Blekinge Institute of Technology
Doctoral Dissertation Series No. 2025:12
ISSN 1653-2090
ISBN 978-91-7295-512-7

On Quantifying Software Craftsmanship Concepts

Anders Sundelin

DOCTORAL DISSERTATION
for the degree of Doctor of Philosophy at Blekinge Institute of Technology to be publicly defended on November 28, 2025 at 09:00 in J1630

Supervisors
Prof. Javier Gonzalez-Huerta, Blekinge Institute of Technology, Karlskrona, Sweden
Prof. Krzysztof Wnuk, Blekinge Institute of Technology, Karlskrona, Sweden
Prof. Tony Gorschek, Blekinge Institute of Technology, Karlskrona, Sweden

Faculty Opponent
Prof. Miroslaw Staron, Chalmers University of Technology, Gothenburg, Sweden
Grading Committee
Prof. Jesper Andersson, Linnaeus University, Växjö, Sweden,
Prof. Marcela Genero Bocco, Universidad de Castilla-La Mancha, Ciudad Real, Spain
Prof. Dietmar Pfahl, University of Tartu, Tartu, Estonia

Abstract

Background: Books on software craftsmanship typically focus on small teams or individual behavior, and are seldom associated with large, globally distributed organizations that develop and maintain long-lived software systems.

Objective: This thesis aims to quantify the effects of systematically derived aspects of software craftsmanship in industrial settings involving large-scale organizations, with developers spread around the globe.

Method: We employ mixed-methods studies, utilizing both qualitative and quantitative data collection and analysis. A Systematic Literature Review (SLR), together with a longitudinal industrial case study, is used to derive an initial anatomy of software craftsmanship, and we use case studies, experience reports and action research to explore and quantify aspects of this anatomy. We use Bayesian methods to analyze data obtained via archival analysis, as well as Likert-scale data obtained from a survey using the Technology Adoption Model (TAM). Qualitative data has been analyzed using thematic coding, and we use focus groups to validate our conclusions with the studied subjects.

Results: Based on the SLR results and a industrial case study, we derive an anatomy of software craftsmanship, based on four themes, 17 principles and 47 practices. The effects of some practices from this anatomy are then quantified in subsequent articles in the thesis.

Conclusion: Based on literature and case study results, we have found a usable conceptual map of software craftsmanship. However, it remains to be seen how this map will stay relevant, in the face of how cloud migrations and AI-powered Large-Language Model tools will impact future software engineers.

Keywords: Software Craftsmanship, Professionalism, Large-scale software development

Blekinge Institute of Technology
Doctoral Dissertation Series No. 2025:12

On Quantifying Software Craftsmanship Concepts

Anders Sundelin

Doctoral Dissertation in Software Engineering

Department of Software Engineering
Blekinge Institute of Technology
SWEDEN

Copyright pp Anders Sundelin
Paper I © by the authors
Paper II © by the authors
Paper III © by the authors
Paper IV © by Springer Nature. Used by permission.
Paper V © by the authors (Manuscript unpublished)

Blekinge Institute of Technology
Department of Software Engineering

Blekinge Institute of Technology Doctoral Dissertation Series No. 2025:12
ISBN 978-91-7295-512-7
ISSN 1653-2090
urn:nbn:se:bth-urn:nbn:se:bth-28614

Printed in Sweden by Media-Tryck, Lund University, Lund 2025

Media-Tryck is a Nordic Swan Ecolabel
certified provider of printed material.
Read more about our environmental
work at www.mediatryck.lu.se

MADE IN SWEDEN

"You've got to trust the welder."
Paula Wallenburg, Captain (N)
Commanding Officer, 1st Submarine Flotilla,
Swedish Armed Forces

Acknowledgements

While every learning journey is ultimately a solitary path trodden within the mind of the learner, I do wish to express my sincere gratitude to the following fellow travellers, who have enlightened and encouraged my way:

First, my family has been instrumental in supporting me and my journey throughout these years. My deepest thanks to my loving wife, Katti, and to my children for bearing with me when I have been deep in thought, and absent in mind.

Secondly, the staff at the BTH Department of Software Engineering, including my supervisors, Javier, Kris and Tony, have always been supportive, and encouraging of my (sometimes wild) ideas.

Finally, my past and present colleagues at Ericsson AB, who generously allowed me to dedicate portions of my professional time to this endeavour, and who graciously agreed to serve as subjects in my studies.

List of Papers

Paper I

Sundelin, A., Gonzalez-Huerta, J., Wnuk, K., & Gorschek, T. (2021). Towards an Anatomy of Software Craftsmanship. ACM Transactions on Software Engineering and Methodology (TOSEM), 31(1), 1-49. DOI: 10.1145/3468504

Paper II

Sundelin, A., Gonzalez-Huerta, J., & Wnuk, K. (2020, June). The Hidden Cost of Backward Compatibility: When Deprecation Turns Into Technical Debt—An Experience Report. In Proceedings of the 3rd International Conference on Technical Debt (pp. 67-76). DOI: 10.1145/3387906.3388629

Paper III

Sundelin, A., Gonzalez-Huerta, J., Torkar, R., & Wnuk, K. (2025). Governing the Commons: Code Ownership and Code-Clones in Large-scale Software Development. Empirical Software Engineering, 30(2), 43. DOI: 10.1007/s10664-024-10598-7

Paper IV

Sundelin, A. (2025). Learning Observability Tracing Through Experiential Learning. In: Scanniello, G., et al. Product-Focused Software Process Improvement. PROFES 2025. Lecture Notes in Computer Science. Springer, Cham.

Paper V

Sundelin, A., Gonzalez-Huerta, J., & Wnuk, K. (2025). Reducing Friction in Cloud Migration of Services. Submitted to The Journal of Systems and Software (In Practice). First major revision resubmitted 29 July 2025. DOI: 10.48550/arXiv.2503.07169

Author's contribution to the papers

The chapters of this compilation thesis are based on five publications, all of which have had Anders Sundelin as the main author. As main author, he took the main responsibility for conceptualization, methodology, software, formal analysis, visualization, validation and writing of all contributions. He and the co-authors describe their contributions to the chapters in detail, utilizing the contributor role taxonomy *CRediT* [1]:

Paper I

Anders Sundelin: Conceptualization, Methodology, Software, Formal Analysis, Investigation, Data Curation, Writing – Original Draft, Writing – Review & Editing, Visualization, Project Administration.

Javier Gonzalez-Huerta: Conceptualization, Methodology, Investigation, Writing – Review & Editing, Supervision, Funding acquisition

Krzysztof Wnuk: Conceptualization, Methodology, Investigation, Writing – Review & Editing, Supervision

Tony Gorschek: Conceptualization, Writing – Review & Editing, Supervision, Funding acquisition

Paper II

Anders Sundelin: Conceptualization, Methodology, Software, Formal Analysis, Investigation, Data Curation, Writing – Original Draft, Writing – Review & Editing, Visualization, Project Administration.

Javier Gonzalez-Huerta: Conceptualization, Methodology, Investigation, Writing – Review & Editing, Supervision, Funding acquisition

Krzysztof Wnuk: Conceptualization, Methodology, Investigation, Writing – Review & Editing, Supervision

Paper III

Anders Sundelin: Conceptualization, Methodology, Software, Formal Analysis, Investigation, Data Curation, Writing – Original Draft, Writing – Review & Editing, Visualization, Project Administration.

Javier Gonzalez-Huerta: Conceptualization, Methodology, Investigation, Writing – Review & Editing, Supervision, Funding acquisition

Richard Torkar: Methodology, Writing – Review & Editing

Krzysztof Wnuk: Conceptualization, Writing – Review & Editing, Supervision

Paper IV

Anders Sundelin: Conceptualization, Methodology, Software, Formal Analysis, Investigation, Data Curation, Writing – Original Draft, Writing – Review & Editing, Visualization, Project Administration.

Paper V

Anders Sundelin: Conceptualization, Methodology, Software, Formal Analysis, Investigation, Data Curation, Writing – Original Draft, Writing – Review & Editing, Visualization, Project Administration.
Javier Gonzalez-Huerta: Conceptualization, Methodology, Investigation, Writing – Review & Editing, Supervision, Funding acquisition
Krzysztof Wnuk: Writing – Review & Editing, Supervision

Abbreviations

DAG	Directed Acyclic Graph.
DRM	Design Research Methodology.
ESE	Empirical Software Engineering.
GQM	Goal Question Metric.
LLMs	Large Language Models.
RCT	Randomized Controlled Trial.
SLR	Systematic Literature Review.
SWEBOK	Software Engineering Body of Knowledge.
TAM	Technology Adoption Model.

Table of Contents

1 Introduction

1.1 Overview

Software Engineering is, relative to other social sciences, a relatively young discipline, originating in the 1960s. The conventional starting point is often considered a 1968 NATO-sponsored conference in Garmisch, although the term had been coined a few years before [2, 3]. Since its founding, several trends have dominated the contemporary view of the field, from the Waterfall processes of the 1970s to the Agile and Lean focus on team-based collaboration of the early 2000s.

In parallel, practitioners and authors have argued that software development is more closely related to arts and crafts than science. When discussing commercial application development, McBreen [4] chose the term Software Craftsmanship over Software Engineering to shift the focus from manufacturing to "the metaphor of skilled practitioners intent on mastering their craft, of pride in and responsibility for, the fruits of their labor." Other authors, such as Martin [5–7] and Mancuso [8] have continued using this terminology. In his latest book, Robert Martin [9] traces a lineage of innovators in software development and engineering, from the unrealized mechanical dreams of Charles Babbage to the arrival of modern AI-assisted coding tools.

Today, interest in programmers and in programming has also become mainstream in popular culture. Books such as *Coders* by Clive Thompson [10] portray software developers as a "tribe destined to rule the world." At the same time, the rapid rise of Generative AI and Large Language Models (LLMs) is transforming the industry, with many business leaders expecting that the routine, detail-oriented tasks traditionally performed by skilled and experienced programmers and testers will increasingly be delegated to these AI systems[1]. The very terms "code" and "coding" still carry an aura of mystery—evoking the idea of secret languages known only to an elite few. This mystique certainly applied in the early days of computing, when pioneers such as Grace Hopper developed the first working compilers, long before the advent of the first high-level languages that today's students might still regard as surprisingly low-level.

Empirical Software Engineering (ESE) research has so far paid minimal attention to the Software Craftsmanship movement, especially when contrasted with the

[1] Example tools includes, but is not limited to, Amazon Q™ and GitHub Copilot™

substantial academic and industrial focus on Agile and Lean principles [11–13]. In this thesis, the term empirical is used in its scientific sense—referring to knowledge, data, and conclusions derived from systematic observation, hands-on experience, or structured experimentation, rather than mere theoretical reasoning. Despite the practical significance of Software Craftsmanship, in Paper I, our literature review confirms a notable gap: no prior ESE studies have rigorously examined what principles or practices it comprises. This gap is particularly striking given that the primary goal of ESE is to observe, measure, and understand real-world software development phenomena and their impact on development processes or product quality across both academic and industrial contexts. By investigating Software Craftsmanship principles and practices in large-scale industrial settings, our work addresses this gap, providing empirical evidence to inform both practitioners and researchers, and contributing a new dimension to the broader discourse on effective software development methodologies.

This first chapter of the thesis introduces the reader to the overall research area, explains the structure of the thesis, and how it relates to the research goal of quantifying the impact of Software Craftsmanship practices. Section 1.2 introduces the background of Software Craftsmanship, Bayesian inference, and quantitative metrics in Empirical Software Engineering. Section 1.3 describes our goals and overall Research Questions, and Section 1.4 discusses our methods and the conceptual and philosophical frameworks we used to conduct our mixed-methods studies. Section 1.5 provides our overall contributions, and in Section 1.6 we discuss these. Finally, we conclude this chapter in Section 1.7, before the rest of the papers.

1.2 Background

As described in Paper I, the concepts of Software Craftsmanship have early roots in computing, going back to Brooks [14]. In later years, book authors such as Martin [5–7] and Mancuso [8] have popularized the concept.

Empirical Software Engineering, meanwhile, has struggled to formulate laws and core concepts. The authoritative Software Engineering Body of Knowledge (SWEBOK) [15] has recently been updated, and is now in its fourth revision, but it is mainly descriptive, not prescriptive, like the Software Craftsmanship books. Earlier, books by Endres et al. [16] and Glass [17] have tried to popularize facts and laws about Software Engineering in terms suitable for practitioners. According to Winters et al. [18], engineers at Google refer to Software Engineering as "programming integrated over time," and describe three fundamental principles that software development organizations must adhere to: (i) *Time and Change*, considering the usable lifetime of the program at hand, (ii) *Scale and Growth*, considering how the organization needs to change over time, and (iii) *Trade-Offs and Costs*, relating to the decision-making process, considering the first two principles.

1.2.1 Studied Context

Our studies focus on software that addresses the needs of specific *stakeholders*, who are typically different from the developers. Thus, a gap exists between the program creators and the program users. Furthermore, we study software of the E-program type. In a classic article, Lehman distinguished between three types of programs [19]:

(I) An S-program is written according to an exact, immutable, *specification* of what tasks the program should perform. Programs that implement specific algorithms, or solve "closed problems," such as board games (e.g., AlphaGo for Go, DeepBlue for chess), are instances of S-programs.

(II) A P-program is written to model a *real-world problem*, such as weather prediction, or route calculation, given a city map and congestion information.

(III) An E-program automates a human or societal activity, like Uber's route planner, which allocates drivers based on current and predicted demand to maximize profit, expedite ride-hailing, and ensure customer satisfaction. These programs become *embedded* in society, which itself evolves in response to the program output. Due to this inherent feedback loop, E-programs are the most susceptible to changes driven by evolving requirements.

Many books on Software Craftsmanship focus on shaping the professional behavior of individuals and small teams—explaining what constitutes clean code, why testing and test-driven development are important, how to say *No* in a professional manner, and how to plan future career paths [6–8]. In contrast, our studies examine larger-scale software-intensive projects, in which multiple teams of varying sizes collaborate to build software products or services. Consequently, the most granular unit of analysis in our research is the *team*, typically comprising developers, testers, and architects with diverse expertise and experience levels.

The act of software creation is perhaps unique in that a single, exceptional author can have a huge impact and create tools of great value and complexity. Examples of this are Linus Torvalds creating the first version of the Git version control system[2] during four intense days in 2003, or Donald Knuth creating the first working version of the TEX typesetting system[3] at the dawn of the personal computer age. While impressive, these feats fall outside the scope and context of our studies.

However, we would consider the current state of the Open-Source Git tool ecosystem, where multiple authors, reviewers, and testers collaborate, valid for our research. Similarly, the TEX and LATEX software ecosystems have grown considerably in both capabilities and contributors, and would also qualify as valid subjects for study.

Thus, even though individual "master craftsmen (and –women)" can significantly influence software tools and products, successful software will, over time,

[2] https://git-scm.com/

[3] https://tug.org/whatis.html

lead to the growth of the organization surrounding it, making that organization itself a worthy subject of study. Therefore, we consider our research relevant to both individual practitioners and software development organizations.

1.2.2 Bayesian Data Analysis

Paper III and Paper IV adopt Bayesian Data Analysis in place of the frequentist statistics commonly used in ESE studies. This reflects a broader shift in the applied sciences toward moving away from *p*-values and the implicit assumptions of methods such as ANOVA and Student's *t*-test, and toward making causal and distributional assumptions explicit through Bayesian statistical modeling [20–22]. By specifying causal structures explicitly (for example, with a Directed Acyclic Graph (DAG)), and applying the causal framework developed by Pearl [23], researchers can interpret results more robustly and reduce bias arising from inappropriate control variables [24].

We acknowledge that Bayesian Data Analysis is not yet widespread in ESE research, but we expect its use to grow as tooling improves and visibility increases among researchers [20, 25, 26]. There are many textbooks that cover the theory and practice of Bayesian analysis and causal modeling, and readers are encouraged to consult them for foundational guidance [23, 27].

1.2.3 Quantitative Measures in Software Engineering

Software Engineering has a long tradition of quantitative measurements, dating back to its founding as a way to control what was perceived as "the software crisis" (that is, the exploding cost of software, relative to the, for the time, expensive hardware) [2]. Books such as "Software Metrics", by Fenton et al. [28] have been published and updated multiple times over the years.

Typically, a *measurement* is a mapping from the empirical world we live and work in, to the formal, relational world, where logic and mathematics rule the day. A *measure* is the number or symbol assigned to an entity in order to characterize an attribute. Measurements come in different scales, such as:

Nominal scale, where entities are classified into different classes, but no other relation exists among the classes.

Ordinal scale, where the classes have an inherent order (e.g., S, M, L, XL), with respect to the attribute being measured.

Interval scale, where differences between the ordered classes are preserved, but ratios do not make sense.

Ratio scale, where both differences and ratios between classes are supported, and there is a zero element, meaning the absence of the entity being measured.

The ratio scale always starts at zero and progresses in equal intervals, known as units.

Absolute scale, which is equal to the absolute count of the entity (e.g., the number of faults in a component, or people in a project) being measured.

Different analytical methods impose different assumptions about the analyzed data and its underlying distribution. For example, a researcher using parametric frequentist techniques (e.g., *t*-tests or ANOVA) on Likert scale data assumes that the responses follow an interval scale (i.e., that respondents perceive equally spaced levels) [29]. Therefore, in Paper IV, we instead analyze the responses of the survey questions using Bayesian Data Analysis and a cumulative probit function, which only assumes that the responses are coded using an ordinal scale [30].

1.3 Goals and Research Questions

In light of the research gap identified in Section 1.1, we adopt the Goal Question Metric (GQM) framework pioneered by Basili et al. [31] to formulate a single, overarching goal: **To explore and quantify the impacts of Software Craftsmanship principles and practices in large-scale software development.**

Based on this goal, we formulated the following research questions:

RQ1: Which principles and practices of Software Craftsmanship have been described in prior literature, and which can be identified in a large-scale industrial project?

The literature component of this research question is addressed by the systematic review in Paper I, while the empirical component is informed by the case study findings in the same paper. Together, these results form the complete concept map presented on page 38.

RQ2: Can we quantify the impacts of these principles and practices, using empirical tools from modern Software Engineering?

In Paper II, we investigate and discuss the spread of Technical Debt (TD), and the need to keep code (including integration test code) clean and up-to-date. In Paper III, we discuss and model how teams, working in an industrial setting with weak ownership rules, exhibit different behavior related to how and when they introduce code clones. In Paper IV and in Appendix A, we discuss how an organization used mentoring and learning events to spread the knowledge about a newly introduced distributed tracing tool. In Paper V, we show how the lack of end-to-end tests and the lack of feedback (end-to-end user feedback) contribute to inefficiencies that might be benign when

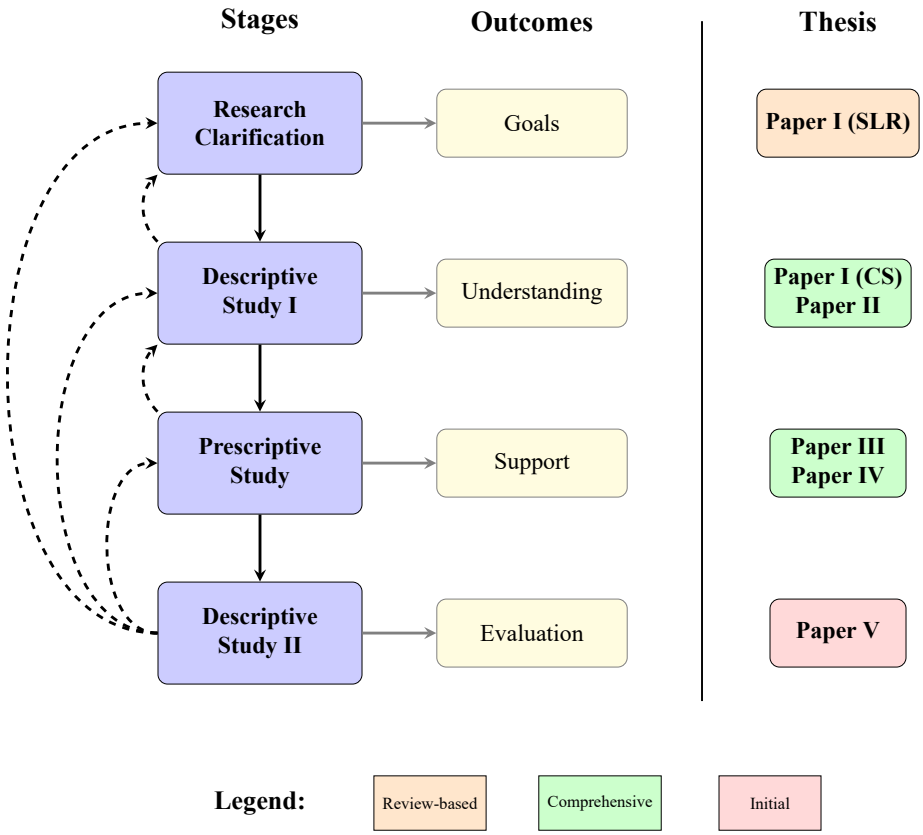


Figure 1.1: The four interlinked stages of Design Research Methodology, their main outcomes, and how the papers in this thesis relate to them.

deploying on fixed and purchased hardware, but become pain points when deploying to a cloud provider.

These research questions are associated with metrics as explained in detail in each paper, and in general terms in Section 1.5.3.

1.4 Methods

In this thesis, we follow *Design Research Methodology (DRM)* [32], as the overarching research methodology. As illustrated in Figure 1.1, DRM projects are divided into four stages, each having different main outcomes. However, DRM projects are not required to contain all stages, and they do not need to undertake each stage in equal depth. The arrows in the figure indicate that many DRM projects iterate between the different stages in a project.

Blessing et al. [32] defines seven different types of DRM studies, of various

Table 1.1: Summary of research methods, data collection, and analysis methods per paper.

Research Method	I	II	III	IV	V
Literature Study (SLR)	•				
Exploratory Case Study	•	•	•		•
Action Research				•	
Data Collection Method	I	II	III	IV	V
Archival Analysis	•	•	•		•
Survey				•	
Semi-structured Interview	•				•
Focus Group (validation)			•		•
Data Analysis Method	I	II	III	IV	V
Frequentist Inference	•	•			
Bayesian Data Analysis			•	•	
Thematic Analysis	•		•		•

complexity and length. Most studies start in the *Research Clarification* stage, which typically is *Review-based*. In this thesis, this corresponds to the Systematic Literature Review (SLR) part of Paper I. The case study part of Paper I, together with the descriptive experience report in Paper II form *Descriptive Study I*. In Paper III and Paper IV, we conduct *Prescriptive studies*, focusing on visualizing team behavior related to code clone introduction and system architecture using distributed tracing tools. All of these are to be considered *Comprehensive*, as they contain both reviews of relevant literature and results produced by the researcher. Finally, Paper V closes the project, as *Descriptive Study II* of the *Initial* type. In this paper, we use parts of the anatomy produced by the prior studies, and find missing parts that might be candidates for inclusion in future versions of the anatomy.

Table 1.1 summarizes the research methods, data collection, and analysis used for all contributed papers. Each method is discussed and described in the respective chapter, corresponding to the study in which it was applied.

1.4.1 Overview of studies

Figure 1.2 illustrates how our studies are framed using the ABC framework [33]. The majority of studies are *Field Studies*, which prioritize realism of Context over measurability of Behaviors and generalizability over Actors. Paper III also contains a *Computer Simulation* part (as it use Hamiltonian Monte Carlo to build a Bayesian model of team tendency of introducing code clones), and Paper IV has traces of a *Field Experiment*, as it measures the effects of structured mentoring a group of professional developers, using experiential learning. The Systematic Literature Review part of Paper I, which uses both gray literature (books) and peer-reviewed papers as data sources, could be seen as a *Sample Study*, bordering on *Judgement Studies* (as the books purport to be written by experts in the field).

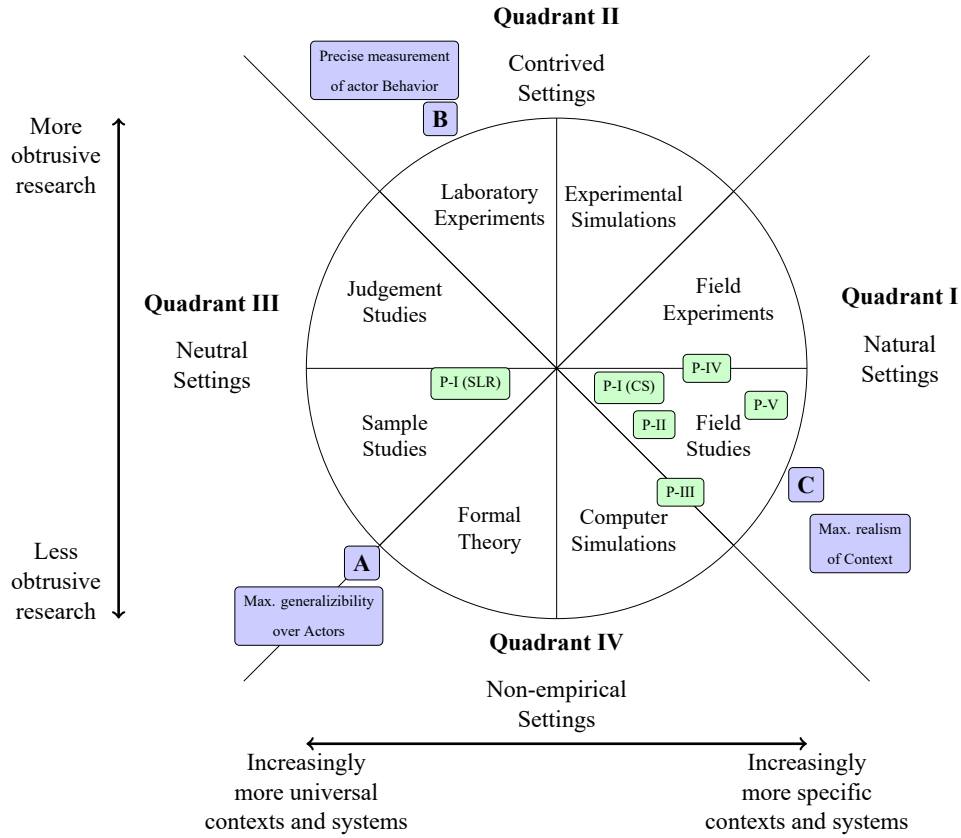


Figure 1.2: The placement of the five studies in the ABC framework [33]

All studies have been conducted in similar—but not identical—contexts, which are illustrated in Figure 1.3, together with key actors. A *Product Development Organization* (consisting of *Architects*, *Developers*, *Testers*, and *Product Owners*, in various *Team* constellations, plus associated *Manager* roles) develops a software *Product*, which is sold to *Customers*. However, before the customer can use the product, *Sales Architects* specify use cases and *Solution Integrators* customize the product, which is realized as an installed *Deployment*, exposing various *Services* that are being used by *End Users*. The installed deployment is supervised by *Operation Engineers*, who may provide feedback to the Product Development Organization (e.g., when a fault or issue is found). Papers I–IV study various Product Development Organizations, whereas Paper V also studies customized deployments, and the feedback that they provide. All the studied Product Development Organizations are categorized as either *large-scale*, or *very large-scale* according to the taxonomy developed by Dingsøyr et al. [34].

All the organizations in our studies develop various business-critical, highly available, and customizable software systems, but although there are similarities (e.g.,

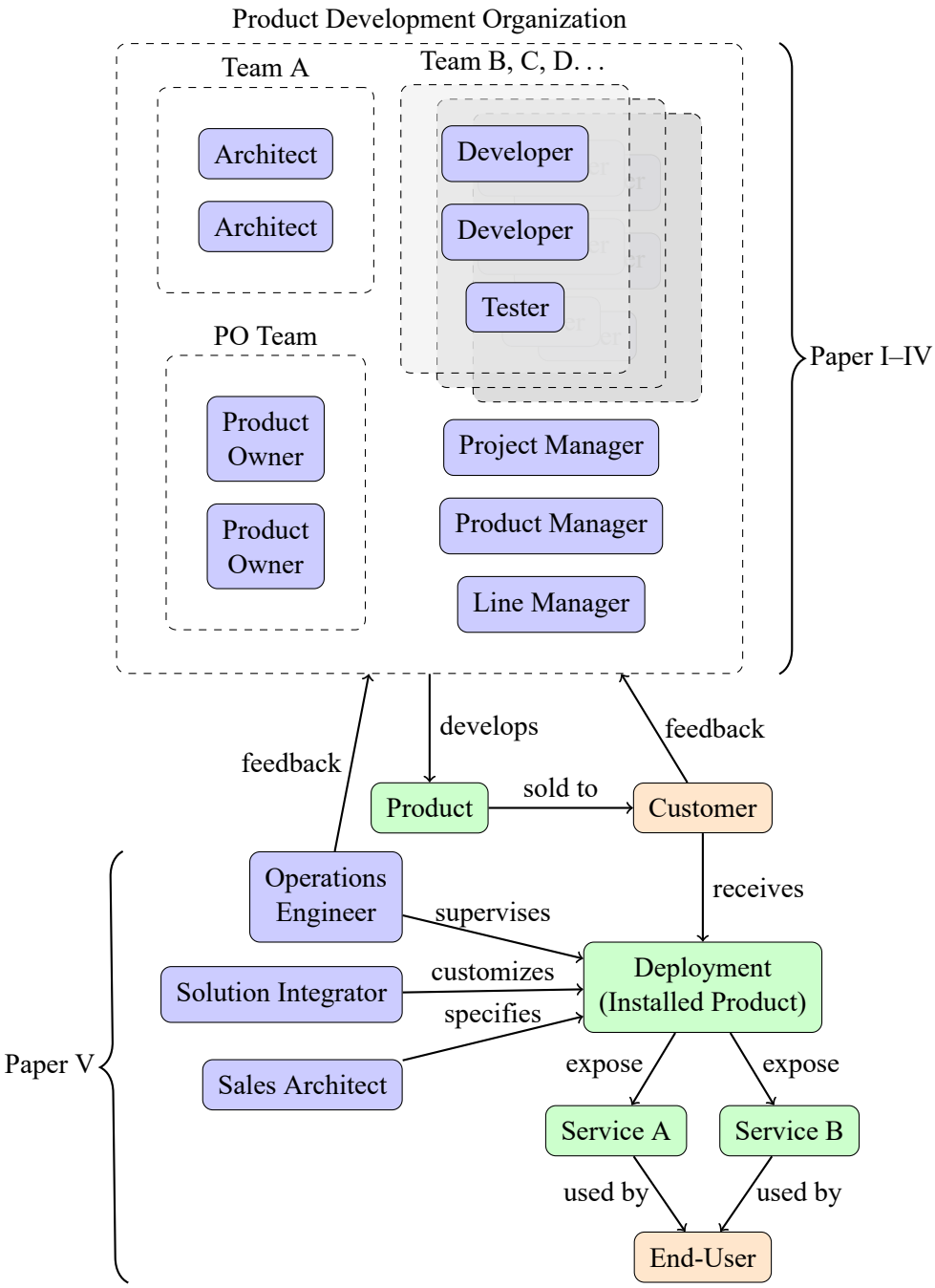


Figure 1.3: The conceptual actors and context of the various studies in the thesis. Paper I--IV focus on the product development organization, whereas Paper V mainly focuses on feedback from deployed systems, including how the installed product has been customized and is being used.

the same parent company), the studied context also exhibits differences. In Paper I and Paper II, quantitative data is collected from the start of product development, and consequences of early decisions are followed for several years, via archival analysis. In Paper III, the studied product, whose development had been put on hold a few years before, was restarted, with partly new development teams (as seen in the organizational chart in the paper). We utilized three years of quantitative data to model the tendency of the teams to introduce code clones in various components. Paper IV, on the other hand, studied an organization where all but a few individuals had little prior knowledge of the structure of an existing large-scale software product. In Paper V, we instead focus on how the deployment cost would change when migrating (in a "lift–and–shift" manner) a large legacy product to a particular cloud provider, and how usage data from the deployed system could be used to inform about inefficiencies.

Figure 1.4 shows the process of constructing this thesis, including the conceptual framework. Based on *Prior Experience*, *Books*, peer-reviewed *Literature*, and the *Context*, we derived the *Anatomy*, as explained in Paper I. The impact of a subset of the principles and practices from the anatomy is then measured and quantified, both in Paper I, and in the following papers II-V.

