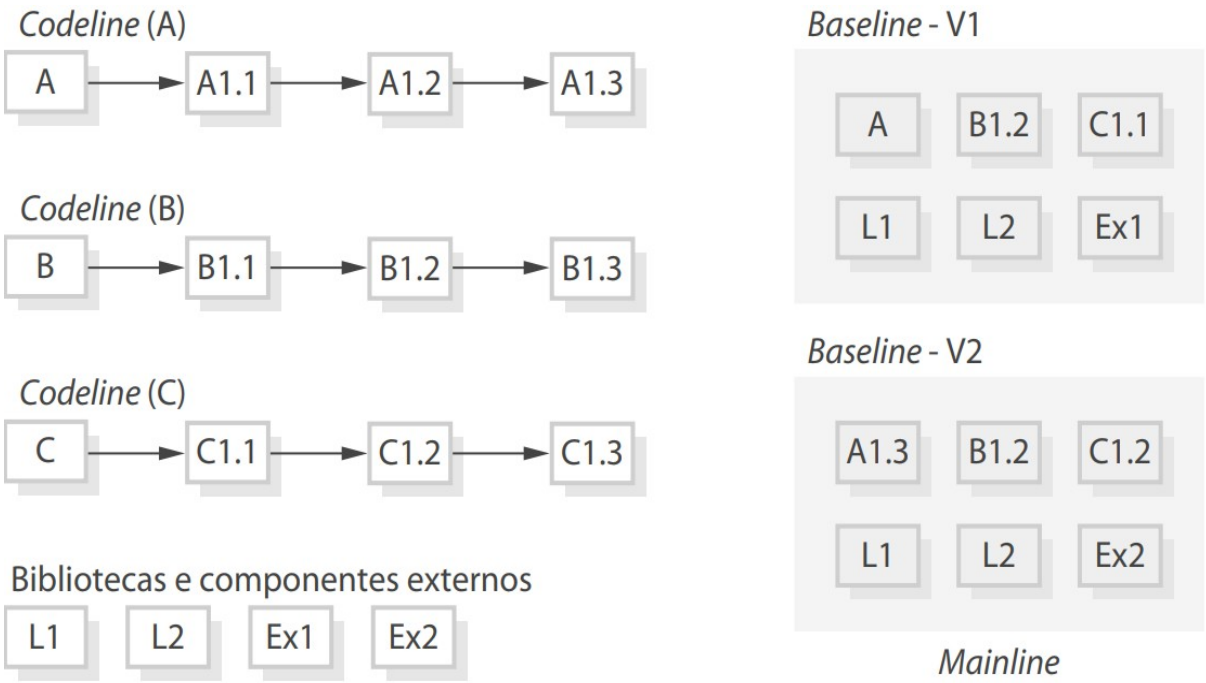
Para Sommerville (2012), o gerenciamento de versões nada mais é que o acompanhamento de versões diferentes de todos os artefatos de software. O gerenciamento de versões pode ser visto como o processo que controla *codelines* e *baselines*.

> Hoje o repositório é uma “coisa” um banco de dados que age como o centro de acumulação e de armazenagem de informações de engenharia de software. O papel da pessoa (o engenheiro de software) é interagir com o repositório usando ferramentas integradas com ele (PRESSMAN, 2011 p. 519).

Baselines são todas as versões de componentes que formam um sistema. *Codelines* são versões de um componente de software e itens de configuração que dão origem para esse componente. A importância das baselines está em poder restaurar um sistema inteiro em uma determinada versão (SOMMERVILLE, 2012).

Na Figura 2, é possível de forma gráfica compreender a diferença entre *codelines* e *baselines*. Enquanto o primeiro é referente a apenas um componente o segundo é relacionado a um conjunto de componentes.

Figura 2 - Diferença entre *baselines* e *codelines*



Fonte: Sommerville(2012).

Um *version control system* (VCS), ou sistema de controle de versão, é um sistema que rastreia versões incrementais, ou revisões, de arquivos e de diretórios ao longo do tempo. O que torna um VCS útil é o fato de que ele permite explorar as mudanças que resultaram em cada uma dessas versões facilitando o *recall* arbitrário do mesmo (PILATO et al, 2013).

Normalmente VCS fornecem uma variedade de recursos (SOMMERVILLE, 2012):

a) **Identificação da versão:** Quando gerenciadas, as versões ganham identificadores, usualmente esses identificadores são baseados no nome do IC.

b) **Gerenciamento de armazenamento:** Ao invés de armazenar arquivos completos de todas as versões, os VCS mantêm apenas uma lista com as diferenças, conhecidos como deltas. Dessa maneira, o armazenamento de dados se torna muito menor.

c) **Registro de histórico de mudanças:** São registradas listas com todas as mudanças feitas no código de um sistema ou em seus componentes.

d) **Desenvolvimento independente:** O VCS garante que mais de um desenvolvedor pode trabalhar no mesmo componente ao mesmo tempo, sem que os trabalhos de um interfira no trabalho dos outros.

e) **Suporte a projeto:** Um VCS apoia no desenvolvimento de inúmeros projetos que compartilham os mesmo componentes.

No últimos anos, diversas propostas foram feitas para automatizar o controle de versão. Estão disponíveis no mercado muitas ferramentas, inclusive de maneira *open source*. Todas elas trabalham de maneira semelhante, contudo as diferenças são encontradas nas maneiras do processo de construção e na sofisticação dos atributos que são usados (PRESSMAN, 2011).

2.3 Ferramentas de controle de versão e mudança

Para Humble e Farley (2013), o gerenciamento de configuração refere-se ao processo pelo qual todos os artefatos relevantes para um projeto e as relações entre eles são armazenados, recuperados, identificados e modificados. A estratégia de gerenciamento de configuração de uma empresa determina como todas as mudanças que acontecerem em um projeto serão controladas, ela registrará a evolução dos sistemas e aplicações.

> Configuration management refers to the process by which all artifacts relevant to your project, and the relationships between them, are stored, retrieved, uniquely identified, and modified (HUMBLE; FARLEY, 2013, p 31).

Embora os sistemas de controle de versão sejam as ferramentas mais óbvias no gerenciamento de configuração, a decisão de usar um é apenas o primeiro passo no desenvolvimento de uma estratégia de gerenciamento de configuração. Cada equipe deve usar um, não importa o tamanho da equipe e do projeto (HUMBLE; FARLEY, 2013).

Duas das mais famosas ferramentas de controle de versão são:

a) Git

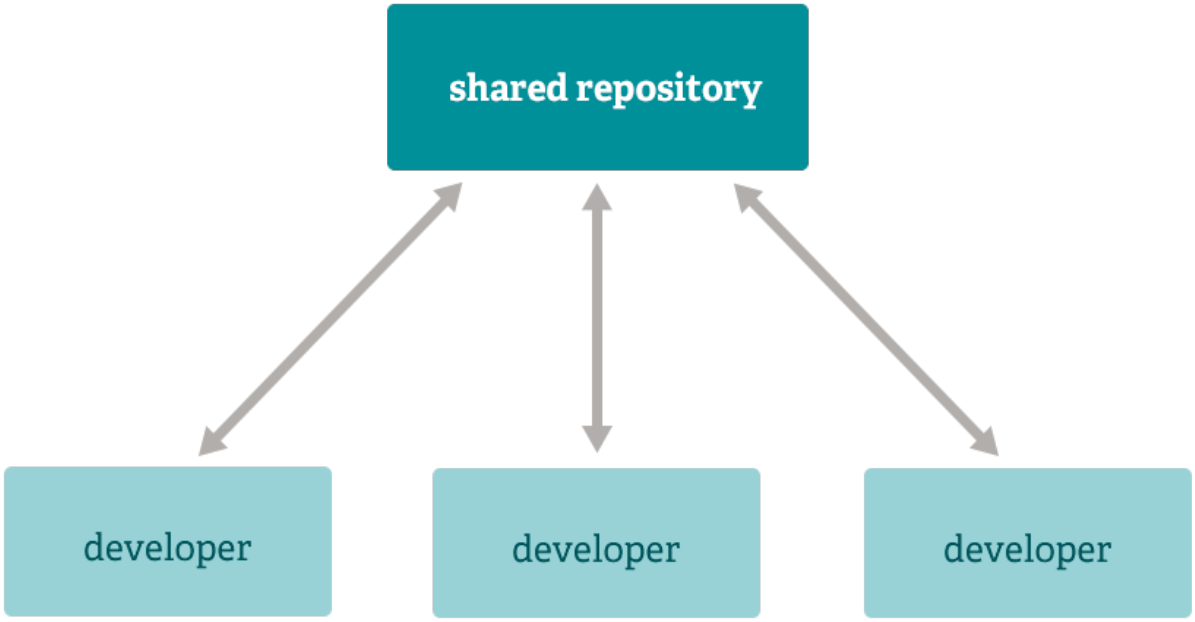
b) Subversion

O Git é um sistema de controle de versões de arquivos distribuído, que pode funcionar local, no próprio computador, ou em servidores. Ele possui um sistema com múltiplos backups, isso significa que existem diversas cópias do trabalho. Basicamente, todos os

usuários tem um backup completo da aplicação, pois para trabalhar eles clonam o projeto. Desta maneira cada desenvolvedor tem o seu projeto e suas mudanças não interferem no trabalho dos outros (GIT, 2017).

O Subversion, ao contrário do Git, é um sistema de controle de versões centralizado. Os desenvolvedores trabalham diretamente no mesmo repositório, para maior controle é necessário que esse repositório seja multiplicado. A Figura 3, demonstra o funcionamento de um repositório centralizado (SUBVERSION, 2017).

Figura 3 - Repositório centralizado



Fonte: Git (2017).

Segundo Sommerville (2012), todas as mudanças do software devem ser submetidas a um sistema de gerenciamento de mudanças. Pois é necessário uma análise para identificar os reais benefícios da implementação de alguma mudança, e se eles justificam os custos que a mudança causará.

Existem diversos benefícios de usar um software de gerência de mudanças, desse modo todas as propostas de alteração são tratadas de forma consistente e todas as modificações feitas em arquivos são realizadas de forma controlada e são registradas. De maneira que, todas as propostas e modificações são facilmente encontradas. Sendo que diversos indicadores podem ser extraídos a fim de melhorar a qualidade do desenvolvimento (SOMMERVILLE, 2012).

Duas ferramentas muito populares no controle de mudança são:

a) Redmine

b) Jira

O Redmine é um sistema de gerenciamento de projetos com ótimas ferramentas de controle de mudança. É possível criar tarefas, vincular arquivos ter um percentual de conclusão. O Redmine é essencialmente uma ferramenta de rastreamento para erros, atualizações de recursos ou qualquer tarefa que precise ser rastreada (REDMINE, 2017).

Diversas ferramentas que fazem o Redmine um bom software de gerenciamento de mudanças (REDMINE, 2017):

a) Cria gráficos de Gantt, para controle de prazos de tarefas.

b) Calendário de tarefas.

c) Arquivos podem ser adicionados individualmente por tarefa.

d) Categorias de tarefas customizáveis.

O Jira é um software de rastreamento de problemas, desenvolvido pela empresa Atlassian. Ele fornece rastreamento problemas, bugs e possui funções de gerenciamento de projetos. O Jira é oferecido em três pacotes (JIRA, 2017):

a) Jira software: gerenciamento de projetos ágeis, gerenciamento de mudança.

b) Jira core: gerenciador de projetos.

c) Jira service desk: gerenciamento de suporte em TI.

2.4 DevOps

O termo DevOps foi popularizado no ano de 2008 em Toronto durante um congresso sobre desenvolvimento ágil. Durante sua apresentação Debois (2008) apresentou ao mundo um novo conceito que integraria a equipe de desenvolvimento e a equipe de operações.

A palavra DevOps é a abreviação de desenvolvimento e operações, basicamente é uma metodologia de trabalho baseada em princípios ágeis. Trabalhar da maneira DevOps exige que todos os envolvidos no projeto os *stakeholders*: gestores, parceiros, fornecedores e as equipe de arquitetura, design, desenvolvimento, qualidade, operações e segurança colaborem para entregar o software de maneira contínua. Excluir qualquer *stakeholder*, interno ou externo, leva a uma implementação incompleta de DevOps (SHARMA, 2014).

Entregar software de maneira contínua significa aproveitar melhores oportunidades do mercado, além de reduzir drasticamente o tempo de retorno para os clientes (SHARMA, 2014).

> After a few years of talking about “What is DevOps?”, a general con sensus has started to form around DevOps being a cultural movement combined with a number of software development practices that en able rapid development (WALLS, 2013 p. 519).

Para Walls (2013), a implementação de novos processos e o uso de novas ferramentas para DevOps são simples quando comparadas com a mudança de mentalidade, que junto com as novas práticas, devem ser utilizadas nas empresas que procuram ser orientadas a DevOps.

DevOps, de várias maneiras, é um conceito pai que se refere a todas as coisas que melhoram a interação entre de desenvolvimento e operações. O conceito DevOps foi a resposta à crescente conscientização do problema na comunicação entre desenvolvimento e operações. Estes desentendimentos se manifestam como conflitos e ineficiência dentro das empresas (EDWARDS, 2010).

Para Sharma (2014), as aplicações hoje são compostas por diversas tecnologias: banco de dados, dispositivos de usuário final, servidores, nuvem, etc. E apenas uma abordagem DevOps será bem sucedida ao lidar com todas essas complexidades.

