Study trends in code smells in microservices-based architecture, Compare with monoliths

AMIT PURI

FINAL REPORT

MAY 2021

ABSTRACT

Background

The rapid adoption of Microservices-based architecture and its predecessor Service-oriented Architecture's influence on most software development affects code quality unprecedentedly, with every redesign, rewrite, and refactoring efforts in the brownfield projects. Even a thoughtful attempt in the paradigm shift to microservices is iterative, massive investment, and code rework that attracts technical debts. Technical debt is inevitable; controlling and reducing the impact is the only alternative. Moreover, owing to several factors

- challenging deadlines, lead time to market, cost constraints,
- ignoring warnings, bugs, and code smell from the static code analysis tools,
- the porting code to a more recent version of programming language or framework adds to more code smells, anti-patterns,
- and security vulnerabilities.

Few organisations are resorting to No-code/Low-code platforms as SaaS to avoid churn. No-code/low-code platform providers are leveraging microservices and serverless architecture for building their platforms and products. Hence, few organisations are paying for these SaaS platforms instead of building their custom solutions.

Methods

A systematic study of code smells from the public datasets acquired from other research work on the monolith codebases and prepare a dataset for Microservices projects with static code analysis tools to get code smells. There are artefacts created in Microservice architecture for Infrastructure as Code (IaC) for CI/CD pipelines and containerisation. These artefacts are also a candidate for code smells, as researched by few. This study factor in Dockerfile, YAML, and other IaC artefacts for code smells. Perform exploratory data analysis of code metrics data from research work in monolith software and data collection from microservices-based code repositories to generate code metrics. These code metrics would undergo systematic data analysis, feature engineering, and machine learning model evaluation to study the patterns, the significance of code metrics, and analysis with no-code/low-code platforms to provide recommendations over microservices/monoliths.

Findings

Data class, large class and long method are no more significant code smell found in microservices than monoliths, while unnecessary/unutilised abstraction and long statement continue to remain as significant contributors to code smell in microservices. The magic number code smell remains indifferent in monolith and microservices codebases.

Deficient encapsulation, cyclic-dependent modularisation and complex method and broken hierarchy are significantly less or none in microservices.

Conclusions

Microservices are comparatively less prone to code smells on GitHub public repositories, but not code smells free and expect higher code smells in private repositories codebases.

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

HTTP – HyperText Transfer Protocol

API – Application Programming Interface

SMEs – Small and Medium-sized Enterprises

ML – Machine Learning

ESB – Enterprise Service Bus

IaC – Infrastructure as Code

EDA – Exploratory Data Analysis

SVM – Support Vector Machine

NN – Neutral Network

LOC – Lines of code

KLOC – Thousands lines of code

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 Background of the Study

Software quality degrades with the degree of non-conformance to the requirements. Is every need covered in the business requirement specifications? Is it practically feasible to document all non-functional requirements? Are these take precedence over value-driven functional requirements? Software "-ilities" are the most ignored requirements in the service industry over the quality of operational requirements – the application should work; it does work most of the time! Otherwise, debug, fix, and release a hotfix. An ongoing process till there is a new buzzword in the market like Microservices to overhaul the architecture. That is the reality of most software applications, so there is a code smell that is a topic of many researchers to prevent bad smell in the codebases or proactively detect it and eliminate them.

Several researchers have investigated the software applications' change-proneness and analysed the open-sourced projects' commit history. (Palomba et al., 2013) Whereas there has been a study on the evolution of bad smells in objected-oriented code discussed several design problems over a while due to maintenance activities. (Chatzigeorgiou and Manakos, 2010) The statistical analyses of various refactoring concluded that the long method, feature envy, god class, and state checking smells are a few active code-smells. Rewriting of code causes a behaviour change, whereas refactoring preserves the behaviour. There is a significant impact on the design with the cumulative effect of successive refactoring, despite aimed to simplify.

Moving away from monoliths to microservices-based architecture is/was an opportunity for the industry to reset the code smells metrics, write clean code, and improve the overall code quality. Microservices are increasing in popularity, being adopted by several companies, including SMEs and big players such as Amazon, LinkedIn, Netflix, Spotify, and SoundCloud. (Taibi et al., 2020) Martin Fowler describes microservice architecture as "an approach to developing a single application as a suite of small services, each running in its process and communicating with lightweight mechanisms, often an HTTP resource application programming interface (API)." (Fowler, Martin, 2014) Every new architectural style calls for revamping the software applications, learning curve, unlearning old style, and systematic effort to benefit from the recent paradigm change.

Moving to the cloud and rearchitected into microservices is another massive opportunity but with another learning curve. There is no silver bullet in software engineering. (Brooks, F., 1987) It applies in the case of microservices. There is research work in which the researchers have collected evidence of bad practices by interviewing developers experienced with microservice-based systems to identify microservice-specific bad smells. They then classified the bad practices into 11 microservice bad smells frequently considered harmful by practitioners. (Taibi and Lenarduzzi, 2018) These 11 bad smells are proper sets of classification for this study work for future investigation.

This project studies code metrics that influence increasing code smells in polyglot microservices, how No-code/Low-code platforms(Woo, 2020) are emerging as an alternative. Exploratory data analysis, feature engineering like handling outliers, categorical imputation, feature split or scaling of different code metrics, and ML technique can help classify and correlate code smells from monoliths and microservices codebases, and programming languages.

These code metrics categorised in the quality dimensions of size, complexity, coupling, encapsulation, and inheritance in table 23 https://link.springer.com/article/10.1007/s10664-015-9378-4/tables/23. (Arcelli Fontana et al., 2016)

Compare with Microservices projects factoring anti-patterns, pitfalls, (Moha et al., 2010) 11 microservice bad smells, namely: wrong cuts, hard-coded endpoints, cyclic dependency, too many standards, API versioning, Inappropriate service intimacy, shared libraries, ESB usage, shared persistency, microservice greedy and not having an API gateway.

This study also factors code smells in Infrastructure as Code (IaC) (Schwarz et al., 2018) and Dockerfile smells (Wu et al., 2020) in Microservice codebases.

1.2 Problem Statement

Technical debt affects software maintainability in the long run, and researchers are keen on detecting code smells using machine learning techniques. The code smells are categorised in Bad Smells in Software – a Taxonomy and an Empirical Study (Vanhanen, 2014) as the bloaters, the object-orientation abusers, the change preventors, the dispensables, and the couplers.

Static code analysis tools can catch these code-smells, but there is subjectivity in the developer's interpretation of those code smells. There are two different categories of code smell detection: rule-based factors and different metrics in various scenarios that define a set of rules—other approaches based on machine learning techniques that are the main metrics oriented. There are no study factors in both monolith and microservices code metrics; compare them to analyse with EDA with a machine-learning algorithm to understand its significance.