1.4.2 Philosophic stance

Social science—that is, science involving human behavior—can broadly be said to encompass two traditions, each corresponding to a particular philosophical stance. The quantitative paradigm (involving numerical values, measurements, and mathematical treatment of data) has broadly been linked to *positivism*, which states that objective knowledge (the facts of the world) can be gained from direct experience or observations. According to positivists, objective facts are the only knowledge available to science. Theoretical entities are rejected, and the purpose of science is to develop universal *causal laws* [35]. In contrast, the qualitative paradigm (such as discourse analysis, grounded theory, and other analyses of text and non-numeric data) has wider philosophical underpinnings, but is often linked to *constructivism*. Constructivists emphasize the world as it is lived and felt by people, as they act in social situations, and consider that the task of the researcher is to understand the multiple *social constructions* of meaning and knowledge, rejecting the positivist's claim that there are any causal laws and objective truths, devoid of social meaning.

In our studies, we have instead adopted a philosophical stance called *pragmatism*. As pragmatists, we seek a middle ground between positivists and constructivists, recognizing the existence and importance of both the physical world and the inner, psychological world of humans, which might influence behaviors. Following the pragmatist approach, all of our studies have employed mixed methods (quantitative as well as qualitative).

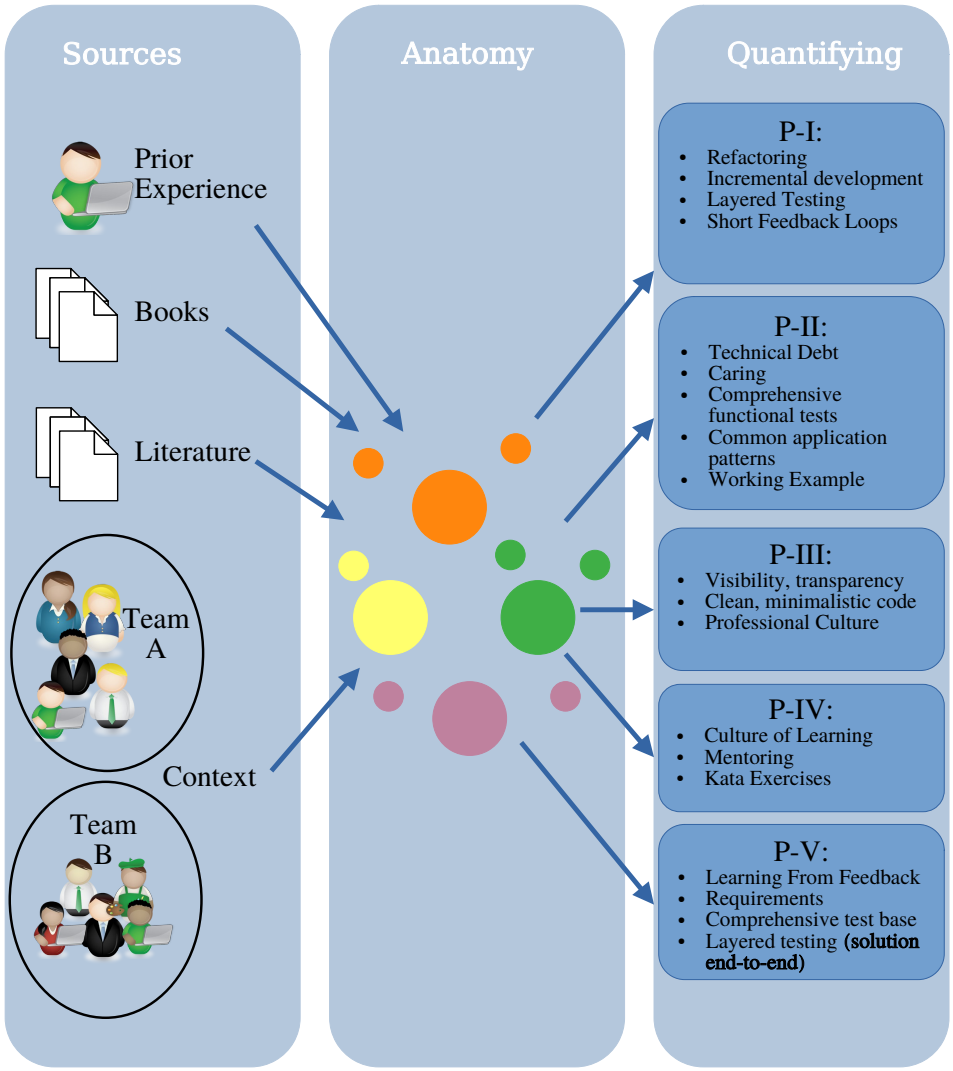


Figure 1.4: The process used to construct the conceptual framework guiding the thesis.

Pragmatism and mixed-methods social studies have been criticized for lacking rigor, having an "anything-goes" attitude toward scientific methods. Therefore, researchers such as Lipscomb [36] and McEvoy et al. [37] have argued for adopting the *critical realist theory* of Bhaskar [38], and employing methods such as triangulation, which involves observing the same event from different perspectives (methodological or observational).

1.4.2.1 An example of critical realism usage

We have strived to adopt critical realism in all our studies, and exemplify our process by explaining in detail the development of the study underlying Paper III.

Paper III focuses on the detection of code clones and maps the introduction of these onto authoring teams. This data is then used to develop (via Bayesian inference) a model that portends to predict the probability of a given team introducing code clones in future changes (given particular data about the change; full details in the paper). Having developed the model based on historical data for several components, we then presented the findings to four different teams in five focus groups to elicit their reactions and identify possible explanations for our findings.

The ontological assumptions underlying this study are:

(i) There exists entities called "source code," which are associated with "components."

(ii) Sometimes, identical chunks of source code appear in the same component.

(iii) There exist tools, such as SonarQube, that can find duplicated chunks.

(iv) Duplicated source code always arises due to human intervention by programmers or architects.

(v) Programmers and architects have different experience levels, and care to different degrees about the source code they work with.

The epistemological assumptions are:

(i) The tools and analytical methods we use act "objectively," without introducing systematic bias that would render the result invalid. Note that this does not imply that the tool or analysis method is fault-free, only that it produces results that are *applicable* in the studied situation (e.g., the tool must not exclude certain authors, nationalities, gender, or other subgroups, and analysis methods must follow validated procedures).

(ii) In the systems we study, duplicated code chunks are, in general, seen as "bad practice," because of ripple effects when modifying code. This implies that responsible programmers should be mindful of the number (or likelihood) of clones being introduced.

(iii) However, in some cases, clones are expected, even required. Syntactic clones that represent different entities (e.g., input data and output data) might be identical from a syntactic perspective, but from a semantic perspective, if the input and output interfaces have different lifecycles (e.g., different clients), clones are not "bad practice," but instead the result of responsible programmers [39]. This implies that it is unrealistic to enforce zero clones as a target result for the clone detection tool. Another way to express this is that the clone detection tool operates on the *structural* domain, whereas a proper (or perfect) tool would instead work on the *semantic* domain.

When developing the study, we first formulated our assumptions as a causal Directed Acyclic Graph (DAG), then used archival data (source code and organizational charts) and tool support to gather and analyze quantitative data. Eventually, we arrived at a model that we considered interesting—for us, as researchers. To judge whether the architects and developers of the system under study agreed with our conclusions, and to get their explanations for what we found, we used five focus groups (two for the Architect team, and one each for the three main teams in the organization), and used Open Coding to summarize their opinions, and to guide us into further studies.

Thus, we wanted to judge the usefulness of the tool from the *perspective* of the architects and developers who were responsible for the code clones. Due to organizational changes, we were unfortunately unable to judge tool effects on team behavior via a longer case study, but we made sure to publish the data and analysis as a complete replication package, so other researchers (including ourselves) can replicate the study in other contexts.

1.4.3 Validity

In this section, we discuss the validity of the conclusions in the overall thesis according to the criteria provided by Runeson et al. [40]. Each individual study also include a section on validity threats, for Papers I, II, III and V as a section in the paper, and for Paper IV these are described in Appendix A.

Construct validity refers to the extent to which the studied measures represent what the researcher has in mind, according to the research questions. For example, in an interview or survey situation, are the questions clear and interpreted by the respondents the way the researcher intended?

To address construct validity, we have sought feedback from both co-authors (e.g., regarding survey and interview questionnaires) and the studied organizations (e.g., discussing the initial draft of the anatomy with the studied organization in Paper II). When analyzing source code data from Git repositories, we have considered (and checked with representative developers and architects)

the Git practices related to branching, merging, and squashing commits within the studied organization.

Thus, to improve the construct validity of our studies, we have used *incremental feedback* to validate that our conclusions are plausible and relevant for the studied organization.

Internal validity refers to whether there are other, not studied factors that could explain apparent causal relationships. Cook et al. [41] define thirteen specific threats to internal validity (*History*, *Maturation*, *Testing*, *Instrumentation*, *Regression*, *Selection*, *Mortality*, *Selection interactions*, *Ambiguity about causal direction*, *Diffusion*, *Compensatory equalization of treatments*, *Compensatory rivalry*, *Resentful demoralization*). Typical strategies to improve internal validity include conducting an Randomized Controlled Trial (RCT), using blinding, and designing control groups carefully, which can be challenging to achieve in industrial contexts.

None of the papers in this thesis has used randomization or controlled experimental designs, which is a weakness and a threat to internal validity. In Paper III, we instead report our causal assumptions in the form of a DAG [23]. This makes our causal assumptions explicit and allows for testing the strength of the relations among variables in replication studies.

To improve internal validity, we employed observer triangulation during data collection and analysis, as well as methodological triangulation (e.g., by using both quantitative and qualitative data sources). Still, we cannot rule out that other factors could have influenced our findings, and we cannot completely rule out all threats to internal validity.

External validity is also known as **Generalizability**, and refers to whether the findings are possible to generalize to people or situations outside the studied case.

Case studies have been criticized for a lack of generalizability, with researchers claiming that it is impossible to generalize from a single case. However, as Flyvbjerg [42] notes, cases are essential to human learning, and it is indeed possible to generalize, in social as well as natural sciences, from a single case, depending on how the case is chosen. Flyvbjerg gives Galileo's rejection of Aristotle's theory of gravity as an example of a "black swan" case study (conducted first in theory, and later validated with an experiment). Thus, depending on how the studied case is chosen, analytical generalizability can be achieved even from a single case.

One limitation to external validity is that all studies have been conducted in an industrial setting, specifically within a single company operating in a business-to-business market. To aid readers in assessing the generalizability of our findings, we have described the particular context and characteristics of the studied

organization in every paper in the thesis. Where possible, we have provided fully anonymized replication packages, which should aid future researchers in replicating our studies in other contexts.

Reliability refers to whether the findings are dependent on the specific researchers.

To increase reliability, wherever possible, we have used observer triangulation by having co-authors independently assess, code, and review the data, analysis, and results. In most papers, we have also used methodological triangulation, using both qualitative and quantitative data and analysis methods. Still, we cannot completely rule out researcher bias, given that the author of this thesis has a long professional career as a software developer and architect.

Quantitative metrics were collected using industry-standard tools such as Git and SonarQube, with replication packages provided for numerical analyses where feasible. Qualitative data, such as anonymized transcripts and coding schemes, are also available on request, when possible.

1.5 Contributions

With this thesis, we aim to provide three kinds of contributions: (i) *theoretical* contributions, which add the concept of Software Craftsmanship to the existing Software Engineering principles, (ii) *methodological* contributions, by using Bayesian inference to assess the behavior of developer teams, and (iii) *scientific*, by quantifying the impacts of some principles and practices from Software Craftsmanship.

1.5.1 Theoretical Contributions

In Paper I, we systematically derive an anatomy of Software Craftsmanship, containing 17 principles and 47 hierarchical practices, linked to four core themes (*Feedback*, *Iterative Design*, *Shared Professional Culture*, and *Value-added Architecture*). We base this anatomy on research papers, books, and experience from an industrial case study spanning several years, which, on average, involved 48 developers across multiple teams.

Using this anatomy, it should be possible to analyze future organizations through a Software Craftsmanship lens.

In Paper V, we note that the lack of *end-to-end usage data* and *solution feedback* hides inefficiencies that become pain points when migrating to modern cloud environments. Thus, we believe that the anatomy should include more *End-to-End Feedback*, and that the comprehensive test suite should be extended to cover the full solution, if elastic deployment models (e.g., modern cloud solutions) are being used.

1.5.2 Methodological Contributions

In Paper III, we illustrate how to use Bayesian inference to assess the likelihood of teams introducing code clones, based on historical behavior. As we include a fully reproducible replication package and verify the tools' utility with the studied organization, we hope that this contribution sparks an interest from contemporary Software Engineers to utilize statistical tools beyond classical frequentist analysis, such as Bayesian Multi-Level Generalized Linear Models.

In Paper IV (with detailed methodology discussed in Appendix A), we use Bayesian methods to summarize Likert-scale data collected as part of an action-research study. This paper also contributes a reproducible replication package, including both collected data, Bayesian models, and analysis of the model output. We hope that this will aid other researchers analyzing this type of data in future studies.

1.5.3 Scientific Contributions

We collect quantitative data on the following practices:

Paper I: We report summary statistics on the number of **Refactoring** commits taking place during the study period, finding that developers performed $\approx 17\%$ refactoring commits, notably more than the $\approx 13\%$ fault corrections.

We find that the studied system grew linearly, using **Incremental Development** during the entire studied period, and that **Layered Testing** was used, with the test code base (both unit and integration tests) growing faster than production code.

We found that the organization prioritized defining smaller features, which enabled the use of **Short Feedback Loops**, where the typical small feature was developed in about one sprint (three weeks).

Paper II: The paper studied **Technical Debt** in detail, particularly debt occurring in regression tests due to deprecated services. We find that the spread of deprecation debt was highly uneven among the studied files, and that highly used services would have benefited from more **Caring**.

We also find **Comprehensive Functional Tests**, which would have benefited from **Common Application Patterns**, and having updated **Working Examples**.

Paper III: The paper focuses on quantifying one aspect—introduction of code clones—of the **Professional Culture** of development teams in an organization employing collective code ownership. Reducing duplications is a core tenet of **Clean, Minimalistic Code**, and the paper highlights the benefits of having high **Visibility and Transparency** about the clone introduction rate. A key finding of

the paper is that teams are expected to behave differently in different components, and that a team that has taken on "self-selected ownership" of a component is expected to introduce fewer code duplicates in that component. Another interesting finding is that, at least for some teams and repositories, existing complexity in the changed file does not seem to affect the code clone introduction rate. However, we found that all teams were affected by the number of existing clones in a file, further strengthening the evidence of the "Broken Windows Theory" [43].

Paper IV: This action research paper focuses on fostering and nurturing a **Culture of Learning**, via spreading knowledge about an existing dynamic tracing feature in the studied system. The training was in action, similar to **Kata Exercises**, and several developers took part in the **Mentoring** of fellow developers.

Paper V: The paper highlights the importance of **Learning from Feedback**, particularly real-life customer feedback, which strengthens the **Requirements** gathering. While our original anatomy in Paper I included **On-site customer (proxies)**, it did not consider the fact that objective evidence (e.g., logs, traces, events) these days can be gathered automatically and directly from deployed systems, and analyzed using large-scale cloud infrastructure to gain insights that would help both product development and operations.

One cause for the missing feedback is the lack of a **Comprehensive Test Base**, particularly in the acceptance **Test Layer** (focusing on end-to-end solution verification). As shown in the paper, in modern cloud deployments, it is crucial to consider and address inefficiencies resulting from poor service integrations in end-to-end solutions.

1.6 Discussion

In the following sections, we discuss the contributions from both scientific and practical perspectives. Section 1.6.1 discusses the anticipated implications for research and Section 1.6.2 the implications for practice. In Section 1.6.3 we discuss limitations to our thesis, and in Section 1.6.4 we postulate about future work.

1.6.1 Implications for Research

We find that the anatomy of software craftsmanship presented in Paper I partially overlaps with the competence areas in SWEBOK, which in its fourth revision encompass nine areas related to software development (*Requirements*, *Architecture*, *Design*, *Construction*, *Testing*, *Maintenance*, *Configuration Management*, *Quality*, and *Security*), six areas related to Software Engineering (*Operations*, *Management*, *Process*,

Models and Methods, *Professional Practice*, and *Economics*), and three foundational areas (*Computing*, *Mathematical*, and *Engineering*) [15]. However, while SWEBOK mentions Process and Professional Practices, the anatomy places more emphasis on different aspects of a *Shared Professional Culture*, including a *Culture of Learning*, which reflects the ever-evolving practices and methods in the Software Engineering profession. We also find that SWEBOK lacks an explicit *Feedback* area, which is an important theme in the anatomy of software craftsmanship.

As the principles and practices are interlinked, we aim to stimulate further research into how organizations can leverage principles and practices from Software Craftsmanship to develop and maintain software in a clear, repeatable, and predictable way. However, we do acknowledge that our anatomy is derived from a particular context, and that other contexts might require additional areas from SWEBOK or other bodies of knowledge.

Furthermore, this thesis illustrates how Bayesian models and causal inference can be applied to infer team behaviors [23]. Using this approach, organizations can objectively and reproducibly assess the effects of professional culture and systematically model behaviors and effects across their organizational structures.

1.6.2 Implications for Practice

We structure the implications for practitioners according to the themes in the anatomy.

From an **architectural** standpoint, we note that *clean and minimalistic code*, with *common application patterns*, remains the ideal. As illustrated in Paper II, deprecation due to the desire to keep *backwards compatibility* will sometimes interfere with the ideal. Hence, the decision to keep backwards compatible APIs needs to be *balanced* towards the desire to keep the code base clean.

The principle of **iterative design, development and verification**, with particular focus on *layered, well-structured, automated regression testing*, is an enabler for keeping the code base clean via *refactorings* or re-architecting old solutions to fit new environments or constraints. However, regression tests require hardware (owned or rented) and take time to execute, which also interacts with the need to consider the value and life-cycle of the features exposed by the product. In Paper V, we find hidden inefficiencies caused by the lack of a comprehensive suite of end-to-end test cases in a solution that was intended to move to cloud deployment, while keeping the legacy architecture intact.

Having a **shared professional culture** might seem obvious, but there are differences that are hard to quantify. Some of the practices in our anatomy are relatively simple, binary verdicts, such as: *standard development environment*, *visible backlog*, *kata exercises*, and *cross-team forums*. However, others are harder to quantify, such as *caring*, *pride*, *accountability*, and *mentoring*. In Paper IV, we used the Technology Adoption Model (TAM) to assess how likely participants in dual training events were

to utilize a new tool, with skilled team members serving as mentors for the others. We complemented the TAM questions with qualitative questions and summarized the findings using Bayesian inference with a cumulative probit function. While there are certainly other methods to assess cultural fit, we hope that this method can be used in other professional contexts as well.

As mentioned above, much of the **feedback** principles tie into existing principles, such as *learning* and *mentoring*. *Reviews* are one form of feedback, and *static review tools* are one form of automated feedback that teams might use. In Paper III we presented a novel way of assessing how likely a given team was to introduce code clones in a particular component, based on their historical behavior. As this is a purely tool-based statistical approach, we believe it might lend itself to automation and integration into tools that currently rely on static rules, such as SonarQube. Another large contributor to feedback is the *comprehensive test base*, which is executed by *continuous integration* systems. However, as we see in Paper V, to make efficient use of the developed products, data should be collected end-to-end and analyzed for efficiencies. Thus, we believe that the *Feedback* theme is somewhat lacking in emphasizing the need for feedback from *existing installed systems*. As described in the paper, such feedback can steer both local (customer-specific) installations and product development directions (influencing the direction of *Product Owners*, and *Requirements* in general).

1.6.3 Limitations

Our studies were conducted within different sub-organizations of a single large-scale company (Ericsson AB) across multiple locations in Europe and India. The research focused on industrial settings rather than academic or open-source contexts, with most participants being experienced industrial practitioners. As such, contributions from less experienced developers may be limited. However, the professionalization of software development is progressing rapidly, as shown in the 2025 Stack Overflow developer survey, where 65% of respondents reported having more than ten years of coding experience[4].

Since our evidence is drawn from a single industry partner, the applicability of our findings to other industry domains, academic institutions, or open-source organizations—each operating under distinct constraints and incentives—remains uncertain. We encourage further research to explore and validate these principles in other contexts.

[4] `https://survey.stackoverflow.co/2025/developers`

1.6.4 Future Work

The principles and practices of Software Craftsmanship, with an emphasis on *Feedback*, *Iterative Design*, *Professional Culture*, and *Value-added Architecture*, are sure to persist even in the face of the oncoming LLM-powered AI wave. The need for professionals who can tame the LLMs, making them perform *the right* task, not just a *quick* task, is sure to increase. For this, there is a need to convert business requirements (related to functions as well as quality attributes such as performance, stability and robustness), into automatically verifiable test cases, which function as the "guard rails," or "boundary conditions" of the LLM-generated code.

While it is clear that LLMs are here to stay, it is equally evident that the principles of Software Craftsmanship remain essential. These principles encompass a strong focus on testing, writing minimal and clean code, fostering a professional engineering culture, and maintaining a process of continuous feedback—a need that will only grow as LLM-driven development accelerates.

At the same time, emerging technologies such as quantum computing are moving into the mainstream, potentially establishing new limits on the achievable minimum feedback time. Already today, in fields such as machine learning or artificial intelligence, even a slight change to a single hyperparameter can necessitate hours or days of retraining the model. Moreover, debugging these models is highly complex, which influences the architecture of surrounding systems—for example, to ensure strict adherence to input parameter constraints. Such fields may require additional or alternative engineering principles beyond traditional craftsmanship.

Similarly, when developing software in highly regulated domains such as medical devices or in hard real-time systems like embedded platforms, different principles may also apply. In these environments, further research is needed to determine how Software Craftsmanship practices can be effectively adapted and applied.

1.7 Conclusion

With this thesis, we have endeavored to conceptualize and quantify the impact of principles and practices associated with Software Craftsmanship. We hope that this will stimulate future research into these and provide practitioners with tools to assess the adoption of these practices within their organizations.

A common theme in Papers I–III is that *Visibility and Transparency* is important, as are the *Accountability* to **act** on what is visualized. In Paper II, we found that deprecation debt grew much slower in files where deprecations were made visible in development tools, relative to the plain-text files used for integration tests. In Paper III, we found that we could use a Bayesian Multi-Level Generalized Linear Model to attribute introductions of code clones to different development teams. The model reveals that clone introduction correlates with both change size (added/removed lines)

and the existing complexity and number of duplicates in the changed file, following the Broken Window Theory [43]. However, important variations emerged: some teams remained unaffected by existing file complexity, and teams with self-selected component ownership introduced fewer duplicates in these components. While Paper IV finds that developers in general perceive the tracing tool as useful to learn the product architecture, we also found that this type of tool is rarely used in practice. This implies that industrial use of these types of tools needs to be followed over a longer period of time than the months that we were able to do in this study.

In Paper V, we found that the most optimistic cloud cost savings fall short of the realities of migrating a large legacy product (using the "lift-and-shift" approach) to a large public cloud provider. In particular, legacy databases operate with license models that might impact the economic viability of such migrations. We also found that the *End-to-end feedback loop*, including customized solutions, is important, as is *Visibility* into what drives the cost of the solution. Thus, we postulate that the anatomy could benefit from more focus on *end-to-end feedback*, where *operational usage data* is used to drive both the product direction (e.g., *Requirements*) and the *Iterative development* processes and *Value-focused Product Architecture*.

It remains to be seen how the explosive growth and interest in LLM-powered tools will impact the tasks of a professional Software Engineer. Will the Software Engineering profession go down the path of the bowyers, coopers, and wheelwrights of medieval Europe, as LLM-driven automation replaces most human activities? Or, will the increased speed that LLMs provide enable more efficient use of time for experienced professionals, allowing them to more quickly produce better software? If so, how will these tools affect the junior developers who are not yet skilled and might take the output of these black-box machines at face value?

There are many questions, but only the future will hold the answer. We remain convinced that the interconnected principles and practices of Software Craftsmanship will play some role in the future of the Software Engineering profession.

Paper I

Towards an Anatomy of Software Craftsmanship

I

Anders Sundelin, Javier Gonzalez-Huerta, Krzysztof Wnuk, and Tony Gorschek.
In: ACM Trans. Softw. Eng. Methodol. 31, 1, Article 6 (January 2022).
https://doi.org/10.1145/3468504

Abstract

Context: The concept of software craftsmanship has early roots in computing, and in 2009, the Manifesto for Software Craftsmanship was formulated as a reaction to how the Agile methods were practiced and taught. But software craftsmanship has seldom been studied from a software engineering perspective.

Objective: The objective of this article is to systematize an anatomy of software craftsmanship through literature studies and a longitudinal case study.

Method: We performed a snowballing literature review based on an initial set of nine papers, resulting in 18 papers and 11 books. We also performed a case study following seven years of software development of a product for the financial market, eliciting qualitative and quantitative results. We used thematic coding to synthesize the results into categories.

Results: The resulting anatomy is centered around four themes, containing 17 principles and 47 hierarchical practices connected to the principles. We present the identified practices based on the experiences gathered from the case study, triangulating with the literature results.

Conclusion: We provide our systematically derived anatomy of software craftsmanship with the goal of inspiring more research into the principles and practices of software craftsmanship and how these relate to other principles within software engineering in general.

1 Introduction

I

The notion that programmers should be responsible for what they produce has early roots. Already in 1975, Brooks [14] mention "invention and craftsmanship" as prerequisites for efficient optimization techniques, and he also envisioned "the surgical team" as an efficient way of developing mission-critical software. In 2002, McBreen published a book [4], formalizing the software craftsmanship concept, and since then, several books have been written on the subject [5–8]. Another early inspirational work was published in 1999 by Hunt and Thomas [43].

The Manifesto for Software Craftsmanship[1] was published in March 2009, seven years after the Agile Manifesto[2]. The original signatories intended to address what they saw as deficiencies in how the Agile Manifesto principles had turned out in practice, as taught by coaches and certified institutions, and to emphasize the need to "make the thing right." The Software Craftsmanship movement lives on, twelve years after the manifesto was published. There are associated communities and conferences such as Socrates[3] in Europe and SCNA[4] in North America. However, we have not found any systematic definition of software craftsmanship principles and practices in research.

This article moves towards this goal by providing an anatomy of software craftsmanship based on a systematic literature study and a longitudinal case study of a software product developed by an organization that was following software craftsmanship principles. In doing so, it moves towards systematizing and making explicit the software craftsmanship principles and practices to the broader research community, as there seems to be a lack of research papers in this area, as evidenced in Section 4.

The case study subject was a unit within Ericsson developing a new software product for seven years. The product operates in the financial sector and is in use in around twenty installations around the world. Due to the stringent requirements of financial systems and the values of the developing organization, the product was developed from scratch, highly inspired by craftsmanship principles, such as test-focused, agile, and lean software development, with a high focus on clean code and refactoring. These principles were also spread to new developers joining the product.

The article is structured as follows: In Section 2, we give the background and related work of software craftsmanship and define the terms we use throughout the article. In Section 3, we report on our research methodology, with Section 3.1 focusing on the systematic literature study, Section 3.2 focusing on the case study methodology, including the studied context, and Section 3.3 focusing on the process of building the anatomy. In Section 4, we report on the results of the **Systematic Literature Review (SLR)**, and in Section 5, we merge this with the quantitative and qualitative

[1] http://manifesto.softwarecraftsmanship.org/

[2] http://www.agilemanifesto.org/

[3] https://www.socrates-conference.de/

[4] https://scna.softwarecraftsmanship.com/

results of the case study to produce our version of the anatomy of software craftsmanship. In Section 6, we discuss the implications for the software development community at large. In Section 7, we discuss the threats to the validity of the study. In Section 8, we draw on the analysis, outline future work and research directions, and make conclusions.

2 Background and Related Work

The Craftsmanship movement builds upon Agile and Lean principles and practices, but with a stronger emphasis on building high-quality products by teams with a shared professional culture. The Manifesto for Software Craftsmanship was published in March 2009, following a summit in December 2008, where around 30 participants gathered to discuss what they perceived had been lost as the software industry adopted the Agile Manifesto. In particular, the lack of focus on the more technical practices in Agile processes such as **Extreme Programming (XP)** was a concern.

There have been several books and seminal works before 2008 (e.g., the books by Brooks [14], Hunt & Thomas [43], McBreen [4], Martin [5–7] and later also Mancuso [8]) that provide insights into the concept, the practices, and the potential benefits of Software Craftsmanship. However, very few research works delve into the formalization of the concept, with its principles and practices, with buttressing, real-world empirical evidence from cases where craftsmanship principles were put into operation.

If we look at the Agile Software Development, on the one hand, there are a plethora of SLRs (e.g., [44–46]), Systematic Mapping Studies (e.g., [47]) and even Tertiary Studies (e.g., [11]) that portray how academia has studied Agile Software Development. In addition, several studies report on the benefits of Agile and XP practices in industrial settings (e.g., [48], [49], and [50]). Likewise, multiple studies address the potential benefits and drawbacks of **Test-Driven Development (TDD)**, with several experiments (e.g., [51, 52]), case studies (e.g., [49]), and SLRs (e.g., [53])

Lean Software Development was popularized by Poppendieck [54] and has been studied in an industrial setting [55, 56]. Several SLRs and Systematic Mapping Studies report results on metrics related to Agile and Lean software development and their relevance in the software industry [57–59].

3 Research Methodology

This article uses the SLR method, using Wohlin's snowballing approach [60], and a case study method following guidelines by Runeson et al. [40]. We focus on the following research questions:

RQ1 How has prior literature described the principles and practices of software craftsmanship?

RQ2 Which of the identified principles and practices can we see applied in a real-life, commercial case study?

RQ3 What are the consequences of applying these principles and practices of software craftsmanship?

We aim at answering RQ1 by performing an SLR. We aim at answering RQ2 by collecting quantitative measures on the studied system and triangulating them with interview findings with developers and the lead architect of the product. RQ3 is answered by extracting and synthesizing the literature review results and combining them with case study findings.

3.1 Systematic Literature Review Methodology and Execution

We conducted an SLR using the snowballing method described by Wohlin [60]. We used a hybrid search strategy by combining the database search with iterative citations and references analysis [61]. Forward snowballing (citation analysis) greatly improves the precision, while backward snowballing (references analysis) greatly improves the recall of literature reviews.

3.1.1 Start Set Identification

We performed a database search in Google Scholar in December 2018, using the terms "software craft" OR "software craftsmanship" OR "software craftsman" OR "software craftsmen" OR "software craftsperson." We got 980 results that were analyzed by two authors, based on the following criteria:

1. Is the paper published in an English-language journal, conference, or workshop proceedings, indexed by Google Scholar?

   This step excludes books, book reviews, and thesis works, including M.Sc. and Ph.D. theses.

2. Does the paper describe themes, practices, or otherwise conceptualize software craftsmanship?

   This step excludes papers only referring to other works, such as [5], without providing any additional detail.

Criterion 1 excluded 522 papers and criterion 2 excluded 346, resulting in 112 papers, which were screened as potential seeds. Based on analysis of the title and abstract, we selected papers discussing various aspects of software craftsmanship, which resulted in four initial seed papers, denoted P1, P2, P3, and P4. According

to Wohlin [60], the start-set should include papers from different publishers, authors, communities and should not be too small. Since diversity and scale are important for snowballing, we decided to broaden our set with relevant papers identified from our experience and recommendations, not only the database search. After some initial deliberation and analysis, we decided to add another five seed papers, denoted P5, P6, P7, P8, and P9. We also decided to drop our initial requirement to include only peer-reviewed papers since some of the included papers are magazines. At least two researchers applied the inclusion and exclusion criteria. When two reviewers had an initial disagreement, the conflicts were resolved by consensus.

3.1.2 Snowballing iterations

We performed four snowballing iterations summarized in Table 1 and stopped when we found no new relevant papers, applying the inclusion and exclusion criteria following the process described in Section 3.1.1. The full results of the SLR are available here[5].

Since the Software Craftsmanship concept comes both from the Craftsmanship Manifesto and seminal books, we extended the literature review with the final forward snowballing iteration focusing on books. In other words, we followed the references of the found papers and created a pool of books ready for analysis by partially following the guidelines for Multivocal Literature Reviews presented in [62]. This resulted in 146 books. As in the protocol we followed for "white" literature, two researchers applied the inclusion and exclusion criteria, and the conflicts were resolved by consensus. We divided the books between three of the researchers by letting each researcher analyse two-thirds of the books, making sure each book was reviewed twice. After applying the second exclusion criterion (2), we discarded a total of 135 books. The pairwise Cohen's Kappa results are 1.0, 0.59, and 0.48, which is less than the recommended criteria of 0.7. All three researchers discussed the seven books where disagreements were identified, and four of these were included in the final result after consensus had been reached. We decided not to iterate on other works citing included books since the number of citations for the included books is extremely high, and the main references from the paper-set had already been included. Section 4 contains the full results of the SLR.

