NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

MARGARIDA MARQUES COELHO

Licenciatura em Engenharia Informática

CORQ - CODE REVIEW FOR QUALITY MEASUREMENT

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
outubro, 2023

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

CORQ - CODE REVIEW FOR QUALITY MEASUREMENT

MARGARIDA MARQUES COELHO

Licenciatura em Engenharia Informática

Orientador: Luís Vale de Andrade Botelho Pereira
Engenheiro Especialista, Opensoft

Coorientador: Artur Miguel de Andrade Vieira Dias
Professor Auxiliar, FCT-NOVA

Júri

Presidente: Margarida Paula Neves Mamede
Professora Auxiliar, FCT-NOVA

Arguente: Mário José Parreira Pereira
Professor Auxiliar, FCT-NOVA

Orientador: Luís Vale de Andrade Botelho Pereira
Engenheiro Especialista, Opensoft

Coorientador: Artur Miguel de Andrade Vieira Dias
Professor Auxiliar, FCT-NOVA

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
outubro, 2023

CORQ - Code Review for Quality Measurement

Este documento foi gerado com o processador (pdf/Xe/Lua)LaTeX e o modelo NOVAthesis (v6.10.10) [46].

Ao Francisco, a minha força motriz.

Agradecimentos

Esta dissertação representou uma etapa fundamental do meu percurso académico, um desafio que me permitiu crescer. Apesar de ser um trabalho solitário, reúne contributos de várias pessoas que acompanharam o seu desenvolvimento.

Primeiramente, gostaria de expressar a minha gratidão ao meu orientador, o Engenheiro Luís Pereira, e ao arquiteto, o Engenheiro Ricardo Mascarenhas, por todo o apoio, recomendações e disponibilidade durante a conceção da dissertação.

Agradeço igualmente ao coorientador Artur Miguel Dias pela orientação que ofereceu para a construção deste documento e prontidão em dar *feedback*.

Agradeço também à Opensoft pela oportunidade, e pelo ambiente caloroso e de aprendizagem que proporcionou.

Por fim, e com igual importância, gostaria de agradecer ao meu amor, à minha família e a todos os amigos que me apoiaram, inspiraram e deram ânimo para dar o meu melhor nesta etapa.

Resumo

Existem diversas métricas que avaliam projetos de *software*, como linhas de código, complexidade ciclomática, dívida técnica, conformidade com padrões, cobertura de testes, etc. Porém, os projetos de *software* continuam a ser difíceis de prever - têm sempre risco associado.

Ademais, é um desafio obter informação relevante para conduzir um projeto de *software*, entre a vasta informação disponível e dispersa nas ferramentas de apoio ao desenvolvimento utilizadas por uma empresa.

Por conseguinte, existem ferramentas de *software analytics*, que consomem e efetuam o processamento dos dados de variadas fontes de informação, para gerar informação que condensa a atividade de desenvolvimento num formato fácil de consumir, através de um *dashboard web*. Com isto, os gestores podem obter facilmente *insights* precisos que direcionam a tomada de decisão.

Posto isto, em 2021, a Opensoft avançou com uma solução tecnológica automática, a ferramenta CORQ, que a partir dos dados dos repositórios Bitbucket dos projetos da empresa, apresenta num *dashboard web* a evolução ao longo do tempo de métricas de produtividade aquando do desenvolvimento do código, com grau de granularidade por colaborador e/ou por projeto, e da empresa.

Esta dissertação veio aprimorar a ferramenta, e adicionar métricas de qualidade do código produzido, com recurso à ferramenta SonarQube.

Como resultado, alcançámos uma versão significativamente mais eficiente desta ferramenta. Esta não só beneficia de uma fonte adicional de informação, mas também oferece um *dashboard web* mais abrangente, que proporciona aos utilizadores uma experiência mais enriquecedora, permitindo a exploração das informações apresentadas com diferentes graus de granularidade.

Palavras-chave: *Software Analytics*, Gestão ágil, Decisões orientadas por dados, SonarQube

Abstract

There are various metrics to evaluate software projects, such as lines of code, cyclomatic complexity, technical debt, compliance with standards, test coverage, etc. Nevertheless, software projects remain difficult to predict - there is always a risk associated with them.

In addition, it is a challenge to obtain relevant information to manage a software project, among the vast information available and dispersed through the development support tools used by a company.

Therefore, there are software analytics tools that consume and process data from various sources to generate information that summarises development activity into an easy-to-consume format, via a web dashboard.

That being sad, in 2021, Opensoft advanced an automatic technology solution, the CORQ tool, which uses data from the Bitbucket repositories of the company's projects to display on a web dashboard the evolution over time of productivity metrics during code development, with a degree of granularity per collaborator and/or per project, and for the company.

This dissertation has improved the tool and also added metrics of the quality of code produced, using the SonarQube tool.

As a result, we have achieved a significantly more efficient version of this tool. Not only does it benefit from an additional source of information, but it also offers a web dashboard, which provides users with a more enriching experience, allowing them to explore the information presented with varying degrees of granularity.

Keywords: Software Analytics, Agile management, Data-driven decisions, SonarQube

Índice

Índice de Figuras

Glossário

Issue É uma violação de uma regra de qualidade de código, que aponta para um bloco de código que precisa de ser revisto ou corrigido. *(pp. 42, 49–52)*

Job É uma aplicação que faz o processamento de uma quantidade finita de dados, sem interrupção da sua execução. *(p. 45)*

Sprint É um período curto e fixo em que uma equipa trabalha para concluir uma quantidade definida de trabalho, e portanto, permite que esta equipa entregue incrementalmente o produto, conferindo-lhes mais flexibilidade de adaptação às mudanças *Sprint*. *(pp. 1, 15, 72)*

User Stories É uma explicação informal e geral de uma caraterística de software escrita pela perspetiva de um utilizador final [73]. *(p. 25)*

dashboard Interface gráfica para utilizadores, que organiza informação essencial e, para fácil compreensão, apresenta-a graficamente, em tempo real. *(pp. 2, 63, 71)*

scheduler Aplicação que efetua a execução dos *Jobs* num momento específico ou em resposta a um evento desencadeado específico. *(p. 55)*

colaborador Conforme utilizado neste documento, este termo refere-se a um profissional associado ao desenvolvimento de *software*, que desempenha o papel de programador, sendo responsável por produzir, testar e manter código-fonte de *software*. *(p. 1)*

Inteligência Analítica É um campo abrangente, que utiliza matemática, estatística e aprendizagem automática para encontrar padrões e conhecimento significativo em dados. *(p. 5)*

Scrum É uma estrutura ágil para a gestão de projetos, que enfatiza a colaboração em equipa para produzir um produto numa série de iterações, cada uma chamada *Sprint* [76]. *(p. 1)*

Ágil Do inglês *Agile*, designa uma disciplina que estuda um conjunto de comportamentos, processos, práticas e ferramentas utilizados para o desenvolvimento de produtos. *(p. 1)*

Siglas

API Application Programming Interface *(pp. 18, 19, 24, 31, 35, 37, 54)*
AT Autoridade Tributária e Aduaneira *(pp. 23, 44)*

CORQ Code Review for Quality Measurement *(pp. 2, 6, 17, 23, 24, 31, 33, 41, 45, 47, 52, 54, 60, 63, 70)*

HTTP Hypertext Transfer Protocol *(pp. 38, 61)*

IDE Integrated Development Environment *(p. 31)*

JAR Java Archive *(pp. 35, 36)*
JSON JavaScript Object Notation *(p. 38)*

RCI Rules Compliance Index *(pp. 52, 53)*

SGBD Sistema de Gestão de Base de Dados *(p. 18)*
SSO Single Sign-On *(pp. 17, 72)*

URL Uniform Resource Locator *(pp. 31, 44, 54)*

1

Introdução

Este capítulo proporciona uma visão abrangente para contextualização desta dissertação, esclarece a motivação que impulsionou a sua realização, estabelece os objetivos a serem alcançados com a sua elaboração e, por fim, delineia a estrutura deste documento.

1.1 Contexto

Uma equipa de gestão, numa empresa de desenvolvimento de *software*, tem um papel fundamental em direcionar o rumo da empresa. Esta equipa colabora para definir metas a serem alcançadas e planos de ação para atingir essas metas. Tem também como função tomar decisões estratégicas que afetam o rumo da empresa, como o investimento em produtos novos, ou a expansão para outros mercados. São responsáveis pela alocação dos recursos para os vários projetos da empresa, equilibrando as prioridades e assegurando que os recursos são usados eficientemente.

É a equipa de gestão que monitoriza o desempenho geral da empresa, avaliando o progresso em relação às metas estabelecidas - identificam áreas que necessitam de melhoria e tomam medidas corretivas. Ademais, uma equipa de gestão é responsável por liderar as equipas de desenvolvimento que compõem uma empresa de *software*, o que inclui a sua motivação, a avaliação de desempenho e o desenvolvimento da formação dos colaboradores. Portanto, uma equipa de gestão, tem de assegurar que a empresa alcança as suas metas eficazmente, continuando competitiva e adaptada ao ambiente em constante mudança da indústria de tecnologia da informação.

Para gerir os projetos e as equipas de desenvolvimento, a equipa de gestão da Opensoft adota uma estrutura Ágil, o Scrum. Aplica esta metodologia através de: eventos de 15 minutos, designados de "Daily"(designação breve para "Daily Scrum"), para sincronização de uma equipa de desenvolvimento, em que cada colaborador identifica o que fez de mais relevante no dia anterior, define um plano de trabalho para o dia e identifica quaisquer impedimentos; planeamento de um *Sprint*, definindo quais as tarefas a realizar e atribuindo a cada uma estimativa de tempo; revisão do resultado do *Sprint*, apresentando o trabalho desenvolvido num *Sprint*, apontando o que se concretizou; retrospetiva sobre o *Sprint*,

analisando a qualidade e eficácia do *Sprint*.

Também, a equipa de gestão utiliza ferramentas de gestão e apoio ao desenvolvimento dos projetos: o Jira da Atlassian, que permite o registo de tarefas a realizar com estimativa de tempo, a atribuição de responsabilidades, o acompanhamento do progresso das tarefas e o ajuste de prioridades de acordo com as necessidades; o SonarQube da SonarSource, que produz métricas e relatórios sobre a qualidade do código dos projetos, e portanto, permite o acompanhamento da evolução da qualidade de código ao longo do tempo, e o *feedback* imediato, incentivando a melhoria contínua; o Confluence, também da Atlassian, que permite que a equipa de gestão mantenha a documentação dos projetos organizada, atualizada e acessível, o que é importante para que os colaboradores tenham informação para trabalhar eficazmente nos projetos; o Bitbucket e os repositórios SVN, como ferramentas de controlo de versões, têm informação sobre os *commits* dos colaboradores.

Porém, como constatamos a partir deste contexto, a informação que a equipa de gestão extrai para o acompanhamento das equipas de desenvolvimento, está dispersa nas várias ferramentas referidas. Isto dificulta que a equipa de gestão obtenha uma visão holística sobre os projetos e colaboradores, o que, por sua vez, afeta a agilidade da sua gestão. Para mais, nenhuma das ferramentas referidas produz informação sobre a qualidade do código produzido por um colaborador, o que é importante para que a equipa de gestão compreenda o impacto sobre a qualidade geral do *software* produzido.

Ora, atualmente, existem soluções tecnológicas que abordam esta lacuna, fornecendo uma visão geral das informações relevantes para uma equipa de gestão. Estas ferramentas agregam informação de várias fontes, apresentando-a visualmente, o que ajuda uma equipa de gestão a tomar decisões ágeis.

Posto isto, em 2021, a Opensoft avançou com uma solução tecnológica, a ferramenta Code Review for Quality Measurement (CORQ). Esta ferramenta agrega informação dos repositórios Bitbucket dos projetos da empresa e apresenta num *dashboard web* métricas de produtividade sobre o desenvolvimento do código ao longo do tempo, com grau de granularidade por colaborador e por projeto.

1.2 Motivação

Até ao começo desta dissertação, a ferramenta CORQ apenas apresentava métricas de produtividade. Com estas métricas, a equipa de gestão obtém uma visão sobre o ritmo do trabalho em desenvolvimento, e portanto, identifica, com facilidade, algum *bottleneck*, i.e. algum obstáculo que impede que o trabalho progrida eficientemente.

Todavia, a ferramenta não abordava ainda um aspeto igualmente vital do desenvolvimento de *software* - a qualidade do código produzido. Para que a equipa de gestão obtenha uma compreensão mais abrangente do desenvolvimento e saúde geral dos projetos, foi fundamental o desenvolvimento desta ferramenta, para introduzir métricas de qualidade do código produzido, com grau de granularidade por projeto, mas também por colaborador.

Com esta abordagem, a ferramenta CORQ habilita a equipa de gestão para uma gestão Ágil, através de um acompanhamento mais eficaz das equipas de desenvolvimento, que proporciona o equilíbrio entre as entregas rápidas e a excelência do *software* produzido.

Ademais, a ferramenta CORQ será mais tarde disponibilizada para proveito não só da equipa de gestão, como das equipas de desenvolvimento.

1.3 Objetivos

Essencialmente, o objetivo desta dissertação foi a extensão da ferramenta CORQ, com a adição de métricas de qualidade de código. Para tal, o pretendido era obter informação sobre a qualidade do código produzido com recurso ao SonarQube, e definir pelo menos uma métrica de qualidade de código.

Também, foi estabelecido que o *dashboard* da ferramenta ofereceria a funcionalidade de consulta e pesquisa das informações das métricas de produtividade e/ou das métricas de qualidade de código, com vários graus de granularidade: referente à empresa, a um projeto, a um colaborador, ou sobre o trabalho de um colaborador num projeto. Esta pesquisa pode também ter aplicado um filtro por intervalo de tempo.

Por fim, a ferramenta tem que ser desenvolvida mantendo a sua extensibilidade, permitindo a adaptação da mesma em futuras evoluções, com a adição de métricas e de fontes de informação.

1.4 Estrutura do Documento

A organização dos restantes capítulos deste documento é a seguinte:

2. Trabalho relacionado: O capítulo 2 apresenta uma discussão sobre ferramentas de *software analytics*, das implicações morais que podem resultar com a sua utilização, mas também vantagens que oferecem. Por fim, são descritas algumas ferramentas existentes relevantes para este trabalho.

3. Tecnologias e Ferramentas utilizadas: O capítulo 3 apresenta brevemente as tecnologias e ferramentas utilizadas com o desenvolvimento desta dissertação.

4. Trabalho Prévio: O capítulo 4 apresenta a Opensoft, e a ferramenta CORQ, proporcionando uma visão geral da versão inicial desta ferramenta, que serviu como base para a evolução descrita nesta dissertação.

5. Solução e Implementação: O capítulo 5 explora em detalhe a evolução desta ferramenta, os aprimoramentos efetuados e a implementação das funcionalidades acrescidas, relatando, simultaneamente, os obstáculos que surgiram durante esta fase.

6. Avaliação: O capítulo 6 apresenta a avaliação do trabalho realizado nesta dissertação.

7. Conclusão: O capítulo 7 finda esta dissertação com um tom reflexivo, apresentando as contribuições desta dissertação para a ferramenta CORQ e sugestões para trabalho futuro.

2

Trabalho relacionado

Existem várias ferramentas de gestão e apoio ao desenvolvimento dos projetos, que produzem informação variada sobre os projetos de *software*. Ademais, existem artefactos disponíveis em todas as etapas do ciclo de vida do desenvolvimento de *software*. Porém, os projetos de *software* são difíceis de prever - têm sempre riscos. Como dito por Buse e Zimmermann [9], os projetos de *software* são facilmente mensuráveis, mas frequentemente imprevisíveis.

Simultaneamente, a natureza dinâmica da indústria de *software* está associada a necessidades de tomada de decisões em todas as camadas de negócios de *software*, desde a visão de uma empresa até ao planeamento da gestão de um projeto e implementação pelos colaboradores [1].

Hindle et al. [31] afirma que, qualquer linguagem de programação é complexa, flexível e poderosa, mas os programas que os indivíduos escrevem são maioritariamente simples e bastante repetitivos, e por isso, têm propriedades estatísticas previsíveis que podem ser capturadas por ferramentas analíticas. Isto significa que, ferramentas analíticas podem capturar informação que ajude gestores a organizarem eficientemente recursos e a identificarem obstáculos, por análise de padrões.

Posto isto, profissionais e investigadores recorrem a ferramentas analíticas para orientarem a tomada de decisões ao longo do desenvolvimento de *software*.

2.1 *Software Analytics*

Por conseguinte, surgiu o termo *software analytics*, um campo de Inteligência Analítica aplicado ao domínio do *software*. Designa uma abordagem aos dados que utiliza tecnologias como a aprendizagem automática, a extração de dados, a estatística, o reconhecimento de padrões, a visualização das informações e o processamento de grandes volumes de dados para obter *insights* a partir dos dados gerados por *software*.

O termo *insight* é utilizado neste contexto para se referir a uma perceção adquirida através de uma análise aprofundada dos dados. Por exemplo, os *insights* respondem não só a questões como "O que aconteceu?", mas também a questões mais essenciais para

entender "Como aconteceu e porquê?". Portanto, um *insight* oferece informação valiosa que pode ter um impacto benéfico para a tomada de decisões sobre o desenvolvimento de *software*. Ou seja, um *insight* não representa apenas informação, como também ações concretas [67]. Por conseguinte, os *insights* são capazes de direcionar uma equipa de gestão, para que a tomada de decisões seja mais fácil, rápida, precisa e confiante [1].

2.2 Preocupação Moral

Um dos objetivos da ferramenta CORQ é que disponibilize métricas sobre produtividade aquando do desenvolvimento de código e métricas sobre qualidade do código produzido, com respeito a um colaborador, ao longo do tempo. Todavia, o acompanhamento ao longo do tempo do desempenho de um colaborador traz preocupações morais.

Qualquer um dos conceitos, produtividade ou qualidade, é demasiado amplo para ser reduzido a uma única métrica. Isto porque, quando criamos uma métrica, examinamos com esta apenas uma fatia ínfima do tempo e trabalho desenvolvido por um colaborador [37]. Portanto, é impossível reduzirmos a complexidade do trabalho com que um colaborador contribui a uma única medida.

Tal como formulado por Ko [42], uma consequência imprevista que resulta de medirmos produtividade através de uma única métrica é que arriscamos otimizar um resultado intermédio em vez de um objetivo final. Por exemplo, admitindo que medimos produtividade através de uma métrica baseada em "tempo para *release*". Isto significa que, *commits* produzidos com maior rapidez, mesmo que contendo mais *bugs*, aceleram a *release* de um artefacto, e portanto, incrementam o valor desta métrica. Porém, se não medirmos simultaneamente os resultados desta *release* - resultados positivos, como o grau de satisfação do cliente, e resultados negativos, como falhas do *software* -, estamos a otimizar a métrica à custa destes resultados.

O desenvolvimento de código não é a única tarefa de um colaborador. Este tem uma variedade doutras tarefas, como: produzir documentação, configurar ferramentas, procurar soluções, desenhar funcionalidades de um produto, etc. Existe, também, uma variedade de tarefas de carácter social que podem ter um impacto significativo sobre o desempenho de uma equipa ou mesmo da empresa, como, por exemplo, a orientação de um colaborador recentemente adicionado a uma equipa. Estas tarefas exigem tempo, e portanto, por conseguinte, afetam a produtividade aquando do desenvolvimento de código, que diz respeito apenas ao tempo em que o colaborador produz código. Todo o restante trabalho não é visível, nem contabilizado, ou não existe nenhuma informação que dê contexto a casos em que, como exemplificado, outras tarefas afetam a produtividade de um colaborador.

Pode haver casos em que a diminuição da produtividade de um colaborador, aumenta a produtividade da equipa - por exemplo, uma interrupção pode ser um incómodo para um colaborador que está a desenvolver código, pois desvia a sua atenção do que está a fazer, mas se este tiver o conhecimento que falta a outro para avançar com uma solução,

mesmo que isto reduza brevemente a sua produtividade, aumenta a produtividade da equipa a longo prazo.

Treude, Figueira Filho e Kulesza [70] realizaram um estudo sobre que informação os colaboradores esperam que seja incluída num resumo da sua atividade de desenvolvimento. Foi pedido aos participantes para referirem ou construirem uma métrica para medir o *input*/*output* de um colaborador. Ou seja, não foi dado nenhum conjunto de opções de resposta, e portanto, as respostas a esta questão não foram enviesadas. Como resultado, houve participantes que referiram medidas objetivas - por exemplo, linhas de código adicionadas -, e métricas subjetivas - por exemplo, documentação produzida, eficiência do código produzido, assim como o seu estilo, legibilidade, se o colaborador participa nas reunião de equipa, se fornece ideias. Porém, vários expressaram como opinião que a atividade de desenvolvimento é impossível de medir.

Por exemplo, um participante referiu que métricas quantitativas como linhas de código não refletem a dificuldade de uma tarefa de código, sendo inadequadas - modificações a uma arquitetura ou uma refatoração concetual podem ter um impacto significativo, mas não são visíveis por esta métrica.

Outros, tendo consciência que a atividade de desenvolvimento é constituída por variadas tarefas, responderam simplesmente que é impossível medir com uma simples métrica, pois há vários aspetos do trabalho de desenvolvimento que não podem ser quantificados, mas que são equivalentemente importantes - por vezes, até mais importantes. Também, qualquer métrica objetiva, como linhas de código adicionadas, horas de trabalho registadas, tarefas completadas, *bugs* resolvidos, não pode ser avaliada sem atender ao contexto do trabalho. Ademais, vários apontaram que a fase de desenvolvimento em que o projeto se encontra é relevante, pois as métricas deveriam ser diferentes conforme a fase do ciclo de desenvolvimento de um projeto.

Ko [42] refere também que uma empresa que adote uma solução para monitorizar a produtividade dos colaboradores, está inadvertidamente a reduzir a produtividade dos colaboradores - como efeito de serem tão minuciosamente observados, os colaboradores vão monitorizar quaisquer ações, despendendo mais tempo nas suas ações.

Uma preocupação comum entre os colaboradores é que uma métrica seja mal interpretada, em particular por um gestor que não têm conhecimento das limitações inerentes à métrica. Se uma métrica for utilizada para avaliação e classificação do desempenho de um colaborador, esta terá impacto sobre o reconhecimento do seu trabalho e esforço.

Ademais, a preocupação que existe com a coleta de métricas sobre o desempenho dos colaboradores resulta também do receio que um colaborador seja avaliado em função do seu *output*. Ora, medir o *output* ao longo do tempo é uma boa solução para medir o impacto de máquinas, mas não de humanos. Não podemos confundir *output*, que refere-se aos resultados de uma ação em específico, a *outcome*, que representa o impacto geral ou o valor criado pelo *output*. Todavia, enquanto que o *output* é fácil de medir, o *outcome* nem sempre o é. Isto ocasiona que, por vezes, os gestores avaliem *output* em vez de *outcome*, confundido-os inadequadamente.

Portanto, os colaboradores receiam que se o resultado de uma métrica for mal interpretado ou oferecer apenas informação parcial sobre o seu desempenho, que esta pode, numa situação extrema, ser a razão pela qual são despedidos [37].

Em contrapartida, este risco incentiva a que uma métrica seja *gamed*, i.e. falseada, manipulada, como previsto pela "lei de Goodhart"[12, 30]. Ou seja, se a empresa implementar um sistema em que são atribuídos bónus quando os colaboradores alcançam os valores esperados para uma métrica, isto pode trazer a longo prazo consequências. De acordo com [70], o mais provável é que os colaboradores, sendo os indivíduos mais capazes para tal, manipulem o sistema para otimizarem o seu próprio bónus. Isto aplica-se particularmente a métricas simples como linhas de código, em que um colaborador pode introduzir código desnecessário para que se verifique um incremento da métrica.

Outro fator apontado foi a diferença entre um colaborador com pouca qualidade que resolve muitas tarefas em pouco tempo, comparativamente a um colaborador que resolve menos tarefas, que não são reabertas ou mais tarde rejeitadas. Por esta razão, é importante que sejam coletadas métricas que acompanham a qualidade, assim como a quantidade.

Martínez-Fernández et al. [48] apontam que a maioria das ferramentas de *software analytics* que existem falha em oferecer informação sobre objetivos referentes à qualidade do trabalho realizado com a atividade de desenvolvimento. O seu estudo explora os benefícios da integração de modelos de qualidade em ferramentas de *software analytics* para avaliar eficazmente e melhorar a qualidade do *software*.

Ko [42] refere que para determinarmos quais os fatores que afetam a produtividade, é fundamental termos uma visão qualitativa sobre o comportamento de uma equipa e respetivos colaboradores, em conjunto com informação quantitativa. Porém, uma equipa está sempre a mudar, ora porque é adicionado um colaborador ou um colaborador sai, ora porque ao longo do tempo adquirem mais ou melhores competências, mudando a forma como escrevem código, comunicam, coordenam e colaboram entre si. Portanto, é difícil obter informação sobre o comportamento de uma equipa. Ora, a autora defende que os gestores estão melhor posicionados para obter informação qualitativa interagindo com a sua equipa. Posto isto, a autora argumenta que em vez de medir produtividade, uma empresa deve investir em formar bons gestores, para que observem a produtividade como parte do seu trabalho diário com os colaboradores. Com isto, a autora não exclui que não se utilizem estas ferramentas para ajudar a refletir e a tomar decisões, pois podem até ajudar os gestores com menor experiência a desenvolverem competências de gestão, mas sublinha que a gestão não pode advir somente das métricas coletadas.

Enfim, são estas razões assim tão más que nos impeçam de tentar medir o desempenho das equipas e colaboradores?

2.3 Outra Perspetiva

Para compreendermos quais as razões que podem suplantar as preocupações morais que resultam de medirmos o desempenho das equipas de desenvolvimento e respetivos

colaboradores, temos que compreender que informações uma equipa de gestão requer para tomar decisões, e porquê a resolução de que o campo de *software analytics* é a resposta.

Enquanto que Ko [42] rejeita que podemos medir produtividade, Jaspan e Sadowski [37] defendem que, embora não exista uma métrica única para medir a produtividade de um colaborador, podemos definir um conjunto de métricas, direcionadas a uma questão específica, que ajudem a tomada de decisões. Este raciocínio alternativo segue o paradigma "Goal-Question-Metric"[6]. Isto significa que, primeiramente, são clarificadas as questões e motivação, depois são criadas métricas direcionadas a essas questões específicas. Mais tarde, estas métricas são validadas para garantirem que medem o objetivo original.

Cunha et al. [16] procuram compreender a tomada de decisões em gestão de projetos de *software*, e identificaram fatores que afetam a tomada de decisão dos gestores. Os autores classificaram estes fatores como individuais - por exemplo, a base de conhecimento que um indivíduo possui por experiência -, ou contextuais - por exemplo, o envolvimento ativo dos clientes, que é caraterístico de uma metodologia Ágil. Este estudo foi realizado sem considerarem uma abordagem orientada por *software analytics*.

Ora, os gestores nem sempre compreendem todos os fatores que influenciam os seus projetos. Porém, a *software analytics* permite-nos "confiar, mas verificar"as intuições humanas [50]. Se alguém afirma que "isto ou aquilo"é importante para um projeto de *software* bem sucedido, a análise permite-nos tratar essa afirmação como algo a ser verificado (em vez de uma lei sagrada que não pode ser questionada). Ademais, uma vez que a afirmação é verificada, a análise pode atuar para monitorização, para verificarmos continuamente se "isto ou aquilo"não é ultrapassado por desenvolvimentos subsequentes. A boa notícia é que, utilizando *software analytics*, podemos corrigir estes equívocos.

Dito isto, Menzies e Zimmermann [50] explicitam porque é que necessitamos das ferramentas de *software analytics*, através de um listagem de factos que refutam convicções comuns de gestores ou colaboradores da indústria de *software*.

