DAS Departamento de Automação e Sistemas
CTC Centro Tecnológico
UFSC Universidade Federal de Santa Catarina

Ferramenta para análise correlacional contínua entre a qualidade de um produto e seu processo de desenvolvimento de software

Relatório submetido à Universidade Federal de Santa Catarina

como requisito para a aprovação da disciplina:

DAS 5511: Projeto de Fim de Curso

Michael de Caro Camilo

Florianópolis, Agosto de 2019

Ferramenta para análise correlacional contínua entre a qualidade de um produto e seu processo de desenvolvimento de software.

Michael de Caro Camilo

Esta monografia foi julgada no contexto da disciplina

DAS 5511: Projeto de Fim de Curso

e aprovada na sua forma final pelo

Curso de Engenharia de Controle e Automação

Prof. Rômulo Silva de Oliveira

____

Banca Examinadora:

Thalles Felipin Rigobello
Orientador na Empresa

Prof. Rômulo Silva de Oliveira
Orientador no Curso

Agradecimentos

Gostaria de agradecer a todos da Hexagon, primeiramente ao Hugo Fagundes e ao Thalles Rigobello por fazerem essa ponte que aproxima uma empresa de tão alto nível da universidade. Agradeço ao Luan Silveira, os companheiros da minha primeira equipe, na área de desenvolvimento de software embarcado, e ao professor Rômulo Silva de Oliveira por tudo o que me ensinaram ao longo de todo o trabalho desenvolvido.

Gostaria de agradecer também à minha equipe atual, na área de desenvolvimento de software web, pela convivência e ensinamentos, e principalmente a Carolina Lorini por me trazer para a empresa, para sua equipe, por todas as palavras de direcionamento, e principalmente por acreditar no meu potencial.

Por fim gostaria de agradecer a minha família pelo apoio incondicional ao longo de todos os anos e situações, e principalmente ao meu filho que despertou o melhor em mim.

Resumo

A Hexagon é uma empresa global líder em sensoriamento, software e soluções autônomas. Sua divisão de agricultura, a Hexagon Agriculture, oferece ferramentas tecnológicas que otimizam todas as etapas agrícolas (do planejamento do cultivo ao rastreio da safra). Tendo como objetivo ser o líder na criação dos Ecossistemas Autônomos Conectados (ACE) nos segmentos que atuam, é essencial inovar com qualidade, em tempo hábil e de forma que satisfaça o cliente. Visando a competitividade, é necessário otimizar a produção ao maximizar a qualidade do produto e velocidade do desenvolvimento, enquanto se minimiza os custos. Assim, o presente estudo desenvolveu uma ferramenta que realiza a coleta automática das métricas de qualidade dos códigos para cada nova versão de cada projeto, servindo como infraestrutura para a análise correlacional contínua entre a qualidade de um produto e seu processo de desenvolvimento de software. Estando concentrada na extração de dados, manipulação e escrita no banco de dados (em inglês, Extract, Transform and Load), esta ferramenta permitirá que a análise seja recorrente, entregando dados atualizados e estruturados de uma maneira simples de forma contínua. O projeto foi estruturado em duas etapas: Estudo do Processo (entendimento do processo de desenvolvimento atual e escolha das métricas a serem consideradas) e Coleta de Dados (modelagem do banco de dados e estruturação e implementação da ferramenta), contendo na conclusão deste relatório o estado atual da ferramenta e sua aplicação). Dentre as metodologias/ferramentas utilizadas no desenvolvimento do projeto, destaca-se a Metodologia Ágil, Scrum, Jira, Git, Bitbucket, Pipeline, Modelo Estrela, pgModeler, Amazon (S3, AWS CLI, SQS, RDS, Lambda), HashiCorp Terraform e Flyway. A ferramenta está atualmente realizando a coleta das métricas de 6 repositórios, além de ser inserida automaticamente em novos repositórios, e realizando a coleta de todas as atividades existentes no JIRA da equipe de desenvolvimento de software web.

Palavras-chave: AWS, Microsserviços, ETL, Lambda, Python, Scrum, SOA.

Abstract

Hexagon is a global leader company in sensing, software and autonomous solutions. Its agriculture division, Hexagon Agriculture, offers technology tools that optimize all stages in agriculture (from crop planning to crop tracking). With the objective of being the leader in the creation of Connected Autonomous Ecosystems (ACE) in the segments where they operate, it is essential to innovate with quality, fast and in a way that satisfies the customer. Aiming the competitiveness, it is necessary to optimize production with maximizing quality and development speed while minimizing costs. Thus, the present study developed a tool that automates the metrics collection of each new version for each projects, serving as infrastructure for the continuous correlation analysis between the quality of a product and its software development process. Being focused on data extraction, manipulation and load (ETL) in the database, this tool will allow the analysis to be recurrent, delivering updated and structured data in a simple way, continuously. The project was structured in two stages: Process Study (understanding the current development process and choosing the metrics to be considered) and Data Collection (modeling the database and structuring and implementation of the tool), presenting at this report conclusion, the current state of the tool and its application. Among the methodologies / tools used in the project development are the Agile Methodology, Scrum, Jira, Git, Bitbucket, Pipeline, Model Star, pgModeler, Amazon (S3, AWS CLI, SQS, RDS, Lambda), HashiCorp Terraform and Flyway. The tool is currently performing the collection of metrics from 6 repositories. Also is being automatically inserted into new repositories, and collecting all issues in JIRA from the web software development team.

Keywords: AWS, Microservices, ETL, Lambda, Python, Scrum, SOA.

Lista de ilustrações

Figura 1 – Custo Relativo de defeitos de Software. Fonte: IBM Systems Sciences Institute . . . 16
Figura 2 – Ciclo de desenvolvimento Scrum para atividades. Fonte: Atlassian . . . 20
Figura 3 – Ciclo de desenvolvimento Scrum para Sprints. Fonte: Wikimedia Commons 21
Figura 4 – Armazenamento de dados no Git. Fonte: Git . . . 22
Figura 5 – Pull Request do Bitbucket. Fonte: Atlassian . . . 24
Figura 6 – Evolução dos fluxos de trabalho de tecnologias CI. Fonte: DevOps.com 25
Figura 7 – Fluxo de atividades do Projeto de Análise do Desenvolvimento . . . 31
Figura 8 – Diagrama de casos de uso de negócio do Projeto de Análise do Desenvolvimento . . . 32
Figura 9 – Diagrama Sequencial do primeiro serviço. . . . 34
Figura 10 – Diagrama de Classes do segundo serviço. . . . 35
Figura 11 – Fluxo de trabalho atual. Fonte: Hexagon Agriculture . . . 38
Figura 12 – Fluxograma da coleta de dados por serviço. . . . 41
Figura 13 – Esquema para armazenamento das métricas dos códigos fontes . . . 42
Figura 14 – Esquema do banco de dados para análise de código . . . 44
Figura 15 – Diagrama de Classes Final do primeiro serviço. . . . 48
Figura 16 – Coleta de dados anterior ao trabalho. . . . 51

Nomenclatura

API	Application programming interface
AWS	Amazon Web Services
BD	Banco de Dados
BI	Business Intelligence
CD	Continuous Deployment
CI	Continuous Integration
CLI	Command Line Interface
ER	Entity–Relationship
ETL	Extract, Transform and Load
IBM	International Business Machines
PR	Pull Request
QA	Quality Assurance
RDS	Relational Database Service
S3	Simple Storage Service
SQS	Simple Queue Service
UML	Unified Modeling Language

Sumário

1 INTRODUÇÃO . . . 15

2 FUNDAMENTAÇÃO TEÓRICA . . . 19
2.1 Metodologia Ágil . . . 19
2.2 Scrum . . . 20
2.3 Git - Sistema de Controle de Versões Distribuídos . . . 21
2.4 Bitbucket . . . 23
2.5 Pipelines . . . 24
2.6 JIRA . . . 25
2.7 Modelo Estrela . . . 26
2.8 pgModeler . . . 26
2.9 Service-oriented architecture (SOA) . . . 27
2.10 Amazon Web Service (AWS) . . . 27
2.10.1 Amazon Simple Storage Service (S3) . . . 27
2.10.2 Amazon Simple Queue Service (SQS) . . . 27
2.10.3 Amazon Relational Database Service (RDS) . . . 28
2.10.4 Amazon Lambda . . . 28
2.10.5 Amazon CloudWatch . . . 28
2.11 HashiCorp Terraform . . . 28
2.12 Flyway . . . 29

3 PLANEJAMENTO . . . 31

4 ESTUDO DO PROCESSO DE DESENVOLVIMENTO . . . 37
4.1 Organização das Tarefas . . . 37
4.2 Workflow . . . 37
4.3 Pipeline . . . 39
4.4 Métricas . . . 39

5 COLETA DE DADOS . . . 41
5.1 Modelagem do Banco de Dados . . . 42
5.1.1 Schema Repositories . . . 42
5.1.2 Schema Jira . . . 43
5.2 Implantação do Banco de Dados . . . 44
5.3 Extração dos Dados - Repositórios . . . 44
5.3.1 Scripts para extração das Métricas . . . 45

5.3.2 Pipeline 45
5.3.3 Lambda 46
5.4 Extração dos Dados - JIRA 47
5.4.1 Lambda 47

6 CONCLUSÃO 51
6.1 Perspectivas 53

REFERÊNCIAS 55

1 Introdução

A Hexagon é uma empresa global líder em sensoriamento, software e soluções autônomas. Ao utilizar dados para aprimorar a eficiência, produtividade e qualidade, otimiza aplicações industriais, de manufatura, infraestrutura, segurança e mobilidade [1].

Como sua divisão de agricultura, a Hexagon Agriculture oferece ferramentas tecnológicas que otimizam todas as etapas agrícolas, desde o planejamento do cultivo e transporte da colheita, até o rastreio da safra [2]. Mantém como objetivo ser a líder na criação dos Ecossistemas Autônomos Conectados (ACE) nos segmentos que atuam. Para isso, é essencial inovar, com qualidade, em tempo hábil, e de forma que satisfaça o cliente. Dessa forma, de modo a ser mais competitivo, é necessário otimizar a produção ao maximizar a qualidade e velocidade do desenvolvimento enquanto se minimiza os custos.

O propósito do projeto é ajudar a Hexagon Agriculture em sua missão de entregar soluções digitais de uma forma mais eficaz. Assim, visa possibilitar a análise contínua do seu processo de desenvolvimento de *software*, evidenciando quais práticas ou subprocessos impactam na qualidade e quais não têm relação. Ao fazer essa análise é possível utilizar a filosofia de gestão *Lean* [3], reduzindo processos que não geram resultados, e focando nos que afetam positivamente.

Da ótica do processo de desenvolvimento de software, um produto entregue sem erros evita retrabalhos e, consequentemente, tempo para o corrigir. Quanto mais tarde esse erro for identificado, mais distante estará da sua concepção, ou seja, mais tempo deverá ser empregado para o identificar, dentro de diversas aplicações e comportamentos esperados que mesclam, compondo o produto.

Após identificar o erro, é necessário verificar sua origem, ou seja, se o código detém um erro de implementação, ou se está correto e foi resultado de especificação equivocada ou parcial. Para isso, o erro deve ser inicialmente isolado no código e então comparado com a especificação. O rastreio se torna mais difícil com o passar do tempo, pois o desenvolvedor deverá analisar novamente todo o fluxo do código e sua lógica novamente, para garantir que o problema não está lá, e então direcionar sua resolução para o setor de design, ou então o corrigir. Caso a empresa não tenha um estilo de programação (*code style*) bem definido, mais tempo pode ser necessário caso outro desenvolvedor seja responsável pela correção, tendo ainda que se habituar as práticas usadas na criação daquela parcela do código.

No caso do erro ser identificado após a sua liberação ao cliente, seu impacto é muito maior, afetando na satisfação do cliente e consequentemente na visão que tem da empresa. Além da possível perda de um cliente, custo já considerável, a divulgação negativa pode

prejudicar uma empresa em escala ainda maior.

Uma noção quantitativa da proporção do custo um *bug* em relação ao processo de desenvolvimento de software é apresentada pelo Systems Sciences Institute da International Business Machines (IBM), na qual o custo de retificar um erro na etapa de implementação é seis vezes maior do que quando encontrado na etapa de design, e até cem vezes mais quando identificado durante a fase de manutenção [4]. Ou seja, o custo aumenta exponencialmente conforme o erro avança no processo de desenvolvimento do software, como ilustrado na Figura 1.

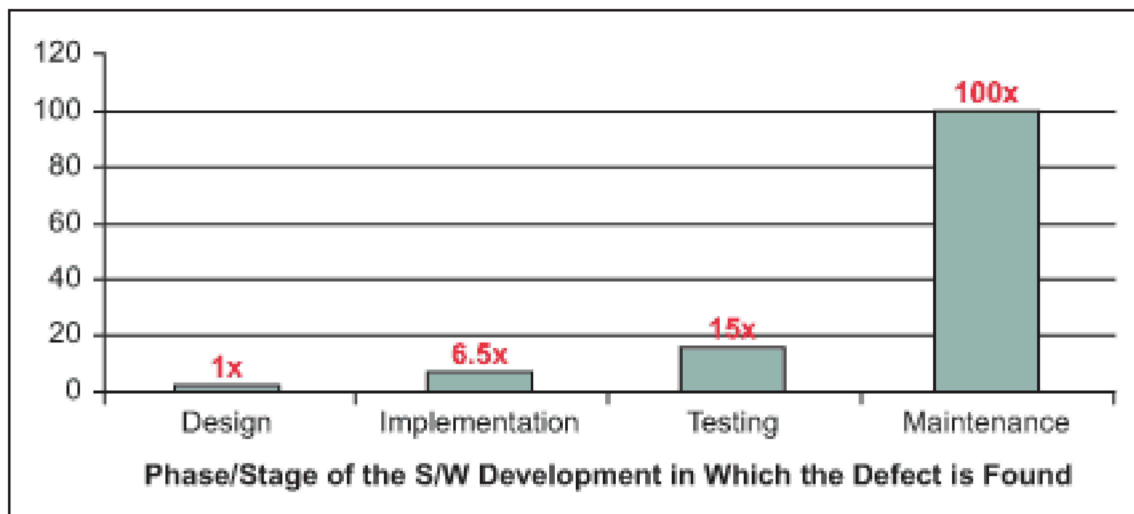

Figura 1 – Custo Relativo de defeitos de Software. Fonte: IBM Systems Sciences Institute

Para possibilitar a análise contínua do processo de desenvolvimento de software, além de um profundo estudo sobre o processo de desenvolvimento atual, é necessária a coleta de dados, de maneira a quantificá-lo e uma análise dos mesmos para a criação de indicadores que evidenciem suas correlações. Dessa forma, este trabalho detém como objetivo geral a criação de uma ferramenta que, de forma automática, faça a extração desses dados, manipulação e escrita no banco de dados, ou em inglês, *Extract, Transform and Load* (ETL). Esta ferramenta permitirá que a análise, feita na empresa de forma esporádica e manual, seja recorrente, entregando dados atualizados e estruturados de uma maneira simples, possibilitando análises a qualquer momento, continuamente.

