UNIVERSIDADE DE PASSO FUNDO
Programa de Pós-Graduação em Computação Aplicada

Dissertação de Mestrado

APLICAÇÃO DA INTEGRAÇÃO CONTÍNUA EM UM SISTEMA DE AUXÍLIO À TOMADA DE DECISÃO

MARINA DEZORDI LOPES

UPF

UNIVERSIDADE DE PASSO FUNDO
INSTITUTO DE CIÊNCIAS EXATAS E GEOCIÊNCIAS
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA

APLICAÇÃO DA INTEGRAÇÃO CONTÍNUA EM UM SISTEMA DE AUXÍLIO À TOMADA DE DECISÃO

Marina Dezordi Lopes

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Computação Aplicada na Universidade de Passo Fundo.

Orientador: Prof. Carlos Amaral Hölbig
Coorientador: Prof. Willingthon Pavan

Passo Fundo
2020

CIP – Catalogação na Publicação

L864a Lopes, Marina Dezordi
Aplicação da integração contínua em um sistema de auxílio à tomada de decisão / Marina Dezordi Lopes. – 2020.
49 f. : il. ; 30 cm.

Orientador: Prof. Carlos Amaral Hölbig.
Coorientador: Prof. Willingthon Pavan.
Dissertação (Mestre em Computação Aplicada) – Universidade de Passo Fundo, 2020.

1. Software - Desenvolvimento. 2. Software integrado. 3. Sistemas de suporte de decisão. I. Hölbig, Carlos Amaral, orientador. II. Pavan, Willingthon, coorientador. III. Título.

CDU: 004.42

Catalogação: Bibliotecário Luís Diego Dias de S. da Silva – CRB 10/2241

UPF

PPGCA
Programa de Pós-Graduação
em Computação Aplicada
Instituto de Ciências Exatas e Geociências | ICEG

ATA DE DEFESA DO TRABALHO DE CONCLUSÃO DE CURSO DO ACADÊMICO

MARINA DEZORDI LOPES

Aos vinte e cinco dias do mês de março do ano de dois mil e vinte, às quatro horas, realizou-se, no, prédio D01 sala 01, da Universidade de Passo Fundo (UPF), a sessão pública de defesa do Trabalho de Conclusão de Curso " Aplicação da integração contínua em um sistema de auxílio à tomada de decisão", de autoria de Marina Dezordi Lopes, acadêmica do Curso de Mestrado em Computação Aplicada do Programa de Pós-Graduação em Computação Aplicada – PPGCA. Segundo as informações prestadas pelo Conselho de Pós-Graduação e constantes nos arquivos da Secretaria do PPGCA, a aluna preencheu os requisitos necessários para submeter seu trabalho à avaliação. A banca examinadora foi composta pelos doutores Carlos Amaral Hölbig, Willingthon Pavan, Alexandre Lazaretti Zanatta, e Santiago Vianna Cuadra. Concluídos os trabalhos de apresentação e arguição, a banca examinadora considerou a candidata APROVADA. Foi concedido o prazo de até quarenta e cinco (45) dias, conforme Regimento do PPGCA, para a acadêmica apresentar ao Conselho de Pós-Graduação o trabalho em sua redação definitiva, a fim de que sejam feitos os encaminhamentos necessários à emissão do Diploma de Mestre em Computação Aplicada. Para constar, foi lavrada a presente ata, que vai assinada pelos membros da banca examinadora e pela Coordenação do PPGCA.

Prof. Dr. Carlos Amaral Hölbig – UPF
Presidente da Banca Examinadora
(Orientador)

Prof. Dr. Willingthon Pavan – UPF
(Coorientador)

Prof. Dr. Alexandre Lazaretti Zanatta – UPF
(Avaliador Interno)

Prof. Dr. Santiago Vianna Cuadra- Embrapa Informática
(Avaliador Externo)

Prof. Dr. Rafael Rieder
Coordenador do PPGCA

AGRADECIMENTOS

Primeiramente gostaria de agradecer à Universidade de Passo Fundo, presente na minha vida acadêmica e profissional, a instituição me proporciona avançar cada vez mais nessas duas áreas, oferecendo sempre qualidade, oportunidades e um ambiente favorável para a construção do conhecimento.

Agradeço ao meu orientador Prof. Dr. Carlos Amaral Hölbig por estar sempre disponível, por ter me guiado e incentivado durante todo o processo de desenvolvimento deste trabalho, além das correções e contribuições muito oportunas. Agradeço também ao meu coorientador Prof. Dr. Willingthon Pavan por compartilhar o conhecimento, auxiliar nas definições, contribuir com ideias e sanar dúvidas que surgiram durante a evolução do processo.

Finalmente, gostaria de agradecer profundamente ao meu esposo Fernando pelo incentivo contínuo, paciência, apoio e compreensão no decorrer da realização dos meus estudos; aos meus pais Marcos e Ana, e ao meu irmão Lucas, por serem meus primeiros incentivadores, por me guiarem no caminho até aqui e me darem todo o suporte que precisei para a conquista dos meus sonhos.

Muito obrigada.

Marina Dezordi Lopes

APLICAÇÃO DA INTEGRAÇÃO CONTÍNUA EM UM SISTEMA DE AUXÍLIO À TOMADA DE DECISÃO

RESUMO

Durante o processo de desenvolvimento, de melhorias e de novas funcionalidades de um sistema, os desenvolvedores realizam uma série de operações manuais para garantir que cada alteração tenha o resultado esperado e que não afete de maneira errada outras funcionalidades da aplicação. Para isso são realizados testes, integração de códigos, verificação da nova versão, entre outros. Se realizado manualmente, esse processo pode demorar e não ter a confiabilidade desejada, onerando custos e compromentendo a integridade do sistema. Como uma abordagem para solucionar este problema sugere-se a utilização da técnica de Integração Contínua, visando a automatização desse conjunto de operações. Devido a estes fatores, este trabalho objetiva a utilização da Integração Contínua aplicada ao sistema de tomada de decisão DSSAT. Como ferramentas para a aplicação da Integração Contínua foram selecionados as ferramentas Travis CI, CircleCI e Jenkins, com base em critétios que atendem as necessidades do DSSAT. Todas as ferramentas possuem integração com o Github e com o Docker, o que facilitou sua aplicação ao DSSAT. O processo de integração realizado com o uso das três ferramentas mostrou-se satisfatório, apresentando aspectos positivos para cada uma das ferramentas, possibilitando, assim, a efetiva automatização do processo de versionamento do DSSAT e, por consequência, uma redução no trabalho realizado pelos seus administradores. Por fim, mostrou-se ser uma ferramenta adequada para os desenvolvedores realizarem a conferência de suas alterações, ocasionando uma melhora no fluxo do processo de integração e um controle mais efetivo dos testes realizados.

Palavras-Chave: CircleCI, DSSAT, Integração Contínua, Jenkins, Travis CI.

APPLICATION OF CONTINUOUS INTEGRATION IN A DECISION SUPPORT SYSTEM

ABSTRACT

During the process of development, improvements and new functionalities of a system, developers perform a series of manual operations to ensure that each change has the expected result and that it does not wrongly affect other functionalities of the application. For this, tests are carried out, code integration, verification of the new version, among others. If performed manually, this process can take time and not have the desired reliability, increasing costs and compromising the integrity of the system. As an approach to solve this problem, the use of the Continuous Integration technique is suggested, aiming at the automation of this set of operations. Due to these factors, this work aims at the use of Continuous Integration applied to the DSSAT decision-making system. As tools for the application of Continuous Integration, Travis CI, CircleCI and Jenkins were selected based on criteria that meet the needs of the DSSAT. All tools are integrated with Github and Docker, which facilitated their application to DSSAT. The integration process carried out with the use of the three tools proved to be satisfactory, with positive aspects for each of them, thus enabling the effective automation of the DSSAT versioning process and, consequently, a reduction in the work done by its managers. The CI showed be an proper tool for developers to check their changes, an improvement in the flow of the integration process and a more effective control of the tests performed.

Keywords: CircleCI, Continuos Integration, DSSAT, Jenkins, Travis CI.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

A entrega de uma nova funcionalidade ou alteração de um software em produção para utilização do usuário deve ser confiável e passar por todas as rotinas de testes necessárias para garantir a integridade do sistema. Dependendo do tamanho do software, esses testes podem ser demorados, demandando mais tempo dos desenvolvedores para chegar a um resultado satisfatório.

Um sistema de apoio a tomada de decisão tem a necessidade de a cada atualização e integração do sistema verificar se a aplicação continua entregando os resultados esperados, considerando as diferenças que os executáveis de cada versão e/ou de cada sistema operacional apresentam, as dependências do software e as configurações necessárias para sua correta execução.

Do ponto de vista dos desenvolvedores é importante que se tenha uma organização durante o processo de codificação, de testes, de integração e de implantação do sistema, que garanta a geração de menos erros, que reduza riscos e que faça com que falhas sejam descobertas e corrigidas o quanto antes, evitando tomar uma proporção maior e tornar mais difícil e cara a correção.

A entrega de uma nova funcionalidade de um sistema exige que os desenvolvedores façam uma verificação por meio de testes para validação da nova versão, garantindo a sua integridade. Entretanto, quando realizado manualmente, o processo pode demorar e impactar na confiabilidade do sistema. Agregado a isso, um sistema multiplataforma que possui grande quantidade de diferentes dependências e de configurações exige uma quantidade maior de testes e de trabalho.

Os desenvolvedores dedicam tempo executando todos os passos a cada nova funcionalidade que desejam testar para, finalmente, estarem com as alterações prontas para serem integradas ao sistema. Quando executam estes passos de forma manual, tem-se pouca garantia de que está sendo entregue uma alteração consistente e que essa alteração não influenciou em outras funcionalidades do sistema.

Um processo de organização de desenvolvimento de sistema, com verificação automática de dependências e de configurações, de testes e de integração pode tornar o trabalho do desenvolvedor mais rápido e mais confiável. Além disso, acredita-se que é possível a obtenção de uma maior agilidade na entrega de novas versões de sistemas, com mais qualidade e segurança.

Este trabalho utiliza a técnica de Integração Contínua (CI, do inglês *Continuous Integration*) [1, 2] para organização do processo de desenvolvimento e de automatização da compilação e dos testes antes da realização da integração de novas versões de sistemas, a

partir de ferramentas específicas de utilização do desenvolvedor e dos administradores do repositório de versionamento.

A falta de uma automatização e organização do processo de integração de novas versões da suíte DSSAT (Sistema de Suporte para Decisão e Transferência em Agrotecnologia) [3, 4], incentivou a realização deste trabalho. Atualmente os desenvolvedores e administradores do DSSAT fazem de forma manual as atualizações, as integrações, as verificações de resultados, os testes e a implantação das novas versões da suíte. O objetivo principal é o processo de automatização de versionamento da suite DSSAT incluindo a compilação, os testes e o resultado final.

O capitulo 2 apresenta um referencial teórico abordando os conceitos da Integração Contínua, um estudo comparativo sobre suas principais ferramentas e como eles se adequam ao DSSAT e uma revisão de literatura sobre CI, apresentando trabalhos que abordam, principalmente, o uso do CI e de algumas de suas ferramentas em aplicações reais. A seguir, no capítulo 3, será abordado a metodologia utilizada para a implantação da CI no DSSAT, incluido o cenário atual, demonstrando o fluxo de como funciona a integração hoje, a escolha das ferramentas e o ambiente tecnológico construído. No capítulo 4 é realizada uma apresentação e análise dos resultados obtidos com a aplicação do CI no processo de versionamento do DSSAT. Por fim, no capítulo 5, são apresentadas as conclusões gerais da pesquisa realizada por este trabalho bem como apresentadas algumas possibilidades de continuidade de trabalho relacionado a implantação deste processo no DSSAT.

2. REFERENCIAL TEÓRICO

Este capítulo aborda a história e os conceitos relacionados a Integração Contínua, além de uma descrição sobre algumas ferramentas utilizadas para implantação da CI em sistemas computacionais. Na última seção deste capítulo, foi realizado um levantamento bibliográfico de alguns trabalhos que apresentam estudos e diversas aplicações da Integração Contínua em diferentes áreas e sistemas computacionais.

2.1 INTEGRAÇÃO CONTÍNUA

O princípio de Integração Contínua foi apresentado num contexto de orientação a objetos por Grady Booch [1] em 1991. A ideia era ter, em intervalos regulares, integrações com uma nova funcionalidade executável a cada atualização. Esse conceito foi amplamente utilizado na metodologia *Extreme Programming* (XP) [5]. Porém, foi Martin Fowler que, por meio de uma publicação em seu blog, foi o responsável por disseminar a ideia de Integração Contínua alcançando um nível maior de aceitação [2].

Integração Contínua é um processo realizado durante o desenvolvimento de software em que a integração dos trabalhos de uma equipe ocorre com muita frequência, ao menos uma vez ao dia. Em cada integração é executada uma nova *build*[1] automatizada do software e essa abordagem possibilita que a equipe identifique possíveis problemas e que possa resolvê-los rapidamente [6].

A prática da Integração Contínua acontece toda vez que uma nova alteração é adicionada ao sistema e enviada ao repositório remoto. O passo seguinte é executar a *build* e realizar os testes automáticos para cobrir todo o sistema, incluindo as partes recém-adicionadas. O desenvolvedor recebe um retorno sobre os códigos integrados e, em caso de falha, deve priorizar os ajustes [7].

Na Figura 1 é possível visualizar o processo de Integração Contínua apresentado em etapas e a agregação de dois conceitos que são amplamente relacionados: Entrega Contínua (*Continuos Delivery*) e Implantação Contínua (*Continuos Deployment*). Primeiramente é realizado o *commit*[2] das alterações, em seguida a execução da *build* e a realização dos testes automatizados, fases da Integração Contínua. Na etapa seguinte é realizada a Entrega Contínua, quando acontece a implantação em ambiente de homologação e a realização dos testes de integração. Finalmente, na Implantação Contínua, é feita a implantação das alterações em ambiente de produção [8].

[1] Versão compilada do software.

[2] Pacote de modificações realizadas em um determinado repositório, contendo informações dos arquivos alterados, autor e comentário sobre a alteração.

