OULUN AMMATTIKORKEAKOULU

Teemu Kiiski

SEARCHING FOR VALUE

Using Apache Solr to Find Relationships Between Work Items in a Product Backlog

SEARCHING FOR VALUE

Using Apache Solr to Find Relationships Between Work Items in a Product Backlog

Teemu Kiiski
Master's thesis
Autumn 2023
Degree Programme in Data Analytics
and Project Management
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Data Analytics and Project Management

Author(s): Teemu Kiiski
Title of thesis: Searching for Value: Using Apache Solr to Find Relationships Between Work Items in a Product Backlog
Supervisor(s): Teppo Räisänen
Term and year when the thesis was submitted: Autumn 2023
Number of pages: 45

Software bugs are an inevitable part of software development. The question often arises as to who should pay for the bugfix. In a dual business model where a software product is developed under internal product development and customer projects, assigning a bugfix to the correct category is crucial. If a bug cannot be assigned to the customer project that caused it, it will become technical debt that competes for the capacity reserved for internal product development.

This Master's thesis project was commissioned by Innofactor, a Finnish software company that operates in the Nordic countries and provides Microsoft and in-house solutions for promoting a digital organization. The project tackled the problem of identifying bugs that have been caused by customer-requested code changes. The proposed solution was a helper tool that looks for similarities between a new bug report and other work items in a product backlog.

The problem of looking for similarities between work items based on text data is an information retrieval problem. In order to quantify the problem, quantitative data collection and analysis was carried out. Likewise, the processing of software bugs is related to software quality assurance. The domains of information retrieval and software quality assurance formed the theoretical framework for the project. Action research was chosen as the research method.

The goal of the project was to create a proof of concept for a helper tool, which could be used to find the work item related to the code change that caused the bug. A proof of concept was built on the Apache Solr search platform. The backlog data was exported from a task management tool and indexed in Solr. The MoreLikeThis feature was used to test the solution, which produced promising results but also revealed further development needs.

Keywords: Apache Solr, information retrieval, software quality assurance, software bug, product backlog, proof of concept, data analysis

CONTENTS

1 INTRODUCTION

The goal of this thesis project is to create a proof of concept for a helper tool that can find relationships between work items in a product backlog. The helper tool would enable software testers to connect a new bug report with the work item under which the feature or change that caused the bug was developed. A helper tool that can show that a bug was caused by the development of a customer-requested code change would prevent the bug from becoming technical debt that competes for the capacity reserved for internal product development.

The topic for this Master's thesis came up at my workplace at Innofactor. Innofactor also commissioned this thesis project. Innofactor is a Finnish software company that operates in the Nordic countries and provides Microsoft and in-house solutions for promoting a digital organization. My role as a Scrum Master in the Dynasty software development team revolves around developing and maintaining the Dynasty product family. Innofactor's clients, mostly from the public sector, use the Dynasty suite for information and case management processes.

In software development, new features are created by software developers. These features are described in a product backlog, typically under work items called product backlog items or user stories. Software testers test these new features for any software faults, which are recorded in the task management tool under work items called bugs. Developers not only develop new features but also fix bugs caused by earlier features or other changes to the source code, such as earlier bugfixes where a developer fixed something but broke something else.

Knowing what change caused a bug makes it possible to determine who should pay for its fixing. In a software product with a large number of customer organizations, different customers pay for different features, and some features are developed at the software provider's own expense to attract new customers. Bugfixes are an inevitable part of the development process. However, if a bug report cannot be connected with the customer-requested change that caused it, it will have to be fixed at the software provider's own expense.

The proposed solution is to create a helper tool that looks for similarities in the bug report and any previous work items in a product backlog. The comparison of similarities between work items will be based on text fields such as title, description, and tags. The problem of looking for similarities

between work items based on text data is an information retrieval problem. The helper tool will be built on Apache Solr, an open-source enterprise search platform. Solr also has analysis capabilities that could be explored for additional value outside the scope of this project.

Having a helper tool that can help determine which work item caused the bug would enable software testers to assign otherwise uncertain bugs to customer projects. This would lead to more work that could be allocated to customer projects, which would generate business value. It would also give insight to the amount of technical debt that is caused by specific customer projects and internal product development. When the number of bugs that are fixed each year is in the thousands, a more efficient allocation of bugs to customer projects can have a positive impact.

The goal of this thesis project is to create a proof of concept for the helper tool. Proof of concept is a small-scale realization of a method or an idea to demonstrate its feasibility (IATE 2023). The research project will take the form of action research. Action research is a qualitative research method that synergizes research and practice. In the iterative process of action research, the researcher and a practitioner collaborate to diagnose a real-life problem, introduce action intervention by developing a solution, and conduct reflective learning (Avison, D. et al. 1999).

The theoretical framework of this study is related to the fields of software quality assurance and information retrieval, which will be covered in the literature review. My personal motivation for this thesis project comes from the opportunity to combine my work experience in software quality assurance with a desire to acquire and demonstrate the technical know-how necessary to produce the proof of concept. I also wish to demonstrate that the skills acquired during the project can be used to generate measurable business value.

2 THEORETICAL FRAMEWORK

This chapter describes the theoretical framework of the project, which is formed by the domains of software quality assurance and information retrieval.

2.1 Software Quality Assurance

2.1.1 SQA Concepts and Software Bug Types

An early definition for the term software by the Institute of Electrical and Electronics Engineers (IEEE) in 1991 defines software as “computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system”. According to Galin, software quality assurance always includes the quality of all these four components. (Galin 2004, 14–34) While the definition for software may evolve over time, the notion is that software quality assurance is not limited to the quality of the code behind the software.

The terms software quality and software quality assurance should be differentiated. Galin proposes the definition of software quality by Roger Pressman, which sets three requirements for software quality: compliance with specific functional requirements, compliance with software quality standards mentioned in the contract, and good engineering practices that may not be explicitly stated in the contract. According to Galin, Pressman’s definition for software quality “provides operative directions for testing the degree to which these requirements are met”. (Galin 2004, 14–34.)

A narrow definition for software quality assurance is concerned with the software development process and conformance to technical requirements. Galin proposes an expanded definition that covers both the software development process and the maintenance process of a software product. In other words, the process should conform to technical requirements as well as managerial requirements, such as budgeting and scheduling issues. According to Galin, the broader definition should yield better results and customer satisfaction. (Galin 2004, 14–34.)

Software quality assurance should also be differentiated from software quality control, which can be mistaken to be synonymous. Quality control is concerned with evaluating the quality of products

where the main objective is to withhold products that do not qualify. Quality assurance is concerned with detecting, correcting and preventing errors throughout the development process where the main goal is to minimize the cost of guaranteeing quality. In other words, quality control forms only a part of all the activities related to quality assurance. (Galin 2004, 14–34.)

Following the earlier definition that identified four components of software, software errors can similarly be divided into code errors, procedure errors, documentation errors, and data errors. Galin identifies nine causes of software errors: “faulty requirements definition, client-developer communication failures, deliberate deviations from software requirements, logical design errors, coding errors, non-compliance with documentation and coding instructions, shortcomings of the testing process, procedure errors, and documentation errors.” (Galin 2004, 14–34.)

A code error is a typical type of software error. Galin describes the way and the terminology used when a mistake in a software’s source code becomes a problem that affects the user. Firstly, software error is a section of the code that is partially or totally incorrect. It can become a software fault that causes the software to function incorrectly, or it may remain unnoticeable to the user. When the software fault is activated by the user that is attempting to perform a specific function in the software, it becomes a software failure. (Galin 2004, 14–34.)

In the software industry, software faults are typically called software bugs, or just bugs. Sometimes the term defect is used to add further distinction. Based on a review of online materials, it appears that there does not exist a consensus for differentiating the terms bug and defect. For example, defect may be used to refer to requirements mismatch or when a software fault is detected in production. On the other hand, defect may sometimes be used as a synonym for bug. In this thesis report, the term bug is used in the broad sense that includes all types of software issues.

In a 2019 study, Catolino and colleagues analyzed 1280 bug reports in different software projects and proposed a taxonomy of bug types. The following nine bug types were proposed: configuration issue, network issue, database-related issue, GUI-related issue, performance issue, permission/deprecation issue, security issue, program anomaly issue, and test-code related issue. The study found that the bug types have different characteristics on how developers deal with them, for example in terms of reaction time and fixing time. (Catolino et al. 2019.)

Existing classification schemes for software bugs were also discussed in the study. Orthogonal Defect Classification (ODC), developed at IBM, is an early and popular bug taxonomy. The ODC taxonomy, which has 13 categories, classifies bugs based on the underlying program structure. According to Catolino et al., a limitation of the ODC classification is that it does not address the type of the issue. Other bug taxonomies have also been proposed, typically with a focus on specific application types or particular bug types. (Catolino et al. 2019.)

Classification schemes for software bugs have also been proposed based on the circumstances of their occurrence. Grottke and Trivedi, in their 2005 fast abstract, proposed a classification scheme that offered definitions for the terms Bohrbug, Heisenbug, Mandelbug, and aging-related bug. However, some these terms were originally coined decades earlier. For example, the term Bohrbug was first used in the literature in 1985. According to Grottke and Trivedi, the terms were used inconsistently, creating a need for explicit definitions. (Grottke and Trivedi 2005.)

According to the proposition by Grottke and Trivedi, every software failure is either a Bohrbug or Mandelbug. Bohrbug is "a fault that is easily isolated and that manifests consistently under a well-defined set of conditions". On the other hand, a Mandelbug is "a fault whose activation and/or error propagation are complex …". The activation of the failure could be influenced by other elements of the software system, such as the operating system, or there could a time lag between fault activation and failure occurrence. (Grottke and Trivedi 2005.)