The detection of code smells in monolith software had been an important research topic in software engineering. The researcher and practitioner had employed several machine-learning-based techniques to classify code smell or not. They had used multi-label classification algorithms like decision tree, random forest, Naïve Bayes, SVM, NN. (Kiyak et al., 2019). Another research conducted a large-scale study of 32 different machine learning algorithms to four types of code smell, i.e., Data Class, Large Class, Feature Envy, and Long Method. (di Nucci et al., 2018) The empirical benchmark of 16 machine learning techniques for detecting four code smell types by Arcelli (Arcelli Fontana et al., 2016) is the most comprehensive related work. Furthermore, code bad smell detection through evolutionary data mining (Fu and Shen, 2015) and Historical information for smell detection (HIST) approach (Palomba et al., 2013) are two other approaches studied.

Refactoring is a systematic technique used for improving the design of the existing code, moving away from dirty code to clean code. Refactoring techniques, namely composing methods, moving features between objects, organising data, simplifying conditional expressions, simplifying method calls, and dealing with generalisation. Refactoring detection in Refactoring Miner 2.0 tool (Tsantalis et al., 2020) is a related work to assist the code review process; researchers can create refactoring datasets from commit history and see patterns of self-admitted technical debt removal. We have discussed various code smells-related work in the monolith software, and research on microservice of eleven code smells in the literature review.

1.3 Aim and Objectives

This study aims to find common trends in code smells injected by several microservices-based architectures in the brownfield projects and how the code metrics negatively impacted maintainability aspects of the software:

- Investigate code metrics of the monolith software vs microservices-based architecture,
- Identify code metrics that are of significant influence in the modernised software (microservices),
- List out critical drivers for moving away from custom development to no-code/low code platforms.

This study uses an existing set of the dataset and current work by researchers on code smells in monolithic architecture and compares them with code smells in a microservices architecture. This study also considers factors against custom development over low-code/no-code platforms - Microsoft Power Platform, Google AppSheet, and Amazon Honeycode.

1.4 Research Questions

- Does the code quality deteriorate with modernisation investment in the journey to cloud migration?
- What are code metrics predominately impacted, moving from monolith to microservices architecture?
- Are these modernisation glitches paving the way to no code/low code platforms?
- What are those metrics that are in favour of the no-code/low code platforms?

The code metrics are studied with feature engineering and classification to the model-dependent variable categorical in terms of one or more independent variables, using the sigmoid function in logistic regression.

1.5 Significance of the Study

The code smells in microservices-based architecture that significantly affects code quality would help the practitioner factor these in the static code analysis, develop highly maintainable software, understand change-proneness, focus on clean code and testability of the code. This study would have a global impact on software engineering practices.

The containerisation of software to cloud-native, rewritten into microservices, is common. This study would help provide recommendations on the cloud journey. It could also help compare these metrics between monolith, microservices, and no-code/low platforms and make trade-offs in modernisation efforts.

1.6 Structure of the study

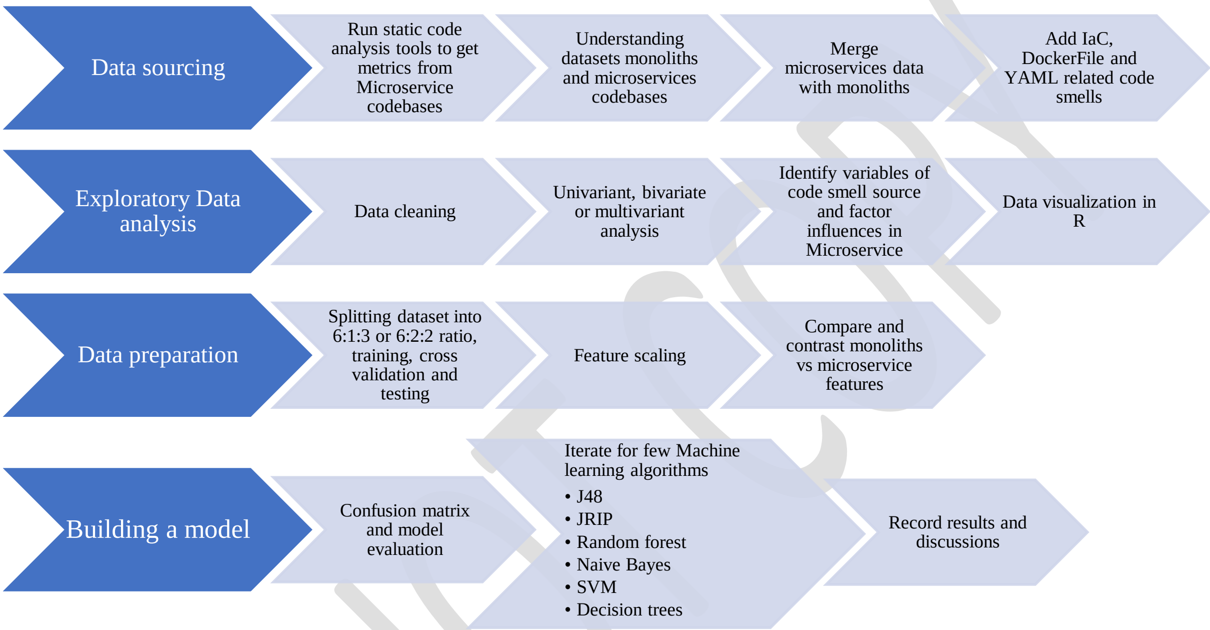

Figure 1.1 structure of the study

CHAPTER 2: LITERATURE REVIEW

2.1 Introduction

Microservices are turning out to be the de-facto architecture in modern enterprise applications. However, a particular section of software professionals recognises that this architecture is not suitable for all scenarios. Based on a study, the monolithic based architecture software has performed better than the microservices in a small number of users < 100 with reasonably lesser application load. The monolithic software has performed at a higher throughput on average, with a fixed number of requests per second in a study. So, monolithic software primarily aims when the developer to handle user requests more quickly. (Al-Debagy and Martinek, 2018)

The microservices architectural style benefits monoliths

- faster delivery,
- agility in smaller teams,
- improved scalability,
- greater flexibility
- and breaking down the complexity monster.

Like monolithic, microservices are also prone to code smells.

Code smells are like bad smells or irregularity within codebases that do not necessarily impact the software's performance or correctness. But the poor programming practices deteriorate program quality in reusability, testability, and maintainability. It is more critical in microservice-based architecture; the benefits of a logically distributed development lifecycle increase the chances of getting code smells if it went unnoticed. Due to microservices' distributed nature, microservice-specific code smells often focus on across modules issues rather than with modules. Traditionally, code-smells detection tools can detect code-smell, but it becomes tougher to handle it in discrete modules if it goes unnoticed during the development process. It amounts to a greater degree of technical debt in a microservices architecture. (Walker et al., 2020)

New areas of code smell in Microservices

CI/CD practice enables faster deployment in the microservices paradigm. Infrastructure-as-code (IaC) is trending as de-facto practice for continuous deployment by defining machine-readable files automation. Terraform is another catalogue of software quality metrics to complex deployment in several microservices, secrets, config maps, YAMLs, Dockerfile, Ansible playbooks. (Dalla Palma et al., 2020)

There is a significant uptake on containerisation via Microservices in cloud applications management. A container in the container technology holds a self-contained, lightweight package. The various parts of an application - presentation layer, middleware, and business logic packaged as containers to run the applications. (Pahl et al., 2017)

Containerisation is a new area for researchers to explore the possibility of avoiding code smells and controlling the maintenance cost for newly written codebases. Such studies would also help in reducing the deployment complexity and making it less error prone.