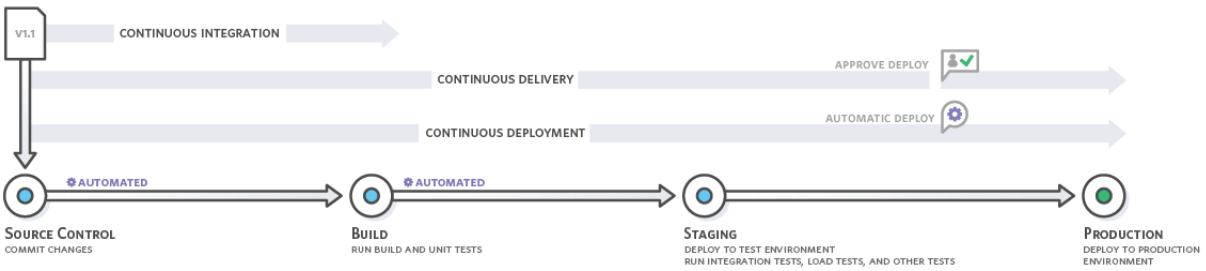

Figura 1. Apresentação dos passos da Integração Contínua [9]

A Entrega Contínua expande a Integração Contínua e está nos estágio de verificação de liberação do software a partir dos testes automatizados. A Implantação Contínua vai além e tem a automação da *build*, testes e implantação [10].

Existem algumas práticas necessárias para que a Integração Contínua funcione de forma ideal, as quais devem ser seguidas pelos desenvolvedores. Conforme Duvall, Matyas e Glover [11], estas práticas são as seguintes:

- **Enviar ao repositório as alterações do código frequentemente:** essa ação facilita a integração e permite que a equipe mantenha seus códigos atualizados com as últimas mudanças. Para isso, realizar pequenas mudanças sem envolver vários componentes e partes do software, é uma boa prática, sempre enviando ao repositório cada uma dessas pequenas alterações.

- **Não enviar ao repositório código que não funciona:** a boa prática nesse caso é realizar testes durante o desenvolvimento do código antes de enviá-lo ao repositório.

- **Ajustar erros na *build* imediatamente:** toda vez que a *build* principal demonstrar algum tipo de problema como erro de compilação, teste ou inspeção com falha, problema com o banco de dados, ou uma implementação errada, o conserto desse problema deve se tornar prioridade pela equipe. Conforme Fowler [6], não é necessário que toda a equipe se dedique ao problema, apenas a quantidade de pessoas suficientes para resolvê-lo. O importante é que se aja com a consciência de que a priorização nesse caso é necessária.

- **Escrever testes automatizados:** testes para um sistema de Integração Contínua devem ser automatizados.

- **Todos os testes e as inspeções devem passar:** para que a *build* seja aprovada é necessário que os testes automatizados não apresentem problemas.

- **Executar *builds* privadas:** como forma de prevenir *builds* com erros, os desenvolvedores devem simular uma integração local depois de realizar os testes unitários. Localmente é feita uma integração a partir da obtenção das mudanças dos outros

desenvolvedores que estão disponíveis no repositório, antecipando os possíveis problemas, o que torna mais provável que a integração não falhe.

- **Evitar integrar códigos falhos:** se a *build* disponível tiver algum erro é melhor esperar que seja consertado ou ajudar nessa tarefa pois utilizar essa *build* fará com que se perca tempo.

Na aplicação da Integração Contínua no DSSAT algumas dessas práticas foram utilizadas: a escrita de testes automatizados é atendida por meio do RCroptest, API responsável pelos testes do DSSAT. A automatização dos testes acontece a partir da utilização de ferramentas de CI, também responsável por realizar a execução de *builds* privadas cada momento que o desenvolvedor realiza um *push*, essa característica colabora também com que o desenvolvedor evite requisitar integração de códifgos falhos. Além disso o administrador do DSSAT consegue visualizar automaticamente no momento que recebe um pedido de integração se o código está apto ou não a ser integrado.

A Integração Contínua causa uma necessária mudança no padrão de desenvolvimento e exige disciplina da equipe para seguir essas rotinas, importantes para um bom funcionamento e andamento dos projetos [6].

Existem algumas diferenças quando o código é integrado de forma tradicional ou quando é feita por Integração Contínua. Essas diferenças podem ser visualizadas na Tabela 1. Com a utilização da Integração Contínua os desenvolvedores trabalham em apenas um pequeno requisito, cada nova funcionalidade é testada e corrigida antes de ser integrada e dedicam menos tempo com documentação. Os testes das alterações para integração são automatizados, sendo necessário investir em servidor e em uma ferramenta de Integração Contínua.

Com base nas informações contidas na Tabela 1 é possível observar que projetos que adotam a CI costumam reduzir o tempo de desenvolvimento e, também, o tempo de identificação e ajuste de possíveis falhas [7].

2.2 FERRAMENTAS PARA CI

Algumas opções diversas de ferramentas para Integração Contínua estão disponíveis no mercado. Cada ferramenta tem particularidades que podem se adequar às necessidades do projeto que está sendo executado.

Com o objetivo de visualizar com mais clareza as características das ferramentas de Integração Contínua mais utilizadas, uma tabela comparativa (Tabela 2) foi construída demonstrando quais características cada ferramenta oferece. As características avaliadas foram as seguintes:

Tabela 1. Diferenças entre o processo de integração tradicional e a Integração Contínua (Adaptada de [7]).

Critério de comparação	Antes de Integração Contínua	Depois de Integração Contínua
Tempo de desenvolvimento	Desenvolvedores trabalham em todos os requisitos do novo recurso	Desenvolvedores trabalham em apenas um requisito
Erros	Como o teste é feito somente depois de terminar uma grande parte do código, o número de erros é maior do que quando o teste é feito após a implementação apenas de pequenas partes do código.	Todas as funcionalidades são testadas e corrigidas depois de concluídas e não precisam esperar até que a versão completa seja concluída.
Tempo de entrega	Depois de terminar o novo recurso e testá-lo	Depois de terminar cada funcionalidade e testá-la
Qualidade de teste	O teste é feito para os lançamentos como um todo. Ambiente de teste não é o mesmo que o ambiente de produção	O teste de um recurso é feito assim que é integrado. Pequenas funcionalidades são testadas individualmente e os resultados dos testes são enviados imediatamente de volta para os desenvolvedores. O ambiente de teste é quase exatamente o mesmo que o ambiente de produção
Documentação	O novo recurso está documentado corretamente A especificação do recurso é documentada antes de iniciar o trabalho. E o trabalho de desenvolvimento é documentado em detalhes após o término da mudança	Os requisitos são documentados, mas há uma quantidade mínima de documentos produzidos pelos desenvolvedores. As ferramentas automáticas geram estatísticas e dados sobre os recursos desenvolvidos com base nos requisitos, nos resultados dos testes e nos erros gerados, etc.
Alteração de gerenciamento	A mudança só é aceita após um longo processo e aprovações. As alterações são introduzidas através de um novo lançamento	A mudança é aceita a qualquer momento apenas pela adição de novos requisitos pelo proprietário às equipes de análise de negócios e desenvolvimento de software
Custo do modelo	O tempo e o esforço de fazer manualmente a criação e a integração de software	O custo de obter servidor e ferramentas de Integração Contínua

- **Custo/Preço:** fornece as funcionalidades da ferramenta gratuitamente para projetos de código aberto.
- **Integração com GitHub:** possui integração com o repositório Github.
- **Suporte a várias linguagens:** ferramenta que pode ser utilizado para várias linguagens diferentes.

- ***Build Matrix*:** ferramenta que possibilita executar testes com diferentes versões de linguagens e pacotes, personalizando de maneiras diferentes [12]. Por exemplo, as falhas de alguns ambientes podem acionar notificações, mas não causam falha em toda a *build*. É chamada de matriz pois combina as três principais configurações; tempo de execução, ambiente e inclusões/exclusões [2].
- **Testes em paralelo:** processo de executar vários casos de teste em várias combinações de sistemas operacionais e navegadores ao mesmo tempo. O processo é automatizado e, geralmente, executado em máquinas virtuais.
- **Baseado em nuvem:** não é necessário servidor dedicado.
- **Plugin *system*:** possibilidade de adicionar plugins para a customização da Integração Contínua aplicada às necessidades do projeto.

Com base nestas características, na Tabela 2 são apresentadas algumas ferramentas de Integração Contínua disponíveis e suas principais características. Foram escolhidas para comparação ferramentas que se adaptassem a diferentes linguagens de programação e ferramentas de desenvolvimento e que tivessem integração com a plataforma GitHub. As ferramenas selecionadas foram: Travis CI[3], Circle CI[4], Jenkins[5], AppVeyor[6], Semaphore[7], Solano[8] e Code Fresh[9].

As ferramentas escolhidas que serão descritas na seção 2.3 foram: Travis CI, CircleCI e Jenkins. As ferramentas selecionadas podem ser utilizadas junto ao Github e ao Docker[10]. Elas oferecem gratuidade quando o repositório é privado, como é o caso do repositório do DSSAT no Github. Além disso, não possuem um limite no número de integrações que prejudique a execução dos testes deste trabalho aplicados ao DSSAT.

2.3 FERRAMENTAS SELECIONADAS

Nesta seção as ferramentas Travis CI, CircleCi e Jenkins, selecionadas para serem as ferramentas utilizadas para a implantação da CI na suíte DSSAT, são descritas de forma mais detalhada.

[3] https://travis-ci.org/
[4] https://circleci.com/
[5] https://jenkins.io/
[6] https://www.appveyor.com/
[7] https://semaphoreci.com/
[8] https://www.solanolabs.com/
[9] https://codefresh.io/
[10] O Docker cria contêiners, que são pacotes de software em unidades padronizadas com todos os recursos computacionais necessários para que o código possa ser executado.

Tabela 2. Comparação entre ferramentas de Integração Contínua. As ferramentas que possuem a característica estão marcadas com o símbolo ✓.

	Travis CI	Circle CI	Jenkins	AppVeyor	Semaphore	Solano	Codefresh
Custo (gratuito)	✓	✓	✓	✓	✓	-	-
Integração com GitHub	✓	✓	✓	✓	✓	✓	✓
Suporte a várias linguagens	✓	✓	✓	✓	✓	✓	✓
Build Matrix	✓	-	-	✓	-	✓	-
Testes em paralelo	✓	✓	✓	✓	✓	✓	✓
Baseado em nuvem	✓	✓	-	✓	✓	✓	✓
Compatível com Docker	✓	✓	✓	✓	✓	✓	✓
Plugin system	-	-	✓	-	-	-	✓

2.3.1 Travis CI

O Travis CI, de acordo com a documentação fornecida pelos responsáveis em seu website, é uma plataforma de Integração Contínua gratuita para projetos de código fonte aberto. Ele dá suporte ao processo de desenvolvimento, construindo e testando automaticamente alterações de código e fornecendo retorno imediato sobre as alterações. O Travis CI também pode automatizar outras partes do processo de desenvolvimento, gerenciando implantações e notificações.

Quando uma compilação de código é executada, o Travis CI clona o repositório do GitHub em um novo ambiente virtual e executa uma série de tarefas para criar e testar o código. Se uma ou mais tarefas falharem, a compilação é considerada interrompida. Se nenhuma das tarefas falhar, a compilação é aprovada e o Travis CI pode implantar o código em um servidor Web ou em um host de aplicativo.

Um ambiente de compilação e um conjunto de etapas padrão para cada linguagem de programação são fornecidos pela ferramenta Travis CI. Apesar disso, a personalização é possível utilizando a adaptação do arquivo YAML em `.travis.yml`. Este arquivo fica na raiz do repositório e pode ser minimalista ou ter muita customização. Alguns exemplos do tipo de informação que o arquivo `.travis.yml` pode ter:

- Linguagem de programação que o projeto utiliza;
- Comandos ou *scripts* desejados para executar antes de cada *build* como, por exemplo, para instalar ou clonar as dependências do projeto;
- Comando usado para executar o conjunto de testes;
- E-mails para notificação sobre falhas de build.

Na Figura 2 é apresentado um exemplo de como o arquivo `.travis.yml` é construído. Nesse caso está contido no arquivo a especificação do sistema operacional, comando de instalação de dependência e também uma linha de `script`. Nesse arquivo é possível observar a divisão por fases, `before_install`, por exemplo é a fase antes do início da instalações das dependências, por exemplo.

O Travis CI oferece alguns ambientes de infraestrutura prontos que podem ser utilizados, são eles: Ubuntu Linux, que roda em máquina virtual; macOS: pode ser utilizada uma das várias versões desse sistema operacional, não necessariamente precisa ser utilizado quando se desenvolve nesse sistema, e sim quando seus projetos necessitam ser testados nesse ambiente; e, finalmente, o Windows Server na versão 1803.

Um container Docker também pode ser utilizado como ambiente para realização de *builds* por meio do Travis CI, rodando uma imagem do Docker já existente. Também é possível modificar e enviar imagens novas ao repositório do Docker de acordo com as exigências do projeto e de suas transformações.

Figura 2. Exemplo de arquivo `.travis.yml`

O Travis CI possibilita a criação de *Build Matrix* que são baseadas em opções de configuração como versão, ambiente e exclusão/inclusão e ainda é possível adicionar o sistema operacional. Cada configuração possui uma lista de possibilidades que, combinadas entre si, expandem-se para um número maior de *jobs*[11] individuais.

Há uma extensa lista de linguagens de programação compatíveis para uso com o Travis CI na documentação[12] da ferramenta, entre elas estão: C, Go, Java, Python, R, entre outras. Cada uma dessas linguagens possui uma documentação detalhando as formas

[11] Processo automatizado que clona o repositório em um ambiente virtual e, em seguida, executa uma série de etapas, como compilar o código, executar testes etc.

[12] https://docs.travis-ci.com/

de configuração que podem ser aplicadas ao arquivo `.travis.yml`, tais como definição de versões, gerenciamento de dependências, compiladores, configurações, entre outros, além de apresentar modelos de arquivos para exemplificação.