Shafer (2009) denomina o desentendimento entre a equipe de operações e a equipe de desenvolvimento como *“Wall of Confusion”*, muro da confusão em tradução livre. Este “muro” é criado por todos os conflitos, motivações, processos e ferramentas das equipes conflitantes. É exatamente esse “muro” que o DevOps quer derrubar.

Figura 4 - Muro da confusão

Fonte: Adaptado em 04/10/2017 de Stott , 2016.

Desenvolvedores normalmente tem a mentalidade de que eles são pagos para realizar mudanças e que o negócio depende do produto responder às mudanças de mercado, por isso eles são incentivados a modificar o software tanto quanto necessário. A equipe de operações toma cuidados com mudanças, para eles o negócio depende da capacidade de manter os serviços online e entregando o produto (EDWARDS, 2010).

A Figura 4, representa o ciclo de desenvolvimento tradicional, nele o time de desenvolvimento “arremessa” o software sobre o muro da confusão. O time de operações recebe os artefatos de software e começam a preparar a implantação, para isso é necessário editar configurações para que elas afetem o ambiente de produção e não de testes nesse momento. Na melhor das hipóteses eles estão duplicando o trabalho que já foi realizado, e na pior estão prestes a apresentar ou descobrir novos erros (STOTT, 2016).

Para Edwards (2010), DevOps contribui diretamente para possibilitar duas poderosas estratégias de qualidade de negócio: agilidade de negócio e alinhamento de TI. Uma definição

simples de agilidade de negócio seria a habilidade, que uma organização tem, de rapidamente se adaptar às mudanças do ambiente e de mercado de forma produtiva e econômica. Alinhamento de TI, de maneira simples, quer dizer um estado desejado no qual uma organização pode usar a TI de maneira efetiva para atingir todos os objetivos de negócio.

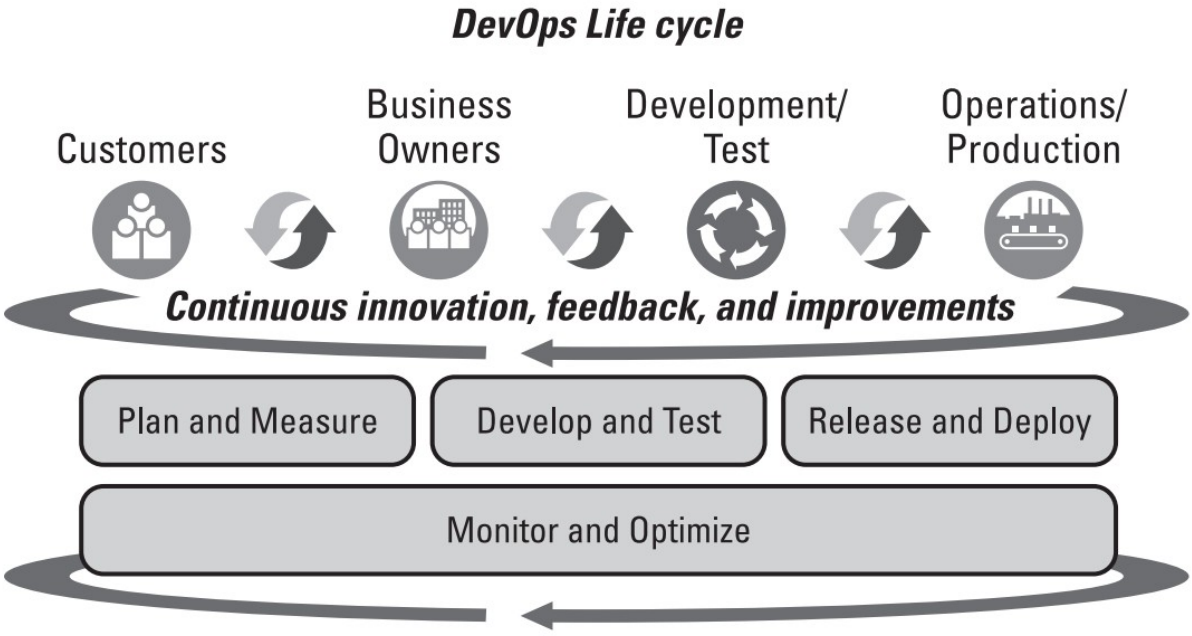
O relatório anual da Puppet, empresa que oferece software de gerenciamento de configuração, contendo mais de 27 mil respostas de desenvolvedores mostra algumas vantagens de aplicar a metodologia DevOps, empresas com alta performance de DevOps obtém (FORSGREN, 2017):

a) Frequência 200 vezes maior nas implementações.

b) Tempo de recuperação 24 vezes mais rápido.

c) Chance de falha 3 vezes menor.

Figura 5 - Ciclo de vida DevOps



Fonte: Adaptado em 05/10/2017 de Sharma, 2014.

Para Sharma (2014), as capacidades que compõem o DevOps são amplas e devem abranger o completo ciclo de vida do fornecimento de software, como visto na Figura 5. O caminho de adoção do DevOps deve seguir os seguintes princípios:

a) **Planejamento e dimensionamento:** Consiste em se concentrar nas linhas de negócio e seus processos. Para atingir esses objetivos, as organizações medem

o progresso, descobrem o que os clientes realmente querem e, em seguida, mudam de direção, atualizando seus planos de negócios em conformidade, permitindo que eles tomem decisões de *trade-off* contínuas em um ambiente com recursos limitados.

b) **Desenvolvimento e testes:** Para adoção deste processo duas práticas são necessárias: desenvolvimento colaborativo e teste contínuo. A ideia do desenvolvimento colaborativo é que os desenvolvedores integrem regularmente seus trabalhos com toda a equipe, essa integração ajuda a descobrir precocemente riscos tanto ao projeto, quanto ao cronograma. Teste contínuo significa testar mais cedo e continuamente ao longo do ciclo de vida, o que resulta em custos reduzidos.

c) **Disponibilizar e implantar (*release* e *deploy*):** A integração contínua leva o processo de automação da implantação do software para testes, de sistema e produção ambientes. As empresas que adotam DevOps geralmente usam o mesmo processo automatizado em todos os ambientes para melhorar a eficiência e reduzir o risco introduzido por processos inconsistentes.

d) **Monitoramento e Otimização:** Incluem duas práticas que permitem às empresas monitorar como está a performance de aplicativos lançados e como está o retorno, *feedback,* dos clientes. Esses dados permitem que as empresas reajam de forma ágil e alterem seus planos comerciais, conforme necessário. Isso acontece através do monitoramento contínuo e contínuo aperfeiçoamento.

2.4.1 Integração Contínua

O movimento ágil foi responsável pela popularização da integração contínua. O objetivo dessa metodologia é que os desenvolvedores regularmente integrem seus trabalhos com os demais desenvolvedores, e que depois, testes sejam realizados sobre esta integração. A integração regular dos trabalhos leva à descoberta precoce e exposição de riscos e de problemas que o software possa apresentar (SHARMA, 2014).

> Integração contínua é uma das práticas originais de extreme programing que encoraja desenvolvedores a integrar seus trabalho frequentemente para que possíveis problemas sejam detectados e corrigidos rapidamente (SATO, 2014, p. 139).

Para Humble e Farley (2013), a implementação de integração contínua depende de três itens: controle de versão, sistema automático de *build* e um acordo entre a equipe.

Em alguns projetos o controle de versão não é utilizado pois seus desenvolvedores acham que ele não é grande o suficiente para ter esta necessidade. Humble e Farley (2013) acreditam não existir projeto que deva ser desenvolvido sem controle de versão.

Um sistema automático de *build* ajuda a diminuir as chances de acontecerem erros humanos, além de diminuir o tempo de *feedback*. O *build* de um projeto envolve todas os processos necessários para conseguir executá-lo. Por exemplo: cálculos das métricas de qualidade, empacotamento, vinculação com bibliotecas externas, respostas das dependências compilação e *download* (SATO, 2014).

A integração contínua é uma prática, não uma ferramenta. Exige um certo grau de comprometimento e disciplina da equipe de desenvolvimento. É necessário que toda a equipe concorde que a tarefa de maior prioridade no projeto é corrigir qualquer alteração que faça o software parar de funcionar. Se a equipe não adotar a disciplina necessária, a integração contínua não levará a melhoria de software desejada (HUMBLE; FARLEY, 2013).

Se as pessoas não adotam a disciplina necessária para que ela funcione, suas tentativas de integração contínua não levarão à melhoria da qualidade que desejada.

A Entrega Contínua possui vários objetivos, por exemplo (SHARMA, 2014):

a) Permitir teste contínuo e verificação de código.

b) Validar que o código produzido e integrado executa conforme projetado.

c) O sistema em desenvolvimento deve ser testado continuamente.

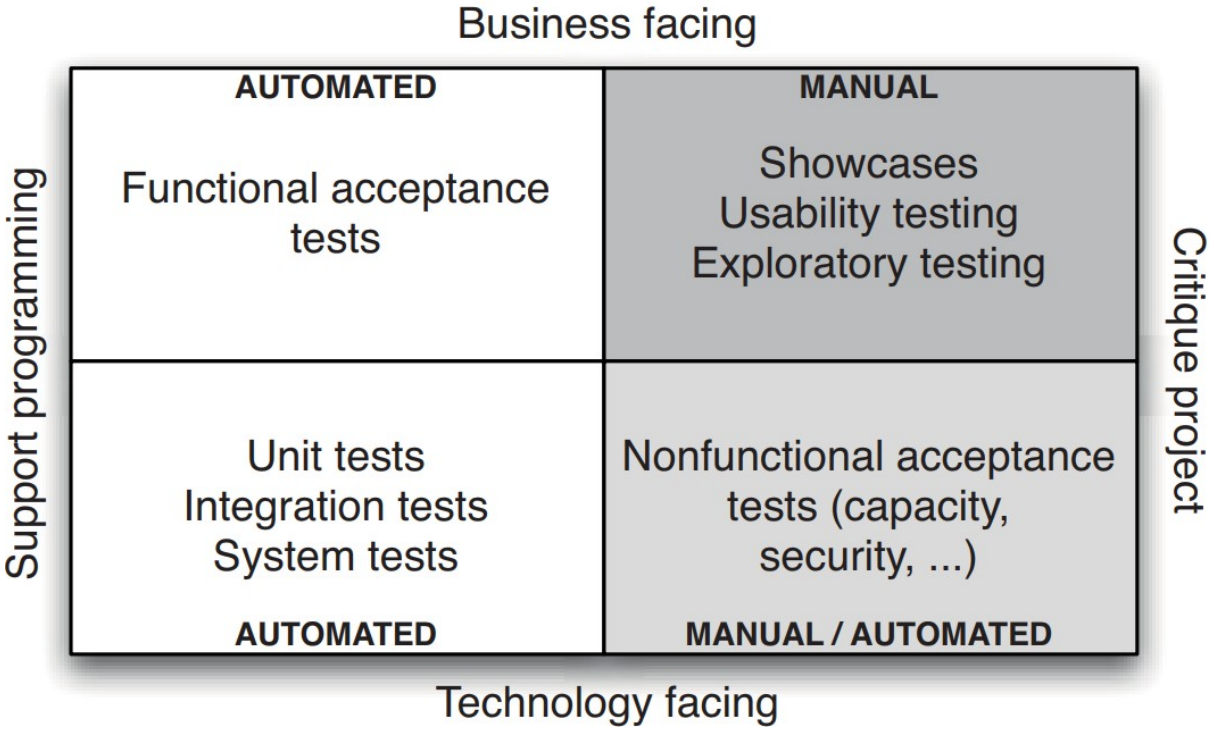
Para Sharma (2014), teste contínuo significa testar mais cedo e continuamente, durante todo o ciclo de vida do projeto. Isso resulta em custos reduzidos.

Ao disponibilizar um sistema algumas coisas podem dar errado, o software pode conter bugs ou alguma funcionalidade pode ter sido implementada de maneira errada. Por isso

é essencial para a cultura DevOps que testes automatizados sejam incluídos no processo de *build* (SATO, 2014).

Existem muitos tipos de testes que podem ser automatizados, Brian Marick surgiu com a proposta representada pela Figura 6, que é amplamente utilizada para demonstrar os vários tipos de testes devem ser executados para que se entregue um produto de qualidade (HUMBLE; FARLEY, 2013).

Figura 6 - Tipo de testes



Fonte: Adaptado em 10/10/2017 de Humble; Farey, 2013

No lado esquerdo da Figura 6, ficam os testes que dão suporte à programação, são os testes que os desenvolvedores executam com muita frequência para obter um *feedback* da aplicação sobre as mudanças introduzidas em cada *commit*. No lado direito, ficam os testes que avaliam o produto com um olhar de usuário, avaliando sua qualidade externa e tentando identificar problemas durante a utilização (SATO, 2014).

Na parte inferior estão os testes descritos tecnicamente do ponto de vista dos desenvolvedores, conhecidos como testes voltados à tecnologia. Na parte superior estão os testes voltados ao negócio. Esses, por sua vez, são descritos usando uma linguagem que faz sentido para especialistas da área de negócio em questão (SATO, 2014).