Complementarmente, os objetivos específicos do projeto são:

- Segurança: devido a manipulação de dados internos e privilegiados da empresa.
- Confiabilidade: como executará automaticamente, o ETL precisa ser confiável, ou seja, processar sem erros.

- Modularidade: buscando o isolamento de falhas, bem como a independências entre subprocessos, permitindo que alterações pontuais não afetem o funcionamento total da ferramenta.
- Facilidade de leitura do código: organização e clareza no código é essencial para seu entendimento e sua incrementação.
- Facilidade de manutenção do código: essencial para correção de erros.
- Repetibilidade: sua estrutura deve ser facilmente replicável, para a criação de novas ferramentas destinadas a outros tipos de análises.

Dessa forma, este relatório inicia com uma fundamentação teórica dos conceitos e ferramentas utilizados, e em seguida se estrutura em três etapas: Estudo do Processo Atual, Coleta de Dados e Conclusão;

1. Na primeira etapa, Estudo do Processo Atual, há o entendimento do fluxo do processo e escolha das métricas a serem consideradas para sua análise.
2. Na segunda, Coleta de Dados, há a modelagem do banco de dados e a implementação da ferramenta, identificando como é feita a coleta das informações e métricas do processo, seu processamento e inserção no banco de dados. Em todo o capítulo são ressaltados os motivos que levaram o projeto a ser desenvolvido com essa estrutura.
3. Na última etapa, Conclusão, é apresentado o estado atual da ferramenta desenvolvida, seu impacto, ou seja, como está sendo aplicada, e perspectivas acerca do futuro desse projeto na empresa.

Ao longo deste relatório, as decisões de arquitetura foram guiadas pela arquiteta de software Carolina Lorini. As demais tarefas foram realizadas exclusivamente pelo seu autor, exceto quando explicitamente apontado o contrário.

2 Fundamentação Teórica

Ao longo do relatório vários conceitos, bem como diversas ferramentas serão abordados. Sendo criada tendo como base o processo de desenvolvimento de software atual, a ferramenta acompanha a metodologia ágil SCRUM utilizada, e outras ferramentas, como o Bitbucket, facilitando sua implementação e diminuindo seu impacto.

De modo a não interromper a descrição do projeto nos capítulos subsequentes, uma breve fundamentação teórica será apresentada a seguir. A contextualização dessas ferramentas e conceitos se dará no decorrer deste relatório.

2.1 Metodologia Ágil

Já utilizada pela Hexagon Agriculture, a Metodologia Ágil guia o desenvolvimento de software da empresa, sendo um objeto central na análise deste processo, descrita no capítulo 4.

Tal metodologia surgiu em 2001 devido a necessidade do mercado em atender às demandas dos clientes de maneira mais dinâmica e flexível, com maior produtividade. Seu texto base, o Manifesto Ágil [5], entre vários pontos prioriza satisfazer o cliente através de entregas rápidas e contínuas de software, aceitar mudanças de requisitos, a qualquer momento do desenvolvimento, focar na motivação e integração dos desenvolvedores e contínua atenção à excelência técnica e bom design, a fim de aumentar a agilidade.

Este método utiliza uma abordagem de planejamento incremental e iterativa. O desenvolvimento e a documentação do software ocorrem juntos em cada iteração, que inicia um novo ciclo com novas definições por parte do cliente para as próximas entregas. Dessa forma, mudanças de requisito a qualquer momento têm um impacto menor no projeto, minimizando o retrabalho necessário.

Visando sempre versões regulares funcionais, esse processo necessita da proximidade e colaboração constante entre o cliente e a equipe de desenvolvimento, reduzindo possíveis falhas de comunicação, permitindo mudanças de forma ágil no projeto, e garantindo que o produto final tenha uma utilidade e qualidade maior para o cliente. [6]

Utilizando das entregas recorrentes, a ferramenta deverá extrair as métricas de cada versão funcional, ou seja, de cada versão entregue ao cliente, para análise de qualidade ao longo do tempo.

2.2 Scrum

Dentre as metodologias ágeis, como, por exemplo, Scrum, XP e Kanban, a empresa utiliza o SCRUM. Para inserir a ferramenta no processo de desenvolvimento de software baseado nessa metodologia, foi necessário, a priori, entendê-la.

Baseada no Manifesto Ágil, o SCRUM inicia com uma visão clara do projeto e das funcionalidades a serem desenvolvidas para o produto em ordem de importância. Essas funcionalidades compõem as tarefas a serem feitas do produto, seu *Backlog*. O desenvolvimento é feito em pequenas equipes de desenvolvedores, de forma iterativa em ciclos de uma até quatro semanas chamados de *Sprints*. Em cada *Sprint* a equipe faz uma reunião de planejamento na qual escolhe no *Backlog* as funcionalidades/tarefas que acredita que completará, formando assim o *Sprint Backlog*. [7]

As atividades são organizadas em quadros com colunas que descrevem seus estados, para que outros desenvolvedores saibam em qual estado está cada tarefa. O nome, número e fluxo de trabalho (*workflow*) dos estados pode variar dependendo da empresa, mas em geral seguem a dinâmica de tarefas a serem feitas, em progresso, em testes, e finalizadas. Pode haver, também, o estado bloqueado, indicando a existência de algum impeditivo, como por exemplo, a falta de um *hardware* ou de uma ferramenta específica, acesso a algum sistema ou devido a dependência de outra tarefa. A Figura 2 ilustra um fluxo básico das tarefas dentro de um Sprint Scrum.

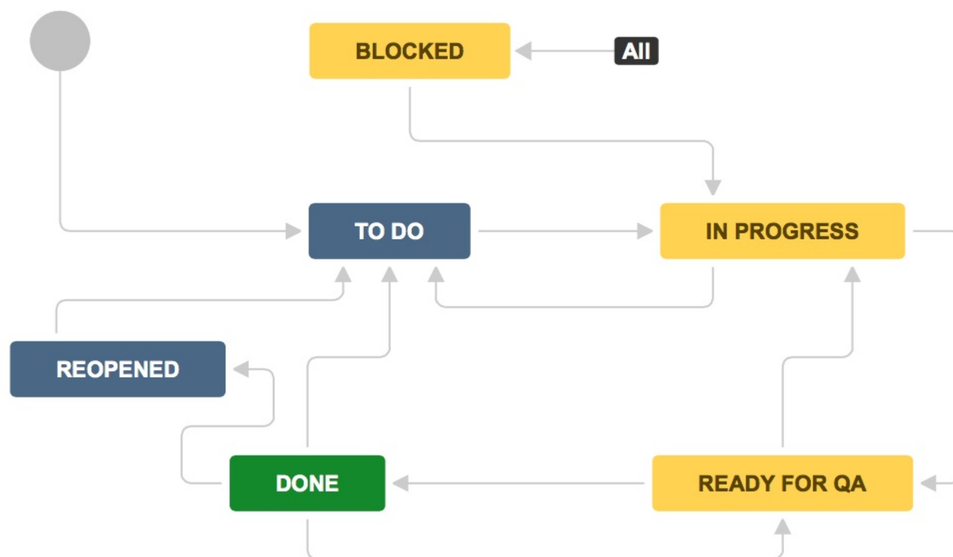

Figura 2 – Ciclo de desenvolvimento Scrum para atividades. Fonte: Atlassian

Dessa forma se inicia o *Sprint*, cujo escopo não deve ser alterado, permitindo a equipe focar em sua conclusão. Novas tarefas que apareçam são adicionadas ao *Backlog*, para ser analisada para a próxima iteração [7]. Durante o *Sprint*, há um encontro diário

rápido de aproximadamente 5 minutos para atualizar os membros da equipe, no qual cada um diz o que fez no dia anterior, o que fará no presente dia e o quais suas dificuldades.

Ao final do ciclo a equipe apresenta os seus resultados e coleta os *feedbacks* que irão guiar o próximo *Sprint*, além de fazer uma retrospectiva focada em como melhorar. Tal reunião, que encerra o ciclo, direciona a equipe aos três pilares do Scrum: transparência, inspeção e adaptação. Uma representação de todo o ciclo apresentado é mostrado na Figura 3.

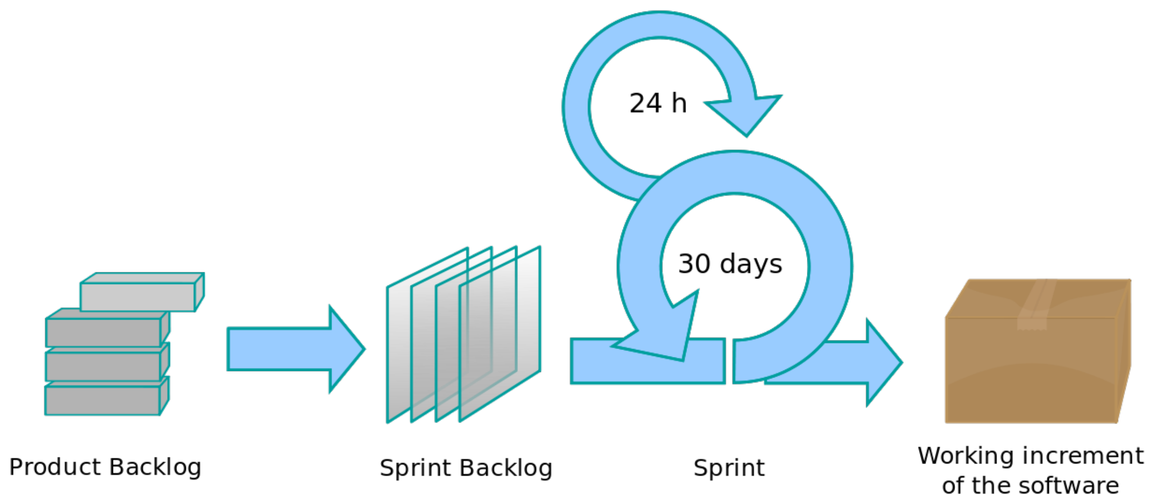

Figura 3 – Ciclo de desenvolvimento Scrum para Sprints. Fonte: Wikimedia Commons

2.3 Git - Sistema de Controle de Versões Distribuídos

Da mesma forma que o SCRUM, o git é o sistema de controle de versões já utilizado pela empresa. Como o objetivo desse projeto é coletar as métricas do processo de desenvolvimento de *software*, o domínio dessa ferramenta foi essencial para se entender como extrair informações de cada versão, e para isso, entender o que é um sistema de controle de versões.

O controle de versão é um sistema que registra as mudanças feitas em um arquivo ou um conjunto de arquivos ao longo do tempo de forma que você possa recuperar versões específicas de onde estiver ou verificar o responsável por cada alteração e a data dela [8]. Permite ainda reverter arquivos para um estado anterior, reverter um projeto inteiro para um estado anterior, comparar mudanças feitas ao decorrer do tempo, verificar a última alteração que gerou inconsistências no projeto, etc, de forma simplificada. Assim, caso um arquivo do projeto se perca ou se corrompa, seu restabelecimento não será um grande problema.

Já os sistemas de controle de versão distribuídos, tais como Git, Mercurial, Bazaar ou Darcs, são sistemas de controle de versão nos quais os usuários fazem cópias completas do repositório (local no qual os dados gerenciados por um controle de versão estão armazenados). Assim, se um servidor falha, qualquer um dos repositórios dos usuários pode restaurá-lo, e cada *checkout* (verificação) é, na prática, um *backup* completo de todos os dados. Além disso, esses sistemas em geral lidam bem com a existência de vários repositórios remotos passíveis de colaboração, permitindo que você trabalhe em conjunto com diferentes grupos de pessoas, de diversas maneiras, simultaneamente no mesmo projeto. Também permite que se estabeleça diferentes tipos de *workflow* como, por exemplo, o uso de modelos hierárquicos.

Atualmente, empresas da área de tecnologia que não mantém um versionamento distribuído de seus códigos fonte são quase nulas. A importância de se ter um Sistema de Controle de Versões Distribuídos é tanta que Joel Spolsky, co-fundador do Trello e da Fog Creek Software, e CEO da gigante Stack Overflow, em seu teste sobre a qualidade no desenvolvimento de *software* [9] traz este ponto como primeiro questionamento a ser feito para analisar o processo de desenvolvimento.

Entre os existentes no mercado, o *open source* Git, criado pela comunidade de desenvolvedores do Linux (em particular Linus Torvalds, o criador do Linux), está entre os mais populares, sendo muito rápido e confiável. O Git difere de outros sistemas de controle de versão distribuídos principalmente na maneira com que armazena os dados, pois considera-os como um conjunto de *snapshots* (captura de algo em um determinado instante) de um mini-sistema de arquivos. Cada vez que o estado do seu projeto é salvo ou consolidado (*commit*) no Git, é o equivalente a um registro de todos os seus arquivos naquele momento e ao armazenamento de uma referência para esse registro. [8]

Caso nenhum arquivo seja alterado a informação não é armazenada novamente, é criado apenas um link para o arquivo que já foi armazenado. Um esquemático desse funcionamento é descrito na Figura 4.