2.2 Evolving Code and Refactoring

Refactoring is a technique of altering the software's internal anatomy without any changes to the external behaviours. The developers use this technique to address code smells or anti-patterns in the codebase. These code smells are categorised into broader units as the bloaters object-orientation abusers, change preventers, dispensable, encapsulators, and couplers.

This paper concluded that 28% of the researcher applied automatic detection tools for discovering code smells, while 27% of them applied empirical methods to do the same. There are empirical studies that consider many datasets, and those monoliths study highly reveal God class, Long Method, and Feature Envy smell. (Singh and Kaur, 2018)

Bad smells evolution in object-oriented design is also another topic of interest for researchers. Few researchers see it as a problem of inability to design principles adherence, violation of design heuristics, lack of understanding design patterns and appropriate usage of the same or apply anti-patterns. Source code also reflects architectural decisions by recording the design's evolution in the changesets and can be valuable in maintaining maintainability. The previous studies that mainly focused on identifying the refactoring emphasise findings and assumptions regarding the problems themselves and the reasons causing their appearance and removal during software evolution. (Chatzigeorgiou and Manakos, 2010)

Code writing, code removal, class/method removal, and intentional refactoring activity are reasons for eliminating the bad smells. This study also summarises all identified bad smells (long method, feature envy, state checking) for different code projects in bad smell evolution. This study depicts the average time of persistence of a bad smell in the system and examined specific refactoring to remove smells and reasons for code smells like design problems, refactoring activities, and a significant percentage of the problem introduced time.

2.3 Self-admitted technical debt

Self-admitted technical debt is another area of researcher interest. This study (Maldonado et al., 2017) inferred from commit comments of 10 open-source projects, namely Ant, ArgoUML, Columba, EMF, Hibernate, JEdit, JFreeChart, JMeter, JRuby and SQuirrel SQL. These projects' comments were manually classified into specific technical debt types such as design, requirement, defect, documentation, and test debt. 61,664 comments from this dataset (i.e., those classified as design self-admitted technical debt, requirement self-admitted technical debt and without technical debt) are trained the maximum entropy classifier. Then this classifier was used to identify design and requirement self-admitted technical debt automatically.

2.4 Code smell tools

Code smell tools have been developed for high-level design, architectural smells, and language-specific code smells, measuring code smells and the application's quality.

The field of automatic code-smell detection continues to evolve with an ever-changing list of code smells and languages to cover. Code analysis is expected to identify code smells; for instance, stylecheck, stylecop detects the code patterns that resemble a code smell.

In a distributed environment of microservices, there have been multiple code smells identified. In one study, these smells include improper module interaction, modules with too many responsibilities, or a misunderstanding of the microservice architecture.

Code smells can be specific to a particular application perspective, including the communication perspective or the application's development and design process.

The study on Microservice code smells the definition of eleven microservice-specific code smells from a recent exploratory study by (Taibi et al., 2020) and concluded automated tools correctly analysed both testbed systems and successfully identified the applications microservice code smells.

Code smells do not always break the system or cause system-crashing bugs, but they are problems but are poor programming practise indicators. One of the main validity threats is the three code smells microservice greedy, wrong cuts, and too many standards.

While these code smells are defined explicitly as to what they are, they do not have an established system for detection or solution in the Microservices architecture. However, additional hard-coded dependencies in container images might require further research to identify them correctly. Besides, for compiled languages, source code analysis is not possible within a containerised environment.

2.5 Machine learning techniques for code smell detection

An extensive study used 16 different machine-learning algorithms on four code smells (Data Class, Large Class, Feature Envy, Long Method) and 74 software systems, with manually validated code smell samples.

They found that all algorithms achieved high performances in the cross-validation data set. The highest performances were obtained by J48 and Random Forest, while support vector machines achieved the worst performance.

However, the lower prevalence of code smells, i.e., imbalanced data, in the entire data set caused varying performances that need to be addressed in future studies. And the researchers have concluded that machine learning in code smell detection could provide high accuracy (>96 %). Only a hundred training examples are needed to reach at least 95 % accuracy.

2.6 Summary

Most of the studies have concluded a set of code smells prominent in most of the source code available in several GitHub repositories, daily dump available from GHTorrent dataset. The researchers have applied different machine learning techniques for detecting the code smell, comparing several machine learning techniques to find a higher accuracy rate of different machine learning algorithms. 30-60% of the research papers have studied code smell detection tools, machine learning techniques and addressed code smells, i.e. God class, Feature Envy, Long Method, Long parameter list and data class. (Santos et al., 2018) A study on quantifying code smells on maintenance effort uses multiple linear regression analysis to conclude that the code smell is not a significant driver of the effort. A developer can manage to perform the given tasks efficiently, ignoring code smells. (Sjoberg et al., 2013)

There is more weightage of measuring code smell or detection tools, and various approaches around code smell detection than identifying the code smells that could negatively impact the code quality in each architectural paradigm.

CHAPTER 3: RESEARCH METHODOLOGY

3.1 Introduction

This study compares code smells datasets from existing research work and the code smells identified in microservices-based architecture codebases available on GitHub. Also, identify trade-offs of custom development vs low-code and no-code platforms.

This study leveraged the code smell dataset of monolithic software from existing research work that includes various quality dimensions of size, complexity, coupling, encapsulation, and inheritance. Furthermore, try to answer the following questions:

Does the code quality deteriorate with modernisation investment in the journey to cloud migration?

What are code metrics predominately impacted, moving from monolith to microservices architecture?

Are these modernisation glitches paving the way to no code/low code platforms?

What are those metrics that are in favour of the no-code/low code platforms?

It becomes imperative to identify how microservices architecture is different from monolithic in terms of possible code smells that can deteriorate the code quality. Such code quality may be an outcome of a rewrite or refactoring effort. (Fritzsch et al., 2018)

Collate the code metrics from data class, feature envy, god class and long method datasets. Identify only those code metrics that can potentially impact the code quality in the microservices architecture. E.g. the number of classes and methods, number of interfaces, children and implemented interfaces and validate this hypothesis by adding similar metrics from microservices codebases. (Rahman et al., 2019)

There are several studies done using Java, Python codebases. This study would use GHTorrent, and GitHub GraphQL to curate C# projects. Along with GitHub awesome .NET core projects to use sonar runner on these C# projects. Furthermore, measure code smells, vulnerabilities, duplication, and cognitive complexity metrics to determine the extent of the code smell in microservices codebases written in C#. (Sharma et al., 2017)

Also, curate GitHub repositories for microservices-based artefacts as described in these research works on the code smell of Dockerfile and Infrastructure as a Code (IaC). (Schwarz et al., 2018; Dalla Palma et al., 2020; Wu et al., 2020)

Various parameters of the different no-code/low-code platforms like integration, customisation, scalability, ease of use and deployment using Gartner factored into to conclude the study.