Sato (2014) explica que existem diversos tipos de testes que podem ser automatizados, cada um com seus objetivos: usabilidade, exploratórios, desempenho, etc. Embora não exista uma classificação ou terminologia largamente aceita na indústria de software os testes mais comuns são:

a) **Testes de unidade:** Unidades são a menor parte que pode ser testada de um programa, uma unidade pode ser uma função individual. O teste de unidade valida dados através de entrada e saídas aplicadas por um desenvolvedor. Esses testes normalmente executam rapidamente, podendo ser executados frequentemente.

b) **Testes de integração ou testes de componente:** Testam uma parte da aplicação ou como ela interage com suas dependências.

c) **Testes funcionais ou testes de aceitação:** Normalmente, esse tipo de teste é realizado do ponto de vista de um usuário e tem como objetivo validar o sistema como um todo. Para ser realizado, a maior parte do sistema deve estar em execução.

Naturalmente a integração contínua leva à prática de Entrega Contínua: o processo de automatização da implantação do software no teste, teste de sistema, encenação e ambientes de produção. As empresas que têm a cultura DevOps geralmente usam o mesmo processo automatizado em todos ambientes, buscando melhorar a eficiência e reduzir riscos de incluir processos inconsistentes.

2.5 Entrega Contínua

Para Humble e Farley (2013), o lançamento de uma nova versão de um sistema normalmente é um dia tenso. Isso acontece porque na grande maioria dos projetos existe um grande risco associado ao processo, que faz a entrega ser algo assustador.

Grande parte das aplicações modernas são complexas independente do tamanho, elas contém inúmeras dependências. Muitas empresas entregam versões de software de uma maneira manual. É necessário realizar julgamentos a cada etapa do processo para se assegurar que a nova versão está funcional, o que torna o processo mais sujeito a erro humano. É

possível identificar uma série de sinais nesse antipadrão de entrega (HUMBLE; FARLEY, 2013):

a) Documentação extensa sobre os processos e como eles podem falhar.

b) Detalhamento de testes manuais para confirmar se o sistema está funcionando corretamente.

c) Desenvolvedores com frequência precisam explicar o motivo de algo não funcionar no dia da entrega.

d) Ajustes no processo de entrega no dia em que uma versão está sendo lançada.

e) Ambientes diferentes para a mesma aplicação.

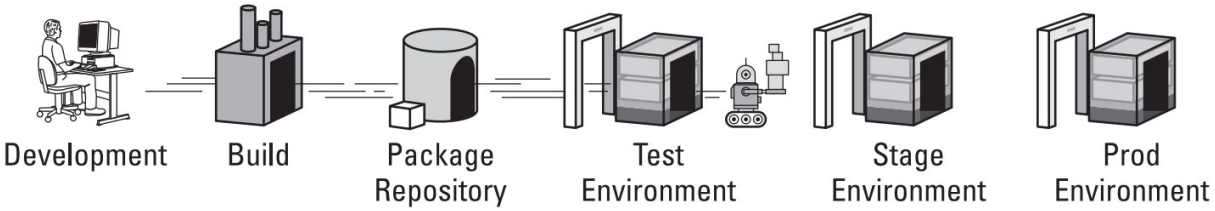
Ao invés de seguir diversas etapas manuais, a implantação deve mirar à automação completa. Apenas duas tarefas devem ser manuais durante o lançamento de uma nova versão em ambientes de produção, testes ou desenvolvimento são elas: escolher a versão e ambiente, e iniciar o processo. A instalação, o empacotamento e a entrega devem ser um único processo automatizado (HUMBLE; FARLEY, 2013).

2.5.1 Pipeline de entrega

A Entrega Contínua eleva o conceito de integração contínua para outro patamar, essa prática possibilita a criação de um *pipeline* de entrega, uma linha de produção em tradução livre. Esse *pipeline* facilita a implantação contínua de software para controle de qualidade e a produção de forma eficiente e automatizada (SHARMA, 2014).

Um *pipeline* de entrega é composto pelos estágios pelos quais uma aplicação passa desde o desenvolvimento até a entrega. A Figura 7 demonstra os estágios típicos de um *pipeline* de entrega, os estágios podem variar conforme as necessidades de cada empresa (SHARMA, 2014).

Figura 7 - Pipeline de entrega



Fonte: Adaptado em 10/10/2017 de Sharma, 2014

Para Humble e Farley (2013), o *pipeline* de entrega é um nível abstrato de implantação, é uma manifestação automatizada do seu processo para fornecer o software do controle de versão nas mãos de seus usuários. Todas as mudanças no software passam por um processo complexo no caminho para ser lançado.

2.6 Provisionamento

Todas as aplicações para funcionar precisam de alguma dependência, ou algum tipo de serviço que executa em *background*, sejam servidores, linguagens de programação, etc. Para ter o software rodando em um novo servidor é necessário que sejam instalados todos as dependências, e esse processo muitas vezes não é documentado (TAVARES, 2013).

O tipo de servidor provisionado manualmente é chamado por Fowler (2012) de *snowflake server*, servidores floco de neve em tradução livre. Eles geralmente não possuem nenhuma forma de backup e são difíceis de serem reproduzidas.

Uma maneira elegante de evitar os *snowflakes* é manter toda a configuração operacional do servidor em algum tipo de receita automatizada. O ponto de usar uma receita não é apenas para facilmente reconstruir o servidor, mas também para entender facilmente sua configuração e assim modificá-la com mais facilidade (FOWLER, 2012).

Provisionamento é um processo sistemático, que através de um conjunto de passos, receita, a serem aplicados em uma imagem inicial de um sistema operacional, para que se tenha todas as dependências de um software instaladas de maneira correta e automática. Duas ferramentas que se tornaram referência no provisionamento são Puppet and Vagrant (TAVARES, 2013).

Provisionamento é um termo comumente usado em empresas de TI para se referir a todas as etapas que permeiam a configuração de um novo recurso, seja ele um servidor, um celular ou um computador. Equipes de operações consideram a equipe de desenvolvimento

como usuários finais, e o provisionamento de um servidor é considerado como concluído quando ele estiver disponível na rede (SATO, 2014).

2.6.1 Ferramentas para provisionamento

Puppet e Vagrant se tornaram as ferramentas mais populares na área de provisionamento.

O Puppet é uma ferramenta que auxilia no gerenciamento de configuração *open source*. O principio do Puppet é que se tenha a configuração de todas os servidores físicos e virtuais em um local centralizado. E que essa configuração seja espalhada para diversos nós de uma rede. As principais tarefas desempenhadas pelo Puppet são: Gerência de Configuração, garantir auditoria e automação na instalação de pacotes (PUPPET, 2017).

O Vagrant é uma ferramenta para criar e gerenciar servidores virtuais em um único fluxo de trabalho. Com um fluxo de trabalho fácil de usar e foco total na automação, o Vagrant reduz o tempo de configuração de ambientes desenvolvimento, aumenta a paridade de produção e elimina diferenças de ambientes de execução do software (VAGRANT, 2017).

3 METODOLOGIA

Neste trabalho, foi necessário adotar uma metodologia que regesse seu andamento, cujo método científico utilizado será apresentado neste capítulo.

Fonseca (2002) explica que o significado de metodologia é o estudo da organização. São os caminhos que devem ser percorridos, para que se realize uma pesquisa, estudo ou fazer ciência.

Para Minayo (2007), existem diferenças entre metodologia e métodos que devem ser destacadas. Enquanto a metodologia se foca no caminho escolhido para chegar aos objetivos da pesquisa, os métodos são o conjunto de técnicas e procedimentos que serão usados no estudo.

> Todas as ciências caracterizam-se pela utilização de métodos científicos; em contrapartida, nem todos os ramos de estudo que empregam estes métodos são ciências. Podemos concluir que a utilização de métodos científicos não é da alçada exclusiva da ciência, mas não há ciência sem o emprego de métodos científicos (MARCONI; LAKATOS, 2003. p 20).

Conforme os objetivos do presente trabalho, foi realizada a construção de uma ferramenta que auxilia na distribuição de software em uma grande quantidade de servidores, alocados em diferentes plataformas. A ferramenta também oferece mecanismos que auxiliam no processo de provisionamento, manutenção e qualidade do código-fonte. Para isso, temas como manutenção de código-fonte e provisionamento de servidores na nuvem foram estudados. De maneira que, como definido por Gil (2010), este trabalho se caracteriza como pesquisa exploratória.

Pesquisas exploratórias visam proporcionar uma maior familiaridade com o problema em questão, para torná-lo explícito ou criar hipóteses. Portanto, é necessário um levantamento bibliográfico, entrevistas com especialistas e análise de exemplos (GIL, 2010).

Desse modo, o presente trabalho realizou uma pesquisa que quantificou os dados extraídos durante a validação da ferramenta. Além disso, a ferramenta desenvolvida foi utilizada como uma abordagem de pesquisa qualitativa para que o seu próprio uso seja avaliado, através de especialistas da área da Engenharia de Software. Portanto a pesquisa do presente trabalho será qualitativa, tanto quanto, quantitativa.

> A pesquisa qualitativa é particularmente útil como uma ferramenta para determinar o que é importante para os clientes e porque é importante. Esse tipo de pesquisa fornece um processo a partir do qual questões-chave são identificadas e perguntas são formuladas, descobrindo o que importa para os clientes e porquê (MORESI, 2003. p 69).

Segundo Gerhardt e Silveira (2009), os métodos qualitativos buscam explicar determinados acontecimentos, sem a necessidade de quantificação, pois os dados obtidos são não-métricos e se valem de diferentes abordagens.

> A pesquisa quantitativa se centra na objetividade. Influenciada pelo positivismo, considera que a realidade só pode ser compreendida com base na análise de dados brutos, recolhidos com o auxílio de instrumentos padronizados e neutros. A pesquisa quantitativa recorre à linguagem matemática para descrever as causas de um fenômeno, as relações entre variáveis, etc. A utilização conjunta da pesquisa qualitativa e quantitativa permite recolher mais informações do que se poderia conseguir isoladamente (FONSECA, 2002. p 20).

Conforme Fonseca (2002) explica, pesquisas quantitativas tem resultados que podem ser quantificados. Normalmente tenta-se atingir uma grande amostra de uma população, a fim de que, possa ser montado um retrato real da população-alvo da pesquisa.

A ferramenta é proposta, desenvolvida, testada e utilizada em um ambiente controlado, contendo dados fictícios. Dessa maneira, seu uso foi feito em ambiente de laboratório. Santos (1999) afirma que o ambiente de laboratório é necessário pois muitas vezes no campo, fatos e fenômenos que acontecem escapam ao padrão desejado de observação. Sendo assim, o laboratório é o ambiente onde os testes serão reproduzidos de forma artificial e controlada, permitindo captação e análise adequadas.

No momento em que a ferramenta teve seu desenvolvimento concluído, foi submetida a avaliação de especialistas da área de Engenharia de Software. A avaliação foi realizada através de um questionário quali-quantitativo, de maneira que permitiu aos especialistas avaliarem as funcionalidades e os recursos, e se os objetivos da ferramenta foram cumpridos.

Wainer (2007) define questionários como uma maneira rápida e simples para que opiniões, objetivos, preferências, sejam avaliados. No entanto, por serem simples, se concebidos de maneira leviana podem trazer consequências negativas.

A parte quantitativa do questionário de avaliação da ferramenta, usou a escala de Likert. Para Wainer (2007) é uma prática comum usar a escala de Likert em questionários quantitativos, as questões devem ser fraseadas de maneira afirmativa, sendo que, o respondente escolhe uma alternativa.

4 ESPECIFICAÇÃO DO PROJETO

Neste capítulo será apresentada a visão geral da plataforma desenvolvida, bem como a definição de seus requisitos funcionais e não funcionais, telas da interface, tecnologias que foram utilizadas, artefatos, bem como arquitetura do software.

4.1 Visão geral do projeto

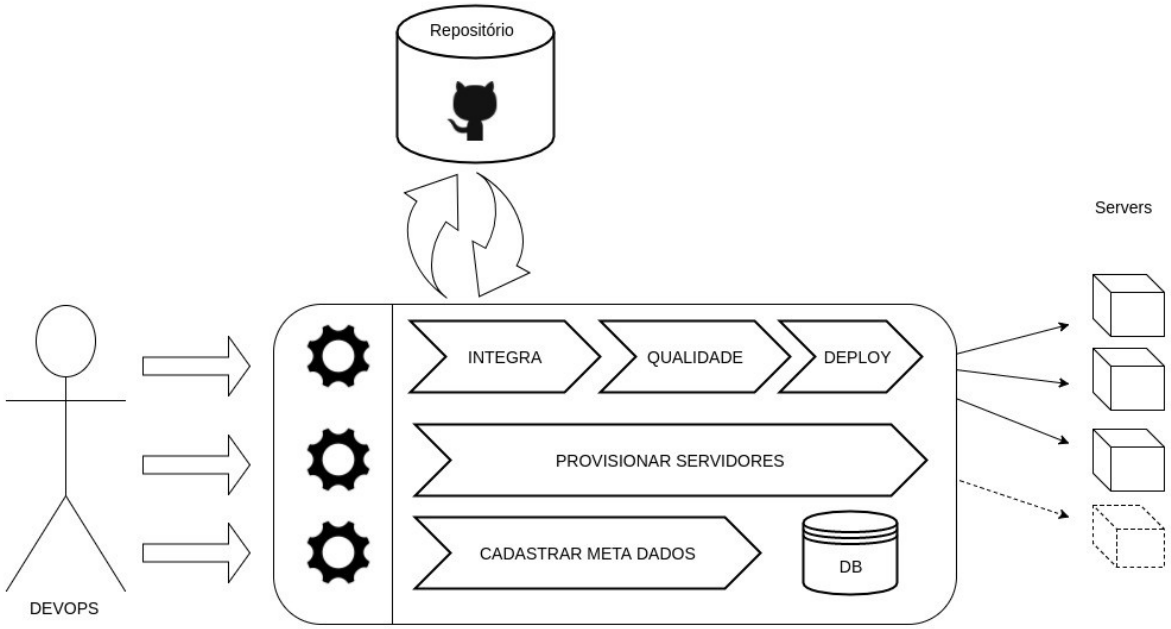
O presente trabalho teve como objetivo criar uma plataforma web, que facilite o processo de provisionar servidores e realizar *deploy*.