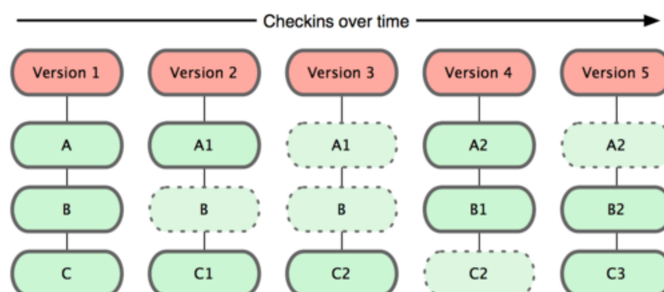

Figura 4 – Armazenamento de dados no Git. Fonte: Git

2.4 Bitbucket

Para manter uma cópia remota de cada repositório de dados, para que todos os usuários autorizados possam acessá-lo, cloná-lo e publicar suas modificações, o Git, necessita de um servidor.

O Bitbucket, da empresa Atlassian, é o serviço *web* de hospedagem de repositórios [10], já utilizado pelo processo de desenvolvimento de *software* atual, que além de facilitar o gerenciamento dos diversos repositórios Git, possui um sistema de segurança, um sistema de controle de acesso e integração com a plataforma de organização JIRA (explicada na seção 2.6). As versões dos códigos de todos os projetos estão armazenadas nele, entretanto, as métricas deveriam ser extraídas apenas das versões remotas completas, ou seja, dos pontos que representam produtos que foram ou serão entregues.

Entre suas ferramentas do Bitbucket mais utilizadas pela empresa estão o Pull Request (PR), integração com o JIRA, e Pipelines. O PR é seu sistema de revisão de alteração de código, no qual configura-se quais pessoas deverão revisar aquele código, podendo aprovar, recusar ou inserir impeditivos (*task*), a ser verificados antes das modificações serem integradas no código principal. Também é possível analisar as alterações pelo seu diferencial ao ponto em que será integrado, pelos *commits* ou por todas as atividades no código contido desde a abertura do PR.

Na Figura 5 é possível verificar na parte superior o estado *MERGED* (já integrado), qual o *branch* de origem e qual o de destino. Na descrição há um resumo do que foi desenvolvido, e em *Files changed* há o resumo do diferencial de todos os arquivos entre os dois *branchs*, seguido pelas linhas de código alteradas.

A integração com o JIRA tem sua importância devido a facilidade de verificação dos requisitos, presentes na tarefa que deu origem aquele código, e também devido a transição automática do estado da atividade, explicado posteriormente na seção 2.6.

O Pipelines é uma ferramenta CI/CD utilizada para agilizar o processo de desenvolvimento de software. Desempenha um papel central na ferramenta desenvolvida, gerando e enviando as métricas do código armazenado pelo Bitbucket.

Durante o PR é possível verificar de forma clara se o código a ser integrado passou em todas as etapas do Pipelines, como mostrado na parte superior a direita da Figura 5, onde há também o número de Pipelines bem sucedidos e seu total. Caso o último executado falhe, o ícone, ao invés de verde, fica vermelho.

Figura 5 – Pull Request do Bitbucket. Fonte: Atlassian

2.5 Pipelines

Os Pipelines são ferramentas de *Continuous Integration* (CI), uma prática de DevOps capaz de realizar tarefas de forma automatizada, agilizando o processo de desenvolvimento de códigos.

O termo DevOps provém da junção das palavras *Development* e *Operations*. Sua característica principal é a automação e monitoramento nas fases de construção do *software*, de sua integração, testes, liberação para se implantar e gerenciamento de infraestrutura. Isso é possível através de ciclos de desenvolvimento menores, com maior frequência na implantação, liberações de versões mais estáveis, com maior qualidade e em alinhamento com os objetivos de negócio da empresa.

Ao longo dos últimos anos a integração contínua evoluiu bastante, e o que começou como um simples processo de automatizar a construção da aplicação (*build*) e testes unitários para cada mudança se transformou em fluxos de trabalho bem mais complexos. Inicialmente as instruções de *build* e testes eram realizadas de uma só vez, sequencialmente (Figura 6.1), sendo bem sucedido apenas se todas as instruções também fossem. Em seguida, desenvolveu-se a ferramenta chamada Pipeline, capaz de suportar estágios separados para cada uma das fases (Figura 6.2), no qual o resultado de cada estágio é antecipado frente

ao final do fluxo [11]. Tal evolução facilitou a identificação de erros, ao isolar o erro em cada estágio, e também permitir que os estágios fossem reutilizados de forma mais simples, devido a sua modularidade.

O Pipeline então progrediu de modo a permitir estágios paralelos (Figura 6.3), possibilitando realizar o teste em mais de um ambiente ou linguagem, e tornando-o bem mais rápido comparado ao fluxo clássico. Dessa forma, um mesmo Pipeline poderia ser executado em mais de uma máquina, chamado de agente, com testes independentes simultâneos, agilizando-os.

Por fim, esta ferramenta evoluiu ainda mais, permitindo fluxos condicionais, seja sequencial, paralelo (através de lógicas *AND* e/ou *OR*), realizando estágios em caso de sucesso ou até mesmo de falha, aumentando em muito a liberdade e flexibilidade da prática CI (Figura 6.4). [11]

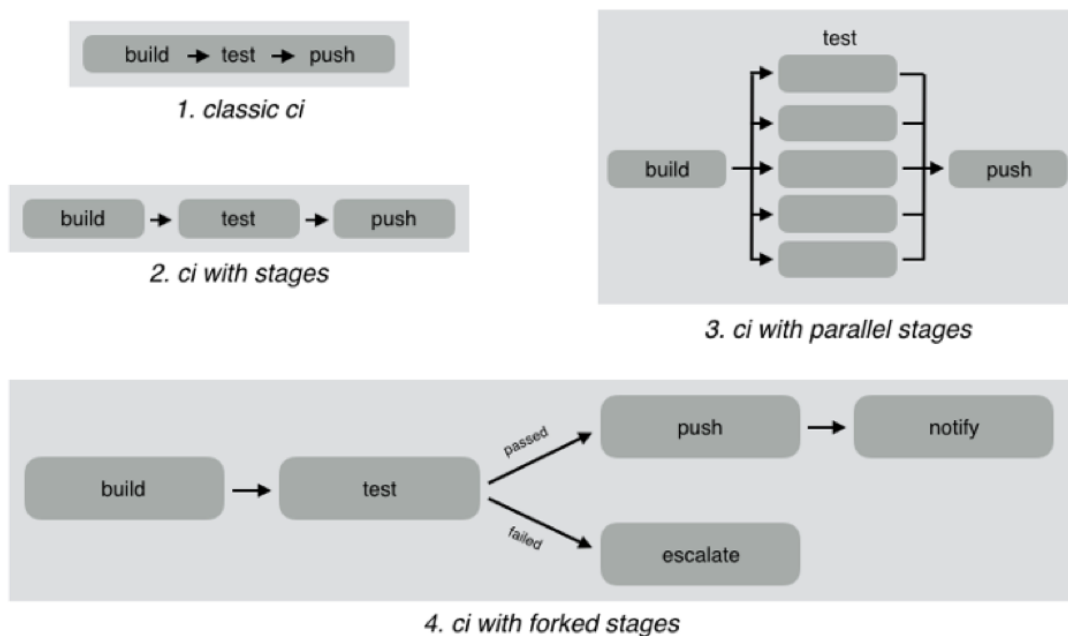

Figura 6 – Evolução dos fluxos de trabalho de tecnologias CI. Fonte: DevOps.com

2.6 JIRA

O JIRA, também da empresa Atlassian, é uma plataforma de desenvolvimento ágil com a qual se organiza o processo de desenvolvimento de *software*. Com ela é possível criar quadros *online* compartilhados com atividades, nas quais pode se criar campos de modo a lhes detalhar, facilitando o entendimento e, consequentemente, o processo de desenvolvimento. Também pode-se acompanhar o progresso das atividades, do projeto em que ela se encontra e de cada usuário [12].

Devido aos campos personalizados das tarefas, pode-se mapear diversos comportamentos e tendências no processo de desenvolvimento de *software*, e assim, o JIRA é outra ferramenta que apresenta papel central para a criação dessa ferramenta.

Sua integração com o Bitbucket também permite o rastreio das atividades de uma plataforma para a outra, bem como a transição automática das atividades quando, por exemplo, um PR é aberto ou concluído.

2.7 Modelo Estrela

A modelagem dimensional, ou multidimensional, é o nome do projeto lógico utilizada normalmente em *data warehouses*, diferente e contrastante da modelagem entidade-relacionamento. De acordo com Kimball, precursores dos conceitos de *data warehouse* e sistemas para análise de dados transacionais, a modelagem dimensional é a única técnica viável para banco de dados (BD) designados para atender um usuário em um *data warehouses*. [13]

O modelo estrela é um tipo de estrutura dimensional, geralmente implementado em um sistema de gerenciamento de BDs relacionais, sendo também um dos esquemas possíveis para a construção de um *data mart*, parte ou segmento de um *data warehouse*. Este modelo se caracteriza por tabelas fatos que referenciam tabelas dimensões através da relação de chaves primárias e estrangeiras. Tabelas dimensões contém atributos descritivos, usado por aplicações de análise de negócios, em inglês *Business Intelligence* (BI), para filtrar e agrupar os fatos [14].

A fim de se criar uma estrutura que facilitasse o acesso do usuário final, ao manipular os fatos em uma aplicação BI através das dimensões, foi utilizado esse tipo de modelagem de banco de dados. No entanto, um *data warehouse*, assim como um *data mart*, necessita de uma base de dados do sistema transacional para alimentá-lo, normalmente um banco de dados de modelagem entidade-relacionamento.

2.8 pgModeler

Com o intuito de realizar a modelagem do banco de dados, foi utilizado a ferramenta *open source* pgModeler, pois além de facilitar o design de um banco de dados de forma gráfica, ou seja, visualmente, também fornece os comandos SQL para a sua criação [15]. Isto permite a diagramação rápida de vários modelos de testes antes de se ter um definitivo, e até visualizar um banco de dados já criado, a fim de um melhor entendimento de sua arquitetura.

2.9 Service-oriented architecture (SOA)

Proposto pela primeira vez em 1996, no artigo "*Service Oriented Architectures*", pelos pesquisadores Roy Schulte e Yefim Natis do Gartner Group, a arquitetura orientada a serviços, ou SOA, é um estilo de computação multi camadas que permite a criação de serviços de negócios interoperáveis que ajudam organizações a compartilhar dados e lógica entre aplicações e empresas [16]. De acordo com a IBM, objetivo da arquitetura orientada a serviços é separar a lógica de integração de negócios da implementação para que um desenvolvedor de integração possa focar na montagem de um aplicativo integrado em vez de nos detalhes da implementação [17].

A criação da ferramenta que compõe esse projeto se baseou na arquitetura orientada a serviços, conceito já explorado pela equipe de desenvolvimento *web* da Hexagon, ao modularizar os componentes da aplicação, separando o código fonte da regra de negócio, e permitindo uma posterior reorganização de sua lógica, se necessário, ou a troca de algum componente.

2.10 Amazon Web Service (AWS)

A Amazon Web Services, ou simplesmente AWS, é a plataforma de nuvem mais adotada e mais abrangente do mundo [18]. Detentora de muitos serviços, executados com muita segurança a confiabilidade, a Amazon proporciona uma infraestrutura completa para negócios orientados a serviços, com preços muito competitivos. A facilidade de se ter serviços em um só ambiente, diminui o tempo de aprendizado, quando comparado a várias plataformas diferentes, e, consequentemente, o tempo de implementação e de manutenção. Por esses motivos, e pelo gerenciamento simplificado oferecido, os serviços da AWS foram utilizados para compor o presente projeto.

2.10.1 Amazon Simple Storage Service (S3)

O Amazon Simple Storage Service (Amazon S3) é um serviço de armazenamento de objetos da Amazon Web Services (AWS) [19]. Fácil de usar, configurar e escalar de forma segura, com a criação de *buckets*, de diferentes tamanhos e para diferentes propósitos. Utilizado na ferramenta a fim de armazenar o histórico das métricas obtidas, e permitir um reprocessamento caso seja necessário. Pode, também, ser facilmente integrado com outros serviços da Amazon, como a Lambda e o Amazon Simple Queue Service.

2.10.2 Amazon Simple Queue Service (SQS)

O Amazon Simple Queue Service (SQS) é um serviço de filas de mensagens gerenciado que permite o desacoplamento e a escalabilidade de microsserviços, sistemas

distribuídos e aplicativos sem servidor ao permitir o envio, armazenamento e a recepção de mensagens entre componentes de *software* em qualquer volume, sem as perder ou precisar que outros serviços estejam disponíveis [20]. Oferece suporte a *dead-letter queues*, usadas como destinos para mensagens que não foram processadas (consumidas) com êxito, evitando a perda de mensagens, e isolando mensagens com erros, facilitando a depuração.

2.10.3 Amazon Relational Database Service (RDS)

O Amazon Relational Database Service (Amazon RDS) é um serviço de banco de dados relacional fácil de configurar, operar, escalar, podendo redimensionar sua capacidade. Permite a utilização de mecanismos de armazenamento como Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle Database e SQL Server [21]. Pode ser utilizado com o Database Migration Service que permite realizar migrações ou replicações em bancos já existentes.