Rique et al. [56, 57] reportam os resultados obtidos com entrevistas realizadas a indivíduos com diferentes cargos de gestão numa empresa, sobre a perspetiva destes acerca da adoção de ferramentas de *software analytics* para suporte de decisões. Os participantes possuíam uma vasta experiência com a indústria de *software*.

Por conseguinte, os participantes identificaram 19 casos de uso, que se traduzem nas informações que desejam obter com estas ferramentas. Por exemplo, a totalidade dos participantes referiu a gestão da produtividade de uma equipa de desenvolvimento, a monitorização do calendário de um projeto e a gestão dos riscos de um projeto. Os restantes casos de uso são apresentados pela figura 2.1 (adaptada de [56]), com respetiva descrição.

Esta lista de casos de uso foi organizada em quatro dimensões:

Qualidade: refere-se à qualidade dos produtos e processos;

Indivíduos: compreende os vários aspetos das equipas de desenvolvimento, como o bem-estar dos elementos e necessidades de formação;

Gestão de projeto: refere-se aos aspetos que afetam o progresso dos projetos de *software*, como o calendário dos projetos e riscos que surjam;

Gestão de conhecimento: engloba o registo e compartilhamento de informação e experiência adquirida por uma empresa, para melhorar a resolução de problemas, promover uma cultura de aprendizagem contínua e impulsionar a inovação.

Id	Use case	Description
Quality		
UC1	Managing product quality	Assessing how quality assurance is being performed regarding a software product (e.g., do we have the OK from the quality team for the upcoming build? Does source code meet quality standards?).
UC2	Identifying the source of bugs	Identifying what is causing bugs in a software project (e.g., who are the members responsible for most bugs?).
UC3	Managing bugs	Managing bugs through their entire life cycle (i.e., from identifying the bug to closing the bug).
UC4	Supporting the definition of the development/quality process	Defining the adequate practices to meet requirements (e.g., will load testing be performed? what percentage of unit test coverage is required? will Scrum or Kanban be used?).
UC5	Monitoring process compliance with standards	Guaranteeing that the defined practices are being followed to achieve the desired quality level (e.g., are Daily Meetings taking place daily? are the load tests being performed before deployment in the approval environment? are the deployment environment being monitored?).
UC6	Monitoring technical debt	Verifying if technical debt is increasing or decreasing.
UC7	Supporting project compliance with customer expectations	Getting feedback on customer satisfaction (i.e., if what was delivered met customer expectations).
People		
UC8	Identifying training needs	Identifying aspects in the project that indicate a training need (e.g., low-quality code, code with many bugs).
UC9	Assessing how much a project depends on a certain person	Assessing the extent to which a project depends on a certain person (e.g., how much a project depends on a developer?).
UC10	Monitoring the well-being of teams	Getting feedback on the well-being of a team (e.g., is the team satisfied? is the team motivated?).
Project management		
UC11	Managing team productivity	Managing different aspects of team productivity (e.g., the most productive people, story points delivered, how long it takes for a developer to complete a task).
UC12	Monitoring project schedule	Having an overview of several conditions regarding project schedule (e.g., will the team be able to deliver what was agreed for a Sprint? is the schedule delayed?).
UC13	Supporting schedule delay management	In the face of schedule delays, enabling stakeholders to take steps that lessen the impact on the project (e.g., how much extra effort does the team need to make to mitigate the schedule delay?).
UC14	Identifying new features for projects	Identifying new functionality for projects (e.g., to renew the contract, the customer wants new features for the project so that he/she keeps that investment in the organization).
UC15	Managing project scope	Managing scope issues (e.g., changes in the scope agreed for a Sprint or assessing if a new scope affects team velocity).
UC16	Managing project risks	Managing the factors that compromise the progress of the project, reducing their impacts (e.g., what is the planning of the team to mitigate the impacts of changes in the project?).
UC17	Increasing delivery value	Including a greater number of user stories into a Sprint planning (i.e., a product increment that delivers direct value to the customers).
Knowledge management		
UC18	Gathering decisions from different areas of the organization	Getting feedback on past decisions with regard to problems affecting different sectors of the organization (e.g., when a problem needs to go through different areas of the organization, the solutions proposed by each sector will help the manager make a decision).
UC19	Solving problems with the support of lessons learned	Using the organization's experience in solving problems (e.g., solutions that worked and did not work for a problem that happened in the past so that managers will have support to solve the problem in the present).

Figura 2.1: Lista dos casos de uso identificados.

Davenport, Harris e Morison [18] identificam seis categorias de questões que a *software analytics* pode ajudar a responder. Estas questões estão organizadas por duas dimensões, como apresentado pela figura 2.2:

Intervalo de tempo

Estamos a observar o passado, presente ou futuro?

Inovação

Distingue entre questões de informação, que se referem aos valores das métricas - por exemplo, "Quantos *bugs* o código-fonte de um projeto tem?-, e questões de *insight*, que se referem a informação adquirida através de uma análise aprofundada - por exemplo, "O projeto vai sofrer atrasos?". Ora, são as questões de *insight* que

proporcionam aos gestores um entendimento significativo sobre a atividade de desenvolvimento e uma base sobre a qual tomar decisões.

	Past	Present	Future
Information	What happened? (Reporting)	What is happening now? (Alerts)	What will happen? (Extrapolation)
Insight	How and why did it happen? (Modeling, experimental design)	What's the next best action? (Recommendation)	What's the best/worst that can happen? (Prediction, optimization, simulation)

Figura 2.2: Questões chave abordadas pela análise. (Retirado de [18])

O objetivo global desta análise é ajudar os gestores a irem para além da informação e em direção ao *insight*. Porém, esta transição não é fácil. O *insight* requer gestores que apresentem conhecimento do domínio e capacidade de identificar padrões envolvendo múltiplas métricas. Por exemplo, para compreender por que razão o número de relatórios de *bugs* está a aumentar e se esta é uma preocupação, é importante compreender que outras tarefas estão a ocorrer: Há uma *release* para breve? A equipa está a trabalhar em correções de *bugs* ou em adições de funcionalidades? A *software analytics* pode ajudar os gestores a encontrarem rapidamente agulhas importantes em palheiros de informação [10].

Dam, Tran e Ghose [17] apontam que ferramentas de *software analysis* são comummente consideradas como uma "caixa negra", e portanto, defendem que os colaboradores desconfiam e apresentam relutância em aceitar ferramentas destas. Por conseguinte, os autores afirmam que é importante desenvolver a explicabilidade das ferramentas de *software analytics*, para que, consequentemente, os colaboradores desenvolvam a sua confiança nestas ferramentas.

O que retiramos com esta discussão é que ferramentas como as que realizam análise do *software* devem ser aplicadas com cuidado, tendo em atenção o conjunto de métricas escolhidas e o que medem. Ou seja, o mais importante é considerarmos que não são mais

que uma ferramenta que faz parte de um conjunto de ferramentas maior para coordenar e gerir o trabalho de desenvolvimento de *software*. E esta é a visão de gestão da Opensoft.

2.4 *Dashboards*

Nos dias que decorrem, a definição de *dashbaord* mais comummente usada, em desenvolvimento de *software*, é a definição de Few [24], que dita que um *dashboard* é uma ferramenta que apresenta visualmente informação importante para atingir um ou mais objetivos, e que ocupa um único ecrã de um computador, para que as informações sejam monitorizadas num relance.

Por conseguinte, através de um *dashboard*, um utilizador pode identificar tendências, padrões e anomalias nas informações que são apresentadas, raciocinar sobre o que vê e tomar decisões com base em informação [58]. O objetivo de um *dashboard* é apresentar, num formato facilmente consumível, informação que pode traduzir-se em ações. Ou seja, um *dashboard* é uma ferramenta que permite acompanhar projetos e colaboradores [71].

Posto isto, os *dashboards* são ferramentas fundamentais para compreender e extrair conhecimento de grandes volumes de dados, e por isso, frequentemente utilizadas por ferramentas de *software analytics*.

Todavia, o poder que os *dashboards* oferecem para análise de informação introduz uma série de riscos a ter em atenção [65, 71]:

- **Os *dashboards* favorecem informação quantitativa a informação qualitativa:** a maioria dos artefactos com que os colaboradores trabalham são textuais, como por exemplo, requisitos dos projetos, documentação, etc. Como tal, têm sido propostos *dashboards* qualitativos [7], que contêm resumos de texto. Estes resumos de texto têm potenciais vantagens, como a possibilidade de explicarem em vez de simplesmente medirem, a oportunidade de incluírem o contexto sempre que necessário e a redução do esforço cognitivo necessário para consumir, interpretar e dar sentido a grandes volumes de dados.

  Porém, apresentar o conteúdo destes artefactos exige soluções para processamento de texto, o que está atualmente em investigação. Em consequência, a maioria dos *dashboards* apresenta informação quantitativa, o que, comummente, é insuficiente para capturar a totalidade dos aspetos da atividade de desenvolvimento. Por exemplo, apesar de uma equipa ter mais *bugs* reportados num projeto que noutro, não significa que a qualidade do projeto seja pior, pode simplesmente ser por este projeto ser cognitivamente mais desafiante.

- **Os *dashboards* falham em apresentar contexto importante:** um projeto de *software* é único. Quando exploramos um *dashboard*, observamos apenas uma fração das informações referentes a um projeto. Ou seja, o contexto que envolve este projeto é em parte excluído. Todavia, quando um *dashboard* apresenta informação sobre os

diversos projetos de uma empresa, pode erradamente incentivar quem o usa a fazer comparações entre projetos.

- **Os *dashboards* não apresentam explicações:** os *dashboards* não explicam as informações que exibem e, como também não apresentam contexto, muitas vezes é difícil compreender e saber como agir com base nas informações apresentadas.

- **"Lei de Goodhart":** "Quando uma medida se torna um objetivo, deixa de ser uma boa medida.". Esta lei descreve outro risco já referido com a secção 2.3. Isto é, se um *dashboard* concetualiza produtividade como o número de *commits* realizados, então os colaboradores em resposta terão mais cuidado com os *commits* que realizam. Ou seja, isto pode incentivar a que os colaboradores realizem *commits* desnecessários para atingir a meta quantitativa.

- **Os *dashboards* dependem do *input*:** isto é, dependem da qualidade das informações que são alimentadas como *input*. Se estas estiverem erradas, então a utilidade deste é comprometida.

- **Os *dashboards* só apresentam informações coletadas por ferramentas:** existem várias ferramentas que capturam vários aspetos da atividade dos colaboradores, mas nem todas as atividades podem ser capturadas por uma ferramenta - como, por exemplo, uma reunião entre colaboradores que pode ser o resultado de uma correção de um determinado *bug*. Ora, informação que não consta de um repositório, não pode ser visualmente apresentada num *dashboard*. Portanto, os utilizadores de um *dashboard* têm que ter consciência que um *dashboard* não oferece uma imagem completa[65].

- **Os *dashboards* nem sempre respondem a objetivos:** métricas referidas pelo item acima, por exemplo, o grau de satisfação dos clientes, não são coletadas, e portanto não podem ser apresentadas num *dashboard*. Porém, podem estar mais alinhadas com os objetivos da empresa do que outras métricas, como a contagem das tarefas em aberto.

  Simultaneamente, um *dashboard* não pode responder simultaneamente aos objetivos de gestores e colaboradores, pois as necessidades de informação são diferentes. Um colaborador pretende obter conhecimento sobre o seu desempenho, enquanto que as equipas de desenvolvimento pretendem monitorizar o desempenho como equipa ou dos projetos com que trabalham, e a equipa de gestão pretende usar um *dashboard* para tomar decisões. Em resultado, têm sido propostos *dashboards tailored* [74, 43] (i.e. o termo *tailored* significa elaborado conforme requisitos/especificações). Um *dashboard tailored* é um *dashboard* que pode variar a sua aparência e funcionalidades para corresponder aos requisitos dos utilizadores.

2.5 Ferramentas de *software analytics*

Existem uma variedade de ferramentas para *software analytics*. Nesta secção, exploramos uma seleção destas ferramentas, que se destacam pela criação de *dashboards*. Porém, todas as ferramentas referidas são comercializadas.

2.5.1 Power BI

O Power BI [75], desenvolvido pela Microsoft, permite criar *dashboards* sofisticados a partir de uma variedade de fontes de informação, inclusive uma base de dados ou um serviço cloud. Depois, a ferramenta permite a um utilizador definir qual o modelo de dados a utilizar sobre os dados das fontes.

Ou seja, esta ferramenta não foi concebida como uma ferramenta para análise de código, mas pode ser integrada noutra ferramenta que colete informação sobre produtividade e qualidade de código. Ou, utilizando a linguagem que esta faculta, a linguagem DAX (Data Analysis Expressions), podemos criar métricas personalizadas que agregam, calculam ou resumem os dados das fontes.

Oferece uma ampla variedade de gráficos e recursos de formatação para criar visualizações personalizadas, que atendem aos requisitos dos utilizadores. Ademais, podemos aplicar filtros sobre uma visualização, ou até utilizarmos a funcionalidade *drill-through* para visualizarmos com maior detalhe um elemento de uma visualização.

Esta ferramenta é comummente usada em diversos setores e, inclusive, pode ser adaptada para o propósito desta dissertação, i.e. como ferramenta que apresenta visualizações sobre produtividade e/ou qualidade do código produzido, com vários graus de granularidade. Isto porque, o Power BI é, mais especificamente, uma ferramenta de *data analytics*.

2.5.2 Tableau

Outra ferramenta também amplamente utilizada para *data analytics*, e para *software analytics*, que possui esta mesma adaptabilidade, é o Tableau [66], desenvolvida pela Salesforce.

Esta é bastante similar à ferramenta Power BI. Porém, o Tableau é frequentemente considerado mais poderoso quanto aos recursos avançados que oferece para a análise das visualizações. Isto porque, o Tableau beneficia da integração com a ferramenta Einstein AI [21, 2]. Esta é uma ferramenta para análise preditiva, que utiliza algoritmos de aprendizagem automática para identificar tendências e padrões nos dados, gerando previsões e *insights*.

De facto, ambas as ferramentas, o Power BI e o Tableau, estão entre as melhores classificadas, tanto como ferramentas para *software analytics*, como ferramentas para visualização de informação, pela Forbes [8, 69].

Todavia, existem ferramentas mais adequadas ao objetivo desta dissertação, i.e. ferramentas analíticas de gestão do desenvolvimento.

2.5.3 WayDev

Esta é uma ferramenta automática, concebida para que gestores obtenham visibilidade sobre o desenvolvimento Ágil de *software* [44]. O WayDev promete fornecer *insights* aos gestores, para que tomem decisões suportadas por informação. Ademais, promete aprimorar o desenvolvimento de *software*, impulsionando a produtividade e a qualidade do código produzido.

Esta ferramenta pode ser integrada com repositórios de código-fonte, como o Git, ferramentas como o Jira para planeamento e acompanhamento das tarefas dos projetos, ferramentas de integração e entrega contínuas, como o Jenkins, calendários e chats de comunicação.

Com base nas informações destas diversas fontes, o WayDev oferece várias visualizações. Ora, uma visualização é caraterizada por um conjunto definido de métricas computadas. Por conseguinte, permite a utilizadores criar um *dashboard* personalizado, utilizando as visualizações que a ferramenta oferece, ou criando visualizações personalizadas.

Oferece visualizações sobre o trabalho de um colaborador específico. Por exemplo, existe uma visualização designada "Developer Summary"[20], que apresenta uma visão condensada sobre um sumário de métricas referentes ao trabalho de um colaborador - que é formado por métricas como a contagem dos *commits* realizados, e a eficiência, que corresponde à percentagem do código produzido que é produtivo, ou seja, isto requer o equilíbrio entre a produção de código e a longevidade do código. Em suma, compreende métricas sobre produtividade e sobre a qualidade do código produzido. Ademais, existe uma visualização, designada "Developer Progress"[19], que apresenta uma visão sobre o progresso de um colaborador de *Sprint* em *Sprint*, e permite compreender a sua dinâmica de trabalho e identificar oportunidades de formação.

Também, existem visualizações sobre o trabalho de uma equipa. Por exemplo, existe uma visualização designada "Work Log"[78], que apresenta uma visão sobre o trabalho realizado pela equipa e, também, sobre a comunicação entre os colaboradores da equipa. Permite navegar para o perfil "Developer Summary"de um colaborador presente num "Work Log". Simultaneamente, existe uma visualização, "Team Progress"[68], que oferece uma visão sobre o progresso de uma equipa de *Sprint* a *Sprint*, mês a mês, ou trimestre a trimestre.

Esta ferramenta permite obter uma visão granular para gerir melhor uma equipa com informação objetiva. Dispõem destas visualizações em tempo real, e portanto, impulsiona a eficiência, acelera o desenvolvimento de *software*, e aumenta a produtividade da equipa.

2.5.4 Velocity

É uma ferramenta automática [72], desenvolvida pela Code Climate, que fornece visibilidade e previsibilidade aos gestores através de informação objetiva. Esta informação é

apresentada através de visualizações, e com isto obtemos *insights* que se traduzem facilmente em ações, i.e. sobre os quais os gestores podem tomar decisões suportadas por informação clara. Estes *insights* permitem responder questões críticas, verificar suposições dos gestores, identificar padrões e acompanhar a evolução do progresso com métricas chave.

Esta ferramenta integra-se nas ferramentas que uma equipa utiliza, como o Bitbucket, o Jira, etc. e ingere a partir destas a informação que produzem, para realizar uma análise. Esta informação é transformada e apresentada em visualizações que a ferramenta oferece, e que dispõe num *dashboard*.

Por exemplo, para promover a excelência dos colaboradores de uma equipa, a Velocity oferece a visualização "Developer360"[35], que fornece uma visão abrangente sobre o trabalho, desempenho e competências/aptidões em linguagens de um colaborador. Com isto, os gestores podem determinar se algum colaborador está com dificuldade a superar um obstáculo que requeira ajuda. Ora, esta visualização é composta por várias métricas, sendo uma delas o impacto, que mede a magnitude de modificações efetuadas ao código produzido por um colaborador ao longo do tempo. Esta métrica pode ajudar a revelar colaboradores que merecem reconhecimento, ou servir como um alerta precoce de que um colaborador pode ter necessidade de alguma formação. Esta visualização compreende, portanto, métricas sobre produtividade, mas também sobre a qualidade do código produzido.

Depois, a ferramenta apresenta também uma visualização designada "Team360"[11], que oferece uma visão sobre o trabalho de uma equipa e o seu desempenho. Com isto, os gestores podem, por exemplo, assegurar que a equipa se concentra em trabalho prioritário, cumprindo as metas estabelecidas para a entrega incremental do *software*.

Estas visualizações ajudam os gestores a compreender se as equipas estão a ter sucesso e em que é que podem ajudá-las a melhorar. Ademais, esta ferramenta pode ser utilizada também por colaboradores, para acompanharem o seu progresso e o progresso em equipa, e identificarem áreas a melhorar.

2.5.5 Outras ferramentas

Apesar de não abordar nesta secção mais ferramentas, apresentando-as em detalhe, existem outras também relevantes como a Swarmia [32], a Flow (desenvolvida pela Pluralsight) [51], a LinearB [3], etc.

3

Tecnologias e Ferramentas utilizadas

Este capítulo aborda as tecnologias e ferramentas utilizadas nesta dissertação, que ou fazem parte da metodologia da Opensoft e/ou são específicas do projeto CORQ.

3.1 Confluence

O Confluence [14] é um *wiki* corporativo que existe sobre um repositório centralizado de conhecimento, desenvolvido pela empresa Atlassian. É uma ferramenta versátil que procura melhorar a colaboração das equipas de desenvolvimento, a documentação e a gestão dos projetos.

Esta ferramenta permite criar e editar páginas, colaborativamente, o que, por sua vez, permite que os colaboradores de uma empresa capturem e colaborem em produzir documentação dos projetos em tempo real. Também permite que uma equipa crie um espaço dedicado por projeto para organizar e partilhar informação sobre o trabalho realizado para esse projeto. Isto promove uma colaboração eficaz entre os colaboradores de uma equipa, evita qualquer mal-entendido e facilita que a equipa trabalhe eficientemente, alinhada com os objetivos que têm em comum. Ademais, a ferramenta oferece recursos avançados de pesquisa e organização, tornando mais fácil para os utilizadores localizar com rapidez as informações que necessitam.

Portanto, a instância Confluente da Opensoft tem um espaço dedicado ao projeto CORQ, que contém documentação sobre este.

3.2 Crowd

É uma ferramenta [15] também desenvolvida pela Atlassian, que centraliza e simplifica o controlo dos acessos aos seus aplicativos - como o Confluence, o Jira e o Bitbucket -, oferecendo uma única fonte de autenticação, o que é designado de Single Sign-On (SSO), para os colaboradores. Por conseguinte, os colaboradores têm acesso às diversas ferramentas utilizando a mesma credencial.

Sendo o Crowd a ferramenta que gere a identidade dos colaboradores, a ferramenta CORQ obtém através da Application Programming Interface (API) do Crowd a informação sobre os colaboradores da Opensoft.

3.3 MariaDB

É um Sistema de Gestão de Base de Dados (SGBD) *open-source* [47], que provém do MySQL. É dos mais populares entre os SGBD relacionais. Esta popularidade é atribuída à sua robustez, escalabilidade e desempenho, sendo uma escolha confiável para uma variedade de aplicações.

Esta é a base de dados utilizada pela ferramenta CORQ, que foi escolhida e configurada com o trabalho prévio, ou seja, fora do contexto desta dissertação, mas é uma escolha eficaz para atender aos requisitos específicos do projeto.

3.4 SonarQube

O SonarQube [13] é uma ferramenta automática para revisão do código dos projetos, em mais de 30 linguagens de programação. Esta ferramenta *open-source* foi desenvolvida pela empresa SonarSource. É através desta que a Opensoft procura aprimorar a qualidade do código-fonte dos seus projetos.

Portanto, a ferramenta realiza análise estática sobre código-fonte de um projeto, gerando um relatório que aponta um ou mais blocos de código que apresentam *bugs*, vulnerabilidades, etc. que afetam a qualidade de código. Pode também ser apontado qualquer bloco que requeira uma refatoração ou otimização, pois esta constitui também uma oportunidade de melhoria da qualidade do código-fonte.

Para cada bloco identificado com um problema, o SonarQube fornece uma descrição detalhada que explica a natureza deste problema. Ora, este também oferece suporte eficaz para a resolução dos problemas identificados - para orientar os colaboradores com a correção, a ferramenta apresenta exemplos de código em que o problema ocorre, bem como exemplos que demonstram como o problema pode ser resolvido. Essa abordagem não só agiliza a correção, mas também enriquece a compreensão dos colaboradores sobre as melhores práticas de codificação.

Ademais, a análise estática coleta várias métricas sobre fatores que afetam a qualidade do código-fonte, como a sua complexidade, legibilidade, segurança, conformidade com padrões de codificação definidos para a linguagem em uso, entre outros.

Esta ferramenta oferece uma visão abrangente sobre a saúde do código, permite acompanharmos a evolução desta e promove a melhoria contínua da qualidade do código-fonte dos projetos em todas as etapas do ciclo de vida dos projetos.

Nesta dissertação, o objetivo principal foi tirar partido desta análise estática efetuada pela ferramenta SonarQube para obter uma métrica sobre a qualidade do código produzido diariamente, pelos colaboradores da Opensoft.

3.5 Jira

É uma ferramenta [41] para planeamento, acompanhamento e gestão do tempo requerido para a concretização das tarefas definidas para um projeto. Esta ferramenta permite definir a prioridade das tarefas, e atribuí-las a um dos colaboradores de uma equipa de desenvolvimento. É outra das ferramentas oferecidas pela empresa Atlassian.

Portanto, esta ferramenta foi utilizada para planeamento e registo das horas de trabalho nesta dissertação, com o projeto CORQ.

3.6 Bitbucket

O Bitbucket [4] é uma plataforma que armazena os repositórios de código-fonte dos projetos de uma empresa. Disponibiliza às equipas uma plataforma centralizada para gerirem os repositórios de código, com controlo de versões, para acompanharem as modificações, e colaborarem para o desenvolvimento dos projetos. É também um produto da empresa Atlassian.

Sendo a ferramenta que gere os repositórios da Opensoft, a ferramenta CORQ obtém através da API do Bitbucket a informação sobre os projetos.

3.7 Spring

É uma *framework open-source* [61] que oferece uma infraestrutura de suporte para o desenvolvimento de aplicações Java. Esta *framework* simplifica significativamente o desenvolvimento de aplicações Java, por meio de várias abordagens.

Uma das abordagens é a sua arquitetura, que assenta nos princípios de inversão de controlo e injeção de dependências, o que significa que o Spring assume a responsabilidade de criar objetos e gerir as suas dependências, automaticamente.

Outra abordagem provém dos diversos módulos que o Spring oferece, e que constituem soluções prontas a uso para o desenvolvimento Java. Dito isto, estes módulos economizam tempo e esforço, permitindo que os colaboradores se concentrem em criar as funcionalidades específicas das suas aplicações.

Por exemplo, a ferramenta CORQ utiliza o módulo Spring Boot [63], que acelera o início do desenvolvimento de um projeto, pois automatiza a sua configuração. Também recorremos ao módulo Spring Data JPA [64], que agiliza o acesso à base de dados para escritas e consultas. Por fim, também utilizamos o módulo Spring Batch [62], para o processamento em *batch* eficiente.

3.8 *Batch Job*

Fez sentido introduzir nesta secção o conceito *Batch Job*, já que este será posteriormente referido várias vezes com o capítulo 5.

O processamento *batch* (significa "em lote") é definido como a execução de tarefas em *background*, programadas para executarem numa determinada janela horária, conforme os recursos permitirem.

Ora, como a ferramenta CORQ pretende, a partir das informações diárias geradas pelas ferramentas de apoio ao desenvolvimento, gerar métricas, tem de realizar este processamento diariamente. Portanto, todo este processamento tem que ser confiável, pois tratam-se de dados críticos, e tem de ser feito periodicamente e sem interrupção. Este tipo de aplicação pode ser implementada mais eficientemente como um *Job* (i.e. uma tarefa). Um *Job* é uma aplicação que faz o processamento de uma quantidade finita de dados, sem interrupção da sua execução.

Implementar um *Job* pode ser custoso, pois algumas tarefas são simples, outras mais complexas. Algumas são rápidas, outras morosas. É fundamental ter em conta estes requisitos quando se pretende uma implementação adequada. Porém, estas questões são já suportadas pela *framework* Spring Batch [62].

O Spring Batch segue a tradicional arquitetura de *Batch Job*, em que um processamento *batch* é definido por um *Job*. Este *Job* é composto por uma ou várias etapas, que têm a designação de *Steps*, e que executam sequencialmente ou paralelamente, conforme configurado. Cada *Step* executa uma ou mais tarefas individuais, que são designadas de *Tasklets*.

Um *Job* é executado por um *JobLauncher* e, os metadados sobre a configuração e execução do *Job* e respetivos *Steps* são armazenados num *JobRepository*, que funciona como uma unidade de persistência de uma execução do *batch*.

Comummente, uma *Tasklet* lê um conjunto de dados de *input*, que são processados (transformados ou validados), originando o conjunto de dados de output, que é, por fim, escrito. Nesta dissertação, os *jobs* da ferramenta CORQ leem a partir da base de dados e após o processamento, escrevem para a base de dados.

3.9 JGit

O JGit [40] é uma biblioteca *open-source* para Java, que oferece uma implementação do sistema de controlo de versões distribuído Git. Esta biblioteca foi criada e é mantida pela Eclipse Foundation.

Com esta biblioteca podemos criar ou clonar repositórios Git, realizar *commits*, criar *branches*, entre outras funcionalidades, diretamente a partir de aplicações Java, sem dependermos de uma consola externa.