A plataforma desenvolvida permite aos usuários cadastrar metadados, sendo que esses metadados contém informações sobre os projetos e seus respectivos servidores. Cada servidor pode conter diversos projetos e um projeto pode estar disponibilizado em diversos servidores. Dentro do cadastro dos metadados, os usuários podem criar *tags* de identificação dos servidores, isso possibilita que eles sejam filtrados separadamente por suas *tags*. Desta maneira, o usuário pode realizar um *deploy* para um grupo específico de servidores, um exemplo seria, somente para servidores no Rio Grande do Sul por exemplo.

A ferramenta desenvolvida, primeiramente conta com deploy e provisionamento exclusivos para a *provider* Vultr, além de um processo de entrega contínua. Conforme pode ser visto na Figura 8, que contempla as principais funcionalidades da ferramenta, na parte central, ficam os principais processos: *deploy*, provisionamento e cadastro de metadados.

O processo de entrega contínua é composto por uma integração com um repositório de código-fonte, representado pelo GitHub e o *deploy* para os servidores escolhidos.

Figura 8 - Visão geral da plataforma desenvolvida



Fonte: Feito pelo autor

4.2 Requisitos

Para que a ferramenta fosse considerada implementada, alguns requisitos funcionais foram desenvolvidos. Da mesma maneira, alguns requisitos não funcionais foram atendidos para que a ferramenta seja executada da maneira esperada.

4.2.1 Requisitos funcionais

A seguir serão dispostos os principais grupos de requisitos funcionais que foram implementados para a plataforma proposta:

a) Permitir cadastro de receitas para instalação em servidores, deve ser informados nome, descrição e o comando para instalação. Sendo que, apenas nome de comando são obrigatórios.

b) Permitir a edição de receitas para instalação em servidores.

c) Listar todas as receitas cadastradas na aplicação.

d) Permitir exclusão de receitas.

e) Permitir exportar em um arquivo no formato CSV os registros no cadastro de receitas.

f) Permitir apenas para administradores da aplicação o cadastro de novos *providers*, apenas o nome deverá informado.

g) Permitir apenas para administradores da aplicação a edição de *providers*.

h) Permitir apenas para administradores da aplicação a exclusão de *providers*.

i) Permitir apenas para administradores da aplicação exportar em um arquivo no formato CSV os registros do cadastro de *providers*.

j) Listar todos os *providers* cadastrados, apenas usuários administradores podem ver essa lista.

k) Permitir o cadastro de projetos hospedados em um servidor git, devem ser informados o nome, descrição do projeto e o link do repositório.

l) Permitir a edição dos registros de projetos cadastrados.

m) Permitir a exclusão de registros de projetos cadastrados.

n) Listar os registros de projetos cadastrados.

o) Exportar um arquivo no formato CSV com os registros de projetos cadastrados.

p) Permitir o cadastro de categorias para servidores, deve ser informado apenas o nome da categoria.

q) Manter um registros de logs de acesso com usuário e hora de acesso.

r) Manter um registro de erros que aconteçam na aplicação. Deve ser mantido a mensagem de erro e a linha do arquivo em que ele aconteceu.

s) Permitir cadastro de usuários com diferentes níveis de permissão.

t) Provisionar servidores em *providers* disponíveis.

u) Permitir o cadastro de servidores, devem ser informados os seguintes campos: nome, categoria, *provider*, receitas, local do servidor, plano e sistema operacional.

v) Permitir a definição de uma categoria para um servidor.

w) Permitir a escolha das receitas que devem ser instaladas ao despertar do servidor.

x) Obter uma lista de locais disponíveis da *provider* para cria um servidor.

y) Obter uma lista de planos disponíveis da *provider* para criar um servidor.

z) Obter uma lista de sistemas operacionais disponíveis da *provider* para criar um servidor.

aa) Realizar conexão via SSH para os servidores criados.

ab) Acessar os servidores web provisionados na aplicação e realizar o *deploy* do projeto desejado.

ac) Fornecer uma lista de projetos disponíveis para o *deploy*.

ad) Obter credenciais, ip e senha de um servidor provisionado na aplicação para realizar o *deploy* de um projeto desejado.

ae) Obter o IP definido para o servidor após seu provisionamento.

af) Obter todas as informações do servidor provisionado pela ferramenta que são disponibilizados pela *provider*.

ag) Criar *scripts* de inicialização para o servidor, os *scripts* devem conter as receitas selecionadas pelo usuário.

ah) Cadastrar todas as receitas que foram instaladas durante o provisionamento de um servidor.

ai) Cadastrar *deploys* realizados.

aj) Excluir toda a movimentação de um servidor ao deletá-lo. Essa movimentação incluí os projetos e receitas que foram utilizadas nele.

4.2.2 Requisitos não funcionais

Para o desenvolvimento da ferramenta os seguintes requisitos não funcionais foram ser atendidos:

a) Utilizar PHP 7.2 como linguagem de programação no *back-end*;

b) Utilizar Apache2 como servidor de páginas HTTP;

c) Ser compatível com navegadores Google Chrome (versão 61 ou superior) e Mozilla Firefox (versão 54 ou superior);

d) Utilizar PostgreSQL versão 10 como banco de dados da aplicação.

e) Desenvolver a interface *front-end* seguindo os padrões HTML5 e CSS3;

f) Utilizar JavaScript como linguagem de programação para o *front-end*;

g) Utilizar a biblioteca JQuery no desenvolvimento *do front-end;*

h) Utilizar a biblioteca Materialize para o desenvolvimento da interface do *front-end*;

i) Fornecer uma estrutura pronta para adição de novos *providers*, de maneira que, apenas a classe específica seja adicionada.

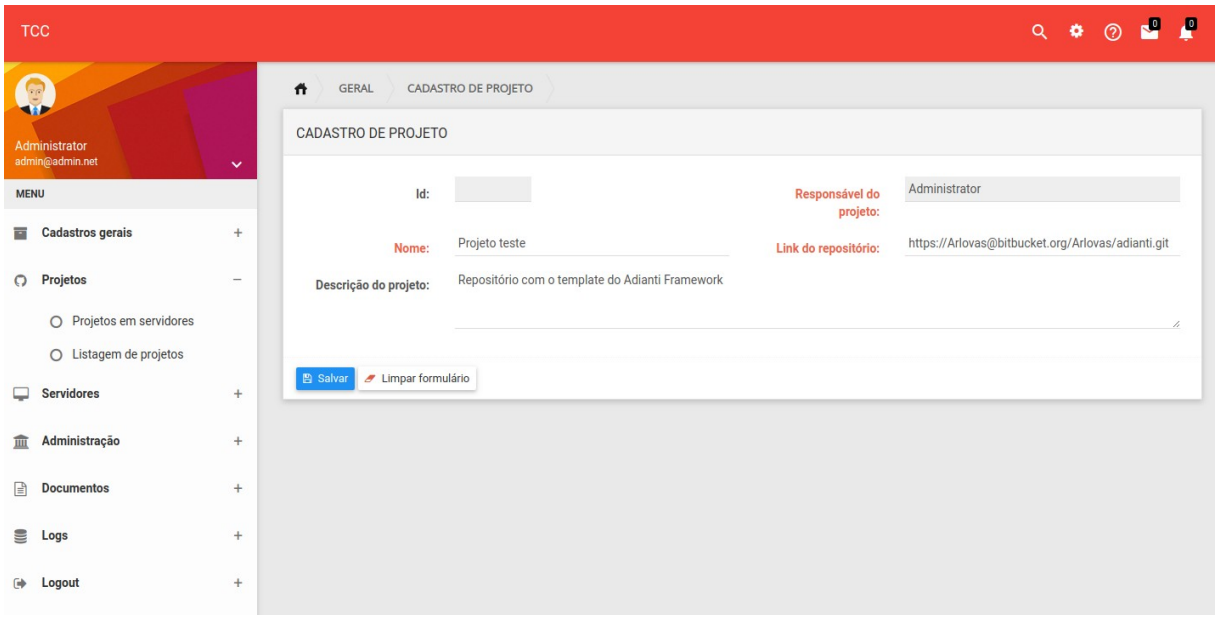
j) Criar painel com todos os servidores do usuário em formato de cartões.

4.3 Interface da ferramenta

Nesta seção são apresentadas algumas telas da ferramenta desenvolvida que demonstram sua interface.

Na Figura 9, é demonstrado o cadastro de um projeto, nesse momento são informados o nome do projeto, o responsável, a descrição do projeto e o *link* do repositório Git.

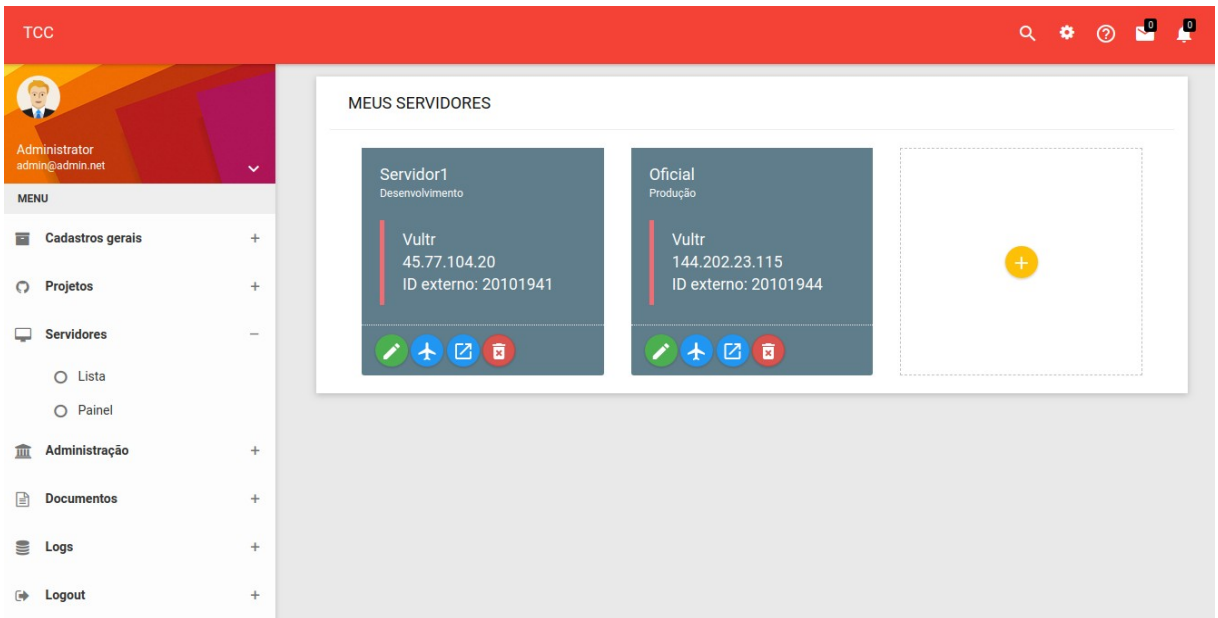
Figura 9 - Cadastrar projeto



Fonte: Feito pelo autor

Na Figura 10, é exibido o painel de servidores do usuário, cada bloco representa um servidor provisionado. Por meio dessa tela é possível acessar, realizar *deploys*, excluir e ver as informações de cada servidor, de maneira rápida e com uma interface bonita. Também é possível provisionar um novo servidor nesta mesma tela.