The Mandelbug software fault type has two subtypes: Heisenbug and aging-related bug. Heisenbug is an elusive fault where the act of observing, for example as a consequence of using a specific observation method or tool, influences the failure behavior or causes it. Lastly, an aging-related bug is another subtype of Mandelbug where the error condition does not lead to software failure right away. A software failure that does not fit either subtype is called just a Mandelbug. (Grottke and Trivedi 2005.)

According to Catolino et al., bug triage is "the process of assigning the fixing of a reported bug to the most qualified developer". This is typically conducted manually, which can be time-consuming. Research has been made on automated classification techniques that make use of approaches such as machine learning, natural language processing, and text mining. For example, models have been created to classify bugs based on classification schemes, to differentiate between bugs and feature requests, and to assign the right bug to the right developer. (Catolino et al. 2019.)

2.1.2 DevOps and Future Directions of SQA

The software development team at my workplace in Innofactor uses Large Scale Scrum (LeSS) to organize its work. Large-Scale Scrum is framework that scales the one-team Scrum framework for multiple teams. Scrum is defined by the Scrum Guide as “lightweight framework that helps people, teams and organizations generate value through adaptive solutions for complex problems” (Schwaber and Sutherland 2020). Another common definition for Scrum is an agile project management or development system.

Agile software development refers to methods and practices that emerged in the late 1990s and early 2000s to challenge more traditional approaches that advocate “extensive planning, codified processes, and rigorous reuse to make development an efficient and predictable activity”. Agile frameworks and methodologies, such as Scrum, rely on “people and their creativity rather than on processes”, and emphasize response to change. The Agile Manifesto, written in 2001, popularized a set of values and principles associated with agile. (Dyba and Dingsøyr 2008.)

Building on the context of software quality assurance and agile, the concept of DevOps should be discussed. DevOps is a methodology used for integrating and synergizing activities related to software development and operations. According to a 2020 study by Mishra and Ziadoon, DevOps is a recent concept and there does not yet exist consensus in the research literature on an exact definition. Various definitions for DevOps emphasize collaboration between development and operations. (Mishra and Ziadoon 2020.) The term DevOps was coined in 2008 (Leite et al. 2019).

According to a 2019 study by Leite and colleagues, DevOps is an evolution of the agile movement, which “proposes a complementary set of agile practices to enable the iterative delivery of software in short cycles effectively”. The DevOps movement originated from the conflict between developers, who develop new features, and operators, who manage software modifications. These challenges resulted in inefficient implementation of agile practices. In this context, new collaboration between developers and operators gave birth to the DevOps movement. (Leite et al. 2019.)

In their 2020 study, Mishra and Ziadoon analyzed previous research on the implications of DevOps on software quality and concluded that activities associated with DevOps have a positive effect on software quality. Activities related to automation, sharing (collaboration and information sharing) and measurement (following performance metrics) were found to have a strong relationship with

software quality. The authors characterized DevOps as ”a fast feedback loop enabler which is essential in achieving software quality”. (Mishra and Ziadoon 2020.)

One of the goals of DevOps is automating the software delivery process. Therefore, DevOps is often closely associated with the concepts of continuous delivery and continuous deployment. These terms should not be used interchangeably. Continuous delivery is a process where software changes go through a pipeline with various stages, such as automated tests, and the changes are automatically released to the repository. In continuous deployment, all the changes that go through the pipeline are automatically deployed to production. (Leite et al. 2019.)

As mentioned earlier, the goal of DevOps is not limited to automating the delivery process. According to Leite et al., “DevOps initiatives have also focused on using automated runtime monitoring for improving software runtime properties, such as performance, scalability, availability, and resilience”. Site Reliability Engineering (SRE) is another approach that applies DevOps practices at runtime. (Leite et al. 2019.) The concept of Site Reliability Engineering originates from Google and has been characterized as an implementation of DevOps.

DevOps makes use of automation tools. Leite et al. identify six categories for DevOps tools: knowledge sharing, source code management, build process, continuous integration, deployment automation, and monitoring and logging (Leite et al. 2019). In continuous integration (CI), code changes are frequently merged to the main branch for validation by automated tests. A process using DevOps practices may be called a CI/CD pipeline, although the term can have different meaning depending on the degree of automation. (Red Hat 2022.)

In summary, software quality assurance can be defined in a broad sense to cover both the software development process and the maintenance process of a software. In other words, software quality assurance is not limited to the quality of the code behind the software. (Galin 2004, 14–34.) DevOps, which has been characterized as an evolution of the agile movement, integrates software development with operations. The focus of DevOps on operations is not limited to deployment automation but also extends to activities related to software maintenance. (Leite et al. 2019.)

2.2 Information retrieval

2.2.1 IR Concepts and Terminology

As an academic field of study, Manning et al. define information retrieval (IR) as "finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)." The field can also cover other data and information problems, and aspects related to supporting users from a system design perspective in tasks such as browsing a document collection, filtering documents based on desired criteria, and applying further processing on returned results of a query. (Manning et al. 2009, 1–18.)

Practical experimentation and research for information retrieval on computer systems began in the late 1940s (Manning et al. 2009, 1–18). The application of information retrieval was originally limited to the scientific community, and it was later used commercially and in vertical search engines (Shahi 2015, 39–56). The term information retrieval was coined by Calvin Mooers in 1950. The words (information) retrieval and search are often used synonymously, although search is nowadays more commonly used. (Manning et al. 2009, 1–18.)

Applications related to information retrieval include search engines, recommendation systems, document classification and clustering, filtering, and question-answering systems. The retrieved information can come from any source, in any format and volume. Similarly, information retrieval can be applied to almost any domain. (Shahi 2015, 39–56.) Commercial applications such as web search engines have made information retrieval escape the confines of professional use and made it an everyday affair for millions of people (Manning et al. 2009, 1–18).

Manning et al. identify three distinctive scales at which information retrieval systems operate: web search and personal information retrieval are both at the extreme ends when considered by the scale of use, and enterprise, institutional, and domain-specific search (or vertical search) are in the space between. Each scale has its own distinctive issues that need to be addressed, such as the enormous scale of the internet for web search and keeping the search system lightweight and low maintenance in personal information retrieval applications. (Manning et al. 2009, 1–18.)

Vertical search focuses on a specific domain (or a vertical) such as people (LinkedIn), travel reservations (Booking.com) or scientific publications (Google Scholar). The benefits of a vertical search engine include a narrower scope that results in more precise information. A vertical search system is designed to facilitate a specific task or a workflow and to provide users with vertical expertise. While web search engines typically guide users from point A to point B, vertical search engines provide users with the ability to complete an intended action. (Bocskocsky 2020.)

The Association for Intelligent Information Management (AIIM) defines enterprise search as "the practice of identifying and enabling specific content across the enterprise to be indexed, searched, and displayed to authorized users." (AIIM 2023). In enterprise search, data typically has to be indexed from a number of different systems. Additional difficulty emerges from introducing new document formats or systems, and from the use different languages used in the documents, which increases the vocabulary. (Manning et al. 2009, 84, 91, 146.)

Information retrieval systems address different types of information retrieval problems, which can be broken down as tasks that a system must perform. An ad hoc retrieval task is defined by Manning et al. as the most standard information retrieval task where "system aims to provide documents from within the collection that are relevant to an arbitrary user information need, communicated to the system by means of a one-off, user-initiated query." A document is considered relevant it meets the user's information need. (Manning et al. 2009, 1–18.)

The effectiveness of an information retrieval system can be assessed by evaluating the returned results of a query (Manning et al. 2009, 1–18). The primary evaluation metrics are precision and recall. Precision can be described as a measure of quality that measures if the returned results are relevant. Recall can be described as a measure of quantity that measures the fraction of relevant documents that are returned. One of the two metrics can be more relevant depending on the information retrieval problem. (Shahi 2015, 39–56.)

2.2.2 Apache Solr, Full-Text Search and Inverted Index

Apache Solr is an enterprise search platform used for building document retrieval and analytics applications. Its versatile features also enable secondary uses cases related to, for example, data analysis, statistics, and data storage. Solr's features are documented online in the Apache Solr

Reference Guide documentation. Solr is an open-source software licensed under the Apache 2.0 license. (ASF 2023.) Solr is used by a number of companies and popular websites for their search features, including Netflix (ASF 2019).

The history of the Solr platform goes back to the early 2000s. Solr was originally created by Yonik Seeley at CNET in 2004. In 2006, CNET donated the Solr project to the Apache Software Foundation. (Shahi 2015, 1–9). The Apache Solr project was established in 2006, originally as a subproject of Apache Lucene. It was established as its own Top Level Project (TLP) in 2021. Solr is managed by the Solr PMC (Project Management Committee) and developed by contributors to the Apache Software Foundation. (ASF 2023.)

> *"The fundamental premise of Solr is simple. You give it a lot of information, then later you can ask it questions and find the piece of information you want." (ASF, s.a.)*

In Solr terminology, a set of data that describes something is called a document. "In Solr, a Document is the unit of search and index." (Tan, Kelvin 2023). Documents are composed of fields, which contain the document's data. Adding documents to Solr is called indexing, and asking questions about the data is called querying. When a document is added to Solr, the data in the document's fields is added to the index. When a query is performed, Solr searches the index and return documents that match the query. (ASF, s.a.)

The search in Solr is based on full-text search. Full-text search systems are based on indexing data. In Solr, the standard way to build the index is with an inverted index. Inverted index (also referred to as the inverted file) is a keyword-centric data structure that maps content to its location. It contains a list of terms that are used in the documents that have been indexed. (ASF, s.a.; Tan, Kelvin 2023.) The inverted index indicates the documents where the term appears in and the number of documents that contain the term (Manning et al. 2009, 1–18).