3.2 Case Study Methodology

The goal of the case study is to analyze different craftsmanship practices followed in developing a product over seven years.

[5] https://tinyurl.com/Sundelin-SWC-SLR

Table 1: Snowballing Iteration Statistics and Results

Iteration	Number of citations and references screened	Included papers and books
Seed-1		P1 [63], P2 [64], P3 [65], P4 [66]
Seed-2		P5 [67], P6 [68], P7 [69], P8 [70], P9 [71]
Iteration 1	213 references and 186 citations	P10 [72]
Iteration 2	30 references and 1 citation	P11 [73], P12 [74], P13 [75], P14 [76]
Iteration 3	217 references and 517 citations	P15 [77], P16 [78], P17 [79]
Iteration 4	18 references and 78 citations	P18 [80]
Ref. Books	146 referenced books	B1 [14], B2 [4], B3 [81], B4 [5]
		B5 [82], B6 [83], B7 [84], B8 [6]
		B9 [8], B10 [85], B11 [86]

3.2.1 The Case

The product studied in the case study is a FinTech global product that enables access to financial services via mobile phones and the Internet. The system is a high-availability, transaction-intensive product, with incoming and outgoing interfaces, a database, and scheduled tasks such as sending notifications. As it operates in the financial sector, security plays a central role in development.

Our investigation focuses on the financial core, containing the core business logic, such as financial transaction management, and associated user interfaces. A deployed product also contains other components (both third-party hardware and software) and customer adaptations, which are out of our analysis scope. All other components use the services of the core to perform financial tasks. The system is built in Java, using EJB 3[6] patterns, and uses a custom framework for deployment.

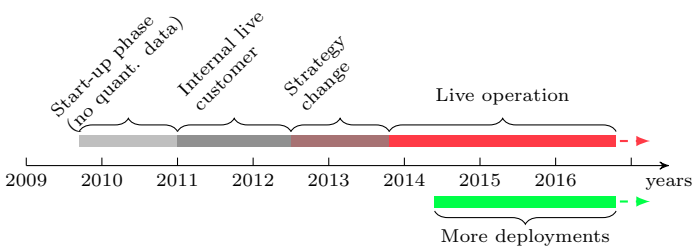


Figure 1: Timeline of major events in the studied system.

Figure 1 depicts the timeline of the studied period, together with major events in the life cycle of the product. The first line of code was written in September 2009, and the first live demo for external parties was held in late October 2009. During 2011, Ericsson's strategy was to provide the solution as a service for end-users, and the system was deployed and taken into live operation in this manner. Following a business strategy change, the company decided to decommission the service and adopt a product-line approach. In late 2013 the first installation of the product went

[6] http://download.oracle.com/otndocs/jcp/ejb-3.1-pfd-oth-JSpec

Table 2: Quarterly Developer Statistics

Metric	$\bar{x}$	σ	$Q_{25\%}$	$Q_{50\%}$	$Q_{75\%}$	Min	Max
Developers per quarter	48.9	17.7	36	48	53	25 (Q1 2011)	91 (Q2 2016)
Quarters per developer	7.6	6.5	3	5	11	1 (14 dev)	24 (5 dev)

into operation at a customer site. Subsequently, the roll-outs continued, and the product was serving several tens of millions of end-users in more than 15 deployments worldwide during 2016.

As of December 2010, there are quantitative data available in the Git Version Control System. Before that, the project used ClearCase, a licensed product whose storage is unavailable for analysis.

The initial phase of the product (between 2009 and 2011) can be characterized as "the startup phase," with frequent changes of direction and no market deployment. Between 2011 and 2013, the internal customer provided feedback on the operation and deployment of the system. When the first external customer contract was signed in 2013, and the first system was taken live later that year, the direction became more stable, with increasing inflow of customer requirements.

The product used one primary and one supporting development site for most of the studied period. From mid-2011 until mid-2012, one development team was based in China. Following a change in product strategy, in mid-2013, two development teams from India were on-boarded instead, and this continued until the end of the studied period.

During the whole studied period, ending in December 2016, the product has been developed in an agile manner, first using two-week and later three-week sprints, heavily inspired by the craftsmanship principles and practices, as discussed in Section 5.

During the studied period, 155 individual developers have contributed to the studied system (measured via the Git *Author* tag). The first author of this article was a developer from the project start until October 2016. Table 2 contains the distribution of developers per quarter and quarters per developer. On average, 48.9 developers contributed to the code base each quarter. The peak of activity was reached in Q2 2016 with 91 contributors. In total, 24 quarters were studied, and in 75% of these, more than 36 authors contributed code. This clearly shows that the product is larger than what a single agile team can accommodate, requiring inter-team collaboration and communication.

On average, each developer stayed almost two years (7.6 quarters) in the product, though 50% of the total 155 authors contributed five quarters or less, and 25% contributed three quarters or less. This turnover data for the studied period show similar characteristics as the cases reported in previous research in the area [87]. The distribution is slightly right-skewed, as indicated by the minimum and maximum values, with five authors contributing during all 24 studied quarters and 14 authors

Table 3: Case Study Interviewee Background, Ordered by Industry Experience

Legend	Description	Experience
SwArch1	Lead Architect	20+ years in industry, 8 years in the product, starting 2009
Test2	Test-focused developer and Scrum master	≈20 years in industry, 8 years in the product, starting 2009
Test1	Test-focused developer and Scrum master	≈15 years in industry, 2 years in the product, starting 2015
Dev2	Developer	≈15 years in industry, 4 years in the product, starting 2013
Dev1	Developer	≈10 years in industry, 4 years in the product, starting 2013
Dev3	Developer and Scrum master	≈10 years in industry, 5 years in the product, starting 2012

contributing during a single quarter.

Although most contributors have been software developers, more persons and roles such as requirement engineers, system testers, product-, project- and line managers have contributed to the product. These roles are not studied in this article.

3.2.2 *Data Collection*

We used two data collection methods. We gathered qualitative data through interviews with different roles involved in developing the product at different points. We also gathered data using archival analysis, using different artifacts (e.g., Version Control Systems, documentation, requirements, and defect reports) to measure the potential effects of craftsmanship practices on the product and the development process. We interviewed six participants for this case study, two female and four male subjects. Two of the interviewees worked in India, and four worked at the primary development site. Table 3 details the participants' background, as well as the legend used in citations and tables.

The interviews were organized as semi-structured interviews, using the interview instrument to structure the discussion. The interview protocol, which is publicly available here[7], was built and reviewed by the researchers and adapted as the interviews progressed to focus more on each interviewee's areas of expertise. At least two researchers conducted all the interviews, intervened in the discussion at will, clarifying statements, and introducing new topics and areas. All the interviews were audio-recorded and transcribed before analysis.

3.3 Consolidated Data Analysis: Building the Anatomy

In this subsection, we describe how we analyzed both the SLR and case study results.

The interview transcripts and the SLR results were analyzed using **Thematic Analysis (TA)**, following the guidelines by Braun and Clarke [88]. We opted for TA since we were not exploring a completely alien phenomenon (i.e., Software Craftsmanship). Therefore there is no need to build an entirely new theory that emerges directly from the data, as is one of the main strengths of Grounded Theory [89], that in general is better suited to answer broader questions, such as *"what is going on*

[7] https://tinyurl.com/Sundelin-SWC-Interview

there?" [90]. TA is a robust and systematic framework for coding and analyzing qualitative data, identifying patterns across datasets in relation to research questions [91]. TA is also best suited when most of the collected data belong to a precise context, which then will move to generalizations and finally will allow building theories [92]. We carried out a theoretical or *deductive* approach for Thematic Analysis[88] by starting with a *theory* (a set of codes and themes), updating this as new data emerged.

Figure 2 summarizes the process for building the Anatomy of Software Craftsmanship. We first generated the initial set of codes (i.e., craftsmanship principles and practices), represented in the form of a mind-map (i.e., the Anatomy). This first set of codes was built based on the Software Craftsmanship Manifesto and themes from books, as indicated in Table 5. The first author then discussed the initial anatomy with the other authors in devil's-advocacy-type sessions.

Then the papers and the books included from the SLR were analyzed and coded, searching and reviewing the emerging codes and themes. When coding the books included as gray literature, two researchers read each book. Once the coding was finished, the two researchers met to discuss the codes found and went through the coding conflicts, which were solved by consensus.

The next step was coding the interview transcripts. The first author performed the initial In-Vivo coding [93] of all six interviews. Next, the second and third authors independently coded three transcripts each, assuring that at least two independent researchers coded each interview, prioritizing the interviews in which each researcher was present. Once coding was finished, the researchers met to discuss the potential coding conflicts, which were resolved by consensus. The coding was done using the corresponding version of the Anatomy with the codes. During the coding process, codes were merged, renamed, and new codes and themes were identified and added to the Anatomy, as suggested in Figure 2. This process triggered the need to review the already coded materials to identify potential instances of the new codes and themes in the data.

Taking the "Requirements" concept as an example to illustrate the process:

1. The first author of this article had experience from the case study, as well as noting the importance of localized customers, as stated in several of the reviewed books, see Table 5. Based on this, he initially decided on the code *On-site customer*, as it is a concept from XP [94] that aligns well with the requirements process of the case study. After discussions with the additional authors, this code was used to explore the SLR results and to guide the interviews. However, neither the coding system nor the tentative map was shown to the interviewees before the interview.

2. Both books B2 [4] and B8 [6] mention the importance of communication between development teams and requirement owners, indicating that the requirements concept should be somewhere near the Feedback theme.

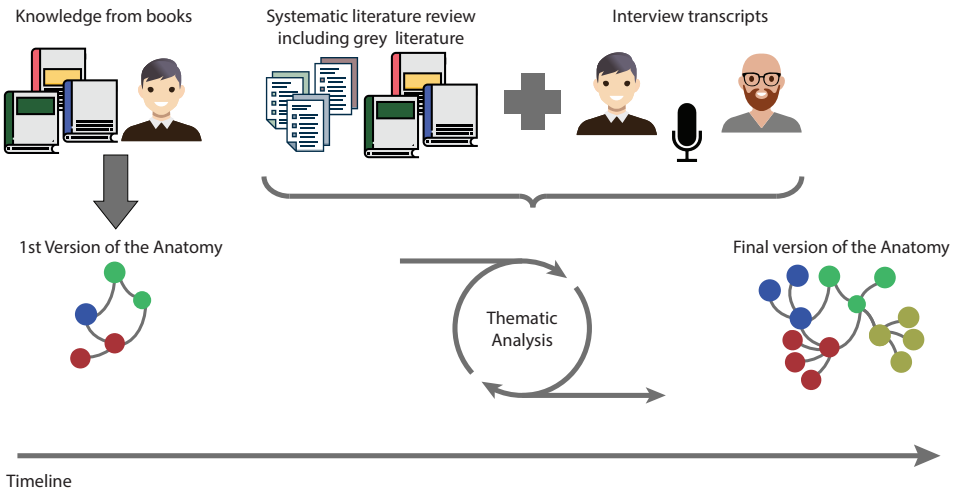


Figure 2: Process for Building the Anatomy of Software Craftsmanship.

3. Furthermore, while conducting interviews, evidence was made more apparent that requirements were written in cooperation between the developers and the *On-site-customer*, though the case study used the Scrum term **"Product Owner" (PO)**. This was mentioned by several interviewees, for example, "We had our requirements in [the wiki-based requirement tool]. And the PO owned them—or the team—sometimes the team helped formulate them. But you walked through them [with the PO]. In [a different product], where I am now, it is completely different…"(Test2)

4. Two other interviewees (Dev1, Test1) also indicated that the requirements were collaborative, mentioning the importance of looking "top-down" while simultaneously keeping a "bottom-up" perspective. This was also found in seven books and two papers in the literature, see Table 11 for details. In B2 [4], McBreen cites a study by Curtis, Krasner, and Iscoe, where this was stated as "Characteristically, customers also underwent a learning process as the project team explained the implications of their requirements. This learning process was a major source of requirements fluctuation." [95]

5. The importance of *Accessible* requirements was also made clearer during the interviews. Having a clear, accessible requirement base was important for being able to work in parallel: "A strength in [the case study project] was that we could start testing in parallel with development. And we had clear requirements in one place [the wiki-based requirement tracking tool]. Based on this, the developers did their analysis, and testers did theirs in parallel. So we could write our acceptance test cases while development was ongoing."(Test2). Another interviewee supported this claim, and eventually, the *Accessible* require-

ment code was also found in book B2 [4] and papers P3 [65] and P9 [71].

6. Based on these data points, we decided to add the **F1.1.2** *Collaborative* and **F1.1.1** *Accessible* practices to the **F1.1** *Requirements* practice, connected to the original *On-site-customer* principle. The decision to keep the whole sub-tree in the **F** *Feedback* theme was confirmed while analyzing additional data, such as when an interviewee discusses interactions between the requirements owner and the development team: "I would say we talk to [the requirements owner] every day, almost…Or, maybe at least for half an hour every other day…It's quite often we encounter things, in code and so on, that is not really how the requirement was imagined…Then you have to discuss that."(Test1). In total, four interviewees, three books, and three papers confirmed the importance of **F2** *Short feedback loops* between requirements engineers (regardless of title or term used), the development team, and the verification engineers.

7. While this article was in revision, a reviewer rightly pointed out that our so-called "On-site customers" were not really customers, but mere proxies for real, paying customers. Therefore, we decided to rename the principle to **F1** *On-site customer (proxies)*, indicating that sometimes you have to work with proxies for real customers (or end-users).

To increase validity and get feedback on our work, we shared the interview transcripts with the interviewees to ensure that we properly captured their opinions. We also presented an intermediate version of the craftsmanship map for company employees, including those currently working with the product. This provided valuable, though unstructured, feedback, which validated our structure.

We used statistical methods such as descriptive statistics and graphical representations to analyze and describe the case study's quantitative data.

4 Systematic Literature Review Results

In this section, we summarize the main findings of the SLR. Table 4 outlines the results of the analysis of papers P1 to P18. Only 6 out of 18 papers can be considered empirical studies. Opinion papers and personal experience papers dominate the non-empirical studies and receive rigor scores between 0 and 1 and relevance scores between 1 and 2, making these papers partly relevant for our work. We used rigor and relevance criteria proposed by Ivarsson and Gorschek [96]. Rigor can have scores from 0 to 3 and is related to describing the context (maximum 1 point), study design (maximum 1 point), and validity (maximum 1 point). Relevance can have scores from 0 to 4, considering industrial participants (max 1 point), industrial context (maximum 1 point), realistic size of the study (maximum 1 point), and the usage of research methods that facilitates investigating real situations (maximum 1 point).

Table 4: Papers Resulting from the Systematic Literature Review

Paper [ref]	Found in	Refs	Cited	Rigor	Relevance	Venue	Year	Empirical	Main contribution
P1 [63]	Seed1		P10	0	0	Journal	2003	No: vision paper	Craft metaphor for software creation
P2 [64]	Seed1			4	3	Journal	2013	Yes: qualitative and quantitative, longitudinal study	Craftsmanship forums and chats as a part of community of practice
P3 [65]	Seed1			3	3	Conf.	2013	Yes: questionnaire and focus groups	Community of practice as a part of software craftsmanship
P4 [66]	Seed1			2	3	Conf.	2014	Yes: qualitative interviews and focus groups	Different conceptualizations of craft in building software
P5 [67]	Seed2			1	4	Workshop	2016	Yes: experience report	Analyzes software craftsmanship values in a Scrum project
P6 [68]	Seed2			0	2	Magazine	2014	No: opinion paper and anecdotal evidence	Engineering is craft supported by a theory
P7 [69]	Seed2			0	2	Non-academic conference	1994	No: experience report mostly based on anecdotal evidence	Stresses the importance of craftsmanship
P8 [70]	Seed2			0	2	Non-academic journal	2003	No: opinion paper	Discusses general craftsmanship and software craftsmanship models
P9 [71]	Seed2			1	2	Workshop	2008	No: personal experience	Focus more on agile than craftsmanship
P10 [72]	Iter1	P11, P12, P13, P14	P1	2	3	Journal	2015	Yes: qualitative and quantitative surveys	Community of practice and software design
P11 [73]	Iter2		P10, P16	1	2	Journal	2013	No: theoretical	Epistemology of craft in modern programming
P12 [74]	Iter2		P10	0	1	Non-academic journal	2010	No: opinion paper	Katas as a part of craftsmanship
P13 [75]	Iter2	P15	P10	2	2	Magazine	2014	Yes: experiment using katas	Katas as a way of learning and personal improvement
P14 [76]	Iter2		P10, P16, P17	0	1	Conf.	2006	No: opinion paper	The birth of the crafting paradigm preceding SE in the 1960s
P15 [77]	Iter3		P13	1	2	Conf.	2012	No: personal experience of the course instructor	Courses that involve craftsmanship practices
P16 [78]	Iter3	P11, P14, P17	P18	0	1	Conf.	2016	No: observations of the authors	Professional practice is craftswork
P17 [79]	Iter3	P14	P16	1	1	Workshop	2012	No: previous version of P11	Previous version of P11
P18 [80]	Iter4		P16	0	1	Journal	2008	No: personal opinion paper	Mentions craftsmanship in the history of SE

Among the non-empirical papers, two papers view craftsmanship from the perspective of the history of software engineering. Among them, P18 gives a brief history of Software Engineering, referring to Dijkstra declaring programming to be a discipline rather than a craft. Paper P14 also looks into the history of Software Engineering and uses the term "software crafting" to describe the (lack of stringent) processes for programmers during the 1960s.

On the philosophical stance, papers P11 and P17 discuss the theoretical underpinnings of the epistemology of craft in modern programming. Paper P1 provides a similar discussion, advocating that software methods should find ways of incorporating vernacularism and objects to a strictly rational software design process.

Six non-empirical papers present opinions, visions, or experiences. Among these, paper P6 argues that engineering is a craft supported by theory, while paper P16 argues that professional practice is craftwork. Paper P8 discusses the general craftsmanship model and the software craftsmanship model. Paper P7 highlights the importance of craftsmanship. Paper P9 focuses on the relation between agile and craftsmanship, and paper P12 brings opinions about using katas. Paper P15 summa-

rizes experiences holding a course involving craftsmanship principles.

None of the six empirical papers takes a holistic view of software craftsmanship. Instead, they focus on practices (e.g., a community of practice for papers P3 and P10; craftsmanship forums and chats for paper P2; using katas to learn and improve for paper P13).

Empirical papers P4 and P5 are the closest to this work. Paper P4 empirically derives different conceptualizations of craft in building software, using a sample of 12 participants, whose subjective opinions were collected via interviews and a focus group. Paper P5 attempts to outline the craftsmanship practices based on the experiences from a project run with Scrum. The paper discusses steadily adding value vs. responding to change, a community of professionals, customer collaboration, and productive collaboration. Despite being highly relevant, paper P5 appears to be an experience report from a project manager's point of view, providing quantitative analysis of technical debt (number of lines removed over time) and velocity in backlog hours versus tool estimated technical debt. However, the paper lacks systematic connection between the presented experiences and evidence. It appears that it is one person's experience that summarizes what the team has done rather than interviews with team members triangulated with quantitative data analysis.

Table 5 contains the books found in the SLR, with the books used by the first author to build the initial anatomy map marked in boldface. Many books (e.g., B1, B4, B8, and B9) describe personal experiences from skilled software development professionals. Others, such as books B3 and B11, detail process patterns for large-scale organizations, whereas book B7 contains transcribed semi-structured interviews with 15 senior developers, focusing on their personal development experiences and opinions. Books B2 and B5 are more philosophically inclined, and book B10 describes experiences from teaching XP and pair programming using deliberate practice.

To the best of our knowledge, this article is the first attempt to empirically derive the anatomy of software craftsmanship based on a more encompassing view of the seminal books, supplemented by academic literature in the area, and buttressed by insights from an in-situ longitudinal industry case study.

5 The Anatomy of Software Craftsmanship

In this section, we present the concept map, synthesized from the analysis of the case study findings and the SLR results. Figure 3 depicts four themes with associated principles and practices as interconnected nodes.

The ***A*** *Value-focused architecture* theme has three principles (**A1** to **A3**) with ten associated practices (**A1.1** to **A3.4**). The ***D*** *Iterative design, development, and verification* theme has three principles (**D1** to **D3**) with ten associated practices (**D1.1** to **D3.2**). The ***C*** *Shared professional culture* theme has six principles (**C1** to **C6**) with 18 associated practices (**C1.1** to **C6.3**). The ***F*** *Feedback* theme has five principles

Table 5: Books Resulting from the SLR. Boldface Books Influenced Initial Anatomy.

Book [ref]	Cited	Year	Main contribution
B1 [14]	P14 P16	1975, revised 1995	Originally published in 1975, the referenced version was published for the twentieth anniversary and also includes subsequent essays on software engineering. Details experiences from the development of the IBM System/360 in the 1960s, where the author was the project lead.
B2 [4]	P1 P8	2002	Argues that craftsmanship is a better metaphor for software development than software engineering, which is described as focusing on multi-year, large-scale, low-skilled-developer projects.
B3 [81]	P3	2008	While focusing on patterns for using Scrum and Lean practices in large-scale system development, the authors also illustrate the importance of skilled developers that practice their craft, mentoring less-skilled peers.
B4 [5]	P6 P11 P17	2008	Personal experiences from the authors are combined with a set of concrete rules, exemplified in Java, to create a catalog of smells and heuristics, including remedies.
B5 [82]	P4	2008	Philosophical book, arguing that Linux and other open-source projects embody the spirit of craftsmen, as epitomized by the hymn of Hephaestus.
B6 [83]	P13 P15	2009	Originally sourced from a wiki, this book describes Software Craftsmanship as a pattern language, centered around learning themes such as ``emptying the cup'', ``walking the long road'', ``accurate self-assessment'', ``perpetual learning'' and ``construct your curriculum''.
B7 [84]	P11	2009	Contains 15 interviews the author conducted in 2008 with leading developers from the 1960s until today. Of the 11 interviewees asked, eight would identify software development as a ``craft'' Other opinions voiced were: ``art'', ``mathematics'', ``science'', ``engineering'' or ``a style of writing''.
B8 [6]	P6	2011	Using the author's experience as an example, describes rules and principles for professionalism in committing to a task, developing, testing, and dealing with teams and people under delivery pressure. Advocates for practicing and mentoring as tools to reach higher productivity.
B9 [8]	P6	2014	Wide treatment of Software Craftsmanship, ranging from personal experiences, professional attitude and technical practices to how to interview for recruitment and foster a culture of learning.
B10 [85]	P10	2014	Describes best practices and lessons learned while teaching the four rules of simple design [97] via code kata exercises for various groups of people over the course of five years.
B11 [86]	P6	2015	Blends the two fields of Agile software development and Human Performance Technology, a field closely related to human resources and learning professionals, described in 1978 by Gilbert[98]

(**F1** to **F5**), with nine associated practices (**F1.1** to **F5.2**).

Some practices are connected to more than one principle, indicated in the figure via interconnected edges. Some practices are hierarchical. For instance, the practice ***F1.1*** *Requirements* contains the sub-practices ***F1.1.1*** *Accessible* and ***F1.1.2*** *Collaborative*, indicating that the requirements gathering and clarification process was performed in collaboration between the requirements engineer ("On-site customer") and the development team.

The principles are presented together with the supporting empirical findings found in the literature and the case study.

5.1 **A** Value-focused architecture

The software craftsmanship manifesto states as a principle: "Not only responding to change, but also steadily adding value," and a well-crafted system should have a software architecture that enables this goal.

Table 6: References to ***A*** *Value-focused architecture*

Id	Name	Books	Literature	Qualitative
A1	Participating Software Architects	B1, B2, B6		SwArch1, Dev1, Dev2, Dev3, Test1
A1.1	Passionate	B2, B3, B6, B7, B8, B9	P1, P8	SwArch1, Dev3, Test2
A1.2	Skilled core	B1, B2, B3, B6, B7	P1, P8, P11, P15, P17	SwArch1, Dev1, Dev2, Dev3, Test2
A1.3	Empowerment	B1, B2, B3, B9	P3	Dev3, Test1, Test2
A1.4	An architect also implements	B3, B9	P7	SwArch1, Dev3, Test1, Test2
A1.4.1	Working Example		P5, P7	Dev2, Dev3, Test2
A2	Encapsulation & separation of concerns	B1, B2, B4, B7, B8 B10	P11, P13	SwArch1
A2.1	Isolated and layered architecture	B2, B4, B6, B7, B10	P3, P11, P18	SwArch1, Test1
A3	Clean, minimalistic code	B1, B2, B3, B4, B7, B9, B10, B11	P5, P11, P15, P17	SwArch1, Dev3, Test2
A3.1	Minimalistic frameworks	B2, B4, B7	P4, P8, P11	SwArch1, Dev1, Dev2
A3.2	Judicious use of third-party-products	B2		SwArch1
A3.3	Common application patterns	B3, B7, B10		SwArch1, Dev1, Dev2, Dev3, Test2
A3.4	Refactoring	B1, B3, B4, B6, B7, B8, B9, B10, B11	P3	SwArch1, Dev1, Dev2, Dev3, Test1, Test2

The three principles and ten practices related to value-focused architecture are listed with references in Table 6. To enable the value-focused architecture, software architects have to participate in guiding the team into a modular and layered architecture, where changes do not ripple across subsystems, and code is kept clean and as simple as possible through refactoring. The first rule of refactoring [99] is that there must be sufficient test coverage before it occurs, so the architecture should also enable the development of a comprehensive, layered test base.

A1 Participating Software Architects

- **Literature:** Brooks, in B1 mentions the *chief programmer* as a role which today could be called lead software architect, and discusses the benefits of *conceptual integrity*, by using a "small architecture team."

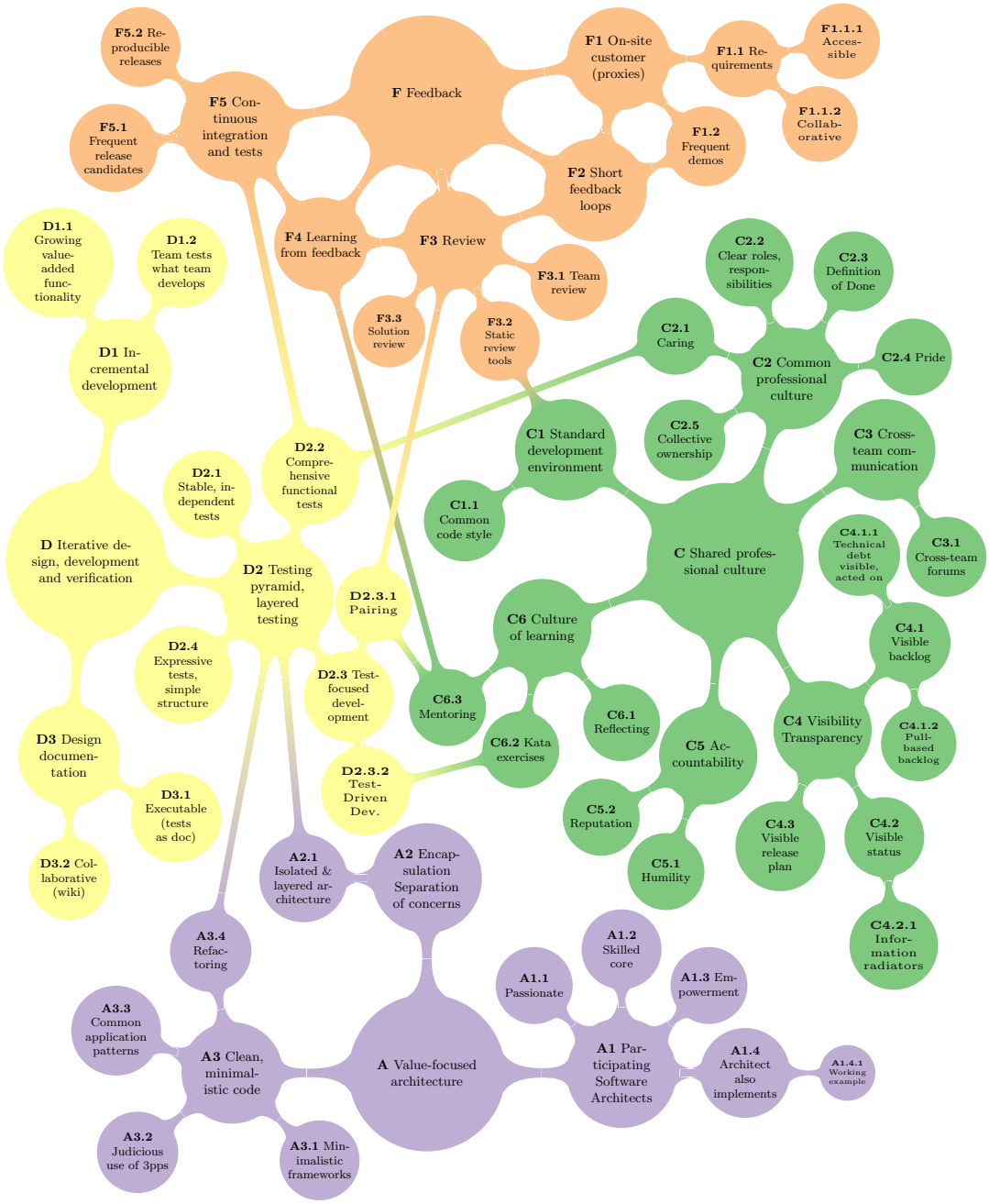


Figure 3: The anatomy of Software Craftsmanship.

Books B2, B6, and B8 also discuss the importance of architects that participate in the end-to-end solution, for instance, by specifying and giving examples of integration tests. Looking outside the SLR scope, Hunt and

Thomas [43] calls the role "technical head," tending to the big picture, and Martin [7] states that software architects need to participate in the development to spot problems and guide directions.

Books B1, B3 and paper P3 refer to team empowerment in the context of cross-functional teams [100], while book B9 states that an empowered a team of craftsmen can be the difference between project success or failure. Book B2 states that users should be empowered to interact with developers, who know how to use this to deliver robust applications.

Paper P7 mentions the *constant attention to architectural issues* and *lead developers that participate* in the product from early prototypes to delivery. Paper P5 states that their product used an *initial domain model* and *an early definition of basic architectural mechanisms*.

The importance of skills and passion for the craft is discussed in seven books and eight papers, as depicted in Table 6, e.g., paper P2 elaborates on the role of a passionate leader in increasing engagement.

- **Empirical findings:** The studied system had the same chief software architect, who implemented a lot of code, including a minimal container framework, based on partial support of EJB 3 standards. "I tried not to interfere too much with the teams. Instead, I tried to ensure that the platform they were building on was stable and good enough, so whatever they did, they will most likely get it right. Because that reduces the load on me and my team."(SwArch1)

  As the product grew beyond two teams, one senior developer from each team was designated **"Team Architect" (TA)**, with the intent to spread the knowledge from the chief software architect. This is further discussed in item **C3**, and similar to the one reported in [101].

  Teams were empowered to come up with their own solutions and to improve on existing solutions. The TA group also had some votes in the resource planning, for instance, regarding "onboarding" procedures for the outsourced teams, as mentioned in item **C2**.

  Several interviewees mention the passion and the pride they took in the product, e.g. "We cared a lot for our product. Some people ended up in different areas…Some features were like one's nursing child."(Test2)

  Team architects were expected to both participate in the team's daily work and mentor them into a coherent way of working: "[Our team was formed by] mixed newly graduates and senior developers. And our TA, I guess he preached a lot. He has gotten me into Domain-Driven Design. During my education years, I was using strings everywhere. So, he really opened my eyes to the benefits of DDD. And now, I try to spread the word [to my new team]."(Dev3)

There are also contradictory views that the product was lacking a communicated architectural vision: "My dream architect should know the code, know how we want it to work, and also say 'Now when you are into this part, I want you to think about this also, improving, preparing for future…' And also being able to delegate this."(Test1)

- **Analysis:** Striking the correct balance between participating and empowering is not trivial. While Bass et al. [102] do include "Implementing the product" and "Testing the product" as two of the ten technical duties of a software architect, they also list seven non-technical duties, nine non-technical skills, and ten knowledge areas that should be mastered.

  In the studied case, the developers showed lots of passion for the product and worked together towards the same goal. However, there were still expressions that there was a lack of a communicated vision and a desire for tasks and responsibilities to be delegated more.