3.2 Methodology

To study code smells patterns in monolith software and compare them with microservices, this study uses using the following datasets:

- Monoliths Repos dataset from The Qualitas corpus (Tempero et al., 2010)
- Select Java and C# Microservices Repos dataset from GitHub (Rahman et al., 2019)

The Qualitas corpus is a collection of curated software 112 open-sourced Java systems with 15 systems of 10 or more versions and around 754 total versions of Java systems available in the corpus. This corpus is made available to be intended for empirical studies of code artefacts.

Several static analysis tools were evaluated to conduct this study, including PMD (https://pmd.github.io), SonarQube and a curated list of static analysis tools on GitHub. Furthermore, considering code smells in Java and C# as focus areas of this study, uses Designite (Sharma, 2016), a Software Design Quality Assessment Tool for analysing code smells. This tool detects architectural, design, implementation, methods and types-level metrics.

This study uses the following for code smells analysis in monolith and microservices codebases:

3.3 Data Sourcing: Source code smells data curation

The Qualitas Corpus that uses Perl script unpack the downloaded contents r, e and f distribution located on http://qualitascorpus.com/download that is around 22.9 GB of tar files. So it is about 60+ GB on unpacking using Perl script.

The bash script is used to iterate through 112 open-sourced Java systems located in the systems folder of the corpus downloaded from the previous step. Furthermore, run Designite for Java and C# versions and get code smells metrics in comma-delimited values(CSV) files.

These generated .csv files are collected for code smells in monoliths and microservices codebases written in Java and C# languages.

The cloc Perl script for R is used to get a line of code metrics, installed from https://github.com/hrbrmstr/cloc or https://github.com/AlDanial/cloc

The R script is written for further analysis of code smell metrics that includes

- Read .csv files in R data frames
- Perform exploratory data analysis
- Merge monoliths and microservices data
- Study design and implementation metrics for codebases
- Use GH Archive from https://github-sql.github.io/explorer to curate microservices
- Use the GHTorrent project (Gousios, 2013), about 353 GB of .csv files used with MySQL for GitHub projects, commits, pull request commits, and issues. This study had the plan to use this data for getting insight into commits to fix code smells related bugs.

3.4 Exploratory Data Analysis

- Study trends of the code smell within the various version of codebases available in the corpus.
- Study trends of the code smell across corpus systems.
- Study trends of the code smell at method and type levels.
- Study trends of the code smell in microservices codebases cloned from public git repos.
- Merge monoliths and microservices metrics and study the trends of the code smell.

3.5 Data preparation

The monoliths and microservices are merged to perform data analysis.

CHAPTER 4: IMPLEMENTATION

4.1 Introduction

This study uses Java programming language codebases from Qualitas Corpus to study the code smell trends. Table 4.1 depicts 12 monoliths projects selected for initial analysis to compare with 11 available microservices projects (Java and C# programming languages) from GitHub. These monoliths projects are selected based on the star rating of the project available in the GHTorrent dataset and the top 5 projects in terms of lines of the code. The first and recent available versions are selected for the analysis.

#	Project.Name	version	language	loc
1	antlr	4.0	Java	34359
2	antlr	2.4.0	Java	22504
3	derby	10.6.1.0	Java	619171
4	derby	10.1.1.0	Java	393031
5	hibernate	3.6.9	Java	350490
6	hibernate	0.8.1	Java	3482
7	lucene	4.3.0	Java	423351
8	lucene	1.2-final	Java	7684
9	springframework	3.0.5	Java	322007
10	springframework	1.1.5	Java	103989
11	tomcat	7.0.2	Java	181184
12	tomcat	5.0.28	Java	152043

Table 4.1 LOC Monolith codebases

Sources Lines of code (SLOC) is one of the vital software metrics to qualify software complexity. Still, sometimes it becomes an indicator of the order of magnitude or measure of productivity that could also result in more code smells, more complexity, and increased chances of introducing new bugs. In this study, the Line of code is a factor for comparing monoliths codebases with microservices ones.

There are 3 R script and a bash script written for code implementation, namely:

1. Process-cs-data.R
2. Curate-data.R
3. Analyze-data-func.R
4. Collect-codesmell.sh

The process-cs-data.R file is an R script for processing, analysing and visualising code smell data collected. The curate-data.R script is for running sloc utility to collect a line of code information for monolith and microservices projects. The analyse-data-func.R has required R functions for reading code smells from curated comma-separated values (CSV) files.

The bash script iterates through various folders in the downloaded and extracted folder of the Qualitas corpus to run the Designite tool, collect code smells in the .csv files, and merge those .csv files. There are four different types of code smells generated in .csv files, namely:

1. design code smells
2. implementation code smells
3. method-level code smells
4. class-level code smells

Table 4.2 depicts microservices-based sources for the code smells study.

#	Source	language	LOC
1	LakesideMutual	Java	10675
		YAML	738
		Dockerfile	112
2	microservice	Java	1833
		YAML	107
		Dockerfile	43
3	microservice-consul	Java	1750
		YAML	211
		Dockerfile	41
4	Tap-And-Eat-MicroServices	Java	624
		YAML	111
		Dockerfile	48
5	spring-boot-microservices-example	Java	156
6	cqrs-microservice-sampler	Java	1344
		YAML	252
		Dockerfile	30
7	spring-cloud-microservice-examples	Java	2170
		YAML	287
		Dockerfile	45
8	e-commerce-microservices-sample	Java	566
		YAML	146
		Dockerfile	63
9	spring-cloud-netflix-example	YAML	560
		Java	292
		Dockerfile	52
10	spring-netflix-oss-microservices	Java	560
		YAML	243
		Dockerfile	56
11	ftgo-application	Java	10332
		YAML	1259
		Dockerfile	43
12	spring-petclinic-microservices	Java	1243
		YAML	255

		Dockerfile	26
13	EnterprisePlanner	C#	4179
		YAML	57
		Dockerfile	21
14	eShopOnContainers	C#	43372
		YAML	10444
		Dockerfile	672
15	nhibernate-core	C#	580757
		YAML	457
16	NormanVu-EnterprisePlanner	C#	1959
		Dockerfile	38
17	pitstop	C#	7298
		YAML	2081
		Dockerfile	104
18	vehicle-tracking-microservices	C#	5460
		YAML	244
		Dockerfile	98

Table 4.2 Microservices codebases

4.2 Dataset

As described in the research methodology, the Qualitas corpus and microservices source code from GitHub is used as datasets to extract code smell using the Designite tool and code metrics using cloc. Table 4.3 depicts an output version as comma-separated values (CSV) from Designite. Archtype field to denote monoliths as zero and microservices as one in the last column of the table.