When a document is added to Solr, it goes through an analysis process called field analysis, which involves processing steps where transformations are applied to normalize (or simplify) the terms before building the index. Transformations include lowercasing, removing accent marks, and mapping words to their stems. The output of an analysis process is a token stream, and the normalized tokens are added to the index. Analysis also takes place at query time, where the values in the user's query are analyzed and tokenized before searching the index. (ASF, s.a.)

In the information retrieval terminology, the list of terms in an inverted index is called a dictionary. More specifically, the word dictionary can be used to refer to the data structure while vocabulary refers to the set of terms in the dictionary. Each term is indicated a list of documents where the term is used. The list of documents is called a postings list, each item on the list a posting, and all the posting lists together the postings. Depending on the implementation, the list of terms may be sorted alphabetically. (Manning et al. 2009, 1–18.)

Full-text search can be compared to how search works in traditional relational database management systems (RDBMS). In relational search, the database is queried for the searched value using structured queries. Relational search is good at answering precise queries that fit the tabular structure in which the data is stored. (Nithianandan 2009.) However, while most RDBMSs support keyword searching on fields that contain free-form text, the performance is slower, and the search results less relevant than in a full-text search system (Lucidworks 2019).

While relational databases excel at handling structured data, full-text search systems perform well on searching large amounts of text, which can be unstructured, semi-structured or structured. Full-text search systems perform better on larger amounts of data, have richer search capabilities, and have the ability to provide more relevant search results. (Lucidworks 2019.) A full-text search system can also be used to improve the search experience of a service built on an RDBMS, so the two are not mutually exclusive (Nithianandan 2009).

On the technical side, Apache Solr is built on Apache Lucene, an open-source information retrieval library based on the Java programming language (Shahi 2015, 1–9). “The relationship between Solr and Lucene, is like that of the relationship between a car and its engine.” (Tan, Kelvin 2023). Communication with Solr happens via the HTTP protocol, and JSON (JavaScript Object Notation) is the default response format. Solr runs as a stand-alone search server, and it can be scaled across multiple servers by setting up a cluster of Solr servers. (ASF 2023.)

Elasticsearch is another popular search platform with similar features and is also built on the Lucene library. Elasticsearch was originally released in 2010 and is developed by the Elastic company. (Shahi 2015, 1–9.) According to a 2016 comparison by Akca et al., close performance results between the two platforms were obtained in many earlier studies, while more specific use cases

show a greater difference in performance characteristics. To choose between Solr and Elasticsearch, the study suggests a test based on the desired use case. (Akca et al. 2016.)

Elasticsearch started as an Apache 2.0 licensed open-source project. In 2018, it moved to a dual license model with part of the source code remaining under the Apache 2.0 license. In 2021, the licensing model was changed with the Apache 2.0 license being replaced with the Server Side Public License (SSPL) as a countermeasure towards Amazon Web Services (AWS) using its Apache-licensed open-source code to develop a competing service. SSPL is not recognized by the Open Source Initiative as an open-source license. (Coenradie 2021; Elastic 2021.)

3 RESEARCH METHODOLOGY

This thesis project took the form of action research. Action research is a qualitative research method that synergizes research and practice. In the iterative process of action research, the researcher and a practitioner collaborate to diagnose a real-life problem, introduce action intervention by developing a solution, and conduct reflective learning (Avison, D. et al. 1999). A proof of concept for a helper too for software testers was created within this thesis project, attempting to tackle the problem of identifying bugs related to customer-requested code changes.

The project was implemented in four parts. Firstly, a theoretical framework for the project was identified and discussed in a literature review. Literature review is an overview that summarizes and evaluates previously published works on a specific topic (Knopf 2006). Reasons for conducting a literature review include demonstrating the author's knowledge about a particular subject (proof of knowledge) and informing the author of influential researchers and research groups in the field (identifying a research family) (Randolph 2019).

The literature review resulted in a better understanding about key topics related to the project: information retrieval and software quality assurance. Existing literature for the literature review was searched primarily on Google Scholar, a web search engine for scholarly literature. Keyword searches on Google was an effective way to discover recent and relevant online sources on specific topics. An important source was the Apache Solr Reference Guide, an online documentation for the Apache Solr search platform by the Apache Software Foundation.

Secondly, data collection and analysis were conducted for quantifying the problem. The data was collected by using a querying feature in the digital task management system used at the workplace. The results of the data collection were compiled in a tabular form for analysis. Because some of the data may reveal information about the company's business, the presentation of data had to be considered carefully. To this effect, some of the values are presented in relative terms in the thesis report without revealing the absolute or real number differences.

The data was analyzed by using a mix of statistical methods. Statistics is defined by the Cambridge Dictionary as "a collection of numerical facts or measurements, as about people, business conditions, or weather" (Cambridge Dictionary). The statistical methodology has two main approaches:

descriptive analysis, which summarizes data from a sample or population and inferential analysis, which draws conclusions from the data (Kaur et al. 2018). Both descriptive statistics and inferential statistics were utilized in the backlog analysis.

Thirdly, a proof of concept for the helper tool was described and produced. Describing the helper tool included making the necessary technology choices and justifying them. Producing the helper tool involved various practical phases, which are documented in this thesis report. Fourth, the solution was tested, and the test results were compiled in a tabular form and analyzed. The rationale for choices regarding the test setup and its possible limitations were described. The test results were discussed and ideas for future development were presented.

Data security had to be considered to protect customer and company data during the project. Building the helper tool involved exporting backlog data from the task management system. The backlog export was necessary to provide data for the helper tool. The backlog export contained data such as customer names, and information related to feature requirements and bug reports. Data security was strictly observed when handling the backlog data, and no data was uploaded to an online environment during the project.

4 QUANTIFYING THE PROBLEM

4.1 Backlog Analysis

I work at Innofactor in the Dynasty software development team. The software development team is divided into Scrum teams, which develop and maintain the Dynasty product family. The history of the Dynasty product stretches back to the late 1980s. The company that was developing Dynasty was later acquired by Innofactor, which was founded in 2000 (Innofactor 2022). The development of the current Dynasty 10 series started in 2016 and since then it has grown into a product suite used by hundreds of customers.

The growth in business has increased the number of people working directly in product development. At the same time, product development processes have evolved to reflect the larger scale of work. The quality assurance process has become more extensive and bug reporting practices on the backlog have become more refined. For example, new tags have been introduced in the product backlog to distinguish different types of work. The tags make it easier to create queries to search for work items in the product backlog.

A high-level analysis of the past allocation of work items to customer projects and internal product development gives insight into the need to improve the identification of customer work. The review period for the analysis was 2019–2022. Only full years were considered because the ratio between customer work and internal product development fluctuates over the year. While customer-requested features are developed throughout the year, most of the new features that are part of internal product development are developed during the last quarter of the year.

The data for the analysis was obtained by using queries in the product backlog in Azure DevOps, which is the task management system used by the product development team. Queries is a feature that can be used to search for and filter items in the backlog using different fields, such as tags or effort, as criteria. Challenges to the analysis were posed by the high number of work items and evolving backlog practices over the years. The Azure DevOps queries used in the analysis will be saved in the system, allowing others to continue, refine or verify the analysis.

Table 1 shows the percentage of PBIs and bugs that fall into the categories of customer projects and internal product development. The queries included PBIs or bugs that were finished in a given year. In this case, finishing covers both coding and software testing phases as it was not feasible to remove the latter phase. Therefore, work items that were coded but not tested each year do not appear in that year's figures. In addition, PBIs that were not related to coding and thus did not cause bugs were filtered from the queries. The comparison is based on the number of work items.

Table 1. The ratio of completed work between customer projects and internal product development for PBIs and bugs based on the number of work items

Year	PBI (number of items)		Bug (number of items)	
	Customer projects	Product development	Customer projects	Product development
2019	44%	56%	35%	65%
2020	50%	50%	42.5%	57.5%
2021	57%	43%	47.5%	52.5%
2022	53%	47%	42%	58%
Average	51%	49%	42%	58%

Table 1 shows that in the four-year period 2019–2022, the share of development work related to customer projects was on average 51%. In the same period, the share of bugfixes related to customer projects was on average 42%. The trend of under-allocation of bugs to customer projects stands out consistently each year. The ratio of finished work between customer projects and internal product development for bugs is higher for internal product development while the opposite is true for PBIs.

The effort field can also be used as a basis for the analysis as seen in Table 2. Each PBI is given an effort by developers before starting work. The effort is a numerical estimate, which is measured in person-days (e.g. PBI that has the value 1 in the effort field is estimated to take one person-day to complete). Because bugs are not efforted, the table shows the ratio of bugs based on the number of work items whereas for PBIs the ratio is based on the effort field. The table only shows the years 2021 and 2022 as the effort field was used more inconsistently in the previous years.

Because of evolving backlog practices, the effort field was also occasionally empty in PBIs from 2021 and 2022. PBIs that were not efforted were removed from the calculations. This affected the ratio of work between customer projects and internal product development for PBIs. In 2021, the number of PBIs that had an effort estimation was 9.8% higher for PBIs related to customer projects, and in 2022, 5.6%. The figures were adjusted accordingly by subtracting the differences mentioned above from the effort of PBIs related to customer projects.