A2 Encapsulation, Separation of concerns

- **Literature:** Encapsulation is the materialization of one of the most traditional Software Engineering principles: "the separation of concerns" [103]. While developing a complex system, there is a need to develop and evolve different parts of the system independently [102]. The layered architectural pattern is the most widely spread practice for architectural subdivision [102, 104]. The pattern segments the software systems in a way that enables modules to evolve and be developed separately so that each module has only one main reason to change.

  Five books in the SLR findings (B2, B4, B6, B7, B8, and B10) advocate proper encapsulation, loose coupling, and isolation of changes. Book B2 explicitly mentions that designing for testability is important because it discourages coupling and encourages cohesive module design. Outside the SLR findings, Richards et al. state that layered architectures increase the efficiency of testing [105]. Papers P3 and P11 state *simplicity* as a key trait of craftsmanship.
- **Empirical findings:** One of the first architectural decisions was to rely on an EJB 3-alike application framework, developed internally, to solve product requirements regarding installation, upgrades, and configuration. The framework is further discussed in item **A3**.

  The architecture enforced business logic to be split into interfaces and implementations and used dependency injection, using naming patterns to reduce the need for boiler-plate configuration. Inter-process communication initially used serialized Java objects, though this was later replaced with an XML-based interface, supported by a schema definition language.

This change made it easier to enforce backward compatibility across different protocol versions by defining a published protocol that was shared with external parties.

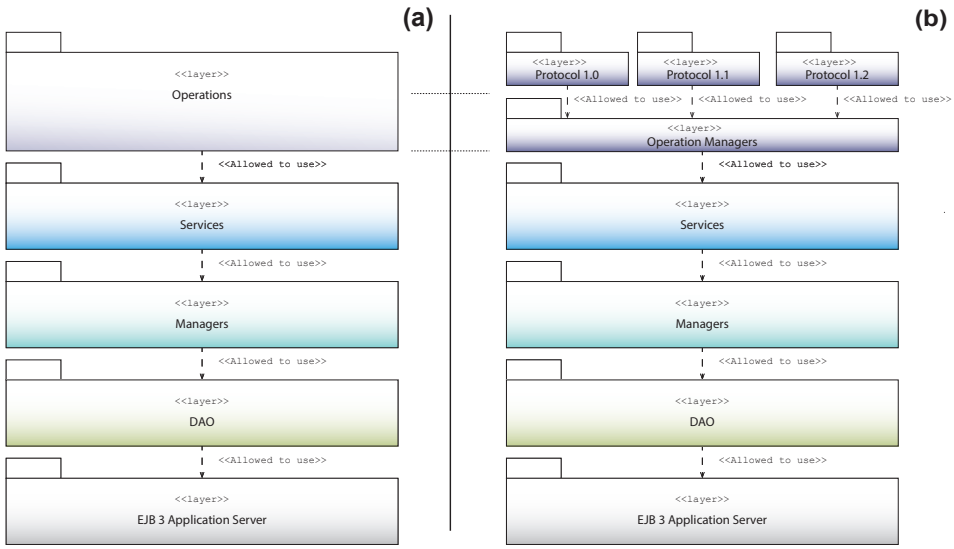


Figure 4: Layered view of the Initial architecture (a) and Layered view of the Architecture after separating protocols from business logic (b).

Figure 4 (a) depicts the initial layered architecture using UML[8] stereotypes packages as layers and stereotyped *allowed-to-use* UML dependencies, as suggested in [106]. The application server is represented as a bottom layer in this figure, although it also supports all layers with cross-cutting concerns, such as transactions, security, and logging. The **Data Access Objects (DAO)** encapsulate the access logic to the database, and upper layers add business logic and protocol support.

When faced with the problem of supporting clients using earlier protocol versions, the suggested solution was to add another layer in the architecture, as depicted in Figure 4 (b). The old "Operations" layer was split in two, where the new "Operations Manager" layer contained code common to the different versions of each operation, and the protocol version layer converted between the specific protocol versions and the operations layer.

The lead system architect had strong opinions about the architecture: "If you look at each service, it has a normal, layered architecture, because everything else is wrong."(SwArch1)

[8] http://www.uml.org

He also discussed the architecture's tree-based structure: "The dependencies between the different services should look like a tree because it's easier."(SwArch1)

Particular care was taken to separate the architectural framework from the business logic: "The bottom layer is, of course, just an interface. You don't rely on implementation because implementations can change. Then you build data access on top of that, then on top of that you build managers and compound features, and so on."(SwArch1)

Architecture should simplify the creation of business value. This includes "making it easy to make the right decisions" such as container-managed transactions and no explicit threading in business logic. It also should simplify wanted non-functional aspects, such as simple unit and integration testing, a defined data model management policy, absolute transaction security, and scaling. This was mentioned as beneficial by three developers: "There was a good framework at the product level, so you avoid doing things which are wrong."(Dev2)

"[Application developers] should not need to know everything that is behind the scenes. If they need to see it, then something is wrong. Then we haven't described a certain interface good enough."(SwArch1)

The desire to simplify testing was also a driving factor: "…[listeners are used as] reversed dependency injection, to inject behavior that is needed for a particular customer…instead of trying to mush everything into the same thing. Because that will take a longer time to build, longer time to test. It will be a lot more complex to understand, and it won't be readable."(SwArch1)

Layered architecture also supported business flexibility, allowing the system to be customized for different installations while keeping a stable core. All deployments used the same core engine with customer-specific adaptations added as optional packages.

- **Analysis:** Following software craftsmanship principles means focusing on simplicity and testability when making architectural decisions. Similarly, the developers were supported in their evolution of the system through the hiding of unnecessary detail and having clear interfaces to features, affecting both functional and quality attributes. The architecture supported the smooth replacement of deployed code, data models, and existing data, showing that there was a *long-term commitment* to the product.

A3 Clean, "minimalistic" code

- **Literature:** As detailed in Table 6, eight of the studied books describe the

importance of keeping the code clean and the design simple. Books B2, B4, and B7 advocate for minimalistic frameworks, and B2 also mentions that care should be taken when choosing to depend on other products. Both paper P12, and books B3, B7, B10, [7] exemplify and describe the importance of using common application patterns to communicate a design. However, in book B7, one interviewee (Brendan Eich) concedes that he never bought the Design Patterns book [107].

Nine books list refactoring as the key principle to achieve a clean codebase, indicating that clean code typically arises from successive refinements; it is not written directly. This is also stated by Hunt and Thomas [43]. Paper P3 also mentions *refactoring* as a principle of software craftsmanship. According to Fowler et al.[99], refactoring involves "improving structure without affecting existing functionality."

Papers P5, P11, P15, and P17 mention *clean code* principles, using *exploratory programming* and *reflections* to make the code cleaner.

Papers P4, P8, P11, P17, and P18 discuss how *tools are important* to a craftsman and how to *fight against homemade complexity*, using *clean abstractions*. Of particular importance is the ability to *choose the tool* based on the task at hand.

Paper P12 mentions the importance of *understanding the styles, idioms, and patterns* to be effective in a language and how the Lisp and APL[9] communities have championed the use of *kata-like exercises* to spread common idioms for developers to be productive.

- **Empirical findings:** Both items **A1** and **A2** mention the in-house developed architectural framework. In early 2011, the framework consisted of 299 Java files totaling 19 kLOC production code, which grew linearly (LOC p-value $< 2 * 10^{-16}$, $R^2 = 0.968$) to 72 kLOC Java production code in 1027 files in late 2016. This is clearly fewer lines of code than, for example, the JBoss (also known as Wildfly[10]) application server, which in its 7.0 release (July 2011) comprised 2886 Java files, totaling 205 kLOC, and the 10.1 release (Aug 2016) comprised 7272 Java files, totaling 433 kLOC.

  The importance of the minimal framework was stated by the chief software architect: "…all these application servers, they have to support 100% of the standard. The difference with us is that we support the 5% that we need…System handling, such as installing, upgrading, configuration, and so on is usually not covered in the normal application servers."(SwArch1) Another driving force of the framework was the ease

[9] https://en.wikipedia.org/wiki/APL_(programming_language)
[10] https://github.com/wildfly/wildfly

of development: “[The foundation] is built so that it is easy to develop and debug, also locally, on your local laptop. You have the basic services, cross-functional things with interceptors, and so on. The application developer should be able to focus on the value for the customer.”(SwArch1).

In the project, all interviewees mention refactoring as a used practice, though two say that it has to be “hidden” in the normal work rather than being a planned activity. One interviewee stated that refactorings larger than a week have to be planned, but smaller ones take place “in the regular feature work.”

Several interviewees also mentioned the desire to refactor more, to clean up more, but states that the balance tends to tilt towards finishing the current feature.

The project required developers to use strict commit messages, including the reason for the change. Possible reasons for a change included feature development, spontaneous or official (documented) bug fixes, spontaneous refactorings, or build-related changes (e.g., preparing for releases or version changes). Table 7 shows the percentages of commits of the different sorts on the master branch, not including back-ported commits to maintenance branches. The table shows that the refactoring percentage of commits varied between 27% and 7% each quarter, with both mean and median around 16%. The number of fault correction commits was lower, between 22.3% and 6.3%, with a mean of 12.6% and a median of 12.4%.

Table 7: Summary Statistics of the Proportion and Type of Main Branch Commits per Quarter

Metric	$\bar{x}$	σ	$Q_{25\%}$	$Q_{50\%}$	$Q_{75\%}$	Min	Max
Commits per quarter	3362	1189	2699	2994	3767	1090 (Q4-2016)	6361 (Q4-2015)
Feature development	52.8%	10.6%	46.1%	54.3%	59.2%	28.8% (Q1-2011)	74.5% (Q4-2015)
Refactorings	16.8%	4.5%	14.7%	16.6%	18.2%	7.7% (Q2-2014)	27.6% (Q1-2011)
Fault corrections	12.6%	3.3%	10.9%	12.4%	13.9%	6.3% (Q4-2015)	22.3% (Q4-2012)
Build related	16.8%	6.5%	12.9%	13.6%	19.0%	8.8% (Q4-2015)	30.6% (Q4-2016)
Unclassified	0.2%	0.1%	0.2%	0.2%	0.3%	0.1% (Q1-2013)	0.5% (Q4-2013)

There were concerted efforts to clean up the code in the project and keep a consistent style throughout the codebase. As mentioned by one of the respondents, the developers should “…strive to leave the code a little cleaner than you found it.”(Dev3)

In the project, several interviewees mention the help they got from the well-defined application patterns used in the product, including the security patterns (encryption, key management, and fingerprinting). “Identify the patterns. Actually, you have thousands of classes and code, but you can summarize them into one or two use cases. You need to have examples….”(Dev1)

- **Analysis:** The results regarding refactoring, see Table 7, confirm that the organization was consistent in refactoring and in keeping the constant improvement culture. Both refactorings and spontaneous bugfix percentages were higher at the beginning of the project when the codebase was smaller and more volatile. However, the inter-quartile range indicates that during 12 of the studied 24 quarters, the ratio of spontaneous refactoring commits varied between one in seven ($\approx 14\%$) and two in eleven ($\approx 18\%$).

  Others have studied the effects and efficiency of refactoring operations embedded in feature development (e.g. [108, 109]).

Summary: The architecture of a system developed with craftsmanship in mind should strive to *maximize value-creation* over a *long-term commitment* to the product. The way to achieve this is to develop and frequently validate a *comprehensive regression test base*, enabling developers to *refactor* the codebase into a *clean and simple representation*. It is as important to *care* for the test base as for the production code.

5.2 **D** Iterative design, development, and verification

The first principle in the software craftsmanship manifesto states, "Not only working software but also well-crafted software." The practices outlined in Table 8 are centered on verification and iterative refinement of the software and its requirements. There are also dependencies to an architecture focused on testability and clean code; as stated succinctly by Martin[6] in book B8: "The fundamental assumption underlying all software projects is that software is easy to change. If you violate this assumption by creating inflexible structures, then you undercut the economic model that the entire industry is built on."

D1 Incremental development

- **Literature:** Ten of the studied books relate to an incremental development in some form, and the majority of them refer to "growing" software rather than "building," "designing," or "architecting" software, see Table 8. This implies that software construction is an act of successive refinement, where the software is constantly tended to and updated as the requirements or environment changes.

  Papers P1, P7, P11, and P17 discuss the *iterative development* and the *moving between designing, making, evaluating, and reflecting phases* of software development.

  Papers P1, P3, P5, P7, and P17 mention *prototyping* and how *testing is done in parallel with development*.

Table 8: References to ***D*** *Iterative design, development, and verification*

Id	Name	Books	Literature	Qualitative
D1	Incremental development	B1, B2, B3, B4, B5 B6, B7, B8, B9, B10	P1, P11, P12, P17	SwArch1, Dev1, Test2
D1.1	Growing value-added functionality	B1, B2, B3, B4, B5, B7, B9, B10	P1, P3, P5, P7	SwArch1, Dev1, Dev2, Dev3, Test1, Test2
D1.2	Team tests what team develops	B3, B7, B8, B9, B11	P3, P17	Dev1, Dev2, Dev3, Test2
D2	Testing pyramid, layered testing	B1, B2, B3, B4, B6, B7, B8, B11	P1, P3, P5, P11, P17	SwArch1, Dev2, Test1, Test2
D2.1	Stable, independent tests	B8, B9, B10		SwArch1, Dev1, Test2
D2.2	Comprehensive functional tests	B1, B2, B3, B4, B7, B8	P3, P11, P17	SwArch1, Dev1, Dev2, Dev3, Test1, Test2
D2.3	Test-focused development	B2, B7, B9		SwArch1, Dev1, Dev2, Dev3, Test1, Test2
D2.3.1	Pairing	B3, B6, B7, B8, B9, B11	P5, P8, P10, P12, P13, P15	Dev3
D2.3.2	Test-Driven Development	B3, B4, B6, B7, B8, B9 B10, B11	P1, P3, P11, P13, P15	Dev1, Dev2
D2.4	Expressive tests, simple structure	B2, B4, B7, B8, B9, B10		SwArch1, Dev1, Dev3, Test2
D3	Design documentation	B1, B7		
D3.1	Executable (tests as doc)	B1, B2, B4, B7, B8	P4, P5, P9	Dev1, Test2
D3.2	Collaborative (wiki)		P2, P3, P4, P8, P9, P15	Dev1, Dev2, Test1, Test2

Books B3, B7, B8, B9, and B11 state that teams should be cross-functional and autonomously analyze, implement, and verify functional requirements. Book B8 states: "QA should find nothing," implying that QA is a separate team, focusing on verifying other requirements than pure functions, for example, usability, stability, security, and other quality requirements of the produced system. Paper P3 also mentions the introduction of *cross-functional teams*, as one part of transforming a large organization into Lean Software Development.

- **Empirical findings:** Testing of functions and requirements took place in the same team, and in the same sprint, as where the development of the production code took place. Because developers using the original functional test tool could not keep up with the development pace, a couple of developers wrote a new Java-based test case runner, where functional test cases was specified in a custom XML-based language. This allowed development of test cases to proceed alongside development of production code.

  Table 9 shows the linear evolution of the codebase over time for the major types of source code in the product. All studied types grow linear over time, with all p-values less than 10^{-13} and adjusted R^2 between 0.91 and 0.98. For the Java- and XML-based code, the column **Initial size** reflects the state at the start of data collection in January 2011, while the Scala-based code was first developed in Q3 2012. The column **Growth per quarter** is the calculated linear regression coefficient, and **End size** is the size at the end of the studied period, in December 2016.

  By calculating Pearson's correlation coefficient (r_{xy}) between different

Table 9: Summary Code Statistic for the Five Major Code Types

Code type	Lang.	Initial size [kLOC]	Growth per quarter [kLOC/qtr]	p-value	Adjusted R^2	End size [kLOC]
Production	Java	150	26.1	1×10^{-13}	0.91	753
Unit tests	Java	64	24.7	3×10^{-14}	0.92	620
Integration tests	XML	83	67.2	3×10^{-14}	0.92	1560
Web GUI prod.	Scala	9	3.3	1×10^{-13}	0.97	65
Web GUI tests	Scala	2	8.4	4×10^{-15}	0.98	129

types of code, we confirm that the volume of the different types of code varies together. Production code are related to unit tests by a correlation coefficient of $r_{prod,unit} = 0.998$ (p-value $< 2 \times 10^{-16}$), and to integration tests by $r_{prod,int} = 0.996$ (p-value $< 2 \times 10^{-16}$). The web GUI production code are related to the web GUI tests by $r_{webprod,webtest} = 0.975$ (p-value $< 6 \times 10^{-12}$).

All interviewees mention the highly iterative development process, and one developer contrasts this with regular consultancy work: "In a consultancy, they focus more on the delivery than on the craftsmanship…We used an iterative, test-driven way, to be prepared for what can happen in the future."(Dev1)

Several interviewees also mention tests being developed alongside the production code, e.g., "We used to ensure that whatever test cases had been written in the [test plan, a shared Excel document] will translate into some automated test cases."(Dev2)

"A strength was that we could test in parallel with development, based on a clear requirement base, in [the wiki-based requirement tracking system], where everyone could read it."(Test2)

- **Analysis:** Incremental development is part of getting reliable and actionable feedback and so is tightly tied to **F2** Short feedback loops. Because the teams owned "the whole development process," including functional testing, they took responsibility for the entire development phase, including documenting used solutions.

  The fact that all five types of code grow linearly, together, indicates that software was developed incrementally throughout the studied period. In a non-incremental scenario, we would have expected integration tests to lag behind the production code as the focus of the organization moved to test phases that followed growth of production code and unit tests. We see no such findings in our data.

  The organization took action when it discovered that the originally used functional test tool could not keep up with the development pace and created an alternative solution based on structured text files. However,

the amount of function test code soon eclipsed the production code, and it continued to grow faster throughout the study.

D2 Testing pyramid, layered testing

- **Literature:** Eight books and five papers stated that tests should be layered into different categories, see Table 8. The importance of having a stable base of test cases, independent of each other, is mentioned in three books, B8, B9, and B10.

  Papers P3, P11, and P17 mention how *solutions can be proposed by writing tests*, for instance, using Behavior-Driven Development, and the practice of high-level integration tests is also stated in books B1, B2, B3, B4, B7, and B8.

  Focusing on the development of tests, whether using *TDD* or a less stringent method, is mentioned in nine books and six papers, with papers P3, P13, and P15 explicitly mentioning *TDD* as a craftsman skill to practice.

  The practice of having automated tests of different kinds with a readable, simple structure is stated in five books, with the most pointed citation mentioned in book B8: "Unit tests and acceptance tests are documents first, and tests second. Their primary purpose is to formally document the design, structure, and behavior of the system."

  The "Agile Testing Quadrants" model [110] can be used to classify tests along the lines of "supporting the team" and "criticizing the product," versus "business-facing" (verifying customer requirements) and "technology-facing" (verifying individual implementation decisions). Outside the SLR findings, the books [43] and [7] also state that designing for testability increases the likelihood of tests being developed.

  Paper P5 explains how a *successful test run triggers a new executable package and deployment* to a DevOps pipeline, followed by further non-functional testing and further validation.

- **Empirical findings:** In the studied case, already from the start of the product, considerable focus was placed on verification on several layers, as illustrated by the test pyramid [111]. While some developers preferred TDD, others instead preferred to write tests after the production code, but tests were expected to be developed close to the production code, minimizing feedback time (item **F2**). As stated by the lead architect: "I call it *Test-Focused Development*, because one of the ground rules is that, if you build something, it should be easy to test. Always easy to test…If it is easy to unit test and function test, then it is better than building the small, slimmest solution. So, I always have this pyramid…You should

work with tests from Day 1. If you don't do that, you're doing it the wrong way"(SwArch1)

Another interviewee confirmed the test focus, by comparing with another product: "I think it [relates to] how we introduced ways of working in [studied case] We focused much on test coverage, and there was solid practice related to which test cases to write, how to review and present them. There was much more focus on testing, on automation and those areas."(Dev2)

The amount of (functional) integration test code soon eclipsed the production code, while the unit tests grew at the same pace as the production code. The same pattern repeated itself when the new Scala-based web-GUI was developed in 2012, as its functional test codebase, also written in Scala, grew faster than the GUI production code.

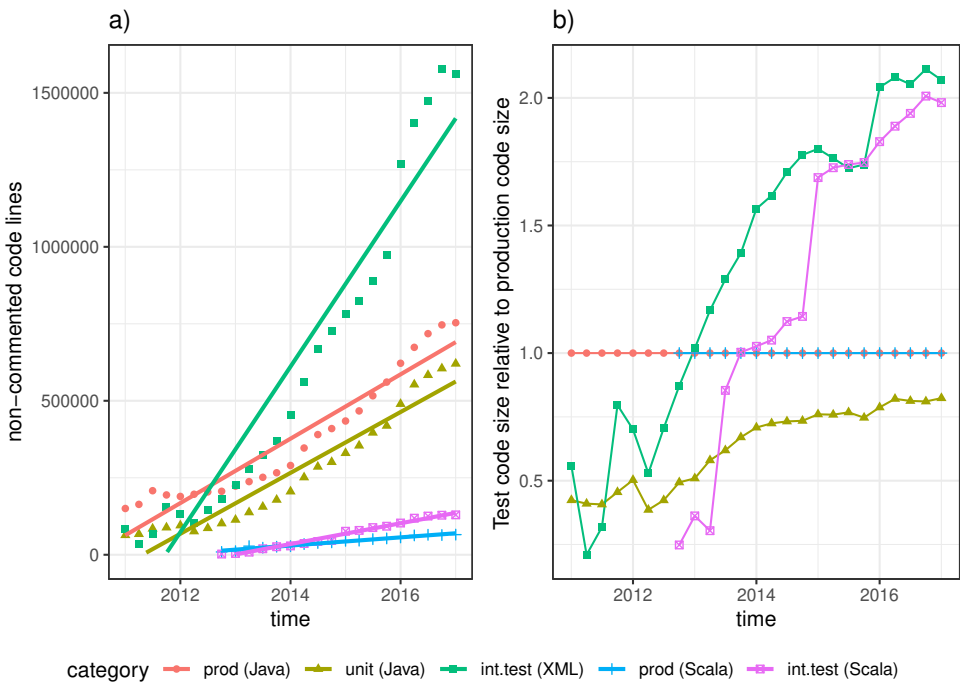


Figure 5: Ratio of test code vs. production code over time.

Figure 5a shows the numbers of non-commented source code lines for the production code (*prod (Java)*), unit tests (*unit (Java)*), integration tests (*int.test (XML)*), web GUI (*prod (Scala)*) and web GUI integration tests (*int.test (Scala)*), and Figure 5b shows the relative size of the unit tests and integration tests versus the Java production code, and the relative size of the Scala-based integration tests versus the Scala-based GUI production

code.

The figure shows that the integration tests were growing much more than the production code, while the unit tests kept approximately the same growth rate. As reported in Table 9, all five codebases grew linearly throughout the studied period. The three dips in integration test size during Q1 2011, Q1 2012, and Q4 2016 were due to product realignments, where old protocols and functions were removed. Both integration tests (written in XML) and GUI tests (written in Scala) grew to about twice the size of the corresponding production code, although the GUI code was much smaller. The unit test base was initially slightly less than half the size of the non-GUI production code but grew to about four-fifths ($\approx 80\%$) of the production codebase.

The unit tests can be further subdivided into "pure unit tests" (no interaction with the outside world) and "fixture tests," where the tests interact with a locally installed and prepared database. Non-functional testing used dedicated hardware, including dedicated simulators. The product placed a relatively large emphasis on unit tests that interacted with a locally installed database, using the *Transaction Rollback Teardown* pattern [112]. At the end of the studied period, 8,327 integration tests, 18,412 database-interacting unit tests, and 5,328 "pure" unit test methods had been developed. The number of pure unit test cases were higher, as these also included parameter-driven tests generated from the code via reflection, see item **F3** about the "meta-tests."

Each developer knew how to use and develop integration tests, though, in practice, one or two persons per team focused on writing them. "Anyone should be able to do the testing…One or two persons in the team, part of the team, developing [integration] test cases. He used to get assistance from other developers, in case required."(Dev2).

Another developer mentions, "…some testers might not have the correct background or understanding, so I gave them a template, like: 'This is how I think, now you explore more into your scenarios…'"(Dev1)

The lead architects decided to include "test helpers" in the functional verification phase, which facilitated efficient integration testing. "And then add some test packages on the side, which are used in the testing. So it's not black-box, but more gray-box. You use those packages to make your test flow a little better."(SwArch1)

The *Definition of Done* for feature development (see item **C2.3**), stated that functional verification should be automated before feature delivery. How to achieve this was regularly discussed in cross-team forums (see item **C3**). "Everything should be tested, and there should be automatic test cases for everything…"(Dev3) Despite this, some manual functional

test cases still existed. At the end of the studied period, there were 24 documented manual functional test cases, mostly related to data aging (importing/exporting archived database data) or security issues. These were executed based on a "risk-based judgement," typically when changes had been made in the tested area or before major releases of the product. The system testing team also focused on manual testing, such as validating instructions for administrators or integrators. This test phase was the first with a full hardware deployment, including Hardware Security Modules, application firewalls, and load balancing hardware. In contrast, functional testing in development teams utilized plain Linux virtual machines.

One developer mentions that the team structured their work so they would interact all the time and used this as a form of pair programming: "We did not divide tasks [in functional areas], such as GUI, persistence and so on. Instead, we pair-programmed a lot. We were encountering each other's code all the time, communicating verbally: 'Hey, this method you did—can I change it, make it better?'"(Dev3)

- **Analysis:** Specifying requirements as test cases will lead to the volume of test code eventually outgrowing the production code, as is visible in Table 9 and Figure 5. Therefore, it is important that these test cases (requirements documents) are easily readable, frequently maintained and executed to ensure that they still reflect the state of the product. Bjarnason et al. [113] describe five different variants of using test cases as requirements, based on a multi-case study made at three companies of various sizes. In particular, while the largest company had failed to completely specify end-to-end behavior, including user interactions, as test cases, they reported success in using the process when developing **application programming interfaces (API)**.

  Having this layered testing architecture as a regression test base enables safe refactoring and transformation into clean code (item **A3**). Thus, the test base enables clean production code, and the tests are required to be clean in order to be readable and maintainable. Overall, this enables an evolutionary growth of the software, without "big-bang" integration phases. However, cleaning and refactoring the tests themselves are harder to achieve. When changing test cases, care must be taken that the changed tests cover the original requirements. How to achieve this remains an unanswered question.

  There will always be some tests that are not possible or economically viable to automate. In the studied product, the developers identified 24 test cases out of a functional regression test base of 8327 test cases (0.29%) as belonging to this category.

  The different layers of tests are important to enable the feedback loops

necessary to guide incremental design and development. Each layer has different trade-offs related to reflecting the true production environment behaviour versus being fast and efficient to develop and trouble-shoot. In the product, many unit tests interacted with a locally installed database, which has the disadvantage of adding lead time to the feedback loop. However, there is also an advantage in that relatively large parts of the system can be tested down to the SQL level without mocking behavior.

D3 Design documentation

- **Literature:** Five books mention documentation in relation to craftsmanship, as *self-documenting programs*, in B1, *tests as documentation*, in B4 and B8, and B7 references to Knuth's work on *literate programming* [114]. Book B2 states that "a lesson from software engineering is that hardware and software never quite match their documentation." One solution to this proposed in both B7 and [43] is to extract documentation from the source code.

  Papers P2, P3, P4, and P15 mention *collaborative documentation* through Wikis or shared recordings. Paper P9 states that a *shared user story repository* gives immediate feedback on changes. Papers P4, P5, and P9 mention *code as communication*, exemplified by Domain-Driven Design, and acceptance tests in the form of *executable user stories*.
- **Empirical findings:** The studied product had no formal design documents (e.g., component descriptions) maintained by the development teams. Instead, they relied on a wiki system to document design principles and executable test cases as documentation of required behavior. The organization used deliberate practice (see **C5**) as a tool to teach development principles.

  As part of defining the external API, a tool was developed based on the *Javadoc*[11] tool, converting code comments and annotations, including validation rules, into a form suitable for customers or system integrators. This documentation evolved together with the API.

  The integration test cases also frequently served as documentation of how the product behaved, putting pressure on their quality and descriptions. The test case structure, including directory and file names, became part of the documentation, as it became harder to know where to look as the test base grew. As discussed in item **F5**, the automated test cases were continuously executed, and their results verified, meaning that the current tests reflected the actual state of the product.

[11] https://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/doclet/overview.html

Several interviewees mentioned that they were using tests as documentation: "The test was the documentation…even if we had followed [the requirement tracking tool]."(Test2)

One interviewee mentioned the lack of design documentation as a hindrance: "There are different levels of documentation. There are many complaints [from developers] that, for instance, data models are not documented, there is a lack of a leitmotif. On an overarching level, to get the big picture, there is quite good product documentation, though.."(Test1)

- **Analysis:** Executing design documentation towards a working system means that inconsistencies quickly surfaces, enabling quick corrections. However, as the test base grows, the internal and external structure becomes extremely important. Each test case needs to be self-sufficient, describing its needed environment and its setup. Business-facing tests should be specified in an appropriate high-level language, such as a Domain-Specific Language, to be accessible to people not directly involved in development.

  Collaboratively edited wiki pages documented the core design principles, with automatically executed test cases documenting the detailed product features. Documents targeted for customers or support personnel were kept at a high functional level. Detailed protocol documentation was generated directly from the source code, so it would automatically match the delivered product.

  We see some evidence that there was a perceived lack of certain aspects of documentation, though the overall product level seems acceptable. This could indicate that having a more structured approach to design documentation than a wiki system could have long-term benefits. At the same time, we see that developers were using the test base as documentation, meaning that as long as the tests are readable and at an appropriately high level, the system's code and behavior would be understandable.

Summary: When focusing on incrementally growing software, it is essential to focus on, and build, a comprehensive regression test base to validate that what was built still adheres to prior requirements. The regression tests will serve both as a safety net and as the actual specification of the behavior of the system under construction. As such, they should be readable by both programmers and requirement owners. To meet this goal and to ensure quick feedback, tests shall be structured in different layers. Higher-level tests shall use a language closer to the business domain than the ordinary programming language to support its usage as system documentation.

Note that not only the tests but also their organization and structure act as documentation. This is because the volume of tests will eventually eclipse the production code, and all developers should realize that it is as important to work with and care for the tests as with the production code.

Table 10: References to **C** *Shared professional culture*

Id	Name	Books	Literature	Qualitative
C1	Standard development environment	B1, B2, B6, B7, B9	P2, P4, P8, P9, P12	SwArch1, Test1
C1.1	Common code style	B4, B7, B9, B11	P2, P3, P4, P5	SwArch1, Dev1, Dev3
C2	Common professional culture	B2, B7, B8, B9	P3, P7	SwArch1, Dev1, Dev2, Dev3, Test1, Test2
C2.1	Caring	B3, B4, B8, B9	P1, P3, P7	Dev1, Dev2, Dev3, Test1, Test2
C2.2	Clear roles, responsibilities	B3, B8	P3	SwArch1, Dev1, Dev2, Dev3, Test2
C2.3	Definition of Done	B3, B7, B8, B9, B11	P3	Dev1, Dev3, Test2
C2.4	Pride	B5, B6, B7, B8	P4, P17	Dev3, Test2
C2.5	Collective ownership	B7, B8		SwArch1, Dev3
C3	Cross-team communication	B3, B7, B9	P1	Dev1, Dev2, Dev3, Test2
C3.1	Cross-team forums	B3, B9	P1, P2, P3, P4	Dev1, Dev2, Dev3, Test2
C4	Visibility / Transparency	B1, B3, B6, B7, B9 B11	P3, P9	
C4.1	Visible backlog	B3, B9, B11	P3	Dev2, Test1, Test2
C4.1.1	Technical debt visible, acted on	B9, B11	P5	SwArch1, Dev1, Dev2, Dev3, Test1
C4.1.2	Pull-based backlog	B3	P3, P5	
C4.2	Visible status	B3, B8, B9	P3, P9	Test1, Test2
C4.2.1	Information radiators	B3	P3, P9	
C4.3	Visible release plan	B9	P3	Test1, Test2
C5	Accountability	B2, B3, B7, B8, B9	P3, P8	Dev3, Test1
C5.1	Humility	B6, B8		Test1
C5.2	Reputation	B2, B6, B7, B9	P2, P8	
C6	Culture of learning	B1, B2, B3, B6, B7, B8, B9, B11	P3, P11, P12, P15	SwArch1, Dev1, Dev3, Test1, Test2
C6.1	Reflecting	B2, B3, B6, B9, B11	P1, P3, P5, P11, P15, P17	
C6.2	Kata exercises	B6, B8, B9	P3, P10, P12, P13 P15	SwArch1, Dev1, Dev2, Dev3, Test1, Test2
C6.3	Mentoring	B1, B2, B3, B5, B6 B7, B8, B9	P1, P3, P8, P15	

5.3 **C** Shared professional culture

The software craftsmanship manifesto states: “Not only individuals and interactions, but also a community of professionals,” as well as “Not only customer collaboration, but also productive partnerships,” which implies a long-term commitment to what is produced.