#	Project.Name	version	Code.Smell	Smells	Archtype
1	derby	10.6.1.0	Long Statement	4271	0
2	springframework	3.0.5	Unutilized Abstraction	3463	0
3	derby	10.1.1.0	Long Statement	2577	0
4	lucene	4.3.0	Long Statement	2574	0
5	lucene	4.2.1	Long Statement	2473	0
6	lucene	4.2.0	Long Statement	2471	0
7	hibernate	3.6.10	Unutilized Abstraction	2432	0
8	springframework	3.0.5	Long Statement	2427	0
9	hibernate	3.6.9	Unutilized Abstraction	2421	0
10	hibernate	3.6.8	Unutilized Abstraction	2417	0
11	EaaS	1	Long Statement	240	1
12	EaaS	1	Unutilized Abstraction	136	1
13	dddsample-1.1.0	1	Unutilized Abstraction	87	1
14	dddsample-1.1.0	1	Long Statement	62	1

15	research-modifiability-pattern-experiment	1	Unutilized Abstraction	57	1
16	EaaS	1	Long Parameter List	43	1
17	research-modifiability-pattern-experiment	1	Long Statement	27	1
18	EaaS	1	Deficient Encapsulation	26	1
19	cloud-native-microservice-strangler-example	1	Unutilized Abstraction	25	1
20	cqrs-microservice-sampler	1	Unutilized Abstraction	25	1

Table 4.3 top code smells Monoliths and Microservices codebases

Level	Metrics	Description	Threshold value
Method-level metrics	Lines of code (LOC)	Total Lines of source code in the method	100
	Cyclomatic complexity (CC)	Measures the number of linearly independent paths a source code takes to complete a code execution. It defines the complexity of a program.	8

Table 4.4 Method-level metrics Reference from Designite tool

Level	Metrics	Description	Threshold value
Class-level metrics	Parameter count (PC)	Total count of parameters in the method	5
	Number of fields (NOF)	Total count of internal fields in the class	20
	Number of methods (NOM)	Total count of methods/functions in the class	30
	Number of properties (NOP)	Total count of properties in the class	20
	Number of public fields (NOPF)	Total count of public properties in the class	0
	Number of public methods (NOPM)	Total count of public methods/functions in the class	20
	Lines of code (LOC)	Total lines of code in the class	1000

	Weighted methods per class (WMC)	The sum of cyclomatic complexities of all the methods belonging to the class	100
	Number of children (NC)	Total count of children (sub-classes) of the class	10
	Depth of inheritance tree (DIT)	The maximum inheritance path from this class to the root class	6
	Lack of cohesion of methods (LCOM)	Measures of the cohesion of the class, the correlation between the methods and the instance variables of the class. It is in the range of 0 to 1 and LCOM -1 if type undecidable)	0.8
	Fan-in	Total count of classes that reference as incoming dependencies by the class	20
	Fan-out	Total count of classes referenced as outgoing dependencies by the class	20

Table 4.5 Class-level metrics Reference from Designite tool

4.3 Exploratory Data Analysis

The various code smells are detected using the designite tool in the initial exploratory analysis of the codebases after downloading the Qualitas Corpus and microservices from GitHub.

This screenshot is a popular tool named PowerToys in Microsoft GitHub organisation.

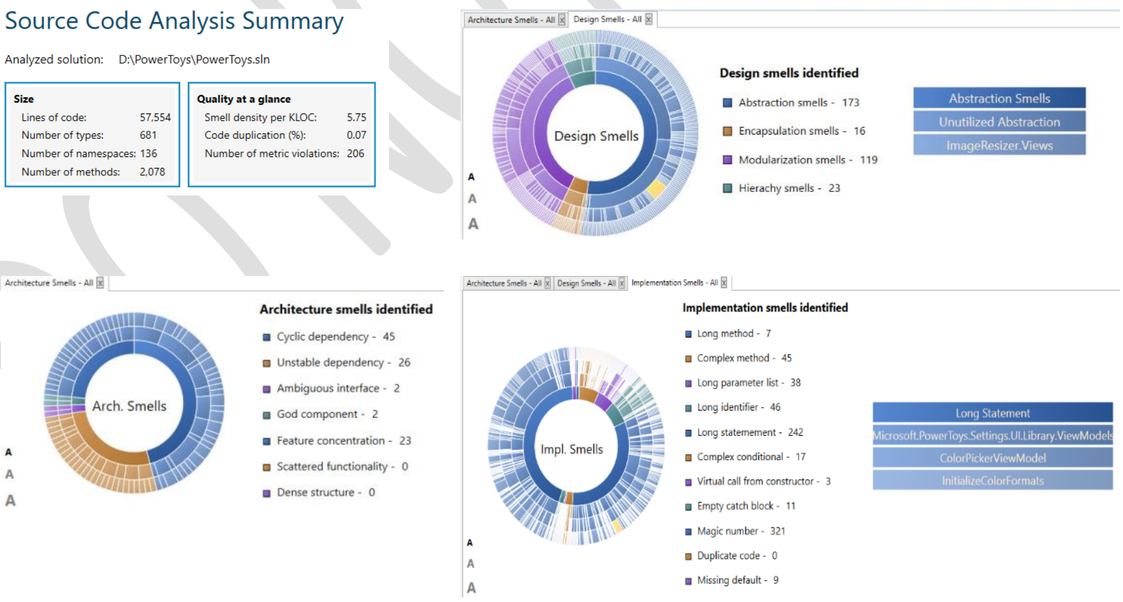

Figure 4.1 PowerToys code smell from Designite tool

The PowerToys tool is a newer codebase of 57KLOC, and significant code smells, mostly in 2-3 digits in this monolith tool codebase.

Notice that complex method and long method are low in number even in the monolith codebases. This pattern could be due to more awareness and consciousness toward the PowerToys tool's maintainability aspect and the use of static analysis in the CI pipeline that the code smells are reduced in the monoliths.

However, another popular codebase of NHibernate monolith codebase of over 261KLOC, with significant code smells. More lines of code, a greater number of code smells, reduced maintainability. All categories of code smells – design, implementation, and architecture, are impacted by increased number of code smells in this case. There is no specific pattern identified in this case other than a more significant number of lines of code.