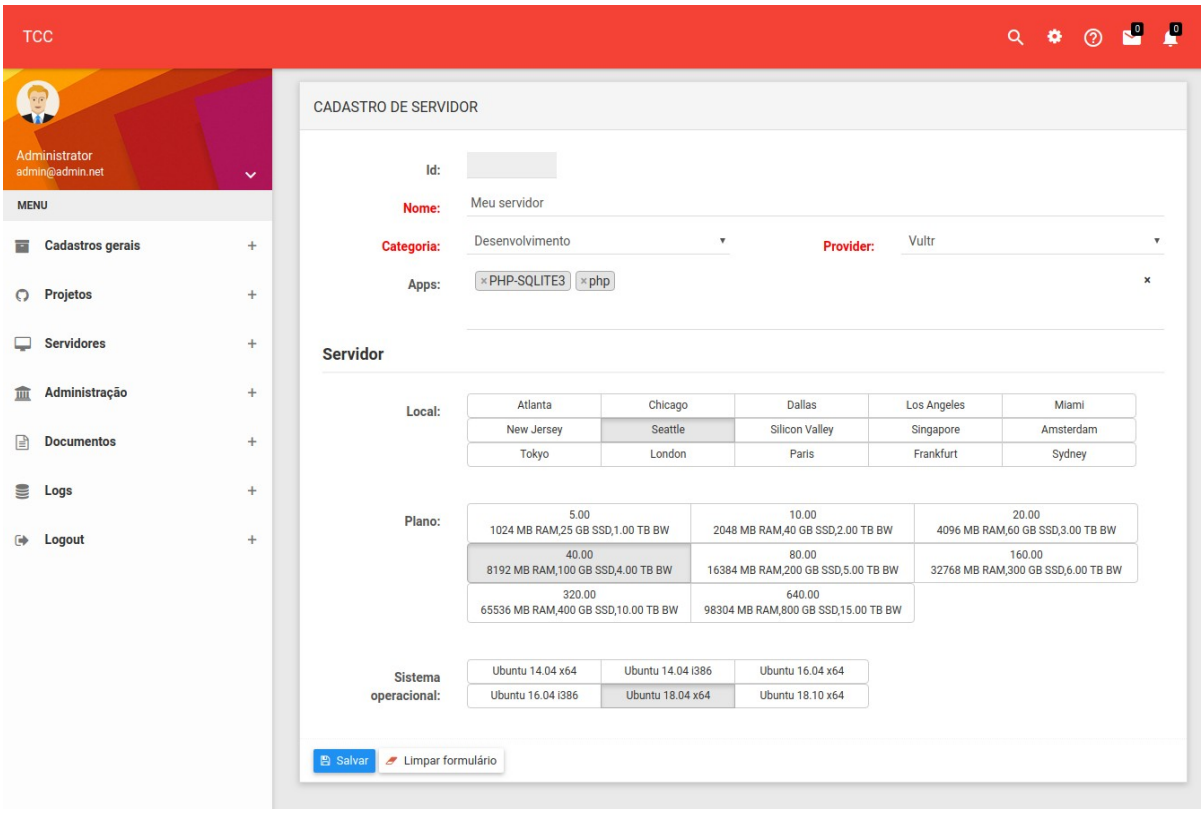
Figura 10 - Painel de servidores



Fonte: Feito pelo autor

A Figura 11, exibe a tela de provisionamento de servidores, por ela o usuário deve escolher algumas opções de hardwdare e algumas tecnologias que virão na instalação do servidor. É importante ressaltar que os campos específicos do servidor: local, plano e sistema operacional, são exibidos de maneira padrão da aplicação. No entanto seu conteúdo, é dinâmico sendo carregado pela API da *provider* escolhida. Isso significa que, no momento em que a *provider* disponibilizar um novo local para provisionar servidores, ele será automaticamente exibido dentro da ferramenta desenvolvida.

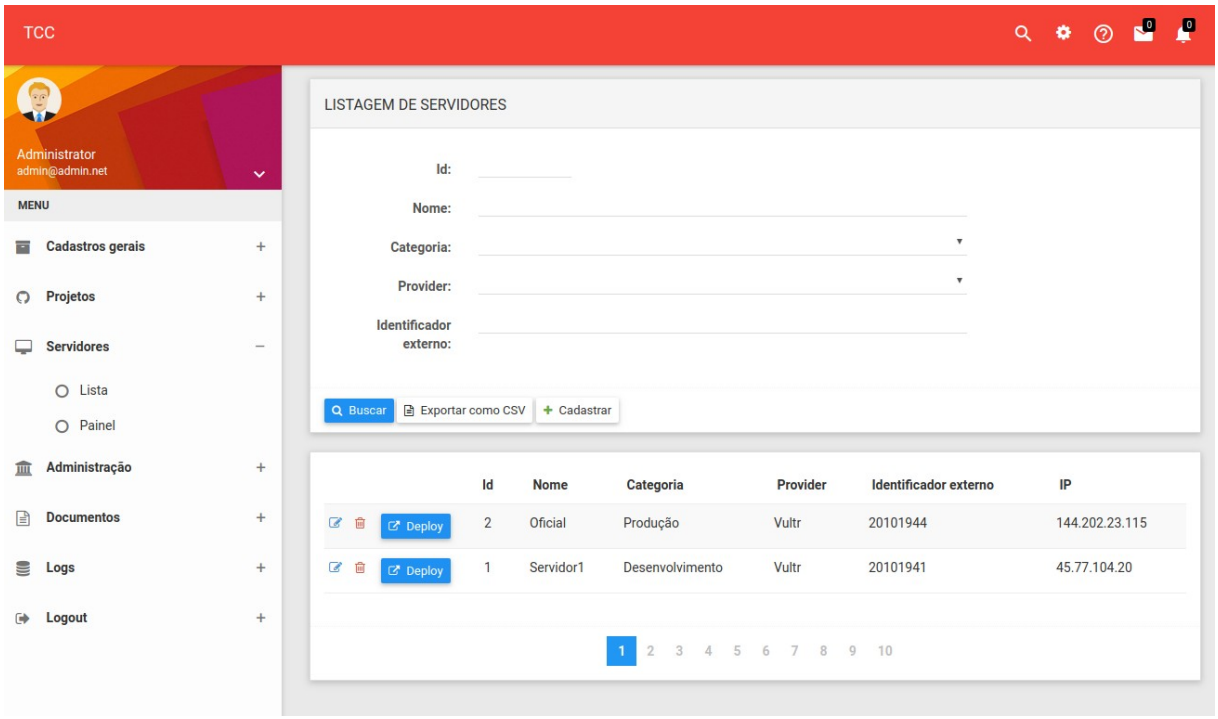
Figura 11 - Provisionar servidores



Fonte: Feito pelo autor

A Figura 12, exibe a tela lista de servidores, nela estão todos os servidores que foram provisionados pela ferramenta. Ela tem as mesmas funcionalidades do painel de servidores, portanto, com ela também é possível realizar *deploys*, provisionar e deletar servidores. No entanto esta tela conta com uma usabilidade um pouco diferente, ela funciona em formato de tabela, onde, os botões de ação ficam do lado esquerdo e as informações do servidor do lado direito. Na parte superior da tela também é possível notar um formulário de filtragem que por sua vez não aparece no painel de servidores.

Figura 12 - Lista de servidores



Fonte: Feito pelo autor

4.4 Tecnologias que foram utilizadas

O desenvolvimento da ferramenta se baseou em algumas tecnologias reconhecidas na área de TI, entre elas:

a) **PHP (PHP Hypertext Preprocessor):** É uma linguagem de programação do lado do servidor desenvolvida primordialmente para desenvolvimento web, mas também é usada em em propósitos genéricos. O PHP é responsável por receber as requisições feitas no *front-end*, processá-las e apresentar um resultado que pode ser inserir registros no banco ou devolver informações para a interface (PHP, 2017).

b) **PostgreSQL:** É um sistema de gerenciamento de dados objeto relacional, com ênfase em estabilidade e cumprimento de normas. O PostgreSQL é utilizado para armazenar dados de acesso, logs e permissão dos usuários além dos metadados (POSTGRESQL, 2017).

c) **Adianti Framework:** É um *framework* para PHP *open-source* baseado em padrões de projeto, que foi utilizado para o desenvolvimento da ferramenta. O

Adianti Framework foi escolhido pois ele busca reduzir os custos de desenvolvimento oferecendo componentes de alto nível para criação de softwares (ADIANTI, 2017).

d) **jQuery:** É uma biblioteca de JavaScript simples que acelera o desenvolvimento *front-end*, pois ele torna a manipulação de elementos HTML mais fácil. Ele ainda generaliza algumas funções que podem variar de navegador para navegador (JQUERY, 2017). O jQuery foi utilizado em momentos que era necessário manipular elementos da tela sem que uma requisição fosse gerada para o servidor.

e) **Materialize:** É um moderno *framework* responsivo para baseado nas diretrizes do Material Design. O Materialize foi desenvolvido com objetivos de acelerar o desenvolvimento web, melhorar a experiência do usuário e possui uma pequena curva de aprendizado. Ele foi utilizado para adicionar responsividade para a aplicação (MATERIALIZE, 2017).

f) **JavaScript:** É uma linguagem de programação que é executada no lado do cliente, evitando que todo o processamento tenha que passar por um servidor, atualmente já existem aplicações que rodam JavaScript no lado do servidor. O JavaScript manipulará elementos HTML para tornar a interface mais interativa (JAVASCRIPT, 2017).

4.5 Arquitetura

A arquitetura da ferramenta, desenvolvida para este trabalho, foi do tipo monolítica formada por quatro camadas: apresentação, negócio, persistência e banco de dados. Esse tipo de arquitetura foi escolhido pois facilita todo o controle de transações que são executadas pelos usuários e fez com que o desenvolvimento da plataforma mais rápido.

A Figura 13 representa o bloco da arquitetura da aplicação, com ela é possível ver a disposição das camadas juntamente com alguns itens que a compõem.

Figura 13 - Arquitetura da aplicação

Fonte: Feito pelo autor

A camada de apresentação contém tudo que compõem o visual da ferramenta, para este trabalho foi utilizado o Adianti Framework, o que resultou em um grande ganho na velocidade de desenvolvimento da interface. Ele oferece ao desenvolvedor muitos componentes de interface prontos. Um exemplo seria os campos de texto, ficam disponíveis ao desenvolvedor classes prontas, de modo que, não é necessário que o desenvolvedor se preocupe com funcionalidades básicas, criar estrutura HTML por exemplo.

Na camada de negócio ficam normalmente classes de serviço, elas são responsáveis por comunicações externas, validações elaboradas, entre outros. Um exemplo de classe de serviço usada no trabalho é a classe responsável por ser a fachada entre os *providers* disponíveis. Em todos os locais da ferramenta onde é necessário estabelecer uma comunicação com a *provider* são usados os métodos disponibilizados pela classe fachada. Isso torna qualquer trabalho de manutenção ou melhoria futura mais simples, evitando qualquer necessidade de varrer por completo o código-fonte da aplicação.

A camada de persistência é a interface que conversa diretamente com o banco de dados. Ela é responsável por validar as instruções que serão enviadas para o banco, de maneira que, inconsistências sejam criadas. O presente trabalho usa o padrão de projetos Active Record, por meio do Adianti Framework, fazendo assim a comunicação com o banco de dados.

Por fim, a camada mais baixa desta aplicação é o banco de dados, nele ficam cadastrados todos os metadados dos usuários, bem como informações administrativas da ferramenta e todos os registros de log. Neste projeto foi utilizado o banco de dados PostgreSql. A estrutura da aplicação foi pensada de maneira que, uma possível troca de banco de dados não se torne problema crítico. É utilizada uma classe PHP que traduz todas instruções SQL para o banco específico que é utilizando.

4.6 Artefatos

Durante o desenvolvimento da ferramenta, além de código-fonte alguns artefatos de software foram produzidos. Estes artefatos ajudam a documentar os processos da ferramenta, bem como auxiliam o entendimento das funcionalidades.

Os artefatos gerados foram:

a) Modelo entidade relacionamento.

b) Diagrama de classes.

c) Diagrama de casos de uso.

4.6.1 Modelo entidade relacionamento

Nesta seção serão observados três modelos de entidade relacionamento, isso ocorre pois durante o desenvolvimento da ferramenta, foi constatado ser vantajoso separar o escopo de informações em três partes:

a) Metadados, todas as informações específicas da aplicação.

b) Logs.

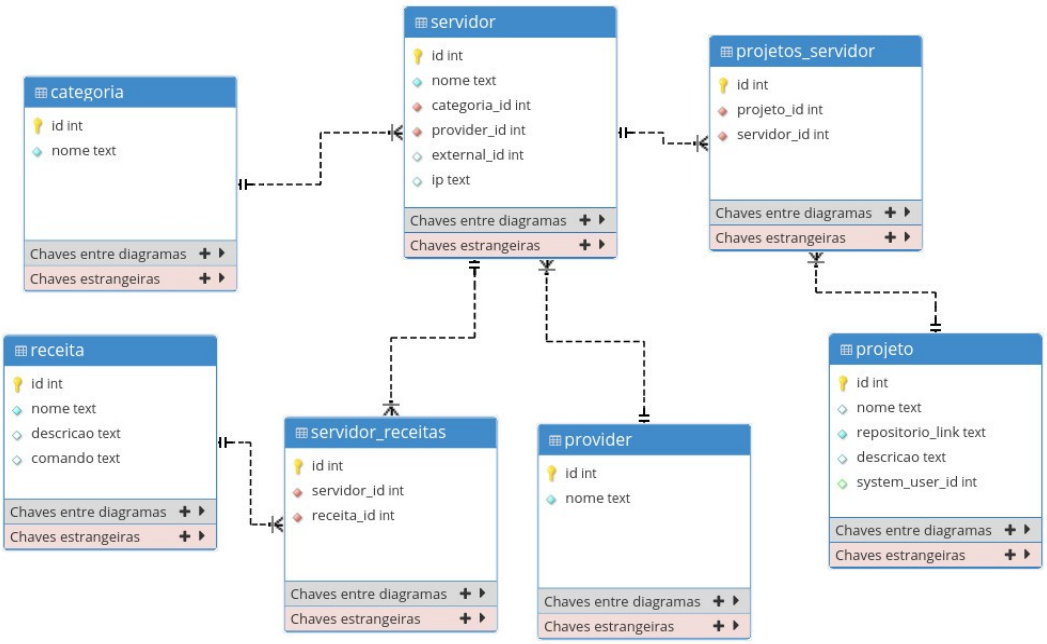
c) Permissões.

Esta divisão deve-se principalmente ao fato do projeto visar ser o mais organizado possível. De maneira que, permite rotinas futuras de backup separadas por banco de dados. É

comum acontecer que o banco de Logs cresça rapidamente e precise ser zerado e com este tipo de estrutura, manutenção individuais é uma tarefa que exige um menor esforço.