2.3.2 CircleCI

O CircleCI, de acordo com documentação fornecida pelo seu website, é uma opção de ferrameta de Integração Contínua baseado em nuvem, em que não é necessário um servidor dedicado e não é preciso administrá-lo. No entanto, ele também oferece uma solução local que permite executá-lo em sua nuvem privada ou *data center*. A ferramenta possui um plano gratuito, até mesmo para fins comerciais.

A ferramenta possibilita a automatização dos testes em máquinas virtuais ou em unidades padronizadas chamadas de contêineres que possuem tudo o que o software precisa para executar, como o código fonte, as bibliotecas e as ferramentas de sistema.

Uma lista pequena de linguagens é apresentada na documentação do CircleCI, são elas: Clojur, Elixir, Go, Java, JavaScript, PHP, Python, Ruby e Android. Entretanto, isso não significa que outras linguagens não podem ser utilizadas, porém essas são as que possuem os guias no site da ferramenta cada uma com suas especificações.

A Integração Contínua utilizando CircleCI funciona como um ciclo, como pode ser visualizado na Figura 3. Ela começa a partir da integração com o GitHub, GitHub Enterprise ou Bitbucket e, para cada envio ao repositório remoto, o CircleCI cria uma nova *build* e aciona testes automatizados em um contêiner ou máquina virtual. Cada vez que algum problema com a *build* ocorre é emitida uma notificação de falha para que possa ser o quanto antes ajustada. Quando a *build* passa, ou seja, não apresenta erros ou problemas, a implantação é realizada de forma automática. O ciclo se repete toda vez que uma nova funcionalidade é enviada ao repositório remoto.

O CircleCI envia uma notificação por e-mail de sucesso ou falha após a conclusão dos testes. Além disso, comporta a inclusão de notificações integradas a ferramentas como Slack e IRC. O CircleCI pode ser configurado para implantar código em vários ambientes, incluindo AWS CodeDeploy, Serviço de Contêiner AWS EC2 (ECS), AWS S3, Google Kubernetes Engine (GKE), Microsoft Azure e Heroku.

As configurações do CircleCI podem ser realizadas em um arquivo YAML chamado `config.yml`. Existem sete seções principais no arquivo que correspondem às fases do processo de integração *Build*-Teste-Implantação, que são as seguintes:

- **Máquina:** ajusta o comportamento da máquina virtual;
- ***Checkout**:* clonar código de um repositório;
- **Dependências:** instalar as dependências específicas da linguagem do projeto;

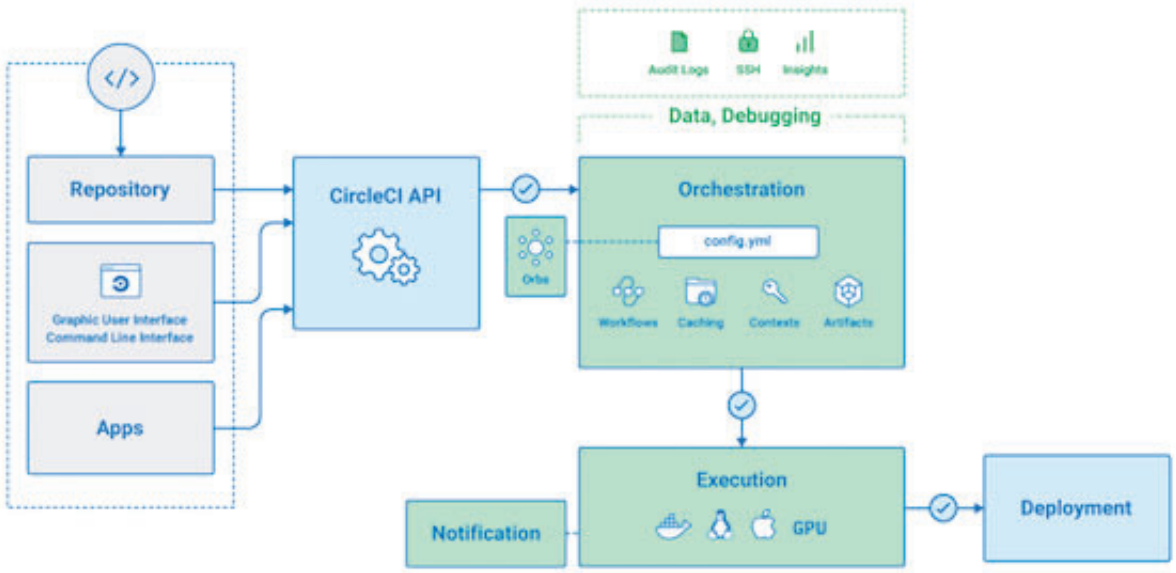

Figura 3. Visão geral do funcionamento da ferramenta CircleCI.

- **Banco de dados:** preparar o banco de dados para os testes;
- **Compilar:** compilar o projeto;
- **Teste:** executar os testes;
- **Implantação:** implantar o código nos servidores web.

Na Figura 4 é apresentado um exemplo do arquivo `config.yml`. Nesse caso está configurado um fluxo de trabalho simultâneo em que os *jobs* de compilação e de teste são executados simultaneamente. O exemplo também traz a utilização do Docker, conteiner previamente configurado com as dependência existentes no projeto.

O CircleCI precisa de poucas adaptações para começar a ser utilizado, normalmente é utilizado em projetos pequenos e que precisam de pouca customização. Desta forma ele também é uma boa opção para se começar a utilização de Integração Contínua rapidamente .

2.3.3 Jenkins

O Jenkins, de acordo com sua documentação, é um servidor de automação independente e de código aberto que pode ser usado para automatizar todos os tipos de tarefas relacionadas à criação, teste e distribuição ou implementação de software. Ele pode ser utilizado como um servidor de Integração Contínua simples ou ser transformado em um repositório de Entrega Contínua para qualquer projeto.

```yaml
version: 2
jobs:
  build:
    docker:
      - image: circleci/<language>:<version TAG>
    steps:
      - checkout
      - run: <command>
  test:
    docker:
      - image: circleci/<language>:<version TAG>
    steps:
      - checkout
      - run: <command>
workflows:
  version: 2
  build_and_test:
    jobs:
      - build
      - test
```

Figura 4. Exemplo de arquivo `.circleci/config.yml`

O Jenkins pode ser instalado por meio de pacotes nativos do sistema, ou ser executado de forma independente por qualquer máquina, a partir de um programa baseado em Java, com pacotes para Windows, Mac OS X e diversos sistemas operacionais Unix. Para ser utilizado localmente o Jenkins necessita do Java Runtime Enviroment (JRE) ou do Java Development Kit (JDK) instalado, e, também, do Docker. Algumas das linguagens de programação suportadas pelo Jenkins são: Android, C/C++, Java, PHP, Python e Ruby.

Há centenas de plugins disponíveis para utilização com o Jenkins, fazendo essa ferramenta ser altamente customizável, e fornecendo possibilidades múltiplas para o que o Jenkins pode fazer. Além disso, a ferramenta integra-se a praticamente toda a cadeia de ferramentas de Integração Contínua e Entrega Contínua.

Existem diversas possibilidades de customização para o Jenkins por meio dos *plugins*: realizar análise dos dados gerados pelo uso da ferramenta, integrar com plataformas diferentes das oferecidas por padrão e apresentar diferentes formas de interface, de administração e de gerenciamento de código fonte e de *build*. Caso o plugin não seja encontrado já pronto, é possível desenvolver um para as necessidades específicas do projeto, a documentação para esse desenvolvimento está disponível.

O conjunto de *plugins* utilizado para implementação e pipeline de entrega contínua é chamado de Jenkins pipeline ou apenas pipeline. Um pipeline de entrega contínua é uma expressão que significa automatizar o processo de levar o controle de versão de software até os usuários e clientes.

O Jenkins Pipeline fornece um conjunto de ferramentas para criar pipelines de entrega simples a complexas em formato de código. Um Jenkins Pipeline geralmente é gravado em um arquivo, chamado `Jenkinsfile`, como pode ser visualizado na Figura 5. Este arquivo deve estar na raiz do repositório do projeto em questão.