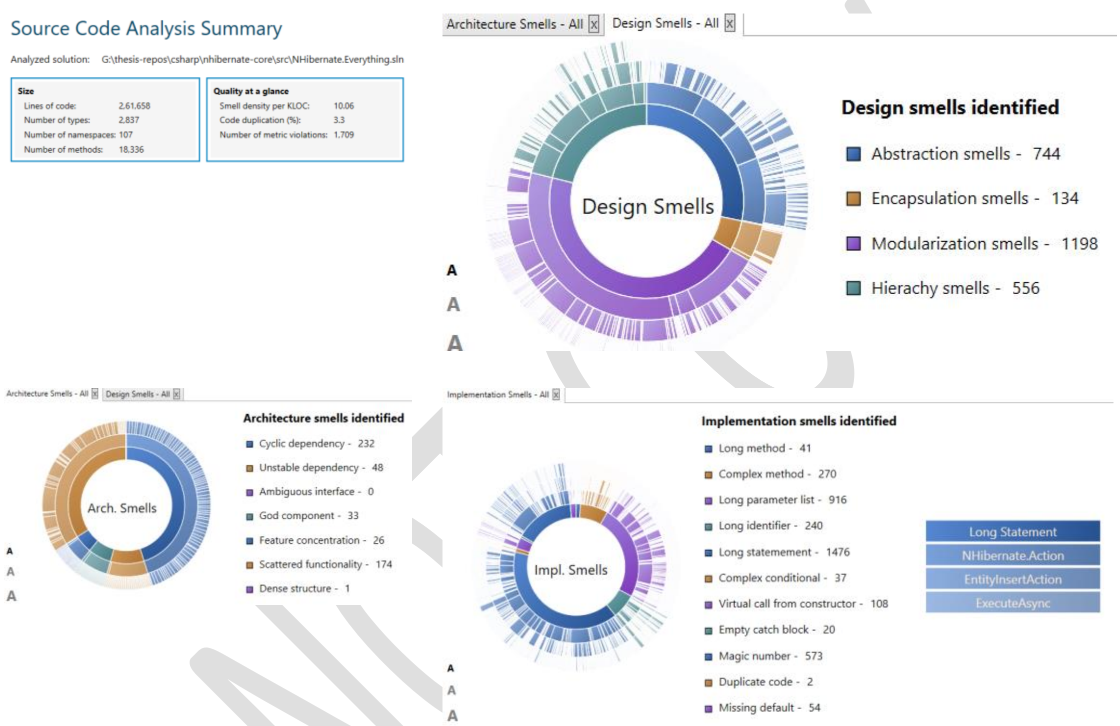

Figure 4.2 NHibernate code smell from Designite tool

Java codebases from the Qualitas corpus are examined for code smells using the Designite tool.

Java LOC = 9663		
#	Code smell	Smells
1	Unutilized Abstraction	67
2	Broken Hierarchy	13
3	Insufficient Modularization	8
4	Deficient Encapsulation	7
5	Cyclic-Dependent Modularization	6
6	Broken Modularization	1
7	Magic Number	40
8	Complex Method	38
9	Long Statement	22
10	Complex Conditional	15
11	Empty catch clause	5

12	Long Method	5
13	Missing default	2
14	Long Parameter List	1

Table 4.6 Project ant 1.1 code smells

Java LOC = 18816 (increased by almost 2x)				
#	Code smell	Smells	Trend	Δx
1	Unutilized Abstraction	135	↑	2x
2	Broken Hierarchy	41	↑	3x
3	Cyclic-Dependent Modularization	18	↑	3x
4	Insufficient Modularization	16	↑	2x
5	Deficient Encapsulation	12	↑	2x
6	Wide Hierarchy	2	New	
7	Broken Modularisation	1	Same	
8	Missing Hierarchy	1	New	
9	Unexploited Encapsulation	1	New	
10	Magic Number	71	↑	2x
11	Complex Method	63	↑	2x
12	Empty catch clause	37	↑	7x
13	Long Statement	36	↑	2x
14	Complex Conditional	24	↑	2x
15	Long Method	8	↑	3+
16	Long Parameter List	5	↑	3+
17	Missing default	5	↑	4+

Table 4.7 Project ant 1.2 code smells

Java LOC = 128434 (increased by more than 6x)				
#	Code.Smell	Smells	Trend	Δx
1	Unutilized Abstraction	843	↑	6x
2	Broken Hierarchy	208	↑	5x
3	Deficient Encapsulation	195	↑	16x
4	Cyclic-Dependent Modularization	185	↑	10x
5	Insufficient Modularization	153	↑	9x
6	Unnecessary Abstraction	46	New	
7	Unexploited Encapsulation	13	↑	13x
8	Wide Hierarchy	11	↑	5x
9	Broken Modularization	9	↑	9x
10	Missing Hierarchy	6	↑	6x
11	Multipath Hierarchy	6	New	
12	Rebellious Hierarchy	6	New	
13	Imperative Abstraction	3	New	
14	Hub-like Modularisation	2	New	
15	Multifaceted Abstraction	2	New	
16	Long Statement	540	↑	15x

17	Magic Number	396	↑	5x
18	Complex Method	369	↑	15x
19	Complex Conditional	230	↑	9x
20	Empty catch clause	196	↑	5x
21	Long Parameter List	86	↑	17x
22	Long Method	43	↑	5x
23	Missing default	40	↑	8x
24	Long Identifier	22	New	

Table 4.8 Project ant 1.8.4, 23rd versions code smells

Java LOC = 22504 (more than 2x loc from ant 1.1 version)				
#	Code.Smell	Smells	Trend	Δx
1	Unutilized Abstraction	68	↑	1+
2	Cyclic-Dependent Modularization	51	↑	8x
3	Broken Hierarchy	46	↑	3x
4	Deficient Encapsulation	32	↑	4x
5	Insufficient Modularization	25	↑	3x
6	Missing Hierarchy	8	New	
7	Unexploited Encapsulation	7	New	
8	Broken Modularisation	5	↑	1+
9	Multipath Hierarchy	3	New	
10	Unnecessary Abstraction	2	New	
11	Complex Method	127	↓	-3x
12	Magic Number	89	↓	-4x
13	Missing default	89	↑	2x
14	Long Method	54	↑	11+
15	Complex Conditional	50	↓	-5x
16	Long Statement	46	↓	-12x
17	Long Parameter List	23	↓	-4x

Table 4.9 Project antlr 2.4.0 code smells

Java LOC = 34359 (around 1.5x more loc from antr 2.4.0, v1)				
#	Code.Smell	Smells	Trend	Δx
1	Unutilized Abstraction	191	↑	3x
2	Deficient Encapsulation	187	↑	6x
3	Broken Hierarchy	168	↑	4x
4	Cyclic-Dependent Modularization	146	↑	3x
5	Insufficient Modularization	49	↑	2x
6	Missing Hierarchy	10	↑	2+
7	Unexploited Encapsulation	10	↑	3+
8	Unnecessary Abstraction	9	↑	4x
9	Hub-like Modularization	2	New	
10	Rebellious Hierarchy	2	New	
11	Wide Hierarchy	2	new	
12	Long Statement	318	↑	6x

13	Magic Number	267	↑	3x
14	Complex Method	80	↑	2x
15	Long Parameter List	64	↑	3x
16	Complex Conditional	44	↓	-6
17	Long Identifier	20	New	
18	Missing default	13	↓	-6x
19	Long Method	6	↓	-7x
20	Empty catch clause	5	New	

Table 4.10 Project antlr 4.0 22nd version code smells

#	Code.Smell	monoliths	Microservices	Trend	Δx
1	Unutilized Abstraction	191	416	↑	2x
2	Long Statement	318	385	↑	-67
3	Magic Number	267	226	↓	41+
4	Long Parameter List	64	56	↓	8+
5	Broken Hierarchy	168	39	↓	129+
6	Deficient Encapsulation	187	37	↓	150+
7	Long Identifier	20	33	↑	-13
8	Cyclic-Dependent Modularization	146	31	↓	115
9	Empty catch clause	5	18	↑	-13
10	Unnecessary Abstraction	9	13	↑	-4
11	Complex Method	80	11	↓	69+
12	Complex Conditional	44	6	↓	38+
13	Missing default	13	5	↓	8+
14	Imperative Abstraction	0	3	↑	-3
15	Insufficient Modularisation	49	3	↓	46+
16	Long Method	6	3	↓	3+
17	Broken Modularisation	5	2	↓	3+
18	Rebellious Hierarchy	2	1	↓	1+