Table 2. The ratio of completed work between customer projects and internal product development for PBIs and bugs based on effort for PBIs and the number of work items for bugs

Year	PBI (effort)		Bug (number of items)	
	Customer projects	Product development	Customer projects	Product development
2021	64%	36%	47,5%	52,5%
2022	62%	38%	42%	58%
Average	63%	37%	45%	55%

When PBIs are considered based on the effort field, the ratio of work related to customer projects increases. While 55% of the number of PBIs were related to customer projects in 2021 and 2022, as shown in Table 1, 63% of work was related to customer projects when the comparison is based on the effort field. This indicates that PBIs related to customer projects were typically larger in terms of estimated effort. This information is important because larger code changes typically produce more bugs.

It should also be investigated how many bugs a typical PBI produces. Knowing the ratio of bugs per PBI affects the expected number of reported bugs for each category. Table 3 shows that the average number of bugs caused by the development of a PBI for a customer project was 1.4 and for internal product development 1.8. The number is calculated by dividing the number of bugs reported (but not necessarily fixed) each year for each category with the total number of PBI's completed (the sum of PBIs in both categories) in that year.

Table 3. The ratio of bugs per PBI for customer projects and internal product development

Year	Customer projects	Product development	Difference
2019	0.9	1.7	0.8
2020	1.4	1.8	0.4
2021	1.5	1.7	0.2
2022	1.4	2	0.6
Average	1.3	1.8	0.5

Table 4 shows the ratio of bugs per person-day. As in Table 2, only the years 2021 and 2022 are shown because of the more inconsistent use of the effort field in the previous years. The table shows that the average number of bugs caused by 1 person-day of PBI development for a customer project was 0.85 and for internal product development 1.05. The number is calculated by dividing the number of bugs reported (but not necessarily fixed) each year for each category with the total effort of PBIs completed (the sum of PBIs in both categories) in that year.

Table 4. The ratio of bugs per person-days for customer projects and internal product development

Year	Customer projects	Product development	Difference
2021	0.9	1	0.1
2022	0.8	1.1	0.3
Average	0.85	1.05	0.2

Table 3 shows that the ratio of bugs per PBI was on average 0.5 bugs smaller for PBIs related to customer projects in 2019–2022. Similarly, table 4 shows that the ratio of bugs per person-days was 0.2 bugs smaller for PBIs related to customer projects in 2021 and 2022. This indicates that PBIs related to customer projects typically produce less bugs. However, this contradicts the earlier interpretation that PBIs related to customer projects, which are on average larger in terms of estimated effort, should produce more bugs.

The smaller bug ratio for PBIs related to customer projects could explain why less bugs are assigned to customer projects even though more PBI work is related to customer projects, as shown in tables 1 and 2. On the other hand, the smaller bug ratio for customer projects could also be explained by the fact that bugs were not always correctly assigned to customer projects. The under-

allocation of bugs to customer projects would inflate the number of bugs related to internal product development, resulting in a higher bug ratio for that category.

Overall, comparisons based on estimated person-days (tables 2 and 4) are considered more reliable. The ratio of completed PBI work between the two categories, as shown in table 2, also corresponds to what was known before the analysis. The difference in the ratio of bugs per person-day, as shown in table 4, is very small (0.2 bugs on average). Assuming that 1) PBIs related customer projects produce more bugs and 2) bugs were under-allocated to customer projects, the ratio of bugs per PBI should in reality be somewhat higher for PBIs related customer projects.

The ratio of completed PBI work between the two categories, based on estimated person-days, can be used as a basis to estimate what the ratio should have been for bugs. Table 2 shows that in 2021 and 2022, 63% of completed PBI work was related to customer projects whereas only 45% of bugfixes were assigned to customer projects. Assuming that the ratio of bugs per PBI is higher for PBIs related the customer projects, more than 63% of bugfixes in 2021 and 2022 should have been assigned to customer projects.

This leads to the question of whether 18% more work could have been charged from customers in 2021 and 2022. Of course, the overall picture is not that simple, because organizing the development work and customer projects requires collaboration between two business units. These challenges will be covered in the next chapter. Nevertheless, the backlog analysis confirms that bugs have been under-allocated to customer projects each year, resulting in technical debt that has to be fixed under internal product development.

4.2 Understanding the Bigger Picture: Customer Projects

The product development unit cooperates with the project unit, which manages the customer projects. The project unit orders new features in the context of customer projects, and the product development team in the product development unit creates these new features while also doing internal product development. Because the total amount that the company can earn from a customer project is usually known in advance, the question is more about how the money is allocated within the company, between business units.

Of course, a bug that cannot be assigned to the customer project that caused it, or any other customer projects, is ultimately a cost to the company. If the budget of a fixed price customer project is exceeded, for example because of bugfixes that weren't considered in the project's estimated completion time, the excess has to be paid by the supplier. If these excess costs cannot be assigned to the customer project, the unit that has to use its own employee resources for the bugfixes takes the hit from a financial perspective.

Balancing customer projects with internal product development involves various challenges. Because customer projects operate within a budget, there exists an incentive to keep the project's costs down by limiting and prioritizing the number of bugfixes that can be assigned to it. A project that is close to going over its budget may only be able to accept critical bugfixes whereas non-critical bugs must be left outside the scope of the project. This technical debt must be fixed as part of internal product development, which means less capacity for developing new features.

Project delivery dates also affect the prioritization of bugfixes. Several customer projects are open at the same time, and the development capacity that can be allocated to one project is limited. A tight schedule in one project might leave time for only critical bugfixes in addition to the development of new features. A bugfix that can't be assigned to the project that caused it cannot necessarily be assigned to other projects either. As a result, it may become technical debt that competes for the capacity reserved for internal product development.

The customer perspective should also be considered. A customer may have a quality expectation different from that of the supplier. Sometimes the customer's quality expectation for the product could be lower. For example, dependencies and implications related a specific code change, such as adding a new feature, may not be understood. Ignoring these effects could degrade the overall quality of the product and result in technical debt. For a product used by hundreds of customers, it is important to keep the overall quality of the product in mind.

Before a bugfix can be considered to be added to a customer project, the bug must be found and determined if the bug is likely caused by a customer-related code change (a new feature or another bugfix). The prerequisites are therefore 1) timely discovery of bugs and 2) identification of those bugs that were caused by the development of customer-requested features. Fulfillment of these conditions enables informed decisions to be made, even if a bugfix cannot always be assigned to the customer project that caused it.

An effective SQA process is in a key role. If new changes are not tested timely, bugs are left unfound and technical debt is created. The relationship between a bug and a customer-requested change must be identified while the customer project is still active. A bugfix cannot be charged from the customer project if it was found or fixed after the completion of the project. However, knowing the total amount of technical debt caused by different customer projects and internal product development provides value in itself because it enables informed decisions.

One practice is to assign a bug to a customer project if it was found during the testing of a customer-requested feature and is related to the same functionality. However, not all bugs are found at once. If a bug surfaces later and is not related to the feature being tested, the software tester has to find out what change caused it to determine whether to assign the bug to a customer project or internal product development. This can be time-consuming, and the answer is not always obvious, especially if a lot of time has passed since the development.

Deciding on which bugs to fix is part of the SQA process, which must meet business requirements. Although the quality of the product is an important consideration, the quality must be financially sustainable. It might not bet worthwhile to fix all the bugs, and whether a bugfix can be assigned to a customer project is an important consideration. However, further analysis of challenges related to assigning bugfixes to customer projects extend beyond the scope of this thesis project, which is focused on the problem of identifying work items that caused the bugs.

5 SETTING UP THE PROOF OF CONCEPT SOLUTION

5.1 Defining the Solution

This chapter provides a high level description of the proof of concept helper tool and a rationale for the technology choices.

5.1.1 High Level Description of the Helper Tool

The problem that this thesis project attempts to tackle is the identification of bugs that have been caused by customer-requested code changes. The proposed solution is a helper tool for software testers. The helper tool will look for similarities between the new bug report and other work items in the backlog. The comparison of similarities between items will be based on text fields such as title, description, and tags. The problem of looking for similarities between work items based on text data is an information retrieval problem.

The helper tool could be used both on new bugs where the connection to a customer project is unclear, or older bugs where a connection to a customer project was not discovered. It would prevent project bugs from becoming technical debt that competes for the capacity reserved for internal product development. It could also be used to check if a bug has already been reported to avoid duplicate bug reports. Other benefits of the helper tool would include gaining a better insight to the amount of technical debt that is caused by specific customer projects.

The goal of this thesis project is to create a proof of concept realization of the helper tool. Proof of concept is a small-scale realization of a method or an idea to demonstrate its feasibility (IATE 2023). In practical terms, the goal is to install an Apache Solr search server and test if the search platform can be configured to find meaningful connections between work items in the product backlog. The search would take a bug report as an input and turn it into a query, and the search results would list matching work items and other relevant information.

The relevancy of the search results will be analyzed by testing, the scale of which depends on the available time for the project. For example, a basic test would be to see if the helper tool can identify

already known relationships between pairs of items. The success rate would be the percentage of correctly identified relationships for a set of test items. A relationship could be considered as being correctly identified if it was listed in the top x search results. Additionally, the performance of Solr's search should be compared to the search feature available in Azure DevOps.

The product backlog will be exported from Azure DevOps manually as a file and fed to Solr for indexing. It should be noted that automating the process of updating changes from Azure DevOps to Solr would be necessary for making the solution an everyday tool. However, building such a data pipeline is outside the scope of the proof of concept realization. Depending on the amount of time available for this project, the backlog could be imported to Solr in its entirety, or a smaller portion of it could be chosen based on a specific time frame.

5.1.2 Technology Choices

Apache Solr was chosen as the search platform for this project mainly for three reasons. Firstly, the author had previously conducted a desktop study by researching the platform for another project and was thus familiar with Solr concepts and terminology. Secondly, there is existing Apache Solr expertise available among my colleagues, allowing me to seek guidance and support in the technical aspects of the project. Thirdly, Solr is licensed under the permissive Apache 2.0 license, which makes it a more economical and a future-proof choice.