Embora esta biblioteca tenha sido essencial para a ferramenta CORQ gerar métricas de produtividade e de qualidade do código produzido, foi com dificuldade que compreendi como utilizá-la para atender a objetivos específicos da implementação.

Por exemplo, contrariamente ao que esperava, esta biblioteca não possui nenhum método que, a partir de um *commit*, retorne os ficheiros modificados que foram *committed*. Isto é, um *commit* é representado por esta biblioteca como um objeto *RevCommit*, que

contém um objeto *RevTree*, o que representa a referência para a árvore de ficheiros do *commit*. Por conseguinte, para obtermos os ficheiros modificados do *commit* especificado, temos de percorrer a árvore de ficheiros deste, e compará-la com a árvore de ficheiros do *commit* pai.

Durante a fase de implementação, encontrei um repositório com fragmentos de código que demonstram como utilizar esta biblioteca para vários casos de uso específicos [39], o que foi fundamental para aprofundar o meu entendimento.

3.10 Apache Maven

É uma ferramenta *open-source* [49] de compilação automática, amplamente utilizada em projetos de desenvolvimento de *software*, particularmente em Java, embora ofereça suporte a várias outras linguagens de programação. Esta ferramenta foi criada e é mantida pela Apache Software Foundation.

Esta ferramenta é conhecida por simplificar significativamente a gestão das dependências, pois adiciona eficientemente bibliotecas externas a um projeto, assegura que estas estão disponíveis e que são utilizadas versões compatíveis, o que é essencial para a estabilidade e confiabilidade dos projetos.

Portanto, utilizamos esta ferramenta para compilação do código-fonte da ferramenta CORQ e para gerir as dependências desta.

3.11 Jenkins

O Jenkins [38] é um servidor de automação *open-source*, que automatiza o *build*, teste e *deploy* dos projetos, facilitando a integração e entrega contínuas.

Portanto, este é utilizado para o *build* e teste do projeto CORQ, gerando um arquivo zip que contém as dependências e o JAR da aplicação. Posteriormente, o Jenkins faz o *deploy* deste arquivo zip, instalando-o numa diretoria da máquina remota do CORQ, onde a ferramenta executa.

3.12 ELK Stack

Ora, a ELK Stack [22] é formalmente constituída pelos serviços Logstash, ElasticSearch e Kibana. Porém, comummente, arquiteturas que utilizam a ELK Stack adotam o serviço Beats, que coleta *logs*, i.e. ficheiros com extensão ".log", contendo informação e transmite-os à ELK Stack.

Portanto, a ferramenta CORQ aproveita esta arquitetura e, quando computa métricas, produz simultaneamente um *log* com a informação das métricas. Este *log* é coletado pelo Beats, e transmitido ao Logstash, que agrega esta informação e a formata, para que seja depois armazenada num índice do ElasticSearch dedicado ao CORQ. É a partir deste

índice que o Kibana obtém a informação para gerar visualizações. A figura 3.1 apresenta esta arquitetura esquematizada.

w

Figura 3.1: Arquitetura da ELK Stack com Beats. (Retirado de [23])

4

Trabalho Prévio

Esta dissertação foi proposta pela empresa Opensoft, em âmbito de um projeto seu - a ferramenta CORQ. Por conseguinte, neste capítulo, apresento a Opensoft, e também a ferramenta CORQ com algum detalhe, i.e. a primeira versão do CORQ, anterior às contribuições desta dissertação para a sua evolução. Sobre esta, retrato a sua arquitetura, quais fontes de informação utiliza, que métricas sobre produtividade computa, e que visualizações o *dashboard* apresenta.

4.1 Opensoft

A Opensoft é uma empresa portuguesa fundada em 2001, especializada em consultoria tecnológica e desenvolvimento de soluções tecnológicas. Tem por missão criar soluções tecnológicas inovadoras e de referência, que simplificam o dia-a-dia das sociedades.

Tem colaborado com clientes de referência a nível nacional, quer da Administração Pública, como é o caso da Autoridade Tributária e Aduaneira (AT), quer do Setor Privado, como a SIBS, a Sociedade Interbancária de Serviços. Com a AT, o trabalho da Opensoft tem sido utilizado para transformar várias áreas, como a reformulação da declaração de IRS, através do produto Lightweightform. O produto mais antigo da Opensoft, o Sistema Integrado de Métodos Notariais (em acrónimo SIMN), faz a gestão de cartórios notariais, automatizando várias tarefas, como as comunicações obrigatórias à AT. Este produto está em efetivo nos cartórios de todo o país. Ademais, o produto SAFTPRO (acrónimo de SAF-T Processing System) foi desenvolvido para várias ATs, e em 2020 teve como primeiro cliente o Ministério das Finanças de Cabo Verde.

Foi reconhecida em 2022 com o estatuto PME líder [53], pela aposta em estratégias de crescimento e liderança baseada em critérios como a inovação e a internacionalização. Foi, também, reconhecida em 2021 com o estatuto PME excelência [52], pelo desempenho económico-financeiro e de gestão, por ser uma empresa competitiva num contexto económico cada vez mais exigente.

4.2 Ferramenta CORQ

O CORQ é uma ferramenta de *software analytics*, que coleta e efetua o processamento de grandes volumes de dados, para gerar informação que permite à equipa de gestão obter *insights* que direcionem a tomada de decisões. Por conseguinte, apresenta aos seus utilizadores um *dashboard web*, que contém visualizações sobre métricas de produtividade aquando do desenvolvimento de código.

Uma das visualizações, designada "Lines per day per project", apresenta por dia, as linhas de código adicionadas para os vários projetos da Opensoft. Outra visualização, a "Lines per day per user", apresenta por dia, as linhas de código adicionadas pelos vários colaboradores da Opensoft. Um utilizador pode aplicar a estas visualizações um filtro por intervalo de tempo.

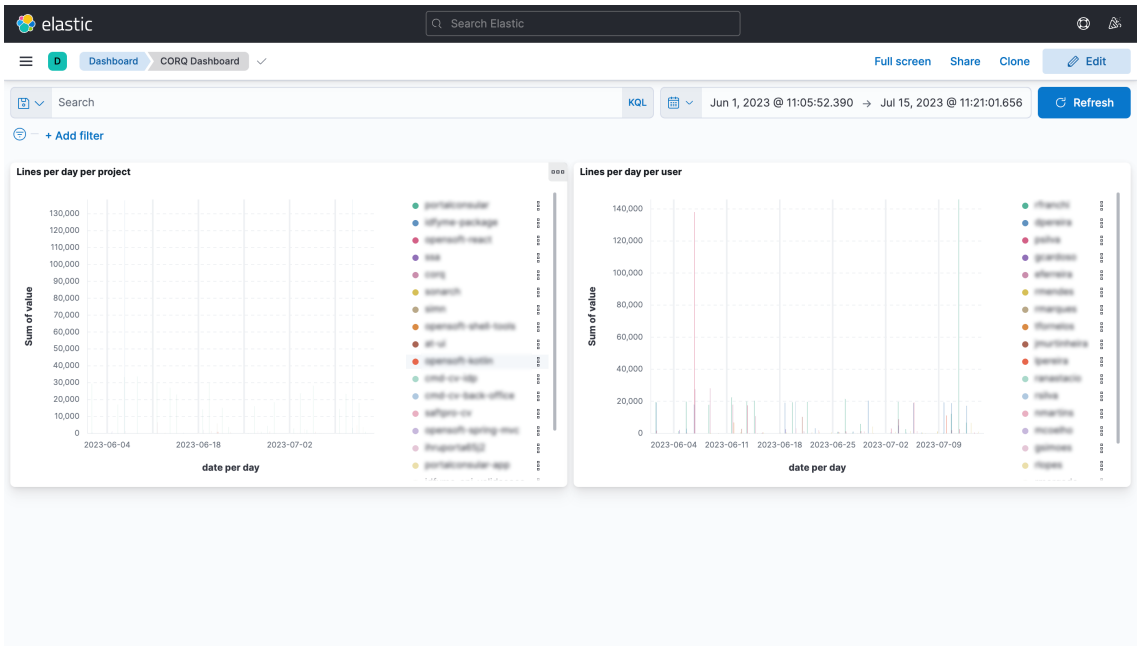

Figura 4.1: *Dashboard* CORQ

Estas visualizações são construídas baseadas numa métrica que é computada pelo *back-end*, a métrica `LINES_ADDED`, que representa as linhas de código adicionadas num dia com o trabalho de um colaborador num projeto. Em verdade, o *back-end* computa também outras métricas, `LINES_REMOVED` e `LINES_CHANGED`, que representam, respetivamente, as linhas de código removidas num dia com o trabalho de um colaborador num projeto, e a diferença entre linhas adicionadas e removidas. Porém, sobre estas não existe nenhuma visualização.

O *front-end* é construído com recurso ao serviço Kibana, que é parte da ELK Stack. Isto significa que, o *back-end* está integrado com a ELK Stack.

O *back-end* da ferramenta é constituído, em parte, por um componente `corq-batch`, que é uma aplicação *Batch Job*. Este componente obtém, a partir da API do Bitbucket, os

commits realizados por dia, por um colaborador para um projeto, e com estes, computa métricas sobre linhas. Estas métricas computadas são registadas na base de dados pelo componente `corq-common`, que gere a camada de persistência da ferramenta.

Com a figura 4.2 apresentamos a arquitetura da ferramenta, que é, resumidamente, composta por: base de dados, componente `corq-common`, componente `corq-batch` e ELK Stack.

Como podemos observar pela figura, a componente `corq-batch` também estabelece comunicação com a API do Crowd, para obter informação sobre os colaboradores da Opensoft.

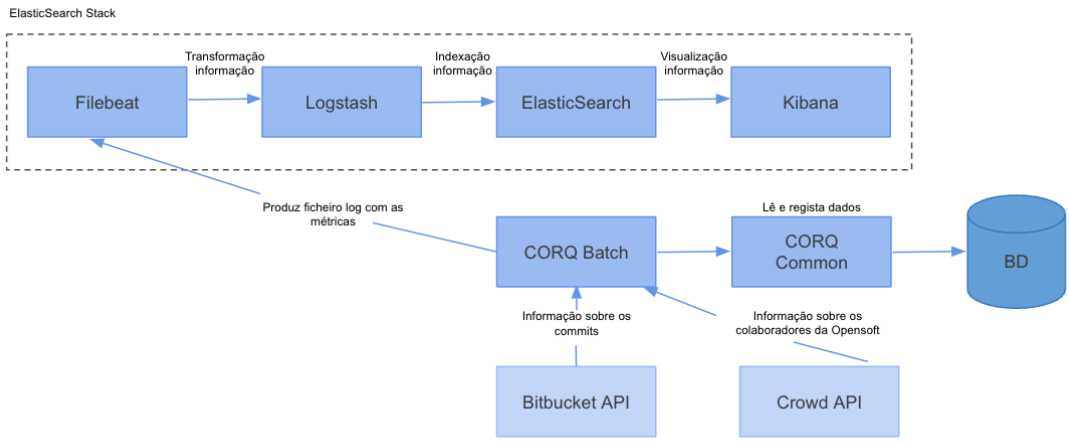

Figura 4.2: Arquitetura da ferramenta CORQ.

A ferramenta CORQ tem que executar a cada 24h, para gerar informação sobre o trabalho realizado nas 24h anteriores. Por conseguinte, a informação disponibilizada pelo *dashboard* é atualizada a cada 24h. Ademais, é esperado também que o *dashboard* esteja sempre disponível para consulta dos utilizadores.

4.3 *User Stories*

Os requisitos funcionais são descritos seguidamente, através de *User Stories*, que foram definidas com a primeira versão da ferramenta:

US01 - Como utilizador, pretendo autenticar-me perante o CORQ utilizando credenciais.

US02 - Como utilizador autenticado, pretendo consultar informação sobre métricas de produtividade e/ou qualidade de código de um colaborador específico, num projeto específico.

US03 - Como utilizador autenticado, pretendo consultar informação sobre métricas de produtividade e/ou qualidade de código de um colaborador específico.

US04 - Como utilizador autenticado, pretendo consultar informação sobre métricas de produtividade e/ou qualidade de código de um projeto específico.

US05 - Como utilizador autenticado, pretendo consultar informação sobre métricas de produtividade e/ou qualidade de código da Opensoft.

US06 - Como utilizador autenticado, pretendo consultar informação sobre métricas de produtividade e/ou qualidade de código num intervalo de tempo específico.

Destas, nenhuma está concretizada completamente, uma vez que a ferramenta não possui métricas sobre qualidade do código produzido, nem o dashboard está disponível para uso, pois não tem a autenticação com o Crowd integrada.

Ora, uma vez mais, com o trabalho desta dissertação é esperado resolvermos o que falta para concretizarmos completamente as *User Stories*.

5

Solução e Implementação

Este capítulo apresenta, com detalhe, a solução adotada para a ferramenta CORQ gerar informação sobre qualidade de código, apresentando a implementação que contribuiu para a evolução e aprimoramento da ferramenta CORQ.

5.1 Análise e desenho de uma métrica de qualidade de código

O principal objetivo da dissertação foi mensurarmos a qualidade do código produzido pela Opensoft, para produzirmos e apresentarmos através de uma visualização, informação que auxilia a equipa de gestão a responder à questão "Como evolui a qualidade do código produzido pela Opensoft ao longo do tempo?".

Com o início do trabalho foi-me pedido que explorasse a ferramenta SonarQube, a fim de a partir desta construir uma métrica para mensuração da qualidade do código produzido por dia, por um colaborador específico, integrado num projeto em específico. Por conseguinte, esta métrica poderia ser, para um dia em específico, agregável por colaborador ou por projeto, ou por ambos, i.e. da Opensoft.

Ora, o SonarQube, sendo uma ferramenta de análise estática com base em regras, apresenta o conceito de *Quality Profile* [54]. Um *Quality Profile* é um conjunto predefinido ou personalizado de regras de qualidade de código a aplicar a um projeto, para avaliarmos e assegurarmos a sua conformidade com as boas práticas de desenvolvimento. Porém, cada linguagem de programação tem as suas regras que definem o que é considerado código de boa qualidade. Portanto, para uma análise com precisão, o SonarQube oferece diversos *Quality Profiles*, cada um adequado a uma linguagem específica. Durante a análise estática realizada pelo SonarQube, as regras de um ou mais *Quality Profiles* correspondentes às linguagens que o projeto apresenta, são aplicadas ao código-fonte. Quando é identificado um bloco de código que viola uma das regras, o SonarQube gera um problema, que a ferramenta designa de *Issue* [36]. Um *Issue* é, por isso, uma violação de uma regra de qualidade de código, que aponta para um bloco de código que precisa de ser revisto ou corrigido, para melhorarmos a qualidade e legibilidade do código. Para além do bloco de código relevante, um *Issue* apresenta a descrição do problema e, até mesmo sugestões de

como podemos corrigir o problema.

Existem três tipos de *Issues* [36]:

1. *Bug*: um lapso quando se desenvolve código que pode levar a um erro ou comportamento inesperado em tempo de execução;
2. Vulnerabilidade: um ponto do código que está vulnerável a ataques;
3. *Code Smell*: um problema de manutenção que torna o código confuso e portanto difícil de manter.

Um *Issue* é caraterizado pela sua severidade, podendo ser uma das cinco seguintes [36]:

1. BLOCKER: *Bug* com elevada probabilidade de ter impacto sobre o comportamento da aplicação em produção. Por exemplo, a rotura da memória ou uma conexão JDBC que não foi fechada são *Issues* com severidade BLOCKER que devem ser resolvidos de imediato!
2. CRITICAL: *Bug* com baixa probabilidade de afetar o comportamento da aplicação em produção ou um *Issue* que representa uma falha de segurança. Por exemplo, um bloco catch vazio ou injeção SQL são *Issues* com severidade CRITICAL. O código deve ser revisto imediatamente!
3. MAJOR: Uma falha de qualidade que tem grande impacto sobre a produtividade dos colaboradores. Por exemplo, blocos de código duplicados, parâmetros não usados, blocos de código que não têm *coverage*, i.e. que não estão testados com testes unitários, são exemplos de *Issues* com severidade MAJOR.
4. MINOR: Uma falha de qualidade que afeta ligeiramente a produtividade dos colaboradores. Por exemplo, linhas de código não devem ser demasiado longas, um switch deve ter pelo menos 3 cases, são exemplos de *Issues* com severidade MINOR.
5. INFO: Não é um *bug* nem uma falha de qualidade, apenas uma descoberta. Por exemplo, métodos que aceitam um número variável de argumentos são apontados como *Issues* com severidade INFO.

Ou seja, através da aplicação dos *Quality Profiles* e da consequente resolução dos *Issues* que surgem, os colaboradores melhoram gradualmente a qualidade do código - reduzindo a ocorrência de problemas comuns, aumentando a manutenibilidade e facilitando a deteção precoce de potenciais *bugs* e vulnerabilidades.

Consequentemente, optámos por construir uma métrica de qualidade de código baseada sobre a mensuração de *Issues*, pelas razões que em seguida apresento:

1. Os *Issues* gerados pelo SonarQube fornecem uma medida objetiva e quantificável da qualidade do código produzido.

2. Dito isto, ao acompanharmos e mensurarmos os *Issues* gerados com a análise, podemos estabelecer uma métrica de qualidade de código que avalia a evolução da qualidade de código ao longo do tempo - isto porque, a diminuição da quantidade ou gravidade dos *Issues* ao longo do tempo indica uma melhoria contínua da qualidade de código.

3. Como os *Issues* são baseados em regras de qualidade de código que refletem as melhores práticas de desenvolvimento, ao concebermos uma métrica que tem em conta os *Issues*, medimos a aderência do código aos padrões e boas práticas estabelecidas, o que é essencial para assegurarmos código com boa qualidade.

4. Ademais, ao concebermos uma métrica que considera os *Issues*, promovemos a transparência e visibilidade dos problemas de qualidade de código para toda a equipa de desenvolvimento. Isto facilita a comunicação e colaboração, pois permite que todos os elementos de uma equipa estejam cientes dos desafios e trabalhem juntos para resolvê-los - os colaboradores terão *Issues* objetivo sobre as suas contribuições e poderão trabalhar ativamente para a resolução dos *Issues* identificados. Por resultado, promovemos um ciclo de melhoria contínua, onde as habilidades e conhecimentos dos colaboradores são aprimorados e a qualidade do código é elevada.

5. Da análise de uma métrica de qualidade com base nos issues, podemos tomar decisões informadas para aprimorarmos o processo de desenvolvimento - a métrica agregada por projeto e empresa permite ter uma visão ampla da qualidade do trabalho em diferentes contextos. Isto facilita que identifiquemos padrões recorrentes ou áreas críticas com problemas de qualidade, o que permite priorizar as melhorias e alocar recursos de forma eficiente.

Portanto, ao concebermos uma métrica de qualidade de código baseada nos *Issues* gerados pelo SonarQube para avaliar o trabalho de um colaborador, estamos a adotar uma abordagem objetiva, centrada na qualidade do código, agregável por projeto e empresa, que promove a melhoria contínua e ajuda a garantir um alto padrão de qualidade no desenvolvimento de *software*.

5.2 Construção da métrica

Após a fase de desenho, seguiu-se a fase de implementação. Esta fase começou com uma premissa - o objetivo é medirmos a qualidade do trabalho realizado num dia e, a unidade de trabalho é o *commit*, logo pretendemos obter uma medida de qualidade de código por cada *commit* realizado por um colaborador.

5.2.1 SonarQube

A empresa possui uma instância do SonarQube, onde estão hospedados vários projetos. Contudo, a intenção não é utilizarmos esta instância para analisarmos todos os *commits*

de todos os projetos, pois isso sobrecarregaria a instância.

Uma instância do SonarQube é composta por três componentes, como apresentado pela figura 5.1 [34]:

1. O servidor SonarQube, que executa os seguintes processos:
   - Um servidor *web* que apresenta a interface de utilizador do SonarQube;
   - Um servidor de pesquisa que utiliza o ElasticSearch;
   - Um motor de computação encarregue de fazer o processamento dos relatórios de análise de código e de os armazenar na base de dados do SonarQube.
2. A base de dados, para armazenamento do seguinte:
   - Métricas e *Issues* sobre qualidade de código e segurança gerados durante as análises do código;
   - A configuração da instância do SonarQube.
3. Um ou mais scanners em execução nos servidores de *build* ou integração contínua do utilizador para analisar projetos.

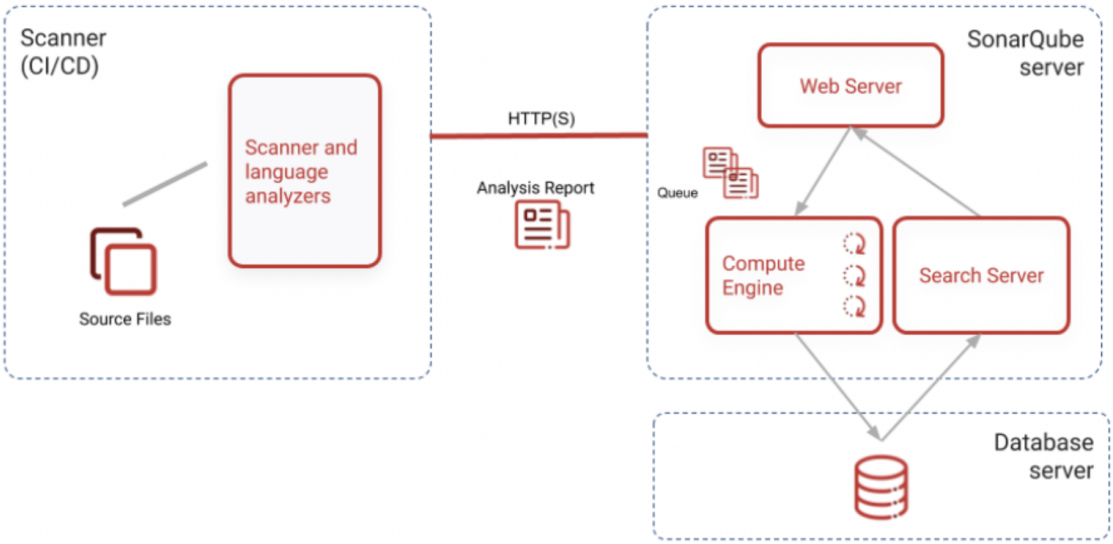

Figura 5.1: Desenho de uma instância do SonarQube e seus componentes.

Ou seja, os scanners são responsáveis por fazer a análise do código-fonte de um projeto. São também responsáveis por gerar com cada análise um relatório, e enviá-lo para o servidor do SonarQube. Um relatório de análise de código contém os *Issues* encontrados durante a análise. O servidor recebe e faz o processamento dos relatórios de análise de código, fornecendo uma interface para visualização dos resultados. Através da interface, os

colaboradores podem obter informação detalhada sobre cada *Issue* encontrado e visualizar métricas de qualidade de código. Este é o funcionamento básico do SonarQube.

Portanto, com a análise do código-fonte de um projeto, é enviado o relatório resultante para o servidor. Porém, para fazermos análise *commit* a *commit*, bastaria utilizarmos os scanners para obtermos o relatório da análise de código, sem que estes publiquem os resultados para o servidor. Para mais que não nos interessa o que a interface de utilizador do SonarQube oferece, pois a ferramenta CORQ tem a sua própria interface.

Como tal, e porque existe uma elevada gama de parâmetros de análise opcionais que podemos definir, comecei por verificar se não haveria um parâmetro que nos permitisse "desligar"a publicação dos resultados para o servidor, o que não existe. Inclusive, modifiquei o URL do servidor para comprovarmos se assim o relatório não seria publicado. Todavia, como descobri após esta tentativa e por análise do *output* em modo *debug*, verifiquei que ao início da execução de uma análise, os scanners fazem vários pedidos ao servidor para obterem os *Quality Profiles* de um projeto, regras de cada *Quality Profile*, *settings* do projeto, etc. Portanto, modificando o parâmetro que define o Uniform Resource Locator (URL) do servidor, impossibilitamos desde logo que a análise se inicie. Como não pretendemos fazer modificações ao código-fonte do SonarQube, pois apesar de ser código *open-source*, este trabalho não contribui para essa ferramenta, não podemos simplesmente definir o URL do servidor e depois de o relatório ser gerado, e antes da publicação do mesmo, modificar este parâmetro para que a publicação falhe.

Procurei ainda se existe algum método da API do SonarQube que desencadeia uma análise, mas sem sucesso. Aliás, como mais tarde li, esta API possui apenas os métodos que uma instância do SonarQube precisa para fazer uma análise e os que os utilizadores utilizam para obter os resultados após análise.

Em suma, não obtive nenhuma solução para, recorrendo ao SonarQube, fazer a análise dos *commits* de um projeto.

5.2.2 SonarLint

Porém, a empresa SonarSource possui outros produtos *open-source* para análise estática de código-fonte, como o SonarLint [45]. Contrariamente ao funcionamento do SonarQube, o SonarLint funciona localmente, sendo uma extensão para integração com um Integrated Development Environment (IDE), i.e. um ambiente de desenvolvimento integrado. O SonarLint oferece análise estática do código-fonte, gerando relatórios em tempo real. Isto significa que, enquanto um colaborador escreve código, o SonarLint verifica continuamente o código-fonte, destacando *Issues* e fornecendo *feedback* imediato sobre a qualidade de código.

Esta integração contínua com o IDE é vantajosa, pois ajuda a que mantenhamos um alto nível de qualidade do código desde as fases iniciais do desenvolvimento, facilitando a correção de *Issues* antes que se tornem mais complexos e custosos de resolver.

É importante destacarmos que esta ferramenta não substitui o SonarQube, mas complementa as suas funcionalidades, permitindo que um colaborador realize análises de qualidade de código em tempo real, diretamente a partir do seu IDE de preferência. Comparativamente, o SonarQube está separado dos IDEs e realiza análises mais profundas e abrangentes de projetos inteiros, em momentos específicos, e fornecendo relatórios detalhados sobre a qualidade, métricas e *Issues* identificados. Também, sendo uma plataforma centralizada, os resultados das análises são armazenados no servidor do SonarQube, permitindo o monitoramento contínuo da qualidade do código e possibilitando a colaboração da equipa.

Portanto, usando ambas as ferramentas, os colaboradores obtêm uma abordagem mais completa para aprimorarem a qualidade de código em todas as etapas do ciclo de desenvolvimento.

Porém, o SonarLint, utiliza o *Quality Profile default* para cada linguagem, o que corresponde ao *Quality Profile* "Sonar Way". Já o SonarQube também apresenta este *default*, mas permite que os utilizadores modifiquem o conjunto das regras dos perfis, ou mesmo redefinam completamente perfis, personalizando conforme o que pretendem. Isto significa que, num momento, um projeto pode ser avaliado pelo SonarLint através de um *Quality Profile* diferente comparativamente ao *Quality Profile* utilizado pelo SonarQube, e portanto, os *Issues* encontrados por ambos podem ser diferentes. Por esta razão, o SonarLint apresenta um modo *connected* [59], em que o SonarLint faz *download* dos *Quality Profiles* que estão ativos numa instância SonarQube e respetivas regras. Ou seja, este modo *connected* sincroniza os *Quality Profiles* utilizados pelo SonarLint com os utilizados pelo SonarQube.

Ora, o facto de que com esta ferramenta a análise ser realizada localmente, ou seja, sem que o relatório da análise de código seja publicado num servidor remoto, e o facto de que a ferramenta apresenta um modo *connected*, fez com que considerasse que através desta conseguiria fazer a análise *commit* a *commit*.

5.2.3 O módulo `corq-scanner`

Nesta fase, explorei o código-fonte da ferramenta, nomeadamente a versão `8.17` da biblioteca "sonarlint-core"[60], que é a biblioteca que implementa a análise do SonarLint.