Table 4.11 monoliths vs microservices combined code smells trends

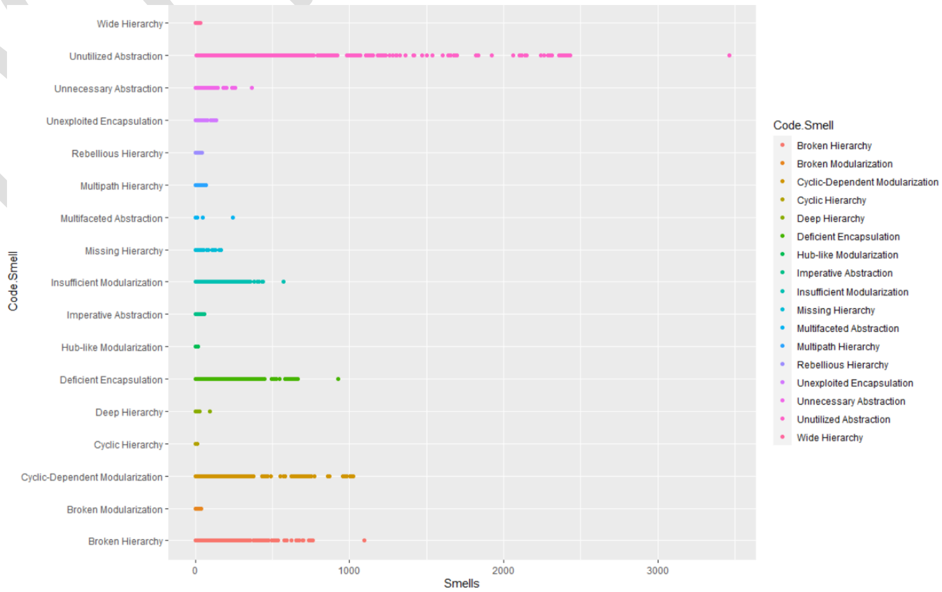

Figure 4.3 combined monolith codebases from the Qualitas Corpus

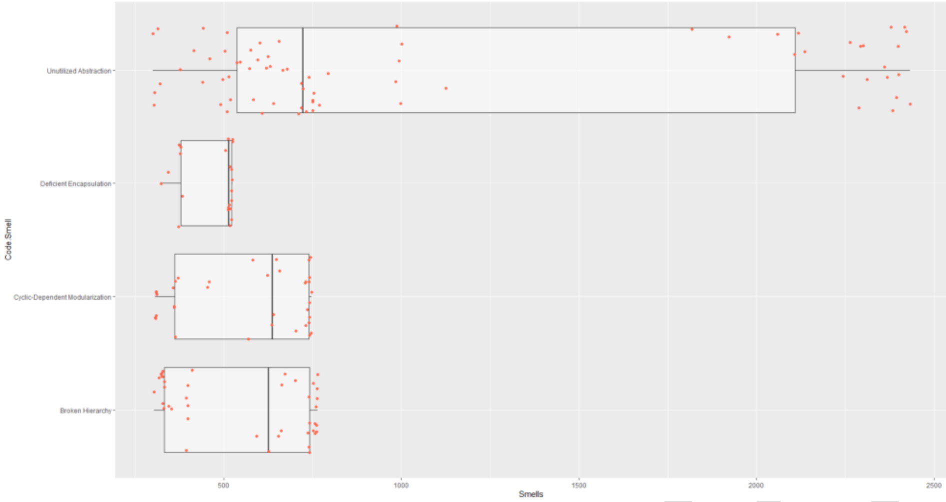

Figure 4.4 combined code smells in monolith codebases from the Qualitas Corpus code smell counts>300

Figure 4.5 combined code smells in monolith codebases from the Qualitas Corpus code smell counts<=300

4.4 Data Cleaning

Data cleaning is taken care of as a part of data collection and curation process. So, no specific data cleaning is required.

The code smell named magic number was removed during analysis around 182160 in number in the monolith codebases, hence removed from the code smells metrics collected to reduce the impact of this code smell.

The code smell named unutilised and unnecessary abstraction is present in significant numbers (136004 Unutilised Abstraction and 9987 Unnecessary Abstraction) in monolith codebases but not removed in the analysis.

4.5 Data Partitioning

The data partitioning is taken care of at the data curation level from the Qualitas Corpus, repositories from GitHub, and use of Designite tools to get code smell in comma separated values (csv) format exploratory data analysis.

4.6 Summary

Excluding magic number that is highest in the monolith codebases and significant in number in microservices codebases, the code smells like the long statement, unutilised abstraction, Complex methods, long parameter list, and broken hierarchy are top code smells identified in the monolith codebases.

	Code.Smell	Smells	archtype
1	Magic Number	182160	0
2	Long Statement	139749	0
3	Unutilized Abstraction	136004	0
4	Complex Method	47296	0
5	Long Parameter List	41432	0
6	Broken Hierarchy	41200	0
7	Cyclic-Dependent Modularization	38274	0
8	Deficient Encapsulation	31199	0
9	Insufficient Modularization	21515	0
10	Complex Conditional	20351	0

Figure 4.6 top code smells in monolith codebase

Whereas, in microservices codebases, unutilised abstraction and long statement code smells are significantly more compared to other ones.

	Code.Smell	Smells	archtype
1	Unutilized Abstraction	416	1
2	Long Statement	385	1
3	Magic Number	226	1
4	Long Parameter List	56	1
5	Broken Hierarchy	39	1
6	Deficient Encapsulation	37	1
7	Long Identifier	33	1
8	Cyclic-Dependent Modularization	31	1
9	Empty catch clause	18	1
10	Unnecessary Abstraction	13	1

Figure 4.7 top code smells in microservices codebases

CHAPTER 5: RESULTS AND EVALUATION

5.1 Introduction

This study compares code smells from monolith codebases with microservices codebases available on the public repositories on GitHub.

5.2 Results

Data class, large class and long method are no more significant code smell found in microservices than monoliths, while unnecessary/unutilised abstraction and long statement continue to remain as significant contributors to code smell in microservices. The magic number code smell remains indifferent in monolith and microservices codebases.

Deficient encapsulation, cyclic-dependent modularisation and complex method and broken hierarchy are significantly less or none in microservices.

Low-Code No-Code platforms are subscription-based with business modelling, business process execution and reports capabilities and encapsulate the complete software development life cycle for citizen developer and focus on solving their fundamental business problem. Furthermore, these platform providers take API based approach, including microservices, to encapsulate and provide AI, machine learning, RPA or Chatbot capabilities to the citizen developer.

These low-code no-code platforms would soon become candidates for different kinds of code smells, which could be a future study.

5.3 Summary

The broader availability, consumption, community contribution, and newer engineering practices followed in the public repositories could be the reason for fewer code smells in microservices or due to the example/sample nature of the codebase available public domain learning. Even newer monolith codebases are less prone to code smells, could be due to awareness or newer engineering practices.

CHAPTER 6: CONCLUSION AND RECOMMENDATIONS

6.1 Introduction

There are several trends in various codebases available that are studied, but the study's conclusion could not be well derived from the patterns observed. This conclusion could be due to the lack of appropriate codebases available in the study's public domain or time constraint of a short study.

6.2 Discussion and Conclusion

The results are mixed from this study, and it is established that microservices codebases are less prone to code smells in public repositories.