Solr can be installed and deployed in various ways, such as downloading a Solr distribution from the official website as a binary package and installing it manually on a compatible local machine or a server, or running Solr as a Docker container, on a virtual machine, or on cloud platforms such as Amazon Web Services and Microsoft Azure, which may provide preconfigured virtual machine images with Solr installed. Different installation and deployment options for Solr are discussed in the Apache Solr Reference Guide online documentation.

The deployment method chosen for this proof of concept is running Solr as a Docker container. Docker is a platform used to develop, ship, and run container applications. Container is an isolated environment that contains the source code, operating system libraries and all the dependencies needed to run an application. Containers offer the same functionality as virtual machines, but are

lighter, faster, and more resource efficient. (IBM, s.a.; Docker, s.a.) Two official Docker images are provided for each release, corresponding to the binary distributions (ASF, s.a.).

Docker was chosen as the deployment method because it makes distributing and testing the proof of concept solution straightforward. Once the solution is considered ready for testing by other people, the Docker image can be delivered to another computer that has Docker installed. The difference between a Docker container and a Docker image should be clarified. Docker image is a template that contains the instructions for creating a Docker container while a Docker container is a running instance of a Docker image. (Docker, s.a.)

Solr can be run either as a standalone server or as a multi-node cluster. For the latter option, there are two approaches: SolrCloud, which is a cluster that uses Apache Zookeeper for central coordination of the Solr nodes, and user-managed mode, where a Solr cluster can be operated without central coordination. For this proof of concept solution, it was decided to run Solr as a standalone server. In this simpler approach, Solr is run in a Docker container with a single core that stores the data in a local directory inside the container. (ASF, s.a.)

Once the backlog data has been indexed in Solr, the data can be queried. As a recap, the search would take a bug report as an input and turn it into a query. In a more rudimentary solution, a software tester would interact with Solr by using the query builder feature in the Solr Admin UI web interface. However, this would involve manual work because the software tester would have to build the query by hand. A more convenient solution for everyday use would involve creating a Python script that serves as a command-line interface (CLI) tool for interacting with Solr.

The Python script would take one or more bug reports as an input and print the Solr search results as an output. The input parameter for the script could be the ID number of a work item if it was already indexed, or file location of a bug report, for example a CSV file, if it was not already indexed. After receiving the input, the script would interact with Solr to retrieve relevant documents. The output of the script could be in the format of, for example, a printed list or a report file, and it could be customized to display the search results, a relevance score, and other information.

This proof of concept focuses on evaluating whether Solr can find relevant connections between work items, and less on how software testers interact with Solr. Therefore, test queries will be made using the Solr Admin UI web interface. Because the proof of concept solution is based on a one-

time backlog export, the test queries will be based on already indexed bug reports. New bug reports cannot be used because there would be a knowledge gap between the export date and the date of discovery of the new bug.

If the helper tool was ever be taken into practical use, there would have to be support for using new bug reports as the basis of search. In a more advanced approach, new changes in the Azure DevOps backlog would be automatically indexed in Solr through a data pipeline. In a more rudimentary solution, for example during a trial period during which the helper tool is evaluated, the bug report would be extracted from a file. In the latter approach, new changes in the Azure DevOps backlog would have to be manually exported and indexed at regular intervals.

To summarize, a proof of concept for a helper tool will be set up with the goal to look for similarities between work items in a product backlog. The helper tool will be based on the Apache Solr search platform, which will run inside a Docker container. The proof of concept for the helper tool will get data from a one-time backlog export from Azure DevOps. Solr's ability to find relevant connections between work items will be tested by making manual test queries. The helper tool could be distributed to software testers as a Docker image.

5.2 Installing Solr and Making Queries with Sample Data

The first step was to install the Docker Image of Solr. Before starting work on the proof of concept solution, Solr's query features were tested with sample data to get a basic understanding of working with Docker containers and Solr.

The following steps were completed to install the Docker image:

1. Installing the Docker Desktop application.
2. Installing the latest Solr image (version 9.4.0) by running the following command in PowerShell: docker pull solr

Next, the Docker container was configured and started with Docker Compose. Docker Compose is a tool, which is included in the Docker Desktop application. With Docker Compose, a YAML file can be used to configure the services for an application. The services can be created and started from the configuration file with a single command.

The following steps were taken to configure and start the Docker container:

1. Writing a docker-compose.yml configuration file for Docker Compose. An example configuration was provided in the Apache Solr Reference Guide online documentation.
2. Starting the Docker container by running the following command in PowerShell: docker-compose up -d

After starting the container, the Solr Admin UI web interface (Figure 1) could be opened by accessing localhost:8983 on a web browser.

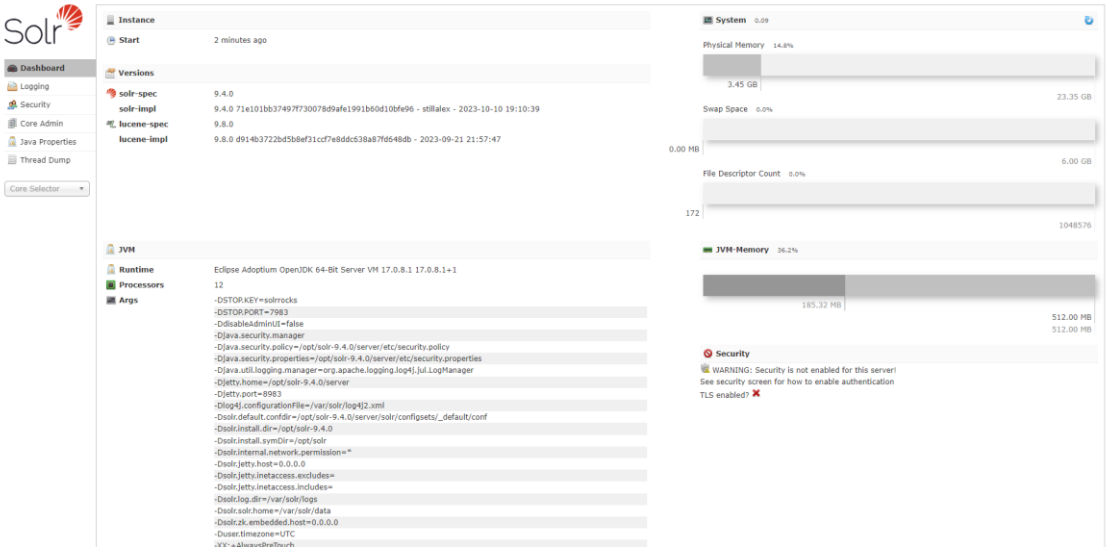

Figure 1. First look at the Solr Admin UI web interface.

Solr needs a core to be able to search and index. In Solr terminology, the term core refers to a single index and its associated configuration files and transaction log. When running Solr as a multi-node cluster, the term collection is used to refer to one or more cores. (ASF, s.a.) However, this proof of concept solution uses the standalone mode. Creating a core can be skipped at this stage because the example configuration used in the previous step includes the solr-precreate command, which prepares the core.

Next, some sample data was added to the index:

1. Locating the sample data that is included in the official Solr image. The data is stored in the /opt/solr-9.4.0/example/exampledocs folder.
2. Adding sample data to the index by using the bin/post tool in PowerShell.

When the sample data was indexed, Solr automatically created a schema based on the indexed data.

In Solr terminology, “schema is the place where you tell Solr how it should build indexes from input documents”. Solr offers two approaches for managing the schema: a managed schema that uses the automatically generated managed-schema.xml file, and a manually maintained schema.xml file. The managed schema mode supports features such as the Schema API, which allows using a HTTP API to manage the schema, and Schemaless Mode, which is a set of features that enable indexing documents without explicitly defining a schema beforehand. (ASF, s.a.)

Making modifications to the schema can be skipped at this stage because sample data is being used.

Finally, it was confirmed that the sample data was successfully indexed by querying it.

1. Selecting the gettingstarted core in the Admin UI web interface and navigating to the Query tab in the left panel.
2. Completing the default search for *:*, which requests all the documents in the index. Alternatively, the search could be conducted in PowerShell by using the curl command and the URL provided in the query builder.

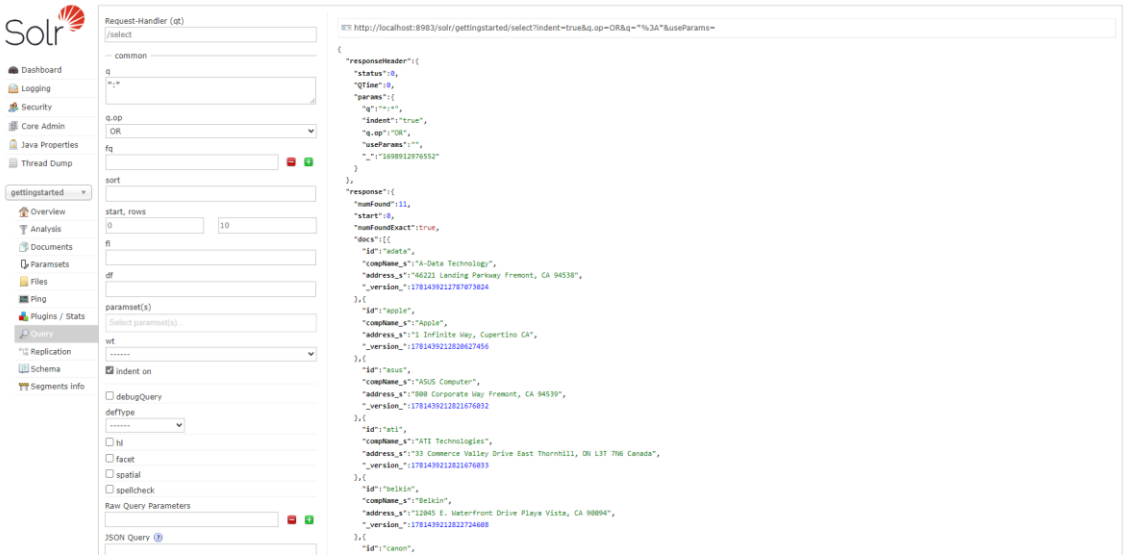

Figure 2. The query tab has a query builder interface, which can be used to build a query by making selections in the user interface. The URL that contains the query is displayed at the top; it can also be used with the curl command. The query results are shown on the right in a JSON format.