O código-fonte desta biblioteca não tem qualquer documentação, e portanto, foi um desafio compreendermos o funcionamento de uma análise com base neste código. A ausência de informação sobre os módulos, classes, métodos e até parâmetros dificultou o estudo, que foi extremamente demorado, pois explorei minuciosamente o código para compreender o seu funcionamento.

Como tal, quando comecei o estudo deste código, não sabia a partir de que método é que a análise tem origem, e portanto, comecei imediatamente à procura de um método "analyze".

Encontrei, de facto, um método `analyze` (figura 5.2), que devolve um objeto do tipo `Analysis- Results`,e que recebe como parâmetro um objeto `ConnectedAnalysisConfigu-`

`ration`, entre outros parâmetros. Este método faz parte de uma classe designada `ConnectedSonarLintEngine`. Achei que este método tinha interesse, pois o objetivo é utilizarmos a análise em modo *connected* do SonarLint, e por isso, foi a partir deste método que comecei o estudo.

```
public AnalysisResults analyze(ConnectedAnalysisConfiguration configuration, IssueListener issueListener,
  @Nullable ClientLogOutput logOutput, @Nullable ClientProgressMonitor monitor)
```

Figura 5.2: Assinatura do método `analyze`.

Primeiramente, comecei pelo parâmetro com tipo `ConnectedAnalysisConfiguration`. Ora, a classe `ConnectedAnalysisConfiguration` estende a classe `AbstractAnalysisConfiguration` - a classe `AbstractAnalysisConfiguration` define parâmetros essenciais para a construção de uma configuração de análise, sendo eles: a diretoria base de um projeto a ser analisado, os ficheiros deste projeto a serem analisados, propriedades extra e a `moduleKey`, a chave de um módulo a ser analisado neste projeto; já a classe `ConnectedAnalysisConfiguration`, por ser a configuração de uma análise em modo *connected*, acrescenta o parâmetro `projectKey`, que é a chave deste projeto numa instância SonarQube.

Para construir uma configuração de análise em modo *connected*, é exigido não uma instância File de um ficheiro a ser analisado, mas sim uma instância `ClientInputFile`, que é uma interface que declara vários métodos que manipulam os ficheiros de *input* de uma análise. Consequentemente, implementei a classe `ClientInputFileImpl`, que implementa a interface `ClientInputFile` e os seus métodos. Esta classe recebe por construtor um vetor de `bytes` do conteúdo do ficheiro, o seu caminho relativo, um `boolean` que indica se é um ficheiro de teste, e a codificação dos carateres do ficheiro.

Os restantes parâmetros que o método `analyze` requer são do tipo `IssueListener`, `ClientLogOutput` e `ClientProgressMonitor`. Para compreendê-los, examinei o método `analyze`. Verifiquei imediatamente que os parâmetros `ClientLogOutput` e `ClientProgressMonitor` são opcionais, ou seja, podem ser nulos. O parâmetro `IssueListener` é em verdade uma interface, que apresenta um único método `void handle(Issue var1)`. Por análise do código, entendi que esta é uma instância que ao longo de uma análise coleta os *Issues*, e portanto implementei a classe `SaveIssueListener`, que implementa a interface `IssueListener`, e que apresenta uma lista do tipo `Issue` como variável global. O método `handle`, adiciona o *Issue* que recebe a esta lista global de *Issues*. Implemento também um método `List<Issue> getIssues()` que retorna a lista, e um método `void clear()` que remove todos os elementos desta lista. Portanto, simplesmente crio uma instância `SaveIssueListener` para fazer a chamada ao `analyze`.

Nesta fase, sabendo já que valores dar aos parâmetros requeridos para a chamada do `analyze`, adicionei ao projeto CORQ um módulo que designei de `corq-scanner`. Sendo um projeto Maven, adicionei um ficheiro pom.xml, com a dependência Maven para a

biblioteca "sonarlint-core". Depois, criei uma classe `Main`, para fazer a chamada ao método `analyze`. Como projeto alvo de uma análise, defini um projeto exemplo "hello-world". Para a `ConnectedAnalysisConfiguration` defini a diretoria base do projeto, os ficheiros de *input*, a `projectKey` e a `moduleKey`, esta com valor nulo, pois não pretendo fazer uma análise por módulo - e isto ignorando por enquanto que existam propriedades extra. Também defini uma instância `IssueListener` e atribuí aos restantes parâmetros o valor nulo.

Ora, neste momento apenas faltou criarmos uma instância `ConnectedSonarLintEngineImpl`, para podermos chamar o método `analyze`. Todavia, uma instância `ConnectedSonarLintEngineImpl` recebe pelo seu construtor um parâmetro do tipo `ConnectedGlobalConfiguration`. Como esta configuração define variáveis que nesta fase eu não entendia o propósito, como uma `connectionId` e uma `storageRoot`, por exemplo, criei esta configuração sem definir nenhuma destas variáveis.

Com isto, pude fazer a chamada ao `analyze` em modo *debug*, a fim de ter perceção do que sucede numa execução deste método, que chamadas este faz, etc., e é isto que apresento com mais detalhe em seguida.

5.2.4 O método analyze

Como pude determinar por *debug*, existem configurações essenciais à análise que são definidas antes deste método `analyze`. Quando chamamos o construtor de uma instância `ConnectedSonarLintEngineImpl`, é chamado o método `loadAnalysisContext()`, que define e carrega o contexto de uma análise.

Este método `AnalysisContext loadAnalysisContext()` chama o método `PluginsLoadResult loadPlugins()`, que carrega os *plugins* necessários à análise. Ora, um *plugin* proporciona *analyzers* específicos para uma linguagem, para que o motor da análise compreenda e analise esta linguagem eficazmente, e identifique, com base nas regras definidas pelo *Quality Profile* desta linguagem, quais as regras que são violadas. Todavia, não existem apenas *plugins* para suporte das linguagens - por exemplo, há *plugins* para adição de regras - como o "SonarQube FindBugs Plugin-, e *plugins* para integração com ferramentas - como o "SonarQube Jenkins Plugin-, entre outros.

O método `PluginsLoadResult loadPlugins()` inicializa um mapa `pluginsToLoadByKey` do tipo `Map<String, Path>`, a ser preenchido. Primeiramente, são extraídos os *plugins* armazenados, que são os *plugins downloaded* de um servidor SonarQube ou SonarCloud. Estes são extraídos de uma instância do tipo `ServerConnection` e adicionados ao mapa.

O tipo `ServerConnection` é uma classe que gere a conexão e interação com um servidor SonarQube ou SonarCloud, e que também gere o armazenamento das informações dos projetos e *plugins* deste servidor. Apresenta uma variável do tipo `PluginsStorage`, que disponibiliza o método `Map<String, Path> getStoredPluginPathsByKey()`, que devolve os *plugins* armazenados.

Verifiquei que esta classe `PluginsStorage` apresenta um método `void store(ServerPlugin plugin, InputStream pluginBinary)`, que escreve o Java Archive (JAR) de um *plugin* para uma diretoria `String rootPath`. Ou seja, é este método que disponibiliza os *plugins* para que o método `Map<String, Path> getStoredPluginPathsByKey()` os devolva. Esta diretoria `String rootPath` é construída a partir da variável `storageRoot` de uma `ConnectedGlobalConfiguration`. A esta é concatenada a variável `connectionId` codificada através de um método `static String encodeForFs(String name)`, disponibilizado pela classe `ProjectStoragePaths`, e depois é novamente concatenada com a `String` "plugins". Ou seja, dado o nosso projeto exemplo "hello-world", sendo a diretoria base deste `C:\Projects\hello-world`, se definirmos a variável `storageRoot` como tendo o mesmo valor que a diretoria base e, se definirmos a variável `connectionId` como tendo o mesmo valor que a `projectKey`, então a diretoria onde são armazenados os *plugins* deste projeto é `C:\Projects\hello-world\68656c6c6f2d776f726c64\plugins`, como ilustrado pela figura 5.3. Isto significa que agora sabemos que valor podemos atribuir às variáveis `storageRoot` e `connectionId` da instância `ConnectedGlobalConfiguration` que damos como parâmetro ao construtor da instância `ConnectedSonarLintEngineImpl`.

Portanto, tendo compreendido como os *plugins* são armazenados e posteriormente carregados, para fazer uma análise tive que, antes de mais: fazer o *download* dos *plugins* do servidor remoto da Opensoft e configurar a instância `PluginsStorage`, para que os *plugins* sejam escritos para a diretoria a partir da qual são lidos aquando do carregamento destes, o que também é representado com a figura 5.3. Este foi o desafio seguinte da implementação.

Primeiramente, defini a diretoria em que os *plugins* são armazenados. Ou seja, defini uma instância `PluginsStorage`, que recebe como parâmetro um caminho que, identicamente a como expliquei antes, construo a partir da diretoria base do projeto concatenada com a codificação da `projectKey`, e depois com a `String` "plugins".

Ao explorar o código da biblioteca "sonarlint-core", descobri a classe `ServerApi` que disponibiliza vários métodos para acesso a diferentes APIs. Entre estes, encontramos o método `PluginsApi plugins()`, que retorna uma instância `PluginsApi`. Esta classe oferece uma API para interação com o sistema de *plugins* do servidor SonarQube. Possui o método `List<ServerPlugin> getInstalled()`, que uso para obter os *plugins* instalados numa instância SonarQube, sendo que cada plugin é devolvido como um objeto do tipo `ServerPlugin`. Destes, são apenas *downloaded* os *plugins* que disponibilizam *analyzers* de código para linguagens, e apenas das linguagens que são suportadas pelo SonarLint. Isto porque, o SonarLint suporta menos linguagens que o SonarQube - estas têm sido introduzidas gradualmente com novas versões da biblioteca.

Tendo a listagem dos *plugins* que pretendemos fazer *download*, para cada instância `ServerPlugin` desta lista, chamamos o método `void getPlugin(String key ServerApiHelper.IOConsumer<InputStream> pluginFileConsumer)`. Este método é também disponibilizado pela classe `PluginsApi`, e permite fazer o *download* de um *plugin*. Seguidamente, para o mesmo *plugin*, chamamos o método `void store(ServerPlugin plugin,`

`InputStream pluginBinary)`, disponibilizado pela classe `PluginsStorage`, para que o JAR seja escrito para a diretoria que antes definimos.

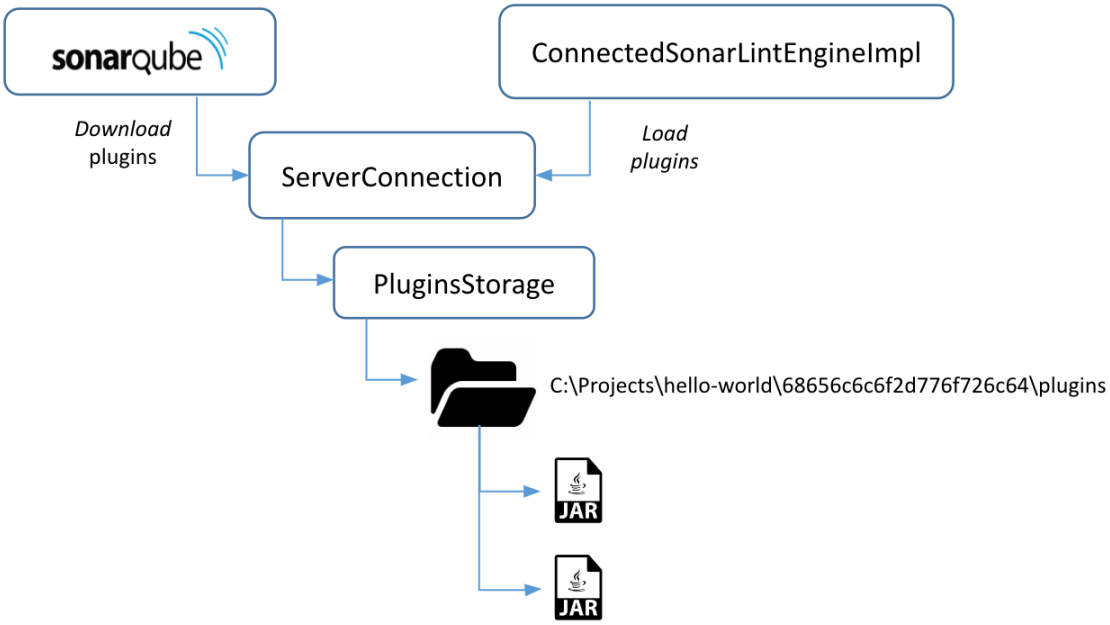

Figura 5.3: *Download* dos *plugins*.

Com isto, o método `PluginsLoadResult loadPlugins()` obtém os *plugins downloaded* da instância do SonarQube e adiciona-os ao mapa `pluginsToLoadByKey`. Tendo um `Set` dos caminhos de todos os *plugins* de `pluginsToLoadByKey`, o método constrói uma configuração do tipo `PluginsLoader.Configuration`, que depois é dada como parâmetro a `(new PluginsLoader()).load(config)`, que faz o carregamento dos *plugins*. É então definido o contexto da análise, que tem em conta os *plugins* definidos em `pluginsToLoadByKey`.

Só após esta inicialização do contexto da análise é que o `analyze` executa. Ora, este método inicializa um objeto do tipo `ActiveRulesContext`, ao qual são adicionadas regras a serem verificadas durante a análise. Estas regras são extraídas, uma vez mais, de uma instância do tipo `ServerConnection` e adicionadas a um mapa `ruleSetByLanguageKey`, sendo esta variável do tipo `Map<String, RuleSet>`. Uma instância `ServerConnection` possui uma variável do tipo `ProjectStorage`, que disponibiliza o método `AnalyzerConfiguration getAnalyzerConfiguration(String projectKey)`. Por sua vez, uma instância `AnalyzerConfiguration` apresenta uma variável `ruleSetByLanguageKey`, do tipo `Map<String, RuleSet>`. Portanto, o mapa `ruleSetByLanguageKey` é preenchido através da chamada `this.serverConnection.getAnalyzerConfiguration(projectKey).getRuleSetByLanguageKey()`.

Ora, logo em seguida é verificado se este mapa, `ruleSetByLanguageKey`, está vazio, e se sim, não são adicionadas regras ao objeto com tipo `ActiveRulesContext` - o que faz com que a análise termine e sejam retornados resultados vazios. O `analyze` inclusive faz log de uma mensagem que informa que a análise não se efetuou, pois não houve

sincronização com o servidor da instância SonarQube.

À semelhança do que acontece com os *plugins*, também a instância `ProjectStorage` possui um método `void store(String projectKey, AnalyzerConfiguration analyzerConfiguration)`, que escreve uma configuração com o tipo `AnalyzerConfiguration` para um ficheiro com extensão ".pb"(advém de *Protocol Buffer* ou *protobuf*), designado de "analyzer_config.pb". A diretoria deste ficheiro é também construída a partir da variável `storageRoot` de uma `ConnectedGlobalConfiguration`, concatenada com a variável `connectionId` codificada, depois concatenada com a `String` "projects", e por fim, novamente concatenada com a variável `connectionId` codificada. Ou seja, dado o nosso projeto exemplo "hello-world", a diretoria deste ficheiro será `C:\Projects\hello-world\68656c6c6f2d776f726c64\projects\68656c6c6f2d776f726c64`, tal como ilustrado pela figura 5.4.

Posto isto, quando experimentei executar a `Main` que defini para a análise do projeto "hello-world", obtive logo um erro por não existir um ficheiro "analyzer_config.pb". Daí que o desafio que se seguiu foi entender o que é uma `AnalyzerConfiguration`, para depois escrevê-la para a diretoria que referi acima.

Uma instância `AnalyzerConfiguration` recebe por construtor não só o parâmetro `Map<String, RuleSet> ruleSetByLanguageKey`, como também um parâmetro `Settings settings`. Ora, sendo uma instância `Settings` constituída por um mapa do tipo `Map<String, String>`, recorri mais uma vez à classe `ServerApi`, que disponibiliza, através do método `SettingsApi settings()`, uma instância `SettingsApi`. Esta, por sua vez, disponibiliza o método `Map<String, String> getProjectSettings(String projectKey)`, que faz um pedido à API do SonarQube para obter os `settings` específicos de um projeto, o que é representado com a figura 5.5.

Quanto ao parâmetro `Map<String, RuleSet> ruleSetByLanguageKey`, como um *Quality Profile* é específico de uma linguagem e estes são compostos por regras, faço download dos *Quality Profiles* ativos para o projeto específico, e extraio para cada *Quality Profile* as regras ativas. Para tal, recorro também à instância `ServerApi`, para obter uma instância `QualityProfileApi`, que disponibiliza o método `List<QualityProfile> getQualityProfiles(String projectKey)`. Este método faz um pedido à API do servidor SonarQube para obter a listagem dos quality profiles ativos de um projeto, especificado pela `projectKey`. Seguidamente, tendo cada objeto `QualityProfile` um atributo chave, que representa a chave única de um *Quality Profile*, extraio as regras ativas. Ora, uma instância `ServerApi` também oferece uma instância `RulesApi`, que disponibiliza uma API para interação com o sistema de regras do servidor SonarQube. Contudo, o método que efetua uma chamada ao servidor para obter as regras ativas de um *Quality Profile* requer, além da chave do *Quality Profile*, um outro parâmetro com o tipo `ProgressMonitor` e, como não sei como fazer a inicialização de um objeto com este tipo, não pude recorrer a esta API.

Por esta razão, implementei um método que faz o pedido das regras de um *Quality Profile*, especificado pela sua chave, ao servidor SonarQube. O método que implementei tem

a assinatura `RuleSet extractRulesPerProfile(String projectKey)` e, através de um pedido Hypertext Transfer Protocol (HTTP), obtém uma resposta em formato JavaScript Object Notation (JSON), a partir da qual extraio os atributos caraterísticos de uma regra. Com estes atributos, cada regra é inicializada como um objeto do tipo `ServerActiveRule`. Este tipo é implementado pela biblioteca "sonarlint-core"e representa uma regra ativa do servidor SonarQube. Após extraídas as regras, o método inicializa com estas um objeto do tipo `RuleSet`, também implementado pelo "sonarlint-core", que representa o conjunto das regras.

Portanto, para cada valor `String` de uma linguagem de um *Quality Profile*, é obtido o correspondente `RuleSet` das regras ativas. Ou seja, obtemos o mapa `Map<String, RuleSet> ruleSetByLanguageKey` que nos faltava para construirmos uma configuração `AnalyzerConfiguration`, o que é também representado pela figura 5.5. Só então, é inicializada uma instância `ProjectStorage` com a diretoria bem formada, diretoria esta mencionada anteriormente, e chamado o método `void store(String projectKey, AnalyzerConfiguration analyzerConfiguration)` desta instância, para que a configuração seja escrita para um ficheiro "analyzer_config.pb", como apresentado pela figura 5.4.

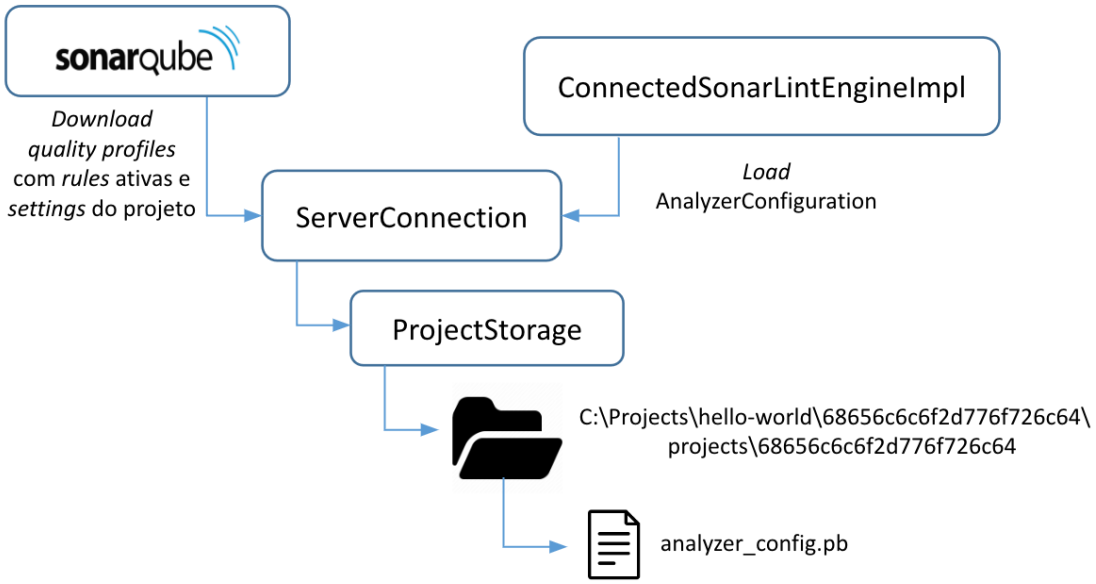

Figura 5.4: *Download* da `AnalyzerConfiguration`.

Todavia, a instância SonarQube da Opensoft não inclui todos os projetos da empresa. Mas, o SonarQube apresenta *Quality Profiles* e mesmo *settings default*. Posto isto, se um projeto não estiver incluído, o pedido feito ao servidor SonarQube para obter os *Quality Profiles* falha. Para este caso, modifiquei o método para que este repita o pedido ao servidor, agora fazendo um pedido dos *Quality Profiles default*. O mesmo apliquei ao método que obtém os *settings* de um projeto, que neste caso obtém os *settings default* para um projeto.

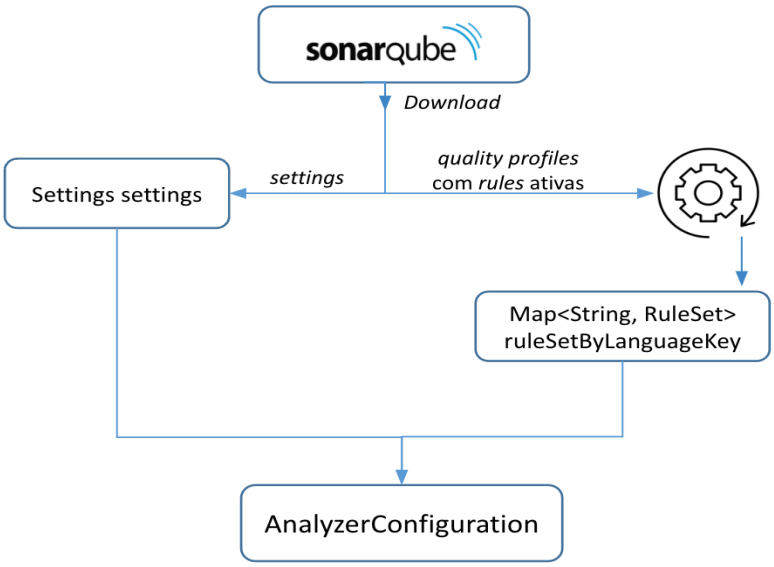

Figura 5.5: Construção da `AnalyzerConfiguration`.

Com isto, o `analyze` obtém as regras contidas nesta `AnalyzerConfiguration` que configurámos antes, e adiciona-as ao contexto `ActiveRulesContext`.

Posto isto, o `analyze` cria uma configuração para a análise, com o tipo `AnalysisConfiguration`, adicionando a esta: os ficheiros de *input*, a `projectKey`, os `settings` definidos pela `AnalyzerConfiguration`, as regras ativas definidas pelo `ActiveRulesContext`, a diretoria base e propriedades extra. Estas propriedades extra são as propriedades definidas por um ficheiro designado "sonar-project.properties", caso exista um com esta designação para a diretoria base. Isto porque não é obrigatório um projeto ter este ficheiro, pois como já foi antes dito, o projeto pode não estar incluído na instância SonarQube. Este ficheiro contém propriedades para a análise do SonarQube.

Ora, algo que antes não referi, é que com a inicialização do construtor de uma instância `ConnectedSonarLintEngineImpl`, é também inicializada uma instância `AnalysisEngine`, representando esta o motor da análise. Esta classe possui uma variável `BlockingQueue <AsyncCommand<?» commandQueue`, que permite que os comandos de análise a serem executados sejam armazenados pela ordem de chegada e assegura a execução assíncrona destes. Existe um método `void start()`, chamado logo pelo construtor, que inicia uma *Thread*, designada "sonarlint-analysis-engine", que executa continuamente o método `void executeQueuedCommands()`, como ilustrado pela figura 5.6. Este método executa os comandos armazenados pela `commandQueue`. Com efeito, existe um método `<T> CompletableFuture<T> post(Command<T> command, ProgressMonitor progressMonitor)` que adiciona um comando de análise à `commandQueue`. Ademais, existe ainda um método `void finishGracefully()`, que sinaliza a interrupção desta *Thread*, mas executa os comandos pendentes, e um método `void stop()`, que interrompe a *Thread* e cancela todos os comandos pendentes.

Aquando uma chamada do `analyze`, é inicializada uma instância com o tipo `Analyze-`

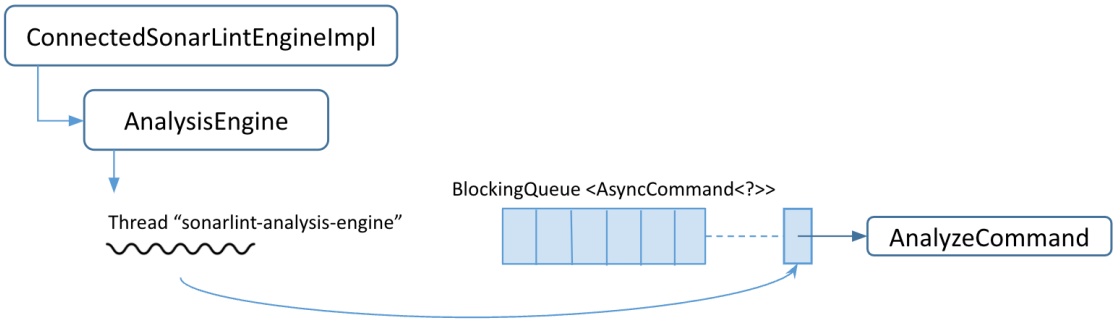

Figura 5.6: Execução dos comandos de análise.

`Command`, que recebe por construtor a configuração da análise, a `AnalysisConfiguration`, assim como a instância `SaveIssueListener` que regista as violações das regras encontradas durante a análise. É então chamado o método `AnalysisResults postAnalysisCommandAndGetResult (AnalyzeCommand analyzeCommand, @Nullable ClientProgressMonitor monitor)` que adiciona este comando de análise à `commandQueue` da instância `AnalysisEngine` e obtém o resultado deste comando, depois de este ser executado.

Por fim, como a instância `SaveIssueListener` dada como parâmetro à instância `Connected SonarLintEngineImpl` é populada durante a análise com os *Issues* descobertos, quando esta análise termina, apenas temos de chamar o método `List<Issue> getIssues()` para obter os *Issues*.

Há mais detalhe nesta implementação, mas é este conhecimento que descrevi nesta secção sobre o funcionamento da análise do SonarLint que foi o fundamental para utilizarmos o `analyze` para fazer a análise de *commits*. Os métodos que configuram a análise de um projeto e que desencadeiam uma análise, chamando o método `analyze` do SonarLint, estão desenvolvidos numa classe que designei de `Analysis`.