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent { docker { image 'maven:3.3.3' } }
    stages {
        stage('build') {
            steps {
                sh 'mvn --version'
            }
        }
    }
}
```

Figura 5. Exemplo de arquivo `Jenkinsfile` para a criação de um Jenkins Pipeline.

Na criação de um `Jenkinsfile` alguns benefícios são obtidos: criação de um processo de *build* para todos os ramos a cada envio de código, revisão de código no Pipeline, relatório detalhado dos comandos e resultados do processo e do `Jenkinsfile` único, que pode ser visualizado e editado por vários membros do projeto.

Muito semelhante ao Pipeline, há a possibilidade de criação de um projeto *Multibranch Pipeline*. Ele é voltado para descobrir, gerenciar e executar automaticamente Pipelines de diferentes *branches* de um repositório que contém um *Jenkinsfile* na raiz. Quando utilizada, essa opção identifica quando qualquer *branch* envia um código via *push* ou *pull request* e dispara a realização do *job*.

2.4 REVISÃO DE LITERATURA SOBRE CI

Muitas aplicações tem utilizado a Integração Contínua com a ideia de facilitar o desenvolvimento, a integração de códigos, os testes e as entregas realizadas. A Integração Contínua pode ser utilizada em diversas aplicações como na indústria, robótica, entre outros, apresentando algumas peculiaridades em cada um dos casos. Alguns trabalhos relacionados a aplicações de Integração Contínua são apresentados a seguir.

O estudo realizado por Martensson et al. [13] é resultado de uma revisão de 74 trabalhos de pesquisa e quatro livros relacionados a práticas de Integração Contínua· Foram realizadas 20 entrevistas com desenvolvedores que implementam sistemas de escala intensiva, sobre os impedimentos da Integração Contínua. Puderam ser elencados 12 fatores que afetam a frequência com que os desenvolvedores fazem *commit* dos seus códigos, agrupados em quatro grupos: planejamento de atividades e execução, pensamento do sistema, velocidade, e confiança através de atividades de teste.

Nas entrevistas alguns pontos importantes foram destacados pelos entrevistados como, por exemplo, se há sentido entregar funções pela metade quando se trata de um sistema de grande porte como os da indústria e como ficaria a documentação e os testes de complexidade maior, além da adaptação do usuário. Na mesma linha, outro entrevistado comenta sobre a necessidade de se ter uma arquitetura modular, fracamente acoplada com o objetivo de facilitar ter o sistema em partes e trabalhar em paralelo. Porém eles não deixaram de atentar sobre a importância de se pensar o sistema como um todo, e a necessidade de confiabilidade em ambientes de teste.

Outro caso de estudo destacado por Martensson et al. [13] foi referente a criação do modelo EMFIS, com o qual as empresas explicam a situação atual da organização em relação a impedimentos de Integração Contínua, e podem visualizar no que a organização deve focar para permitir uma integração mais frequente de software. O modelo é utilizado para realizar uma avaliação dos doze fatores elencados nas entrevistas como impedimentos para o *commit* frequente. O EMFIS foi validado em workshops e entrevistas, totalizando 46 pessoas em cinco empresas de estudo de caso. O modelo foi bem recebido durante a validação e foi destacado por sua simplicidade.

A aplicação de Integração Contínua também foi estudada por Mossige et al. [14]. Eles utilizaram em seus estudos a Integração Contínua focada em teste baseado em restrição. O objetivo do estudo foi automatizar os testes nos CIRs (*Complex industrial robots*), ou ainda robôs complexos industriais, normalmente feitos manualmente. Sistemas como esse possuem alta complexidade para serem testados e validados, já que portam uma quantidade grande de tecnologia e componentes. A automatização pode facilitar os testes, agilizar o processo de entrega, baixar custo e garantir uma maior segurança. O objetivo foi definir uma técnica de testes automatizados com boa relação custo-benefício para validar sistemas de CIRs, em um contexto industrial.

O trabalho de Mossige, Gotlieb e Meling [14] relata uma metodologia, desenvolvida na ABB Robotics em colaboração com a SIMULA, para testes totalmente automatizados em sistemas de controle de CIRs. A abordagem fundamentou-se em princípios de Integração Contínua e técnicas de testes baseadas em restrições bem estabelecidas. Em um ambiente industrial real, os resultados mostram que a metodologia é capaz de detectar falhas já conhecidas, e também falhas completamente novas. O modelo matemático de restrição utilizado juntamente às técnicas de Integração Contínua mostrou-se mais rápido e eficiente.

No trabalho de Kauss et al. [15] são apresentadas as descobertas provenientes de duas rodadas de entrevistas com um fabricante de automóveis sobre o uso de ferramentas na engenharia de softwares e como essas ferramentas suportariam uma adoção mais ampla de Integração Contínua. O domínio automotivo é desafiador quando se trata de CI pois, para permitir tempos de ciclo mais rápidos, algumas barreiras precisam ser superadas para suportar a colaboração interfuncional. Isso implicará grandes mudanças no conjunto de ferramentas atual. As necessidades em relação às ferramentas existentes,

como a escalabilidade, modelos e engenharia dirigida por modelos são importantes para o desenvolvimento de software automotivo, e na Integração Contínua esses modelos precisam ser integrados no nível do sistema, reunindo conhecimento sobre software, aplicativo, hardware/software básico e mecânica.

No estudo de Laukkanen et al. [16] é descrito como a introdução da Integração Contínua foi percebida por diferentes partes interessadas em um programa de P&D na Ericsson. Foram entrevistadas 27 interessados e diferentes percepções de Integração Contínua puderam ser percebidas: a adequação para a organização, como a adesão deve ser organizada e se é possível obter qualidade suficiente através de testes automatizados. Essas diferenças de percepção foram principalmente conseqüências da distribuição geográfica. Com base no estudo de caso, três diretrizes puderam ser propostas: a arquitetura do produto tem um efeito significativo na utilização, sem permitir que os problemas arquiteturais impeçam a implementação da Integração Contínua; dar aos membros da equipe tempo suficiente para superar a fase inicial de aprendizado no início do uso, e evitar centralizar as competências, investindo na comunicação.

Com base em dados quantitativos de seis casos, abrangendo cerca de 2.000 engenheiros, complementados por entrevistas com engenheiros de cinco empresas, no trabalho de Stahl et al. [17] é vista uma investigação referente a correlação entre a continuidade da Integração Contínua e o tamanho do setor. Descobriu-se que não apenas existe uma correlação entre o tamanho e a composição de uma organização de desenvolvimento e sua tendência a se integrar continuamente, como há evidências de que o tamanho da organização influencia nas formas de trabalhar, que por sua vez se correlacionam com o grau de continuidade. Observa-se que o comportamento do desenvolvedor em casos de integração ostensivamente contínua não corresponde necessariamente às expectativas, e que a integração frequente do produto em si não implica automaticamente que cada desenvolvedor individual realize *commits* com frequência.

Um estudo qualitativo das barreiras e necessidades que os desenvolvedores enfrentam ao utilizar Integração Contínua foi utilizado por Hilton et al. [18]. Foram realizadas entrevistas semiestruturadas com desenvolvedores de diferentes indústrias e escalas de desenvolvimento. Chegou-se a conclusão de que os desenvolvedores enfrentam conflitos de escolha entre rapidez ou certeza, entre melhor acesso ou segurança da informação, e entre mais opções de configuração ou maior facilidade de uso. Além disso, são apresentadas as implicações deste fato para desenvolvedores, construtores de ferramentas e pesquisadores.

As ferramentas de análise estática são frequentemente usadas por desenvolvedores de software para detectar precocemente possíveis falhas, vulnerabilidades ou para avaliar a aderência do código-fonte aos padrões e diretrizes de codificação. Além disso, a adoção delas dentro da Integração Contínua tem sido defendido por pesquisadores e praticantes. O artigo de Zampetti et al. [19] estuda o uso de ferramentas de análise estática

em 20 projetos Java de código aberto hospedados no GitHub e usando o Travis CI como infraestrutura de Integração Contínua. Foram investigadas as ferramentas que estão sendo usadas e como elas são configuradas para a Integração Contínua, que tipos de problemas fazem com que a compilação falhe ou aumente os alertas, e como e após quanto tempo são compilados os *builds* com problemas ou alertas. Os resultados indicaram que os erros na *build* devido a ferramentas de análise estática estão relacionados principalmente à aderência aos padrões de codificação, e também pela falta de licenças. Falhas na *build* relacionadas a ferramentas que identificam potenciais defeitos ou vulnerabilidades ocorrem com menos frequência, em alguns casos essas ferramentas são ativadas de modo mais ameno, sem fazer a compilação falhar. Além disso, o estudo revela que falhas na *build* devido a ferramentas de análise estática são rapidamente corrigidas por resolver o problema ao invés de desativar o alerta e, geralmente, são documentadas corretamente.

Vöst [20] realizou um trabalho para entender como a Integração Contínua pode ser aplicada à indústria automotiva em nível de sistema. Ele iniciou o estudo para chegar a um processo adaptado de Integração Contínua incluindo métodos para especificação e seleção de casos de teste. O trabalho foi aplicado como um projeto piloto em um ambiente de produção da BMW, para avaliar a eficácia e reunir dados qualitativos e quantitativos. A intenção foi realizar uma primeira abordagem dentro de um campo de pesquisa que precisa ser investigado mais profundamente, já que pouco se encontra sobre o assunto.

A Técnica de Priorização de Casos de Testes (TCP) em ambientes de Integração Conínua é abordada por Prado Lima e Vergílio [21]. Essa técnica objetiva identificar em qual ordem os testes precisam estar para maximizar objetivos específicos. A pesquisa foi realizada utilizando um mapeamento de estudos relacionados à TCP e às particulatidade da CI, considerando possíveis restrições de tempo e os problemas relacionados aos testes automatizados. A pesquisa demontrou que o assunto é pertinente ao momento, apesar de poucos fazerem alguma relação com a CI. Os estudos nessa área estão em expansão e trabalhos futuros devem considerar cada vez mais as particularidades de CI, problemas de teste como a volatilidade de casos de teste e restrição de tempo e que testes de falhas devem ser considerados para oferecer soluções econômicas e confiáveis para a indústria de software.

Shahin et al. [22] abordou as práticas contínuas que incluem: integração, entrega e implantação em uma revisão sistemática, já que uma maior quantidade de material sobre essas práticas tem sido publicado. O trabalho traz classificações de abordagens e de ferramentas, sintetizando os desafios e as práticas encontradas em CI. Foram identificadas trinta abordagens associadas a ferramentas, com o objetivo de facilitar a implantação de práticas contínuas, as quais são: redução do tempo da *buid* e dos testes na Integração Contínua; aumento da visibilidade e conscientização sobre os resultados da *buid* e dos testes na CI; suporte a testes contínuos (semi-)automatizados; identificação de violações e

falhas na CI; abordagem de problemas de segurança e escalabilidade na implantação de pipeline, e formas de melhorar a segurança e a confiabilidade da implantação do processo.

As práticas contínuas tornaram-se uma área de pesquisa e prática de engenharia de software bastante relevante [22]. Apesar de já existirem muitas abordagens, ferramentas e práticas que estão tratando uma ampla gama de desafios, ainda há outros vários a serem contemplados e lacunas que exigem trabalho de pesquisa, como, por exemplo, é necessário melhorar a comunicação e a captura de informações relativas a diferentes estudos que relatam aspectos de práticas contínuas; obter um entendimento apurado de como mudar arquiteturas de sistemas para comportar práticas contínuas; abordar a falta de conhecimento e de ferramentas que permeiam os processos de engenharia de projeto e a execução segura de rotinas de implantação.

O estudo de Beller et al. [23] foi estimulado por não existirem pesquisas suficientemente claras, segundo ele, que abordam fatores como os benefícios, custos e implicações de se implantar CI. Dessa forma, para estabelecer base para uma pesquisa orientada a dados sobre CI, e considerando que o Travis CI tem se mostrado a ferramenta de CI mais usada para o desenvolvimento de sistemas de código aberto, o TravisTorrent foi criado [13], que é um conjunto de dados baseado no Travis CI e no GitHub, que fornece acesso a centenas de milhares de *builds* já realizados. O objetivo na construção dessa plataforma a partir de conjunto de dados acessível de mais de mil projetos, facilitar uma pesquisa mais holística sobre Integração Contínua, oferecendo aos pesquisadores uma forma de realizar pesquisa em pilha, a partir de uma análise dos logs das *builds* dos repositórios.

O impacto da Integração Contínua nas equipe de trabalho utilizando a ferramenta Travis CI é estudado por Zhao et al. [24]. O estudo considera a adaptação na escrita e no envio de códigos, os aspectos relacionados ao impedimento de integrações e as práticas de testes. O número cresente de *pull request* alinhado com a instrução de realizar *commit* frequentemente, provavelmente encoraja uma mudança para uma distribuição maior do trabalho e com mais *branches*. Foi observado que, com o passar do tempo, existe uma tendência de aumentar o fechamento de requisições de integração, e o contrário é observado com a implantação do Travis CI. Outro ponto importante é que a implantação da ferramente faz aumentar a qualidade e a quantidade de testes automatizados realizados.

O artigo de Islam e Zibran [25] apresenta um estudo empírico para identificação de fatores como: complexidade de uma tarefa; estratégia para *build* e modelos de contribuição, como *push* e *pull* e características referentes ao projeto, como tamanhos de projetos e equipes. Esses fatores têm impactos grande nos resultados da *build*. Foram estudados mais de 3,6 milhões de *builds* em mais de 1.090 projetos de código aberto. O estudo realizado constatou que os resultados da *build* são afetados pelo número de linhas de código fonte alterado, quantidade de arquivos e de *commits* realizados durante o projeto. Por outro lado, o número de linhas de código referentes aos testes alterados, quantidade de

[13] travistorrent.testroots.org

branches, e modelos de *commit*, não possuem impactos significativos nos resultados das *builds*.

No trabalho de Krafczyk et al. [26] a Integração Contínua é abordada em um contexto de software científico. A CI pode aumentar a reprodutibilidade dos resultados científicos fornecendo uma plataforma estabelecida para testar essa característica e demonstrar a outros cientistas como o código e os dados geram os resultados publicados. Para este estudo foram utilizados dois artigos publicados nas áreas de matemática aplicada e física computacional. Foram discutidas as lições aprendidas na reprodução desses artigos, e examinados os testes existentes. A partir disso a ideia foi reproduzir os resultados em um ambiente de CI. Quando necessário um grande esforço para adaptação ao CI, recomendou-se a inclusão de resultados de versões reduzidas do trabalho em questão ao lado dos principais resultados do artigo. Como resultado o trabalho conclui que a aplicação da Integração Contínua pode colaborar para inferir não apenas se o software está funcionando corretamente, mesmo quando os autores estão fazendo alterações, mas também se os resultados são sensíveis ao ambiente de computação. Segundo Krafczyk et al. [26]. À medida que mais cientistas adotam essas alterações em seus fluxos de trabalho, são esperados um número maior de publicações verificadas e um subproduto mais confiável, fortalecendo o registro científico.