Completing the steps in this section provided a basic understanding of working with Docker containers, the Solr Admin UI web interface, how to add data to Solr, and how to query the data. Next, it will applied to build the proof of concept for the helper tool.

5.3 Exporting and Cleaning the Backlog Data

The backlog data had to be first exported from Azure DevOps before it could be uploaded to Solr for indexing.

The first step was to decide the criteria for exporting data from Azure DevOps. The following questions had to be considered:

- What should be the time interval for the exported work items?
- Which work item types should be included in the data set?
- Which fields should be included in the data set?

A three year time interval was chosen for the backlog export. The time interval for the backlog export is different from the backlog analysis, which was discussed earlier in this thesis report, where the review period was from the beginning of 2019 to the end of 2022. This difference is irrelevant because the goal of the backlog analysis was to study the previous allocation of bugfixes between customer projects and internal product development, whereas the goal of the backlog export is to provide data for the helper tool.

Commonly used Azure DevOps work item types by the Dynasty development team are Product Backlog Items and Bugs, and less common item types are Epics, Features, and Tasks. It was decided to include only those work item types that result in code changes: Product Backlog Items, Bugs, and Tasks. The number of fields chosen for the backlog export was somewhat high. A total of 17 fields were selected. Attention was paid to include fields that were informative or assumed to be relevant for the helper tool to be able to find relationships between work items.

The backlog data was exported from Azure DevOps as a comma-separated values (CSV) file and cleaned up in Excel:

1. Running the Azure DevOps query that displays the specified items.
2. Using the Export to CSV option on Azure DevOps to export the data as a CSV file.

3. Opening the CSV file on Excel and reviewing the data. The exported CSV uses semicolon as delimiter, which caused data to be merged into the same fields. The Text Import Wizard feature was used to select the delimiter and display the data correctly.
4. Removing corrupted rows. Some of the rows were cut and divided into multiple rows, likely due to some pattern in the text content. In total, 129 corrupted rows were removed from a total of 18167 rows, which resulted in a loss of 0,7%.

When inspecting the text data in the CSV file, it was noticed that the text formatting used in Azure DevOps was preserved in the backlog export, which resulted in HTML tags in the exported CSV file. Examples of text formatting include bolding, italics, bullet points, and hyperlinks, which are represented in HTML using tags such as for bolding, for italics, for bullet points, and <a href...> for hyperlinks. To prevent the HTML tags being indexed in Solr along with the text content, the HTML tags had to be removed.

To remove the HTML tags from the CSV file, a Python script was created using the ChatGPT AI system:

1. Describing the issue and the desired result to ChatGPT.
2. Copy-pasting the AI generated script to the Visual Studio Code editor, making corrections to the code, and installing the Beautiful Soup library on the local machine for parsing and removing the HTML tags.
3. Running the Python script and reviewing the cleaned data in the new CSV file that was generated. Removing the HTML tags halved the file size from 58 MB to 23 MB.

The backlog export and data cleaning resulted in a CSV file with 18039 rows and the file size of 23 MB.

5.4 Uploading the Backlog Data to Solr for Indexing

After exporting the backlog data from Azure DevOps as a CSV file, the data could be uploaded to Solr for indexing.

The first step was to prepare the schema before indexing the CSV file:

1. Opening the Schema page in the Solr Admin UI web interface.

2. Adjusting fields in the schema, such as setting the field type to text_fi for fields that contain Finnish text. By using a language-specific field type, Solr can use the specific analysis and tokenization rules related to that language for indexing and querying.

Next, the CSV file was uploaded in Solr and added to the index:

1. Copying the CSV file from the local machine to the /var/solr/data directory in the Docker container by running a command in PowerShell.
2. Adding the data in the CSV file to the index by using the bin/post tool in PowerShell.
3. Making a test query in the Solr Admin UI web interface to check that the data was indexed correctly.

Indexing the CSV file resulted in 18037 documents being created in Solr. A single file generated multiple documents because Solr treats each row in the CSV file as a separate document. The number of documents generated by Solr may also differ from the number of rows in the original file, which is explained by how Solr indexes the document. Solr may generate multiple documents from a single row as part of transformations that happen during the indexing process. Solr's behavior during indexing can be adjusted depending on the use case and requirements.

5.5 Constructing the MoreLikeThis query

The MoreLikeThis feature was used to test Solr's ability to find relationships between work items. More specifically, it was used to see if Solr could identify the work item that caused the bug. More-LikeThis uses the contents of a given document to construct a query for finding similar documents in the index. The feature can be used in several ways, the most common of which is using it as a request handler. Request handler is a component that processes incoming requests to Solr and maps requests to specific functionalities, such as MoreLikeThis. (ASF, s.a.)

The first step was to configure the MoreLikeThis request handler, as it was not configured on by default. This was accomplished by using the Config API for making changes to the configuration and a simple curl command, which was provided in the online documentation. After enabling the request handler, the MoreLikeThis feature could be used by including the mlt=true parameter in the query request. The standard way to send a query request to Solr is using a HTTP GET request that is represented by a URL (Uniform Resource Locator).

The next step was to construct a MoreLikeThis query for identifying relationships between work items. Figure 4 displays an example query that provides a bug ID as a parameter and uses the MoreLikeThis feature to find similar product backlog items. The returned product backlog items also have a smaller ID number, because the change caused the bug was, of course, completed first. The query used in this project was similar. However, some fields have been left out so that information about the company's development process is not revealed.

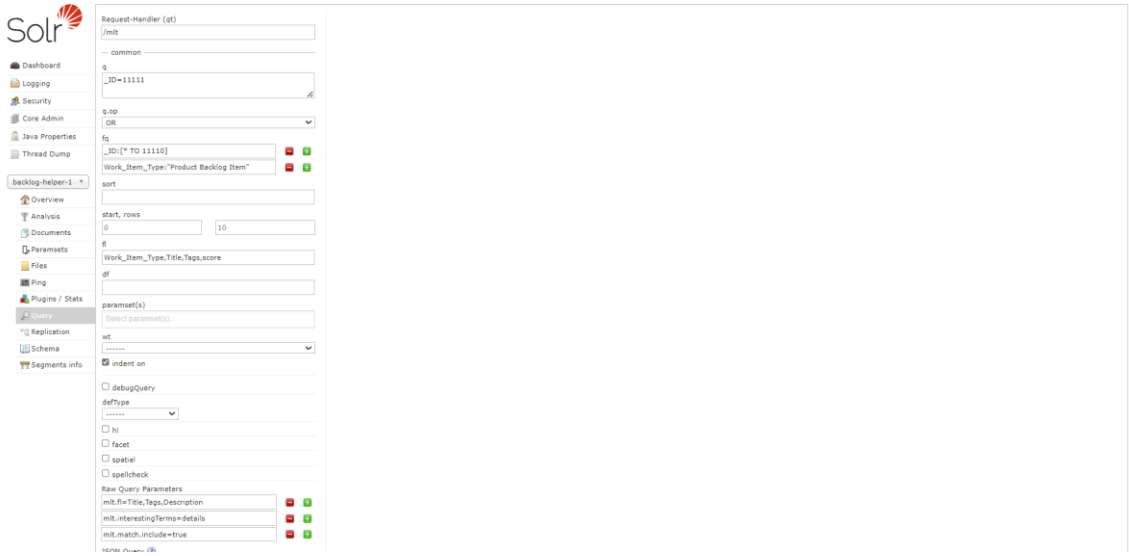

Figure 3. An unexecuted MoreLikeThis example query that takes the ID number of a bug as a parameter and uses the MoreLikeThis feature to find similar product backlog items that have a smaller ID number.

Executing the query in Figure 4 creates a URL that consists of following information:

- **http://localhost:8983/solr/backlog-helper-1**: The base URL, which specifies a Solr server running on localhost on port 8983, and a core named backlog-helper-1.
- **/mlt**: The MoreLikeThis query handler, which is the endpoint for the query within the back-log-helper-1 core.
- **q=ID:11111**: The query parameter, which specifies that the query looks for documents (for example, a bug report) that have the value 11111 in the ID field.
- **fl=ID,Work_Item_Type,Title,Tags,score**: The field list parameter, which specifies that the fields Work_Item_Type, Title, and Tags should be included in the response, as well as the relevance score of any matching documents.
- **fq=_ID:[* TO 11110]&fq=Work_Item_Type:"Product Backlog Item"**: The filter query parameter, which is used twice to specify that the work items that the query returns should

have an ID with a maximum value of 11110 and the work item type should be “Product Backlog Item”.