The focus on the community of professionals also implies a shared, common culture. As illustrated in Table 10, we have found evidence that a shared culture of *learning*, *caring*, *accountability* and *transparency* is beneficial and aligns with the craftsmanship approach.

C1 Standard development environment

- **Literature:** Books B1, B2, B6, B7, and B9 mention the benefits of standardizing on a toolchain. In particular, book B2 notes that the partnership approach highlights the importance of focusing on long-lived development tools.

  Four books (B4, B7, B9, and B11) explicitly mention how shared coding standards help communication and readability. Brad Fitzpatrick, in B7, mentions how Google keeps strict guidelines for programming styles, including code layout, formatting, naming, and which patterns and conventions to use[12].

  Several papers also promote common development standards as beneficial for software craftsmanship in terms of *structured exercises* to learn the correct shortcuts for the particular tool in use (P12), improve *source code quality* (P5), the usefulness of a *wiki page* containing both *coding style* guidelines as well as instructions for *how to set up the environment* (P2), capturing *IDE configuration* in a repository (P9), creating a sense of *commitment to a particular tool* (P4) and *obtaining necessary knowledge* how to best use or not use the latest *technologies, tools, processes, and practices* (P8).
- **Empirical findings:** At the start of product development, the lead architect chose a shared development style and code rules. The unified style helped both understanding the code and aided in merging and backporting fixes to older branches.

  Although standardized, the used toolchain varied over the years. Initially, developers used Eclipse on Windows laptops, later also IntelliJ and Linux laptops, and eventually, Windows was dropped as a development platform. Costs and competence were cited as the reason for changing both IDE and OS. When the vendor released a usable IntelliJ version free of charge, the perceived benefits (relative to the already free Eclipse) outweighed the cost of change. Similarly, when the company introduced Linux laptops as a supported development environment, the organization quickly adopted the new development platform, as it allowed developers to develop software in an environment close to the target environment, which always was Linux. When introducing the new IDE, it was configured to format code in the original Eclipse formatting style.

  The lead architect switched build tool from Apache Ant to the more expressive Gradle tool in mid-2012. The decision was driven by the new tool's stricter dependency management, stricter build scripts, increased performance and the ability to more easily develop plugins. The new tool

[12] https://github.com/google/styleguide

was used to automate more release tasks, and to build a **domain-specific language (DSL)** for deploying test machines in different configurations, resulting in more varied automated integration testing. As stated by the lead software architect: "Large-scale software development requires both structure and flexibility, but these must never cancel each other out. I think Gradle performs a better balancing act than, for example, Maven and Ant, which are at the opposite ends of that spectrum."(SwArch1)

The Eclipse formatting rules were added to a shared repository in November 2011, as part of the first expansion to a remote site. Until then, developers used the standard Eclipse configuration. In January 2016, a similar ruleset was created for IntelliJ.

- **Analysis:** The standardized code style is beneficial for sharing code between the different branches as it helps version control tools merge code automatically, without distracting white-space or formatting differences. Having a shared toolkit also helps people understand and be more efficient in helping each other.

  As evidenced by the empirical findings, standardized tools do not imply a static toolkit. Instead, a learning organization should always be on the lookout for new and better tools that do the task at hand more effectively and efficiently. However, the cost of changing tools will include teaching the organization ways of working with the new tool.

  Some tools are more challenging to switch than others. While the swap of the build tool Ant for Gradle involved few persons and was made abruptly, switching development IDE from Eclipse to IntelliJ took much longer and included trying to configure the new supported IDE so it could peacefully coexist with the older supported tool.

  Static code analysis and having the build process fail in case of violations helped unify the code style, as described in item **F3**.

C2 Common professional culture

- **Literature:** While Boehm in P14 expresses a view of "software crafting" as the "cowboy programmer," who "hastily patches faulty code by pulling an all-nighter," this is not the dominant view in the surveyed literature. Instead, four books (B2, B7, B8, and B9) expressly state the importance of teamwork and how important it is to create a common culture of collaboration. This view is also expressed in P3 and P7.

  Four books, B3, B4, B8, and B9, state the importance of caring for the test suite (the "code production line"). Hunt and Thomas [43] also mention the *broken window theory*, first formalized by Wilson and Kelling [115],

and how it relates to the importance of keeping the test base clean and working at all times.

Any organization larger than an individual would benefit from expressing the expected roles and responsibilities. Larman et al. in B3 recollect how one chief architect states that Scrum helped the team take responsibility for their assigned tasks. In B8, Martin expresses the view of having separate, but jointly collaborating, QA and development teams. Paper P3 reports how **Communities of Practice (CoP)**, together with open spaces, support discussing problems, solutions, and new ideas regarding a specific role, practice, or topic.

Five books (B3, B7, B8, B9, and B11) and paper P3 explicitly mention the concept of **Definition of Done (DoD)**, relating to a Scrum practice. Paper P3 refers to the DoD as partially standardized, while book B8 implies that the actual DoD would vary according to the business requirements, which analysts should write as acceptance test cases.

To take pride in one's work is mentioned by four books (B5, B6, B7, and B8) and two papers (P9, P17), and both Martin in B8 and Hunt and Thomas [43] states how this is related to *responsibility* and *accountability* (**C5**).

The principle of *collective code ownership* is a loaded term with multiple views present. Two experienced interviewees in book B7 lean towards individual code ownership as something that cannot be denied, while Martin, in book B8, states that it is better to break down all walls of code ownership and have the team own all code.

- **Empirical findings:** In the studied case, all lead developers had prior experience working with overseas teams. For this reason, they requested that teams onboarded from China (in 2011) and India (in 2013) were to visit the primary site for several months to learn the product and the professional culture, in particular, the product development process, including team tasks, planning, and verification. When the Indian teams went back to their site, a senior developer joined them for a year to support and guide their development efforts.

  The studied organization had a shared DoD with clear and actionable checks in several areas, such as *Requirements*, *Security*, *Design*, *Test*, and *Customer Documentation and User Experience*. Three different checkpoints were in place:

  – End of initial requirements gathering → start of product development
  – End of product development → start of system testing
  – End of system testing → feature released to the market

Each checkpoint had a template-based DoD checklist, signed off before the feature moved to the next phase. The requirement engineer (PO, see item **F1**) signed off the initial checklist. The Scrum Master in the development team signed off the middle checklist, and the team lead in the System Test team signed off the final checklist.

Developers from both the primary and secondary sites indicate that they felt a similar mindset in both sites. "…work culture in [main site] and India was almost similar…But in [other product] I see lots of difference between every corner of the world."(Dev1)

The developers also appreciated the practice they received and the concrete principles they learned. "…entering into a project with solid principles, these are the layers, with full hands-on experience, was the best."(Dev1) "You have a defined way of working, with respect to how you code the application."(Dev2)

Two interviewees mentioned the pride they took to make sure that what the team produced should also work. "We had some kind of pride in the team. We don't hack together something and just leave it. Rather, when we say that we are done, then we really *are* done.."(Dev3)

The regression test suite was provided with constant attention and care. To counter instabilities, in 2015, the organization set up a separate daily meeting with a participant from each team, discussing unstable or erroneous test cases and distributing them between teams. As described in item **C4**, the teams distributed and managed the identified tasks.

The test code was seen as important as the production code, as this was the documentation of how the system should behave. "The test code was equally important as the production code, because the tests showed what the product could do, like a fact-based answer."(Test2)

Two interviewees also mentioned how all developers cared to avoid security vulnerabilities in this product, relative to other experiences: "[In this product] there was a common way of working, focus on security, risk review, code reviews…These were very good controls. But when I moved to [other product], they did not care about anything…Dev2"(.)

- **Analysis:** The surveyed literature indicates that the "lone cowboy programmer" view of software crafting has little support by practitioners, which also is implied by the manifesto focus on "a community of professionals."

  The *Definition of Done* concept has been studied before [116] and is well-known in a Scrum context. According to the study, the focus of the DoD should be on the *systematic requirements* that are *common* for each user story. The studied organization followed this approach, using three different DoD checklists, corresponding to the three development phases

(elaboration, implementation, and system testing) before a feature was released.

It is undoubtedly the case that developing a large regression test base requires care and thoughtful design of how to prevent instabilities. For developers to trust that the tests reflect the true state of the application, the test base needs to be stable and predictable.

C3 Cross-team communication

- **Literature:** Two books (B3 and B9) and four papers (P1, P2, P3, and P9) mention the importance of communication across teams, for instance, using the concept of *Communities of Practice (CoP)* [117]. These communities are used to source and validate potential solutions, spread knowledge, and instill and reflect upon social and professional norms.

  Paper P2 explicitly states that the studied organization had tens of different CoP, which formed as needed and ceased to work when they were either dysfunctional or had fulfilled their purpose. The paper also states the importance for a CoP to have a *good topic*, *passionate leader*, *proper agenda*, *decision-making authority*, *open communication*, *suitable rhythm*, and *cross-site participation*, where applicable.

  Different communities, known as "Guilds" within Spotify, their challenges and benefits, have also been studied before, e.g. [118, 119].
- **Empirical findings:** In order to establish a common way of working, one developer stressed the cooperation that took place between teams: "It was not unusual to work across team boundaries when working with the test cases. When we discussed and found that the structure would not hold any longer, we discussed how to set the new structure. And then two or three participants would do the actual restructuring and report the progress on our [QA group] meetings."(Test2)

  Indeed, as the number of development teams grew in the product, a need for more efficient communication surfaced, both for architecture and testing activities, causing the organization to establish both an architect group (TA, see item **A1**) and a **Quality Assurance (QA)** group. Each group contained one member from each team, meeting regularly, the TA group twice, and the QA group once per week.

  Four developers mentioned the value of the recurring reviews as a means of competence sharing, for instance: "We used to present how we would implement a particular requirement [in the TA group] and get feedback. A very structured approach."(Dev1)

  "Having coverage — what do we think we need to do? So, implementations were reviewed in the TA forum, and test analysis in the QA forum.

Where the other teams could give their feedback. You explained what you intended to do, and they could comment: 'No, but you missed this area' — because they might have worked in that area recently, and we had never been there."(Test2)

One interviewee mentioned that time-boxing was used to limit the amount spent in meetings: "When we grew with more teams, we had to split up in review-groups, to review each others' [analyses] in detail. Building those groups based on competence to get good competence spread. [In the meetings] we made sure that everyone had read the analysis before the meeting, to be efficient, so we just could focus on the comments [that all members provided]. Sometimes we had mail conversations in these groups as well. But the analysis was documented [on the shared wiki]."(Test2)

In the studied product, each TA member had 20% of their time allocated for TA related improvement tasks, and a similar agreement existed for the QA group.

- **Analysis:** Our evidence supports the benefits of Communities of Practice (CoP), both in spreading knowledge (e.g., via review feedback) and professional norms (e.g. amount of tests needed).

  Participants from both the primary and the remote site participated in the weekly CoP meetings, ensuring that the communication flowed between the sites.

C4 Visibility / Transparency

- **Literature:** The principles of visibility and transparency are closely related to the **C5** Accountability and professionalism inherent in *productive partnerships*.

  Keeping the product backlog visible and up to date is mentioned in three books (B3, B9, and B11) and paper P3. The Lean principles of *keeping options open* and *limiting work in progress* by having a pull-based backlog are mentioned in papers P3 and P5 and book B3.

  The importance of visualizing and acting on technical debt is also mentioned in the books B9, B11, and in paper P5.

  Being open and clear about development status is discussed in three books (B3, B8, and B9) and in papers P3 and P9, where the goal of *maximum project status visibility* is stated. These two papers and book B3 also highlight how the use of *information radiators* helps in this regard.
- **Empirical findings:** As described in item **C3**, the studied organization formed cross-teams forums to counter the blame game often surfacing before meeting a deadline.

One identified problem was the large test base (shown in Figure 5), which required continuous maintenance effort. As described in items **C2** and **C3**, starting in 2015, teams coordinated to discuss, distribute and solve issues in this test base. The QA group was also driving improvements in this area, acting as a discussion board and mentoring others.

Information radiators in the team area, initially two lava lamps, later replaced with nine remote-controlled LED lamps, were used to broadcast the most important build status.

Stressing to make deadlines often cause people to take shortcuts. One often-used shortcut was to tag failing or unstable test cases as *Ignored*. The team mitigated this behavior by using Git logs to determine who had ignored a particular test case. After an initial grace period, automated periodic reminders were sent to this author to either fix or remove the test case. The QA forum discussed and took decisions on how to proceed with such tests.

"Sometimes you had to go in and ignore test cases…And later, you got an automated mail, stating, 'Please fix…' By then, you most likely had forgotten about the ignored test case, so you had like a 'reproach' there."(Test2)

Several interviewees mentioned the importance of visibility, of being honest about the status and potential obstacles, and being aware of the planned releases. "Having a dialogue, saying 'No, we are not done yet, because…' and highlighting potential delays as soon as possible. I think that was a strength also, to be able to de-scope, moving to a later feature. We never skipped [particular phases, e.g., testing], but rather whole areas or scopes.."(Test2)

One interviewee mentioned a particular strategy for dealing with project managers, who tend to prioritize delivery precision over delivery contents or quality: "A senior developer taught me to frame estimates like: 'If I am allowed to do this task, it will take me four weeks. But if we don't do it, the cost will be eight hours per week, per team, indefinitely.' If you start to present those estimates, then [the project manager] will act."(Test1)

Many interviewees also mention that refactorings **A3.4** were important to manage the technical debt: "The best part was that technical refactorings were taken as kind of a task, whereas in [other product] it is taken as a feature, and nobody will budget for it.."(Dev1)

"The legacy that exists that is extremely large…You always build a little debt. But you always need to *know* what your debt is. And work with it continuously.."(Dev3)

"Of course, we would like to refactor more. But I still think that we get a reasonable time for it.."(Test1)

- **Analysis:** As stated above, visibility and transparency are closely related to the principle of *productive partnerships*, where the long-term commitment is seen as more beneficial than the deadline-driven urge to "patch together something."

  The concept of *Technical Debt* [120] was created as a metaphor to illustrate when developers choose or are forced to take shortcuts, such as ignoring test cases. It is important to keep track of such debt, and the studied organization used automated tools to remind the author to take action (i.e., consider how to proceed with the ignored test case).

C5 Accountability

- **Literature:** Showing accountability for what you produce is mentioned as a craftsmanship trait in books B2, B8, and B9. In B7, Joshua Bloch states that "ultimately, you are responsible for your own work." Hunt and Thomas [43] also note that a professional software developer should expect to be held accountable and honestly admit mistakes or errors in judgment, which also plays into item **C4**. Paper P3 also mentions team accountability, whereas paper P8 stresses personal responsibility and sound work habits as characteristics of successful craftspeople.

  Books B6 and B8 stress the importance of humility to counter professional pride. In B6, the authors argue that apprentices should combine humility and ambition to progress in the right direction. In B8, the author stresses the importance for all professionals to show both pride and humility.

  Reputation as a basis for recruitment and professional career are elaborated in four books (B2, B6, B7, and B9) and papers P2 and P8. Paper P8 argues for adopting a value model where software leaders have key qualities, such as *a proven track record* and *a personal approach to solving problems that imparts a signature to their work*. Paper P2 refers to how participation in a Community of Practice enhances professional reputation.
- **Empirical findings:** As mentioned in item **F2**, the project relied on releases built strictly from version-controlled files, including the build system itself. Published code artifacts were signed by each developer using their private key, and the signature was validated towards an application-specific Certificate Authority (CA) at runtime. Components were published by individual developers, while the composite release was assembled and published by a dedicated Build Master role, rotating among senior developers, allowing developers to establish a reputation and enforcing traceability towards accountability.

One developer mentions that team accountability and pride were used to counter the pressure from other stakeholders to "just get it done." Another developer stresses the architects' accountability and responsibility to communicate a vision of the direction.

- **Analysis:** Accountability and responsibility are loaded terms but have long been standard practice in successful open-source projects, such as the Linux kernel, where no code is merged or released without proper sign-off by a responsible release master. These are also highly linked to item **C4** Visibility / Transparency, implying that participants should take responsibility for their creations, highlight issues and learn from mistakes, rather than place the blame elsewhere, which is typically the case in dysfunctional organizations [121].

C6 Culture of learning

- **Literature:** Eight books from the SLR findings state the benefits and necessity of a culture of learning and continuous improvement, which clearly is a major part of software development. Five of these, B2, B3, B6, B9, and B11, also state the importance of reflecting on improving efficiency and becoming a reflective practitioner.

  Papers P3 and P5 stress the notion of *learning from feedback*, such as first-hand evidence or team experiments. Paper P11 calls for *ongoing move-testing-experiments*, where bugs are seen as talk-backs from the material that drives the development process forward. Paper P2 focuses on *knowledge sharing and learning* as a part of Communities of Practice. Paper P15 fosters *self-directed learning skills*. Papers P1, P3, P5, P11, P15, and P17 all mention the importance of *reflecting and improving processes*.

  Three books (B6, B8, and B9) and five papers (P3, P10, P12, P13, and P15) describe using reflective practice via *kata exercises*, sometimes practiced in a *coding dojo*. Paper P12 relates the kata concept to "experience levels," and paper P10 draws conclusions from data gathered during a *global day of kata exercises*.

  Eight books describe mentoring, with B5 vividly describing how the medieval master craftsman Antonio Stradivari failed to pass on his violin-making secrets to his sons, either because he could not mentor them or because he was not aware of them. Papers P1, P3, P8, and P15 mention the importance of *coaching and mentoring* as craftsmanship principles.
- **Empirical findings:** Learning culture was embodied in the project via a set of exercises called code katas, which explained and showed how to use the product development framework to develop functionality with

the tests in focus using TDD. The katas were first developed in 2013, preparing for expansion to the India site, and were updated as the product framework evolved. Eventually, ten katas were developed, building a simple Java application from scratch to a fully-fledged GUI, using Scala and the GUI framework used in the product. The katas built on each other and, depending on the team's experience, took between one and two hours each to complete.

The first couple of teams performed the exercises in a group setting. While this was time-consuming, it also helped the team members to learn about each others' strengths and weaknesses and support each other. Throughout the studied period, newly onboarded developers used the katas to learn how to develop in the product framework. Unlike the initial sessions, these exercises were done individually or in pairs, shifting the learning experience more onto the individual.

During the initial years, sprint demos for the entire development organization were used to spread knowledge and show newly developed features. As the number of people grew, this became too cumbersome, and the cross-team forums were used instead to spread knowledge. "I think those mini-demos we had [in the beginning], for the whole organization, was a way to spread knowledge…Really important also that even though we worked in teams, the decisions we made were shared among the teams [in cross-team forums]."(Test2)

All interviewees mention the katas and agree that they were a vital teaching device.

"It was a straightforward, focused approach. During the kata sessions, I realized that [in my team], we have different people with different backgrounds…I could see what mistake that they were doing and I could coach them…."(Dev1)

"One way of practicing is doing structured practice…Just to learn the IDE shortcuts."(Dev1)

"…always try to stay ahead of everyone else…It's better to fail, and learn something, than not try at all."(SwArch1)

Two interviewees mentioned retrospectives as a way to reflect on their progress: "We used to do retrospectives after each sprint, where we realized: 'OK, we had this problem in this delivery — how can we avoid it the next time?', and we used to collect this in an Excel file to aid the next task."(Dev2)

- **Analysis:** As Brooks stated in B1, software developers are expected to learn new techniques and tools to improve their skills and productivity. He also mentions the importance of mentoring to achieve this goal, taking as an example the legendary IBM CEO Thomas J. Watson, who was

shown how to sell cash registers by an older, more experienced sales manager.

However, the concept of *code katas* takes the *showing* approach one step closer to software development. Several books and papers mention the concept, and the studied project was also highly influenced by katas as a teaching device. As an introductory vehicle to the application framework, they were successful, as stated by all interviewees. However, few used them as *deliberate practice*, which was one of the original goals of the katas.

There is evidence that the teams performing the katas in a group session increased collective learning by making the group discuss individual problems and solutions.

Summary: When teams are developing and testing features in parallel, the importance of having a shared professional culture increases. To keep a coherent architecture, onboarded teams and individuals received structured training, and everyone was expected to contribute to the culture of learning. The shared culture was encouraged by several cross-team forums, and three checklists were used as DoD checkpoints, corresponding to the development phases.

All interviewees stated that the code kata exercises were effective in increasing the understanding of the application framework and the expected professional behavior, including testing strategies. However, there is no evidence that participants used the katas to improve their skills beyond the initial try, indicating that the goal of *deliberate practice* was not met.

5.4 **F** Feedback

Feedback loops have always been important in the software industry, as described both by Royce in 1970 [122] and by Brooks (B1) in 1975 [14]. However, the last 50 years have seen an immense change in *speed* and *automation* of both feedback loops and the software delivery pipeline.

Feedback is one of the five values of the Agile method XP [94], and it is intimately tied to the sprint practice of Scrum [123], which also includes explicit review practices.

Lean Software Development [54] also focuses on feedback. In particular, the practices of *Deliver as fast as possible* and *Build integrity in* highlight the importance of caring for the feedback loops and striving to optimize them, both from a latency and robustness point of view.

Much of the craftsmanship principles detailed in Table 11 are similar to, or complements, Agile or Lean principles, which is acknowledged in several books, for example, as stated by Mancuso in B9 [8]: “Agile methodologies help companies to

Table 11: References to ***F*** *Feedback*

Id	Name	Books	Literature	Qualitative
F1	On-site customer (proxies)	B2, B7, B8	P3, P7	Dev1, Dev3, Test1, Test2
F1.1	Requirements	B7, B9	P9	SwArch1, Test2
F1.1.1	Accessible	B2	P3, P9	Dev1, Test2
F1.1.2	Collaborative	B1, B2, B3, B7, B8 B9, B11	P3, P5	Dev1, Test1, Test2
F1.2	Frequent demos	B2, B3, B8, B9, B11		Test1, Test2
F2	Short feedback loops	B2, B3, B4, B6, B7 B8, B9	P1, P3, P4, P5	SwArch1, Dev1, Dev2, Dev3, Test1, Test2
F3	Review	B1, B2, B6, B7, B8	P3	SwArch1, Dev1, Dev2, Test2
F3.1	Team review	B3, B6, B7, B8, B9	P5	SwArch1, Dev2, Dev3, Test1
F3.2	Static review tools	B4, B7	P5	SwArch1
F3.3	Solution review	B7, B9		Dev1, Dev2, Dev3, Test2
F4	Learning from feedback	B2, B3, B6, B7, B8 B9		SwArch1, Dev1, Dev2, Test1, Test2
F5	Continuous integration and tests	B1, B2, B3, B4, B7 B8, B9, B11	P3, P5, P11	SwArch1, Dev2, Test2
F5.1	Frequent release candidates	B1, B2, B3, B9, B11	P5	SwArch1
F5.2	Reproducible releases	B1, B2, B3, B4, B8, B9	P7	

do *the right thing*…Software Craftsmanship helps developers and companies to do the *thing right*."

F1 On-site customer (proxies)

- **Literature:** Books B2, B7, and B8 all mention the importance of close collaboration between the requirement owner and the development team, something that also is a crucial trait of Agile (e.g. [94, 123]) and Lean [54] processes.

  Papers P3 and P5 use the term *Product Owner*, and report that *close collaboration and communication* between the development team and the requirement engineer reduce the waiting time for clarification or re-prioritization of requirements. Paper P7 is cited as the inspiration for the Scrum process [123] and stresses the technical contributions of the Project Manager and Product Manager roles in the studied product.
- **Empirical findings:** In the studied case, the requirements were version-controlled and located in a single wiki-based tool since early 2012. Prior to that, requirement engineers were using a proprietary tool, much less accessible. "[referring to the old req. tool]—Oh, that was a tool…It took me ages to learn how to upload an Excel file there. We were supposed to tag requirements to test cases. It was terribly unwieldy…But then we got [the new tool]…We could structure it to fit our needs, with requirements as user stories with a version, a history, in one place, reachable for everyone, regardless of whether you are a tester, developer or system tester."(Test2)

  As part of the development phase, teams demoed potential solutions for the proxy customers, who provided feedback and direction.

Table 12: Elapsed Calendar Days Per Feature Size and Activity.

		Development			No QA	QA Performed			
Est.size	N	$\hat{x}$	$\bar{x}$	σ	N	N	$\hat{x}$	$\bar{x}$	σ
X-Small	122	22	28.3	24.8	37	85	7	13.2	16.5
Small	109	29	35.2	30.9	24	85	8	18.9	26.2
Medium	72	47.5	61.3	47.3	10	62	16.5	26.5	31.3
Large	13	62	60.4	49.7	1	12	20.5	21.6	10.7

No QA is the number of features where planned system verification was deemed unnecessary.

"I would say that we talk to the [requirement engineer/proxy customer] at least for half an hour every other day, during the development of a feature. More in the beginning and in the end, and maybe with a more quieter period in the middle. But I would say we talk to them a lot in the middle too…About things that pop up, in code, that maybe are not like the requirement was stated."(Test1)

"…I was just asking the requirements engineer: 'Is it really this, or you wanted something else?'"(Dev1)

- **Analysis:** As stated in both the SLR and case study results, software craftsmanship values cooperation rather than confrontation and constant contract negotiation between developers and requirement owners.

  However, constant cooperation also means that requirements need to be in a *single*, *accessible* and *version-controlled* space, which tracks the evolution of the shared knowledge. This is crucial in order to *know the current status*.

F2 Short feedback loops

- **Literature:** Seven of the studied books (B2, B3, B4, B6, B7, B8, and B9) emphasize the importance of getting quick and relevant feedback on all development tasks. Book B6 explicitly states that practice without periodic feedback risks developing bad habits and voices the importance of giving less experienced developers feedback. As stated in item **D1**, book B2 mentions fast feedback as crucial to incremental development, as it allows adjusting direction before it has progressed for too long. Papers P3, P5, and P9 highlight the importance of *fast feedback loops*, also for distributed teams.
- **Empirical findings:** In the studied case, product development emphasized getting fast, relevant feedback from customers or internal proxies. There was an urge to slice large requirements into several pieces, each building on the previous, but deliverable and testable on its own.

  Table 12 shows data from 316 features, whose size was estimated into one of four categories by an estimation group before development started.

The table contains the number of features of each size (N), and the median ($\hat{x}$), mean ($\bar{x}$) and standard deviation (σ) of the number of calendar days spent in the development (including design analysis) and system verification (QA) phases. The collected data refers to the period between June 2012 and December 2016. We tested each group with linear regression and found no statistically significant change (either positive or negative) between either the development or the verification duration over the studied period.

The table shows that the organization developed more X-Small (122) and Small (109) features than Medium (72) or Large (13) ones. This suggests that rather than spending months developing several large "chunks of related functions," the organization valued getting feedback, both from system testing organizations and real installations. All four groups have median values lower than mean values, indicating right-skewed distributions.

Features deemed unlikely to impact quality attributes such as performance, stability, or usability were not individually validated in system verification. As indicated in the **No QA** column, this affected 30% of the X-Small and 22% of the Small features. Statistics for features in system verification are shown in the **QA Performed** columns.

Half of the X-Small features spent less than 22 days in development, including design analysis. This is interesting as the organization used three-week sprints, indicating that these features took around one sprint to complete. Examining the commit statistics for these features reveals that the median number of days spent in development (i.e., not considering design and analysis) was 12.5, with a larger mean of 20.8 and a standard deviation of 26.7 days. The system testing organization was also using three-week sprints, which could explain why the larger features were using close to 21 calendar days on average.

As described in item **F5**, teams constantly worked to keep feedback loops from the Continuous Integration builds as short as possible. This involved both utilizing hardware by executing tests in parallel and redesigning test cases (e.g., avoiding sleep statements).

- **Analysis:** Table 12 indicates that the majority of features were estimated to be X-Small or Small and that this is also reflected in the development and system verification time. However, as indicated in the table, some features are, due to their nature, impossible to slice into smaller parts. This affected 27% of all features, most of them medium-sized. Planned system verification was omitted in 72 of the analyzed features, meaning that more than one in five (22%) features were deemed only to contain functional aspects, which was validated only by the development team

before being deployed in production.

F3 Reviews

- **Literature:** Reviews have long been used as a tool to judge solutions and provide knowledge sharing, and books B2 and B6 state that the review process goes both ways, where junior developers also review everything produced by the team for the purpose of learning. Book B8 recommends pair programming as an efficient and effective form of instant code review, and papers P3 and P5 confirm the importance of frequent reviews as the core of Software Craftsmanship principles.

  Two books (B4 and B7) and paper P5 mention the importance of tools that automatically perform some review, including enforcing formatting rules.

  Regarding reviews of solution proposals, there are contrary opinions in B7. One interviewee (Brendan Eich) states that this implies a waterfall process, which should be avoided. Still, two other interviewees state that an adequately prepared design review can strengthen the solution. However, they make a distinction between an internal design review, whose purpose is to criticize or find omissions in the implementation, and an external review, involving clients, clarifying that the proposed solutions solve the intended problem.

- **Empirical findings:** In 2012, following the expansion to the first remote site, the studied organization started using a wiki platform supporting page templates to introduce an **Implementation Proposal (IP)**. For each feature to implement, each team was expected to produce an IP to be reviewed by the TA and QA groups (see item **C3**) . While the TA group reviewed the technical solutions, the QA group focused on reviewing test strategies such as test coverage and test structure.

  During the studied period, 586 IPs were produced, of which 460 were using the wiki-based format (starting from January 2012). Surprisingly, we also found 24 requirements without a corresponding IP. In 4 of these cases, the actual requirement was canceled without being completed. In the remaining 20, there was other reasons for omitting the proposal, such as the solution being described elsewhere or the lead architect doing the implementation himself.

  In 34 out of the 460 wiki-based IPs, the first code commit predated the creation of the IP page, and in 15 cases, it happened on the same day. This indicates that teams were prototyping (on a personal or team-based branch) as part of writing the proposed solution. The IP page contained

various sections that were actively updated during both the development and the system testing phases.

Related to code reviews, human reviewers should focus on content rather than style. To meet this goal, as described in item **C1**, mandatory code formatting rules and static checks using the PMD and FindBugs tools were introduced, causing the build to fail in case of violations. An earlier attempt in using advisory Sonar rules (post-commit, sending feedback through email) proved unsuccessful, as most developers ignored these warnings.

The product started using advisory PMD checks in August 2012 and made them mandatory in December 2012. The number of checked rules was initially small but grew over time. At the end of the study, it comprised 373 FindBugs, 155 built-in, and 7 application-specific PMD rules, developed by a team architect to flag particular code patterns as unwanted in the application code.

Starting in April 2012, a number of invariant-checking unit tests, called "metatests," were developed to give fast developer feedback on the expected behavior of the produced code. The meta-tests scanned the project classpath, performing static checks on classes that match particular application-specific criteria. Examples of such tests are "Request and Response classes shall have validation annotations on all fields" and "All remote-invoked methods must have an audit log annotation."

The first Gerrit review took place in June 2013. During the studied period, 3,802 reviews took place, out of 54,637 total commits. One interviewee indicated that the team used pair programming rather than Gerrit-based reviews: "Our team made a decision not to use Gerrit for review. Instead, we were pairing up, reviewing by sitting close, working on the same task, and interacting with each other's code."(Dev3)

- **Analysis:** Reviews can be used both to spread knowledge and to enforce an architectural direction. However, to be effective, they require motivated, knowledgeable, and accessible reviewers.

  As evidenced in the findings, the solution review step did not preclude coding. In over 10% of the found cases, the first line of feature code (presumably a prototypical solution) predated even creating an empty IP page. Instead, the solution review should focus on whether the proposed solution aligns with the overall architecture and direction of the product and sharing the concepts and the approved design between different teams.

  However, feedback frequency is also important—it is wasteful to spend effort in a direction not aligned with the overall product architecture. Thus, architects should discuss the intended solution before starting to write a formal IP.

Static review tools have the advantage that they are objective, consistent, and persistent, but they are limited in scope and have the disadvantage of flagging false positives. The tool can function as a teaching device by tailoring the tool error message or adding application-specific rules. This studied case used the PMD tool to meet this end.