Na Figura 14 é possível visualizar o primeiro dos três modelos entidade relacionamento, o responsável por armazenar os metadados. Embora simples o modelo consegue comportar todos os objetivos do projeto. As principais entidades a serem observadas são: servidor e projetos, elas são a prioridade quando um usuário usa a ferramenta. Praticamente todas as funcionalidades evolvem essas duas entidades direta ou indiretamente.

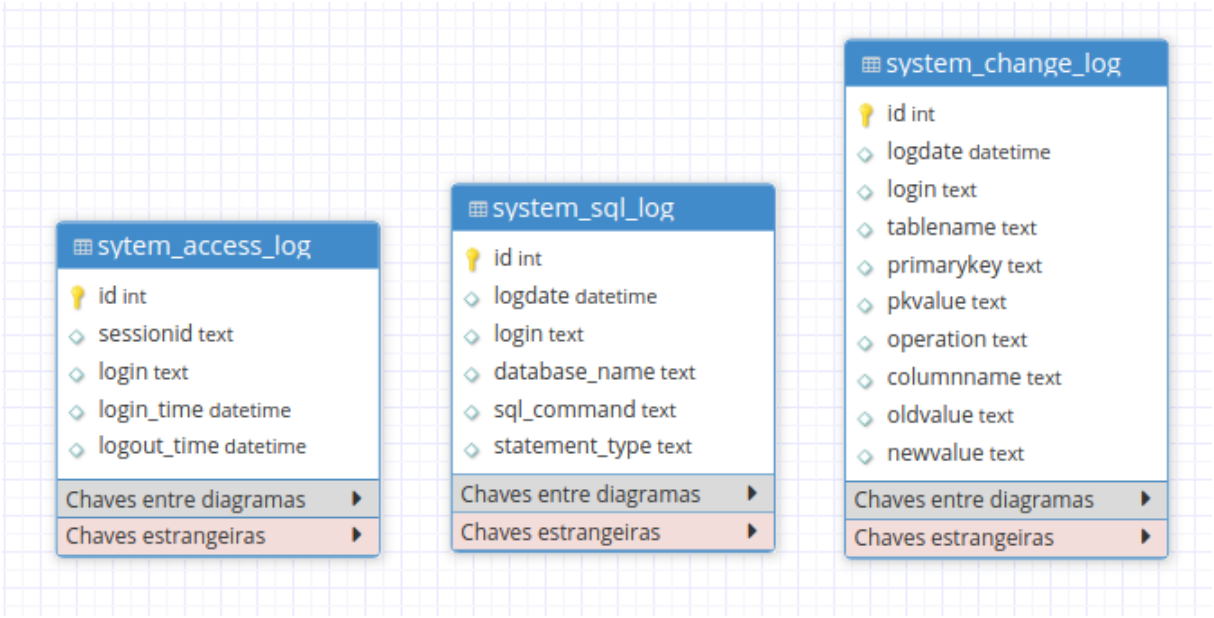
Figura 14 - Modelo entidade relacionamento metadados



Fonte: Feito pelo autor

Na Figura 15 são exibidas as três entidades responsáveis por armazenar os logs gerados na aplicação. A primeira tabela **system_access_log** armazena os dados de acesso à aplicação. A segunda tabela **system_sql_log** guarda todos os comandos enviados para o banco de dados, para configurar qual comando é salvo é necessário modificar a classe responsável pelo registros de instruções SQL. Um exemplo seria, evitar que comandos que selecionam registros sejam mantidos. A terceira tabela **system_change_log** mantém todo o histórico de alteração dos registros, contendo os valores antigos e os novos. Nesta entidade ficam registrados apenas as movimentações de registros de classes de modelo previamente definidas, via programação.

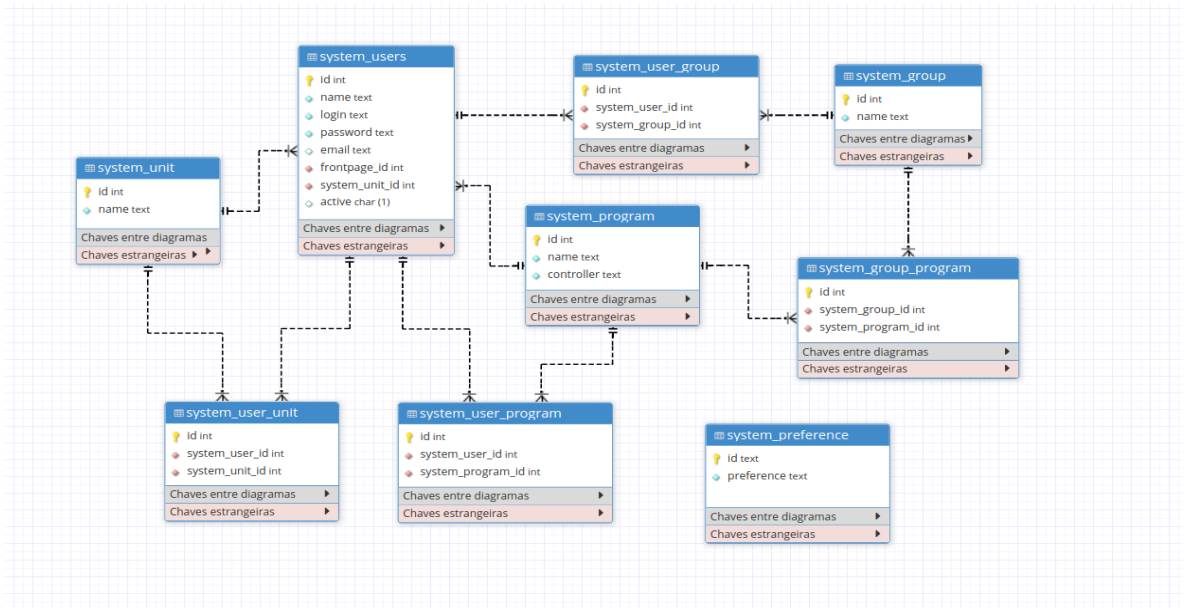
Figura 15 - Modelo entidade relacionamento Log



Fonte: Feito pelo autor

O terceiro e último modelo entidade relacionamento é o responsável por manter os dados administrativos da ferramenta. Ou seja, este modelo representa as entidades que mantém os cadastros de usuários e permissões. Sua modelagem contém uma estrutura compacta e inteligente que proporciona o cadastro de usuários com N níveis de permissão diferentes. Quando neste trabalho se fala em níveis de permissão diferentes, refere-se a possibilidade de um usuário ter acesso a telas diferentes de outro. A Figura 16 exibe a modelagem do banco.

Figura 16 - Modelo entidade relacionamento Permissão

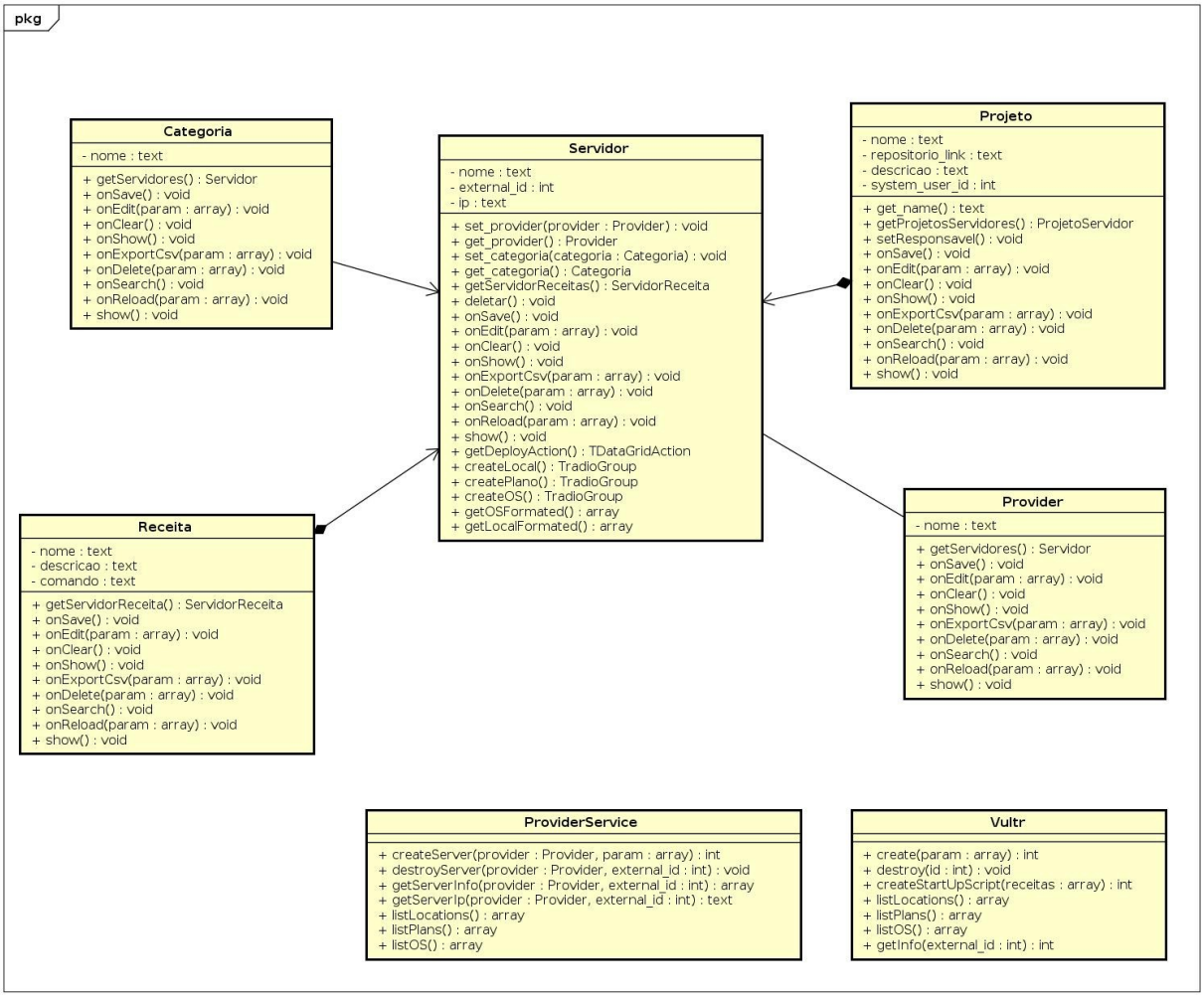


Fonte: Feito pelo autor

4.6.2 Diagrama de classes

A Figura 17 exime o diagrama de classes que representa a estrutura lógica da ferramenta desenvolvida. É possível notar na parte superior uma semelhança entre o modelo de entidade relacionamento e o modelo de classes. Isso se deve pelo fato de ter sido utilizado o padrão de projeto Active Record. O projeto desenvolvido tem sempre classes com os mesmos nomes das tabelas do banco.

Figura 17 - Modelo de classes



Fonte: Feito pelo autor

Na Figura 17, é importante observar a estrutura de fachada representada pela classe **ProviderService**. Ela é a classe que conversa com todo o sistema quando é necessário realizar uma comunicação com um *provider*. Outra classe igualmente importante que pode ser observada no modelo é a classe **Vultr**. Ela é a classe específica do *provider* escolhido.

No momento em que é necessário estabelecer uma ponte de comunicação entre a ferramenta desenvolvida e uma *provider*, por exemplo Vultr, o sistema invoca os métodos da classe ProviderService. Estes métodos, por sua vez, foram criados com o objetivo de serem genéricos e que possam ser utilizados com qualquer *provider*.

A classe ProviderService é responsável por identificar qual classe específica deve ser utilizada em cada caso, seja para realizar um *deploy* ou provisionar um servidor. Deste modo, os métodos que vão interagir com a *provider* são os métodos presentes dentro de uma classe específica, como os que estão demonstrados na classe Vultr e puderam ser vistos na figura 17.

A maneira pela qual a classe ProviderService sabe qual *provider* utilizar em cada caso, envolve um cadastro mantido pelos administradores da ferramenta. Apenas podem ser adicionadas novas *providers* quando suas próprias classes forem desenvolvidas, portanto, primeiro deve-se programar uma classe e depois adicionar esse novo *provider* no cadastro administrativo. Esse cadastro ocorre pela interface da aplicação desenvolvida que, por sua vez, é mantido no banco de dados na tabela **provider** no atributo nome.