- **mlt.fl=Title,Tags,Description**: A parameter which specifies that the fields Title and Description should be used for the MoreLikeThis query.
- **mlt.interestingTerms=details**: A parameter that displays in the response the terms that contribute to the similarity between documents, useful for understanding the relevance scoring.
- **mlt.match.include=true**: The matched document will be displayed in the response.
- **indent=true**: Formats the response with indentation, which is useful for readability.

A query can be constructed in the Query tab in the Solr Admin UI by setting the necessary parameters and values. When Solr generates a URL that represents the HTTP GET request for the query, it goes through URL encoding where special characters are converted into legal characters within the US-ASCII character encoding standard. A query can be executed in the query editor or, for example, by entering its URL in the address bar of a web browser and pressing enter. The query results are presented in JSON format.

6 RESULTS

6.1 Describing the Test Set Up

Work item pairs were chosen for analyzing Solr's ability to find relevant connections. In a nutshell, the goal was to test if Solr could identify the product backlog item related to the new feature that caused the bug by including the product backlog item in the search results. The number of documents in the query response was limited to 10, which is the default. If the work item that caused the bug was not included in the top 10 documents, the order of which is determined by the relevance score, it was interpreted that Solr could not identify the connection.

A set of 100 work item pairs was chosen where the product backlog item that caused each bug was already known. In practical use, the helper tool would be used to discover connections that are not yet identified. However, for testing purposes it makes sense to rely on known connections that have been reliably identified by software testers. In the Azure DevOps backlog, a relationship between two or more work items is indicated by a link. These links were not included in the backlog export, which is why Solr couldn't use them to cheat in identifying the relationship.

While the business case comes from identifying relationships between work items related to customer projects, the test set only contained work items related to internal product development. A relationship between two work items may also be indicated by text data that is that is added in the bug. This text data cannot be removed because it is mixed in more than one fields, which also contain other relevant data for the query. This issue affects both categories, but less so for work items related to internal product development.

It was deliberately chosen to have the work item pair include one bug and one product backlog item, and not two bugs. While bugs are typically caused by product backlog items, which represent new features additions, bugs can also be caused by bugfixes where a developer fixed something but broke something else. The tasting was scoped to the former case because it is more common but potentially more challenging as it includes comparison between different fields, some of which are unique to each work item type.

A manual query was conducted on each bug in the test set. The same standard query was used for all bugs, ensuring that the same fields were considered in the searches. The query results were reviewed individually to determine if they contained the product backlog item that was known to cause the bug. A table was compiled that included information about the query results, such as the ranking of the correct product backlog item and its relevance score, and the number of interesting terms included in the MoreLikeThis query response.

A limiting factor in testing the MoreLikeThis feature was the inability to configure it to use stop words. Stop words are insignificant terms that appear frequently in a data set. Stop words should be filtered out so that they do not degrade the quality of the search results. For example, a work item may use a template that has standard parts that are not removed when filling out the template. A list of stop words was gathered using the mlt.mindf and mlt.interestingTerms parameters. However, the stop words could not be implemented in time for this project.

Wildcard searches were also considered for testing Solr's search capabilities. Wildcard search is a feature that can be used on a single term to add one or more characters in the beginning, middle or end of that term. The question mark (?) character is used to represent one character and the asterisk (*) character is used to represent any number of characters. For example, search string sol* would return search results such as solace, solemn, and solr. Wildcard searches are useful in an inflected language such as Finnish, where cases are applied to words.

To use wildcard searches effectively, one would involve more work because one has to first identify relevant keywords for the query. For example, if a bug that is related to a specific feature, relevant terms related to that feature would be selected. This involves manual work, as opposed to a MoreLikeThis query that identifies the relevant terms automatically. Additionally, wildcard searches are also supported in the functional work item search in Azure DevOps. Due to these factors, it was decided to limit the testing to the MoreLikeThis feature.

6.2 Analysis of Test Results

Table 5 shows the test results. A MoreLikeThis query was run on 100 work item pairs where the product backlog item that caused the bug was known in advance. The query took the ID number of the bug report as an input, and the MoreLikeThis feature was used to find product backlog items

with a smaller ID number and a similar text content. The query responses were inspected to see if they included the correct product backlog item. The hypothesis was that similarity of text data in work items could be used to identify the work item that caused the bug.

Table 5. The compiled results of testing the proof of concept solution on 100 work item pairs. Matched means that the product backlog item that caused the bug was included in the query response.

Category	Amount	Average ranking	Average relevance score	interestingTerms amount
Matched	21	2.5	10.4	19.6
Not matched	79	-	-	10.8
Average	-	-	-	15.2

The test results indicate that in 21% of the test queries, the query response included the product backlog item that caused the bug. The average ranking of the correct product backlog item was 2.5, meaning that it was typically in the top results. This indicates that the number of results in the query response could be limited to fewer than 10. The average relevance score of the correct product backlog item in the query response was 10.4, indicating a possible threshold to consider in a real-world use of the helper tool.

The number of interesting terms returned by the MoreLikeThis query was also recorded. Queries that included the correct product backlog items in the search results returned, on average, around 20 interesting terms in the interestingTerms section in the query response. On the other hand, queries that did not include the correct product backlog item in the search results returned and average of around 11 interesting terms. This indicates that the search results become more relevant when the MoreLikeThis query has more terms for the similarity comparison.

Analysis of the interesting terms returned by the interestingTerms parameter produced some observations. Because the stop words list could not be configured in time for this project, the interesting terms list often included individual terms that were irrelevant, such as words used in a bug template or generic terms and phrases often used in a bug report. The relevancy of the search results could be improved by implementing a stop words list. When Solr is taught which words to ignore, more relevant terms can be used for similarity comparison.

However, a long list of interesting terms did not always produce the correct product backlog item in the search results. Of the 79 queries that did not return the correct product backlog item, 11 query responses included 20 or more interesting terms. In a few cases, the interestingTerms section was clogged by words from an error message, which were copy-pasted in the bug report. In most cases, the interesting terms were mostly relevant but simply did not produce the correct product backlog item in the search results.

A brief analysis of the bug reports suggests that longer bug reports resulted in more interesting terms in the query response and a higher match rate. This would make sense because Solr would have more terms to work with when finding similar work items. In retrospect, it would have been interesting to systematically compare the bug reports and see whether aspects such as a higher word count and better documentation, such as using more descriptive language, correlate with a better match rate.

The testing assumed that each relationship between work items pairs in the test set was correctly identified. Solr was then tested to see if it could find these relationships. However, it should be kept in mind that the linking between work items does not always represent an absolute cause and effect relationship. Sometimes a link between two work items could mean that a bug related to a specific feature was found when a code change related to that feature was being tested. In reality, the bug could have been caused by another code change related to a different work item.

Additionally, a cause and effect relationship is only one type of relationship. The search results likely included other types of relationships that were not identified before. For example, the search results for a bug report may include not only the work item that caused the bug, and also other work items that represent previous developments related to the same feature. However, the search results were not reviewed for other types of relationships, which were outside the scope of this project and would have required additional work.

Overall, the test results were promising. With somewhat minimal configuration, Solr could identify the correct work item in around one fifth of the test queries. The accuracy of the search results could be improved with further configuration, such as implementing a comprehensive stop words list. The testing produced useful information for optimizing the helper tool, such as a stop words list collected from the query responses, and the average ranking and relevance score of correctly identified product backlog items.

7 DISCUSSION

A proof of concept for the helper tool was built on the Apache Solr search platform running in a Docker container. The proof of concept was tested by choosing a set of work items and using the MoreLikeThis feature in Solr to find similar work items. The results show that Solr would include the correct work item in the search results in around one fifth of the test queries. The results were promising, considering that a cause and effect relationship is a specific type of relationship between work items and that Solr was used with somewhat minimal configuration.

While the test results for the proof of concept were promising, its limitations should also be kept in mind. While the accuracy of the search results could be improved by further configuration, the search-based solution cannot suggest relationships with absolute certainty. The search results will have to be reviewed by software testers who would apply their practical knowledge and experience. Some level of automation could be added in the process, for example by offering automatic suggestions on new bug reports.

For the helper tool to be taken into practical use, further development is required. Firstly, the data would have to be synchronized between Solr and the task management system. The proof of concept is based on a one-time backlog export. The data could be synchronized automatically or manually. In a more advanced approach, new changes in the backlog would be automatically indexed in Solr through a data pipeline. In a more rudimentary solution, new changes in the backlog would have to be manually exported and indexed at regular intervals.

Secondly, the user experience of the helper tool should be improved. In the proof of concept, the user has to interact with Solr by using the query builder feature in the Solr Admin UI web interface or by editing the URL manually. Based on initial feedback from software testers, interacting with the query parameters may prove to be too technical. A more convenient solution could be, for example, a Python script that serves as a command-line interface tool for interacting with Solr, taking the bug ID as a parameter, and returning the desired search results.

The practical contributions of this project can be summarized as follows. The backlog analysis resulted in a deep understanding of the issue, confirmed the business need for the helper tool, and provided an operating model for further analysis. Building the proof of concept for the helper tool

and testing it with real backlog data provided promising results, which can be considered when making decisions about possible future development of the helper tool. The proof of concept also showcased the search capabilities of Solr, which can be applied to solve other problems.

This project also contributes to the literature by exploring the utilization of Apache Solr's search capabilities to find further value in a product backlog. Depending on the organization or the business setup where the utilization of Solr is considered, the value may come from identification of bugs related to customer-requested code changes, identification of relationships between work items, understanding of what factors contribute to technical debt, or another way to leverage Solr's advanced search as well as analytical capabilities.

Opportunities for professional development and learning were used during the project. The backlog analysis presented an opportunity to conduct data analysis and apply statistical methods. Researching the theoretical framework provided the author good background knowledge about the fields of software quality assurance and information retrieval. Building the helper tool deepened the author's technical skills, for example in the use of the Apache Solr search platform and Docker containers.