O estudo de Widder et al. [27] contribui com uma revisão da literatura sobre CI, incluido os pontos críticos do processo de integração. Nesse estudo foi realizada uma análise de 132 respostas da pesquisa sobre os motivos para não utilizar o Travis CI ou trocá-lo por outra ferramenta; modelos de regressão em um conjunto de dados de 6.239 projetos GitHub Travis CI, testes de observações existentes na literatura; e, por fim, uma discussão dos resultados e implicações. Os problemas encontrados em ordem de frequência são: tecnologia não suportada como, por exemplo, quando o Travis CI não atende a linguagem do projeto; lentidão para executar a *build*; consistência entre projetos devido a intenção de manter todos os projetos com a mesma tecnologia de integração; falta de testes necessários; pouca frequência de alterações; interface do usuário e documentação insuficientes; ambiente de testes de natureza proprietária e por fim por não conseguir ou achar muito difícil consertar uma falha na *build*.

A aplicação da CI no DSSAT se assemelha com alguns aspectos observados nos trabalhos relacionados. A automatização dos testes nos CIRs, demonstrados no trabalho de Mossige, Gotlieb e Meling [14], pode facilitar os testes e garantir uma maior segurança, como é o cenário do DSSAT. Islam e Zibran [25] atentam ao fato de que os resultados da *build* são afetados tanto pelo número de linhas alterados como pela quantidade de arquivos modificados, fato que provavelmente será observado também na CI aplicada ao DSSAT. A tendência observada por Zhao et al. [24] de uma distribuição maior dos projetos e em mais branches pode se tornar uma característica do DSSAT, após a real aplicação da CI.

Alguns estudos relatam desafios na utilização da CI em diferentes ambientes. No estudo de Kauss et al. [15], por exemplo, a fabricante de automóvel, não tem uma estrutura colaborativa interfuncional, de forma que as integrações entre ferramentas e modelos se tornam mais complexas. Outras pesquisas como a de Laukkanen et al. [16], Stahl et al. [17], Hilton et al. [18] relatam a influência da automatização dos testes na confiabilidade dos sistemas e o comportamento do desenvolvedor relativo a frequência de *commits*. No caso do DSSAT é diferente, a CI automatiza um processo que já é realizado tanto pelos desenvolvedores quanto pelos administradores do DSSAT, além disso, muitas rodadas de testes mais intensivos são realizados até que o DSSAT chegue em nova versão oficial.

3. METODOLOGIA

Neste capítulo é abordada a metodologia utilizada para implantar a integração contínua na suite de modelos de simulação de culturas DSSAT, considerando a escolha das ferramentas e o ambiente tecnológico.

3.1 A SUITE DSSAT E O SEU MÓDULO CROPTEST

O *Decision Suport System for Agrotechnology Transfer* (DSSAT) [3] ou ”Sistema de Suporte para Decisão e Transferência em Agrotecnologia“) realiza a simulação de modelos de mais de 42 culturas como soja, milho, trigo e sorgo, entre outras. O objetivo é simular o crescimento, o desenvolvimento e o rendimento em função da dinâmica solo-planta-atmosfera. O DSSAT incorpora ferramentas que facilitam o uso desses modelos, entre eles programas de gerenciamento de banco de dados para solo, clima, culturas e dados experimentais, utilitários e aplicativos.

O uso do DSSAT é bastante amplo, mais de 14 mil pesquisadores, educadores, consutores, agentes de extensão, produtores, políticos e tomadores de decisão distribuídos em mais de 150 países. Utilizado nas fazendas, em gerenciamento de precisão, avaliações regionais do impacto da variabilidade climática e das mudanças climáticas, modelagem baseada em gene e em seleção de reprodução, uso de água, emissão de gases de efeito estufa e sustentabilidade a longo prazo através dos equilíbrios de carbono orgânico e nitrogênio no solo, entre outras várias aplicações importantes.

As opções de cultura, de solo, de dados climáticos e de gerenciamento permitem ao usuário fazer diversos tipos de combinações que geram uma simulação de experimento virtual sem ser necessário que se dedique um tempo significativo conduzindo experimentos reais. O DSSAT também permite aos usuários a comparação entre simulação de resultados com dados observados. A partir da simulação de prováveis resultados de estratégias de manejo de culturas, o DSSAT oferece aos usuários informações para avaliar rapidamente novas culturas, produtos e práticas a serem adotadas.

O DSSAT é a combinação de modelos de simulação de culturas (CSM do inglês *Cropping System Model*), sistemas de banco de dados e sistemas de suporte à decisão. Os CSMs são os programas de computador que simulam o crescimento e o rendimento das culturas, com base em cálculos previamente estabelecidos e dados de entrada das práticas de manejo do solo, clima e cultura. O DSSAT-CSM pode simular o rendimento de uma variedade de culturas e tem sido usado por muitos cientistas, tomadores de decisão e pesquisadores em todo o mundo há mais de duas décadas.

Na figura 6 é ilustrada a conexão entre os módulos primário e secundário do CSM. O programa principal controla todo o tempo do modelo, enquanto o módulo `Land Unit` é usado para controlar o processamento e a transferência de dados entre todos os módulos primários. Desde dezembro de 2019, o código do DSSAT CSM está aberto e disponível via GitHub em https://github.com/DSSAT/dssat-csm-os. Neste repositório foi omitido apenas o diretório Data, onde estão contidos os dados de simulação privados do DSSAT.

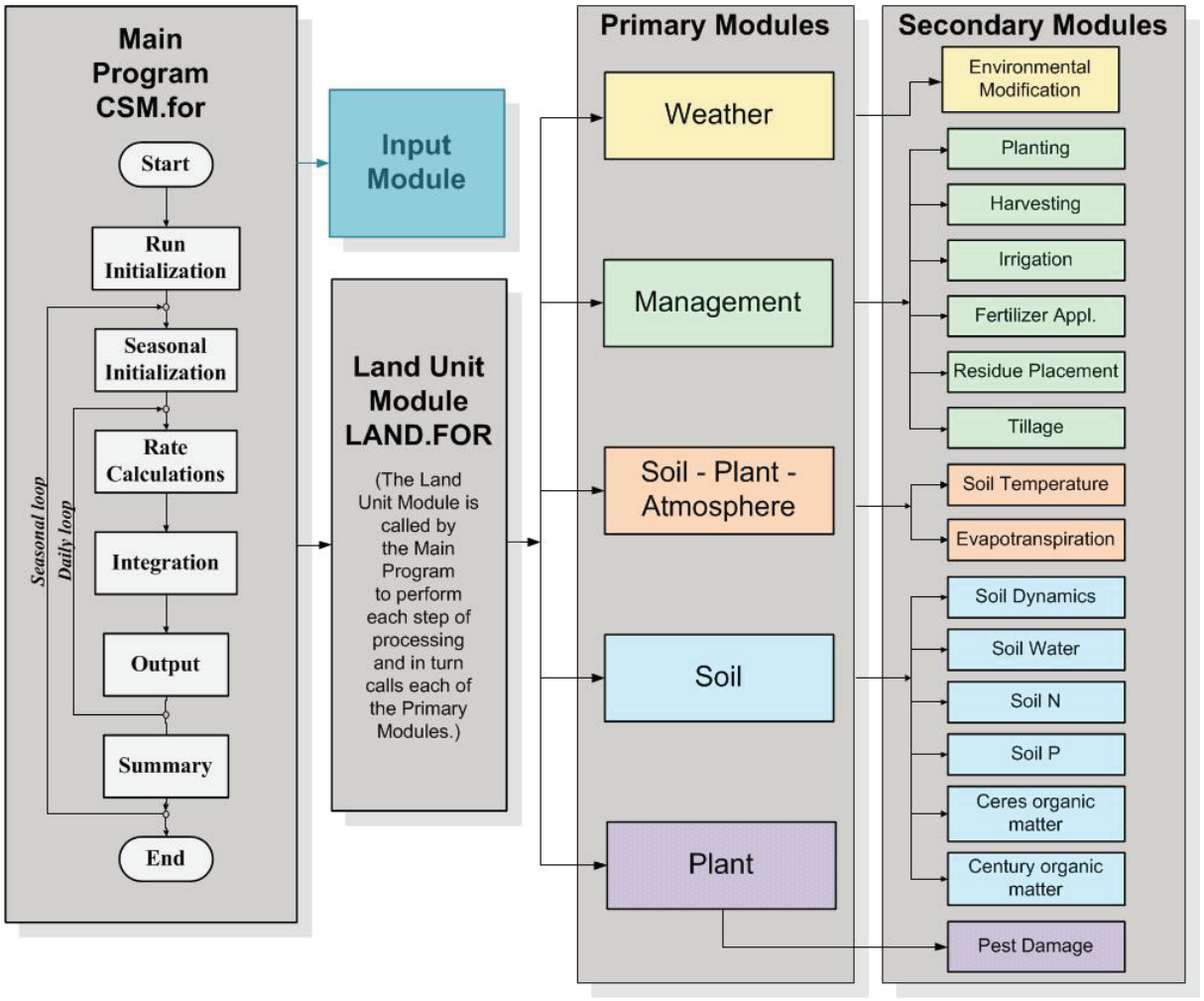

Figura 6. Estrutura modular do DSSAT CSM [4]

.

O CropTest (*Model Comparison Utility* ou Utilitário de Comparação de Modelos) do DSSAT foi desenvolvido com o propósito de comparar as saídas de modelos de culturas de duas versões diferentes do DSSAT. Ele é utilizado pelos administradores do sistema para realizar os testes e concluir se as alterações submetidas pelo desenvolvedor podem ser integradas ao *branch release*. Na sua versão atual, o CropTest recebe os arquivos de entrada, que são processados pelos DSSAT, e geram arquivos de saídas: `plantgrow.out`, `summary.out` e `evaluate.out`. Ele gera uma planilha de diferenças percentuais apontando aquilo que se manteve, foi melhorado ou piorado, para que, a partir disso, possa ser tomada

a decisão. Se os testes alcançaram bons resultados, as alterações podem ser incluídas ao *branch release*.

O código do DSSAT, para o desenvolvimento de novas versões, está compartilhado no Github para usuários autorizados, separado em três ramos, como pode ser visualizado na Figura 7: *master* - última versão atual e estável liberada para utilização, *release* - versão em teste, e *development* - onde os desenvolvedores devem fazer suas alterações. Quem tem acesso ao *development* são usuários cadastrados, porém não são todos que contribuem com a evolução do DSSAT.

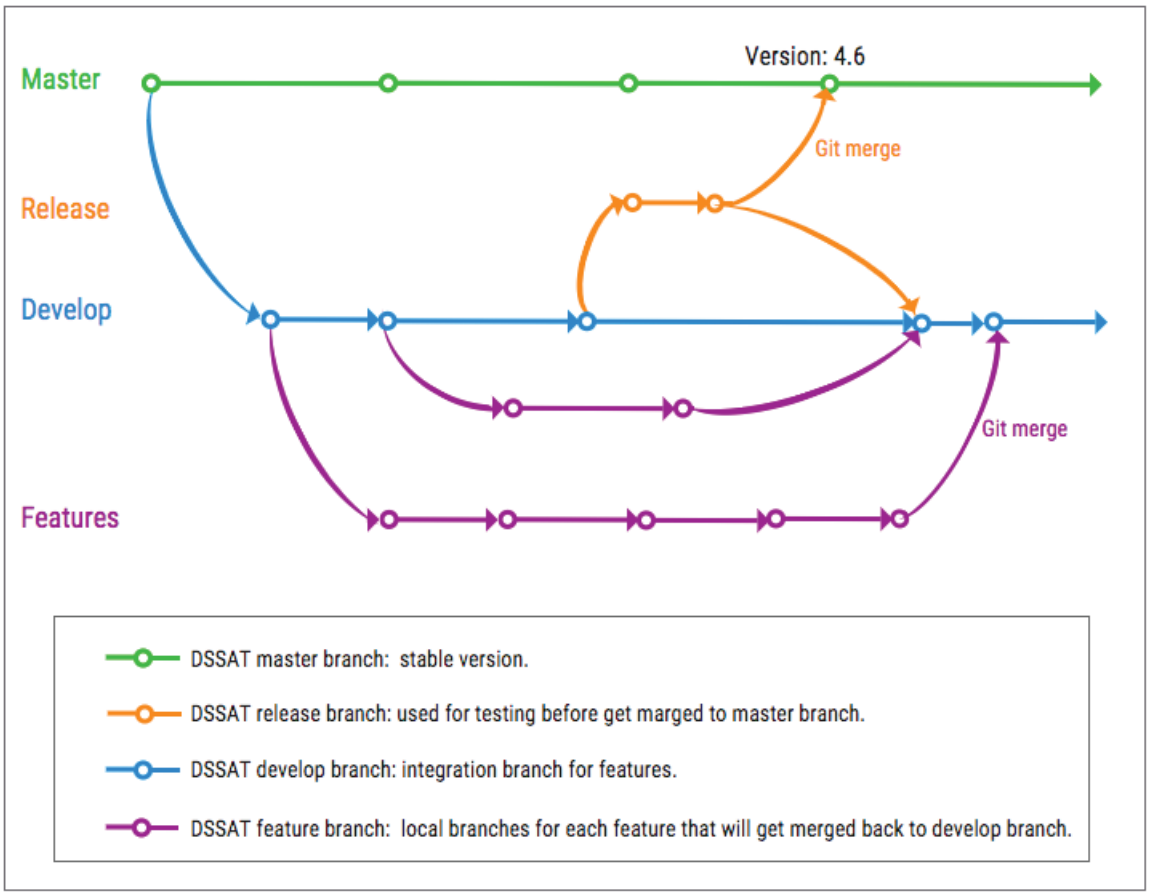

Figura 7. Estrutura de branches no repositório DSSAT

Os desenvolvedores que contribuem com o DSSAT estão em diversas partes do mundo, como o grupo de pesquisa Mosaico da Universidade de Passo Fundo ou pesquisadores da Universidade da Flórida. Cada um colabora para a evolução do DSSAT também pensando em como o DSSAT contribuirá para os objetivos de seus próprios projetos. Para tanto, são desenvolvidos novos modelos de culturas e melhorias em modelos existentes. No caso do Mosaico, o interesse está na melhoria do funcionamento do sistema, trazendo novas tecnologias ao sistema DSSAT como um todo. O Flexible I/O [28, 29] pode ser utilizado como exemplo pois ele é uma ferramenta que facilita a utilização de novas tecnologias ao DSSAT, sem qualquer alteração nos modelos. Outro trabalho desenvolvido pelo grupo

Mosaico foi o DSSAT [30], o qual disponibiliza um padrão e uma abordagem de reuso de software para leitura e processamento de arquivos do DSSAT.

Quando o desenvolvedor faz uma alteração, testa e aprova o resultado, inicia-se um processo longo de testes e aprovação dessas alterações, visando que elas possam ser incluídas nas novas versões do DSSAT. Para dar início ao processo, o desenvolvedor precisa requerir aos administradores do repositório a inclusão da alteração no ramo *develop*. Os administradores do DSSAT utilizam a ferramenta de testes Croptest para realizar a avaliação do que foi desenvolvido.

A partir dessa avaliação, os administradores conseguem decidir se o código está ou não apto a ser integrado, como pode ser visualizado na Figura 8. Toda vez que um *pull request* (1), ou seja, um pedido de integração de novos códigos, é feito inicia-se um longo processo. O administrador recebe a notificação do GitHub (2), baixa o código completo do desenvolvedor (3) e executa a compilação (4), que pode demorar alguns minutos. Quando a *build* roda e apresenta erros (5) o administrador rejeita o *pull request* (7). Caso contrário, quando a *build* não apresenta erros (8) o administrador parte para o processo de testes utilizando o Croptest (9). Neste momento são comparadas as diferenças dos resultados do DSSAT atualizado pelo desenvolvedor e o DSSAT em versão estável. Se o Croptest apresentar diferenças (6) o administrador rejeita o *pull request* (7), mas se não apresentar diferenças, o *pull request* pode ser aceito (10) e, finalmente, integrado ao DSSAT no *branch develop*.

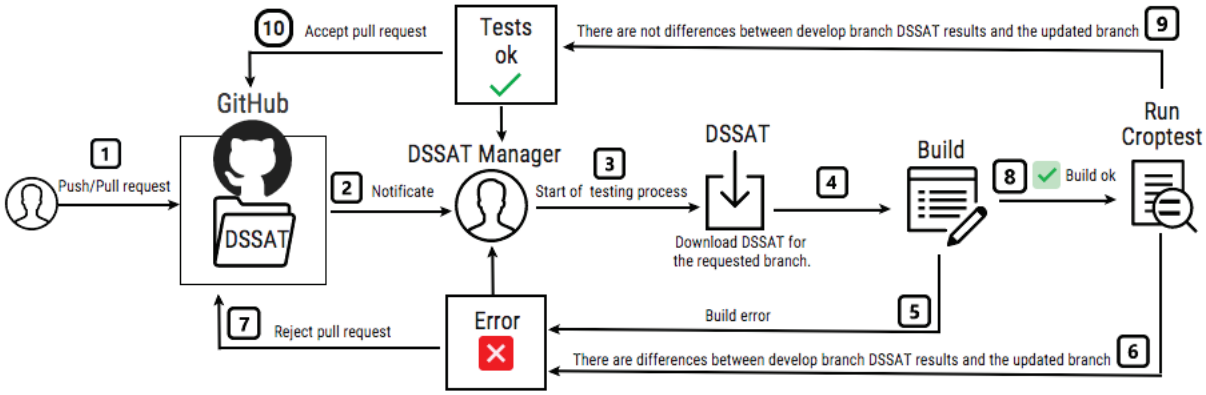

Figura 8. Fluxo de como são realizadas as integrações atualmente no DSSAT, com a desmonstração de cada passo realizado pelos administradores da suíte

Todo ano são realizados encontros nos Estados Unidos, chamados de *sprints*, em julho e janeiro, que fazem parte do processo para disponibilizar atualizações do DSSAT. Nas *sprints*, os desenvolvedores reúnem-se para codificação de melhorias para o `branch development`. No intervalo entre uma *sprint* e outra, são feitas as mini *sprints*, com um grupo de desenvolvedores menores. Neste momento são realizadas sequências de testes intensivas em cada um dos modelos, ajustando aquilo que está com problema. Cada participante fica responsável por testar uma série de modelos, sempre utilizando o Croptest.

O teste final é realizado em um workshop que acontece em maio e reúne pessoas interessadas em desenvolver para o DSSAT. São cinco dias de atividade e elas aprendem a utilizar e trabalham com a ferramenta intensivamente. Nessa ocasião, a nova versão do DSSAT aprovada na mini *sprint* é utilizada como versão beta. Caso durante o workshop não sejam identificados problemas, a nova versão é liberada para utilização em produção. Há uma tentativa de uma vez por ano ser disponibilizada uma nova versão do DSSAT.

3.2 ESCOLHA DAS FERRAMENTAS DE CI

Para esta pesquisa foram selecionadas e utilizadas três ferramentas para realizar a integração contínua do DSSAT. Para que se pudesse decidir quais seriam as ferramentas, foi realizado um estudo comparativo entre elas (descrito na Tabela 1) com o objetivo de identificar quais delas se adequavam melhor ao objetivo do projeto. Com base nesta análise, foram selecionadas as ferramentas Travis CI, Circle CI e Jenkins. Estas três ferramentas possuem integração com o Github e com o Docker, oferecem a possibilidade de serem utilizadas gratuitamente quando o repositório é privado, como é o caso do DSSAT, e, também, não possuem um limite de utilização que prejudique a realização dos testes de integração contínua aplicado ao DSSAT.

O processo manual de integração de códigos, atualmente utilizado pelos administradores do DSSAT, é trabalhoso e demorado. Existem muitos desenvolvedores pelo mundo criando novas funcionalidades e melhorias no DSSAT e, com frequência, são recebidos pedidos para integrar novos códigos ao *branch develop* do DSSAT no GitHub. Antes do *pull request* ser feito, o desenvolvedor faz sua própria análise do que desenvolveu, podendo ou não ter utilizado o Croptest para realização de testes, utiliza o seu próprio ambiente com suas instalações e realiza as suas configurações específicas.

Quando os administradores do DSSAT recebem o *pull request* é necessário que o código modificado seja baixado, compilado e testado, utilizando o Croptest. Isso é feito sempre para avaliar se as alterações podem ser integradas, processo que demanda tempo. Além disso, o desenvolvedor realiza seus testes individuais em ambiente diferente do utilizado pelos administradores, podendo gerar resultados diferentes.

O objetivo da automatização desse processo do DSSAT, a partir da Integração Contínua, a cada *commit*/*push* que o desenvolvedor realiza a compilação do novo código, os testes são realizados automaticamente, gerando um resultado. Em caso de erro, o desenvolvedor precisa trabalhar para corrigir, porque sabe que essa atualização não será aceita. Quando o resultado é positivo o desenvolvedor pode fazer o *pull request*. Dessa forma, as compilações e testes são realizados em ambientes iguais, o desenvolvedor consegue antecipar correções e os administradores recebem *pull requests* com menor possibilidade de rejeição.

3.3 AMBIENTE TECNOLÓGICO

Algumas ferramentas auxiliaram para o funcionamento da aplicação da integração contínua no DSSAT. Elas formaram a estrutura base para que todas as ferramentas de CI pudessem ser utilizadas. São elas: o GitHub, o Docker e o RCroptest que serão descritas a seguir.

O Github é utilizado atualmente como repositório remoto e para o controle de versionamento pelo DSSAT. O repositório é privado e são administrados três ramos principais: *develop*, *release* e *master*. O ramo *develop* é utilizado pelos desenvolvedores e é nele que são integradas as primeiras versões das novas funcionalidades e melhorias realizadas pelos desenvolvedores ao redor do mundo. Foi no ramo *develop* que foram dedicados os esforços para a implantação da integração contínua.

Foi feita uma cópia do repositório DSSAT no GitHub, que é chamada de *fork*, para que se pudesse realizar os testes de integração contínua em um repositório idêntico ao original, porém com acesso total. Dessa forma, foi possível modificar o repositório com as configurações necessárias ao funcionamento do projeto.

Com a finalidade de realizar os testes, os desenvolvedores podem utilizar o Croptest, ele é responsável por fazer comparações entre versões do DSSAT apontanto diferenças de resultados das simulações dos modelos. A Integração automática de uma nova funcionalidade depende dos resultados dos testes executados por meio dessa ferramenta. Para rodar os testes automaticamente foi utilizado o RCroptest [31], ele possui a mesma função que o Croptest mas é um pacote R e foi desenvolvido com a intenção de ser utilizado em um sistema de integração contínua, já que oferece a facilidade de operação por meio de linha de comando. O RCroptest também é um trabalho realizado por integrantes do grupo Mosaico da UPF.

Um script R para executar o RCroptest foi criado, conforme a Figura 9. Primeiramente o pacote RCroptest é carregado, A seguir, a função `testCI`, especialmente desenvolvida para ser utilizada no processo de Integração Contínua, é chamada (linhas 3 e 4). Nela, os dois primeiros parâmetros indicam o local dos arquivos `Summary.OUT` gerados ao rodar o DSSAT do *branch develop* e do *branch* a ser testado - elementos a serem comparados.

Para o procedimento de comparação, o algoritmo compara o primeiro modelo, chamado Modelo A, e a última versão proveniente das alterações do desenvolvedor, chamada de Modelo B. Calculando a diferença relativa entre variáveis numéricas e de data considerando a porcentagem de diferença aceitável passadas como parâmetro (`thNumeric`) e (`thDate`), respectivamente [31]. O procedimento calcula a diferença absoluta e o resultado dessa comparação é testado a partir da linha 6. Quando há diferenças, a execução é interrompida e uma mensagem indica que diferenças entre as versões dos modelos do DSSAT foram encontradas.