At the same time, all cloud providers are encouraging enterprises to move their workloads to the containerised platform on Kubernetes in their own managed cluster. In their chase to move workloads and acquire more significant market share, the monoliths are moved as-is or minimal changes to run on Kubernetes. Kubernetes is one of the popular platforms for hosting microservices workloads. Monolith codebases are tuned to run on these platforms without re-architecture efforts, but technical debt remains running on the microservices centric platform. Though not substantiated by any scientific study, this is a market trend that could be the reasons for increased code smells in monolith codebases running on the containerised platform in the microservices paradigm shift. The public repositories are less prone to code smells, and if more and more enterprise codebases embrace going open source, due to this transparency, these codebases would be less prone to code smells and would increase productivity and save costs in maintaining legacy codebases.

6.3 Contributions

This study attempted to perform a limited study on the codebases available in public repositories for code smells. Performing such studies on private repository would help many organisation identify the extent of technical debt running on their data centres and could be an apparent reason for bottlenecks or low productivity to churn better code to meet their ever-changing business demands.

6.4 Future Work

Database code smells – RDMS and NoSQL are not considered for this study. As a part of the literature review, several papers are available for code smells detection using deep learning techniques. Detecting the code smells is a kind of reactive approach to solving the problem. Most of the static code analysis is post-mortem after the code is already written and prescriptive in nature. There are tools like ReSharper helps in code refactoring, while developer writing code, but these are limited licenses and do not have wider availability. The future study to leverage deep learning techniques and help building tools for broader consumption.

7 REFERENCES

Al-Debagy, O. and Martinek, P., (2018) A Comparative Review of Microservices and Monolithic Architectures. *18th IEEE International Symposium on Computational Intelligence and Informatics, CINTI 2018 - Proceedings*, pp.149–154.

Arcelli Fontana, F., Mäntylä, M. v., Zanoni, M. and Marino, A., (2016) Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, [online] 213, pp.1143–1191. Available at: http://dx.doi.org/10.1007/s10664-015-9378-4.

Brooks, F., (1987) No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 204, pp.10–19.

Chatzigeorgiou, A. and Manakos, A., (2010) Investigating the evolution of bad smells in object-oriented code. *Proceedings - 7th International Conference on the Quality of Information and Communications Technology, QUATIC 2010*, pp.106–115.

Dalla Palma, S., di Nucci, D., Palomba, F. and Tamburri, D.A., (2020) Toward a catalog of software quality metrics for infrastructure code. *Journal of Systems and Software*, [online] 170, p.110726. Available at: https://doi.org/10.1016/j.jss.2020.110726.

Fowler, Martin, J.L., (2014) Microservices Guide. [online] Available at: https://martinfowler.com/microservices/.

Fritzsch, J., Bogner, J., Zimmermann, A. and Wagner, S., (2018) From monolith to microservices: A classification of refactoring approaches. *arXiv*, pp.1–13.

Fu, S. and Shen, B., (2015) Code Bad Smell Detection through Evolutionary Data Mining. In: *International Symposium on Empirical Software Engineering and Measurement*.

Kiyak, E.O., Birant, D. and Birant, K.U., (2019) Comparison of Multi-Label Classification Algorithms for Code Smell Detection. *3rd International Symposium on Multidisciplinary Studies and Innovative Technologies, ISMSIT 2019 - Proceedings*, pp.0–5.

Maldonado, S., Shihab, E. and Tsantalis, N., (2017) to Automatically Detect Self-Admitted Technical Debt. 4311, pp.1044–1062.

Moha, N., Guéhéneuc, Y.G., Duchien, L. and le Meur, A.F., (2010) DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 361, pp.20–36.

di Nucci, D., Palomba, F., Tamburri, D.A., Serebrenik, A. and de Lucia, A., (2018) Detecting code smells using machine learning techniques: Are we there yet? In: *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings*. Institute of Electrical and Electronics Engineers Inc., pp.612–621.

Pahl, C., Brogi, A., Soldani, J. and Jamshidi, P., (2017) Cloud Container Technologies: a State-of-the-Art Review. *IEEE Transactions on Cloud Computing*, 73, pp.677–692.

Palomba, F., Bavota, G., di Penta, M., Oliveto, R., de Lucia, A. and Poshyvanyk, D., (2013) *Detecting bad smells in source code using change history information*. *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*.

Rahman, M.I., Panichella, S. and Taibi, D., (2019) A curated dataset of microservices-based systems. *arXiv*, pp.1–9.

Santos, J.A.M., Rocha-Junior, J.B., Prates, L.C.L., Nascimento, R.S. do, Freitas, M.F. and Mendonça, M.G. de, (2018) A systematic review on the code smell effect. *Journal of Systems and Software*, 144October 2016, pp.450–477.

Schwarz, J., Steffens, A. and Lichter, H., (2018) Code smells in infrastructure as code. *Proceedings - 2018 International Conference on the Quality of Information and Communications Technology, QUATIC 2018*, pp.220–228.

Sharma, T., Fragkoulis, M. and Spinellis, D., (2017) House of Cards: Code Smells in Open-Source C# Repositories. *International Symposium on Empirical Software Engineering and Measurement*, 2017-Novem, pp.424–429.

Singh, S. and Kaur, S., (2018) A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal*, [online] 94, pp.2129–2151. Available at: https://doi.org/10.1016/j.asej.2017.03.002.

Sjoberg, D.I.K., Yamashita, A., Anda, B.C.D., Mockus, A. and Dyba, T., (2013) Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 398, pp.1144–1156.

Taibi, D. and Lenarduzzi, V., (2018) On the Definition of Microservice Bad Smells. *IEEE Software*, 353, pp.56–62.

Taibi, D., Lenarduzzi, V. and Pahl, C., (2020) Microservices Anti-patterns : A Taxonomy. *Book*, pp.111–112.

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H. and Noble, J., (2010) The Qualitas Corpus: A curated collection of Java code for empirical studies. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, pp.336–345.

Tsantalis, N., Ketkar, A. and Dig, D., (2020) RefactoringMiner 2.0. *IEEE Transactions on Software Engineering*, 5589i, pp.1–1.

Vanhanen, J., (2014) Bad Smells in Software – a Taxonomy and an Empirical Study Department of Computer Science and Engineering Software Business and Engineering Institute Mika Mäntylä Bad Smells in Software – a Taxonomy and an Empirical Study Supervisor : November.

Walker, A., Das, D. and Cerny, T., (2020) Automated code-smell detection in microservices through static analysis: A case study. *Applied Sciences (Switzerland)*, 1021, pp.1–20.

Woo, M., (2020) The Rise of No/Low Code Software Development—No Experience Needed? *Engineering*, [online] 69, pp.960–961. Available at: https://doi.org/10.1016/j.eng.2020.07.007 [Accessed 17 Jan. 2021].

Wu, Y., Zhang, Y., Wang, T. and Wang, H., (2020) Characterising the Occurrence of Dockerfile Smells in Open-Source Software: An Empirical Study. *IEEE Access*, 8, pp.34127–34139.

G. Gousios, "The GHTorrent dataset and tool suite," in Proceedings of the 10th Working Conference on Mining Software Repositories, 2013, pp. 233–236