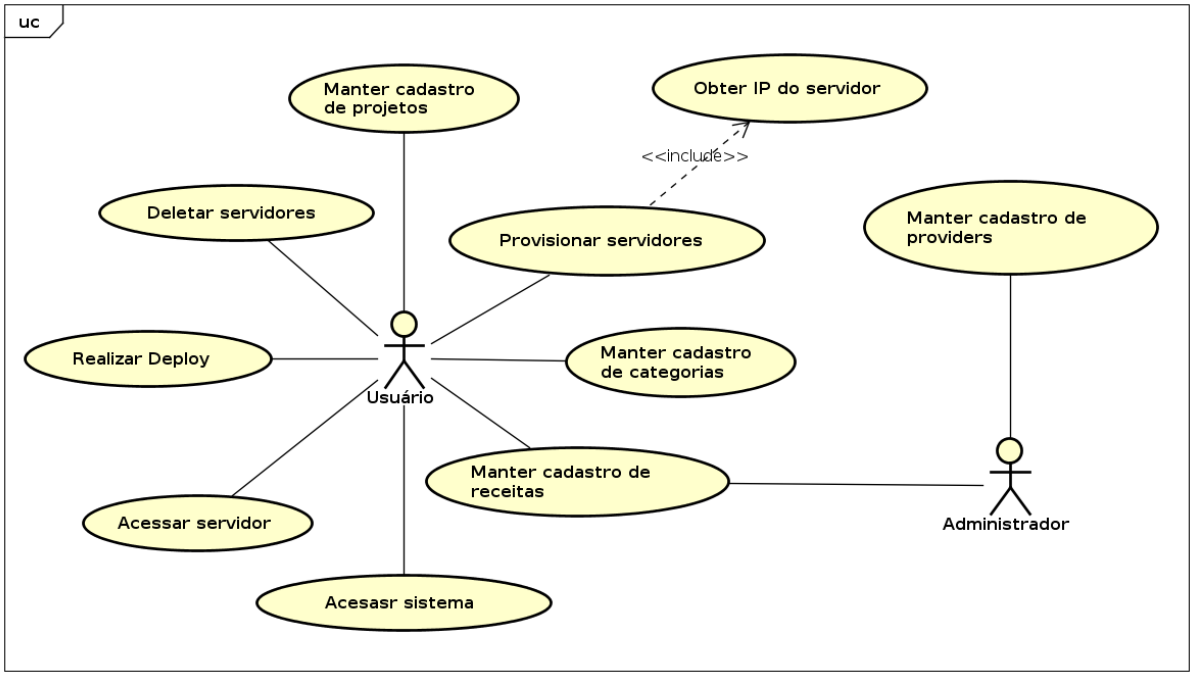
A amarração entre ProviderService e *provider* específica se fecha quando no momento de provisionar um servidor o usuário da ferramenta escolhe a *provider* de destino. Ele somente tem como opções dados que foram homologados pelos administradores da ferramenta. Assim a classe ProviderService pode consumir dinamicamente os métodos previamente conhecidos das *providers* disponíveis.

4.6.3 Diagrama de Casos de Uso

A Figura 18 exibe o diagrama de Casos de Uso da ferramenta desenvolvida, ele foi montado buscando exibir as funcionalidades específicas da ferramenta. Foram ocultados pormenores como por exemplo, cadastros de usuários e visualização de logs.

É possível notar no diagrama que mesmo aparecendo o ator administrativo, todo o foco da ferramenta está no usuário. O que vai ao encontro dos objetivos do trabalho, que são de facilitar as tarefas de *deploy* e provisionamento.

Figura 18 - Modelo de casos de uso



Fonte: Feito pelo autor

Os dois casos de uso ligados ao administrador da ferramenta: manter cadastro de *providers* e manter cadastro de receitas. São duas ações que administrador toma anterior ao uso da ferramenta por um usuário. Isso acontece pois é interessante do ponto de vista da ferramenta que algumas coisas já estejam prontas para o usuário final.

Como explicado no item do modelo de classes e ainda será detalhado no próximo, o cadastro de *providers* só pode ser realizado por administradores, sem esse cadastros não é possível provisionar nenhum servidor.

Não obrigatório, mas de maneira semelhante, o cadastro de receitas pode ser realizado de antemão, de modo que, ao utilizar a ferramenta já estarão disponíveis algumas receitas para o usuário final. Isso tira ainda mais responsabilidades dos usuários, eles não precisarão dominar conhecimentos de comandos para a instalação de receitas. Não será possível criar receitas para todos os pacotes possíveis que podem ser instalados em um servidor, quando se tratar de uma receita específica é provável que o usuário já detenha este conhecimento. Portanto, algumas das receitas mais comuns já estão disponíveis como: PHP, Apache2, PostgreSql, entre outras.

As receitas são as especificações do que será instalado em um servidor. Para cadastrar uma receita na aplicação deve-se informar um nome e o comando de instalação em ambientes

linux. Durante o provisionamento de um servidor, todas as receitas selecionadas serão instaladas.

4.7 Adição de *providers*

Para validação deste trabalho, foi utilizado apenas um *provider*, por esse motivo, todos os servidores são provisionados com ele. A escolha da Vultr como *provider* ocorreu pelo fato de representar um provider de menor porte e o objetivo por trás dessa escolha foi demonstrar que é possível provisionar servidores em empresas não tão populares como a Amazon e DigitalOcean.

Ainda que, apenas um *provider* foi disponibilizada nesta primeira versão da ferramenta. Foram tomados todos os cuidados necessários para que, seja simples, a tarefa de adicionar uma nova empresa *provider*.

Dentro destes cuidados, está a preocupação de criar uma estrutura de fachada onde toda aplicação se comunica apenas com essa interface. Desta maneira, novos *providers* podem ser adicionadas ou antigas serem removidas que a aplicação continuará funcionando conforme o esperado.

Com todo o projeto e modelo de software concluídos, ainda é necessário que programação seja envolvida na adição de um novo *provider*. Isso se deve ao fato, de que, todas as empresas utilizam métodos diferentes de comunicação de suas APIs. Isso obriga que a cada nova adição seja desenvolvida uma nova classe específica para aquele *provider.*

No momento da adição de um novo *provider*, é imprescindível o conhecimento da classe de fachada, pois o desenvolvedor deve construir todos os métodos presentes nela, para que assim, a aplicação não perca nenhuma funcionalidade.

A Figura 19 representa a classe fachada, todos os métodos presentes nela devem estar presentes na classe a ser desenvolvida. Como explicado anteriormente, esse é o momento onde deve ser desenvolvida uma nova classe para a aplicação. Depois de terminar o desenvolvimento desta classe, basta cadastrá-la na ferramenta e mais um *provider* está disponível para o usuário final.

Figura 19 - Classe fachada

```
class ProviderService
{
    public static function createServer($provider, $param)
    { ⋯
    }

    public static function destroyServer($provider, $external_id)
    { ⋯
    }

    public static function getServerInfo($provider, $external_id)
    { ⋯
    }

    public static function getServerIp($provider, $external_id)
    { ⋯
    }

    public static function listLocations()
    { ⋯
    }

    public static function listPlans()
    { ⋯
    }
    public static function listOS()
    { ⋯
    }
}
```

Fonte: Feito pelo autor

Partindo da classe fachada e criando uma nova classe de *provider*, o resultado obtido será igual a classe Vultr demonstrada no diagrama de classes. A duas classes terão a mesma estrutura, por exemplo se uma classe fosse criada para comunicação com a DigitalOcean ela teria os métodos com as mesmas assinaturas da Vultr, apenas chamando-se DigitalOcean.

5 RESULTADOS E DISCUSSÕES

Para realizar a avaliação da ferramenta desenvolvida por este trabalho, foi elaborado um questionário com nove questões quantitativas e três questões qualitativas, num total de doze perguntas, todas elas estão listadas no Apêndice A deste trabalho.

Conforme descrito na metodologia deste trabalho foram realizadas entrevistas com doze profissionais da área TI, todos eles com conhecimento de programação e envolvidos com a área de Engenharia de Software. Dois terços do público entrevistado trabalha atualmente como desenvolvedor de software, sendo que apenas dois dos entrevistados trabalham menos de um ano no atual cargo. Um terço dos entrevistados concluiu o ensino superior os demais estão atualmente cursando uma graduação na área de TI.

Antes de responder o questionário, foi fornecida uma explicação sobre o conceito e os objetivos da ferramenta desenvolvida aos avaliadores. Após a explicação foi disponibilizado para cada avaliador um roteiro de testes a ser executado dentro da aplicação. O roteiro de testes pode ser visto na Figura 20.

Em todos os momentos de avaliação foi incentivado que o avaliador verbalizasse seus pensamentos, compartilhando seus sentimentos para com a ferramenta. As falas dos avaliadores vão ao encontro das respostas obtidas no questionário, todos os avaliadores afirmaram achar a ideia da ferramenta deveras interessante e deram boas sugestões para futuros trabalhos.

Nenhuma crítica ou descontentamento foi contatado durante todo o processo de validação da ferramenta.

Figura 20 - Roteiro de testes

Fonte: Feito pelo autor

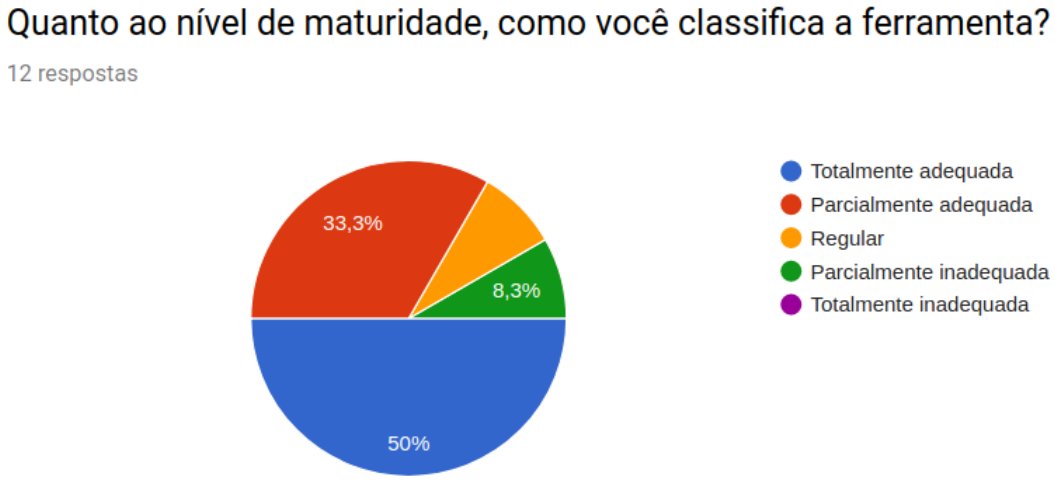
As três primeiras questões do formulário foram criadas especificamente para montar o perfil dos respondentes, deste modo as analises serão focadas nas questões quatro a doze.

A quarta questão visa avaliar a usabilidade do software e todos os avaliadores consideraram que ela tem uma usabilidade adequada, sendo que, 66% afirmaram ser totalmente adequada e o restante afirmou ser parcialmente adequada. Esta questão serviu para indicar que a interface da ferramenta está no caminho correto, fornecendo uma interface bonita e intuitiva, atendendo um dos objetivos da ferramenta.

A quinta questão visa entender a percepção dos avaliadores quanto ao nível de maturidade da ferramenta desenvolvida. As respostas foram ao encontro ao esperado para uma aplicação inicial, que ainda precisa de mais polimento.

Conforme pode ser visto na figura 21, metade dos avaliadores considera a ferramenta totalmente adequada e o restante ficou dividido entre as opções: parcialmente adequada, regular e parcialmente inadequada.

Figura 21 - Pergunta maturidade da ferramenta



Fonte: Feito pelo autor

No que se refere a terminologia da ferramenta, assunto da sexta questão, todos os avaliadores consideraram ser adequada. Da mesma maneira que a questão de número quatro os avaliadores se dividiram em dois grupos onde o maior deles, representam 66% afirmou ser totalmente adequada enquanto os outros 33% afirmaram ser parcialmente adequada.

A sétima e oitava questões, se complementam e ambas abordam o tema de viabilidade de mercado para a ferramenta desenvolvida. As respostas podem ser vistas na Figura 22.

É possível traçar um paralelo muito claro entre a verbalização e a percepção demonstradas pelos avaliadores com as respostas das questões de número sete e oito. Onde mais de 90% afirmou que pagaria para usar esta ferramenta, isso comprova que o caminho tomado no desenvolvimento da ferramenta foi o correto e realmente atendeu os objetivos.

Metade dos avaliadores afirmou que usaria a ferramenta como está, e 41.7% disse usar ela caso novos recursos sejam disponibilizados. A ferramenta desenvolvida está em estágio

inicial e não tem tudo o que precisa para ser disponibilizada comercialmente. No entanto, a avaliação foi muito positiva mesmo com poucos recursos.

Figura 22 - Viabilidade da ferramenta como produto

Fonte: Feito pelo autor

A última das questões quantitativas visa avaliar a quantidade de recursos disponíveis na ferramenta e se eles são suficientes para resolver os problemas propostos dentro do roteiro de testes. Mais de 90% dos avaliadores afirmaram que a ferramenta tem recursos suficientes, onde 58,3% afirmam serem totalmente suficientes e 33,3% dizem ser suficientes, mas com ressalvas, pois ainda faltam alguns recursos. Apenas um avaliador afirmou que a ferramenta foi inadequada.

Na questionário qualitativo os avaliadores se comportaram da mesma maneira que, durante a validação da ferramenta desenvolvida. Todos demonstraram contentamento durante a execução do roteiro de testes, nenhum grande problema ou descontentamento foi registrado.

Vale ser mencionado que, mesmo com uma grande diferença de tempo de experiência entre alguns dos avaliadores os dois grupos enxergaram que a ferramenta desenvolvida é uma boa opção, para solucionar problemas do cotidiano de de um desenvolvedor de software.

Para os menos experientes, a ferramenta desenvolvida assume o papel de sanar uma falta de conhecimento, devido à pouca experiência. Sendo assim, eles já podem começar a disponibilizar os softwares que produzem, mesmo sem ainda saber como com configurar um servidor e realizar um *deploy*.