```r
library(RCroptest)

result = testCI('/DSSAT47/A/ci/Summary.OUT',
       '/DSSAT47/B/ci/Summary.OUT', thNumeric=2, thDate=0)

if (result == 0){
  result
}else{
  stop('There are differences between the models.')
}
```

Figura 9. *Script* R desenvolvido para realizar os testes do DSSAT

A construção de um container é a base do processo de automatização. O Docker é um container que possibilita a criação de ambientes específicos de acordo com a necessidade. Um Docker privado foi construído a partir da escrita de um `Dockerfile`, que pode ser visto na Figura 10. Na linha 2 da figura observa-se que o Docker foi criado a partir de uma imagem Linux com R instalado. A partir da linha 5 estão os comandos para instalação das dependências para compilar o DSSAT. E das linha 18 até 20 visualiza-se a instalação do pacote `RCroptest`, responsável pelos testes realizados no processo de Integração Contínua do DSSAT.

Dockerfile

```
# Install R version 3.5
FROM r-base:3.5.0

# Install Ubuntu packages
RUN apt-get update && apt-get install -y \
    sudo \
    gdebi-core \
    pandoc \
    pandoc-citeproc \
    libcurl4-gnutls-dev \
    libcairo2-dev/unstable \
    libxt-dev \
    libssl-dev \
    cmake \
    gfortran \
    git

RUN R -e "repos <- c('https://marcionicolau.github.io/RCroptestDeps', 'https://cloud.r-project.org') \
    options(repos = c(CRAN = repos)) \
    install.packages('RCroptest', dep=TRUE)"

RUN mkdir cidssat

COPY citest.R home/test/
COPY AllCrops.v47 DSSAT47/A/ci/
COPY AllCrops.v47 DSSAT47/B/ci/

CMD ["bash"]
```

Figura 10. `Dockerfile` utilizado para a construção da imagem do container utilizado pela CI.

O Dockerfile também cria os diretórios que: recebem o clone do repositório DSSAT, na linha 22 e armazenam o *script* R para execução dos testes no RCroptest, na linha 24. Além disso nas linhas 25 e 26 são incluídos os arquivos responsáveis por rodar todos os modelos do DSSAT: `AllCrops.v47`. Dessa forma, o container criado tem todas as configurações necessária para clonar via GitHub o ramo do DSSAT desejado, para compilar o DSSAT e, também, para testá-lo utilizando o Rcroptest.

O Docker, Github e RCroptest são utilizados igualmente pelas três ferramentas de Integração Contínua. O mesmo cenário é replicado para cada uma delas, porém todas possuem uma forma diferenciadas de serem configuradas.

A utilização do Travis CI inicia com o login na ferramenta online, `travis-ci.com/`, fazendo acesso com os dados da conta do GitHub. A partir do primeiro acesso já é possível visualizar uma listagem de repositórios que essa conta possui acesso. Para o Travis CI foi necessário obter uma conta via Github Education, para poder utilizá-lo em repositório privado de forma gratuita.

A chave do funcionamento do Travis CI é a criação de um arquivo YAML na raiz do repositório com o nome de: `.travis.yml`. Nesse arquivo ficam todos os passos necessários para realizar a compilação e testes do DSSAT. Os comandos são executados dentro do Docker previamente preparado com as dependências necessárias.

Existem algumas variáveis de ambiente que foram cadastradas nas configurações na página web do Travis e que são utilizadas no arquivo YAML. Duas variáveis de login: `GIT_USERNAME` e senha: `GIT_PASSWORD` para acessar o repositório GitHub privado do DSSAT, e mais duas variáveis com os acessos para o container privado no Docker - login: `DOCKER_USERNAME` e senha: `DOCKER_PASSWORD`.

Na Figura 11 pode-se observa os passos criados para compilação e testes do DSSAT. Na primeira linha, nenhuma linguagem foi escolhida, optou-se por utilizar linha de comando e como serviço utilizado está o Docker. Os comandos estão divididos em três partes no arquivo:

- `before_script` (linha 11): Nesse momento é feito o login no Docker (linha 13), utilizando as variáveis de ambiente previamente configuradas. Em seguida, com o comando `docker run` linha (14), a imagem docker, preparada para compilar e testar o DSSAT, é executada.

- `script` (linha 17): Durante essa fase são realizadas duas compilações de versões de DSSAT diferentes. Primeiramente é feito um clone do repositório do DSSAT a partir dos comandos do GitHub como pode ser observado na linha 19. Para garantir que seja utilizada a versão mais recente do *branch* develop nos testes, todas as vezes é feita uma nova *build*, os comando podem ser vistos na linha 20 e 21. O mesmo processo de compilação é feito para o *branch* que está sendo analisado - linhas 22 e 23, a variável `TRAVIS_BRANCH` recupera o nome do branch que iniciou o processo de

automatização. É importante notar que os modelos de cada *build* são gravadas em pastas diferentes (`/DSSAT47/A` e `/DSSAT47/B`), nas linhas 24 e 25 todos os modelos das duas pastas são executados a partir da utilização de um arquivo *batch*, que roda todos os modelos, linhas 24 e 25. A partir desse momento o ambiente está pronto para ser testado.

- `after_script` (linha 27): Depois da *build* ser realizada, o ambiente está pronto para a realização dos testes. Na linha 29 é utilizado o comando `Rscript` para executar o arquivo R de testes. Caso a versão do *branch develop* e do *branch* que está sendo testado apresente diferenças, um erro é disparado.

.travis.yml

```
1  language: bash
2
3  sudo: required
4
5  services:
6
7  - docker
8
9  env: SH="docker exec -t ci bash -c"
10
11 before_script:
12
13 - echo "$DOCKER_PASSWORD" | docker login -u "$DOCKER_USERNAME" --password-stdin
14 - docker run --name ci --rm -d -w /home/cidssat marinadezordi/imagercroptest:2.5 tail -f /dev/null
15 - docker ps
16
17 script:
18
19 - $SH 'git clone https://'"$GIT_USERNAME"':'"$GIT_PASSWORD"'@github.com/marinadezordi/dssat-csm.git'
20 - $SH "cd dssat-csm && git checkout develop && mkdir build"
21 - $SH "cd dssat-csm/build && cmake -DCMAKE_Fortran_COMPILER=/usr/bin/gfortran-8 -DCMAKE_INSTALL_PREFIX=/DSSAT47/A .. && make -j3 && make install"
22 - $SH "cd dssat-csm && git checkout $TRAVIS_BRANCH && rm -rf build && mkdir build"
23 - $SH "cd dssat-csm/build && cmake -DCMAKE_Fortran_COMPILER=/usr/bin/gfortran-8 -DCMAKE_INSTALL_PREFIX=/DSSAT47/B .. && make -j3 && make install"
24 - $SH "cd /DSSAT47/A/ci && ../run_dssat B AllCrops.v47"
25 - $SH "cd /DSSAT47/B/ci && ../run_dssat B AllCrops.v47"
26
27 after_script:
28
29 - $SH "cd /home/test && Rscript citest.R"
```

Figura 11. Arquivo YAML do Travis CI com as configurações necessárias para o funcionamento da automatização da *build* e dos testes por meio da Integração Contínua no DSSAT

O CircleCI funciona de maneira muito semelhante ao Travis CI. O login é feito em *https://circleci.com/* com o acesso do GitHub. A partir disso é possível habilitar um repositório para realização da Integração Contínua. Pode-se optar por criar o arquivo YAML automaticamente ou manualmente - o arquivo deve ser criado com o nome de *config.yml* dentro de um diretório de nome *.circleci* na raiz do repositório do DSSAT. Todos os passos para compilação e testes do DSSAT são estruturados nesse arquivo, como pode ser visto na Figura 12.

Na primeira linha do arquivo é atribuída a versão do CircleCI. Após, a indicação do início da *build* e da utilização do Docker como serviço. Na linha 5 é informada a imagem do Docker e também as variáveis configuradas para realização do login. Não é necessário nenhuma outra configuração para que o container seja executado. A partir da linha 12 estão todos os comandos que devem ser executados no Docker - seguindo a mesma ordem dos comandoS utilizados no Travis CI, inclusive, na linha 20, quando os testes são realizados.

config.yml