2.10.4 Amazon Lambda

O AWS Lambda é um serviço que permite a execução de códigos sem a necessidade de provisionar ou gerenciar servidores. Ele se encarrega também de alterar sua escala com alta disponibilidade [22]. Pode ser configurado para que seja acionado automaticamente por meio de outros serviços da AWS, como o S3, utilizado no presente trabalho, ou diretamente, usando qualquer aplicativo móvel ou *web*.

2.10.5 Amazon CloudWatch

O Amazon CloudWatch é um serviço de monitoramento e gerenciamento. Fornece dados para monitorar aplicativos, compreender alterações de performance e otimizar a utilização dos recursos utilizados. Ele coleta dados de monitoramento e operações na forma de *logs*, métricas e eventos, e assim oferece uma visualização unificada dos serviços da AWS. [23]

2.11 HashiCorp Terraform

O HashiCorp Terraform é uma ferramenta *open source* que permite criar, modificar e incrementar infraestruturas de forma segura e previsíveis. Ela codifica Interfaces de Programação de Aplicação (APIs) em arquivos de configuração que podem ser compartilhados, tratados como código, editados, revistos e versionados [24]. Muito usada em integração, entre outros, com a AWS, para agilizar o processo de desenvolvimento e de modificação de vários serviços de uma só vez.

2.12 Flyway

O Flyway é uma ferramenta *open source* para migração de banco de dados. Favorece a simplicidade e convenções à configurações. Fundamenta-se em 7 comandos básicos: Migrar, Limpar, Informar, Validar, Desfazer, "Criar uma Base"(Baseline) e Reparar. As migrações podem ser escritas em SQL ou Java. [25]

A ferramenta pode verificar a versão atual e aplicar a migração para a versão mais recente, verificando se todas as versões são válidas. Caso alguma não seja, o Flyway volta o estado do banco de dados à última versão válida, e mostra o erro que impossibilitou a migração total.

3 Planejamento

Buscando aumentar a eficiência da empresa, entendendo as dificuldades e fragilidades do modelo de desenvolvimento atual, foi idealizado um projeto que buscaria fornecer automaticamente análises atualizadas, de todos os processos que utilizam a plataforma JIRA [12] e o serviço de hospedagem de repositórios BitBucket [10].

Para isso, houve a separação em três serviços. Um para atuar nos dados do Bitbucket, outro nos dados do JIRA, ambos inserindo suas métricas em um BD, e mais um de BI, conectado a esse banco de dados, já configurado com a análise desejada, provendo uma interface para levantar diagnósticos do processo.

A Figura 7 representa o diagrama de atividades do Projeto de Análise do Desenvolvimento, contendo seus dois gatilhos, um programado por tempo e outro sendo parte do processo de desenvolvimento de *software*.

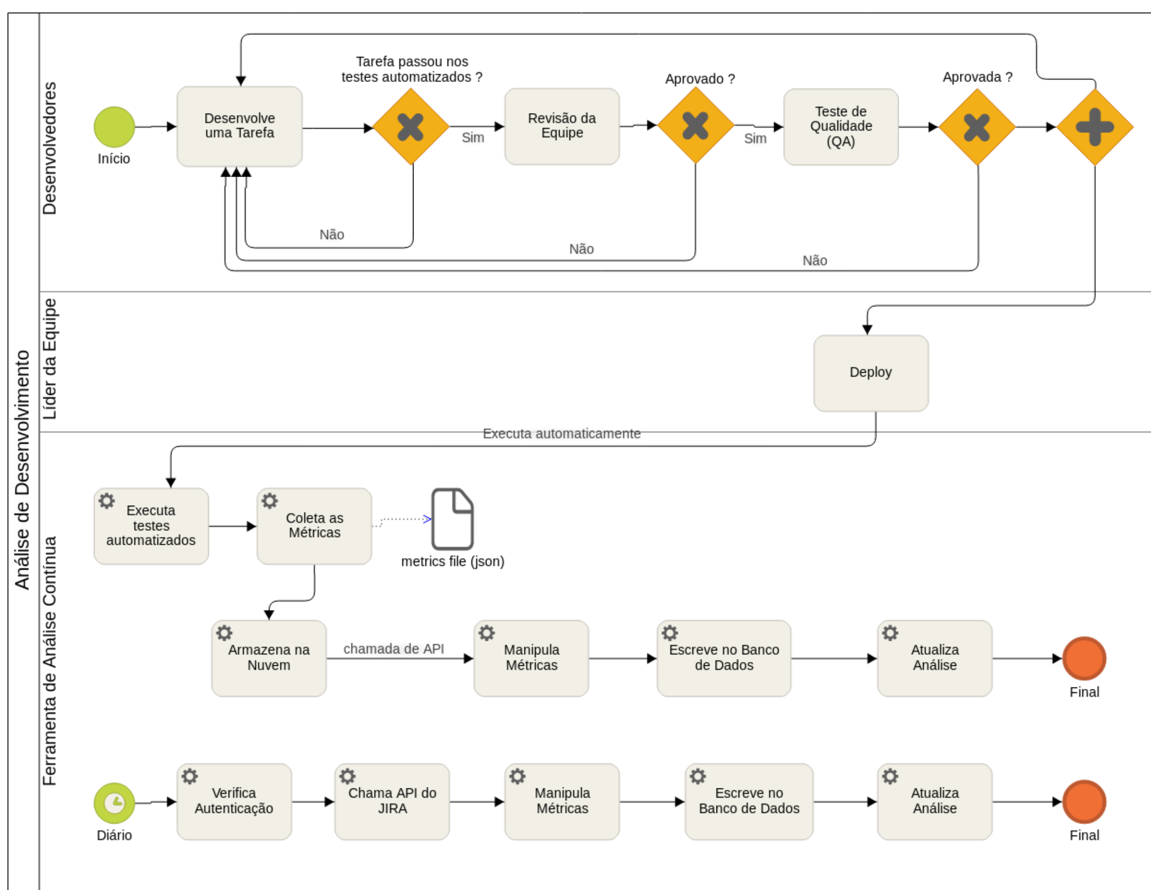

Figura 7 – Fluxo de atividades do Projeto de Análise do Desenvolvimento

A última raia, Ferramenta de Análise Contínua, compreende o funcionamento esperado para o projeto como um todo. Nela, podem ser identificadas os três serviços

citados anteriormente, sendo o primeiro correspondente às cinco atividades posteriores ao Deploy (Executa testes automatizado, Coleta as Métricas, Armazena na Nuvem, Manipula Métricas, Escreve no Banco de Dados), o segundo às primeiras quatro atividades no fluxo programado para ser disparado diariamente (Verifica Autenticação, Chama API do JIRA, Manipula Métricas, Escreve no Banco de Dados), e o terceiro realizando a última atividade dos dois fluxos (Atualiza Análise), consecutivamente. Todas tarefas dessa ferramenta, automatizadas.

O detalhamento dos requisitos funcionais desse sistema, pode ser verificado no diagrama UML (*Unified Modeling Language*) de casos de uso de negócio apresentado na Figura 8. A UML, como linguagem unificada de modelagem, é utilizada para exibir graficamente a especificação e construção do *software*. Os casos de uso mapeiam os requisitos em funcionalidades e interações com atores externos. No diagrama apresentado há dois atores, o *Deployer*, pessoa que disponibiliza ou atualiza o produto no ambiente de produção (que irá ao cliente), e o *Analyzer*, responsável por verificar a análise apresentada, normalmente um líder de equipe, gerente, ou até diretor.

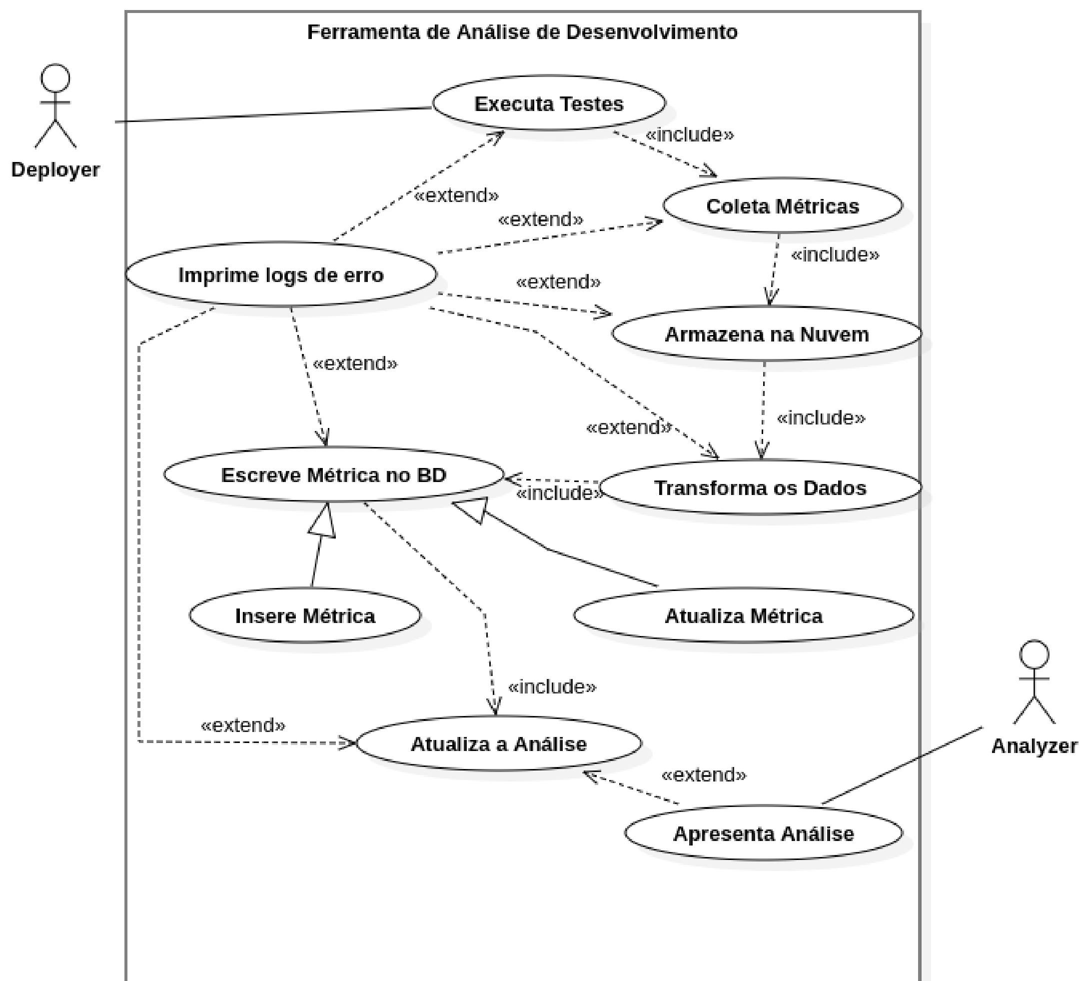

Figura 8 – Diagrama de casos de uso de negócio do Projeto de Análise do Desenvolvimento

Dentre os casos de uso apresentados, a execução dos testes é a interface entre a ferramenta e o *Deployer*, necessária para a criação das métricas. A seguir, a coleta das métricas, deve ser realizada, para a criação do arquivo que deverá ser armazenado na nuvem, a fim de facilitar seu acesso. A transformação dos dados é outro requisito necessário para se escrever informações proficientes no banco de dados. Como mostra o diagrama, tal escrita pode ser generalizada como uma inserção ou atualização das métricas. Atualiza a Análise é o caso de uso responsável por reprocessar os dados contidos no BD para serviço BI, e Apresenta Análise, por exibir os dados já reprocessados durante o acesso à ferramenta de um *Analyzer*. A correta apresentação dos *logs* de erro também é um caso de uso, importante para fins de verificação de problemas, se a ferramenta apresentar comportamentos indevidos, por isso se estende a maioria dos outros casos de uso.

O próximo diagrama UML, diagrama de sequência, descreve a maneira como as diversas ferramentas utilizadas colaboram ao longo do tempo. Este fluxo sequencial, esperado para o primeiro serviço, é melhor detalhado na Figura 9. No decorrer de sua implementação, foram adicionados mais elementos a esse fluxo, como por exemplo o SQS, a fim de gerenciar mensagens não processadas, os quais aumentaram sua complexidade final.