F4 Learning from feedback

- **Literature:** Six books (B2, B3, B6, B7, B8, and B9) report on the importance of learning from received feedback, with book B6 stating that useful feedback needs to be possible to act upon.

  Papers P3, P5, and P11 state the importance of *learning through fast feedback loops* and *ongoing move-testing-experiments*. As discussed in item **C5**, this is also intimately coupled with a culture of learning.
- **Empirical findings:** Five interviewees mention software development as a learning exercise and highlight reviews as a tool to share knowledge and get feedback, not block development. One interviewee reflects on the importance of learning from customer feedback: "[reacting to defect reports by]…taking a step back, and analyze: 'This was an area that the customers were into…Are there more black spots like that?'"(Test1)

  To a large extent, the practices in item **C5**, being focused on learning, also apply here. The TA and QA roles (see **C3**, **A1**) were also expected to guide their team members via regular feedback and share experiences across teams.
- **Analysis:** By focusing on the learning experience of software development and striving to use feedback (whether automated or manual) to learn new and better development practices, it can be argued that the organization as a whole prioritizes learning in a structured way. This is also exemplified by the Lean principle of *Amplify learning* [54].

F5 Continuous integration and tests

- **Literature:** As stated by Brooks in his commentary to the 20th anniversary of the original publication of B1, technological progress has led to that "[Microsoft] rebuild the developing system every night [and run the test cases]" [14]. These days, when 25 more years have passed, the nightly runs have been replaced with on-demand-builds, which run after each check-in. The importance of this evolution is stated in eight of the studied books, and papers P3, P5, P9, and P11 also discuss the benefits of *continuous integration and regression testing* for software craftsmanship.

- **Empirical findings:** Automated build tools, first Hudson, then Jenkins, were used since the inception, including mandatory testing phases following the compilation and building of the software. The organization relied on personal responsibility, with code signing using personal certificates (see item **C5**), although the release building process was highly automated using build tool plugins, enforcing rules about tagging and versioning of artifacts and dependencies.

  As seen in Figure 5 (see item **D2**), the amount of test code soon eclipsed the amount of production code, as the number of test cases kept growing along with the product functionality. Initially, the test suite was executed sequentially, in a monolithic fashion. Later this was broken down into many parallel tasks, each running towards an isolated **System Under Test (SUT)**, to decrease feedback latency. The management (booking, releasing, reinstalling) of these systems was handled by an own-developed test-host installation and reservation system, utilizing the SUT to the highest possible degree. At the end of the study, each commit was triggering up to 181 parallel integration test tasks.

  In some circumstances, concurrency issues (e.g., threading) caused tests to fail sporadically (flaky tests). One such example was related to alarm sending and logging. The first naïve solution by individual developers was to add sleep statements into the flaky test case, delaying the test execution by a fixed amount of time. In addition to being wasteful of resources (as the test host was not performing any useful tests, delaying feedback), this also caused additional instability, as the required delay would be dependent on the CPU and network load on the physical machine running the virtual machine under test. After discussing in the TA group (see item **C3**), a senior developer made a special "test helper" using barrier synchronization to solve the instability. Further test helpers solved most causes of instability. The remainder (e.g., due to dependencies on manipulating features in complex third-party software) were relegated to nightly runs when the test environment was less used and more stable.

  Between December 2010 and December 2016, the team made 721 candidate releases of the main product. Of these, 248 turned into sharp releases (where 36 were major feature releases, and the rest was smaller defect corrections). On average, this amounts to 10.0 candidates and 3.4 sharp releases per month. Between March and December 2016, the continuous integration environment made, on average, 1428.6 builds per month on the master branch (not including feature branch builds).

- **Analysis:** Many authors can testify to the utility of Continuous Integration. However, running the tests is not enough; the organization must

also act on the feedback provided by the test, including fixing errors, unstable tests, and focusing on keeping the feedback cycle time reasonable. The studied organization strove to shorten the feedback loops for the integration tests to give relevant feedback as soon as possible. Test case structure was also regularly discussed in the QA forum (item **C3** and **C4**).

Making frequent release candidates and releases means that manual intervention in the release process needs to be kept to a minimum. Still, the organization valued the accountability given by personal code signing of individual artifacts, release candidates, and sharp releases. One benefit of frequent releases is that there is no "big-bang effect" when making the sharp release. By that time, recurrent Continuous Integration jobs should already have verified the constituent components and the functional difference since the last release should be small and manageable.

Summary: As stated in the introduction, feedback loops have been at the core of software development for at least 50 years. However, the tools and frequency of the feedback have changed over the years. The studied organization not only *used* Continuous Integration practices, but also *worked with* them, striving to *optimize*, and get *faster* feedback.

Similarly, realizing the cost and scarcity of human feedback, the organization strove to utilize *review tools*, such as static code review, invariant-checking unit tests, and web-based review tools such as Gerrit. There was a mandatory design review step to spread knowledge and align directions, but this did not prevent teams from prototyping before describing their first proposed solution.

We also see evidence that in some cases, the agreed process (e.g., reviews, solution descriptions) was not followed. This indicates that the organization tolerated deviations from the process, as long as the perceived benefits of the deviation outweighed the perceived costs (e.g., the lack of competence spread or the risk of lower quality).

6 Discussion and Implications

6.1 The principles and practices of software craftsmanship --- in literature and in our case study

Tables 6, 8, 10, and 11 illustrate the overlaps between the literature and the presented anatomy of craftsmanship. Among the most notable discrepancies and expansions, we consider the following.

A key architectural principle in our anatomy is the **A1** *Participating Software Architects*, i.e., architects need to participate in day-to-day software development. This extends the principles from the literature of passionate, skilled technical leaders who lead empowered teams both practically and concretely. We highlight the decision of

A3.2 *Judicious use of third-party products* as a key practice to follow when setting architectural direction. In addition to functional requirements, quality requirements such as testability and upgradeability must be considered when choosing software components. We note that the architectural direction should be exemplified via concrete, testable **A3.3** *Common application patterns*, rather than comprehensive documentation.

Our results also emphasize that tests should be structured in **D2** *layers*, and every test case should be **D2.1** *stable and independent* to reduce dependencies and enable faster fault isolation and correction. Tests were kept in focus through the principle of **D2.3** *Test-focused Development*, with tests developed close to the production code, using **D2.3.1** *Pairing* and **D2.3.2** *Test-Driven Development*. We also highlight that the relative lack of comprehensive design documentation was alleviated by having a test base of **D2.4** *expressive tests, with a simple structure*, which also served as **D3** *Design documentation*, together with a collaboratively edited wiki system.

An Agile setting expects teams to be self-organizing, without structure imposed by external forces. However, this freedom should be supported by **C2.2** *Clear roles and responsibilities* and shared **C2.3** *Definition of Done* (DoD) criteria, which help all participants in the organization know what to expect, and when to expect it. This is not to say that external forces have to appoint these roles and check on the DoD, only that the team needs to organize so that the roles are set, and the DoD criteria are fulfilled. To gain trust between different stakeholders and to allow corrective actions, **C4** *Visibility* is essential, including backlog, issues, technical debt, and **C4.2** *Visible status*. Another key practice is **C5** *Accountability*, affecting both transparency and **C5.2** *Reputation*.

Like the agile principles, our vision of craftsmanship also focuses on feedback loops, such as **F1.2** *Frequent demos*. The practice of **F3.3** *Solution review* is highlighted to spread knowledge between teams and to ensure that the proposed solution aligns with the architectural direction. It is important to note that, when needed, the proposed solution should be vetted using prototypes and real test cases before the review takes place. The continuous learning organization values **F4** *Learning from feedback* and sees this as positive. Defect reports can be seen as both good and bad. While reoccurring defects are clearly bad practice, the first occurrence of a particular issue is judged from case to case. Metrics are used accordingly.

6.2 What are the consequences of applying the software craftsmanship principles and practices in real life?

Based on the studied case, we found several examples of how software craftsmanship is embodied in practice and the consequences it brings:

- Developing in a **D2.3** *test-focused* way does allow production code to be refactored and shaped into a clear representation. However, as the product accu-

mulates features, the test codebase will grow faster than the production code, more so for the integration test code than for the unit test code. Therefore, it is important to **D2** *test at several layers* and constantly work with the test code, which is as essential to keep **C2.1** *clean as the production code*. Regarding **A3.4** *refactorings*, the studied organization made on average 16.8% refactoring commits during six years, excluding refactorings made as part of regular features.

- The **D2** *test code* serves two purposes — first, it should verify that the system still behaves as it used to do, and second, it should be **D3.1** *readable as a description* of what the system does. In order to meet these goals, the tests need to be **F5** *frequently executed*, and failures or broken builds need to be quickly **F4** *acted upon*. In some cases, organizational support is needed to enforce these norms, and **C3.1** *Cross-team forums* can be used to solve this efficiently.

- There is a trade-off to be made related to verification efficiency and correctly mimicking a deployed system. Solutions to **D2.1** *unstable test cases* can include re-architecting or adding helper functions to make them more stable, increasing testability and trust in the test suite, at the cost of allowing deviations from a production system. As these added functions will not be part of the end-to-end delivery, it is important to keep them **A2.1** *architecturally isolated* from the object under test. Later test phases, such as system testing, should then test the product from a black-box perspective.

- **A1** *Software architects* and **A1.2** *senior developers* play important roles in architectural direction and forming a **C2** *common professional culture*. In the studied case, the creation of a **C** *shared professional culture* was facilitated both by relocating the remote teams to the primary site for a few months, to learn the product and the development process and by the **C6.2** *structured exercises (katas)* used in order to **C6.3** *teach* newcomers the preferred way of developing new features.

- **F2** *Frequent feedback* is important, both from tools, artifacts, and other stakeholders, such as **F1** *requirement owners*, **F3.1** *other peers*, verification engineers, or target installations.

- All interviewees mention the structured, down-to-earth, practical **C6.2** *kata exercises* as important tools to learn the development process and the preferred way of developing the product, particularly in a group setting. However, there are few indications in the studied case that the katas were used as deliberate practice.

- While the organization advocated and the kata exercises taught **D2.3.2** *Test-Driven Development*, the organization also realized that TDD could be a hard technique to master. Nevertheless, tests and verification were kept in **D2.3** *focus* by keeping the development team responsible for automating functional test cases and keeping the manual test cases to a bare minimum.

- Having a **C1** *common toolchain* and striving for **C6** *mastery* of this toolchain is yet another aspect of a common professional culture. Still, this does not mean that the tools should be static. In the studied case, the organization changed tools several times to be more productive. In some cases, the switch was "abrupt" (e.g., version control and build tools), and in some cases, the switch was "gradual" (e.g., supported IDE). The organization should be prepared to **C6.3** *teach members* the new tools, using guidelines, seminars, and **D2.3.1** *pairing*.

We also found instances where the studied organization fell short of the espoused principles—for instance, regarding **C6.2** *kata exercises* being used solely for new developers, in an individual and isolated setting; a few features being developed without the requested **F3.3** *solution review*; and there were certain teams where **D2.3.2** *pairing* and **C6.3** *mentoring* worked better than in others. In this regard, the software craftsmanship principles and practices can be seen more like guiding lights than absolute truths. However, we still think it is worthwhile to study them more.

6.3 Software Craftsmanship vs. Agile Software Development

Following the organization in paper P5 [67], here we compare, in light of the findings from this study, the principles from the Software Craftsmanship Manifesto with the principles in the Agile Manifesto.

6.3.1 Well-crafted software vs. Working software

Software craftsmanship focuses on well-crafted software, while agile software development promotes delivering software as quickly as possible. Therefore, craftsmanship goes beyond project activities reported as the most frequently used agile practices, e.g., *standup meeting*, *backlog*, *sprint/iterations*, and *sprint planning* [46]. According to the State of Agile Report [124], companies applying agile practices rarely report on practices such as **F5** *Continuous integration*, **D2.3.1** *Pairing*, **D2.2** *Automated testing*, **D2.3.2** *Test-Driven Development*, and **A3.4** *Refactoring*. The results of the SLR, together with the findings of our case study, suggest that craftsmanship focuses on offering agile organizations more down-to-earth, technical practices to improve long term stability and quality, e.g., **A2.1** *Isolated and Layered Architecture* or the use of **A3.1** *Minimalistic Frameworks*.

6.3.2 Steadily adding value vs. Responding to change

Rather than only quickly reacting to changes, craftspeople are expected to also come up with their own improvements, such as **A3.4** *refactorings* or improvements in the overall production (e.g., tools, such as optimizing the **C5** *continuous integration* environment or **D2.2** *automated testing*). This is to make sure that **F5.1** *frequent releases* and **F2** *short feedback loops* prevent degradation of the **A** *architecture*, which would limit the ability to continuously and steadily add value.

A review by Kupiainen et al. [59] indicates that the metric with the strongest influence in Agile and Lean contexts was *velocity*, followed by *effort estimate* and *customer satisfaction*. However, we argue that not only velocity but also clean and bug-free code matters. The same authors report that metric information was broadcast in hallways to motivate people to react faster to problems. Thus, our **C4.2.1** *information radiator* practice was also used to influence behavior here.

6.3.3 Community of professionals vs. Individuals and interactions

Emphasizing the community of professionals over individuals implies that craftspeople would be expected to help each other grow through **C6.3** *mentoring*, constructive feedback, and experience sharing [8].

Our literature and case study results confirm the importance of a **C2** *shared professional culture* and **F** *feedback* as essential themes. Quick **F2** *feedback loops* enable organizations to **D1** *develop incrementally*, concentrating on small deliverables with predictable lead-time. This is crucial for keeping a sustainable pace adding value, and, if needed, to "fail fast." The shared professional culture might impact the ability of the organizations to build up a cross-site sense of belonging and foster the creation of shared ways of working in distributed environments.

The growth of open-source communities and the sponsoring and development of open-source software by commercial vendors can also be viewed as emphasizing software development communities.

6.3.4 Productive partnerships vs. Customer collaboration

While Agile focuses on interactions and collaboration with customers, the craftsmanship approach takes a more long-term, strategic view. For craftspeople, the produced artifacts, knowledge, and learning become part of the organizational knowledge and strengthens the ability to respond and assimilate changes. By being **C5** *accountable* and practicing **C4** *visibility and transparency*, craftsmanship brings a balancing force to customer-focused agile practices.

In the studied case, customer collaboration was implemented through customer proxies and in the "Internal live customer" phase, starting after less than a year of development. This proved successful in sharpening the development teams and spreading knowledge about the product and its environment to integration engineers, which helped smoothen the transition to external customer deployments. After deployment

to external customers, the requirement inflow increased, but the organization had already achieved a smooth development process and could keep up with demands without compromising quality.

6.4 Software Craftsmanship vs. Lean Software Development

I

In this subsection, we compare our anatomy, and the case study results, with the seven principles of Lean Software Development, outlined by Poppendieck and Poppendieck in [54].

- *Eliminate waste* can be seen as a core trait also in Software Craftsmanship. By focusing on the *Steadily adding of value*, and principles that encourage that, a responsible craftsman tries to eliminate waste from any processes or tasks.
- *Amplify learning* also lies at the core of craftsmanship, fostering a **C5** *Culture of learning* via **C6.3** *Mentoring* and **C6.2** *Deliberate practice*, and **F4** *Learning from feedback*.
- *Decide as late as possible* is a way to adjust your design up until the last responsible moment, which is core in **D1** *Incremental development*, where **F1.1.2** *Requirement changes* are seen as a comparative advantage.
- *Deliver as fast as possible* puts value on getting real, actionable **F** *Feedback*, on many levels, both via **F3** *Reviews* and **F5** *Continuous integration and tests*, using **F2** *Short feedback loops*.
- *Empower the team* is also at the core of craftsmanship, where the architecture invites **A1.3** *Empowerment*, and the professional culture values **C4** *Visibility and accountability*.
- *Build integrity in* has a direct parallel in the **D** *Iterative design, development, and verification*, where much of the focus is on layered verification in the **D2** *Testing pyramid*, and that the tests should be **D3.1** *usable and readable as documentation* of a running system.
- *See the whole* is arguably the focus of many craftsmanship principles, both from an **A** *Value-focused architecture* theme to the *Productive partnerships* envisioned in the manifesto.

While there are similarities between the lead architect in the studied product and Poppendieck's chief engineer principle [54], there are also differences. The program planning and budgeting were performed by different roles in the studied case, outside the scope of this article. The lead software architect focused solely on the software and its structure to enable efficient development of features valued by customers

while still meeting the required quality requirements. There were also strategic product managers and system managers dealing with customer requirements and strategic directions for the product, also outside the scope of this article.

6.5 Returning to the Software Craftsmanship Manifesto

Looking at the manifesto[13] values through the lens of our anatomy, we find the following:

- "As aspiring Software Craftsmen we are raising the bar of professional software development by practicing it and helping others learn the craft." In the first line of the manifesto, the authors explicitly value the **C6** *Culture of learning*, and the **F4** *Learning from feedback*. The need for constant practice also aligns with **A1** *Participating Software Architects*. Although **F3** *Reviews* are not explicitly mentioned, this is one example of a setting enabling experience sharing, either automated through static review tools or manual, via solution or code review.

- "Not only working software but also well-crafted software" as a statement does not define what distinguishes the two classes of software. Our anatomy considers well-crafted software as being composed of **A3** *Clean, minimalistic code*, which is **D1** *incrementally developed*, during constant **A3.4** *Refactoring*. The architecture enables **A2.1** *isolated features, using layers*, and features are developed with **D2** *layered testing* in mind. Functional tests are written by the **D1.2** *team that develops the feature*, so that they are **D3.1** *readable as documentation*.

- "Not only responding to change but also steadily adding value" focuses on the longer-term perspective and the ability to add value to the software in a predictable manner continually. To meet this goal, in addition to the well-craftedness mentioned above, the **A** *architecture* should focus on helping value-creation, making it easy to validate changes through **F5.1** *Frequent release candidates* and through **F5** *Continuous integration*. To keep track of the current state of the product and the project, **C4** *Visibility and transparency* are important, as is the management of **C4.1.1** *Technical debt*.

- "Not only individuals and interactions, but also a community of professionals" emphasizes the community aspect of software development, and many items in the anatomy focus on a **C** *Shared professional culture*. Important aspects of a **C2** *Common culture* include fostering **C2.1** *caring* for your artifacts, having a shared sense of **C2.4** *Pride*, and **C2.2** *Clear roles and responsibilities*. To

[13] http://manifesto.softwarecraftsmanship.org/

balance the pride, it is also important to keep **C5** *Accountability* and **C5.1** *Humility*, and craftspeople would do well to manage their **C5.2** *Reputation*.

- "Not only customer collaboration, but also productive partnerships" again focus on the longer-term view, where **C5.2** *Reputation* is at stake. Our anatomy mainly focuses on the requirement formalization's collaborative aspects, using the **F1** *On-site customer* approach and **F1.1.2** *Collaborative requirements* elicitation, by constant communication between the design team and the requirement owner (customer proxy). Likewise, verification is a collaborative endeavor, where **D1.2** *teams take responsibility for delivering functionally verified features*.

To sum up, our anatomy makes no references to the "lone cowboy programmer" craftsman stereotype mentioned by Boehm in P14 [76]. Instead, it emphasizes the community aspects of modern software development, the importance of mentoring and tutoring newcomers to the field, and the need for constant learning in software development. While there are undoubtedly programmers that prefer solitude and would rather not communicate with others, our anatomy concretizes most of the manifesto ideas, bringing evidence on how some of the craftsmanship principles can work in practice. It also emphasizes the need for senior developers to engage in teaching and mentoring, in addition to behavioral rules to foster a shared culture of learning and professional development.

To be fair, our anatomy does not emphasize the linear progression of apprentice, journeyman, and master outlined by McBreen in B2 [4]. Rather than designating individuals into specific labeled categories, the anatomy emphasizes everyone's responsibility to contribute to a culture of learning, caring for the codebase and the architecture. Naturally, the more senior developers would take a more leading approach, such as in the cross-team forums. Likewise, leading developers were cognizant of the importance of a shared professional culture and used both team relocation and kata exercises to try to instill a common way of working to new project members, regardless of their prior experience.

7 Validity

In this section, we discuss the threats to validity from four different angles: *construct validity*, *internal validity*, *external validity* and *reliability* [125].

Construct Validity deals with whether the studied measures really reflect the constructs that the researcher has in mind and what is stated in the research questions, and the ability of the metrics to informs about the concept [126].

For the qualitative data, construct validity was enhanced by the two additional authors reviewing the flexible interview protocol, making clarifications based on this

feedback. We also presented an intermediate version of the anatomy to the studied organization, after analysing the interview data, and received valuable feedback.

Much of the quantitative data comes from Git logs, and using such information to illustrate: (i) the proportion of development activities (e.g., feature development or refactoring); (ii) the iterative nature of the development; and (iii) the usage of layered testing; has some risks that can challenge the reliability of the results.

In particular, when dealing with the proportion of development activities, we analyzed individual commit messages and relied on the organization's strict commit tagging policy. Developers had to tag each individual commit with a code depending on the activities they were carrying out. Only 0.2% of the commits were not properly tagged. We tried to mitigate this threat to construct validity by defining a metric on data that was created with the same objective: to be able to identify the development activities. During the studied period, the organization had no organizational goals associated with this metric (e.g., rewards associated to refactorings or bug fixes). Had such goals been used, this metric would not have been reliable, as developers could have been expected to change behaviour to meet such goals.

For analyzing the adherence to incremental development, we use the evolution of the codebase over time, for the major types of source code. One of the main threats to validity in this case is whether the languages (i.e., Java, XML and Scala) are comparable. As XML is much more verbose than Java, it will grow faster, but the main usage in this analysis is not the growth speed itself, but the fact that they grow together in at sustainable pace. In a non-incremental development scenario, we would expect the production code and the unit test code to grow from the start of the project until the start of the development of integration tests, where these two will suffer a sudden decline in their growth and the focus would move to integration. However in this case the different types of code grow linearly, with slightly different speeds.

Finally, regarding our proposed construct of a testing pyramid and layered testing, we use both the fact that developers state that automated tests were important, and the volume and ratio of test code versus production code. Our proposed metrics (lines of code and the ratio of tests versus production code) say nothing about the quality of said code, but they do illustrate that the different classes of code grew over time, and as the product grew more feature-rich, the amount of different test code grew alongside the production code, although at different speeds. We argue that this shows that in this product, developers took care to layer their tests into different categories of tests and that this behavior was consistent throughout the studied period.

An important aspect to consider when using this data source is the branching pattern and how commits were merged or rebased. In the Git version control system, authors may "squash" commits, perhaps performed by different authors at different times, into one new commit, discarding the constituent commits. This was not an approved practice as the studied organization valued seeing the individual commits as they were written and pushed to the central repository.

Most development took place in a single "master" branch for the duration of the study. Features developed in other branches were eventually introduced into the master branch, typically via the Git rebase function, keeping a linear history by rewriting commits. However, during rewriting, the original author information, including the commit date, is preserved, even if the commits are reordered in the git log. This allows statistics based on Git dates to be reliable data sources, as the commit date reflected when the actual code was changed, not when it was introduced into the master branch.

Internal Validity deals with whether there might be other, non-studied factors that could explain some of the findings.

We used the mixed-methods approach of triangulation to increase internal validity. We used Google Scholar to search for papers to form a start set. As we only found 4 relevant papers, we added 5 additional based on experience. This personal bias could threaten internal validity. However, we believe that its impact is minimal after performing four forward and backward snowballing iterations. We have screened 478 references, 782 citations, and 146 books during these iterations. Moreover, Mourão et al. have shown that combining the database search with forward and backward snowballing improves the precision and recall of the literature review [61].

Where possible, we used both quantitative and qualitative data sources. However, there might still be other, non-studied, explaining factors that impact the results. We are aware that the studied development project did not adopt all software craftsmanship principles that we identified in the literature. This remains a threat to internal validity of our work.

External Validity concerns the extent to which it is possible to generalize findings and whether the findings are of interest to people outside of the investigated case.

One of the five misunderstandings about case study research is the inability to generalize from a single case [42]. Following Flyvbjerg, we have focused on analytic generalization rather than statistical generalization by comparing the characteristics of the case to a possible target and presenting case-specific characteristics, as much as confidentiality concerns allowed.

We looked outside the studied case by reviewing other literature for findings or themes to increase external validity.

This buttressing is documented in the SLR section of the article, and the associated data appear as references throughout the results and analysis sections. However, it must be acknowledged that this buttressing is based on limited empirical evidence. Additionally, the results here are only circumscribed to the analyzed context. More studies in other systems and other organizations are needed to better understand the effect that craftsmanship principles might have on the developed product, the development process, and the organization.

Reliability concerns whether the data and analyses are dependent on the spe-

cific researchers, and this is a significant threat to validity for this study, as the first author was part of the studied product development during the whole studied period. To increase reliability, the second and third authors were used in a supporting role, with at least one of them being active participants in all interviews. The first author transcribed all recorded interviews. The transcripts were reviewed by the second and third authors, who separately coded three interviews each, for comparison with the first author's codes, who coded all interviews.

The interviews, conducted between July 2018 and January 2019, used a convenience sample of participants, focusing on including many different aspects, illustrating the concepts and principles used in the development process. Two interviewees were from the outsourced site, and two were women. The lead architect was interviewed separately by the second and third authors, as he had worked closely together with the first author during the studied period.

A threat to reliability is that the interviews took place some years after the actual studied events. In addition to memory errors in the interviewed participants, it also meant that it was hard to reach persons who were part of the product for a shorter time. Thus, the views of such "short-lived" participants may have been different than the interviewees.

We strove to reduce memory errors by seeking additional data in quantitative sources (VCS logs, wikis, requirement tools) using archival analysis whenever possible.

8 Conclusions and Future Work

8.1 Conclusions

Regarding **RQ1**, how Software Craftsmanship has been conceptualized in literature, although the principles have a long history in gray literature, we found comparatively few published research articles. In our SLR, we could find only 18 papers discussing the principles to some extent, see Table 4. Based on these papers, we found 11 books, of which seven were new to us before starting this study.

In order to conceptualize the findings, and to illustrate which of these principles and practices that we can see in our studied case (**RQ2**), we drew the anatomy map, comprising of four key themes, with 17 principles and 47 practices; see Figure 3 and Table 6, 8, 10 and 11.

In answering **RQ3**, what consequences applying the practices bring, we drew examples from our studied case, using both quantitative and qualitative data. Most of these principles align well with core Agile and Lean principles but place a higher weight on the technical practices.

Although the Agile and Lean principles seem quite well-researched, the Software Craftsmanship principles seem to warrant more systematic studies by the re-

search community.

8.2 Future Work

This study was performed in a particular setting, having quick feedback cycles from customers with rapidly changing requirements. Whether the principles still apply in other settings, such as in situations with more static and stable requirements, or different organizations, remains to be seen.

In future studies, we intend to study how these practices have affected the defect statistics, internal and external quality, and how the principles have been applied as the organization has changed. We also plan to explore the relationships between Agile and Lean software development and software craftsmanship. We are aware that both Agile and Lean software development have aspects similar to, and overlapping with, software craftsmanship. Thus, we would like to explore this in detail in subsequent publications.

Paper II

The Hidden Cost of Backward Compatibility: When Deprecation Turns into Technical Debt

Anders Sundelin, Javier Gonzalez-Huerta, and Krzysztof Wnuk.
In: Proceedings of the 3rd International Conference on Technical Debt (TechDebt '20) Association for Computing Machinery, New York, NY, USA, 67–76. https://doi.org/10.1145/3387906.3388629

II

Abstract

Context: The micro-services architectural pattern advocates for the partitioning of functionality into loosely coupled services, which should be backward compatible, to enable independent upgrades. Deprecation is commonly used as a tool to manage multiple versions of methods or services. However, deprecation carries a cost in that tests might be duplicated and might rely on services that have become deprecated over time.

Objective: Using the terms of the Technical Debt metaphor, we explore the consequences of deprecation, and how it has affected the test base during seven years.

Method: We take an exploratory approach, reporting on experiences found before and after servicing parts of the incurred Technical Debt. We mine code repositories and validate our findings with experienced developers.

Results: We found that the growth of deprecation debt varied a lot. Some services experienced substantial growth, but most did not. Unit tests, where deprecation is visible in the developers' tools, were much less affected than integration tests, which lack such visualization mechanisms. While servicing debt of 121 out of 285 deprecated services, we discovered that up to 29% of the spent effort could be attributed to accrued interest. However, this is an upper bound; there could be less impact, depending on whether scripting could be used to service the debt or not.

Conclusion: This paper illustrates that integration tests can be viewed as a debt from the perspective of deprecated services. While the pattern

was that deprecated services (debt principal) experienced no or little accrued interest, some, highly used, services experienced a lot, particularly during stressful times. Java-based tests, where deprecation is visible in the IDE, did not experience a similar pattern of increasing debt. We postulate that deprecation debt should be kept visible, either using developer tools or statistical reports.

1 Introduction

The current trend in software engineering is to split functionality into small, focused, loosely coupled services, often called microservices [127], using the DevOps moniker [128]. To achieve loose coupling, services should be independently developed. Zimmermann et al. [129] highlight that continuous upgrade-ability and backward compatibility are necessary for achieving loose coupling because they allow clients to upgrade to newer available services at will, irrespective of how the service is upgraded.

However, we have not found any study that explores the converse problem—the importance of removing unused services, and how to visualize whether or not the clients use "the most current" version of a service.

Exposing several versions of the same service carries a cost, both from a development and verification point of view. The test base can be seen as "the first client" of a service, as it is the first code that exercises the service in order to verify the expected behavior.

This paper describes the impact of service deprecation on an existing test base, and the effect that this has had on a product over the course of seven years. We take an exploratory approach and mine the Git repository of the studied product for data related to the exposed services.

A core principle used when developing new versions of the services in the studied system has been backward compatibility. A new version of service should behave identical to the currently existing version, though it might accept other types of data, or return more or less data, based on the given input. Deprecation markers have been used to flag which versions to avoid. The studied system contains one non-deprecated version and N (≥ 0) other deprecated versions of any published service.

The paper is structured as follows: In Section 2, we cover the related work in the area. In Section 3, we report on the research methodology, and relate to the background of the studied system. In Section 4, we report the results found during the study. In Section 5, we discuss the threats to validity, and in Section 6, we draw conclusions and elaborate on the implications for the research area and industry in general.

2 Related Work

Deprecation is a way to signal to API consumers that there are other, more up-to-date ways to perform a given task. Morgenthaler [130], views deprecation as software aging made manifest, and maps this directly to technical debt.

A complication is that very often, deprecated methods or services remain for very long times in APIs. As an example, the Java Standard Library class *java.util.Date* has contained four deprecated constructors and 18 deprecated methods since the release of Java 1.1 in 1997. Today (in Java 12[1]), these deprecated methods and constructors are still present, together with two non-deprecated constructors and 11 non-deprecated public methods.

Sawant et al. [131] studied how developers (API consumers) in open-source projects react to deprecation events in popular Java APIs. The overall conclusion is that developers seldom react at all, and rarely keep up with API evolution. A weakness of this study is that it focuses only on open-source projects. In [132], the same authors, apart from increasing the number of studied projects, also conducted a survey, trying to reach out to developers on proprietary projects. The conclusion in this paper is similar to the prior study, and respondents point to the perceived complexity and time consumption as the main reasons for not upgrading to the latest version of the APIs and keep using deprecated versions.

The same finding was made in a study by Kula et al. [133], who investigate the efforts of library migration covering over 4600 GitHub software projects and 2700 library dependencies. Results from this study show that 81.5% of the projects kept their outdated dependencies, citing unawareness, extra effort, and added responsibility.

As of Java 9, the deprecated annotation has been enhanced[2] with an attribute indicating that a construct is deprecated with the intent to remove it in a future version. The motivation for this is that the deprecation marker ended up being used with different purposes, and the theory is that if developers realize that a construct is deprecated with the intent to be removed in a future release, they are more likely to update. As part of this work, there was also additional tools developed, available as part of the Java ecosystem.

Snipes and Ramaswamy [134] proposed a sizing model for managing technical debt related to the deprecation of third-party software components. The model, based on a sigmoid curve, takes into account the lines of code affected, how much (as a percentage) of the API that is deprecated, and the age (in years, with a cutoff proposed to 5 years) of the software component. The authors illustrate the model using Monte-Carlo simulation.

Curtis et al. [135], proposed estimating the principal amount of technical debt based on static code analysis tools and a parameterized formula. The authors illus-

[1] https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/Date.html

[2] https://openjdk.java.net/jeps/277

trate the model by using the tool and formula on 700 applications, comprising 357 MLOC in various languages, from 158 vendors. The article does not state whether only production code was analyzed, or if test code also was included in the metric. A conclusion is that the proposed formula is highly sensitive to the given parameters. The authors illustrate this with three sets of parameter values.