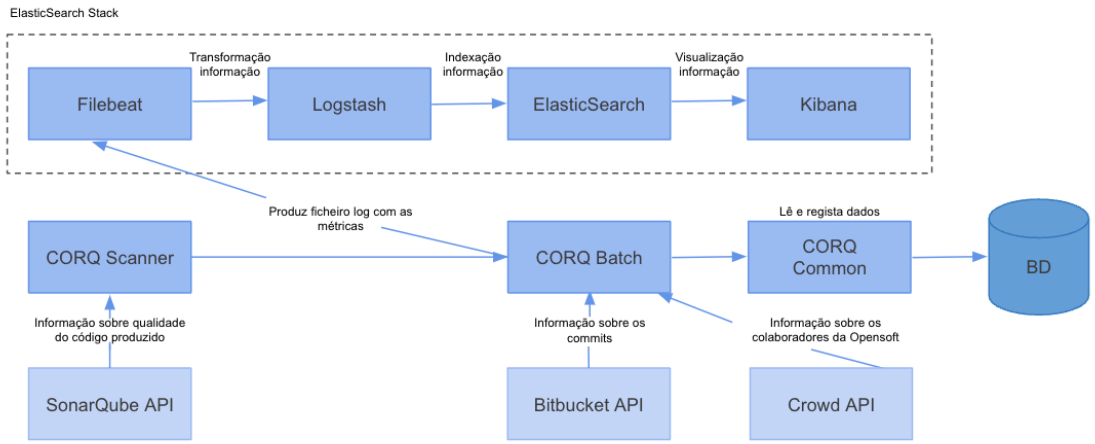

Figura 5.7: Arquitetura atual da ferramenta CORQ.

5.3 Modelo de Dados

O modelo de dados da ferramenta CORQ, apresentado pela figura 5.8, sofreu alterações, pois apenas com a aplicação da primeira versão da ferramenta é que se constataram casos que não se tinham previsto, e que fazem com que o seu funcionamento seja pouco eficiente ou mesmo falhe.

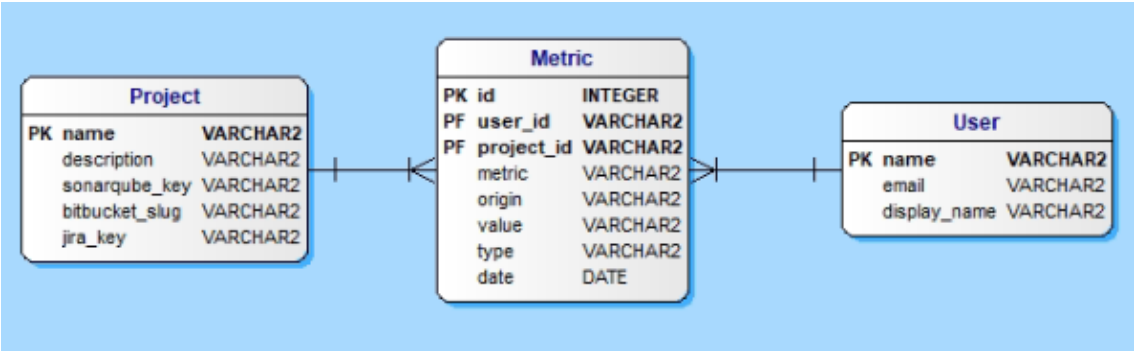

Figura 5.8: O modelo de dados que existia previamente.

Seguidamente, apresento os casos em que este modelo é pouco eficiente ou falha e as modificações introduzidas, que originaram o modelo que figura em 5.9:

1. Caso a ferramenta falhe durante o processamento dos *commits* para cálculo de métricas, é necessário fazer o reprocessamento dos *commits*, pois não há nenhuma informação registada em base de dados sobre os *commits* que foram ou não processados.

   Ou seja, o modelo de dados tem de ser quer de análise de métricas, quer operacional.

   Como tal, introduzimos a entidade `Commit`, que se relaciona com a entidade `Project` através de uma relação um-para-muitos, pois um projeto apresenta vários *commits*, mas um dado *commit* é referente a um único projeto. Um *commit* é descrito por: `id`, o identificador de um *commit* (chave primária), `project_id`, o identificador do projeto a que se refere (sendo este uma chave estrangeira), `hash` e `timestamp` da criação deste *commit*. Nesta entidade, os atributos `project_id` e `hash` compõem a chave natural, pois o *hash* de um *commit* só é assegurado ser único dentro de um repositório de um projeto, e portanto, o conjunto destes atributos deve ser único. Isto traduz-se numa restrição imposta sobre a tabela, a que todas as entradas desta entidade devem respeitar.

   De momento, como existem dois processamentos independentes, cada um referente a uma métrica, e para que a solução seja extensível, optámos por introduzir uma entidade que representa o *commit* processado, designada de `Commit_Processed`. Sendo que um *commit* é processado n vezes, dependendo dos n processos, e que existe um processamento por *commit*, a entidade `Commit_Processed` apresenta uma relação de um-para-muitos com a entidade `Commit`. Por conseguinte, a entidade

`Commit_Processed` é caraterizada por: `commit_id`, o identificador do *commit* a que se refere (chave primária e chave estrangeira), `process` (também chave primária), que identifica o processo, e `processed_date`, que é a data em que o *commit* é processado.

Ademais, como um *commit* é constituído por um ou mais ficheiros e é ao nível destes que obtemos os valores intermediários das métricas, que são depois agrupados por dia, colaborador e projeto, introduzimos ainda a entidade `File`, que está também relacionada por um-para-muitos com a entidade `Commit`, sendo descrita por: `id` (chave primária), `commit_id`, o identificador do *commit* a que se refere (chave estrangeira), `path`, o caminho deste ficheiro (sendo que este acrescenta à diretoria base das configurações de um projeto) e `name`, o nome do ficheiro. Esta entidade tem como chave natural o conjunto dos atributos `commit_id`, `path` e `name`, pois um *commit* não contém ficheiros duplicados, i.e. com o mesmo caminho e o mesmo nome. Portanto, esta é a restrição imposta sobre a tabela, a que as entradas desta entidade têm que respeitar.

Quanto aos valores intermediários das métricas recolhidos por ficheiro, introduzimos para registo destes a entidade `File_Metric`, garantindo a extensibilidade a outras métricas, estando esta relacionada com a entidade `File` por um-para-muitos, pois cada ficheiro origina vários valores de métricas (linhas adicionadas, linhas removidas, *Issues* com severidade BLOCKER, etc.), e cada métrica é um valor que diz respeito a um ficheiro, sendo esta entidade caraterizada por: `file_id`, o identificador do ficheiro a que se refere (chave primária e chave estrangeira), `metric`, a métrica que foi coletada para este ficheiro (também chave primária, tem como valor uma `String` com "LINES_ADDED", "LINES_REMOVED", "ISSUES_BLOCKER", "ISSUES_CRITICAL", "ISSUES_MAJOR", "ISSUES_MINOR"ou "ISSUES_INFO"), `type`, o tipo desta métrica (que tem como valor uma `String` com "boolean", "Integer", "String"ou "float"), que permite-nos descodificar o atributo `value`, que contém o valor da métrica. Através desta entidade são registados desde logo os valores intermediários para o cálculo final, pelo que esta entidade também contribui para que o modelo de dados seja operacional.

2. Um colaborador pode, aquando o momento em que faz *commit*, dar uma identificação (sendo que esta é composta por nome e e-mail) diferente da dos *commits* anteriores que fez, o que não é rejeitado pela ferramenta de controlo de versões (ou aquando a configuração da ferramenta de controlo de versões um colaborador não oferece identificação e esta atribui-lhe uma auto-gerada). Portanto, a identificação de um colaborador nestas ferramentas não é necessariamente a mesma que a do Crowd. Como resultado, como a esta representação de autor não corresponde nenhum colaborador, a ferramenta falha.

   Como um colaborador pode ter várias formas de se identificar nestas ferramentas, concluímos que a identificação dos colaboradores deve ser uma entidade própria,

que designámos de `Author`. Esta está relacionada com a entidade `Collaborator` através de uma relação muitos-para-muitos, pois um colaborador pode ter várias representações de autor, e uma dada representação pertence a um ou mais colaboradores (por exemplo, eu partilho o *username* de um colaborador que já trabalhou com a Opensoft). Como tabela intermediária desta relação, introduzimos a tabela `Collaborator_Author`, que decompõe a relação muitos-para-muitos em uma relação de um-para-muitos entre `Collaborator` e `Collaborator_Author` e outra relação um-para-muitos entre `Collaborator_Author` e `Author`.

Posto isto, a entidade `Collaborator` é caraterizada por `id` (chave primária), `name` e `status`, que distinge se o colaborador está ou não ativo, enquanto a entidade `Author` é caraterizada por `id` (chave primária), e atributos que identificam um autor, i.e. `email` e `display_name`. Ora, esta tabela apresenta uma chave natural composta pelos atributos `email` e `display_name`, pois é através destes que distinguimos diferentes autores, e por isso, esta é uma restrição imposta sobre a tabela.

Já a tabela `Collaborator_Author` é composta pelos atributos: `collaborator_id`, o identificador do colaborador a que se refere (chave primária e chave estrangeira); `author_id`, o identificador do autor a que se refere (também chave primária e chave estrangeira); `start_date`, que representa a data da criação da associação entre colaborador e autor (também chave primária); `type`, que representa se o autor é parte integrante da Opensoft, ou cliente, ou ainda indeterminado, caso seja impossível inferir (por exemplo, se o `email` e `display_name` forem uma `String` de carateres aleatórios); e, por fim, `end_Date`, que representa a data a partir da qual o colaborador não usa mais esta representação de autor. As linhas desta tabela irão ser introduzidas manualmente, uma vez que, muitas vezes, é impossível determinarmos *a priori* a que colaborador pertence um autor.

Ademais, a entidade `Author` está ainda relacionada com a entidade `Commit`, pois um *commit* tem um ou mais autores (um *commiter* e zero ou mais *participants*), e um determinado autor produz um ou mais *commits*, resultando disto uma relação de muitos-para-muitos. A tabela intermediária desta relação foi designada de `Commit_Author` e apresenta os atributos: `author_id`, o identificador do autor a que se refere (chave primária e chave estrangeira); `commit_id`, o identificador do *commit* a que se refere (também chave primária e chave estrangeira); e um atributo `boolean` `is_coauthor`, que distingue se o autor é *participant* ou não. Portanto, esta tabela apresenta a associação entre *commits* e seus autores.

É importante referirmos que podem existir *commits* cujo autor não tem associação com um colaborador, pois existem *commits* que são realizados por clientes entre outros autores externos à Opensoft. Neste modelo de dados, como esses autores não têm associação com nenhum colaborador, os valores das métricas dos seus *commits* não têm influência para o cálculo das métricas diárias, que devem ser apenas sobre os colaboradores da Opensoft.

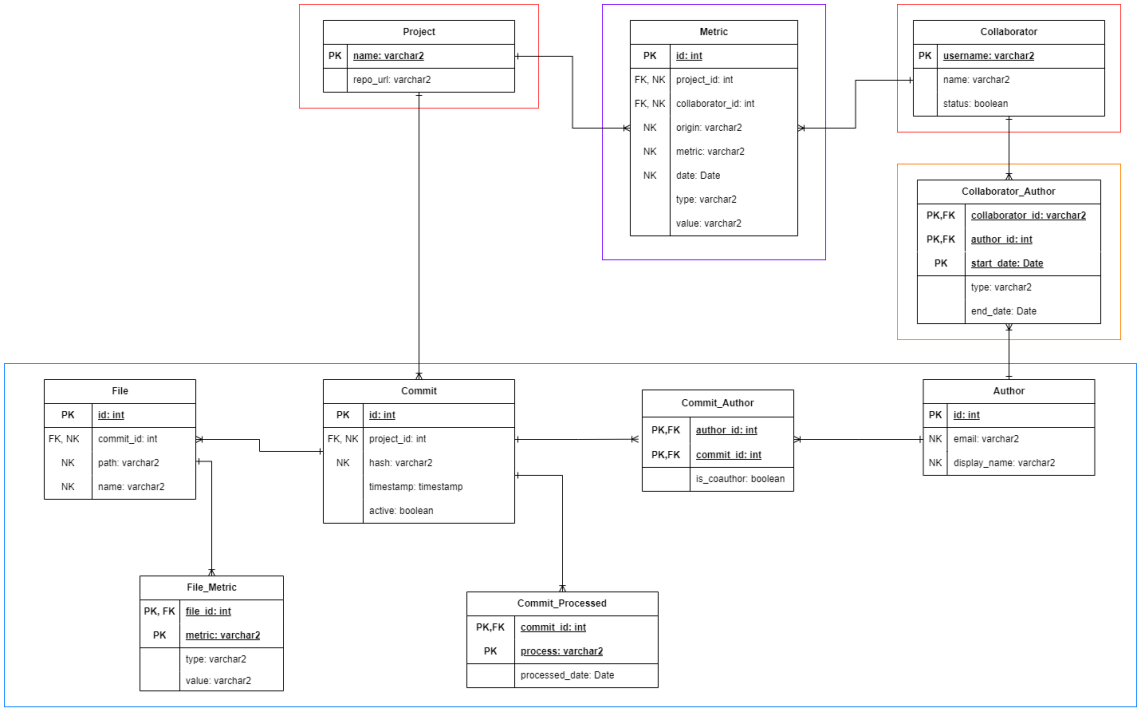

Figura 5.9: O modelo de dados atual.

Durante o design deste modelo, optámos por manter o mínimo de entidades e colunas essenciais para o funcionamento da ferramenta e, a extensibilidade do modelo. Neste sentido, modificámos a entidade `Project`, removendo os atributos: `jira_key`, pois de momento não é necessário para os métricaes que estão a ser calculados; `sonarqube_key`, que faz parte da configuração do pom.xml dos projetos, pelo que é desnecessário mantermos esta informação em base de dados; e alterámos a designação do atributo `bitbucket_slug` para `repo_url`, uma vez que um repositório pode não ser apenas do Bitbucket, como do Azure DevOps Server [5] (caso seja um projeto cujo cliente é a AT), o que é distinguível pelo próprio URL do repositório.

Ora, a entidade `Metric` não sofreu nenhuma modificação, mas adicionei a esta uma chave natural, composta pelos atributos `project_id`, `collaborator_id`, `origin`, `metric`, `date`, pois é através destes que distinguimos diferentes entradas desta tabela, e portanto, esta é uma restrição imposta sobre a tabela.

Este modelo de dados tem uma componente operacional (a azul), composta pelas entidades `Commit`, `Commit_Processed`, `File`, `File_Metric`, `Author` e `Commit_Author`, cujas linhas das tabelas são preenchidas apenas com a informação existente num *commit*. Já a componente de análise é constituída pelas entidades `Project`, `Metric` e `Collaborator`, que possuem as métricas agrupadas por dia, podendo ainda ser agrupadas por projeto e/ou colaborador. Podemos ver as tabelas `Project` e `Collaborator` como as dimensões (a vermelho) e a tabela `Metric` como a tabela de factos (a roxo), como é a tipologia comum dos modelos dimensionais. A tabela `Collaborator_Author` por ser preenchida manualmente e por ser a relação entre a componente operacional e a componente de

análise está representada por uma cor diferente das restantes componentes (a laranja).

Como podemos agora observar pelas diferenças entre ambas as imagens, o primeiro modelo carecia de toda a componente operacional, que garante o correto funcionamento da ferramenta CORQ.

Isto exigiu que modificasse o módulo `corq-common`, adicionando ou alterando entidades e respetivos repositórios.

5.4 *Batch Jobs*

Com as modificações efetuadas ao modelo de dados, os *Jobs* sofreram também ligeiras modificações. A primeira versão da ferramenta CORQ possuía os seguintes *Jobs*:

`Clone Repositories`: clona os repositórios do Bitbucket da Opensoft para a máquina onde o *Job* é executado, i.e. para uma pasta do explorador de ficheiros, cuja diretoria é especificada num ficheiros de propriedades. Este *Job* requer que a tabela `Project` da base de dados já esteja populada com os projetos que se pretende clonar;

`Populate Database`: popula a tabela `Collaborator` da base de dados com as informações dos colaboradores da Opensoft e também a tabela `Project` com os projetos do Bitbucket da Opensoft;

`Local Projects`: popula a tabela `Project` da base de dados com os projetos que existem numa diretoria especificada nas propriedades da máquina que executa o *Job*;

`Generate Metrics`: gera métricas sobre linhas de código adicionadas, removidas e alteradas por um colaborador, inserido num projeto, e num determinado dia. Este *Job* está dividido em 3 diferentes *Tasklets* - são elas `Get Commits`, `Generate Metrics` e `Write Metrics`. Sucintamente, a *Tasklet* `Get Commits` obtém para cada repositório de um projeto os *commits* até ao dia anterior, sendo a *Tasklet* `Generate Metrics` que os processa, obtendo os valores das métricas, que são depois escritas para a base de dados pela *Tasklet* `Write Metrics`;

`Index Reset`: extrai todas as métricas registadas pela tabela `Metric` da base de dados e escreve-as para um ficheiro log, uma linha para cada métrica obtida. Posteriormente, este ficheiro log é transmitido à ELK Stack;

`Send Email`: envia um e-mail com métricas de um intervalo de tempo especificado.

Primeiramente, o *Job* `Generate Metrics` foi implementado novamente. Ora, a primeira razão é logo por o modelo de dados ter sido modificado. Isto porque, a nova versão do modelo de dados adicionou uma componente operacional, que exige que se registem os *commits* de um projeto, a informação de quando e por quem, os seus ficheiros, quando e por qual processo gerador de métricas foram processados, e tudo isto introduziu mais complexidade ao *Job*.

Por outro lado, este *Job* é geral - o pressuposto era este *Job* gerar métricas tanto de produtividade como de qualidade de código. Porém, sendo o objetivo destas métricas tão diferente, optámos por manter o seu cálculo independente e, portanto, implementámos um *Job* dedicado a gerar métricas sobre produtividade e outro *Job* dedicado a gerar métricas sobre qualidade de código.

Ademais, optámos também por mantermos independentes as *Tasklets* deste *Job*, pois cada uma apresenta um objetivo também diferente - uma lê *commits*, outra faz o processamento destes, outras escreve os resultados deste processamento para a base de dados. Portanto, ao todo obtemos 5 *Jobs*: `Read Commits` que lê os *commits*; `Generate Issues Metric` que processa os *commits* quanto a qualidade de código; `Generate Lines Metric` que processa os *commits* quanto a produtividade; `Write Issues Metric` que escreve métricas sobre qualidade de código para a base de dados; e, por fim, `Write Lines Metric` que escreve métricas sobre produtividade para a base de dados.

Esta arquitetura obtém-nos mais flexibilidade aquando a execução, pois ainda que a ordem de execução lógica dos *Jobs* seja `read-generate-write`, como os *Jobs* são independentes, não existe nenhuma ordem de execução a ser seguida.

Também, tendo a carga de trabalho dividida entre vários *Jobs* comparativamente a esta ser responsabilidade de um único *Job*, beneficiamos a qualidade de código da ferramenta. Isto porque, sendo o escopo mais específico, é mais fácil que, em caso de erro, o administrador da ferramenta identifique e corrija o erro e, simultaneamente, que o código seja mantido.

Para mais, esta arquitetura permite que os *Jobs* executem paralelamente, possibilitando um melhor aproveitamento dos recursos e uma execução mais otimizada, e confere extensibilidade à ferramenta, sendo mais fácil adicionarmos *Jobs* para outras métricas.

Nas secções 5.4.1 - 5.4.5 que se seguem, introduzo com mais detalhe o trabalho destes *Jobs*.

Seguidamente, com as secções 5.4.6 - 5.4.9, apresento modificações feitas aos restantes *Jobs* listados, nomeadamente o *Job* `Clone Repositories` o *Job* `Populate Database`. Os *Jobs* `Local Projects` e `Index Reset` não sofreram modificações. Ademais, criei o *Job* `Write ElasticSearch Log`.

Por fim, a figura 5.10 apresenta o modelo de dados com as respetivas entidades agrupadas por *Job*, conforme o *Job* que as manipula, para facilitar a compreensão do funcionamento destes. Relembro que a tabela `Collaborator_Author` tem que ser preenchida manualmente.

5.4.1 *Job* `Read Commits`

O *Job* `Read Commits` regista para a base de dados os commits dos projetos da Opensoft. Inicia a execução a partir de uma diretoria local especificada, que contém os repositórios Git dos vários projetos clonados. Portanto, nesta diretoria existe uma pasta para cada projeto, sendo a designação da pasta o nome do projeto.

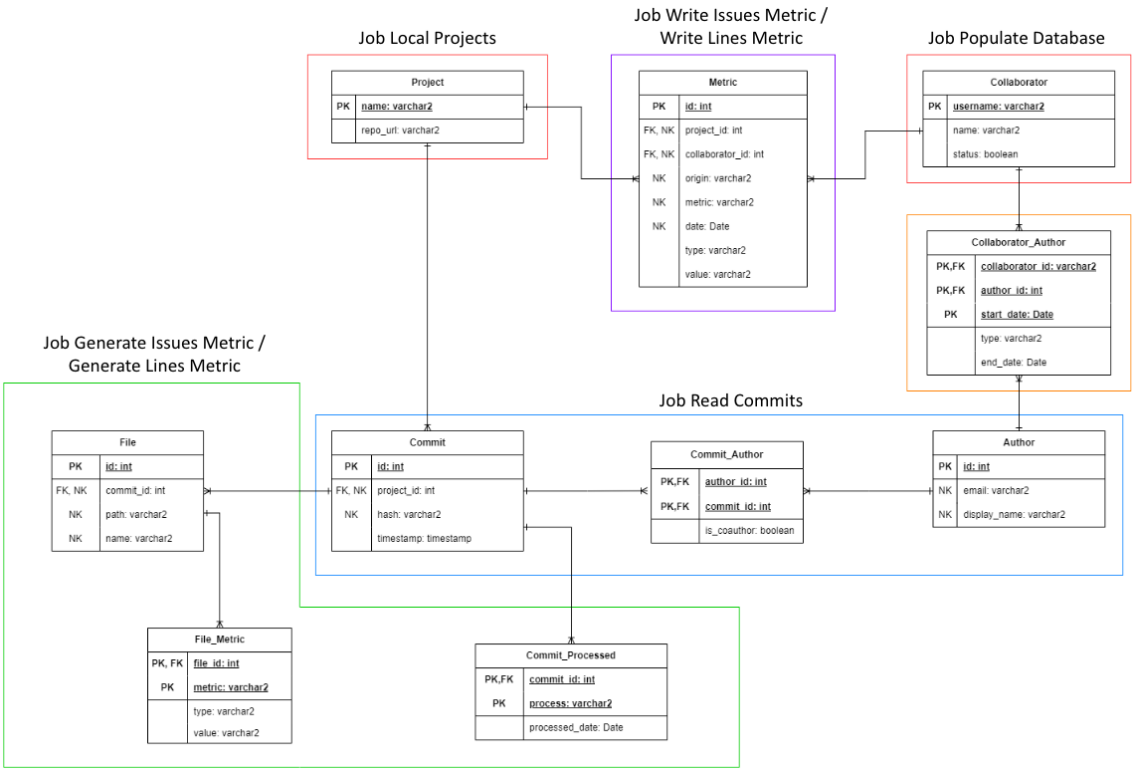

Figura 5.10: O modelo de dados organizado por processo.

Para cada projeto, o *Job* executa o seguinte: solicita, através de uma *query*, os *commits* deste projeto que existem registados em base de dados, e obtém como resultado uma lista com o tipo `Commit`, que designamos de `dbCommits`; solicita ao repositório Git local do projeto os *commits* que este detém, obtendo como resultado uma lista com o tipo `RevCommit`, que nomeamos de `repoCommits`. Inverte esta lista, para que o *commit* mais recente seja o último da lista.

Neste momento, o *Job* verifica se a lista `dbCommits` se encontra vazia, o que se traduz em 2 cenários diferentes possíveis:

- Se a lista `dbCommits` está vazia, então significa que esta é a primeira execução do *Job*, e portanto este tem de registar em base de dados todos os *commits* da lista `repoCommits`;

- Se a lista `dbCommits` não está vazia, então significa que podem existir *commits* do repositório Git que não estão armazenados em base de dados. Neste caso, comparamos ambas as listas através do atributo `hash` de um *commit*, a fim de obtermos os *commits* da lista `repoCommits` que não estão registados em base de dados e registá-los.

Com uma instância `RevCommit` da biblioteca JGit, temos o e-mail e nome do *committer*, i.e. quem realizou o *commit*, e a data em que o fez. Com estas informações, chamamos o método do serviço do CORQ que cria uma instância `Commit` e a adiciona à tabela `Commit`

da base de dados. Seguidamente, também recorrendo ao serviço do CORQ, verificamos através de uma *query* à base de dados se existe já registada alguma instância `Author` com o mesmo e-mail e nome. Caso a *query* retorne `null`, criamos uma instância `Author`, que é registada em base de dados. Por fim, adicionamos a associação entre este `Commit` e este `Author`, criando uma instância `CommitAuthor` e adicionando-a à base de dados.

Posto isto, com o modelo de dados otimizado, como os *commits* são registados em base de dados, evitamos a leitura redundante dos *commits* em execuções sucessivas. Esta modificação é de extrema importância, pois representa um ganho significativo de eficiência, reduzindo o tempo e recursos necessários para a execução deste *Job*, e melhorando consideravelmente o desempenho da ferramenta.

5.4.2 *Job* `Generate Issues Metric`

O *Job* `Generate Issues Metric` faz o processamento dos *commits* registados em base de dados, gerando métricas sobre a qualidade de código de cada ficheiro de um *commit*. Parte da diretoria local que contém os repositórios Git dos vários projetos clonados e, para cada projeto, o *Job* executa a implementação que descrevemos nesta secção.

Solicita, através de uma *query*, os *commits* deste projeto que existem registados em base de dados, cuja data seja anterior à data atual, e que não têm correspondência com uma instância `CommitProcessed` com o processo gerador de métricas sobre qualidade de código, que é designado de `ISSUES_METRIC_PROCESS`. Esta *query* obtém como resultado uma lista com o tipo `Commit`, que nomeamos de `unprocessed`. Caso esta seja retornada como vazia, significa que o repositório está sincronizado, e portanto, o *Job* prossegue para outro projeto. Caso contrário, significa que os *commits* retornados têm de ser submetidos a análise.

Para tal, inicializamos uma instância `Analysis`, do módulo `corq-scanner`, cuja implementação foi apresentada nas secções 5.2.3 e 5.2.4 com detalhe. Esta instância representa o motor da análise. É importante notarmos que esta implementação inicializa uma instância `Analysis` por projeto, uma vez que as configurações da análise dependem do projeto, mas esta instância é utilizada para análise de todos os *commits* de um mesmo projeto.

Para configurarmos o motor da análise, temos de obter a `projectKey` e a `baseDir` do projeto. Ora, os projetos da Opensoft recorrem maioritariamente ao Maven e, comummente, a `projectKey` dos projetos da Opensoft é o resultado da concatenação das variáveis `groupId` e `artifactId` do ficheiro pom.xml. Já a `baseDir` é equivalente à diretoria local que contém os repositórios Git dos vários projetos clonados, incluindo este, concatenada com o nome do projeto.

Seguidamente, chamamos o método `void setConfigurations()`, que resolve outras configurações essenciais à análise, como por exemplo, o formar a partir da `baseDir` a diretoria para a qual os *plugins* são *downloaded*, e a diretoria em que é escrito o ficheiro "analyzer_config.pb", que contém as regras por linguagem a serem aplicadas durante a análise.

Como referimos antes com o capítulo 3, secção 3.9, a biblioteca JGit não disponibiliza nenhum método que retorne diretamente apenas os ficheiros modificados de um *commit*. Para tal, temos de fazer uma comparação entre as árvores de ficheiros do *commit* especificado e do *commit* pai. Posto isto, para cada *commit* da lista `unprocessed`, obtemos a lista dos caminhos dos ficheiros modificados através do método implementado `List<String> getChangedFilesFromCommit(Repository repository, DiffFormatter df, RevWalk revWalk, RevCommit commit)`. Este método verifica quando o *commit* recebido por parâmetro é o *commit* pai do repositório, ou seja, quando não lhe antecedem *commits*, e portanto, neste caso, todos os ficheiros são considerados ficheiros modificados.