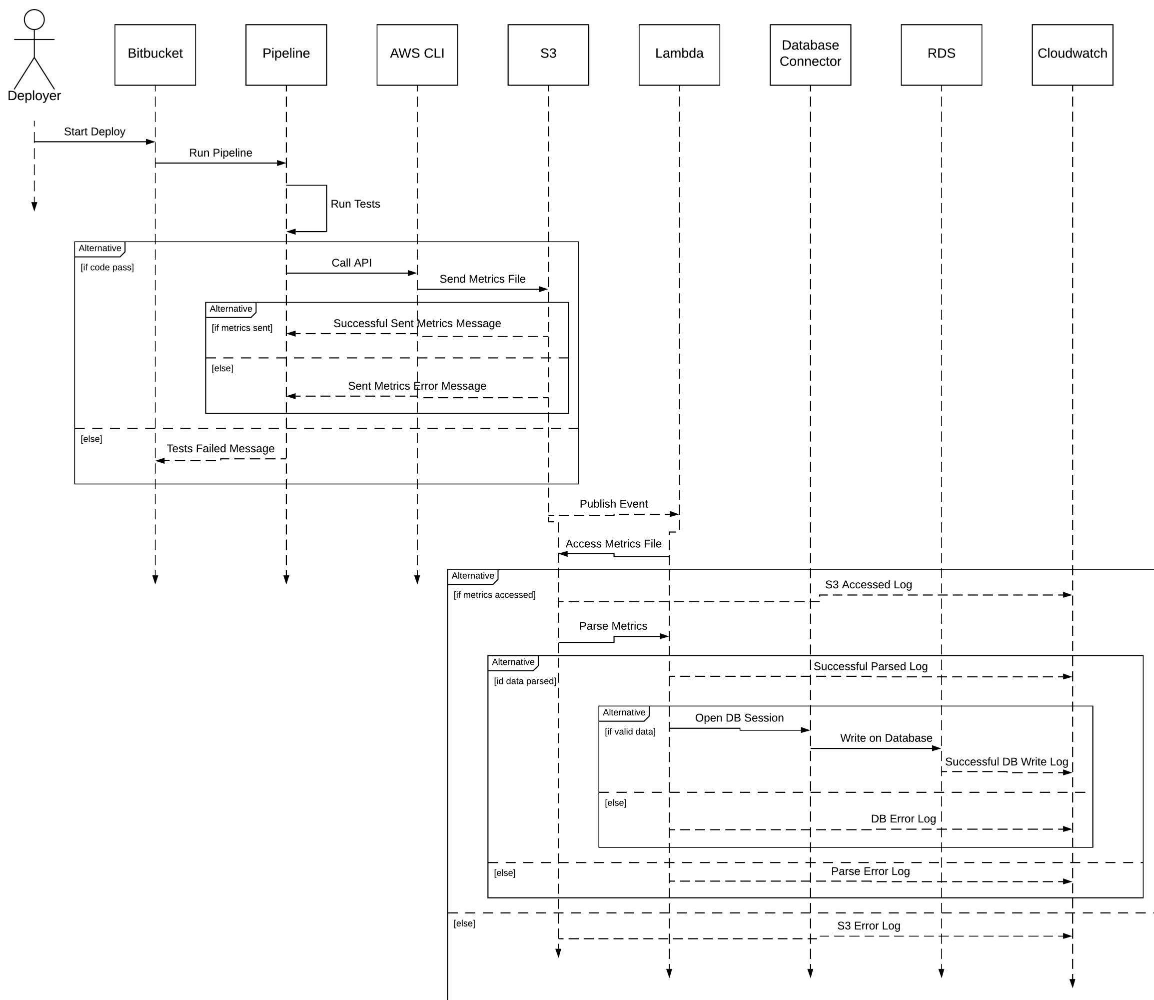

Figura 9 – Diagrama Sequencial do primeiro serviço.

O segundo serviço, tem suas tarefas finais semelhantes, como mostrado na Figura 7, porém não faz uso do Pipeline e do S3, tendo seu gatilho programado por tempo. Dessa forma, o diagrama UML de classes da Figura 10 se mostra mais apropriado para discorrer sobre seu funcionamento. Isto pois, um diagrama de classes é uma representação da estrutura, contendo nome, atributos e métodos, e das relações entre as classes que serão instanciadas, virando objetos no código.

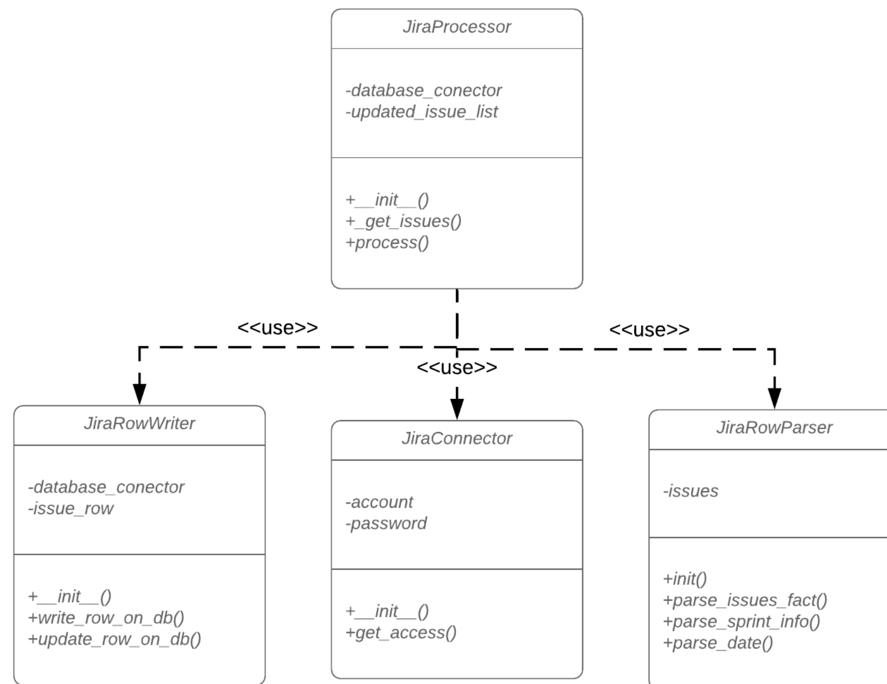

Figura 10 – Diagrama de Classes do segundo serviço.

A parte do diagrama sequencial contendo o acesso ao S3 e toda a sua parte anterior é, nesse serviço substituída pela classe JiraConnector. A parte de realizar as transformações também será feita na Lambda pela classe JiraRowParser, e a escrita no banco de dados pela classe JiraRowWriter, responsável também por verificar a comunicação com o BD, através de seu conector. Por fim temos a classe JiraProcessor orquestrando as demais, e sendo a classe de acesso que a Lambda irá executar.

Sendo implementado através da metodologia de desenvolvimento ágil SCRUM, o presente trabalho foi feito de maneira iterativa e incremental. A cada *Sprint* o autor apresentou seus resultados e obstáculos na realização de cada etapa.

Devido ao grande número de ferramentas, e integrações envolvidas bem como, os requisitos de qualidade de *software*: modularidade, facilidade de compreensão, suporte a trabalho em equipe, facilidade de extensões/alterações, concorrência, e hierarquia; o escopo deste projeto foi reduzido, passando a não mais abranger os três serviços para Análise do Desenvolvimento, mas apenas os dois primeiros, a sua infraestrutura. O terceiro serviço será feito em uma etapa posterior. Por esse fato, as interfaces gráficas que permitiram seu usuário final analisar se o processo está sendo realizado de forma apropriada são a saída do Pipeline, e o CloudWatch.

4 Estudo do Processo de Desenvolvimento

Para se atingir o objetivo deste projeto, desenvolver uma ferramenta para sustentar análises correlacionais entre elementos do processo de desenvolvimento de software e a qualidade do produto final, é necessário inicialmente entender o processo atual. Baseado na metodologia ágil e no SCRUM, apresentados no capítulo anterior, o processo de desenvolvimento de software na Hexagon Agriculture se dá de forma dinâmica e iterativa, em pequenas equipes, focados na qualidade e no cliente.

4.1 Organização das Tarefas

Para organizar o processo de desenvolvimento de uma maneira rápida, visual e de fácil acesso, a empresa usa a plataforma JIRA de gerenciamento, que possui integração com a plataforma de hospedagem de repositórios Bitbucket, oferecendo *links* entre os *branches* e as tarefas e transições automáticas quando, por exemplo, se abre ou fecha um PR. Para melhor organização nessas plataformas é possível separar as tarefas a serem desenvolvidas em projetos, as subdividir em épicos, e também criar campos personalizados para serem preenchidos em sua criação, de modo a categorizar cada atividade, facilitando o seu posterior rastreio e análise.

Campos como o tipo da tarefa (documentação, ferramenta interna, funcionalidade planejada ou correção de *bug*), sua dificuldade estimada, seu tempo de resolução, o tempo em que fica em cada um dos estados (a fazer, em progresso, em revisão, etc) podem trazer informações importantes para se fazer relações como quais projetos apresentam mais *bugs*, quanto tempo é gasto em média para corrigir um *bug*, e consequentemente seu custo, possíveis gargalos, entre outras. Tais relações podem ser usadas nas previsões e no planejamento das próximas etapas dos projeto. Isto pois, a entrega de produto novo e a correção de uma funcionalidade existente têm um impacto direto ao cliente, e dessa forma apresentam mais riscos, principalmente quando essas duas demandas ocorrem ao mesmo tempo.

4.2 Workflow

Durante esta etapa de estudo, o autor verificou que o fluxo de trabalho (*workflow*) continha redundâncias, as quais acarretavam a falta de clareza do estado atual das atividades aos membros da equipe.

Ao não saber o estado de cada atividade, o setor de análise de qualidade (QA) não

sabia quais tarefas já poderiam ser testadas, necessitando verificar constantemente com os desenvolvedores. Por consequência, a SCRUM Master não sabia de uma maneira fácil quais funcionalidades poderiam ser aplicadas à produção.

Para melhorar o fluxo do desenvolvimento, a equipe discutiu e resolveu remover o estado redundante apontado pelo autor, bem como renomear outros, para melhor entendimento de todos. Esta medida também facilitou a automação já fornecido pelo JIRA, de transição do estado das tarefas em seu quadro.

Outra decisão tomada em conjunto ao se analisar o *workflow* está relacionada com o QA. Concluiu-se que o processo do QA estava a parte, mas sucedia o dos desenvolvedores, e por esse motivo deveria ter um quadro de atividades separado. Dessa forma, seria possível também que a SCRUM Master soubesse de forma rápida e fácil as tarefas a serem testadas, as em teste, as que poderiam ser colocadas em produção e as que já estavam.

Utilizando os recursos do JIRA, foi também possível ligar os dois quadros. A última coluna do primeiro quadro, que indicava o término do desenvolvimento da tarefa, passou a enviar as tarefas para a primeira coluna do outro quadro, indicando que elas deveriam ser testadas. Tal medida otimizou o trabalho de todos os envolvidos e deu uma melhor visibilidade a transição entre a etapa de desenvolvimento e a de teste.

O fluxo de trabalho resultante, após essas modificações, pode ser observado na Figura 11. O retângulo vermelho pontilhado representa o novo quadro criado, enquanto os estados fora dele representam o que permanece no quadro original. O estado TO TEST faz a ligação entre os quadros, sendo sua intersecção.

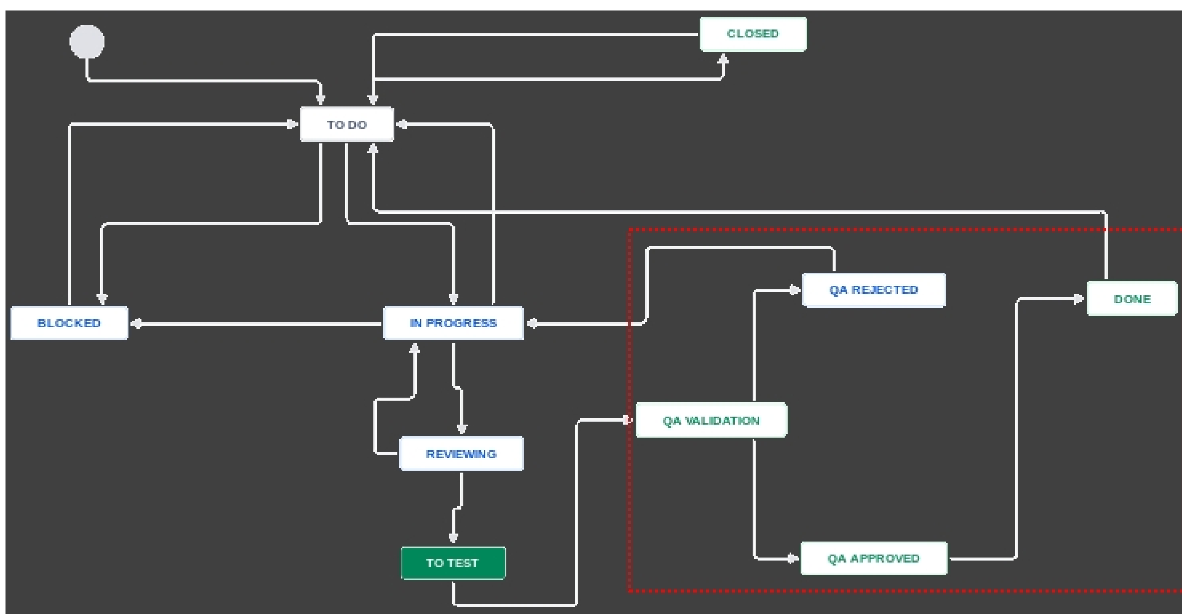

Figura 11 – Fluxo de trabalho atual. Fonte: Hexagon Agriculture

4.3 Pipeline

Como dito anteriormente, o Bitbucket oferece várias facilidades. Uma delas é a ferramenta de integração contínua chamada Pipeline. Incrivelmente personalizável, esta ferramenta pode ser ativada de forma manual ou de forma automática em todos os *branches* ou em *branches* específicos. Sendo personalizável também as suas etapas e os *scripts* e comandos que serão executados em cada uma.

A empresa utiliza o Pipeline para garantir uma integração contínua, testando de forma automática os códigos desenvolvidos em pequenos pedaços, evitando que novos códigos sejam construídos em cima de algum erro. Todas as mudanças passam pelos devidos testes estáticos (ex.: pylint, flake8, etc), testes unitários, de integração, integridade dos serviços na AWS e banco de dados, entre outros, evitando que desenvolvedores percam tempo revisando um PR que não está correto (não passou nos testes) e seu retrabalho não impacta o produto, mantendo a sua qualidade.

Por já ser executado junto ao *branch* automaticamente, o Pipeline foi escolhido para enviar as informações contendo as métricas das modificações feitas.

4.4 Métricas

Com o objetivo de relacionar a qualidade do produto e o processo de desenvolvimento de seu software, as métricas a serem coletadas devem trazer informações relevantes para as análises. Quantificar a qualidade de um produto não é uma tarefa fácil, mas as métricas escolhidas para indicar esse ponto foram a quantidade e a criticidade dos *bugs* encontrados no produto, seja pelo cliente ou pelo setor de QA em uma versão já lançada. Outras opções poderiam ser escolhidas, como a quantidade de tarefas abertas para projetos já finalizados, tempo de implementação, tempo de inserção de nova funcionalidade, e também medidas relativas, referentes ao total de pessoas utilizando perante aos possíveis usuários, ou seja, que já adquiriram o produto.