The ChatGPT artificial intelligence system by OpenAI was used in the project. Provided enough context in the prompt, the chatbot could suggest ways to achieve desired results and solutions to technical problems. The ability to provide context made using the chatbot a convenient method for seeking information and solutions. The author used the AI system in a limited capacity that did not compromise company or customer data. Healthy source criticism was used and knowledge of the chatbot's limitations kept in mind when evaluating the responses.

The use of artificial intelligence tools could be an avenue for future experiments and research related to the problem of finding connections between work items in a product backlog. For example, AI chatbot extensions in popular task management system is a likely future scenario. Provided that the AI chatbot has access to the backlog data and advanced search capabilities, using natural language to find relevant work items would provide a superior user experience. In this scenario, that data would not have to be exported from the task management system.

Solr's analytical capabilities could be explored to find further value in the indexed backlog data. Solr's client APIs (application program interface) also enable client applications to interface with

Solr, similar to a Python script that could serve as a command-line interface tool for interacting with Solr. For example, data visualization libraries, such as Matplotlib, could be used to draw graphs that are not available in the used task management system, or a machine learning model could be trained to find patterns in the data or perform classification tasks.

8 CONCLUSION

This thesis project tackled the problem of identifying bugs that have been caused by customer-requested code changes. The proposed solution was a helper tool for software developers, which could be used to find the work item related to the code change that caused the bug. In order to quantify the problem, quantitative data collection and analysis was carried out. A high-level analysis of the past allocation of work items to customer projects and internal product development confirmed the need for solutions such as the helper tool.

A proof of concept for the helper tool was produced. Firstly, the official Docker image of Apache Solr was installed. Criteria for the backlog export was defined, after which the backlog data was exported from Azure DevOps as a CSV file. The data was cleaned up by removing corrupted rows and HTML tags. The Solr schema was prepared, after which the CSV file was uploaded to Solr and added to the index. Lastly, the MoreLikeThis feature was enabled, and a query was constructed for testing the proof of concept for the helper tool.

The helper tool takes the ID number of a bug report as an input and conducts a MoreLikeThis query for similar work items. A set of work items was chosen to test whether it could find those work items that caused the bugs. For this purpose, 100 pairs of bugs and product backlog were chosen where the product backlog item that caused each bug was already known. A manual query was conducted on each bug in the test set, and the search results were reviewed to see if the correct product backlog item was included. The test results were compiled in a table and analyzed.

The test results for the helper tool were promising. Solr included the correct work item in the search results in around one fifth of the test queries. The accuracy of the search results could be improved with further configuration, for example by implementing stop words. Before the helper tool could be taken into practical use, further development is required. For example, the backlog data would have to be synchronized between Solr and the task management system, and the usability would have to be improved to make it an everyday tool.

SOURCES

Akca, M. A.; Aydoğan, T; Ilkuçar, M. 2016. An analysis on the comparison of the performance and configuration features of big data tools Solr and Elasticsearch. International Journal of Intelligent Systems and Applications in Engineering, 4 (Special Issue), Pages 8–12. Search date 31.8.2023. https://www.researchgate.net/publication/311916747_An_Analysis_on_the_Comparison_of_the_Performance_and_Configuration_Features_of_Big_Data_Tools_Solr_and_Elasticsearch

ASF (Apache Solr Foundation). Apache Solr Reference Guide. Search date 31.10.2023. Apache Solr Reference Guide :: Apache Solr Reference Guide

ASF (Apache Software Foundation) 2019. PublicServers. Search date 10.9.2023. https://cwiki.apache.org/confluence/display/SOLR/PublicServers

ASF (Apache Software Foundation) 2023. Apache Solr project information. Search date 27.8.2023. https://solr.apache.org/whoweare.html

Association for Intelligent Information Management (AIIM) 2023. What is Enterprise Search? Search date 27.8.2023. https://www.aiim.org/What-is-Enterprise-Search

Avison, D., Lau, F., Myers, M. & Nielsen, P.A. 1999. Action Research. Communications of the ACM. Volume 42, Number 1 (1999), Pages 94–97. Search date 13.8.2023. https://dl.acm.org/doi/pdf/10.1145/291469.291479

Bocskocsky Andrew 2020. The rise of vertical search engines. Search Engine Watch. Search date 10.9.2023. https://www.searchenginewatch.com/2020/11/13/the-rise-of-vertical-search-engines/

Cambridge Dictionary. Statistics. Search date 21.10.2023. https://dictionary.cambridge.org/dictionary/english/statistics

Catolino, Gemma; Fabio, Palomba; Zaidman, Andy; Ferrucci, Filomena (2019). Not All Bugs Are the Same: Understanding, Characterizing, and Classifying Bug Types. Journal of Systems and

Software. Volume 152, June 2019, Pages 165-181. Search date 4.11.2023. https://www.sciencedirect.com/science/article/abs/pii/S0164121219300536

Coenradie, Jettro 2021. Let's talk about the Elastic license change. Luminis. Search date 2.9.2023. https://www.luminis.eu/blog/search-en/lets-talk-about-the-elastic-license-change/

Docker. Docker overview. Search date 31.10.2023. https://docs.docker.com/get-started/overview/

Dyba, Tore; Dingsøyr, Torgeir 2008. Empirical studies of agile software development: A systematic review. Information and Software Technology, 50 (9–10) (2008), pp. 833-859. Search date 24.10.2023. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=652db5cfe6f3c217710f40b62800d878a2183016

Elastic 2021. FAQ on 2021 License Change. Search date 2.9.2023. https://www.elastic.co/pricing/faq/licensing

Galin, Daniel 2004. Software Quality Assurance: From theory to implementation. Pearson Education Limited. Search date 26.10.2023. https://ds.amu.edu.et/xmlui/bitstream/handle/123456789/14867/Software%20Quality%20assurance%20-%20617%20pages.pdf?sequence=1&isAllowed=y

Grottke, Michael; Trivedi, Kishor S. 2005. A Classification of Software Faults. Supplemental Proc. Sixteenth International IEEE Symposium on Software Reliability Engineering, pages 4.19–4.20. Search date 9.11.2023. https://www.researchgate.net/profile/Michael-Grottke/publication/228894619_A_classification_of_software_faults/links/5ede5a4592851cf13869833e/A-classification-of-software-faults.pdf

IBM. What is Docker? Search date 31.10.2023. https://www.ibm.com/topics/docker

IATE (Interactive Terminology for Europe) 2023. IATE - Entry ID 1903199. Search date 13.8.2023. https://iate.europa.eu/entry/result/1903199/en

Innofactor 2022. Innofactorin tarina. Search date 20.8.2023. https://www.innofactor.com/fi/yritys/tarinamme/

Kaur, Parampreet; Stoltzfus, Jill; Yellapu, Vikas 2018. Descriptive Statistics. International Journal of Academic Medicine 4(1): p 60-63, Jan–Apr 2018. Search date 21.10.2023. https://journals.lww.com/ijam/fulltext/2018/04010/Descriptive_statistics.7.aspx

Knopf, Jeffrey W. 2006. Doing a Literature Review. PS: Political Science & Politics, Volume 39, Issue 1, January 2006, pp. 127–132. Search date 22.10.2023. https://core.ac.uk/download/pdf/81222467.pdf

Leite, Leonardo; Rocha, Carla; Kon, Fabio; Milojicic, Dejan; Meirelles, Paulo 2019. Survey of DevOps Concepts and Challenges. ACM Computing Surveys, Vol. 52, No. 6, Article 127. Search date 3.11.2023. https://arxiv.org/pdf/1909.05409.pdf

Manning, Cristopher D.; Raghavan Prabhakar; Schütze Hinrich 2009. An Introduction to Information Retrieval. Cambridge University Press. Search date 3.9.2023. https://ds.amu.edu.et/xmlui/bitstream/handle/123456789/14697/Book%20558%20pages.pdf?sequence=1&isAllowed=y

Mishra, Alok; Otaiwi, Ziadoon 2020. DevOps and Software Quality: A Systematic Mapping. Computer Science Review. Volume 38, November 2020, 100308. Search date 29.10.2023. https://www.sciencedirect.com/science/article/pii/S1574013720304081

Nithianandan, Amit 2009. Solr and RDBMS: Designing your application for the best of both. Lucidworks. Search date 3.9.2023. https://lucidworks.com/post/solr-and-rdbms-the-basics-of-designing-your-application-for-the-best-of-both/

Lucidworks 2019. Full Text Search Engines vs. DBMS. Search date 2.9.2023. https://lucidworks.com/post/full-text-search-engines-vs-dbms/

Randolph, Justus 2019. "A Guide to Writing the Dissertation Literature Review," Practical Assessment, Research, and Evaluation: Vol. 14, Article 13. Search date 22.10.2023. https://scholarworks.umass.edu/pare/vol14/iss1/13

Red Hat 2022. What is CI/CD? Search date 4.11.2023. https://www.redhat.com/en/topics/devops/what-is-ci-cd

Shahi, Dikshant 2015. Apache Solr: A Practical Approach to Enterprise Search. Springer Science. Search date 27.8.2023. https://k0d.cc/storage/books/Data-bases/Apache%20Solr/Apache%20Solr.pdf

Schwaber, Ken; Sutherland, Jeff 2020. The Scrum Guide. The Definitive Guide to Scrum: The Rules of the Game. ScrumGuides.org. Search date 22.10.2023. https://scrum-guides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf

Tan, Kelvin 2023. Basic Solr Concepts. Search date 31.8.2023. https://solrtutorial.com/basic-solr-concepts.html