Os avaliadores mais experientes, visualizaram a ferramenta desenvolvida como uma possibilidade de poupar tempo com tarefas operacionais e focar em áreas críticas do desenvolvimento de software.

No Quadro 1, são exibidas todos os comentários positivos, deixados no questionário qualitativo. É importante ressaltar os comentários que comprovam que a ferramenta, tem sim uma interface intuitiva. Além destes, outros enfatizam a velocidade em criar um servidor e realizar um *deploy.*

Quadro 1 - Pontos positivos da ferramenta

Facilidade na busca e localização de serviços disponíveis, pois torna-se custoso a busca pelo melhor provider existentes no mercado, podendo ocasionar a aquisição de de um provider que não atende plenamente os requisitos necessários.
Agilidade em criar servidor, facilidade na utilização
Facilita o trabalho; é de fácil utilização
A ferramenta é de fácil entendimento, agiliza muito o processo, principalmente quem não tem muito conhecimento.
Excelente ideia e interface, já está funcionando corretamente, conforme a proposta.
Auxílio completo na configuração de servidores, de forma rápida e prática.
Conceito interessante, ansioso por ver ela completa.

O Quadro 2, são mostrados os comentários negativos, quanto a ferramenta desenvolvida. Destacam-se os que mencionam melhorias a serem feitas em nomes de funções e mensagens de *status* e os apontam que a ferramenta por estar em estágio inicial precisa evoluir.

Quadro 2 - Pontos negativos da ferramenta

Melhoria na descrição das funções, estas deve ser mais auto explicativas, fazendo com que o usuário possa ter domínio pleno e sem duvidas dos processos que está fazendo.
Alguns erros aconteceram durante o teste, Não estava claro o status do meu servidor, se ele estava criado, sendo criado, deu erro.
Não identifiquei pontos negativos.
Nenhum a informar, precisa apenas de mais polimento, o que é comum em qualquer software do mercado.
Ferramenta em estágio inicial.

A última pergunta qualitativa visa enxergar possíveis melhorias e críticas construtivas quanto a ferramenta desenvolvida. As sugestões foram devidamente anotadas e adicionadas na lista de trabalhos futuros.

Quadro 3 - Comentários, críticas ou sugestões sobre a ferramenta

Excelente ferramenta, proporcionará um ganho para o segmento, onde o analista/programador poderá conhecer novos providers.
Ter algumas maquinas prontas com alguns serviços já prontos como por exemplo, Apache + PHP + Postgres
Ficou muito bom, quando precisei criar droplet na digitalocean sozinho tive bastante dificuldade, com essa ferramenta eu teria poupado muito tempo, desde a pesquisa por preços até a configuração do servidor.
Como o próprio Artur comentou, faltam alguns recursos de segurança, que provavelmente

ainda serão implementados. Acredito que o aplicativo está bem bom, rápido e prático.

6 CONSIDERAÇÕES FINAIS

A importância de ferramentas que auxiliam a entrega e controle de software, é realidade dentro de empresas que buscam agilidade e produtividade. Conforme as empresas crescem o número de servidores aumenta, o gerenciamento manual fica impraticável. Isso fica ainda mais evidente no momento que algum erro grave acontece. Uma ferramenta que permita gerenciar as versões de software em todos os servidores torna mais fácil o trabalho de um DevOps.

Conforme apresentado, a Computação em Nuvem traz muitos benefícios, como agilidade e economia. O presente trabalho está diretamente ligado a este conceito proporcionando que profissionais com pouca experiência consigam em poucos minutos ter seus softwares em ambientes de produção. Da mesma maneira, permite que grandes empresas possam validar ideias em ambientes individualizados, sem a necessidade de investimentos em hardware.

Durante a validação da ferramenta desenvolvida os avaliadores verbalizaram e escreveram o seu contentamento com as possibilidades que a ferramenta oferece, entre elas a agilidade de realizar o *deploy* em novo ambiente.

Um fator que demonstra o sucesso deste trabalho, é o grande interesse que os avaliadores demonstraram durante os testes e o desejo de poder usar esta ferramenta no seu cotidiano.

Por meio das avaliações realizadas, mais alguns pontos foram adicionados aos trabalhos futuros do trabalho. O mais importante deles é adicionar novos *providers* para a

ferramenta. Além de, oferecer mais receitas aos desenvolvedores melhorias na interface e nos nomes das funções. Adições de grandes funcionalidades também estão no *roadmap* de desenvolvimento, como controle de versões do software, habilitando o *rollback* e controle de qualidade de código-fonte utilizando testes de unidade.

REFERÊNCIAS

ADIANTI. **Adianti Framework.** Disponível em: <http://www.adianti.com.br/>. Acesso em: 31 de out. 2017.

AIELLO, B; SACHS, L. **Configuration Management:** Best Practices. Boston: Pearson Educatio, 2010. 230 p.

ANDRADE, J. R. **Gerência de configuração.** São Paulo: Casa de Ideias, 2015. 195p.

BACH, J. **The highs and lows of change control.** Los Alamitos. IEEE Computer Society, 1998. p 115.

CMMI-DEV. **CMMI for development:** Versão 1.3. Carnegie Mellon University: Software Engineering Institute, 2010.

DEBOIS, P. **Agile Infrastructure & Operations.** In: Agile 2008 Toronto, 2008, Toronto. 38 p.

EDWARDS, D. **What is DevOps?** Dev2Ops.org 2010. Disponível em: <http://dev2ops.org/2010/02/what-is-devops/>. Acesso em: 03 de out. 2017.

ENGHOLM JR., Hélio. **Engenharia de Software na Prática.** São Paulo: Novatec Editora, 2011. 439 p.

FONSECA, J. J. S. **Metodologia da pesquisa científica**. Fortaleza: UEC, 2002. 127 p.

FORSGREN, N. **State of DevOps Report**. Puppet 2017. 53 p.

FOWLER, M. **SnowflakeServer.** 2012. Dísponível em: <https://martinfowler.com/bliki/SnowflakeServer.html> Acessp em: 05 de out. 2017.

GERHARDT, T. E.; SILVEIRA, D. T. **Métodos de Pesquisa**. Porto Alegre: UFRGS, 2009. 120 p.

GIL, Antonio Carlos. **Como elaborar projetos de pesquisa**. 5ª Edição. São Paulo: Atlas, 2010. 200 p.

GIT. **Git:** fast-version-control, 2017. Disponível em: <https://git-scm.com/>. Acesso em: 09 de out. 2017.

HIRAMA, Kechi. **Engenharia de Software:** Qualidade e Produtividade com Tecnologia. Rio de Janeiro: Elsevier, 2012. 232 p.

HUMBLE, J.; FARLEY, D. **Continuous Delivery:** Reliable Software Releases Through Build Test and Deployment Automation. Boston: Pearson Education, 2013. 496 p.

IBM. **DevOps for hybrid cloud: an IBM point of view:** How DevOps for hybrid cloud can help organizations succeed with digital reinvention. New York: IBM Systems, 2017. 16 p.

IEEE Std 828-1990. **IEEE Standard Glossary of Software Engineering Terminology.** IEEE Computer Society, 1990. 84 p.

JAVASCRIPT. **Javascript.** Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>. Acesso em: 31 de out. 2017.

JIRA. **Jira software,** 2017. Disponível em: <https://br.atlassian.com/software/jira> Acesso em: 09 de out. 2017.

JQUERY. **jQuery.** Disponível em: <https://jquery.com>. Acesso em: 31 de out. 2017.

MARCONI, M. A,; LAKATOS, E. M. **Fundamentos de Metodologia Científica.** 5ª Edição. São Paulo: Editora Atlas, 2003. 320 p.

MATERIALIZE. **Materialize.** Disponível em: <http://materializecss.com>. Acesso em: 31 de out. 2017.

MINAYO, M. C. S. **O desafio do conhecimento:** Pesquisa qualitativa em saúde. 11ª edição. São Paulo: HUCITEC, 2007. 408 p.

MORESI, E. **Metodologia da Pesquisa.** 2003. 108 f. Pós-Graduação Stricto Sensu, Universidade Católica de Brasília, Brasília, 2003.

NATO, S. C. **Software Engineering:** A Report on a Conference Sponsored by the NATO Science Committee. Garmisch: NATO, 1960. 136 p.

PHP. **PHP**, 2017. Disponivel em: <http://php.net/>. Acesso em: 31 de out. 2017.

PILATO, C. M.; FITZPATRICK B. W.; COLLINS-SUSSMAN B. **Version Control with Subversion:** For Subversion 1.7. Stanford, 2013 468 p.

POSTGRESQL. **PostgreSQL**, 2017. Disponível em: <https://www.postgresql.org/>. Acesso em: 31 de out. 2017.

PRESSMAN, Roger S. **Software Engineering:** Uma abordagem profissional. 7ª Edição. Porto Alegre: AMGH, 2011. 780 p.

PUPPET. **Puppet**, 2017. Disponível em: <https://puppet.com/docs>. Acesso em: 31 de out. 2017.

REDMINE. **Redmine**, 2017. Disponível em: <http://www.redmine.org/>. Acesso em: 09 de out. 2017.

SANTOS, A. R. **Metodologia científica:** a construção do conhecimento. 2ª edição. Rio de Janeiro: DP&A editora, 1999. 166 p.

SATO, D. **DevOps:** Na prática: entrega de software confiável e automatizada. São Paulo: Casa do Código, 2014. 246 p.

STOTT, L. **DevOps the Wall of Confusion understanding the basics of DevOps.** Microsoft 2016. Disponível em: <https://blogs.msdn.microsoft.com/uk_faculty_connection/2016/06/23/devops-the-wall-of-confusion-understanding-the-basics-of-devops/> Acesso em: 04 de out. 2017.

SHAFER, A. **Agile Infrastructure:** A Story in Three Acts. In: Velocity, 2009, San Jose, 189 p.

SHARMA, S. **DevOps For Dummies, IBM Limited Edition.** New Jersey: John Wiley & Sons, 2014. 51 p.

SOMMERVILLE, I. **ENGENHARIA DE SOFTWARE.** 9ª edição. São Paulo: Pearson Prentice Hall, 2012. 529 p.

SUBVERSION, **Apache Subversion:** Enterprise-class centralized version control for the masses, 2017. Disponível em: <https://subversion.apache.org/> Acesso em: 09 de out. 2017.

SWEBOK. **Guide to the Software Engineering Body of Knowledge:** SWEBOK IEEE Computer Society. Versão 3.0. Los Alamitos: IEEE, 2004. 335 p.

TAVARES, B. **Puppet e Vagrant:** Como provisionar máquinas para seu projeto, 2013. Disponível em: <https://www.thoughtworks.com/pt/insights/blog/puppet-and-vagrant-how-provision-machines-your-project> Aceeso em: 04 de out. 2017.

VAGRANT, **Vagrant**, 2017. Disponível em: <https://www.vagrantup.com/intro/index.html>. Acesso em: 31 de out. 2017.

WAINER, J. **Métodos de pesquisa quantitativa e qualitativa para a Ciência da Computação**. In: KOWALTOWSKI, Tomasz; BREITMAN, Karin; organizadores. Atualizações em Informática 2007. Rio de Janeiro: Ed. PUC-Rio; Porto Alegre: Sociedade Brasileira de Computação, 2007.

WALLS, M. **Building a DevOps Culture:** DevOps is as much about culture as it is about tools. Sebastopol: O'Reilly Media, 2013. 27 p.

APÊNDICE

Apêndice A – Questionário de avaliação do ambiente

1. Qual a sua área de atuação profissional atual?

   ( ) Desenvolvedor

   ( ) Analista

   ( ) Gerente de projetos

   ( ) Outros

2. Quanto tempo de experiência você tem na sua função atual?

   ( ) 1 ano ou menos

   ( ) 1 a 2 anos

   ( ) 2 a 4 anos

   ( ) 4 a 6 anos

   ( ) 6 anos ou mais

3. Qual o seu nível de escolaridade?

   ( ) Superior incompleto

   ( ) Superior completo

4. Como você avalia a usabilidade deste software?

   ( ) Totalmente adequada

( ) Parcialmente adequada

( ) Regular

( ) Parcialmente inadequada

( ) Totalmente inadequada

5. Quanto ao nível de maturidade, como você classifica a ferramenta?

( ) Totalmente adequada

( ) Parcialmente adequada

( ) Regular

( ) Parcialmente inadequada

( ) Totalmente inadequada

6. Quanto à terminologia usada na ferramenta, você considera que os termos estão de acordo e são de fácil compreensão?

( ) Totalmente adequada

( ) Parcialmente adequada

( ) Regular

( ) Parcialmente inadequada

( ) Totalmente inadequada

7. Do ponto de vista de consumidor, você usaria esta ferramenta?

( ) Sim, exatamente como está

( ) Sim, somente com mais recursos

( ) Não, pois não me atende

( ) Não, pois não tem os recursos necessários para este tipo de ferramenta

8. Você pagaria para usar esta ferramenta?

( ) Sim

( ) Não

9. Como você avalia os recursos disponíveis na ferramenta?

   ( ) Suficientes, atendem a proposta

   ( ) Suficientes, mas com ressalvas, pois faltam alguns recursos

   ( ) Insuficientes, ferramenta incompleta

10. Destaque pontos positivos na ferramenta

11. Destaque pontos negativos na ferramenta

12. Registre aqui qualquer comentário, crítica ou sugestão sobre a ferramenta