```
1  version: 2.1
2  jobs:
3      build:
4        docker:
5          - image: marinadezordi/imagercroptest:2.5
6            auth:
7              username: $DOCKER_USERNAME
8              password: $DOCKER_PASSWORD
9        steps:
10         - checkout
11         - run: |
12             cd /home/cidssat # run the `echo` command
13             git clone https://github.com/marinadezordi/dssat-csm.git
14             cd dssat-csm && git checkout develop && mkdir build
15             cd build && cmake -DCMAKE_Fortran_COMPILER=/usr/bin/gfortran-8 -DCMAKE_INSTALL_PREFIX=/DSSAT47/A .. && make -j3 && make install
16             cd /home/cidssat/dssat-csm && git checkout $CIRCLE_BRANCH && rm -rf build && mkdir build
17             cd build && cmake -DCMAKE_Fortran_COMPILER=/usr/bin/gfortran-8 -DCMAKE_INSTALL_PREFIX=/DSSAT47/B .. && make -j3 && make install
18             cd /DSSAT47/A/ci && ../run_dssat B AllCrops.v47
19             cd /DSSAT47/B/ci && ../run_dssat B AllCrops.v47
20             cd /home/test && Rscript citest.R
```

Figura 12. Arquivo YAML do CircleCI com as configurações necessárias para o funcionamento da automatização da *build* e dos testes por meio da Integração Contínua no DSSAT.

O Jenkins possui mais diferenças em relação as outras ferramentas para sua configuração. A instalação do Jenkins é o primeiro passo, nesse caso, ele foi instalado numa máquina de testes, porém isso pode ser feito em um servidor. Com o Jenkins instalado esse comando deve ser executado: `java -jar jenkins.war -httpPort=8080`. Finalmente o sistema está pronto para ser acessado no navegador http://localhost:8080 e nesta página são realizadas as configurações e a personalização da Integração Contínua no Jenkins.

O Jenkins possui *plugins* que podem ser utilizados para ajudar ou viabilizar a criação das regras para compilação e testes de aplicações. Mas para que a instalação de plugins funcione é preciso alterar a *url* da central de plugins nas configurações do Jenkins, passando de `https` para `http`. Os plugins instalados foram: GitHub Authentication, Pipeline e Docker Pipeline, com suas respectivas dependências.

As variáveis de ambiente com nome de: `GIT_USERNAME` e `GIT_PASSWORD` foram criadas para login no GitHub. Depois disso é feita a configuração de um novo item - o Jenkins Multibranch Pipeline - esse recurso permite que se utilize um *script* no lugar da interface de usuário disponível. Ele difere da utilização de um Pipeline normal, porque é disparado a partir do envio de novas versões de qualquer *branch* do repositório. Esse tipo de projeto gerencia e executa automaticamente o processo definido no Pipeline do *branch*.

Para o novo item criado deve ser informado o link repositório DSSAT no GitHub, em *Source Branch*, ou seja, origem do *branch*. Na raiz do projeto cria-se um arquivo com o nome de *Jenkinsfile*, como pode ser visto na Figura 13. É possível dar outros nomes para esse arquivo, mas em todos os casos, na configuração do item, este nome deve ser informado.

Da mesma forma como nas outras ferramentas, o arquivo criado contém todos os passos necessários para compilar e testar o DSSAT. Na linha 5 a seção `agent` especifica onde o Pipeline será executado no ambiente Jenkins. Nesse momento é informada a

imagem do Docker a ser utilizada, sendo que o Docker deve estar instalado e logado na máquina onde o Jenkins está rodando.

Para o Jenkins optou-se por realizar a automatização da *build* e dos testes no repositório público do DSSAT. Além das configurações já citadas, foi necessário criar uma imagem do Docker privada, com o diretório `Data` que possui os dados de simulação do DSSAT, já que esse diretório não está disponível para o repositório aberto. Dessa forma, foi possível realizar os testes normalmente, bastando copiar o diretório para o local adequado nos passos da realização da *build*, como pode ser observado na linha 14 e 17 da Figura 13.

Jenkinsfile