A escolha dessas métricas, em especial, leva em conta o impacto desse *bug* no funcionamento do produto: uma falha na autenticação de um serviço afeta muito mais o cliente, pois o impede de usar o serviço ou até expõe seus dados, do que uma mensagem exibida sem acento, por exemplo. Esses *bugs* podem ser mensurados automaticamente através de uma consulta via API, pois cada um deles se transforma em uma tarefa a ser feita no JIRA com o tipo *Bug Fix* (correção de erro) e sua respectiva prioridade.

Para a coleta ser mais abrangente, permitindo uma maior gama de questionamentos, a coleta engloba as atividades independentemente de seu tipo, com suas respectivas informações do *Sprint* e do Projeto ao qual se encontram, tamanho, seu tempo de desenvolvimento, entre outras informações. Assim, é possível fazer relações como, por

exemplo, se projetos com tarefas menores são menos suscetíveis a *bugs*, ou projetos com tarefas mais detalhadas (diretamente relacionado com o tamanho da descrição), ou outros questionamentos que podem surgir eventualmente.

Ao final de cada tarefa também é possível coletar informações do desenvolvimento do código, como cobertura de testes do código, número de testes, número de linhas, etc. O que permite a criação de mais algumas relações como, por exemplo, se uma boa cobertura de código em um projeto traz uma diminuição do tempo de manutenção (tempo de desenvolvimento da tarefa *Bug Fix*); se projetos com alto número de testes unitários são menos propensos a gerar *bugs*; ou se os testes atuais na verdade não geram um ganho na qualidade do código o que indicaria que o processo poderia ser melhorado de alguma outra forma.

5 Coleta de Dados

A execução da etapa de coleta de dados exige uma boa definição da sua arquitetura, ou seja, de como será a extração e o armazenamento deste grande volume de dados, de modo que seja confiável, seguro, de fácil entendimento e, consequentemente, de fácil manutenção, e de uma maneira que possa ser reutilizado.

Para atender estes requisitos, a arquitetura definida, de maneira simplificada, parte das duas fontes de informações disponíveis, dos repositórios Git, que mandam seus dados através dos pipelines executados nos servidores do Bitbucket, armazenando-os no S3, e do JIRA. Em seguida esses dados são processados pelas funções lambda e armazenados no banco de dados relacional RDS para então serem analisados. Uma melhor visualização do processo como um todo, é apresentada no fluxograma da Figura 12.

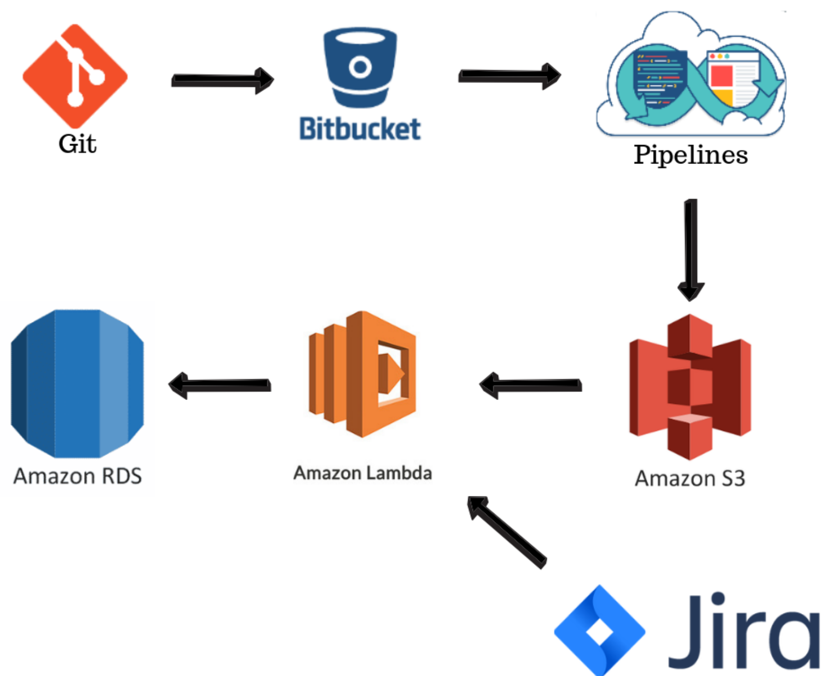

Figura 12 – Fluxograma da coleta de dados por serviço.

A arquitetura do projeto prevê que cada bloco funcione de forma modular, facilitando a compreensão do processo, visando o isolamento de falhas, uma manutenção ágil, a reutilização do código, a replicação para novas análises, o versionamento e o incremento de novos blocos. A primeira etapa para a aquisição de dados, então, é a definição do banco de dados. Apenas com a modelagem do banco de dados pronta, as métricas podem ser adquiridas na estrutura correta, e para um objetivo específico, no caso, o deste projeto.

5.1 Modelagem do Banco de Dados

Neste projeto, como há duas fontes de dados separadas, o código fonte e o JIRA, que irão receber tratamentos diferentes mas que terão seus dados cruzados nas análises, foi escolhido por mantê-los em um mesmo banco de dados, mas em esquemas diferentes, facilitando também sua identificação e inserção.

Cada um dos esquemas terá como estrutura o esquema estrela, apresentado na seção 2.7, devido ao seu alto desempenho e facilidade durante a consulta dos dados para sua análise. A tarefa de modelar um banco de dados, pode ser facilitada por ferramentas, como o pgModeler (seção 2.8), que permitem importar e exportar bancos de dados, interligando o banco de dados físico e sua representação visual, o modelo ER, e possibilitando alterações em tempo real da estrutura do BD graficamente.

5.1.1 Schema Repositories

O esquema feito para armazenar o código fonte dos produtos foi criado para armazenar as métricas de dois fluxos de desenvolvimento diferentes, referente ao desenvolvimento de *software* para o setor dos sistemas embarcados e para o setor dos micro serviços *web*. Para isso ser possível, foram necessárias algumas iterações no processo de arquitetura, até chegar ao modelo atual, apresentado na Figura 13. Este modelo, mesmo apresentando uma tabela fato normalizada e a maioria das outras desnormalizadas, elas não representam uma implementação de um *data mart* propriamente dito. Isto porque suas tabelas "dimensão" não são qualitativas mas sim, em sua maioria, quantitativas.

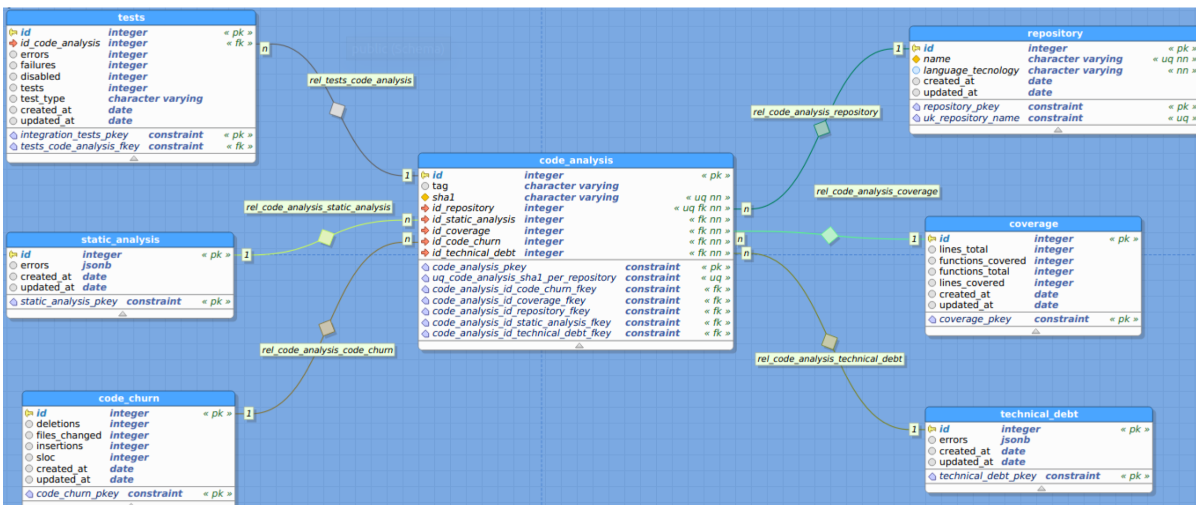

Figura 13 – Esquema para armazenamento das métricas dos códigos fontes

Visando um maior poder de análise, a coleta de dados deve ser a mais completa possível, mas atentando-se ao custo e a velocidade de processamento. Os dados provenientes dos repositórios não estão armazenados em nenhum lugar. Dependem, assim, da execução

de vários *scripts* para a extração e transformação de suas métricas a cada ponto do desenvolvimento. A modelagem foi feita para armazenar todos esses dados, possibilitando uma fonte organizada de informações a ser consultada, e até a criação de um outro esquema que o use como base, caso novas análises sejam necessárias.

A inversão da normalização, quando comparado às outras relações, entre a tabela fato e a tabela testes se deve aos múltiplos testes para cada código, todos com a mesma estrutura de saída.

Mesmo havendo múltiplos tipos de débitos e de ferramentas para análise estática, a inversão da normalização não foi implementada, mas sim um atributo do tipo json binário. Tal decisão considerou que as diferenças entre os tipos de erros e as ferramentas de análise estática criariam uma grande quantidade de atributos em cada tabela, e com cada linha usando apenas os atributos que lhe são pertinentes a tabela ficaria esparsa, perdendo desempenho. A solução através de um atributo json permite armazenar uma quantidade variável de atributos, de tipos também variáveis.

5.1.2 Schema Jira

O segundo esquema projetado tem como objetivo armazenar os dados provenientes do JIRA de uma forma organizada. Pela quantidade de tarefas e também de informações atreladas a cada uma, os campos a serem coletados precisaram ser melhor selecionados neste primeiro momento, mas buscando não diminuir o poder de análise esperado.

Devido a estrutura fornecida pelo Jira, com as tarefas já armazenadas, o modelo apresentado na Figura 14 pôde seguir as recomendações de Kimball, se aproximando de uma estrutura de *data mart*. Tal característica é notada pela desnormalização das tabelas dimensões e atributos predominantementes qualitativos, com a finalidade de servirem de filtros e elementos de agrupamento, para consultas e análises das informações nele presentes.

Ao analisar a tabela fato é possível verificar os atributos necessários para responder algumas das perguntas levantadas no capítulo 4, combinando o tipo da tarefa, sua origem, seus pontos, entre outros. com as dimensões. Dessa forma, possibilita-se fazer análises especificadas por projeto em determinados anos, meses e até dias, caso haja a suspeita, por exemplo, de que *bugs* estão relacionados com códigos testados no final de tarde das sextas feiras.

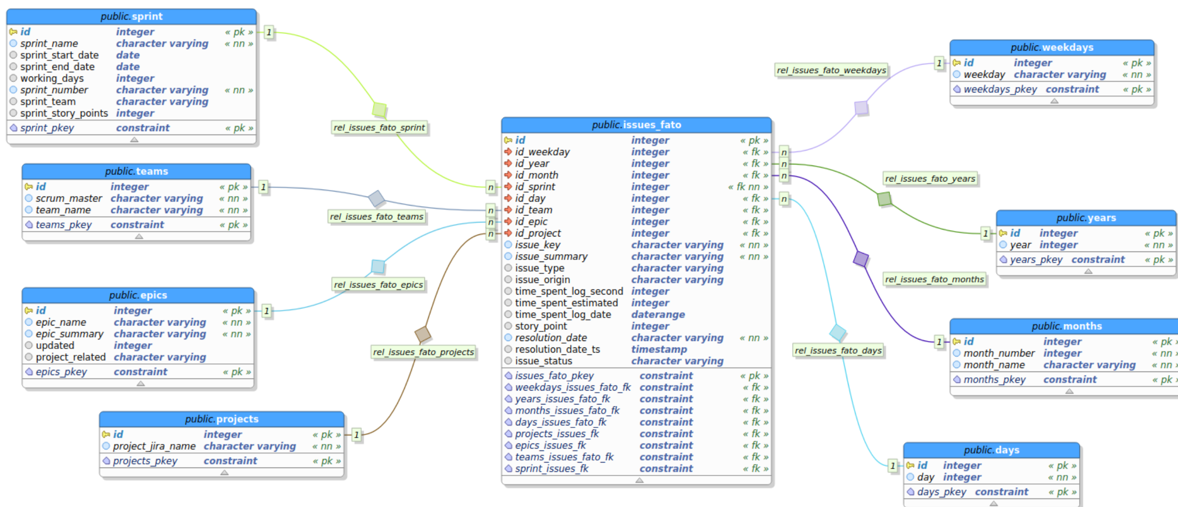

Figura 14 – Esquema do banco de dados para análise de código

5.2 Implantação do Banco de Dados

A implantação do banco de dados utilizou o recurso de visualização de código da ferramenta pgModeler gerando assim o código SQL a partir dos modelos ER feitos. O próximo passo foi a preparação dos códigos SQL para a utilização da ferramenta de migração de banco de dados Flyway.

Mesmo adicionando um certo grau de complexidade e, consequentemente, tempo investido, a decisão de utilizar essa ferramenta se justifica pela atualização automática a partir de qualquer versão do banco de dados para sua versão mais recente. Isso possibilita sua replicação em outros ambientes de maneira fácil, mesmo que haja alterações futuras. Tal decisão, além de auxiliar a configuração de bancos de dados locais com a mesma estrutura para testes, também converge para um dos questionamentos feitos no teste sobre a qualidade no desenvolvimento de software de Spolsky mencionado na seção 2.3, sobre construção da aplicação em apenas um passo.

5.3 Extração dos Dados - Repositórios

O projeto foi arquitetado de uma forma modular, visando o isolamento de falhas, a facilidade na replicação para novas análises, e o versionamento, para incrementá-lo de maneira mais fácil. Por isso, a solução apresenta o envio dos dados dos repositórios inicialmente para o S3, o qual apenas armazena o arquivo de métricas. Caso seja necessário mudar os dados dentro do arquivo de métricas, ou a origem do *upload*, seu *bucket* não se altera. Também, na necessidade de um reprocessamento dos dados, todos os arquivos de métricas ao longo do tempo estão armazenados, organizados automaticamente por projetos.