À lista de caminhos dos ficheiros modificados de um *commit*, que designamos `filesPaths`, aplicamos o método implementado `List<ClientInputFile> getChangedFilesContent(Repository repository, String project, RevCommit commit, List<String> filesPaths)`. Este método percorre a lista dos caminhos e, para cada um, chama o método implementado `byte[] getFile Content(Repository repository, String project, RevCommit commit, String filePath)`, que extraí o conteúdo do ficheiro com caminho correspondente como um vetor de `bytes`. Posto isto, através do conteúdo do ficheiro e do seu caminho, o método constrói uma instância `ClientInputFileImpl`, que adiciona a uma lista `List<ClientInputFile> inputFiles`, que é retornada ao fim, depois de termos percorrido a lista `filesPaths`.

Finalmente, chamamos o método `AnalysisResults analyze(List<ClientInputFile> inputFiles)`, que executa uma análise. Este método é disponibilizado pela instância `Analysis` por nós implementada. O resultado desta análise, com o tipo `AnalysisResults`, disponibiliza, por sua vez, o método `Map<String, <EnumMap<IssueSeverity, Integer> getResult()` devolve, por cada ficheiro do *commit*, a quantidade de *Issues* encontrados por severidade durante a análise.

Tendo os resultados, inicializamos uma instância `File` por ficheiro, caso o ficheiro não exista já registado em base de dados, o que verificamos através de uma *query*; para cada ficheiro, e por cada severidade, registamos uma instância `FileMetric` que regista a quantidade dos *Issues* encontrados com determinada severidade. Ou seja, cada ficheiro possui as seguintes métricas coletadas: `BLOCKER_ISSUES`, `CRITICAL_ISSUES`, `MAJOR_ISSUES`, `MINOR_ISSUES`, e `INFO_ISSUES`. Ademais, registamos também para cada ficheiro a métrica `TOTAL_LINES`, que diz respeito à quantidade total de linhas que o ficheiro detém.

Por fim, adicionamos uma instância à tabela `CommitProcessed`, registando a informação de que o *commit* está processado, pelo processo `ISSUES_METRIC_PROCESS`, na data atual.

5.4.3 *Job* `Generate Lines Metric`

O *Job* `Generate Lines Metric` faz o processamento dos *commits* registados em base de dados, gerando métricas sobre produtividade em desenvolvimento de código, contabilizando as linhas de código adicionadas, removidas e alteradas em cada ficheiro de um *commit*. Parte da diretoria local que contém os repositórios Git dos vários projetos clonados,

e executa projeto a projeto.

Este *Job* tem semelhanças, quanto à arquitetura, com o *Job* `Generate Issues Metric`, pois possui: uma fase de leitura, em que faz uma *query* à base de dados para obter os *commits* cuja data seja anterior à data atual, que não tenham sido processados pelo processo gerador de métricas sobre produtividade em desenvolvimento de código, sendo este designado `LINES_METRIC_PROCESS`; uma fase de processamento, em que extrai métricas para cada ficheiro de um *commit* retornado pela *query*; e, por fim, uma fase de escrita, que regista em base de dados que o *commit* foi processado pelo processo `LINES_METRIC_PROCESS`, na data atual, através de uma instância `CommitProcessed`, e que regista também os ficheiros e respetivas métricas.

Porém, o que diferencia ambos os *Jobs* é a fase de processamento - este *Job* executa para cada *commit* o processo `LINES_METRIC_PROCESS`, que tal como a designação sugere, computa métricas sobre linhas de código. Este processamento já era implementado pelo *Job* `Generate Metrics`, que computava estas métricas para cada *commit*. Ora, com as modificações à base de dados, este processamento foi também modificado, para que fossem computadas métricas para cada ficheiro de cada *commit*. Isto pouco altera a implementação que existia, uma vez que estas métricas quantificam linhas de código e a soma das métricas dos ficheiros de um *commit* é equivalente às métricas de um *commit*.

Ou seja, para cada ficheiro de um *commit*, são coletadas as seguintes métricas: `LINES_ADDED`, que quantifica as linhas de código adicionadas ao ficheiro neste *commit*; `LINES_REMOVED`, que quantifica as linhas de código removidas do ficheiro neste *commit*; e, por fim, `LINES_CHANGED`, que quantifica as linhas de código modificadas do ficheiro neste *commit*.

5.4.4 *Job* `Write Issues Metric`

O *Job* `Write Issues Metric` agrega as métricas intermediárias sobre a qualidade de código dos ficheiros dos vários *commits* de um projeto, por dia e por colaborador, em métricas que regista em base de dados.

Esta implementação teve vários desafios:

1. Primeiramente, o desafio foi definirmos que *commits* de um determinado dia consideramos para o cálculo das métricas.

   Ora, o desafio está em como calculamos a qualidade de código do trabalho realizado por um colaborador, para um projeto, num dia em que este realizou vários *commits* em que pode ter modificado o mesmo ficheiro. Isto porque, como o exemplo especificado pela figura 5.11 apresenta, se simplesmente agregarmos os valores das métricas dos ficheiros dos vários *commits* de um dia, como as métricas são referentes ao ficheiro todo, então potencialmente contabilizamos os mesmo *Issues* mais que uma vez, e portanto, obteremos um valor desviado do valor verdadeiro.

   Como solução poderíamos coletar para cada ficheiro de um *commit*, não só a métrica computada, como também a diferença entre esta métrica e a métrica registada

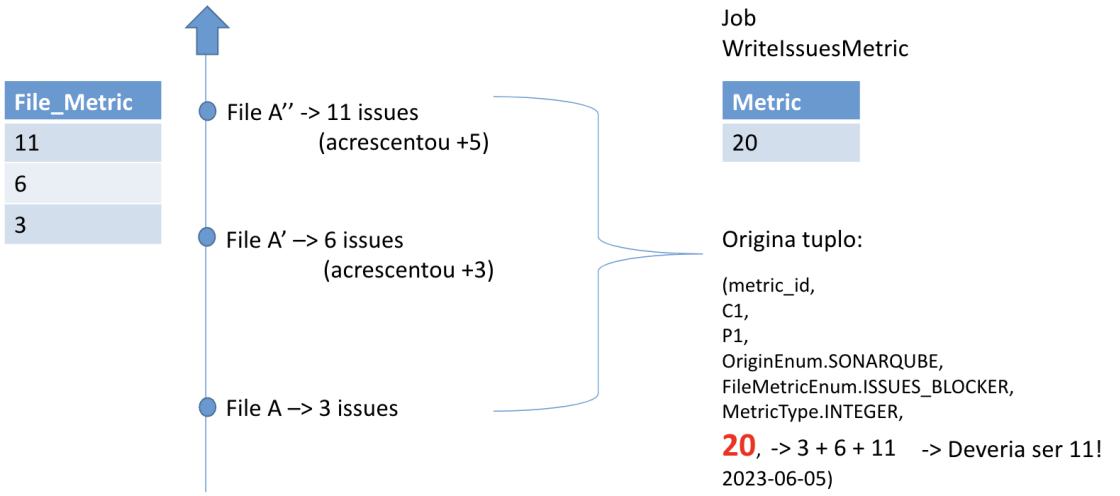

Figura 5.11: Exemplo do resultado da agregação das métricas `BLOCKER_ISSUES` dos ficheiros dos vários *commits* realizados num mesmo dia, 2023-06-05, pelo mesmo colaborador C1, num mesmo projeto P1.

pelo *commit* imediatamente anterior, para o mesmo ficheiro, e independentemente do autor. Isto porque, esta diferença representaria verdadeiramente o trabalho realizado pelo colaborador neste ficheiro, neste *commit*. Ou seja, o colaborador não seria responsabilizado pela totalidade da qualidade de código do ficheiro, mas sim do que efetivamente acrescentou. Ademais, é a agregação das diferenças que resulta num valor verdadeiro.

É importante frisar que o *Job* `Generate Issues Metric` executa a análise sobre a qualidade de código por ficheiro, para a totalidade de um ficheiro de um *commit*. Ou seja, as métricas resultantes são referentes à qualidade de código de um ficheiro por inteiro. Ora, os *commits* realizados pelos colaboradores são constituídos maioritariamente por ficheiros modificados. Isto significa que, cada colaborador modifica um ficheiro e faz *commit*, mais tarde este ficheiro é submetido a análise e como resultado o colaborador é responsabilizado pela qualidade do código que produziu nas modificações que efetuou, mas também pelo código que não modificou, e de que não é autor. Não havendo como contornarmos isto, aceitamos que um colaborador é efetivamente responsabilizado pela qualidade de código que entrega quando faz um *commit*, pois em verdade o trabalho que este realiza pode ser produtivo, mas também de boa qualidade, i.e. desenvolve linhas de código, mas tem uma preocupação ativa com a qualidade do código que desenvolve. Esta implicação não invalida o objetivo pelo qual estamos a computar métricas baseadas em *Issues*, pois o nosso objetivo é visualizarmos a evolução do trabalho realizado por um colaborador, para um projeto, ao longo do tempo.

Posto isto, optámos por uma solução mais simples: contabilizamos apenas as métricas referentes ao último *commit* efetuado num dia, por colaborador e por projeto. Esta

solução não produz valores desviados e é suficiente para asseguramos o nosso objetivo.

2. Seguidamente, o desafio foi construirmos uma métrica sobre qualidade de código - tendo por base as métricas dos ficheiros dos vários *commits* realizados num dia, por um colaborador, para um projeto -, que represente a nossa métrica de qualidade de código.

   Nesta fase, foi importante relembrarmos que:

   - A ferramenta CORQ apresenta métricas sobre linhas de código adicionadas, removidas e alteradas por um colaborador, num dia, sobre um projeto, que constituem a métrica de produtividade;
   - Esta métrica é apresentada visualmente, com recurso a um gráfico;
   - O objetivo é que, com a apresentação das métricas visualmente, seja fácil compararmos ambos e extrairmos deduções lógicas, que são posteriormente transmitidas às equipas e seus colaboradores. Portanto, estas métricas têm de ser comparáveis. É importante que não suceda que quem analise extraía informações diferentes com cada métrica e não obtenha nenhuma conclusão.

   Ora, como temos métricas por severidade para cada ficheiro de um *commit* e, como para computarmos a métrica de um dia para um colaborador basta contabilizarmos o último *commit*, a solução foi, desde logo, somarmos as métricas dos vários ficheiros por severidade, atribuindo um peso a cada severidade. Isto porque, tendo um *Issue* `BLOCKER` maior severidade que um *Issue* `MAJOR`, aquando a soma, devemos penalizar mais o valor de métricas `BLOCKER_ISSUES`, comparativamente a métricas `MAJOR_ISSUES`.

   Para tal, investiguei o SonarQube a fim de descobrir se havia algum peso especificado em documentação para cada severidade. Encontrei o *plugin* do SonarQube "Rules Compliance Index (RCI)"[77], que tal como a designação diz, calcula para um projeto o índice de conformidade com as regras. Este cálculo é o resultado de uma soma pesada dos *Issues* de um projeto, em proporção com o número de linhas de código deste. Para tal, define para cada severidade um peso, como apresentado pela figura 5.12.

   Posto isto, utilizando os valores dos pesos definidos pelo *plugin*, podemos fazer a soma pesada dos *Issues* encontrados pela análise nos vários ficheiros de um *commit*. Designámos esta métrica de `ISSUES_WEIGHTED_SUM`. Esta métrica acrescenta informação sobre qualidade de código à ferramenta CORQ e, é comparável com a métrica de produtividade que existe. Em conjunto, ambas métricas oferecem deduções lógicas sobre o trabalho realizado por um colaborador, num projeto, ao longo do tempo. Ademais, esta métrica suporta várias granularidades de análise: podemos tanto ter a visão das métricas de um colaborador inserido num projeto,

```
Issue weight = blocker violations * weight (10)
             + critical violations * weight (5)
             + major violations * weight (3)
             + minor violations * weight (1)
             + info violations * weight (0)

Rules Compliance Index = max(1.0 - (Issue Weigth / Lines of Code) * 100, 0)
```

Figura 5.12: Fórmula para cálculo do RCI, como efetuado pelo respetivo *plugin* do SonarQube.

como das métricas de um colaborador (independentemente dos projetos), ou das métricas de um projeto, ou ainda das métricas da empresa.

Por conseguinte, inspirados pelo *plugin* RCI, construímos ainda outra métrica sobre qualidade de código, a que designámos de `RCI_INDEX`. Esta métrica é o resultado de uma média ponderada dos valores dos RCI dos vários ficheiros de um *commit*. Considerámos que esta métrica acrescentaria informação como métrica de qualidade de código, pois sendo o *commit* a unidade de trabalho, esta métrica representa o impacto do trabalho realizado num dia por um colaborador, para a qualidade de código de um projeto. Isto porque, podemos afirmar que a qualidade de código de um ficheiro depende da dimensão deste. Ou seja, por exemplo, num mesmo dia, e para o mesmo projeto, o trabalho de um colaborador que acrescenta 5 *Issues* a um ficheiro com 100 linhas de código tem menos qualidade do que o trabalho de um colaborador que acrescenta 5 *Issues* a um ficheiro com 300 linhas de código.

Após termos superado estes desafios, a implementação de uma execução do *Job* foi simples. Este *Job* também executa por projeto.

Primeiramente, o *Job* executa uma *query* que solicita à base de dados o último tuplo adicionado à tabela `Metric`. Caso a resposta seja `null`, significa que a tabela está vazia e que esta é a primeira vez que o *Job* executa, e portanto, logo em seguida, o *Job* pode pedir à base de dados que devolva as métricas por ficheiro e por colaborador dos *commits* processados pelo processo `ISSUES_METRIC_PROCESS`, para computar as métricas `ISSUES_WEIGHTED_SUM` e `RCI_INDEX`, e depois registá-las. Porém, caso a *query* retorne um tuplo de `Metric`, significa que o *Job* tem de apurar quais *commits* processados não têm métricas associadas registadas e, apenas para esses, computar e registar as métricas.

Para tal, o *Job* pede à base de dados que devolva a data de processamento do *commit* referente a esta métrica, pois é a partir desta data que podem existir *commits* processados que não tenham métricas associadas registadas. Com esta, o *Job* solicita que sejam retornadas as métricas por ficheiro e por colaborador dos *commits* processados pelo processo `ISSUES_METRIC_PROCESS`, numa data posterior a esta e, apenas para os resultados retornados, computa e regista as métricas `ISSUES_WEIGHTED_SUM` e `RCI_INDEX`. Se não forem retornados resultados, então não existem *commits* processados a que tenha de computar métricas e registá-las.

5.4.5 *Job* `Write Lines Metric`

O *Job* `Write Issues Metric` agrega as métricas intermediárias sobre a produtividade do desenvolvimento do código dos ficheiros dos vários *commits* de um projeto, por dia e por colaborador, em métricas que regista em base de dados. Estas métricas de produtividade já existiam implementadas pela ferramenta CORQ.

Portanto, a implementação deste *Job* é idêntica à do *Job* `Write Issues Metric`, com exceção das métricas computadas. Este *Job* tem simplesmente de agregar os valores das métricas dos vários ficheiros do último *commit* de um dia, realizado por um colaborador, para um projeto. Isto porque, as métricas coletadas sobre um ficheiro são também as métricas a serem registadas para um *commit*, i.e. `LINES_ADDED`, `LINES_REMOVED` e `LINES_CHANGED`.

5.4.6 *Job* `Clone Repositories`

O propósito deste *Job* foi alterado. Isto porque, o *Job* recorria à biblioteca JGit para fazer a clonagem dos repositórios Bitbucket para uma diretoria local. Porém, este método é bastante demorado comparativamente a fazer a clonagem pelo comando Git.

Por conseguinte, o *Job* `Clone Repositories` apenas escreve para um ficheiro os repositórios armazenados pelo Bitbucket. Portanto, o *Job* cria um ficheiro e faz um pedido à API do Bitbucket para obter a lista dos repositórios. Para cada repositório retornado, o *Job* escreve uma linha para o ficheiro contendo a designação do repositório e o URL para o repositório, delimitados por " -".

Com isto, é executado um ficheiro *script bash* que lê este ficheiro, e para cada linha, verifica se o projeto existe numa diretoria local definida como a diretoria para onde os projetos são clonados. Caso o projeto exista nesta diretoria local, então é simplesmente atualizado. Caso contrário é clonado.

5.4.7 *Job* `Populate Database`

O *Job* `Populate Database` era composto por mais do que uma *Tasklet*, uma dedicada a popular a tabela `Collaborator`, designada `Collaborators Tasklet`, e outra para popular a tabela `Project`, designada `Projects Tasklet`.

Descartei a *Tasklet* `Projects Tasklet`, pois o papel desta era supérfluo. O trabalho desta *Tasklet* era fazer um pedido à API do Bitbucket para obter os repositórios e, para cada um, adicionar uma entrada à tabela `Project`. Porém, obtemos o mesmo efeito com a execução do *Job* `Clone Repositories`, seguida da execução do *Job* `Local Projects`.

Ademais, esta arquitetura era bastante inconveniente, pois estas *Tasklets* executavam em sequência, e portanto, não podia executar apenas uma delas.

Consequentemente, este *Job* é apenas constituído por uma *Tasklet*, tal como todos os outros *Jobs*. Ora, o trabalho desta *Tasklet* é fazer um pedido à API do Crowd para obter os colaboradores e, para cada um, adiciona uma entrada à tabela `Collaborator`. Ou seja, este *Job* popula apenas a tabela `Collaborator`.

5.4.8 *Job* `Send Email`

O *Job* `Send Email` envia um e-mail com as métricas de um intervalo de tempo especificado, por colaborador e por projeto. Ora, para tal, em tempo de execução, o *Job* pedia a um utilizador para introduzir este intervalo de tempo com o formato "yyyy-MM-dd". Ou seja, só após o utilizador introduzir este intervalo é que o *Job* executava.

Porém, o pretendido é que qualquer *Job* execute sem intervenção humana, pois provavelmente a execução dos *Jobs* será depois programada para ser efetuada por um *scheduler*. Portanto, modifiquei o *Job* para este intervalo de tempo ser especificado com o comando de execução do *Job*, através dos parâmetros "START_IN"e "END_IN".

Ademais, modifiquei o *Job* para que o e-mail enviado compreenda também as métricas sobre qualidade do código produzido.

Posto isto, a figura 5.13 apresenta um exemplo de e-mail enviado com as métricas compreendidas entre 2023-06-15 e 2023-06-30.

CORQ - Metrics - 2023-06-15 a 2023-06-30 Caixa de entrada ×

[redacted]
para mim

A informação apresentada nesta mensagem foi obtida a 2023-09-28 15:25.

Os dados apresentados referem-se a informação entre 2023-06-15 e 2023-06-30.

Colaborador	Aplicação	Linhas adicionadas	Linhas modificadas	Linhas removidas	Soma pesada de Issues	RCI
[redacted]	[redacted]	95	0	1	0	20
[redacted]	[redacted]	0	9	0	0	20
[redacted]	[redacted]	6	0	0	0	20
[redacted]	[redacted]	32.886	4.278	3.275	3.397	20
[redacted]	[redacted]	0	4	0	0	20
[redacted]	[redacted]	0	7	0	0	20
[redacted]	[redacted]	10.172	1.540	2.440	463	17
[redacted]	[redacted]	0	1	17	0	20
[redacted]	[redacted]	1.907	483	851	0	20
[redacted]	[redacted]	146	2	3	0	20
[redacted]	[redacted]	802	63	38	0	15
[redacted]	[redacted]	5.847	4.109	190	0	20
[redacted]	[redacted]	2	1	0	0	20
[redacted]	[redacted]	0	1	0	0	20
[redacted]	[redacted]	877	243	452	116	20
[redacted]	[redacted]	1	8	0	0	0

Figura 5.13: E-mail enviado com as métricas compreendidas entre 2023-06-15 e 2023-06-30.

5.4.9 *Job* `Write ElasticSearch Log`

O *Job* `Write ElasticSearch Log` escreve as métricas geradas num dia e armazenadas em base de dados para um ficheiro log, que é depois coletado pelo serviço Beats. Os logs contêm informação que depois de transformada, é indexada pelo ElasticSearch.

Com o comando de execução deste *Job*, podemos especificar um parâmetro "MODE"para a execução. Este parâmetro pode ter como valor "all", que é o valor *default*, o que significa que, independentemente da origem das métricas, escreve-as para um ficheiro log. Fora este, o *Job* aceita também para este parâmetro o valor "lines" - o que significa que apenas métricas sobre produtividade são escritas para o ficheiro log -, e ou o valor "issues" - o que significa que apenas métricas sobre qualidade de código são escritas para o ficheiro log.

Conforme o âmbito deste parâmetro, o *Job* obtém as métricas que não foram escritas para o ficheiro log. Para isso, o *Job* faz um pedido à base de dados por métricas que sejam resultado de *commits* que foram processados "hoje".

Cada métrica retornada pela *query* é escrita para uma linha do ficheiro log como uma `String` que resulta da concatenação dos atributos de um objeto `Metric`, delimitados por ";".

Isto significa que a cada dia é produzido um ficheiro log com métricas sobre os *commits* do dia anterior.

Este *Job* é diferente do *Job* `Index Reset`, que tal como a designação aponta, serve para caso o índice do ElasticSearch que contém as métricas seja apagado, indexar novamente todas as métricas registadas em base de dados. Portanto, mantemos à mesma este *Job*.

5.5 Comandos Git que reescrevem a história de um repositório

O que foi descrito com a secção 5.4 representa, em verdade, o que foi implementado numa primeira fase do trabalho. Isto porque, como os *Jobs* percorrem os repositórios Git dos projetos da Opensoft e, existem comandos Git que reescrevem a história dos *commits*, foi necessário refletirmos sobre as consequências destes comandos quando aplicados num repositório e, em alguns casos, adaptar a implementação dos *Jobs*.

Os comandos Git que reescrevem a história dos *commits* de um repositório são:

git commit - - amend [26]: este comando modifica o *commit* mais recente sem que se altere o seu *snapshot*, modificando a mensagem do *commit*, ou adicionando/removendo ficheiros ao *commit*;

git cherry-pick [25]: este comando seleciona *commits* específicos de um *branch* e aplica-os a outro *branch*;

git reset [29]: este comando move o *branch* atual para um *commit* anterior, removendo os *commits*;

git rebase [28]: este comando aplica uma sequência de *commits* de um *branch* a outro, ou seja, recria *commits* a partir de uma nova base;

git rebase -i (também referido em [28]): este comando acrescenta a *flag* `-i` ao comando rebase. Esta *flag* inicia uma sessão de `git rebase` interativo. Isto permite: reordenar os *commits*; modificar as mensagens dos *commits*; *squash*, i.e. agrupar *commits* num único *commit*; descartar *commits*; editar o conteúdo dos *commits*, adicionando ou removendo ficheiros ao *commit*.

Estes comandos podem ter implicações sobre o funcionamento do *Job* `Read Commits`, que é o *Job* que lê os *commits* dos repositórios dos projetos da empresa e os regista em base de dados. Porém, a documentação Git enfatiza que os comandos que reescrevem a história dos *commits*, com exceção do comando `git reset`, dão origem a uma ou mais (em caso de `git rebase`) referências novas (*hashes*) [55]. Ora, o funcionamento do *Job* `Read Commits` já prevê este caso e não temos de fazer alterações à implementação - o *Job* compara os *commits* que tem registados em base de dados com a totalidade dos *commits* do respetivo repositório Git, e por isso, qualquer *commit* originado por um destes comandos será identificado e adicionado à base de dados, independentemente da sua localização na história dos *commits*.

Quanto ao comando `git reset`, que remove *commits*, modificámos a entidade `Commit`, adicionando um atributo `boolean active`. Este atributo estabelece se um *commit* está ativo ou não, significando isto que o *commit* for removido do repositório. Portanto, modifiquei o *Job* `Read Commits`, para que este além de registar *commits*, também averigue se existem *commits* registados em base de dados que não encontra quando percorre o repositório Git, caso em que os marca como inativos, modificando o campo `active` do *commit* para `false`. Esta comparação é feita através do *hash* de um *commit*.

5.6 Comando `git merge`

Um comando Git que exigiu também alguma reflexão quanto às suas implicações sobre o funcionamento esperado dos *Jobs*, foi o comando `git merge` [27].

Este comando aplica os *commits* de 1 ou mais *branches* num outro *branch* destino. Isto significa que, em resultado de um `git merge`, um *commit* pode ter 2 ou mais *commits* pai associados, não havendo um limite máximo associado. Ora, isto afeta a implementação dos *Jobs* que geram métricas para os ficheiros de um *commit*, pois como foi explicado anteriormente, para o *Job* determinar quais os ficheiros modificados de um *commit*, tem de fazer a comparação entre os ficheiros do *commit* e do seu pai.

Com a primeira implementação, o *Job* apenas considerava que um *commit* podia não ter pai, por ser o *commit* pai de um repositório, ou ter apenas 1 *commit* pai, i.e. o *commit* imediatamente anterior. Por conseguinte, modificámos o método `List<ClientInputFile> getChangedFilesContent(Repository repository, String project, RevCommit commit, List<String> filesPaths)`, para que este obtenha a lista dos ficheiros modificados

de um *commit*, atendendo à totalidade dos seus *commits* pai. Ou seja, caso um *commit* seja resultado de um `git merge`, independentemente dos n *branches* envolvidos, este método obtém a lista dos ficheiros modificados neste *commit*, por comparação com os *commits* pai do *commit*. Portanto, o método obtém, em verdade, a lista dos ficheiros em que foram resolvidos conflitos, pois um *commit* resultante de um `git merge` é composto pelos conflitos entre *branches* resolvidos.

5.7 Comando `git push` tardio

Nesta fase não refletimos só sobre comandos Git que comprometem o funcionamento esperado dos *Jobs*, mas também sobre casos específicos de execução em que o resultado pode não ser o esperado, i.e. o valor das métricas calculadas.

Como referimos anteriormente, a ferramenta calcula métricas sobre o último *commit* efetuado num dia por um colaborador, para um projeto. Porém, para que a ferramenta visualize um *commit* realizado por um colaborador, o *commit* tem que ser adicionado ao repositório remoto, através do comando `git push`. Ora, um colaborador pode fazer *commit* e não fazer logo seguidamente o `git push`, fazendo no dia seguinte, ou até dias mais tarde.

O caso do `git push` tardio de um *commit* pode fazer com que a ferramenta calcule métricas com valores diferentes dos esperados. Isto porque, por exemplo, caso um colaborador efetue, num dia, mais que um *commit*, e com o último *commit* não efetue `git push`, a ferramenta não vê o último *commit*, mas vez os anteriores. Logo, as métricas não são calculadas para o último *commit* do dia, como antes especificámos para a nossa solução. Isto significa que, a métrica calculada tem um valor "errado", e tanto pode ser uma diferença significativa como não haver diferença sequer.

De forma semelhante, caso um colaborador efetue, num dia, um único *commit* e não efetue `git push`, a ferramenta não calcula métricas para esse dia com respeito a esse colaborador. Ora, neste caso a informação não está "errada", porém não está disponível e só será disponibilizada depois de realizado o `git push`.

Sendo um caso extremamente específico e como é expetável que não ocorra com frequência (pois é improvável um colaborador fazer um *commit* e não fazer o `git push`), uma solução simples que considerámos foi não fazer nada para resolver e assumir o erro. Porém, concebemos uma solução simples que é suficiente para assegurarmos que a informação das métricas se mantém correta.

Portanto, tivemos que fazer alterações à implementação dos *Jobs* que escrevem métricas, para verificar se já não existe uma entrada da tabela `Metric` com igual valor para cada atributo que faz parte da restrição de unicidade que existe sobre esta tabela, ou seja, dos atributos projeto, colaborador, métrica, origem e dia. Se existir uma entrada assim, significa que temos o valor da métrica. Senão, basta adicionarmos a entrada à tabela, como fazíamos já.