Codabux and Williams [136], studied an organization adopting Agile practices and focused partially on the management of technical debt. Research questions center on characterizations, consequences on the development process, addressing, and prioritization of technical debt. Test debt in the study relates mostly to missing tests and lacking automation of tests.

Kruchten et al. [137], relate to *Testing Debt* in three different ways: "*Imperfection or suboptimal design and coding of tests*", "*Misalignment between the tests and the actual code*" and "*Challenges of SaaS contexts*". Tests, especially when automated, are also code and need to be designed just like production code. When the purpose is unclear, time is spent trying to figure out whether failures originate from the tests or the production code.

3 Research Methodology

We take an exploratory approach, utilizing mixed methods and triangulation to elicit experiences from the studied organization. As the studied product has been developed for nine years (of which seven in live operation), a complication is that many developers are no longer available for feedback on conclusions. When possible, we try to confirm and contrast our findings with experienced developers, with a long track record in the product (a handful of which has been with the product since the start). We mined the version control system (Git logs), where all changes to files are stored.

The developers of the studied system took principled decisions related to the support of multiple versions of exposed services:

- New versions are introduced when the existing service cannot accommodate newly requested features, but the core function is the same.
- When introducing a new service version, all old service versions are deprecated. This leads to each service having exactly one version that is "current."
- When a version is deprecated, it remains in the product, and existing test cases continue to use it. New test cases are written for the new version, but old ones are not converted, even if they use the old, deprecated version of the service.
- Removing a service version is done as a planned task, based on extensive communication with impacted stakeholders and requires updates to all systems (in-

cluding test cases that might use it for other purposes than explicitly testing the service version itself).

3.1 Research Questions

Based on the above principles, we formulate the following four research questions:

- **RQ1** How has the decision to avoid converting the existing test base when deprecating a service version contributed to the spread of Technical Debt?
- **RQ2** If there is a contribution to Technical Debt in the test code base, are there any differences between the Java-based files (where deprecation is visible in the IDE) and the non-Java-based text files (where no such feedback is visible)?
- **RQ3** How has the growth in deprecated service version usage contributed to the spread of Technical Debt?
- **RQ4** What is the likely cause of the spreading of deprecation debt?

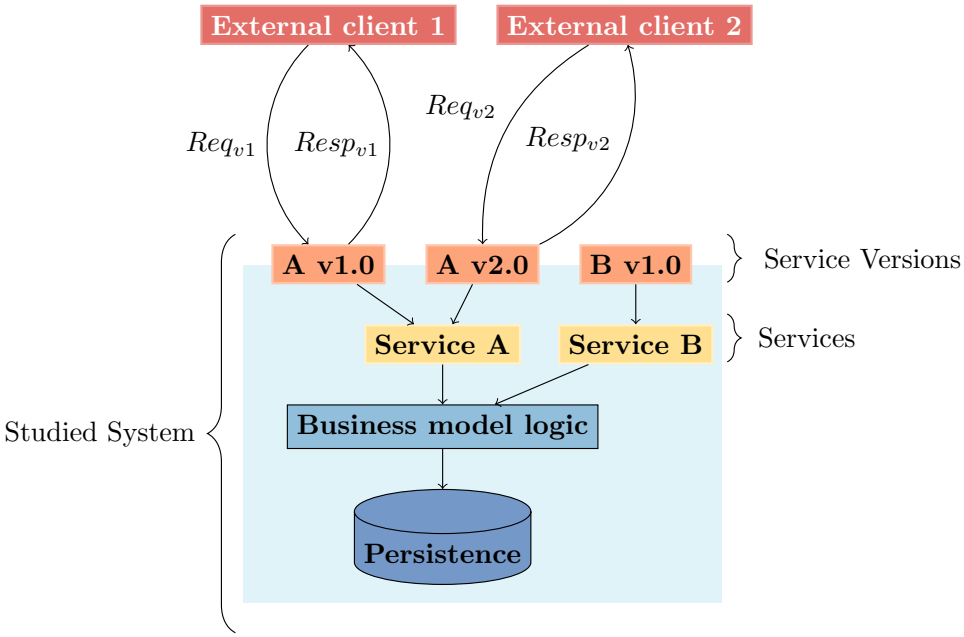
3.2 Case Description

The system under study forms part of a FinTech global product that enables access to financial services via mobile phones and the Internet. It is typically installed in a high-availability configuration, with geographical redundancy, to meet service uptime requirements. Starting in 2013, the system today serves tens of millions of users in around 15 installations around the world, each integrating with several hundred different other systems. The system contains several transaction-intensive applications, with incoming and outgoing interfaces and persistence layers in the form of different databases. As it is a financial application, security plays a dominant role.

Figure 1 illustrates the system and the role of different services and how each service exposes different versions of requests and responses. Clients external to the system may choose to use any of the exposed service versions to invoke a particular service.

The studied system was developed with test automation in mind, using different test strategies, such as unit testing, integration testing using public APIs, and system testing by a different organization than the development teams. The development has been conducted in an Agile manner, with frequent releases, and incorporating core Agile practices, such as continuous integration and as short feedback loops as is practical. Continuous Integration tools such as Jenkins, giving fast feedback to developers on the quality of their product, has been used during the whole development period.

The system exposes a set of HTTP-based services to the surrounding systems, using XML for marshalling and unmarshalling. As the number of clients to the sys-



(a) External client 2 uses service A in version v2.0 (request Req_{v2} and response $Resp_{v2}$) External client 1 has not yet upgraded, and is still using v1.0.

Figure 1: Schematic view of the studied system, exposing services in different versions.

tem is essentially unbounded (or at least unknown at design time), backward compatibility on the protocol level has been an important factor in the design of the system. For this purpose, protocol schemas were developed using XML Schema (XSD), and the system developers assumed that at least some clients, possibly also security firewalls, would enforce strict validation of the request and response messages against the corresponding schema. Due to the compatibility principle, the number of versions of the same service has grown over the years. Another principle has been that when developing a new version of a service, all older versions are marked as deprecated, using the regular Java deprecation mechanism (annotation and Javadoc). This information was included in the product API documentation, which was generated from Javadoc.

As the system has been developed in Java, deprecation warnings appear integrated into the IDE, such as Eclipse and IntelliJ IDEA. However, integration test cases, interacting with the services via officially supported interfaces, such as XML requests over HTTP, has been developed in a text-based language, custom written for the application. This language is also specified in XML and is executed via a custom runner.

The use of XML text is similar to text-based BDD[3] languages, such as Gherkin, but has proved hard to sustain for the 9000 unique test cases. Current developers

[3] https://dannorth.net/introducing-bdd/

are unwilling to modify existing test cases, and express frustration at the lack of IDE support for working with plain-text-based languages. This is in contrast to the comparatively well-developed search and refactoring support in IDEs such as Eclipse or IntelliJ IDEA. As the XML language lacks the deprecation mechanism that is available in plain Java, there is no warning when using a deprecated version of a service. Thus, the developer writing or maintaining test cases lack feedback on whether or not this should be changed.

Figure 2 shows the evolution of the number of files of production code (*prod*) and tests over the studied period. The combination of *int.tc* and *int.setup* files contains the integration test files, with *int.setup* referring to the code used for setup, tear-down, and utility functions used in integration test cases, whereas *int.tc* refers to the actual code of the integration test cases. *Unit* refers to the Java-based test code, typically in the form of unit tests.

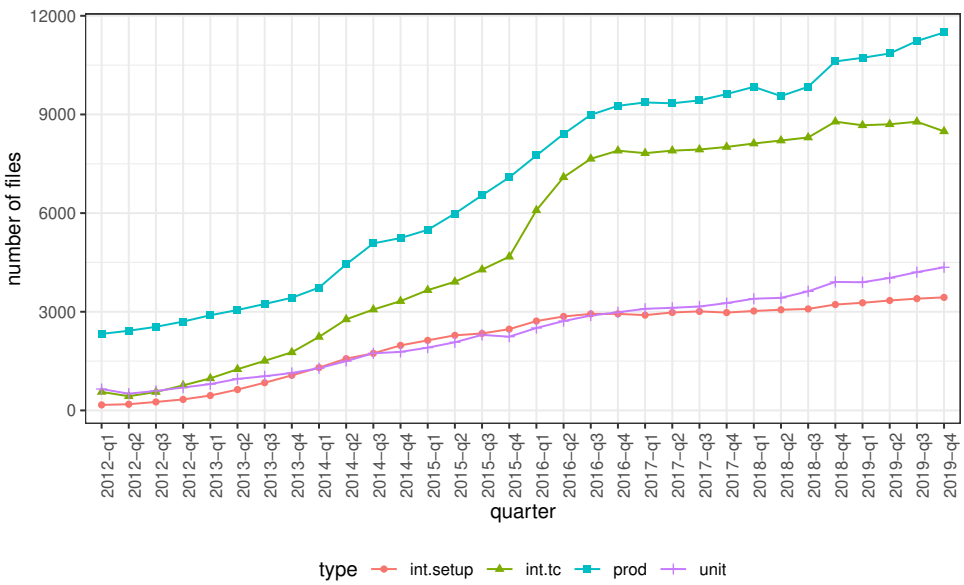


Figure 2: Production and test files per quarter

The Java-based files, both production and unit tests, show steady growth over the studied quarters. Employing standard linear regression, we see that production files grow with 330 files/quarter, p-value less than $2 * 10^{-16}$, and adjusted R^2 of 0.97. Likewise, the unit tests grow with 128 files/quarter, p-value less than $2 * 10^{-16}$, and adjusted R^2 of 0.99.

For the integration test cases (*int.tc*), the picture is somewhat different. The period up to and including Q4 2015 exhibit a growth rate of 300 files each quarter, with a p-value of less than $8 * 10^{-13}$ and an adjusted R^2 of 0.97. Then, the ratio increases, between Q4 2015 and Q4 2016, the growth rate is 801 test-case files, with

Table 1: Average lines per file across quarters.

Type	$\bar{x}$	σ	$\hat{x}$	first	last	min	max
prod	73.4	3.6	73.8	74.2	88.4	66.8	88.4
unittest	172.3	24.4	181.8	139.6	234.5	133.1	234.5
int.tc	155.0	10.3	154.8	187.2	148.5	135.4	187.1
int.setup	83.4	23.1	82.8	117.8	61.5	61.5	135.6

Table 2: Average number of authors each quarter, per year

Year	$\bar{x}$	σ	$\hat{x}$
2011	26.00	7.9	28.0
2012	36.25	2.1	36.0
2013	42.25	6.2	40.5
2014	50.75	1.3	51.0
2015	58.00	3.7	58.5
2016	86.00	5.0	86.5
2017	25.00	10.7	20.0
2018	41.50	15.2	47.5
2019	46.75	3.9	46.5

a p-value of less than 0.010 and adjusted R^2 of 0.89. As of Q1 2017, the growth rate has flattened, and is now 84 files per quarter, with a p-value of $2 * 10^{-5}$ and adjusted R^2 of 0.80. The last quarter of 2019 shows a decrease in test case files, which reflects the fact that one developer removed, in his opinion, unnecessary integration tests, by converting some to unit tests and removing others altogether, in a refactoring operation. For the integration test setup files and shared functions (*int.setup*), the picture is slightly different. Up until and including Q2 2016, the setup files were growing with 175 files/quarter, p-value less than $3 * 10^{-16}$ and adjusted R^2 of 0.98. As of Q2 2016, the growth stagnates to 41 files/quarter, p-value less than $4 * 10^{-8}$ and adjusted R^2 of 0.90.

When discussing efforts in terms of counts of files, it is important to note that each file can grow as well. Table 1 contains data regarding the average lines per file type, and its variation across the quarters. For all four file types, the mean ($\bar{x}$) and median ($\hat{x}$) are quite similar, and the standard deviation is also quite small. The *first* column refers to Q1 2012, and the *last* to Q4 2019 (before servicing the debt). Relative to their size, the setup files have the largest standard deviation. Both integration test cases and setup files have a downward trend in file size across the quarters, while the Java-based files (production code and unit tests) are slightly growing, unit tests more so than production code.

The number of authors per year has varied somewhat, as illustrated in table 2, where the number of unique authors, as identified by the Git *Author:* tag, each quarter is summarized (4 quarters per year). After steady growth until 2016, the last year explosively, there was a contraction during 2017, followed by slower growth until the end of the study period.

4 Results

4.1 The origin of deprecation debt

When faced with changed or additional requirements related to an existing service, developers face a dilemma:

Update the existing service, to accommodate new requests or response parameters, or change existing parameters. An equivalent solution is to delete the old service, and introduce a new one with the same name. No Technical Debt would be associated with this option.

Keep the old service version and introduce a new version of the same service, where the required changes are made. The old service version is deprecated, and is associated with a Technical Debt principal, as it has to be maintained alongside the new version, which solves a similar business purpose (being the same service). The consequences of keeping the deprecated version are not only visible at code level, but might be even more severe at the testing level since the integration tests, as will be illustrated later, can keep using the old versions of the service. The propagation effects of deprecation to the test cases are one example of a code TD-item that appears as a TD-item in the testware [138].

Either of the two options can be exercised, and for the studied system, the favored solution was **Update**, unless this was prevented by protocol conventions. Reasons for being unable to choose **Update** were such that would cause existing clients to break, for instance:

- Changing the XML Schema type of an existing parameter (e.g. `xsd:integer` to `xsd:string`), in either the request or the response.
- Adding a new mandatory request parameter P, as this would break old clients who would not send P in their requests towards the system.
- Making an existing optional request parameter P mandatory, as existing clients would be unaware that P was required.
- Adding an optional or mandatory parameter in a response, as this would break those clients who would perform strict schema validation on received responses.

Some of the reasons for choosing option **Update** were:

- If the existing service had not yet been part of any product release, so external clients would not have had the opportunity to interact with it, or learn about the API.
- Adding an optional parameter P to a request, as existing clients could refrain from sending P.

- Refraining from returning an optional parameter P in a response, as existing clients would already have had to deal with the absence of P when interpreting the response.

Initially, the system had no monitoring of which version of a service is currently used by which customer. This monitoring was introduced in 2017. During late 2019, there were efforts made to clean up the unused, deprecated service versions. Based on usage data, 121 deprecated service versions were identified and removed. At the end of the studied period (before cleanup), the system comprised 632 unique services, which exposed 891 different versions of services. In other words, almost three out of ten (29%) of the service versions were deprecated at the end of the studied period.

Kruchten et al. [137] refer to debt principal as proportional to the effort that a development team would expend to eliminate it. The interest incurred by a technical debt item is the additional effort needed to eliminate the debt if the item is left in the system. This view is also shared by others [130]. For our deprecated service versions, the principal would include the effort needed to remove the integration test cases (*int.setup* and *int.tc*), plus the production code (*prod*) and unit tests (*unit*) maintenance between the introduction of the usage statistics and the pay-back in Q4 2019. We classify our deprecated service versions as two different types of Technical Debt Items (TD-items):

- **Type I TD-items:** deprecated versions of services that the usage metrics reported as not in use in any external customer installation.

- **Type II TD-items:** versions of services, that, although deprecated, were reported as in use in some external customer installations by the usage metrics.

While Type I TD-items require a relatively small effort to remove the service versions (relatively low principal), the second Type II TD-items requires more coordination with the customer adaptation teams, for them to adapt the customer installation to the ultimate versions of the service before the removal of the deprecated service versions both from the code and test base.

We treat production code and unit tests similarly, as they are both specified in Java, a statically typed language. Likewise, we count both integration test cases and their setup files together, as they both are specified in untyped XML.

$$javacode = prod + unittest$$

$$int.test = int.tc + int.setup$$

We will use the following set of equations to calculate the usage of a particular service version v at quarter q in files of type t:

$$FILES(q, t) = \{f : quarter = q, type(f) = t\} \tag{II.1}$$

$$PRESENT(v, f) = \begin{cases} 1, & \text{if } v \text{ occurs in file } f \\ 0, & \text{otherwise} \end{cases} \tag{II.2}$$

$$COUNT(v, q, t) = \sum_{f \in FILES(q,t)} PRESENT(v, f) \tag{II.3}$$

We define $COUNT(v, q, t)$ as the number of files of type t in which service version v was present in quarter q. We choose the number of files rather than the number of occurrences with the hypothesis that the marginal cost of changing an additional occurrence, once a file has been found, is negligible.

To calculate the distribution of service versions throughout the different kinds of files, we select quarter Q4 2019 (before servicing the deprecation debt), and enumerate all the 891 service versions. For each service version, we calculate the $COUNT(v, 2019Q4, t)$, values, where t is either Java (*java*) or XML files (*int.test*). The result is plotted in figure 3, as a histogram with 30 logarithmic bins.

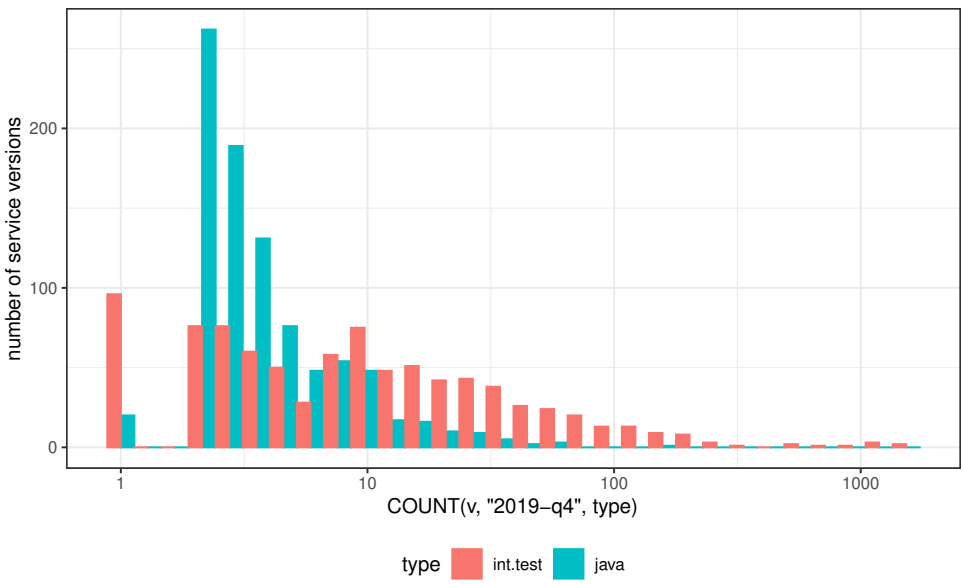


Figure 3: Occurrences of services in Java or integration test code

Note the heavily right-skewed distribution, where relatively few service versions are heavily used, while the majority of service versions are used in only a handful of files. Occurrences in Java-based files occur more seldom (closer to the y-axis) than integration tests, though 96 service versions are present in only one integration test file. This is also visible in table 3, where the Java-based code have a considerably smaller mean ($\bar{x}$) and median ($\hat{x}$) than the integration tests. Table 3 also displays the number (N) of different service versions (noting that 24 service versions that are

Table 3: Occurrences in files.

Type	N	$\bar{x}$	σ	$\hat{x}$	$Q_{90\%}$	$Q_{95\%}$	max
javacode	891	5.3	7.9	3	10	15	165
int.test	867	31.0	110.0	8	57.4	106	1529

used in Java code are not used in integration tests), standard deviation (σ), 90th and 95th percentiles, and the max value.

Production code and Java-based tests, such as unit tests, typically use a broader spectrum of interfaces, such as internal domain-model interfaces, whereas the primary purpose of the integration tests is to exercise and validate the external application interface (that is, the exposed services, in their different versions). The most heavily used service version appears in 1529 integration test files (including setup files), out of around 12300 files.

4.2 Deprecated services

All analyzed services start as "non-deprecated," in version 1.0, but some will evolve together with the product to later versions.

Algorithm 1: Calculating usage of deprecated service versions

Result: Time series of usage data for each deprecated service version following deprecation

$end\_of\_study \leftarrow Q4\_2019$;
for $v \leftarrow all\_deprecated\_service\_versions$ **do**
  $when \leftarrow find\_deprecation\_timestamp(v)$;
  $following\_quarter \leftarrow next\_quarter(when)$;
  for $q \leftarrow following\_quarter$ **to** end_of_study **do**
    $count[v, q, Java] \leftarrow COUNT(v, q, Java)$;
    $count[v, q, XML] \leftarrow COUNT(v, q, XML)$;
  end
end
return *count*

Algorithm 1, illustrates the process to collect the usage data for every deprecated service version, starting at the quarter following the deprecation event as recorded in the Git version control log. Thus, the initial data point for a service version closely reflects the state at the time when it was deprecated. In the ideal world, each service version should show a flat or decreasing line, as its usage decreases as the product evolves. It is important to note that also deprecated service versions, which are still in use and delivered with the product, should have some usage in tests (the target is not zero files). However, typically this number should be relatively low (in the single

digits), as the officially supported version should be used instead.

Figure 4 depicts the situation for the six most frequently used deprecated service versions[4], and figure 5 for the following six. All of these occur most frequently in integration tests. A few service versions are fairly flat, a few show moderate growth, and a few show extreme growth up until Q2 2016. Following the end of 2016, most of the usage is flat, and there are few changes in usage.

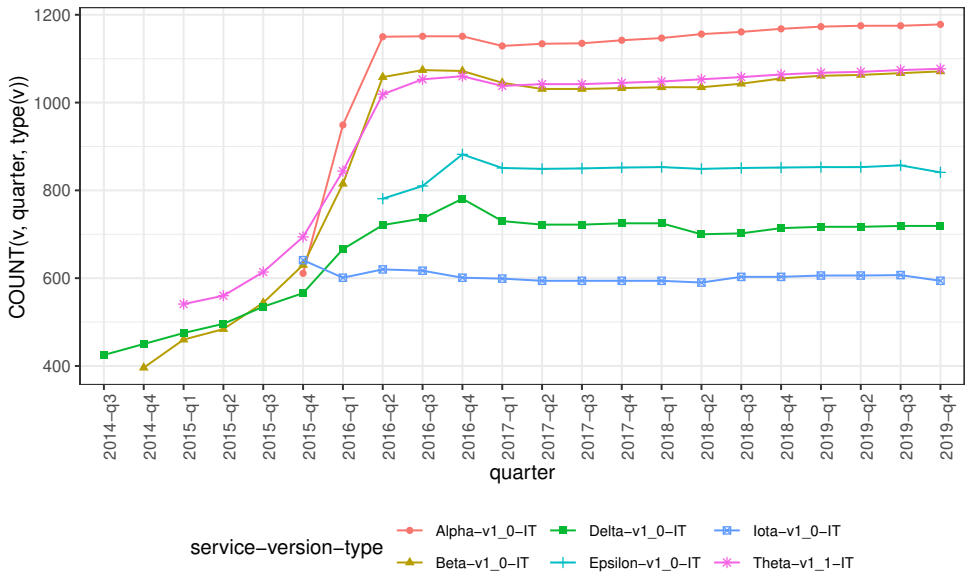


Figure 4: Top 6 used deprecated service versions per quarter

The situation in the six deprecated services most frequently used in Java files is depicted in figure 6. While there is a slight increase for some services, in general the situation is much more stable than for the integration tests, and the number of affected files are also much lower.

We note that some services (e.g., *Epsilon*, *Eta*, *Iota*) appear both among the top 12 integration tests and in the top six Java-based files. These are "core services," central to the application at hand. Furthermore, we note that one service, *Iota*, in different versions, is both the sixth and the eighth most used in integration tests.

Most of the quarters, there is no change in the usage of the deprecated service versions. Out of the 7849 data points (one deprecated service version in one quarter is a data point), growth is flat in 6584, i.e. given a random service in a particular version, and a random quarter following the deprecation of that service version, the chance is nearly 84% that there has been no change in usage. The odds for decreasing usage is 5.1% (413 cases), and increasing is 10.9% (852 cases).

[4]The names and nature of the services are obfuscated due to confidentiality and security reasons.

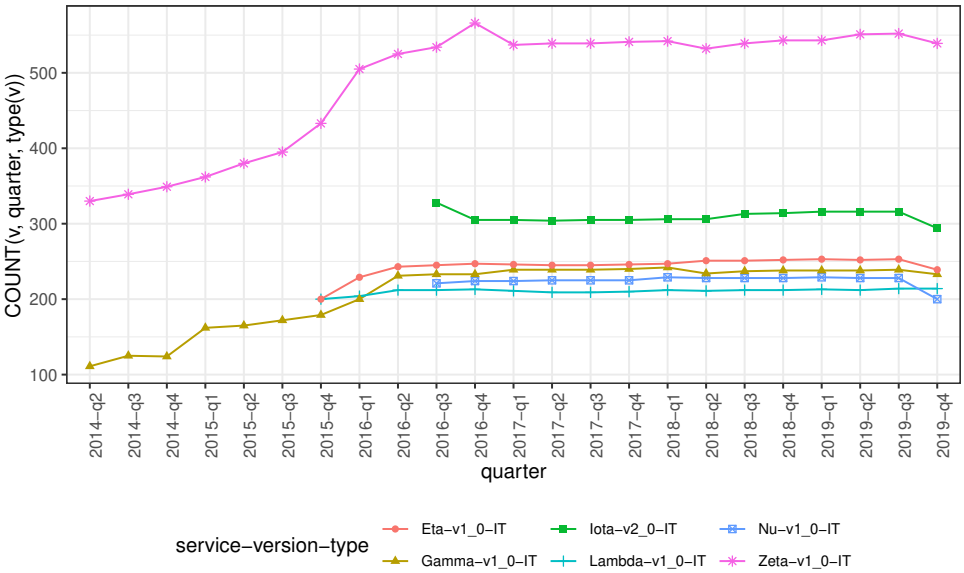


Figure 5: Top 7-12 used deprecated service versions per quarter

Table 4: Most growing deprecated service versions, per quarter.

Service	Quarter	Count	Growth	Dupl	% Dupl
***Alpha v1_0**	Q1 2016	949	338	66	19.5%
Beta v1_0	Q2 2016	1058	243	77	31.6%
***Alpha v1_0**	Q2 2016	1150	201	74	36.8%
Beta v1_0	Q1 2016	815	185	65	35.1%
Theta v1_1	Q2 2016	1019	175	65	37.1%
Theta v1_1	Q1 2016	844	150	48	32.0%
Delta v1_0	Q1 2016	666	100	0	0.0%
Beta v1_0	Q4 2015	630	86	53	61.6%
Theta v1_1	Q4 2015	694	80	42	52.5%
***Zeta v1_0**	Q1 2016	505	72	0	0.0%

Count is the $COUNT(v, q, t)$ metric.
Growth is the change in *Count* since the prior quarter.
Dupl is the number of duplicates among the *Growth*.
% Dupl is the relative number of duplicates.
Type-I TD-items in bold.

Looking at the quarters with the most growth in absolute terms, we see the same three quarters as in figure 4. The top 10 growth operations and quarters are illustrated in table 4. In the table, the service versions *Alpha v1.0* and *Zeta v1.0* is marked in bold, as being Type I TD-items (deprecated and not being used in live operation). The column *Count* contains the $COUNT(v, q, int.test)$ metric for the specified service version and quarter, while the *Growth* column, contains the change since the previous quarter. The *Dupl* column is the absolute number of duplicated files of the

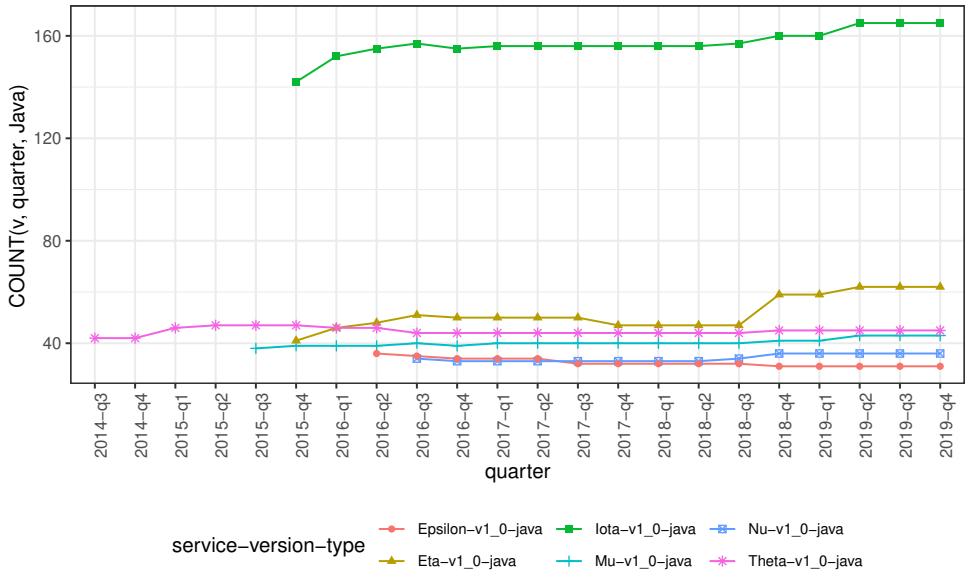


Figure 6: Top 6 used deprecated service versions in Java files

Growth, and the *% Dupl* is the relative percentage of duplicates, compared to the *Growth*. We see that a small number of service versions cause most of the growth in debt. Furthermore, this debt is concentrated to three quarters, spanning Q4 2015 to Q2 2016. Inspecting the *% Dupl* column, we see that for some service versions (*Beta v1.0*, *Theta v1.0*), over half of the added files are duplicates of each other. This indicates that the developers were unaware of a feature that was added to the test execution runner in Q4 2015, where a "shared function repository" could be defined. All of the duplicated files except one are setup files or "utilities" for the test cases.

In figure 7, we plot the relative number of file duplicates across each quarter, and the lines of code in these files. We see that as of 2014, around 15% of the files are duplicated, though these only make up of around 5% of the total code base.

In table 5, we instead look at the service versions and quarters where the most deprecation debt has been repaid (in the form of files that no longer are used). We see that five out of the top 10 service versions were repaid during Q4 2019. Further analysis of the version history logs revealed that this is related to the prior mentioned refactoring illustrated in figure 2, where one developer removed many integration tests, converting some of them to unit tests.

In order to conclude as to how the duplicated files shown in table 4 were introduced, we analyzed the version control logs. The analysis indicated that the files, to a large extent, originated from experienced developers. All in all, 102 commits were affected, authored by 25 different developers, and the top 5 contributors caused 73.5% of the duplicates. These top 5 contributors all had several years of experience

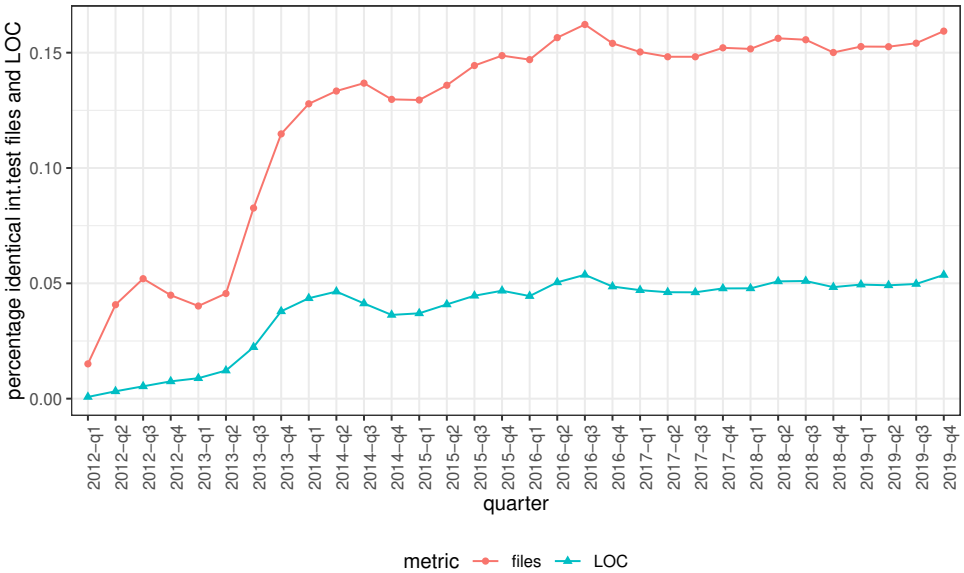


Figure 7: Percentage of identical integration test files and LOC

Table 5: Least growing deprecated services, per quarter.