Este *upload* do arquivo de métricas ao S3 é realizado através de um comando durante a execução do Pipeline do Bitbucket. Esta decisão foi tomada por já ser a ferramenta de integração contínua (CI) e entrega contínua (CD) utilizada pela equipe de desenvolvimento *web*, aproveitando sua integração. É importante que tal ação seja independente, pois no caso de uma falha na inserção no *bucket* da AWS, o Pipeline iria falhar, acusando um falso negativo, levando o desenvolvedor a acreditar que suas modificações foram a causa.

A função Lambda também modular, é responsável por processar esses dados assim que chegarem ao S3, e enviá-los ao BD já na estrutura correta, como foi modelado anteriormente.

5.3.1 Scripts para extração das Métricas

Ao início da implementação, optou-se por analisar em primeiro momento apenas os dados provenientes da equipe de desenvolvimento *web*, por conta de uma priorização da empresa. Também por esse motivo, os dados a serem adquiridos foram limitados a apenas aqueles que já eram gerados.

Dessa forma, foi necessário uma alteração nos *scripts* de testes já existentes. As informações apresentadas apenas na tela passaram a ser direcionadas para um arquivo de formato específico de fácil interpretação e manipulação.

Sendo assim, os testes e comando, já alterados, passaram retornar seus resultados filtrados, se tornando parâmetros na execução de um novo *script*, de transformação. Este último foi desenvolvido especificamente pelo autor para organizar e estruturar todas as métricas, em um arquivo json. Tal arquivo contém em seu nome o código *hash* do *commit* atual, para facilitar sua identificação.

Tal etapa apresentou problemas em uma implementação, pois seu comportamento local diferia de seu comportamento remoto, no servidor do Bitbucket. Pelo processo de funcionamento do próprio Pipeline, uma imagem Docker é instanciada, gerando um container Docker, no qual clona-se o repositório Git. Entretanto, seu diretório de trabalho (*working directory*) continua em um nível acima, uma pasta do Pipeline chamada *build*, que coordena a execução da ferramenta.

Este problema aparentemente simples não possui uma forma de rastreio de erro, apenas falhando comandos e *scripts* de testes que dependiam de caminhos absolutos, dificultando seu entendimento e correção. Ao ser identificado foi realizado um contorno que verificava as duas possibilidades, caso a execução fosse local ou remota.

5.3.2 Pipeline

O próximo passo foi enviar essas informações ao serviço S3 da AWS. Para isso foi utilizado a interface para linha de comando (CLI) [26], também da Amazon. Assim, foi

necessário uma verificação inicial da imagem Docker utilizada, de modo que o ambiente de execução do Pipeline possua as bibliotecas necessárias para a realização do comando a fim de possibilitar o envio. Em seguida, foi modificado o Pipeline padrão, já utilizado pela empresa, direcionando a saídas dos testes realizados aos *scripts* desenvolvidos, citados na seção anterior. Assim, a ferramenta CI/CD do Bitbucket passa gerar as métricas, no formato desejado, cria o arquivo com o nome apropriado, e o envia ao S3, no diretório específico de seu repositório. Esta etapa, ou *step*, como é chamado nos Pipelines, foi configurado para ser executado apenas quando o PR é testado, aprovado e colocado em produção (*deploy*). Ou seja, as métricas a serem armazenadas se restringem a versões oficiais, com funcionalidades completas, prontas para serem liberadas aos clientes. Após desenvolvido e testado, este *step* foi adicionado aos Pipelines de outros repositórios, dando início ao armazenamento de dados.

5.3.3 Lambda

Com o arquivo json no S3, foi necessário desenvolver o serviço que iria processá-lo e adicionar suas métricas no BD. Evitando a necessidade de provisionar ou gerenciar servidores, com um custo de processamento sob demanda e com escalabilidade, o serviço Lambda da AWS foi escolhido como infraestrutura dessa tarefa. Essa decisão visa um baixo custo de operação, e caso sejam necessárias várias execuções do serviço ao mesmo tempo, a AWS gerencia o paralelismo, sem perder performance.

A função a ser inserida dentro do serviço Lambda, desenvolvida neste projeto, tem seu código versionado por um repositório Git no Bitbucket. Nele, estão armazenados não apenas seus *scripts* de execução e as bibliotecas necessárias, mas toda a sua infraestrutura de criação, a qual contém outros serviços, como, por exemplo, sua políticas de autorização, o S3 e suas ligações, através da configuração existente, em formato de código, utilizada pelo Terraform.

O gatilho para disparar a Lambda é qualquer arquivo que seja enviado a um *bucket* específico no S3, contendo um prefixo definido e o formato json. Esta configuração e também o *bucket* foram criados também através Terraform, e por estarem armazenados, caso a função falhe, o rastreio do erro será apresentado e os dados do S3 não serão perdidos. Adicionalmente, com o serviço SQS, os arquivos a serem processados pela Lambda são colocados em uma fila de sua gerência. Se algum não for bem sucedido, ele irá para a fila *dead-letter* como explicado na seção 2.10.2, a qual indica quais arquivos não foram processados. Esta estrutura evidencia a origem de algum erro, facilitando a identificação de sua ocorrência nos *scripts* de geração das métricas, no Pipeline que as enviam, no *bucket* que as armazenam, na lambda que as processam ou no banco de dados.

O código do processamento das métricas foi inteiro escrito em python. Utiliza a biblioteca de mapeamento objeto-relacional chamada SQLAlchemy [27], de modo a abstrair

comandos SQL, e facilitar o desenvolvimento em apenas uma linguagem, com relações claras. Tal *script* trata as métricas, e as escreve no banco de dados, nas várias tabelas definidas anteriormente, bem como escreve suas relações. Também foram implementados alguns testes unitários para a verificação de seu código.

5.4 Extração dos Dados - JIRA

Como ilustrado anteriormente na Figura 12, a extração de dados do JIRA segue um fluxo diferente das métricas dos repositórios, pois suas informações ficam armazenadas no servidor da Hexagon Agriculture na Atlassian, e a chamada de uma API pode obtê-las. Como não há a necessidade de armazená-las novamente, não há a necessidade da utilização do S3. Por esse motivo, esta extração de dados se resume ao processamento das métricas e escrita no banco de dados.

Mesmo aparentando ser mais simples, por não gerar um arquivo e enviá-lo à nuvem, o código desta Lambda apresentou, no entanto, um grau de dificuldade elevado quando comparado ao da primeira desenvolvida, devido a quantidade de campos, formato de retorno da API e possibilidades de casos a serem tratados.

5.4.1 Lambda

Com o avanço do estudo na linguagem de programação Python, e nas boas práticas de programação, aprendidos no decorrer do projeto, o planejamento do segundo serviço, citado no capítulo 3 foi incrementado. Com uma rápida comparação entre o diagrama de classes proposto, na Figura 10 e o diagrama de classes atual, na Figura 15, é possível visualizar essa mudança.

Seu desenvolvimento inicia com a conexão do *script* ao servidor contendo os dados das atividades, *sprints* e projetos do processo de desenvolvimento de software da empresa. Para isso se criou a classe JiraApi, a fim de fazer o acesso à API do JIRA e obter as *issues* (tarefas) lá armazenadas. Como tais informações são privilegiadas, são necessárias chaves de acesso, que, por segurança, não podem estar presentes do código da Lambda (prá tica conhecida como *hard coded*) e nem no repositório.

A fim de proteger as credenciais foi utilizado o protocolo de autorização OAuth 2.0, pois permite o login sem expor as senhas. Projetado especificamente para trabalhar com o Protocolo de Transferência de Hipertexto (HTTP), o OAuth especifica um processo para que proprietários do recurso possam emitir *tokens* de acesso para clientes. Este acesso é mediante autorização do servidor e aprovação do proprietário do recurso. Com o intuito de também não deixar esses *tokens hard coded*, foi usado outro serviço da AWS, o Systems Manager (SSM), que criptografa parâmetros, e somente permite o acesso de serviços autorizados. A criação da classe JiraCredentials se deu pela necessidade de isolar

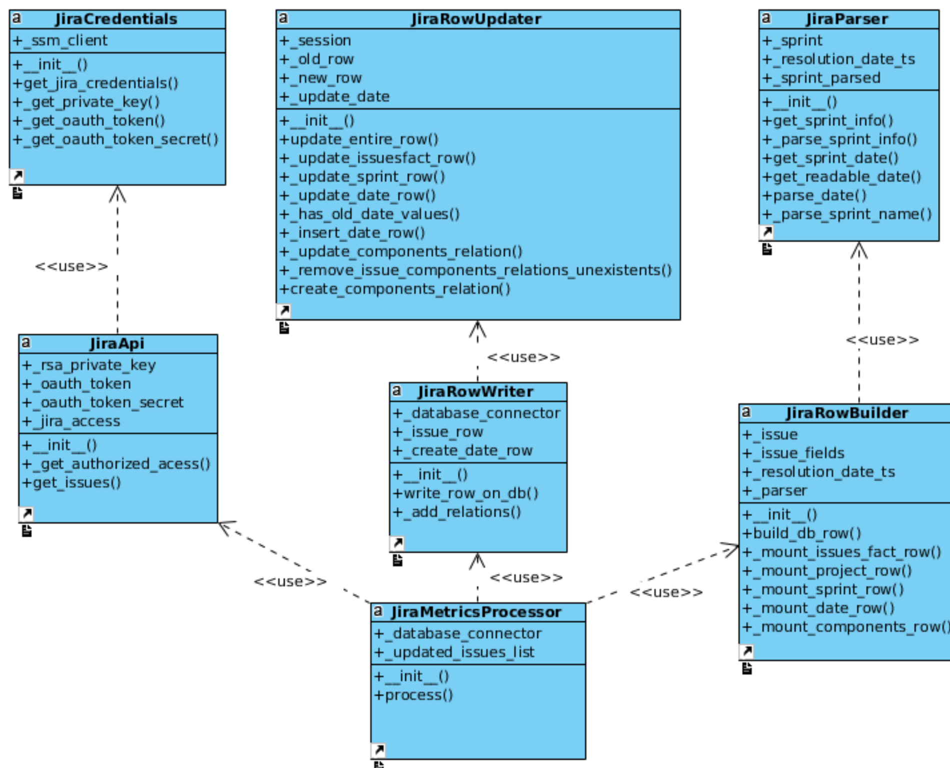

Figura 15 – Diagrama de Classes Final do primeiro serviço.

a chamada da API da aquisição das credenciais, deixando-a apenas para retornar os *tokens* presentes no SSM.

Após a criação da estrutura para proteger o acesso ao servidor, houve uma etapa de análise do retorno da API, provido pela biblioteca escolhida, que abstrai sua chamada direta. Em seguida foram criadas as classes JiraRowBuilder e JiraParses, responsáveis, respectivamente, por gerar as linhas a serem inseridas no banco de dados, e por dividir, limpar e tratar as métricas. A primeira classe, se concentra apenas na utilização das métricas prontas, organizando-as em objetos do SQLAlchemy para facilitar a escrita, enquanto a segunda concentra-se nas métricas que precisarem ser transformadas.

A última etapa, criação da classe JiraRowWriter, realiza o acesso ao banco de dados, abrindo uma sessão para escrever os dados. Como o modelo utilizado para este BD se aproxima bastante de um *data mart*, com dimensões com atributos qualitativos, antes de realizar a adição da nova coluna ao banco, é necessário verificar se aquela "qualidade" já existe, utilizando seu identificador para a criação de uma chave estrangeira na tabela fato caso exista, tarefa facilitada pela biblioteca SQLAlchemy.

Ainda na etapa de escrita, a tabela fato precisa de uma verificação para saber se será adicionada, ou atualizada, o que muda bastante a sua implementação. Caso a tarefa a ser adicionada já exista só é preciso escrever os dados novos, ou diferentes dos que já estão lá, otimizando o tempo de processamento. Dessa forma, há a criação da classe

JiraRowUpdater, que será instanciada pela JiraRowWriter sempre que a validação para atualização for positiva.

Devido a esta arquitetura, cada umas das etapas mencionadas, Acesso as Credenciais, Acesso ao Jira, Construção das Métricas em Linhas do BD, Transformação das métricas, Escrita no BD e Atualização do BD, foram desenvolvidas separadamente com funcionamentos independentes, sendo executadas pela classe JiraMetricsProcessor. Por esse motivo o desenvolvimento de testes unitários, para garantir o funcionamento esperado do código, foi facilitado. Cada uma das classes citadas tem seus próprios testes, e qualquer alteração no código deverá ser realizada evitando sua falha. O teste da classe JiraMetricsProcessor realiza a integração de todas as demais.

A criação da Lambda, com seu código e bibliotecas, o serviço de armazenamento criptografado das chaves de acesso do SSM, as políticas de autorização, e interações entre os serviços, foram implementadas também pela ferramenta Terraform. Ainda por razões desconhecidas, se verificou um erro com o *script* na AWS Lambda. A biblioteca utilizada para acessar o servidor da Hexagon na Atlassian não funcionou no ambiente da Amazon. O contorno efetivo da situação só veio após alguns dias, com uma nova biblioteca. Apesar do código ter sido desenvolvido de maneira a ter as suas partes independentes, a troca, que deveria ocorrer apenas na primeira etapa, afetou também a segunda. O retorno da API, novamente abstraído, era diferente do primeiro, demandando nova análise e nova implementação.

Por fim foi configurado, via Terraform, um gatilho para esta Lambda executar diariamente, em dias úteis, em um horário específico, coletando sempre as métricas do último dia útil.

6 Conclusão

Durante os três meses de desenvolvimento deste projeto, foram estudados no mínimo 15 ferramentas e conceitos, além de diversas bibliotecas Python, que culminaram na criação da ferramenta de infraestrutura para análise do processo de desenvolvimento de *software* proposta no começo do documento. Considerando seu tempo de implementação, o autor fica muito satisfeito com o resultado desse trabalho, tendo concluído a ferramenta proposta e, ainda, evidenciando dúvidas que haviam no fluxo do processo de desenvolvimento, o que ajudou a equipe a reorganizá-lo de modo claro.