5.8 *Deployment* e artefactos produzidos

O *deployment* do CORQ é realizado pelo Jenkins, que executa o ficheiro pom.xml do CORQ. Este ficheiro apresenta informação sobre o *build*, que artefactos são gerados a partir deste, e posteriormente, sobre o *deploy* dos artefactos gerados para a máquina em que o CORQ funciona.

Ora, o *build* gera um ficheiro com extensão ".zip"que contém:

- **Diretoria bin**: apresenta vários ficheiros *script bash*, um para cada *Job*. Um ficheiro contém o comando para executar um *Job*;
- **Diretoria config**: contém o ficheiro ".properties", que por sua vez, contém propriedades utilizadas para a execução do CORQ;
- **Diretora lib**: contém ficheiros JAR das dependências utilizadas pelo CORQ;
- **corq-batch.jar**: ficheiro executável do CORQ.

Este ".zip"é instalado para uma diretoria da máquina do CORQ, especificada também pelo pom.xml.

5.9 *Cron Job*

A ferramenta CORQ tem que executar em cada 24h, o que requer a intervenção de um administrador, para introduzir manualmente os comandos que executam os *Jobs* e monitorizar a execução destes.

Porém, isto pode ser realizado automaticamente por um *scheduler*. Um *scheduler* é uma aplicação que assegura a execução dos *Jobs* num momento específico ou em resposta a um evento específico, e portanto representa uma vantagem comparativamente a uma abordagem manual.

O Linux permite definir um *Cron*, que é um *Job scheduler*, sendo portanto designado de *Cron Job*. Um *Cron Job* é definido com a adição deste ao *Crontab*, que é um ficheiro que contém uma entrada para cada *Cron Job* a executar (portanto, a adição de várias entradas forma uma tabela, daí a designação de *Crontab*).

Esta entrada compreende o comando ou *script bash* a executar, e o momento em específico em que irá executar, ou seja, tem que seguir a sintaxe formulada pela figura 5.14 (retirada a partir de [33]).

Para editar o ficheiro *Crontab*, basta utilizar o comando `crontab -e`.

Ora, a figura 5.15 mostra o *Cron Job* que definimos para o CORQ, editando o *Crontab*. Esta entrada estabelece que o *Cron Job* tem que executar o *script bash* "script-to-run-jobs.sh", todos os dias às 00:05h.

Já o conteúdo do *script bash* "script-to-run-jobs.sh"é apresentado pela figura 5.16. Este define os comandos dos *Jobs* a executar, em sequência, e com a condição que qualquer *Job*

```
*   *   *   *   *  sh /path/to/script/script.sh
|   |   |   |   |              |
|   |   |   |   |      Command or Script to Execute
|   |   |   |   |
|   |   |   |   |
|   |   |   |   |
|   |   |   | Day of the Week(0-6)
|   |   |   |
|   |   | Month of the Year(1-12)
|   |   |
|   | Day of the Month(1-31)
|   |
| Hour(0-23)
|
Min(0-59)
```

Figura 5.14: Sintaxe para definição de um *Cron Job*.

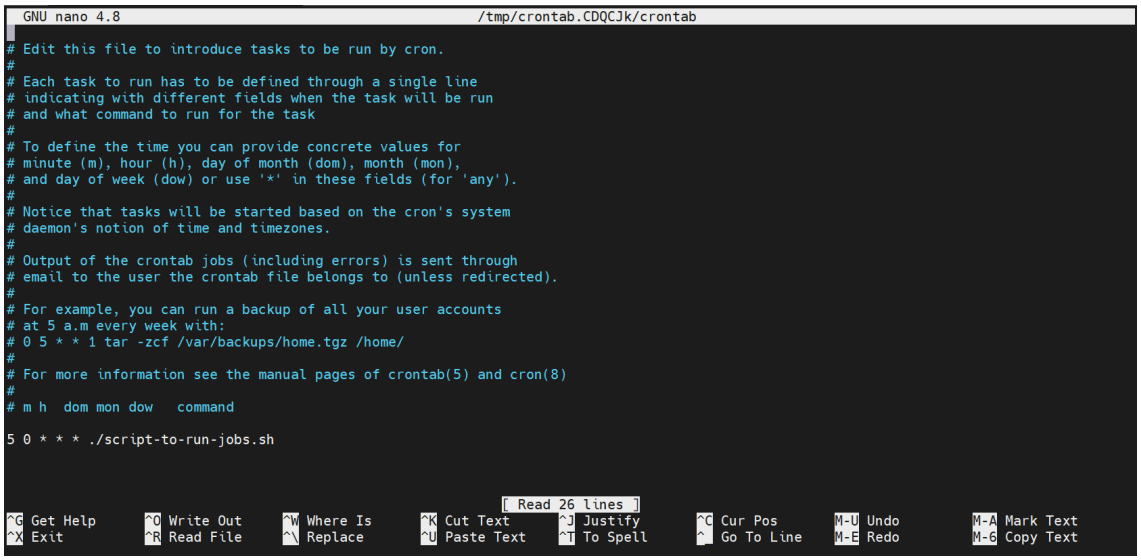

Figura 5.15: O *Cron Job* do CORQ.

que execute após outro, apenas executa se o anterior tiver terminado com sucesso (status de saída 0), o que resulta de usar "&&".

5.10 ELK Stack

5.10.1 Indexação ElasticSearch

Como já referi com a secção 4.2, a ferramenta CORQ tem integrada uma instância ELK Stack. Por conseguinte, em trabalho prévio a esta dissertação, os serviços foram instalados e configurados numa máquina remota, a máquina dedicada ao CORQ.

```
./bin/read-commits.sh &&
./bin/generate-lines-metric.sh &&
./bin/generate-issues-metric.sh &&
./bin/write-lines-metric.sh &&
./bin/write-issues-metric.sh &&
MODE=all ./write-elasticsearch-log.sh
```

Figura 5.16: O *script bash* a executar pelo *Cron Job*.

Ora, a instância do ElasticSearch que a máquina do CORQ possui, apresenta já criado um índice, designado "corq". Este índice é criado ou atualizado através de um comando HTTP, como apresentado pela figura 5.17.

```
$ curl -X PUT "http://localhost:9200/corq" -H "Content-Type: application/json" -d '{
  "settings": {
    "index.mapping.total_fields.limit": 10000
  },
  "mappings": {
    "properties": {
      "metric_enum": { "type": "keyword" },
      "origin": { "type": "keyword" },
      "project": { "type": "keyword" },
      "user": { "type": "keyword" },
      "value": { "type": "integer" },
      "date": { "type": "date" }
    }
  }
}'
```

Figura 5.17: Comando HTTP que cria ou atualiza o índice "corq".

Este comando define os campos e respetivos tipos que compõem os documentos que são armazenados por este índice, através do campo "mappings". Por exemplo, o campo "project"é mapeado para o tipo *keyword*, o que significa que o ElasticSearch trata este campo como texto exato (ou seja, que não permite divisão por *tokens*), o que é útil para pesquisa exata. Portanto, qualquer documento a ser indexado tem que respeitar este formato.

O Beats tem como papel coletar os logs e enviá-los para o Logstash. O Logstash formata cada linha de um log, que representa uma métrica, para compor um documento estruturado conforme o índice. O documento resultado é então encaminhado para o ElasticSearch. Com a receção de um documento, o ElasticSearch faz a sua indexação. O Kibana oferece uma interface para utilizadores comporem visualizações a partir das informações destes documentos indexados ou simplesmente fazerem *queries*. Ou seja, estes serviços funcionam como um pipeline poderoso para processamento dos logs e posterior análise das métricas. Para a comunicação entre eles, utilizam portas específicas.

Dito isto, a figura 5.18 apresenta o ficheiro de configuração do serviço Logstash. Por observação desta configuração, temos que:

1. **input:** secção que define a porta (5044) que o Beats usa para transmitir os logs coletados ao Logstash;

2. **filter:** secção que descreve a transformação que é aplicada ao *input*, para depois ser enviado como *output* para o ElasticSearch:

   a) Primeiramente, é aplicado um filtro `dissect` para decompor uma `String` (linha do log que representa uma métrica) com base num delimitador especificado, ";", para em seguida mapear os segmentos resultantes para os campos `metric_enum`, `origin`, `project`, `user`, `value` e `date`, respetivamente;

   b) Seguidamente, é aplicado um filtro `mutate` que converte os campos anteriormente extraídos para os tipos de dados especificados;

   c) Ademais, é aplicado um filtro ao campo `date`, para converter a partir do formato "dd-MM-yyyy"para o formato ISO 8601, que é uma representação padronizada de data e hora;

3. **output:** secção que define por onde o *input* transformado deve ser enviado. Uma das saídas é a porta (9200) por onde o *output* é enviado ao ElasticSearch, para um índice designado "corq". O campo `document_id` especifica um identificador a atribuir a cada documento a indexar, composto pelos campos anteriormente processados, delimitados por " -". Outra saída é a saída padrão (stdout), por onde o *output* é impresso para a consola usando o codec "rubydebug". Esta saída é boa prática, pois é útil para monitorizar.

```
input {
  beats {
    port => 5044
  }
}

filter {
    dissect {
        mapping => {"message" => "%{metric_enum} ; %{origin} ; %{project} ; %{user} ; %{value} ; %{date}"}
    }
    mutate {
        convert => {
            "value" => "integer"
            "metric_enum" => "string"
            "origin" => "string"
            "project" => "string"
            "user" => "string"
        }
    }

   date {
        match => [ "date", "dd-MM-yyyy", "ISO8601" ]
    }
}

output {
  elasticsearch {
    hosts => ["http://localhost:9200"]
    index => "corq"
    document_id => "%{metric_enum}-%{origin}-%{project}-%{user}-%{date}"
    action => "index"
  }
  stdout {
    codec => rubydebug
  }
}
```

Figura 5.18: Configuração dos documentos a indexar.

Foram feitas modificações a esta configuração, pois o identificador de um documento estava definido como `value-metric_enum-origin-project-user-date`. Por conseguinte,

isto ocasionava que, em caso de um push tardio (secção 5.7), a métrica atualizada produziria um documento diferente, pois o identificador considera o valor da métrica e este pode ser diferente. Ou seja, neste caso, podiam resultar, erradamente, dois documentos para a mesma métrica, com um destes desatualizado.

Posto isto, modifiquei o identificador de um documento, excluindo o campo `value`. Ou seja, o identificador está definido como `metric_enum-origin-project-user-date`, como podemos notar pela figura 5.18. Ademais, com a adição de `action => "index"`, especificamos que a ação a ser realizada pelo ElasticSearch é adicionar ou atualizar documentos já existentes.

Com estas modificações, asseguramos que qualquer documento pode ser atualizado, e que não existe informação duplicada. Isto representa um enorme ganho em eficiência, pois a primeira versão da ferramenta não permitia a atualização de um documento e superar esta limitação, era preciso apagar o índice e gerar um log contendo todas as métricas registadas em base de dados para recriar o índice e indexar novamente tudo.

5.10.2 *Dashboard*

Com a adição das métricas sobre qualidade do código produzido, o *dashboard* da ferramenta CORQ foi também evoluído. Esta fase do trabalho exigiu algum tempo para que construísse as visualizações que idealizámos.

O *dashboard* resultante é retratado pela figura 5.19. Este é composto por visualizações que apresentam as métricas por dia, sob uma vista abrangente, ou seja, da empresa.

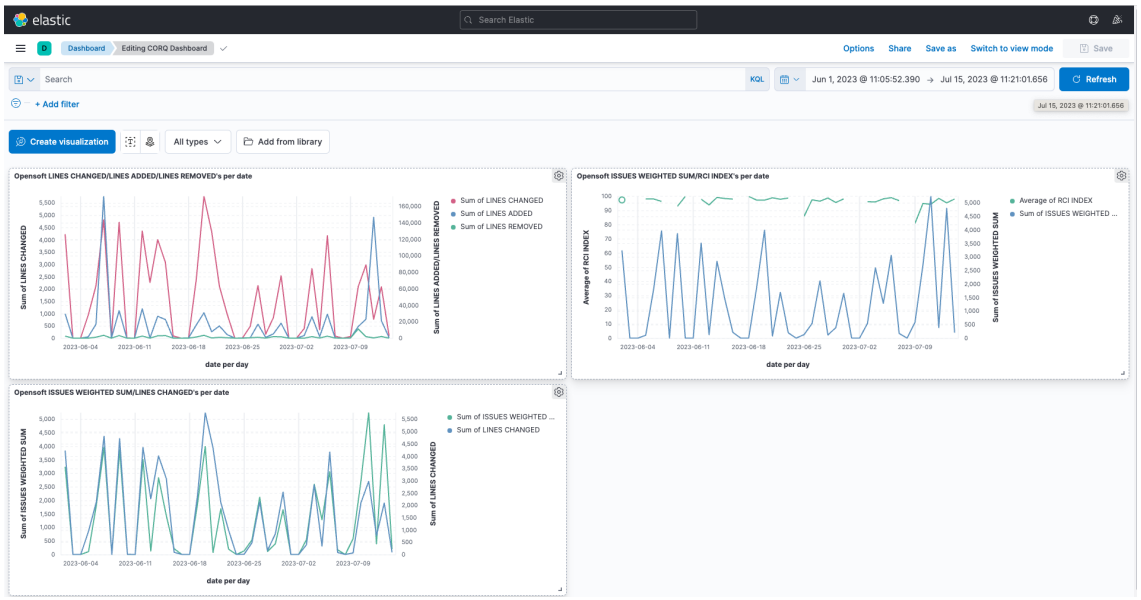

Figura 5.19: *Dashboard* CORQ atual.

Uma das visualizações apresenta a soma das métricas `LINES_CHANGED`, mas também das métricas `LINES_ADDED` e `LINES_REMOVED`, por dia. Ou seja, esta visualização apresenta a evolução das métricas de produtividade ao longo do tempo.

Outra das visualizações apresenta, por dia, a média das métricas `RCI_INDEX`, sendo esta um valor entre 0 e 100, e, simultaneamente, também apresenta a soma das métricas `ISSUES_WEIGHTED_SUM`. Ou seja, esta visualização apresenta a evolução das métricas de qualidade do código produzido ao longo do tempo.

Por fim, existe uma visualização que confronta métricas de produtividade com métricas de qualidade do código produzido, apresentando a evolução das métricas `LINES_CHANGED` e `ISSUES_WEIGHTED_SUM`, por dia.

Podemos aplicar um ou mais filtros a estas visualizações, para obter visualizações com diferentes graus de granularidade, tal como delineado pelas *user stories*. Ou seja, podemos aplicar um filtro por intervalo de tempo e fazer "drill-down"desta vista geral - aplicando um filtro para um projeto especificado (representado pela figura 5.20), ou colaborador especificado (representado pela figura 5.21), ou ambos para obtermos uma visão sobre o trabalho realizado por um colaborador num projeto (representado pela figura 5.22), por dia.

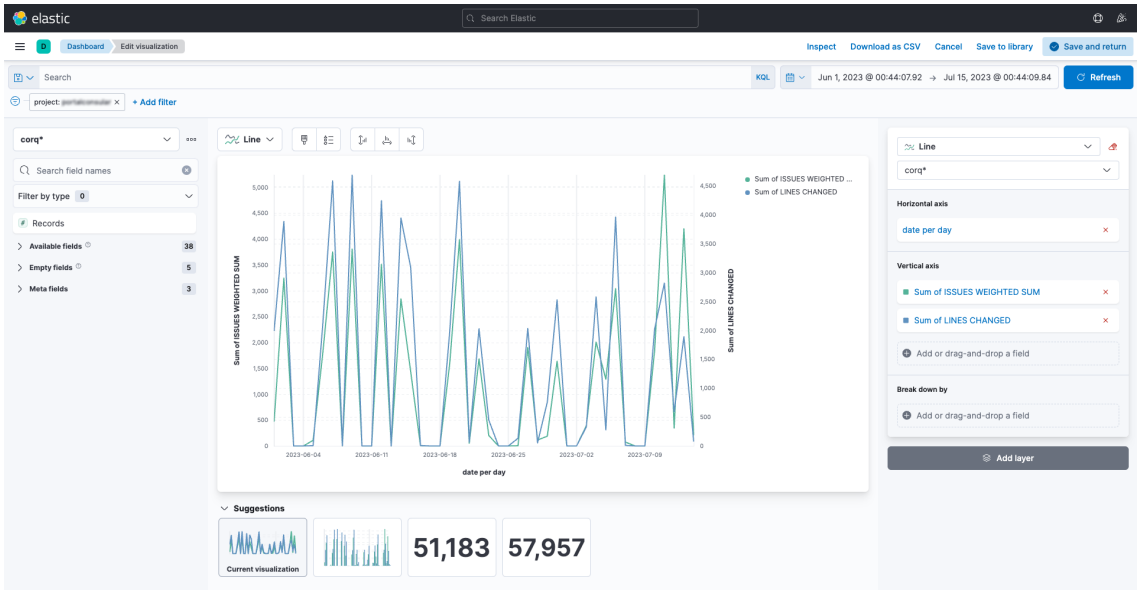

Figura 5.20: Visualização com filtro para um projeto especificado.

Portanto, com base nas visualizações apresentadas, podemos responder a questões como, por exemplo: Como tem evoluído, ao longo do tempo, o tamanho do projeto? Como tem evoluído, ao longo do tempo, a qualidade do código produzido para o projeto quanto à conformidade com os padrões da Opensoft? Como tem evoluído o tamanho do projeto comparativamente à qualidade do código? Qual o impacto do código produzido pelo colaborador sobre a qualidade de código do projeto?

Com a utilização de filtros, adaptamos as visualizações apresentadas aos pedidos dos gestores ou dos colaboradores. Ademais, os utilizadores podem modificar estas visualizações, criando visualizações personalizadas.

Por exemplo, para um diretor de projeto é importante ter uma visão sobre projetos

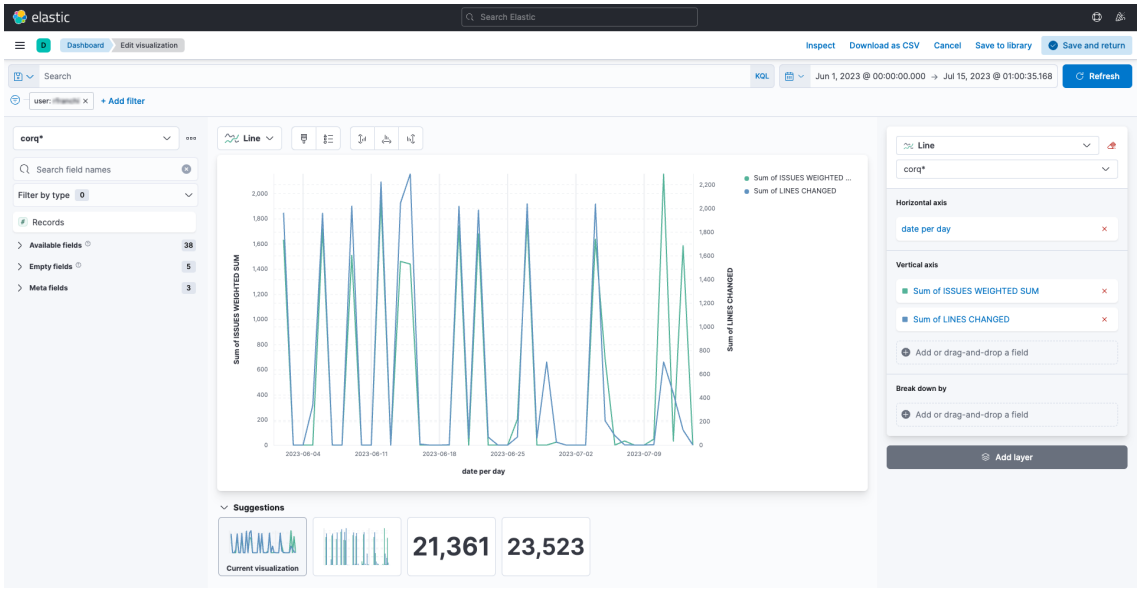

Figura 5.21: Visualização com filtro para um colaborador especificado.

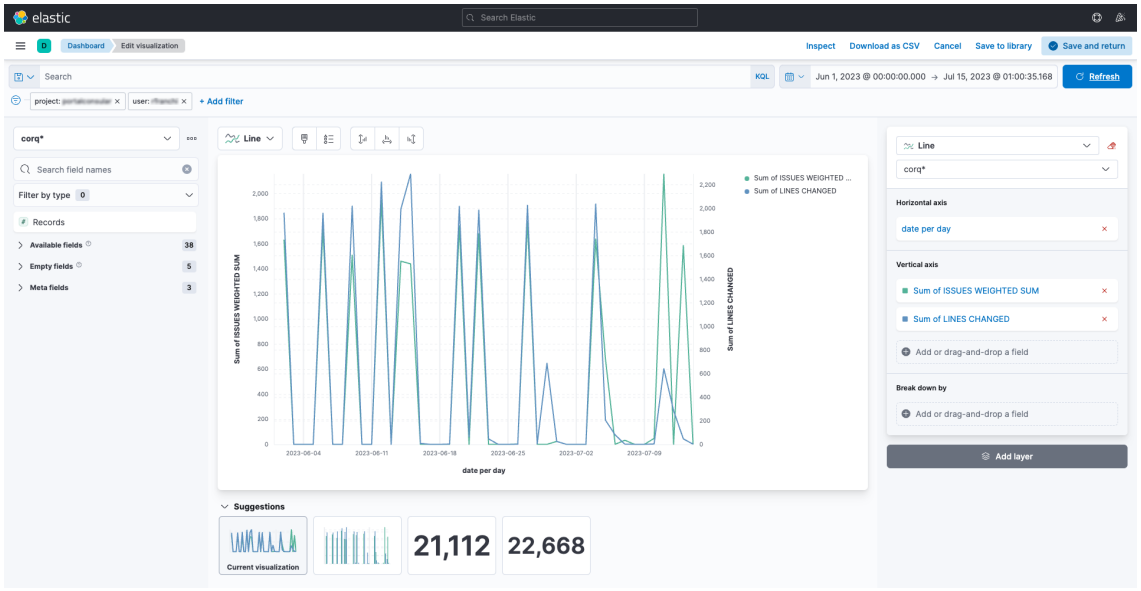

Figura 5.22: Visualização com filtro para projeto e colaborador especificado.

que gere e sobre a equipa de desenvolvimento que coordena. Ou seja, para este exemplo poderia resultar uma visualização como apresentada pela figura 5.23, em que é apresentada a evolução da métrica `LINES_CHANGED` para colaboradores que trabalham num projeto especificado.

Ademais, como podemos observar por esta figura, o Kibana oferece várias sugestões para visualizações.

Todavia, para um colaborador o importante é ter visão sobre a sua contribuição nos vários projetos em que trabalha ou em geral. Ou seja, este exemplo poderia resultar numa

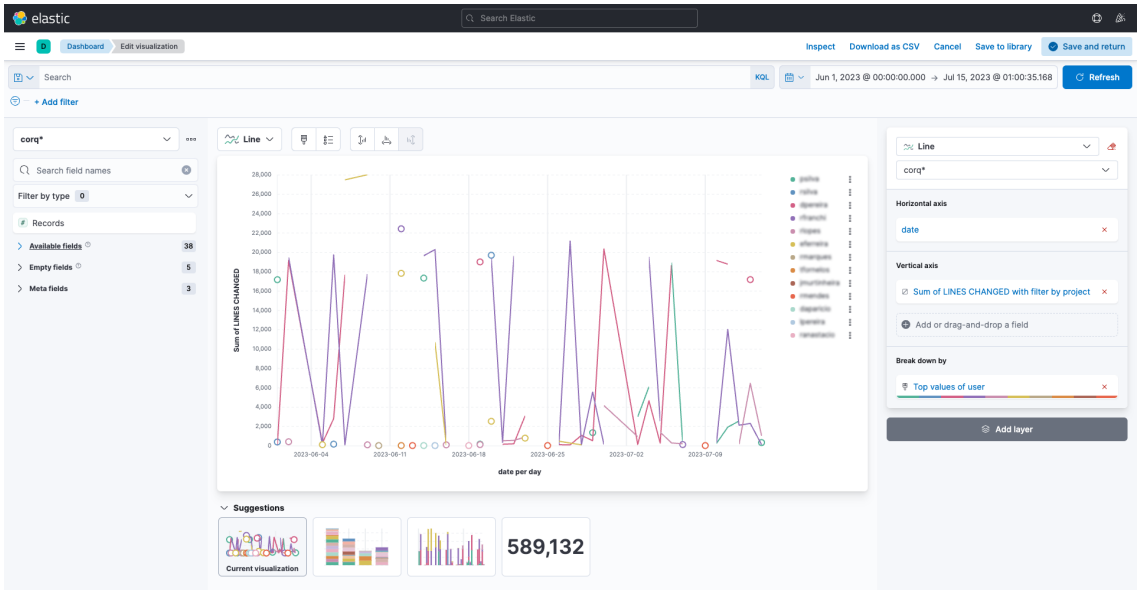

Figura 5.23: Visualização da evolução de uma métrica de produtividade, o `LINES_CHANGED`, para um projeto especificado, por colaborador.

visualização como a que é apresentada pela figura 5.24, em que é apresentada a evolução da métrica `RCI_INDEX` para os vários projetos em que o colaborador trabalha.

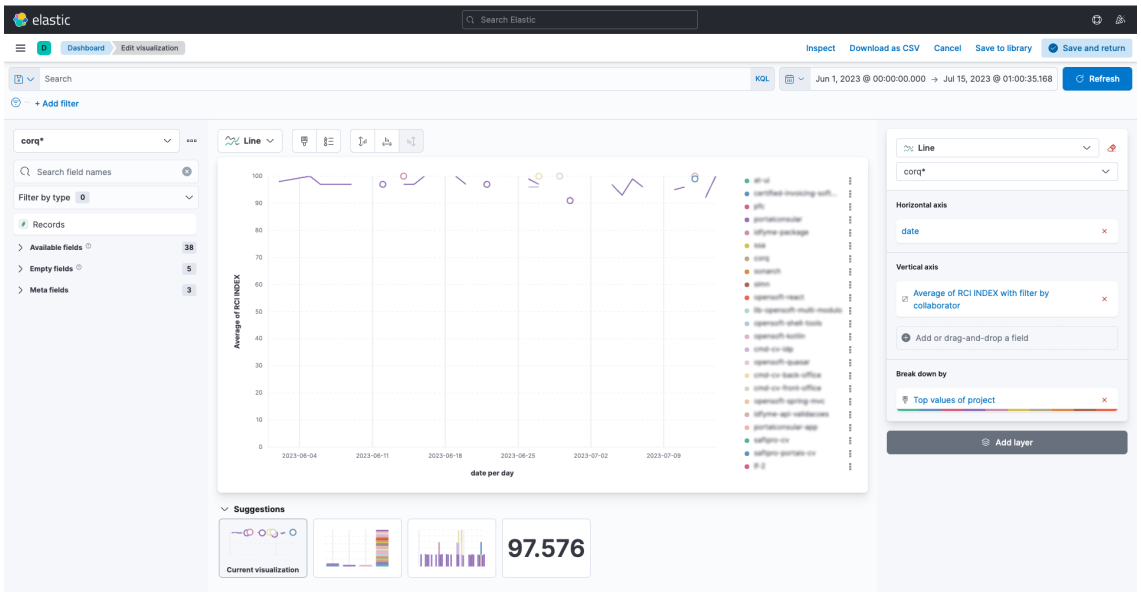

Figura 5.24: Visualização da evolução de uma métrica de qualidade do código produzido, o `RCI_INDEX`, para um colaborador especificado, por projeto.

6

Avaliação

Este capítulo apresenta uma reflexão avaliativa do trabalho desenvolvido com a dissertação, face aos objetivos esperados.