```
1  pipeline {
2    agent any
3      stages {
4        stage('CI'){
5          agent{
6             docker{
7              reuseNode false
8              image 'marinadezordi/imagercroptest:1.0'
9            }
10         }
11         steps{
12           sh 'rm -rf dssat-csm-os && git clone https://github.com/marinadezordi/dssat-csm-os.git'
13           sh 'cd dssat-csm-os && git checkout develop && mkdir build'
14           sh 'cp -r /home/cidssat/Data/* dssat-csm-os/Data'
15           sh 'cd dssat-csm-os/build && cmake -DCMAKE_Fortran_COMPILER=/usr/bin/gfortran-8 -DCMAKE_INSTALL_PREFIX=/DSSAT47/A .. && make -j3 && make install'
16           sh 'cd dssat-csm-os && git checkout ${BRANCH_NAME} && rm -rf build && mkdir build'
17           sh 'cp -r /home/cidssat/Data/* dssat-csm-os/Data'
18           sh 'cd dssat-csm-os/build && cmake -DCMAKE_Fortran_COMPILER=/usr/bin/gfortran-8 -DCMAKE_INSTALL_PREFIX=/DSSAT47/B .. && make -j3 && make install'
19           sh 'cd /DSSAT47/A/ci && ../run_dssat B AllCrops.v47'
20           sh 'cd /DSSAT47/B/ci && ../run_dssat B AllCrops.v47'
21           sh 'cd /home/test && Rscript citest.R'
22         }
23       }
24     }
25 }
```

Figura 13. Arquivo *Jenkinsfile* com as configurações necessárias para o funcionamento da automatização da *build* e dos testes por meio da Integração Contínua no DSSAT.

Na seção `steps` cada `sh` da linha 12 até a linha 20 é um comando executado dentro do container para a *build* e a preparação para os testes e, finalmente, na linha 21, o teste é executado. Da mesma forma como nas outras ferramentas um resultado do *job* completo é esperado na interface do Jenkins, que possui o histórico de cada processo realizado em cada *branch* com os resultados.

4. RESULTADOS E ANÁLISES

A automatização da *build* e dos testes do DSSAT antes das atualizações do *branch develop* foi possível utilizando as três ferramentas de Integração Contínua selecionadas e a estrutura base criada por meio do Docker, conforme descrito no capítulo 3. O container Docker com todas as dependências instaladas e configuradas para compilar e testar o DSSAT otimiza o processo de realizar a *build*. Se o Docker não fosse utilizado, tudo precisaria ser feito a cada *build*, levando mais tempo.

Os desenvolvedores e administradores do DSSAT encontram no novo fluxo, que pode ser analisado na Figura 14, uma maneira diferente de obter os resultados das alterações do sistema. Neste fluxo pode-se observar que o início do processo é sempre quando o desenvolvedor executa uma das ações *push* ou *pull request* (1). Dentro do repositório do DSSAT existe o arquivo responsável por realizar a *build* e os testes (2), o qual aciona o funcionamento da Integração Contínua via a ferramenta (3). O Docker (4) executa todos os comandos do arquivo, e gera um resultado da compilação e dos testes (5) que pode ser visualizado na ferramenta (6). Dessa forma, o desenvolvedor pode acompanhar a evolução das suas alterações a cada nova versão gerada na sua *branch* de trabalho. Além disso, o admistrador do repositório tem menos trabalho para avaliar as alterações por conta da automatização.

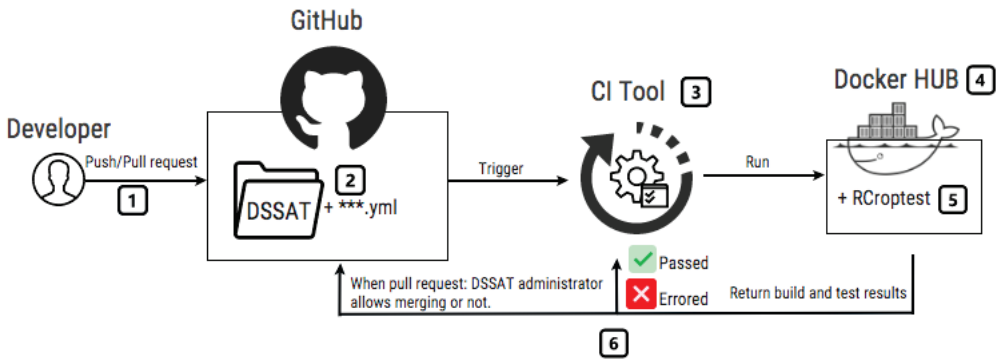

Figura 14. Fluxo da aplicação da Integração Contínua para automatização do processo de *build* e de testes de alterações do DSSAT.

Com esse cenário os desenvolvedores e os administradores do DSSAT conseguem testá-lo no mesmo ambiente, evitando que as diferenças de instalações e configurações de um sistema e ou de outro influenciem no resultado dos testes. Além disso, toda vez que o desenvolvedor faz um *push* ele recebe o resultado da *build*, e esse processo gera alguns benefícios: o primeiro é que isso é realizado automaticamente, economizando tempo do desenvolvedor; em segundo lugar, o desenvolvedor provavelmente não vai fazer um *pull*

request enquanto a *build* apresentar erros, diminuindo as chance de ter suas aterações negadas; e, por fim, ficará sabendo antes dos problemas, antecipando as soluções.

As ferramentas de CI mostram o resultado da *build* de forma semelhante entre elas. Elas possuem uma lista com o histórico de todas as *builds* já realizadas. As ferramentas fornecem o console, com todos os comandos executados e os retornos de cada um deles - importante para que o desenvolvedor possa encontrar erros e saber exatamente como a *build* se comportou. Além disso, elas apresentam um resumo de cada *build*, mostrando o nome da branch, se passou ou não, quanto tempo demorou para executar, entre outras informações que podem ser observadas na Figura 15.

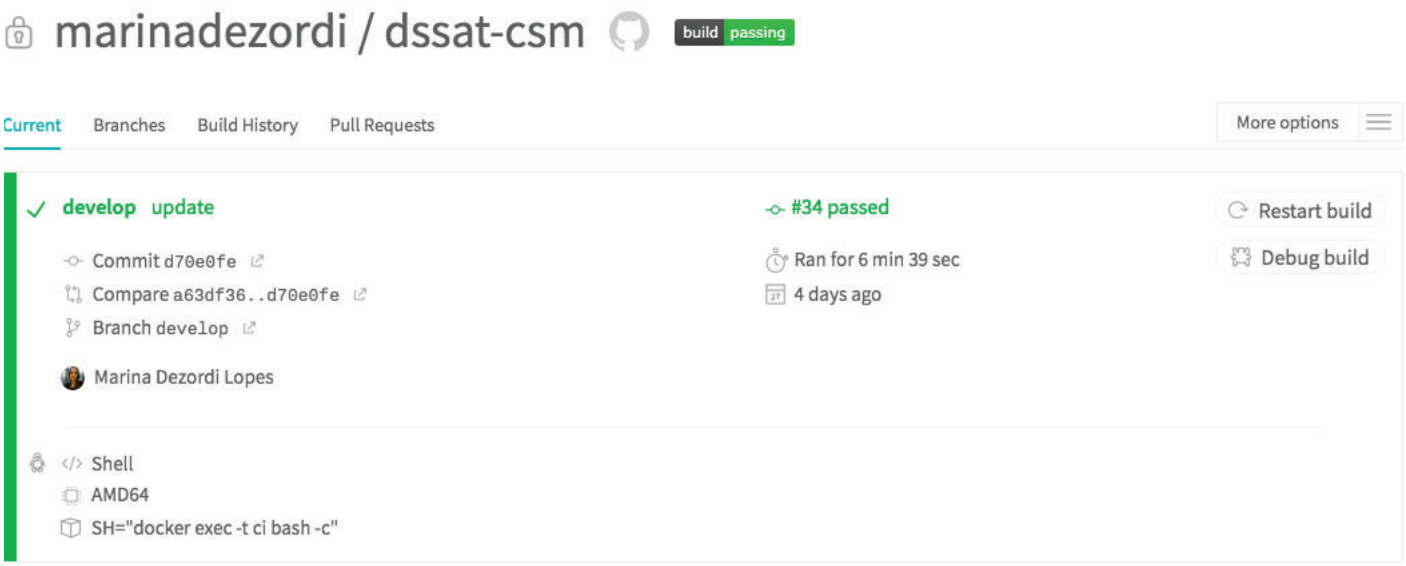

Figura 15. Exemplo de resultado de uma *build* já finalizada no Travis CI

Todas as ferramentas possuem a possibilidade de envio de email com informação da *build*, normalmente configurada para ser enviada quando algum problema acontece. Na Figura 16 observa-se um e-mail relativo ao CircleCI informando que uma *build* falhou e que precisa da atenção do desenvolvedor para proceder com a solução.

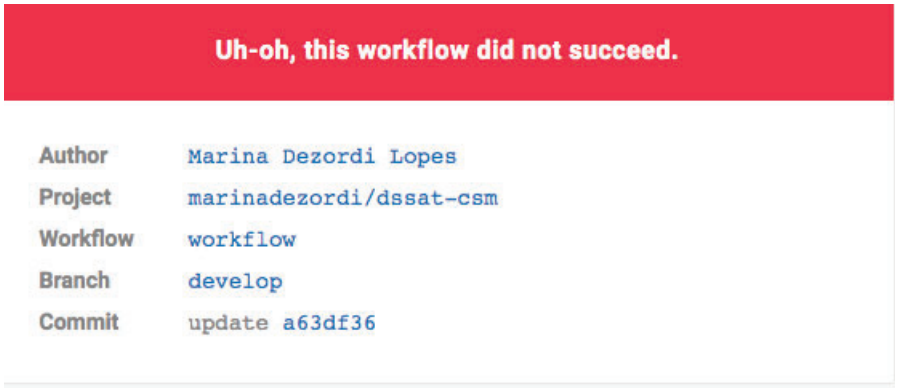

Figura 16. Exemplo de e-mail enviado pelo CircleCI informando uma falha na *build*.

Cada ferramenta utilizada possui algumas diferenças. O Travis CI e o CircleCI se assemelham mais na forma como são configurados, pois ambos são baseados em nuvem e utilizam o arquivo YAML para definir os comandos da *build*. Ainda assim, configurar o CircleCI foi mais prático e rápido, porque possui uma documentação mais assertiva. Além disso, basta informar as credenciais do Docker, e indicar que ele será utilizado como serviço no arquivo YAML que o container definido é instanciado de forma de automática.

O Travis CI possui uma particularidade interessante, ele mostra no GitHub se os testes tiveram sucesso ou não, como pode ser visto na Figura 17. Ele aponta dois resultados, um relacionado a *build* no momento do *push* e outro no *pull request*. Caso os envolvidos não precisem ver o processo de forma mais detalhada, no próprio GitHub é possível visualizar o resultado final.

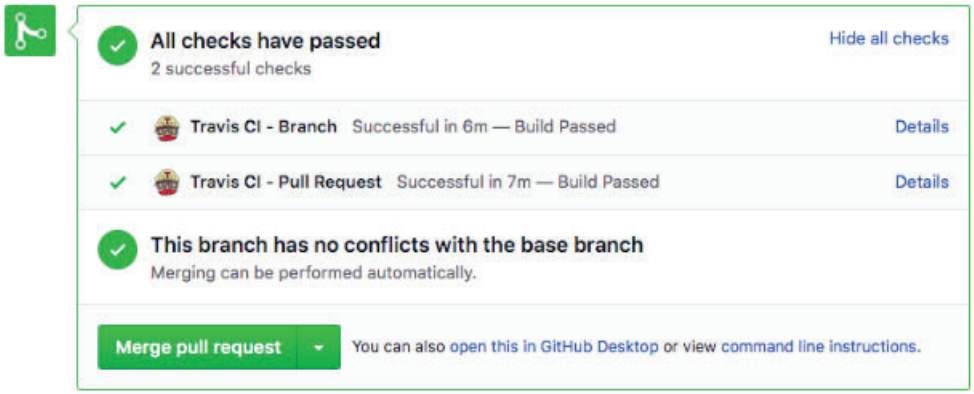

Figura 17. Resultado da *build* do DSSAT realizado por meio da ferramenta Travis CI e que pode ser visualizado no GitHub.

O uso do Jenkins é mais trabalhoso por ser necessário realizar todas as configurações localmente ou em um servidor dedicado. Além disso, para todas as ferramentas necessárias para o funcionamento da Integração Contínua, um plugin precisa ser baixado e configurado - foi o caso do GitHub, do Docker e até mesmo do Pipeline. Esse processo acaba deixando mais complexo o início do funcionamento da automatização. A vantagem disso é que o Jenkins possui maiores possibilidades de personalização, porém para o propósito do DSSAT as outras ferramentas também mostraram-se suficientes.

Em relação ao tempo de execução da *build*, o CircleCI apresenta o menor tempo, a ferramenta completa a *build* em torno de três a cinco minutos, o Travis CI leva por volta de seis minutos e o Jenkins onze minutos.

Todas as ferramentas foram utilizadas gratuitamente. O Travis CI é gratuito para repositórios públicos. Como o DSSAT possui repositório privado, para utilizar a ferramenta de forma gratuita foi necessário garantir maiores acessos através da conta do Github Education (disponível para professores e alunos da UPF). O CircleCI possui um plano gratuito que permite um total de até 250 minutos de *build* por semana, rodando um *job* por vez. Para os testes realizados a versão gratuita da ferramenta foi suficiente, porém limitaria o

uso com todos os desenvolvedores do DSSAT utilizando a ferramenta. Já o Jenkins é totalmente gratuito, e também oferece a possibilidade do desenvolvimento de novos *plugins* caso necessite de algo específico que ainda não exista.

O objetivo de automatizar a *build* e realizar os testes do DSSAT enquanto os desenvolvedores modificam o código em suas *branches* locais e também quando querem integrar ao *branch develop* foi atendido plenamente pelas ferramentas de Integração Contínua selecionadas por este trabalho. Para os administradores decidirem se aceitam não o *pull request* recebido, uma parte do trabalho já está pronta e, para a maioria dos casos, uma rápida análise nas diferenças dos arquivos utilizando o GitHub vai ser suficiente para a tomada de decisão sobre a aceitação ou não do novo código.

5. CONCLUSÕES

A aplicação de ferramentas de Integração Contínua no processo de versionamento do DSSAT traz uma mudança significativa no fluxo de trabalho dos desenvolvedores e dos administradores do sistema. Um processo que hoje é feito de forma manual pode ser automatizado trazendo uma série de benefícios e contribuindo com a evolução do DSSAT.

Para os desenvolvedores a mudança está principalmente no controle da *build* de cada versão enviada ao repositório remoto de trabalho. É possível obter o resultado de cada uma das *builds* utilizando a ferramenta de CI, enquanto desenvolve e realiza um *push* a ferramenta se encarrega de compilar e testar o que é essencial ao DSSAT. No momento em que algum erro é encontrado, o desenvolvedor sabe que precisa corrigir. Ele pode contar com a ferramenta de CI para saber qual problema ocorreu, consultando o console com o histórico de cada comando executado e os resultados.

Sem a automatização, para ter o mesmo resultado, o desenvolvedor teria que compilar o DSSAT na versão que está alterando, baixar o DSSAT do *branch develop* atualizado e compilar também, para depois utilizar o Croptest ou o RCroptest para realizar a comparação dos modelos. Uma rotina longa e demorada que faz o trabalho render pouco, ou faz o desenvolvedor optar por não realizar todos esses passos a cada envio de código.

Quando o desenvolvedor demora mais para saber que o código em que está trabalhando possui problemas, identificar o erro costuma ser um processo mais lento e uma alternativa de solução pode ser mais difícil de ser encontrada. Isso porque a complexidade da funcionalidade na qual se está trabalhando vai aumentando a medida que o código evolui.

Para os administradores do DSSAT o trabalho é reduzido ainda mais. Se atualmente eles precisam compilar as duas versões do sistema - a que o desenvolvedor quer integrar e a do *branch develop* - e depois testar, com a automatização eles recebem esse resultado pronto. Eles podem realizar algumas outras avaliações que consideram pertinentes, como olhar as diferenças entre os arquivos, por exemplo, mas a parte mais demorada e trabalhosa da análise está pronta.

Outro ponto importante é que o desenvolvedor vem realizando os testes no mesmo ambiente que o administrador do DSSAT utiliza quando recebe um *pull request*. Esse fato evita que o desenvolvedor considere que suas alterações estão adequadas, mas, por conta de diferenças de configurações ou de dependências, os testes do administrador podem indicar algum problema. O mesmo ambiente favorece que o administrador do DSSAT receba menos alterações com problema.

Entre as ferramentas utilizadas, o TravisCI se destacou pela integração com o GitHub, e por ser relativamente rápido de começar a ser utilizado, além disso com o acesso

via GitHub Education foi possível utilizá-lo sem restrições e em repositório privado. O CircleCI foi o mais rápido e fácil de começar, apresentou um tempo menor na execução da *build* e tem uma documentação completa e fácil de entender, porém possui uma restrição na versão gratuita referente ao tempo de execução de *build* por semana que pode ser um impeditivo para a sua utilização. O Jenkins é o que mais precisa de configurações e o mais demorado para iniciar a utilização, possui documentações dispersas por conta dos *plugins* desenvolvidos por terceiros mas, por outro lado, é uma ferramenta com muita possibilidade de customização e é totalmente gratuita.

Como trabalhos futuros considera-se primeiramente a implantação da Integração Contínua em um recente repositório público criado pelo DSSAT (citado no capítulo 3), assim que o projeto for aprovado pelos administradores do sistema. Esse repositório de acesso livre precisará de algumas configurações adicionais que já foram simuladas por meio das ferrmamentas TravisCI e Jenkins. Essa implantação deve servir como experiência para realizar as adaptações que se mostrarem necessárias. A partir dessa avaliação inicial e de realizados os ajustes a integração contínua poderá ser implantada no repositório privado. Além desta continuidade de trabalhos junto ao DSSAT, este trabalho possibilitará que se aplique a Integração Contínua em outros projetos em desenvolvimento no grupo Mosaico, especialmente em projetos que contam com parcerias entre várias instituições e que o desenvolvimento está sendo realizado por um grupo de desenvolvedores localizados em diversas instituições.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] BOOCH, G.; GRADY. *Object oriented design with applications*. Benjamin/Cummings Pub. Co, 1991. 580 p. ISBN 0805300910. Disponível em: <https://dl.acm.org/citation.cfm?id=82528>.

[2] HILTON, M. et al. Usage, costs, and benefits of continuous integration in open-source projects. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. New York, New York, USA: ACM Press, 2016. p. 426–437. ISBN 9781450338455. Disponível em: <http://dl.acm.org/citation.cfm?doid=2970276.2970358>.

[3] JONES, J. et al. The dssat cropping system model. *European Journal of Agronomy*, v. 18, n. 3, p. 235 – 265, 2003. ISSN 1161-0301. Modelling Cropping Systems: Science, Software and Applications. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1161030102001077>.

[4] HOOGENBOOM, G. et al. *Decision Support System for Agrotechnology Transfer (DSSAT) Version 4.7.5*. 2019. Disponível em: <https://dssat.net>.

[5] BECK, K. *Embracing change with extreme programming*. [s.n.], 1999. v. 32. 70–77 p. ISSN 00189162. Disponível em: <http://ieeexplore.ieee.org/document/796139/>.

[6] FOWLER, M. *Continuous Integration*. 2018. Disponível em: <https://www.martinfowler.com/articles/continuousIntegration.html>.

[7] HAMDAN, S.; ALRAMOUNI, S. A Quality Framework for Software Continuous Integration. *Procedia Manufacturing*, v. 3, p. 2019–2025, 2015. ISSN 23519789. Disponível em: <http://www.sciencedirect.com/science/article/pii/S2351978915002504>.

[8] STÅHL, D.; BOSCH, J. Cinders: The continuous integration and delivery architecture framework. *Information and Software Technology*, Elsevier, v. 83, p. 76–93, mar 2017. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S095058491630369X>.

[9] Amazon Web Services. *O que significa integração contínua? – Amazon Web Services*. 2018. Disponível em: <https://aws.amazon.com/pt/devops/continuous-integration/>.

[10] LAUKKANEN, E.; ITKONEN, J.; LASSENIUS, C. Problems, causes and solutions when adopting continuous delivery—A systematic literature review. *Information and Software Technology*, Elsevier, v. 82, p. 55–79, feb 2017. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584916302324>.

[11] DUVALL, P. M.; MATYAS, S.; GLOVER, A. *Continuous integration : improving software quality and reducing risk*. [S.l.]: Addison-Wesley, 2007. 283 p. ISBN 9780321336385.

[12] NIEPHAUS, F. et al. smalltalkCI. In: *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies - IWST'16*. New York, New York, USA: ACM Press, 2016. p. 1–9. ISBN 9781450345248. Disponível em: <http://dl.acm.org/citation.cfm?doid=2991041.2991044>.

[13] MÅRTENSSON, T.; STÅHL, D.; BOSCH, J. Enable more frequent integration of software in industry projects. *Journal of Systems and Software*, Elsevier, v. 142, p. 223–236, aug 2018. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121218300906>.

[14] MOSSIGE, M.; GOTLIEB, A.; MELING, H. Testing robot controllers using constraint programming and continuous integration. *Information and Software Technology*, Elsevier, v. 57, p. 169–185, jan 2015. ISSN 0950-5849. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0950584914002080>.

[15] KNAUSS, E. et al. Continuous Integration Beyond the Team. In: *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '16*. New York, New York, USA: ACM Press, 2016. p. 1–6. ISBN 9781450344272. Disponível em: <http://dl.acm.org/citation.cfm?doid=2961111.2962639>.

[16] LAUKKANEN, E.; PAASIVAARA, M.; ARVONEN, T. Stakeholder Perceptions of the Adoption of Continuous Integration – A Case Study. In: *2015 Agile Conference*. IEEE, 2015. p. 11–20. ISBN 978-1-4673-7153-7. Disponível em: <http://ieeexplore.ieee.org/document/7284593/>.

[17] STÅHL, D.; MÅRTENSSON, T.; BOSCH, J. The continuity of continuous integration: Correlations and consequences. *Journal of Systems and Software*, Elsevier, v. 127, p. 150–167, may 2017. ISSN 0164-1212. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121217300328>.

[18] HILTON, M.; MICHAEL. Understanding and improving continuous integration. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. New York, New York, USA: ACM Press, 2016. p. 1066–1067. ISBN 9781450342186. Disponível em: <http://dl.acm.org/citation.cfm?doid=2950290.2983952>.

[19] ZAMPETTI, F. et al. How open source projects use static code analysis tools in continuous integration pipelines. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. [S.l.]: IEEE, 2017. p. 334–344. ISBN 978-1-5386-1544-7.

[20] VöST, S. Vehicle level continuous integration in the automotive industry. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2015. (ESEC/FSE 2015), p. 1026–1029. ISBN 9781450336758. Disponível em: <https://doi.org/10.1145/2786805.2803193>.

[21] LIMA, J. A. P.; VERGILIO, S. R. Test case prioritization in continuous integration environments: A systematic mapping study. *Information and Software Technology*, v. 121, p. 106268, 2020. ISSN 0950-5849. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0950584920300185>.

[22] SHAHIN, M.; ALI BABAR, M.; ZHU, L. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, v. 5, p. 3909–3943, 2017. ISSN 2169-3536.

[23] BELLER, M.; GOUSIOS, G.; ZAIDMAN, A. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2017. p. 447–450.

[24] ZHAO, Y. et al. The impact of continuous integration on other software development practices: A large-scale empirical study. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2017. p. 60–71.

[25] ISLAM, M. R.; ZIBRAN, M. F. Insights into continuous integration build failures. In: *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017. (MSR '17), p. 467–470. ISBN 9781538615447. Disponível em: <https://doi.org/10.1109/MSR.2017.30>.

[26] KRAFCZYK, M. et al. Scientific tests and continuous integration strategies to enhance reproducibility in the scientific software context. In: *Proceedings of the 2nd International Workshop on Practical Reproducible Evaluation of Computer Systems*. New York, NY, USA: Association for Computing Machinery, 2019. (P-RECS '19), p. 23–28. ISBN 9781450367561. Disponível em: <https://doi.org/10.1145/3322790.3330595>.

[27] WIDDER, D. G. et al. A conceptual replication of continuous integration pain points in the context of travis ci. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2019. (ESEC/FSE 2019), p. 647–658. ISBN 9781450355728. Disponível em: <https://doi.org/10.1145/3338906.3338922>.

[28] HÖLBIG, C. A. et al. RCROPGRO package: an approach for encapsulating fortran coded models using R language. In: *Proceedings of the 11th International Conference Applied Computing 2014*. [S.l.]: IADIS Press, 2014. v. 1, p. 119–126.

[29] HÖLBIG, C. A. et al. An approach for encapsulating fortran coded models into a R package. In: *Proceedings of the 7th International Congress on Environmental Modelling and Software*. [S.l.]: IEMSS, 2014. v. 4, p. 2180–2187.

[30] RESENES, J. de A. et al. jdssat: A javascript module for dssat-csm integration. *SoftwareX*, v. 10, p. 100271, 2019. ISSN 2352-7110. Disponível em: <http://www.sciencedirect.com/science/article/pii/S235271101930158X>.

[31] NICOLAU, M. *CROPTEST : data-drive test automation for crop modeling systems*. Dissertação (Mestrado em Computação Aplicada) — Universidade de Passo Fundo, 2018. Disponível em: <http://tede.upf.br/jspui/handle/tede/1531>.