A coleta dos dados dos repositórios, antes feita mensalmente, através da execução manual dos testes e da cópia das métricas para uma planilha, se tornou automática e estruturada, podendo ser acessada diariamente com dados atualizados.

Apenas no setor de desenvolvimento de *software web* há mais de 20 repositórios, portando, extrair suas métricas além de fatigante, ocupa tempo. A Figura 16 exemplifica com estrutura real, mas dados fictícios, a planilha que era preenchida antes da realização da sua automação.

Repo1	**28/11**	**06/02**	**26/02**	**01/04**	**24/04**	**22/05**	**23/06**
sloccount	3.306	3.227	3.380	3.399	3677	3677	4028
Pylint score (max 10)	8,97	9,01	9,06	9,06	9,06	9,06	9,12
Flake8 errors	9	0	0	0	0	0	0
Test Coverage	83%	87%	87%	87%	87%	87%	85%
Number of tests	233	308	332	335	370	370	412
Repo2	**28/11**	**06/02**	**26/02**	**01/04**	**24/04**	**22/05**	**23/06**
sloccount	283	352	352	352	352	352	362
Pylint score	9,27	9,76	9,76	9,76	9,76	9,73	9,73
Flake8 errors	1	0	0	0	0	0	0
Coverage	98%	99%	99%	99%	99%	99%	99%
Number of tests	45	68	68	68	68	69	69
Repo3	**28/11**	**06/02**	**26/02**	**01/04**	**24/04**	**22/05**	**23/06**
sloccount	141	166	166	163	163	164	164
Pylint score	8,36	8,55	8,55	8,52	8,52	8,52	8,52
Flake8 errors	0	0	0	0	0	0	0
Coverage	99%	97%	97%	96%	96%	96%	96%
Number of tests	36	38	38	40	40	40	40
Repo4		**06/02**	**26/02**	**01/04**	**24/04**	**22/05**	**23/06**
sloccount		486	568	568	568	573	608
Pylint score		9,73	9,71	9,71	9,71	9,75	9,77
Flake8 errors		0	0	0	0	0	0
Coverage		92%	92%	92%	95%	95%	95%
Number of tests		55	63	63	64	65	74

Project 1 | Project 2 | Project 3 | Project 4 | Project 5

Figura 16 – Coleta de dados anterior ao trabalho.

Esta nova funcionalidade decorreu de diversas decisões em sua arquitetura e implementação, explicitadas no decorrer deste relatório, evidenciando o conhecimento adquirido e a necessidade de um planejamento, a fim de se evitar retrabalhos. Ainda assim,

o trabalho se mostrou mais extenso e complexo do que o esperado, devido, entre outras coisas, à curva de aprendizado das ferramentas e bibliotecas, em especial o Terraform, e ao retrabalho do código de uma das Lambdas, devido a sua incompatibilidade com o ambiente utilizado.

Outros fatores que contribuíram para sua dificuldade foram os requisitos de segurança, implementado com o protocolo OAuth e as chaves criptografadas no AWS SSM, confiabilidade, com diversos testes unitários e verificação do comportamento da Lambda, clareza no código, através do estilo de código PEP 8, e a repetibilidade, visando expandir a estrutura desenvolvida para outros focos de análise.

A modularidade, outro requisito, apresentado recorrentemente, foi muito bem explorada. No fluxo ETL dos repositórios, as etapas de extração, transformação e criação do arquivo de métricas, armazenamento intermediário no S3, e armazenamento no banco de dados das métricas foram feitas através de diferentes serviços. Implementados, consecutivamente, pela modificação dos testes nos repositórios, criação de um *script* executado no Pipeline, utilização da AWS CLI para armazenar no S3, e pelo desenvolvimento de uma Lambda disparada automaticamente, enviando as métricas ao RDS.

O resultado desta etapa é demonstrado com a coleta automática das métricas de qualidade dos códigos a cada nova versão para cada projeto. Adicionalmente, esta estrutura desenvolvida foi inserida no repositório da empresa, responsável por facilitar a criação de outros repositórios. Dessa forma, novos projetos serão monitorados desde sua origem de forma também automática.

No outro fluxo ETL, das tarefas do JIRA, a modularidade vem com a estrutura do código desenvolvido para a função Lambda apresentada no diagrama UML de classes, Figura 15. A crescente complexidade desta parte do trabalho frente ao planejado, aliada com os requisitos, motivaram a criação das classes adicionais. A nova arquitetura, buscou dividir não apenas fisicamente o código, mas orientado-o a objetos, o que trouxe modularidade, clareza e organização ao código, facilitando seu entendimento. Cada classe pôde ser testada individualmente, garantindo sua confiabilidade e facilidade de manutenção. Com mais de 1000 linhas de código, seguindo o guia de estilo para código Python, PEP 8 [28], e mais 400 linhas de 25 testes unitários para a verificação do comportamento esperado, o resultado foi um serviço que atende seus requisitos e cumpre os objetivos específicos.

Em decorrência desta implementação, as atividades do processo de desenvolvimento de software passaram a ser armazenadas de maneira estruturadas diariamente. Informações como, por exemplo, a quantidade de *bugs* e sua criticidade, organizadas por projeto ou por período, podem ser obtidas com simples consultas em SQL.

A automação de todas as etapas citadas anteriormente, além de permitir a análise correlacional de maneira contínua, elimina a necessidade de trabalhos manuais para essa

finalidade, reduzindo o risco de erro humano inerente.

Inserida inicialmente em apenas um repositório, para a realização de testes mais profundos desta nova ferramenta, ela foi gradativamente sendo aplicada nos outros. Atualmente a ferramenta coleta métricas de 6 repositórios, além de ser inserida automaticamente em novos repositórios.

6.1 Perspectivas

Após a criação deste projeto, foram identificadas outras demandas que poderiam se beneficiar com a sua estrutura. Motivado pela demanda, o escopo de um novo projeto está se delineando, a fim de coletar outras métricas do JIRA para análises de mais alto nível dentro da empresa. Algumas análises já são feitas, através da coleta manual das métricas, e passados à diretoria da empresa.

A continuação deste projeto visa a construção de um *data warehouse* (conjunto de *data marts*) que utilize como fonte os banco de dados criados, mas que transforme as métricas obtidas em indicadores, de forma a explicitar as correlações existentes, traduzindo dados em informações mais relevantes.

Outro desdobramento do projeto será a ligação do bancos de dados criado com o software BI utilizado pela empresa, para efetivamente realizar a análise para a qual a ferramenta desenvolvida serviu de infraestrutura. A análise dos resultados em si será feita em conjunto com outro profissional dentro da empresa, mais acostumado a esse tipo de tarefa, com o intuito de agilizar a curva de aprendizado e os resultados dessa nova etapa.

A empresa, em um desdobramento de uma ação de recursos humanos (RH) irá realizar a análise com uma maior periodicidade de seus colaboradores. Para essa nova análise, será necessário relacionar as métricas de cada projeto, feito pela ferramenta desenvolvida, com as métricas individuais de cada pessoa, a ser feita através da replicação da mesma.

Todo o aprendizado obtido nesse curto período de tempo também permitiu ao aluno se fixar na equipe de desenvolvedores de *software web* da Hexagon Agriculture, podendo dar continuidade, entre outras coisas, a esse projeto.

Referências

1 HEXAGON. Hexagon Agriculture. *About Us*. 2019. Url:https://hexagon.com/about. Acesso em: 9 de junho 2019. Citado na página 15.

2 SCHMITZ, C. *Hexagon reorganiza portfólio para se aproximar do cliente*. 2019. Url:https://revistadeagronegocios.com.br/hexagon-reorganiza-portfolio-para-se-aproximar-do-cliente/. Acesso em: 12 de julho 2019. Citado na página 15.

3 WOMACK, J. P.; JONES, D. T.; ROOS, D. *The Machine That Changed the World*: Based on the massachusetts institute of technology 5-million dollar 5-year study on the future of the automobile. 18th. ed. [S.l.]: Rawson Associates, 1990. Citado na página 15.

4 SAINI, A. *The cost of fixing bugs throughout the SDLC*. 2017. Url:https://www.cbronline.com/enterprise-it/software/cost-fixing-bugs-sdlc/. Acesso em: 9 de julho 2019. Citado na página 16.

5 BECK, Kent, at el. *Manifesto for Agile Software Development*. 2001. Url:http://agilemanifesto.org/. Acesso em: 15 de maio 2019. Citado na página 19.

6 TOMAS, M. R. *Métodos Ágeis*: características, pontos fortes e fracos e possibilidades de aplicação. 2009. IET Working Papers Series. Acesso em: 12 de julho 2019. Citado na página 19.

7 SLIGER, M. *Agile project management with Scrum*. 2011. Paper presented at PMI® Global Congress 2011—North America, Dallas, TX. Newtown Square, PA: Project Management Institute. Citado na página 20.

8 CHACON, S. *Pro Git*. 1th. ed. Estados Unidos: Apress, 2009. 290 p. Citado 2 vezes nas páginas 21 e 22.

9 SPOLSKY, J. *The Joel Test*: 12 steps to better code. 2000. Url:https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better-code/. Acesso em: 10 de julho 2019. Citado na página 22.

10 ATLASSIAN. *Bitbucket*: What is bitbucket? 2018. Url:https://confluence.atlassian.com/confeval/development-tools-evaluator-resources/bitbucket/bitbucket-what-is-bitbucket. Acesso em: 18 de abril 2019. Citado 2 vezes nas páginas 23 e 31.

11 DIFFERENTIATING Between CI Pipelines and DevOps Assembly Lines. 2017. Url:https://devops.com/differentiating-ci-pipelines-devops-assembly-lines/. Acesso em: 20 de abril 2019. Citado na página 25.

12 ATLASSIAN. *Jira Software Server 8.2 documentation*. 2018. Url:https://confluence.atlassian.com/jirasoftwareserver/jira-software-overview-938845024.html. Acesso em: 20 de abril 2019. Citado 2 vezes nas páginas 25 e 31.

13 KIMBALL, R. *A Dimensional Modeling Manifesto*. 1997. Url:https://www.kimballgroup.com/1997/08/a-dimensional-modeling-manifesto/. Acesso em: 15 de julho 2019. Citado na página 26.

14 GROUP, K. *Kimball Dimensional Modeling Techniques*. 2013. Url:http://www.kimballgroup.com/wp-content/uploads/2013/08/2013.09-Kimball-Dimensional-Modeling-Techniques11.pdf. Acesso em: 9 de julho 2019. Citado na página 26.

15 SILVA, R. A. *PostgreSQL Database Modeler Documentation*: Overview. 2018. Url:https://pgmodeler.io/support/docs/i-overview. Acesso em: 28 de abril 2019. Citado na página 26.

16 SCHULTE, Y. N. R. *"Service Oriented"Architectures, Part 1*. 1996. Url:https://www.gartner.com/en/documents/302868. Acesso em: 4 de agosto 2019. Citado na página 27.

17 IBM. *Arquitetura Orientada a Serviços*. 2019. Url:https://www.ibm.com/support/knowledgecenter/pt-br/SSV2LR/com.ibm.wbpm.wid.main.doc/prodoverview/topics/csoa.html. Acesso em: 4 de agosto 2019. Citado na página 27.

18 AMAZON WEB SERVICES INC. *Computação em nuvem com a AWS*. 2019. Url:https://aws.amazon.com/pt/what-is-aws. Acesso em: 4 de agosto 2019. Citado na página 27.

19 AMAZON WEB SERVICES INC. *Amazon S3*: Object storage built to store and retrieve any amount of data from anywhere. 2019. Url:https://aws.amazon.com/s3. Acesso em: 10 maio 2019. Citado na página 27.

20 AMAZON WEB SERVICES INC. *Amazon Simple Queue Service*: Fully managed message queues for microservices, distributed systems, and serverless applications. 2019. Url:https://aws.amazon.com/sqs. Acesso em: 20 maio 2019. Citado na página 28.

21 AMAZON WEB SERVICES INC. *Amazon Relational Database Service (RDS)*: Set up, operate, and scale a relational database in the cloud with just a few clicks. 2019. Url:https://aws.amazon.com/rds. Acesso em: 10 maio 2019. Citado na página 28.

22 AMAZON WEB SERVICES INC. *AWS Lambda*: Run code without thinking about servers. pay only for the compute time you consume. 2019. Url:https://aws.amazon.com/lambda. Acesso em: 10 maio 2019. Citado na página 28.

23 AMAZON WEB SERVICES INC. *Amazon CloudWatch*: Complete visibility of your cloud resources and applications. 2019. Url:https://aws.amazon.com/cloudwatch/. Acesso em: 11 julho 2019. Citado na página 28.

24 HASHICORP. Terraform. *Introduction to Terraform*. 2019. Url:https://www.terraform.io/intro/index.html. Acesso em: 12 maio 2019. Citado na página 28.

25 BOXFUSE GMBH. Flyway. *Documentation*. 2019. Url:https://flywaydb.org/documentation/. Acesso em: 12 julho 2019. Citado na página 29.

26 AMAZON WEB SERVICES INC. *AWS Command Line Interface*. 2019. Url:https://aws.amazon.com/cli/. Acesso em: 11 julho 2019. Citado na página 45.

27 SQLALCHEMY. *SQLAlchemy 1.3 Documentation*: Overview. 2019. Url:https://docs.sqlalchemy.org/en/13/intro.html. Acesso em: 9 junho 2019. Citado na página 46.

28 ROSSUM, G. van; WARSAW, B.; COGHLAN, N. *Style Guide for Python Code*. 2001. Url:https://www.python.org/dev/peps/pep-0008/. Acesso em: 9 junho 2019. Citado na página 52.