6.1 Espírito Crítico

Ora, a primeira fase do trabalho desta dissertação foi explorar o estado de arte relacionado, e o conhecimento que adquiri nesta fase foi importante para o trabalho que desenvolvi nas fases posteriores. Nomeadamente, este conhecimento foi essencial para desenvolver espírito crítico em relação ao tema desta dissertação, e sensibilidade perante uma discussão ainda bastante atual - É pouco ético medirmos o desempenho de indivíduos? Quais as consequências de o fazermos? Como o podemos fazer?

Com esta dissertação procurámos abordar este assunto com cautela, pois a ferramenta trata informação sensível. Desde logo, procurei seguir o paradigma "Goal-Question-Metric"[6], seguindo o que havia aprendido com o estado de arte.

Portanto, partimos de um objetivo - mensurar qualidade do código produzido -, para a questão fundamental "Como evolui com o tempo a qualidade do código produzido pela Opensoft?". Porém, esta questão não é exatamente específica. Isto porque, uma vez mais, como apontado pelo estado de arte, "qualidade do código"é um conceito amplo, e podemos considerar que este pode ser avaliado por diversas perspetivas - pode ser avaliado por ausência de erros, conformidade com padrões, legibilidade, manutenibilidade, etc.

Efetivamente, esta questão apenas foi especificada mais tarde. E, após formularmos a questão (geral), não seguimos para a etapa seguinte, ou seja, para formular um conjunto de métricas direcionadas a esta questão. Em vez disto, foi-me pedido para investigar a análise realizada pelo SonarQube, e integrar o motor de análise na ferramenta CORQ, a fim de analisar *commits* dos repositórios da Opensoft e, a partir dos resultados de uma análise, extrair informação para computar uma métrica sobre a qualidade do código.

Ora, isto já foi discutido anteriormente (capítulo 5, secções 5.2.1 - 5.2.2), e como apresentámos, por razões também já referidas, foi o motor de análise do SonarLint que foi integrado na ferramenta CORQ. Com uma análise, este motor produz um relatório

contendo *Issues* encontrados, com o registo para cada um das linhas do bloco de código que apresenta uma violação das regras de conformidade que a análise segue.

Portanto, somente após superarmos esta fase, é que refletimos que métrica construir direcionada à questão geral que formulámos, com base nas informações que tínhamos disponíveis, i.e. os *Issues*. E, somente após definirmos a métrica `ISSUES_WEIGHTED_SUM` e a métrica `RCI_INDEX`, ambas baseadas em *Issues*, pudemos especificar a questão geral que antes tínhamos formulado como "Como evolui com o tempo a qualidade do código produzido quanto à conformidade com os padrões da Opensoft?".

Posto isto, o que quero apontar com esta reflexão é que, por estar num contexto empresarial, adotámos uma abordagem mais prática nesta fase do trabalho, mas seguimos, efetivamente, o paradigma. Porém, com a abordagem que seguimos, restringimos o tipo de métricas que podíamos construir, para métricas baseadas em *Issues*.

Simultaneamente, por ter como base de conhecimento o que retirei com o estado de arte, defendo que temos que adicionar mais ao conjunto para podermos com exatidão retirar alguma conclusão sobre como a conformidade com padrões afeta a qualidade do código produzido. Ademais, como o objetivo é mensurar qualidade do código pela Opensoft, temos de formular mais questões específicas que contribuam para este objetivo. Ou seja, o trabalho realizado serve de base para o que tem ainda de ser desenvolvido, para que a ferramenta CORQ cumpra o seu propósito idealizado.

Com o decorrer da dissertação, foram vários os obstáculos com que me deparei e em que tive de formular soluções alternativas e apresentá-las à empresa, pois a escolha final não era minha. Porém, conservei uma postura crítica, e é neste sentido que com o capítulo 5 apresento sempre obstáculos, soluções e qual a solução implementada.

Em suma, apesar desta dissertação ter sido realizada em âmbito empresarial, pode ser caraterizada por um contínuo exercício de reflexão crítica e busca incessante por soluções eficazes.

6.2 Avaliação do *Dashboard*

O *dashboard* construído para a ferramenta CORQ concretiza as várias funcionalidades descritas pelas *user stories* (excetuando a US01, referente à autenticação dos utilizadores). Para tal, o *dashboard* apresenta visualizações sobre a evolução da produtividade e qualidade de código, da empresa, e, uma visualização pode ter aplicado um ou mais filtros. Como resultado, a partir desta vista geral, facilmente um utilizador faz "drill-down", por exemplo, obtendo a mesma visualização para um projeto específico.

É importante notar que, este *dashboard* não irá apenas servir a equipa de gestão da Opensoft, como também qualquer equipa de desenvolvimento e respetivos colaboradores. Portanto, existe um público-alvo bastante diverso, com necessidades de informação também diferentes. Por exemplo, um colaborador pode obter uma visão do contributo do seu trabalho num projeto específico em que trabalha.

Ora, a interface que o Kibana oferece é bastante versátil, permitindo personalizar praticamente qualquer aspeto, e portanto, adapta-se aos diferentes requisitos dos seus utilizadores. Daí que, para a fase de desenvolvimento da interface, o trabalho desta dissertação consistiu, resumidamente, em desenvolver um *dashboard* com apenas visualizações gerais, que irão servir de base para os utilizadores adaptarem conforme as suas necessidades. Porém, desenvolvi documentação que orienta os utilizadores a adaptar o *dashboard* e respetivas visualizações de acordo com as suas preferências.

6.3 Impacto

Como resultado do trabalho desenvolvido nesta dissertação, a ferramenta CORQ coleta informação das ferramentas de apoio ao desenvolvimento que a Opensoft utiliza, e computa métricas sobre produtividade e sobre qualidade de código. Esta ferramenta está disponível para uso de utilizadores através de um *dashboard web*, que contém visualizações que apresentam a evolução ao longo do tempo das métricas da empresa.

Com esta ferramenta, obtemos uma visão sobre a evolução da produtividade e qualidade de código da empresa nos últimos 22 anos. Por conseguinte, a equipa de gestão pode identificar tendências e padrões ao longo do tempo, e, num momento atual, reconhecer áreas críticas que requerem atenção imediata.

Ademais, esta ferramenta permite à equipa de gestão e aos colaboradores acompanharem o progresso dos projetos em tempo real, e identificar quaisquer desvios das metas estabelecidas.

Com as visualizações referidas, obtemos uma base para a tomada de decisões, permitindo que a equipa identifique áreas de melhoria e implemente estratégias ágeis eficazes.

É fundamental ressaltarmos que esta ferramenta atua como um suporte à equipa de gestão, não interferindo com a abordagem de gestão já em vigor. O papel desta ferramenta é oferecer informação por meio de um *dashboard*, para auxiliar a equipa de gestão a gerir, mais eficazmente, o desenvolvimento de *software* da Opensoft.

7

Conclusão

Nesta dissertação, foi apresentada a ferramenta CORQ, que irá dar suporte à equipa de gestão. Este foi um projeto bastante interessante e desafiante, que envolveu conhecimento diverso.

Para conclusão desta dissertação, são apresentadas as contribuições que resultaram do trabalho realizado e, é discutido trabalho que proponho ser concretizado numa evolução futura da ferramenta.

7.1 Contribuições

Ora, as contribuições desta dissertação para a ferramenta CORQ são referidas, seguidamente:

- **Evolução da ferramenta CORQ com métricas sobre qualidade de código:** uma das contribuições mais significativas desta dissertação foi a adição de métricas sobre qualidade do código produzido à ferramenta CORQ, que representam informação pertinente para a equipa de gestão da Opensoft;
- **Investigação aprofundada sobre a integração com o SonarQube:** adicionar métricas sobre qualidade de código à ferramenta CORQ, exigiu uma investigação aprofundada sobre como integrar o SonarQube no CORQ. Esta fase implicou o estudo do motor de análise da ferramenta SonarLint - este possui conexão para uma instância SonarQube, e portanto permite configurar uma análise tal como o SonarQube, com o benefício de a realizar localmente;
- **Estudo dos comandos Git que podem afetar o cálculo das métricas:** uma vez que a ferramenta CORQ faz o processamento dos repositórios Bitbucket dos projetos da Opensoft para gerar métricas, foi importante considerarmos comandos Git que podem afetar o funcionamento esperado da ferramenta, por exemplo, gerando valores errados como valores das métricas;
- **Aprimoramento da ferramenta:** nomeadamente, as modificações que efetuei ao modelo de dados, que permitiram uma implementação mais eficiente da ferramenta.

É importante destacar que o *feedback* que obtive sobre a primeira versão da ferramenta foi fundamental para realizar quaisquer aprimoramentos;

- **Definição de um *scheduler*:** a introdução de um *scheduler*, definido para executar automaticamente os *Jobs* que geram e disponibilizam as métricas, é uma vantagem significativa. Esta abordagem liberta recursos, uma vez que não requer intervenção de um administrador para tal.

- **Exploração das configurações da ELK Stack, para assegurar a indexação correta e eficiente das métricas:** isto implicou modificar a configuração que estabelece como as métricas são indexadas pelo ElasticSearch, para permitir a atualização do valor das métricas. Por conseguinte, a indexação é contínua, pois uma atualização não requer excluir o índice e reindexar toda a informação da base de dados, o que é mais eficiente;

- **Construção de um *dashboard* composto por visualizações que concretizam as *user stories* definidas:** as visualizações disponibilizadas apresentam a evolução das métricas globalmente, i.e. da Opensoft. Porém, por aplicação de um ou mais filtros, podemos obter a evolução das métricas para um projeto específico, ou para um colaborador específico, ou para o trabalho de um colaborador específico num projeto específico, para um intervalo de tempo específico;

- **Preocupação com a extensibilidade da ferramenta:** com a implementação realizada nesta dissertação, priorizei a eficiência, extensibilidade e correção da solução;

- **Elaboração de documentação:** em todas as fases do trabalho realizado nesta dissertação, produzi documentação sobre o projeto, enquanto simultaneamente produzia este documento. Portanto, o Confluence da Opensoft apresenta informação estruturada sobre desafios da implementação, alternativas pensadas, decisões tomadas, etc. tópicos que surgiram com o desenvolvimento da dissertação. Também elaborei documentação para o código que produzi.

7.2 Trabalho futuro

Como já afirmei com a secção anterior, ao longo desta dissertação, fiz questão de priorizar a extensibilidade da ferramenta CORQ para futuras evoluções.

Primeiramente, é importante adicionarmos com trabalho futuro mais métricas sobre produtividade, e mais métricas sobre qualidade de código, que respondam às questões específicas a que a equipa de gestão procura suporte. Isto porque, como referi com o capítulo 2, secção 2.2, tanto produtividade como qualidade do código são demasiado amplos como conceito para serem reduzidos a uma única métrica. Portanto, para trabalho futuro devem ser adicionadas outras métricas que complementem a informação que o CORQ já oferece. Ou então, devem ser formuladas outras questões relevantes para a equipa

de gestão, e específicas, para depois definirmos para cada questão um conjunto de métricas direcionado à resposta desta, i.e. seguindo o paradigma "Goal-Question-Metric"[6].

Por conseguinte, podemos integrar outras fontes de informação na ferramenta CORQ. Por exemplo, poderíamos integrar o Jira com o CORQ, para obter uma visão sobre um *Sprint* de um projeto - por exemplo, como evolui o trabalho definido para o Sprint, se é notório algum *bottleneck*, qual o progresso das tarefas face à estimativa de tempo planeada, etc. Uma vez mais, para isso tem de existir uma motivação para adicionar quaisquer métricas.

O processamento que a ferramenta CORQ efetua para gerar, diariamente, métricas sobre produtividade e métricas sobre qualidade do código, envolve grandes volumes de dados e é complexo. Isto posto, para trabalho futuro podemos adicionar *Threads* aos *Jobs* que processam *commits*, para acelerar o processamento. Porém, para tal, terá de ser estudado quantas *Threads* devem ser adicionadas para maximizar o desempenho da ferramenta. Também, devem ser usadas estruturas de dados "thread-safe", para assegurar que os dados partilhados entre *Threads* são protegidos de modificações concorrentes, e que as operações são atómicas, consistentes e isoladas.

Ademais, queremos integrar o Crowd no Kibana, permitindo que os vários utilizadores da Opensoft façam *login* utilizando credenciais SSO do Crowd, simplificando a autenticação e melhorando a segurança do acesso aos dados e recursos do Kibana.

Por fim, a cobertura de teste da ferramenta deve ser aumentada, implementando tanto testes unitários, como testes de integração com as fontes de informação que a ferramenta utiliza. Nesta dissertação apenas adaptei os que já existiam, porque a maioria das entidades foram modificadas com a modificação do modelo de dados da ferramenta. Todavia, não tendo sido considerada uma tarefa de prioridade face às outras tarefas de implementação que haviam a fazer, não estendi a cobertura de testes, apesar de esta também ser bastante importante para assegurar a confiabilidade da ferramenta.

Bibliografia

[1] T. M. Abdellatif, L. F. Capretz e D. Ho. «Software Analytics to Software Practice: A Systematic Literature Review». Em: *2015 IEEE/ACM 1st International Workshop on Big Data Software Engineering*. 2015, pp. 30–36. DOI: 10.1109/BIGDSE.2015.14 (ver pp. 5, 6).

[2] *AI + Analytics | Tableau*. URL: https://www.tableau.com/products/tableau-ai (acedido em 2023-09-15) (ver p. 14).

[3] *Align your teams & resources to deliver better business results*. URL: https://linearb.io/ (acedido em 2023-09-15) (ver p. 16).

[4] *Atlassian Bitbucket Git Code Management Tool for Teams*. URL: https://www.atlassian.com/software/bitbucket (acedido em 2023-09-15) (ver p. 19).

[5] *Azure DevOps Server*. URL: https://azure.microsoft.com/en-us/products/devops/server (acedido em 2023-09-15) (ver p. 44).

[6] V. R. Basili, G. Caldiera e D. H. Rombach. «The Goal Question Metric Approach». Em: vol. I. John Wiley & Sons, 1994 (ver pp. 9, 67, 72).

[7] O. Baysal, R. Holmes e M. W. Godfrey. «Developer Dashboards: The Need for Qualitative Analytics». Em: *IEEE Software* 30.4 (2013), pp. 46–52. DOI: 10.1109/MS.2013.66 (ver p. 12).

[8] *Best Data Analytics Tools & Software (2023)*. URL: https://www.forbes.com/advisor/business/software/best-data-analytics-tools/ (acedido em 2023-09-15) (ver p. 14).

[9] R. Buse e T. Zimmermann. «Information needs for software development analytics». Em: *Proceedings - International Conference on Software Engineering* (2012-06), pp. 987–996. DOI: 10.1109/ICSE.2012.6227122 (ver p. 5).

[10] R. P. Buse e T. Zimmermann. «Analytics for Software Development». Em: FoSER '10 (2010), pp. 77–80. DOI: 10.1145/1882362.1882379. URL: https://doi.org/10.1145/1882362.1882379 (ver p. 11).

[11] *Business Success Starts With a Successful Team*. URL: https://codeclimate.com/velocity/team360 (acedido em 2023-09-15) (ver p. 16).

[12] A. Chrystal e P. Mizen. «Goodhart's Law: Its Origins, Meaning and Implications for Monetary Policy». Em: *Cent Bank Monet Theory Pract Essays Honour Charles Goodhart* 1 (2003-05). DOI: 10.4337/9781781950777.00022 (ver p. 8).

[13] *Code Quality, Security & Static Analysis Tool with SonarQube ...* URL: https://www.sonarsource.com/products/sonarqube/ (acedido em 2023-09-15) (ver p. 18).

[14] *Confluence | Your Remote-Friendly Team Workspace - Atlassian*. URL: https://www.atlassian.com/software/confluence (acedido em 2023-09-15) (ver p. 17).

[15] *Crowd: SSO & Identity Management for On-Premise*. URL: https://www.atlassian.com/software/crowd (acedido em 2023-09-15) (ver p. 17).

[16] J. A. O. da Cunha et al. «Decision-Making in Software Project Management: A Qualitative Case Study of a Private Organization». Em: *2016 IEEE/ACM Cooperative and Human Aspects of Software Engineering (CHASE)*. 2016, pp. 26–32. DOI: 10.1145/2897586.2897598 (ver p. 9).

[17] H. K. Dam, T. Tran e A. Ghose. «Explainable Software Analytics». Em: ICSE-NIER '18 (2018), pp. 53–56. DOI: 10.1145/3183399.3183424. URL: https://doi.org/10.1145/3183399.3183424 (ver p. 11).

[18] T. H. Davenport, J. G. Harris e R. Morison. *Analytics at work: Smarter decisions, better results*. Harvard Business Press, 2010 (ver pp. 10, 11).

[19] *Developer Progress*. URL: https://waydev.co/features/developer-progress/ (acedido em 2023-09-15) (ver p. 15).

[20] *Developer Summary*. URL: https://waydev.co/features/developer-summary/ (acedido em 2023-09-15) (ver p. 15).

[21] *Einstein Discovery | Empower everyone with trusted and ...* URL: https://www.tableau.com/products/einstein-discovery (acedido em 2023-09-15) (ver p. 14).

[22] *Elastic Stack: Elasticsearch, Kibana, Beats & Logstash*. URL: https://www.elastic.co/pt/elastic-stack (acedido em 2023-09-15) (ver p. 21).

[23] *ELK Stack*. URL: https://logz.io/learn/complete-guide-elk-stack/ (acedido em 2023-09-15) (ver p. 22).

[24] S. Few. *Information Dashboard Design: The Effective Visual Communication of Data*. O'Reilly Media, Inc., 2006. ISBN: 0596100167 (ver p. 12).

[25] *Git - git-cherry-pick Documentation*. URL: https://git-scm.com/docs/git-cherry-pick (acedido em 2023-09-15) (ver p. 56).

[26] *Git - git-commit Documentation*. URL: https://git-scm.com/docs/git-commit (acedido em 2023-09-15) (ver p. 56).

[27] *Git - git-merge Documentation*. URL: https://git-scm.com/docs/git-merge (acedido em 2023-09-15) (ver p. 57).

[28] *Git - git-rebase Documentation*. URL: https://git-scm.com/docs/git-rebase (acedido em 2023-09-15) (ver p. 57).

[29] *Git - git-reset Documentation*. URL: https://git-scm.com/docs/git-reset (acedido em 2023-09-15) (ver p. 56).

[30] D. Graziotin, X. Wang e P. Abrahamsson. «How Do You Feel, Developer? An Explanatory Theory of the Impact of Affects on Programming Performance». Em: *PeerJ Computer Science* 1 (2015-05). DOI: 10.7717/peerj-cs.18 (ver p. 8).

[31] A. Hindle et al. «On the naturalness of software». Em: *Proceedings - International Conference on Software Engineering* (2012-06), pp. 837–847. DOI: 10.1109/ICSE.2012.6227135 (ver p. 5).

[32] *How the best software teams get better*. URL: https://www.swarmia.com/ (acedido em 2023-09-15) (ver p. 16).

[33] *How to Automate Tasks with cron Jobs in Linux*. URL: https://www.freecodecamp.org/news/cron-jobs-in-linux/ (acedido em 2023-09-15) (ver p. 59).

[34] *Install the server*. URL: https://docs.sonarsource.com/sonarqube/9.9/setup-and-upgrade/install-the-server/ (acedido em 2023-09-15) (ver p. 30).

[35] *Introducing Developer360: The First Full Picture Of Engineering Work*. URL: https://codeclimate.com/blog/developer360-launch (acedido em 2023-09-15) (ver p. 16).

[36] *Issues*. URL: https://docs.sonarsource.com/sonarqube/9.9/user-guide/issues/ (acedido em 2023-09-15) (ver pp. 27, 28).

[37] C. Jaspan e C. Sadowski. «No Single Metric Captures Productivity». Em: *Rethinking Productivity in Software Engineering*. Ed. por C. Sadowski e T. Zimmermann. Berkeley, CA: Apress, 2019, pp. 13–20. ISBN: 978-1-4842-4221-6. DOI: 10.1007/978-1-4842-4221-6_2. URL: https://doi.org/10.1007/978-1-4842-4221-6_2 (ver pp. 6, 8, 9).

[38] *Jenkins*. URL: https://www.jenkins.io/ (acedido em 2023-09-15) (ver p. 21).

[39] *JGit - cookbook*. URL: https://github.com/centic9/jgit-cookbook (acedido em 2023-09-15) (ver p. 21).

[40] *JGit | The Eclipse Foundation*. URL: https://eclipse.dev/jgit/ (acedido em 2023-09-15) (ver p. 20).

[41] *Jira | Issue & Project Tracking Software - Atlassian*. URL: https://www.atlassian.com/software/jira (acedido em 2023-09-15) (ver p. 19).

[42] A. J. Ko. «Why We Should Not Measure Productivity». Em: *Rethinking Productivity in Software Engineering*. Ed. por C. Sadowski e T. Zimmermann. Berkeley, CA: Apress, 2019, pp. 21–26. ISBN: 978-1-4842-4221-6. DOI: 10.1007/978-1-4842-4221-6_3. URL: https://doi.org/10.1007/978-1-4842-4221-6_3 (ver pp. 6–9).

[43] A. Kruglov, D. Strugar e G. Succi. «Tailored performance dashboards—an evaluation of the state of the art». Em: *PeerJ Computer Science* 7 (2021-10), e625. DOI: 10.7717/peerj-cs.625 (ver p. 13).

[44] *Leverage Engineering Intelligence for Visibility, Velocity & Business Alignment*. URL: https://waydev.co/ (acedido em 2023-09-15) (ver p. 15).

[45] *Linter IDE Tool & Real-Time Software for Code with Sonarlint*. URL: https://www.sonarsource.com/products/sonarlint/ (acedido em 2023-09-15) (ver p. 31).

[46] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (ver p. ii).

[47] *MariaDB Foundation - MariaDB.org*. URL: https://mariadb.org/ (acedido em 2023-09-15) (ver p. 18).

[48] S. Martínez-Fernández et al. «Continuously Assessing and Improving Software Quality With Software Analytics Tools: A Case Study». Em: *IEEE Access* PP (2019-05). DOI: 10.1109/ACCESS.2019.2917403 (ver p. 8).

[49] *Maven – Welcome to Apache Maven*. URL: https://maven.apache.org/ (acedido em 2023-09-15) (ver p. 21).

[50] T. Menzies e T. Zimmermann. «Software Analytics: What's Next?» Em: *IEEE Software* 35.5 (2018), pp. 64–70. DOI: 10.1109/MS.2018.290111035 (ver p. 9).

[51] *Optimize software delivery. Build happier, healthier teams.* URL: https://www.pluralsight.com/product/flow (acedido em 2023-09-15) (ver p. 16).

[52] *PME Excelência*. URL: https://www.iapmei.pt/PRODUTOS-E-SERVICOS/Qualificacao-Certificacao/PME-Lider/PME-Excelencia.aspx (acedido em 2023-09-15) (ver p. 23).

[53] *PME Líder*. URL: https://www.iapmei.pt/PRODUTOS-E-SERVICOS/Qualificacao-Certificacao/PME-Lider/PME-Lider.aspxy (acedido em 2023-09-15) (ver p. 23).

[54] *Quality profiles*. URL: https://docs.sonarsource.com/sonarqube/9.9/instance-administration/quality-profiles/ (acedido em 2023-09-15) (ver p. 27).

[55] *Rewriting history*. URL: https://www.atlassian.com/git/tutorials/rewriting-history (acedido em 2023-09-15) (ver p. 57).

[56] T. Rique et al. «On Adopting Software Analytics for Managerial Decision-Making: A Practitioner's Perspective». Em: *IEEE Access* 11 (2023), pp. 73145–73163. DOI: 10.1109/ACCESS.2023.3294823 (ver p. 9).

[57] T. Rique et al. «Use Cases for Software Development Analytics: A Case Study». Em: *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*. SBES '22. Virtual Event, Brazil: Association for Computing Machinery, 2022, pp. 388–397. ISBN: 9781450397353. DOI: 10.1145/3555228.3555239. URL: https://doi.org/10.1145/3555228.3555239 (ver p. 9).

[58] A. Sarikaya et al. «What Do We Talk About When We Talk About Dashboards?» Em: *IEEE Transactions on Visualization and Computer Graphics* 25.1 (2019), pp. 682–692. DOI: 10.1109/TVCG.2018.2864903 (ver p. 12).

[59] *Sonarlint Connected Mode*. URL: https://docs.sonarsource.com/sonarqube/9.9/user-guide/sonarlint-connected-mode/ (acedido em 2023-09-15) (ver p. 32).

[60] *SonarSource/sonarlint-core*. URL: https://github.com/SonarSource/sonarlint-core (acedido em 2023-09-15) (ver p. 32).

[61] *Spring | Home*. URL: https://spring.io/ (acedido em 2023-09-15) (ver p. 19).

[62] *Spring Batch*. URL: https://spring.io/projects/spring-batch (acedido em 2023-09-15) (ver pp. 19, 20).

[63] *Spring Boot*. URL: https://spring.io/projects/spring-boot (acedido em 2023-09-15) (ver p. 19).

[64] *Spring Data JPA*. URL: https://spring.io/projects/spring-data-jpa (acedido em 2023-09-15) (ver p. 19).

[65] M.-A. Storey e C. Treude. «Software Engineering Dashboards: Types, Risks, and Future». Em: *Rethinking Productivity in Software Engineering*. Ed. por C. Sadowski e T. Zimmermann. Berkeley, CA: Apress, 2019, pp. 179–190. ISBN: 978-1-4842-4221-6. DOI: 10.1007/978-1-4842-4221-6_16. URL: https://doi.org/10.1007/978-1-4842-4221-6_16 (ver pp. 12, 13).

[66] *Tableau: Business Intelligence and Analytics Software*. URL: https://www.tableau.com/ (acedido em 2023-09-15) (ver p. 14).

[67] S.-Y. Tan e T. Chan. «Defining and Conceptualizing Actionable Insight: A Conceptual Framework for Decision-centric Analytics». Em: (2016). arXiv: 1606.03510 [cs.CY] (ver p. 6).

[68] *Team Progress*. URL: https://waydev.co/features/team-progress/ (acedido em 2023-09-15) (ver p. 15).

[69] *The Best Data Visualization Tools Of 2023*. URL: https://www.forbes.com/advisor/business/software/best-data-visualization-tools/ (acedido em 2023-09-15) (ver p. 14).

[70] C. Treude, F. Figueira Filho e U. Kulesza. «Summarizing and Measuring Development Activity». Em: (2015-08). DOI: 10.1145/2786805.2786827 (ver pp. 7, 8).

[71] C. Treude e M.-A. Storey. «Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds». Em: 1 (2010), pp. 365–374. DOI: 10.1145/1806799.1806854 (ver p. 12).

[72] *Trusted engineering insights for maximum business impact*. URL: https://codeclimate.com/ (acedido em 2023-09-15) (ver p. 15).

[73] *User Stories | Examples and Template - Atlassian*. URL: https://www.atlassian.com/agile/project-management/user-stories (acedido em 2023-09-15) (ver p. xii).

[74] A. Vázquez-Ingelmo, F. García-Peñalvo e R. Therón. «Information Dashboards and Tailoring Capabilities - A Systematic Literature Review». Em: *IEEE Access* 7 (2019-08), pp. 109673–109688. DOI: 10.1109/ACCESS.2019.2933472 (ver p. 13).

[75] *Visualização de Dados | Microsoft Power BI*. URL: https://powerbi.microsoft.com/pt-pt/ (acedido em 2023-09-15) (ver p. 14).

[76] *What is Scrum?* URL: https://www.scrum.org/learning-series/what-is-scrum/what-is-scrum (acedido em 2023-09-15) (ver p. xiii).

[77] *willemsrb/sonar-rci-plugin*. URL: https://github.com/willemsrb/sonar-rci-plugin (acedido em 2023-09-15) (ver p. 52).

[78] *Work Log*. URL: https://waydev.co/features/work-log/ (acedido em 2023-09-15) (ver p. 15).