Service	Quarter	Count	Growth
Nu v1_0	Q4 2019	200	-28
Psi v2_0	Q4 2019	44	-28
Chi v1_0	Q4 2019	20	-28
***Zeta v1_0**	Q1 2017	537	-29
Epsilon v1_0	Q1 2017	851	-31
Iota v1_0	Q1 2016	601	-40
Delta v1_0	Q1 2017	730	-51
Zeta v1_7	Q3 2018	10	-60
Phi v1_0	Q4 2019	21	-105
***Zeta v1_4**	Q4 2019	36	-148

Count is the $COUNT(v, q, t)$ metric.
Growth is the change in *Count* since the prior quarter.
Type-I TD-items in bold.

in the product at the time of introducing the duplication.

Figure 8 illustrates the concept of principal, which, for Type-I TD-items, are proportional to the usage in the code base, and interest, which is proportional to the increased usage. The *First* metric is the usage at the quarter following deprecation (different quarter for each service version), and the *Last* metric is the usage at the end of the study, before servicing the debt. Usage before deprecation is not counted as principal by this metric.

In table 6, the $\sum First$ column is the sum of all the *First* values for all N deprecated service versions and the $\sum Last$ column is the sum of all the *Last* values.

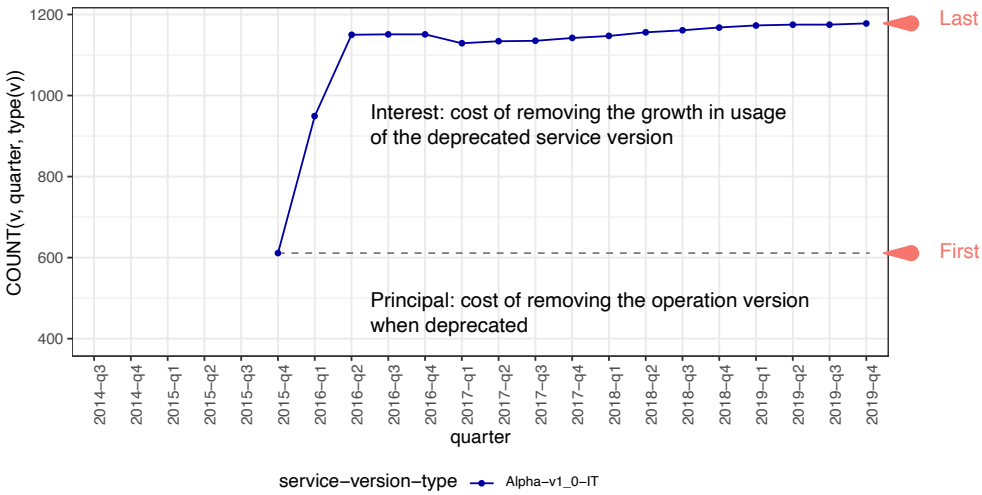


Figure 8: Schematic illustration of Principal, Interest, First and Last metric for integration tests of service Alpha v1.0

Table 6: Affected files for all deprecated services

Type	N	$\sum First$	$\sum Last$	$\sum Add$	$\sum Del$
javafiles	285	1620	1878	292	-34
int.test	282	10619	13526	3819	-912
Sum		12239	15404	4111	-946

The $\sum Add$ column totals the net additions, and the $\sum Del$ column the net removals. In case one file contains multiple instances of the same deprecated service, it is only counted once, but in case it contains several deprecated services, it is counted once per such service. This uses the hypothesis that changing several instances of the same pattern in a single file incurs a relatively minor additional cost, compared to finding the file and changing in one place.

The cost of removing a service version V_{depr}, which has a non-deprecated version V_{actual} can be broken down into the following components:

$$C_{actual}(V_{depr}) = C_{internal}(V_{depr}) + C_{external}(V_{depr}) \tag{II.4}$$

$$C_{internal}(V_{depr}) = C_{removal}(V_{depr}) + C_{update}(V_{depr}, V_{actual}) \tag{II.5}$$

where

- $C_{actual}(V)$ is the actual cost of removing V.
- $C_{external}(V)$ is the cost of removing V by integrators and other clients (outside the development organization).

- $C_{internal}(V)$ is the cost of removing V that is internal to the development organization (code base, including tests).

- $C_{removal}(V)$ is the cost of removing only V and its associated test cases.

- $C_{update}(V_1, V_2)$ is the cost of updating all the remaining tests from V_1 to V_2.

Using the Technical Debt metaphor, the $C_{actual}(V)$ at the time of deprecation can be considered as the principal, as the alternative would be to simply remove V, replacing it with the updated version. We can estimate $C_{update}(V_1, V_2)$ as the additional usage of V_1, not related to the normal tests of this version. In integration tests, it is common that certain operations are used for setting up the data in order to test other services, such as when a login service is used in several integration tests across the test base. This is what we call *excessive usage* of a service version, as opposed to the *expected usage* of a service version which is its source code, its unit tests and integration tests of the service version itself.

For Type-I TD-items, where the $C_{external}$ is zero, we find that the $TD\_principal$ is proportional to the usage at the time of deprecation, that is, $\sum First$. The accrued interest is proportional to the growth in usage, that is, to $\sum Last \ - \ \sum First$.

$$TD\_principal_I \propto \sum First$$

$$TD\_interest_I \propto \sum Last - \sum First$$

Thus, the relative growth of $TD\_interest_I$ for Java files is $\sum Last / \sum First - 1$, that is $(598/505) - 1$, or 18%. For the integration tests, the corresponding number is $(3428/2383) - 1$, or 44%. Thus, the interest rate for integration tests is much higher than for Java-based tests. We also note that there have been some decreases. In total, 268 files have been corrected for the 120 different integration test service versions, but this has not been able to keep up with the growth.

For Type-II TD-items, the situation is somewhat more complicated, as one also has to consider the lack of test cases as a debt principal, what Kruchten et al.[137] refers to as "*Misalignment between tests and code*." We note that there are three service versions not tested in integration tests (Table 6, N for $int.test$), and one of these was among the Type-I TD items (Table 7).

4.3 Servicing the debt

We removed 121 deprecated service versions in late 2019, see table 7. We note that one of the removed service versions was not present in any integration tests (closer inspection revealed that those tests had been migrated to the new version at the time of deprecation, in effect leaving the old version untested for five years).

Table 7: Affected files for unused services (Type-I TD items).

Type	N	$\sum First$	$\sum Last$	$\sum Add$	$\sum Del$
javafiles	121	505	598	105	-8
int.test	120	2383	3428	1313	-268
Sum		2888	4026	1418	-276

Based on data from 21 of the operations that were not part of any other tests, we found that on average, a service version occurred in 4.4 Java files and 3.6 integration test files. This can be used to estimate the $EXPECTED\_USAGE$, using the values from table 7.

$$EXPECTED\_USAGE_{java} = 4.4 * 121 = 532 \text{ files}$$

$$EXPECTED\_USAGE_{int.test} = 3.6 * 120 = 432 \text{ files}$$

Based on these values, we find that out of the removed operations, there was little, if any, excessive usage amongst the Java files. However, there had been some growth due to interest (93 additional files being affected). The majority of excessive usage can be attributed to the integration tests, which also contributed the most of the additional interest.

The removal of the 121 service versions was carried out as a low-priority task during several months by three developers, with between 4 to 25 years of experience in the industry. The time spent (based on Git logs and estimations by the developers, as this time was not separately time reported) was 160 hours. Part of this time was due to back-porting the changes to an older, but still alive, branch, which proved to be almost as costly as doing the change in the original branch. Given this, it is reasonable to estimate that, had there been no additional growth in technical debt (such as the one during 2016 illustrated in figure 4), the removal would have taken between 0% (in case all file changes could have been automated, e.g. via scripts), and 29% ($1 - 2888/4026$) less time, i.e. 114 hours rather than 160. In practice, the truth is somewhere in between, as some changes could be highly automated, and some required more manual actions. These kinds of tasks are also highly dependent on the skill of the developer doing the change, in particular as the plain-text-based XML language lacks IDE support for refactoring operations such as renaming methods. All three developers were well versed in the Git version control system, scripting tools, and regular expressions, and used them, together with the test base and the Continuous Integration environment to verify system behavior before and after each change.

4.4 Commit counts

It could be argued, from a theoretical standpoint, that a file that is untouched and never read has no technical debt, even if it contains TD-items (in our case deprecated

services). In figure 9 we visualize the activity on the top six deprecated service versions, starting from the quarter following deprecation, plotting how many commits per quarter affect files where these services are used.

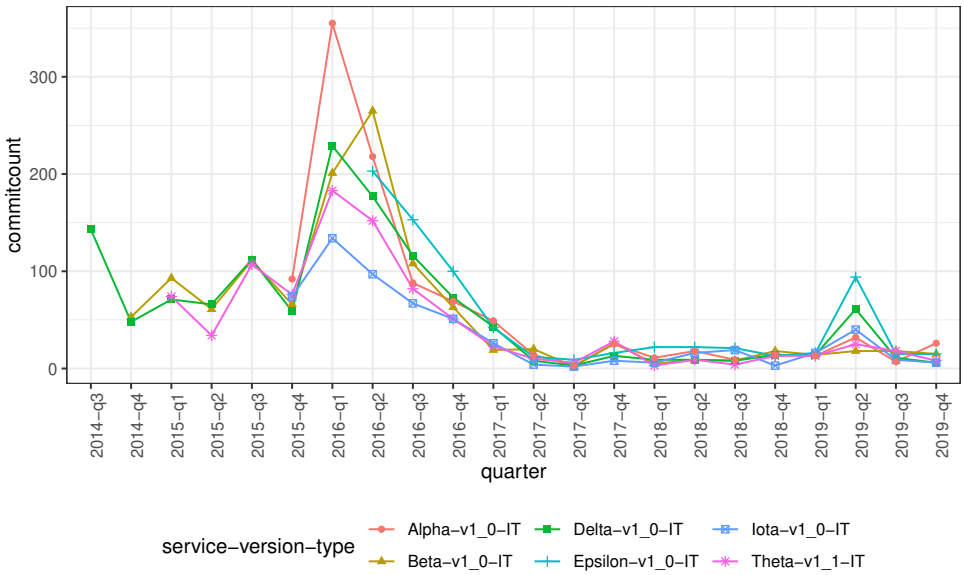


Figure 9: Count of commits affecting files using the top six deprecated service versions across each quarter

We see the many commits during late 2015 and 2016, where most of the commits were introducing the deprecated services into the test base. Then, a low period during 2017 and 2018, followed by high activity during Q2 2019, where many commits were touching files related to the top deprecated services. During this quarter, two new requirements were developed, adding configurability where previously more rigid business models had been used. In total, 78% of the commits for the ten most used deprecated services during Q2 2019 were related to these two requirements. Thus, figure 9 illustrates that for almost two years, there was not much movement in the files related to the top deprecated services. Then, during early 2019, two requirements were to be implemented, which caused much effort in maintaining these files (i.e., interest that could have been avoided if these deprecated versions would have been removed). This illustrates the highly non-linear nature of technical debt management, making it even more important to visualize it so that efforts to mitigate it can be estimated appropriately.

4.5 Discussion

Regarding **RQ 1**, whether the rule to avoid updating existing tests contributed to the spread of Technical Debt, we experienced an uneven distribution of deprecation debt

among the deprecated services. Since a small number of highly used services caused a significant increase in debt in integration tests, we hypothesize that highly used services should be treated differently than others.

Regarding **RQ 2**, whether the XML-based integration tests and the Java-based files show any difference in technical debt, we experienced very little debt for unit test cases since the number of affected files remains small. The integration tests show large growth in debt, originating from a small amount of highly used services, as shown in figure 4 and figure 5. After 2016, we experience a slower growth of integration test cases in general, and the product appears to have been tested using other means (such as Java-based tests), with minimal usage of the integration test framework.

Regarding **RQ 3**, how the growth in deprecated service version usage has contributed to Technical Debt, we note that most of the growth in deprecated services usage occurred during a limited time, between late 2015 and mid-2016. Integration tests were much more affected than Java-based files. For the services removed in late 2019, the accrued interest in the integration tests was calculated to 44%, versus 18% for the Java-based files. We have identified considerable duplication among the integration tests, particularly the setup files. Undue duplication incurs technical debt, even if a particular service is not deprecated, but in this case if a service is deprecated but still used in integration tests might lead to unexpected tests results due to the combination between deprecated and non-deprecated services. Thus, the general higher frequency of copied code in the integration tests could explain why there were more duplicates of deprecated service versions.

On a positive side, we note some repayment of interest for both types of files, though the decrease does not weigh up the increase in debt.

Regarding **RQ 4**, about the likely cause of the spreading Technical Debt, we can identify at least three causes of the spreading of deprecation debt:

- Backward compatibility causes technical debt unless the protocol can accommodate the needed changes without breaking older clients. Depending on the strictness of validation in clients or intermediate systems, different amount of debt is incurred. In the studied system, the rigid protocol rules, and strict security requirements (schema validation in firewalls) contributed to the large number of deprecated service versions.

- By examining the integration tests, we could see a pattern of copied setup files. Even though the integration test engine had the possibility of using a “shared function repository” since Q4 2015, this feature was not widely used during early 2016. Since 2014, the level of duplication in the integration test base has been around 15% on a file-by-file basis, comprising approximately 5% of the codebase, as shown in figure 7.

- Regarding the period with the most growth, late 2015 to mid-2016, we could

not see direct causes due to the growth in personnel during this time, though indirect causes, such as stress on experienced developers, can not be ruled out. Feedback from three developers with knowledge of the situation unanimously report this time as highly challenging (meeting a customer deadline). This is consistent with what is reported by Kruchten et al. [137]: "The most likely cause [of Technical Debt] we have observed is schedule pressure".

To summarize, we state a hypothesis that if an appropriate visualization had been in place for the integration tests, it is likely that both the duplicated files and the added deprecation debt could have been avoided during the stressful times of 2015-6. This might have saved up to 29% of the removal effort in late 2019. It is important to note that the distribution of services is highly right-skewed, meaning that measures of central tendency such as the mean, median, and standard deviation values are not very helpful. Instead, "top-N", or "most-growing" lists should be used to keep track of the code.

5 Threats to validity

We discuss the threats to validity from four different angles: *construct validity*, *internal validity*, *external validity* and *reliability*, following guidelines outlined in [139].

Construct validity deals with whether the studied measures reflect what the researcher has in mind, and what is stated in the research questions.

In this study, we use version control data (source code and revision history information) to draw conclusions. A threat to validity to this approach is that "you only see what was built," which is a form of survivor bias — i.e., you might see what was built, but not how or why it was built that way. To increase construct validity, we presented our conclusions to five of the remaining developers, validating our assumptions about why certain solutions were used.

When using version control logs as a data source, an important aspect is to consider the branching pattern and whether or not the studied organization commonly used rebasing commits. The Git version control system allows authors to "squash" commits, which may have been performed by different authors, at different times, into one new commit, discarding the constituent commits. For the studied system, this was not an approved practice, as the organization valued to see each commit, as it was written and pushed to the central repository. Most of the development took place in a single "master" branch for the duration of the study. Features were developed in other branches and later introduced into the master branch, typically via the Git rebase function, which keeps a linear history by rewriting commits, preserving author information and commit dates.

Internal validity deals with whether there might be other, non-studied factors that could explain some of the findings. We have gathered quantitative data (Git logs,

usage statistics), but also the Git ways of working for the studied system, which allows us to study the phenomena (i.e., the effects of deprecation on TD and its spread). In addition, to validate our conclusions, increasing internal validity, we triangulated our data by providing our conclusions and feedback to five developers of the system, one of which was the author of much duplicates during 2015-2016, and asked about their opinions. This provided valuable insight into the conditions and the time pressure experienced by the development organization.

External validity concerns to what extent it is possible to generalize the findings, and to what extent the findings are of interest to other people outside of the investigated case.

This paper is about experiences from a particular system, with a particular toolset. We have tried to describe characteristics that might enable others to judge whether the findings are applicable for other systems, but we cannot claim generalizability across all possible systems or organizations.

Reliability concerns whether or not the data and analysis are dependent on the specific researchers. Most of the data in this report are collected from quantitative sources, such as Git logs, and processed and visualized using standard statistical tools. As such, there is little room for bias in the processing of the collected data.

Interpretation of the processed data runs the risk of introducing reliability threats. We strove to reduce reliability threats by frequent interactions with the studied organization, especially with the developers that had been part of the product development team during the whole studied period, and elicited feedback from five of them.

6 Conclusions

As illustrated in this paper, to combat deprecation debt, it helps to keep it visible. Many IDEs today show deprecated classes in a different font (e.g., strikethrough), and this could be one reason why the unit tests do not show the same growth as the integration tests.

Another finding is the uneven distribution of the contribution to the debt. For the majority of services, the decision not to update deprecated usages in test cases did not spread any technical debt. However, the decision not to update some highly used "core services" caused up to 29% of increased effort in converting the test base at service removal. The addition of this debt mostly occurred during three quarters, between Q4 2015 and Q2 2016, a time that was identified as particularly stressful, which aligns with the findings of Kruchten et al. [137].

After the downsizing of the product, the integration tests were mostly untouched, as the new developers valued other test principles, such as Java-based testing (relegating much of the prior testing to unit tests). This suggests that as the test base grows, so does the importance of the IDE support (refactorings, static code analysis, duplication detection).

The usage statistics turned out to be a valuable tool to identify unused service versions (Type-I TD-items), supporting the removal of these.

Paper III

Governing the Commons: Code Ownership and Code-Clones in Large-Scale Software Development

Anders Sundelin, Javier Gonzalez-Huerta, Richard Torkar, and Krzysztof Wnuk.
In: Empir. Software Eng. 30, 43 (2025). https://doi.org/10.1007/s10664-024-10598-7

Abstract

Context: In software development organizations employing weak or collective ownership, different teams are allowed and expected to autonomously perform changes in various components. This creates diversity both in the knowledge of, and in the responsibility for, individual components.
Objective: Our objective is to understand how and why different teams introduce technical debt in the form of code clones as they change different components.
Method: We collected data about change size and clone introductions made by ten teams in eight components which was part of a large industrial software system. We then designed a Multi-Level Generalized Linear Model (MLGLM), to illustrate the teams' differing behavior. Finally, we discussed the results with three development teams, plus line manager and the architect team, evaluating whether the model inferences aligned with what they expected. Responses were recorded and thematically coded.
Results: The results show that teams do behave differently in different components, and the feedback from the teams indicates that this method of illustrating team behavior can be useful as a complement to traditional summary statistics of ownership.
Conclusions: We find that our model-based approach produces useful visualizations of team introductions of code clones as they change different components. Practitioners stated that the visualizations gave them insights that were useful, and by comparing with an average team, inter-team comparisons can be avoided. Thus, this has the potential to be a

useful feedback tool for teams in software development organizations that employ weak or collective ownership.

1 Introduction

Code clone detection and management has a long history in Software Engineering research [140–144]. Though copy-pasted code initially functions like the original, issues may emerge as the code evolves, with clones diverging as they change at different rates, particularly when developers are unaware of them. Tornhill [39] states that two code clones that often, but not always, change together can be the source of problems. In a study of five industrial and open-source systems, Juergens et al. [145] found that 52% of all clones were inconsistently changed, and 15% caused faults in the application. For this reason, code clones are seen as a prominent and quite common Technical Debt Item (TDI).

Technical Debt [120] (TD) is a metaphor borrowed from economics to explain the long-term consequences of sub-optimal design decisions taken to speed up the development process. Technical Debt is classified into different categories [146], and the term Technical Debt Item (TDI) refers to a single occurrence of technical debt in software artifacts [130, 137].

III

Software development organizations often divide large projects into components, managed by different teams, and can take varying approaches to code ownership, from strong, to weak, or collective ownership [147]. In weak or collective ownership, different teams are allowed and encouraged to contribute to the same components. Ribeiro et al. [148] have identified advantages of weak ownership, such as (i) knowledge distribution; (ii) increased backup pool of developers; (iii) lower rework; and (iv) better code quality, as well as disadvantages such as (i) increased conflict; (ii) increased errors and failures; (iii) lower understanding of the code; and (iv) increased development time when studying shared code ownership in industrial contexts.

Traditional code ownership involves teams or individuals being responsible for the quality and upkeep of software artifacts [149]. However, in practice, it is challenging to assign responsibilities to development teams, because multiple teams, with different "ownership stakes," frequently contribute to a single component. Having a handful of contributors might cause friction in the development process by introducing overhead or degrading the quality of the product or its source code [150]. For example, teams working in a component where they are not the main contributors might introduce more Technical Debt Items by being less careful with the code they produce, or they might introduce bugs due to their being unaware of the full consequences of their code changes. Therefore, the alignment between team expertise and tasks plays a critical role in the effectiveness and efficiency of the organization [151, 152]. Sedano et al. [153] emphasize that team code ownership is a feeling to be

engendered, not a policy to be decreed.

In an article widely cited in economics and ecology, Hardin [154] introduced the term "The Tragedy of the Commons", where he argued that common ownership in a land of finite resources will cause ruin and devastation for the whole population. Hardin argued in his essay for the centralization of the control of common-pool resources.

However, Dietz et al. [155] found five factors that support effective commons governance: (i) monitoring (including verification and understanding) of resources and human use of resources; (ii) moderate rate of change in resources, populations, technology, and social conditions; (iii) frequent face-to-face communication and dense social networks, which increase trust and allow people to experience the emotional reactions to distrust; (iv) the possibility of excluding outsiders at a relatively low cost; and (v) the users themselves support effective monitoring and rule enforcement (i.e., the users understand the purpose of the rules). We hypothesize that these factors apply to software code commons and that some of these factors can impact the mitigation of the "code tragedy", such as the accumulation of Technical Debt.

The goal of this paper is to develop and evaluate a model to proactively monitor how Technical Debt growth varies by contributor in a large-scale industrial system, and how this aligns with component ownership. We have built a model based on code clone introductions by teams, because clones are a common and easy TDI concept to grasp for most developers, with easy-to-grasp consequences. Contributions were grouped by team because, in the studied organization, teams worked independently and did not have strong ownership (e.g., no mandatory inter-team code reviews). We conducted a case study to collect data and build a Bayesian model that shows how code clones are introduced in components.

We created a Multi-Level Generalized Linear Model (MLGLM), based on a zero-inflated negative Binomial likelihood. The model is fit onto a dataset consisting of 31007 file changes made during 35 months in 8 code repositories by, in total, 10 development teams, belonging to an organization transitioning from collective to weak code ownership as it grew. Based on causal reasoning, the model estimates the expected number of introduced code clones for a given change by a given team in a given repository. The estimate is then used to visualize how teams behave in different repositories, and how they react to predictors such as existing complexity of, the number of existing duplicates in the changed file, or the size of the change.

To validate the reliability and usefulness of the model, we presented the main findings of the study to four of the studied teams in five focus-group sessions. All teams agree that the model predictions and associated visualizations present a useful view of how teams have behaved, and they presented plausible reasons for why some teams "stood out" in certain repositories.

The paper is structured as follows: Section 2 describes the background and related work, including the aforementioned OCAM model. Section 3 describes the research methodology, including the design of the causal and statistical models. Sec-

tion 4 describes the results, both quantitative and qualitative. Section 5 discusses our findings, followed by threats to validity in Section 6 and conclusions in Section 7.

2 Background and Related Work

A key part of the Agile and XP programming principles [94, 156], the principle of collective code ownership have also been espoused by software craftsmanship authors [notably 6, 84].

Software engineering usually maps out individual ownership, based on metrics such as: (i) commits in a component, (ii) files authored or changed, or (iii) lines authored or removed. In a study at Microsoft, Bird et al. [157] defined ownership metrics based on the number of commits and examined the effect of these on software quality. They looked at two large software projects: Windows Vista and Windows 7, and explored whether the number of low-expertise developers and the proportion of ownership for the top owner have a relationship with both pre-release faults and post-release failures. They discovered that more minor contributions (a proxy for weak code ownership) result in more pre- and post-release failures. Other researchers at Microsoft replicated the study, looking in more detail at an intermediate level of granularity that lies between binaries and source files: code directories [158]. They also broadened the metrics describing code ownership to individual ownership for files, directories, and organizational ownership for files and directories. Their results also confirmed that code ownership correlates with code quality.

Several researchers have looked into code authorship and ownership in open-source software projects. Avelino et al. [159] analyzed code authorship in 119 open-source projects, including the Linux kernel, and concluded that: (a) only a small portion of developers (26%) makes significant contributions to the code base—this ratio is almost constant during the Linux kernel evolution; (b) the number of files per author is highly skewed——a small group of top authors (2%) is responsible for hundreds of files, while most authors (75%) are responsible for at most 10 files; (c) most authors in Linux ($\approx 76\%$) are specialists, and the ratio between specialists and generalists tends to be constant; (d) authors with a high number of co-authorship connections tend to work with authors with fewer connections. Similarly, Foucault et al. [160] investigated the relationship between Bird's code ownership metrics and software quality for seven open-source projects. They confirmed the existence of a relationship between code ownership and software quality, but the relative importance of Bird's ownership metrics in multiple linear regression models is low compared to metrics such as the number of lines of code, the number of modifications performed over the last release, or the number of developers of a module.

Maruping et al. [161] examined the role of team collective ownership and coding standards by surveying 73 software project teams belonging to a large US software development firm. The authors analyzed 56 responses, comprising 509 software de-

velopers, and discovered that collective ownership and coding standards play a role in improving software project technical quality.

Faragó et al. [162] investigated the impact of code ownership, defined as the geometric mean of the number of authors of files in a commit, on maintainability for four open-source projects and discovered that code erosion is higher for source files modified by several developers in the past, compared to the files with clear ownership. They concluded that files changed by more authors are more error-prone than those developed by fewer developers.

Orrú et al. [163] investigated the relationship between code ownership and refactoring activities in the Apache Ant software system. Using Bird's notion of ownership as a ratio of changes performed by a single developer versus the total number of changes to a file, they derive two new metrics: Subjective Ownership and Relational Ownership. They concluded that refactoring activities were positively correlated with both Subjective and Relational Ownership, although the relation is weaker in the Relational case.

Borg et al. [164] analyzed 40 proprietary software repositories to understand the relationship between ownership, code quality, and issue resolution time. They discovered that in low-quality source code, marginal owners need 45% more time for small changes and 93% more time for large changes. Marginal owners are particularly hampered when working with low-quality source code, which leads to productivity losses.

On the qualitative research front, Ribeiro et al. [148] conducted an interview study with 19 participants to investigate the advantages and disadvantages of using shared code. They found six advantages and six disadvantages of using shared code ownership, including improved knowledge distribution, increased conflicts between the team members, and increased development time.

The Ownership and Contribution Model (OCAM) was formulated by Zabardast et al. [165], and ranks teams using seven metrics to determine the alignment between contribution and ownership for a particular component. The number of applied metrics is flexible, as is the granularity of contributors and components. The authors validate the model in a longitudinal industrial case study in the paper. In a follow-up study, Zabardast et al. [166] found that the relationship between ownership and contribution alignment was identified by the focus group participants as one of the potential causes of faster accumulation of technical debt.

In another study at Microsoft, Herzig et al. [167] found that test suites whose owners are distributed across organization subgroups with long communication paths are negatively correlated with quality. They recommended reviewing test suites concerning their organizational composition and favored subgroups having clear ownership of test suites.

In addition to the studies mentioned in the introduction, several recent authors also studied code clones detection and management. Ain et al. [168] conducted a systematic literature review (SLR) of code clone detection tools, identifying 13 distinct

tools in 54 selected papers, as well as 13 proposed future tools. Quradaa et al. [169] surveyed the literature about the usage of recurrent neural networks in code clone detection, and concluded that LSTM techniques are the most commonly used. Kaur et al. [170] surveyed the use of machine learning in code clone detection, reporting that Decision Tree, Random Forest, Bayesian Network, and Naïve Bayes are the most popular machine learning algorithms, and that Deep Learning can be used to detect semantic clones. Rongrong et al. [171] used different classifiers to recommend clones for refactoring, and concluded that Decision Trees and Bayesian Networks achieved the highest accuracy in recommending clones for refactoring, but Decision Tree was more stable. Zakeri-Nasrabadi et al. [172] conducted another SLR on clone detection, and identified 136 primary studies, referring to 80 distinct tools, of which almost half support Java, and more than a third support C and C++.

We identified two literature reviews about code clone evolution. Pate et al. [143] conducted an SLR on code clone evolution and found wide variation in the ratio of consistent changes to clones—between 11% and 74% of clones were found to be changed consistently. They concluded that researchers need to study human behavior, together with available data, to understand the evolution of code clones. Zhong et al. [173] surveyed the literature on code clone evolution and found few studies on the visualization of clone evolution.

Yu et al. [174] studied the locations of code clones in two versions of the Linux kernel. They found that clones mostly occurred between files close in the hierarchy, and found that the number of clones grew with the size of the kernel. Although there are valid cases for duplicating code—in particular, if it changes for different reasons [see 7, 175]—we follow the taxonomy of Alves et al. [146] and place code clones in the *Code Technical Debt* category, as several studies, such as Yu et al. [174] have found that the majority of code clones arise due to simple copy-paste behavior.

Compared to the related work, we take a different approach. Although code may be authored and committed by individuals, in proprietary software development, they tend to be organized in teams (that often have significant authority and freedom), and we hypothesize that team culture might have an impact on how developers approach code quality. Therefore, we map these individuals to the team they belong to at the time of authoring (or committing). These teams are then used as categories to build a regression model, which can be used to simulate and visualize how teams introduce code clones in different components. When presented with the results, both architects and development teams stress that they think the model for clone introductions is useful to illustrate and explain their behavior.

However, when modeling other organizations or software development networks, other grouping factors might be used—regardless of *how* the categories are conceived, the Bayesian model will use them when deriving the linear regression. Different organizations may warrant other grouping levels, such as individual authors, organizational sub-units, or different geographical sites or countries.

3 Research Methodology

The research presented in this paper follows a two-staged process, where we start with designing a causal model of how teams introduce clones in components, depending on their level of knowledge and how much they care for these components. We then proceeded by using our causal model to guide us in collecting empirical data, following a case study research method. We used two different sources of data: i) quantitative, for which we use archival analysis as data collection method to mine the software repositories of the studied organization to train models that predict the introduction of code clones in the studied software repositories; and ii) qualitative, for which we use focus group interviews as a data collection method and open coding to analyze the data to validate the reliability and usefulness of the chosen model.

The remaining of the section is structured as follows: Subsection 3.1 provides the details on how the model was iteratively built, while Subsection 3.2 reports on the empirical validation of the model predictions.

3.1 Model Design

3.1.1 Research Questions

The motivation for our study is to explore how collective ownership of source code impacts its quality, in particular, the introduction of code clones. This led us to formulate the following research questions:

RQ1 Can we build a generalized linear model with acceptable accuracy and reliability to visualize team behavior regarding the introduction of code clones in different components?

RQ2 What predictors are most likely to affect the rate of clone introduction, and how do these vary between teams and components?

RQ3 Are the model predictions and their associated visualizations perceived as useful for the studied organization?

For RQ1, we chose to design a Multi-Level Generalized Linear (MLGLM) model, predicting the number of introduced code clones in a file being changed, given various predictors. We chose to focus on this particular class of Technical Debt Item (TDI), as we wanted a model that could predict team behavior without biasing how different classes of TDI should be weighted in the overall technical debt score. Code clones are readily detected by tools such as SonarQube [176], and are also directly connected to the clean–code principle *DRY - Don't Repeat Yourself* [6, 43], that have been shown to be an important indicator of technical debt by experienced developers [177].

For RQ2, we designed several models and compared their out-of-sample prediction capabilities using the LOO-CV metric [178].

Table 1: Used OCAM model metrics.

Metric	Description
N	Number of commits
CHURN	Sum of `max(added, removed)` LOC
ADD_COMP	Sum of added McCabe complexity
REM_COMP	Sum of removed McCabe complexity

For RQ3, we conducted five focus group interviews with four of the studied teams—one meeting each with three development teams, plus opening and closing presentations with the architect team. The meetings were recorded and transcribed; anonymized transcriptions are available upon request. We used open coding to summarize the findings of the teams. Based on team feedback, we also designed a follow-up model building of clone removal behavior.

3.1.2 Measuring the Degree of Ownership Alignment

OCAM was originally defined as a flexible model to assess the degree of alignment between *formal ownership* and team contribution to a given repository [165]. The OCAM model allows metrics and organizational granularity to be configured according to specific needs. As the studied organization was large and some developers switched teams, we decided to keep the team granularity from the original OCAM model.

Table 1 contains descriptions of the OCAM metrics that we collected. We kept the following original OCAM metrics: (i) the number of commits; (ii) code churn—defined as the sum of max(added, removed) lines for each changed file; (iii) added complexity (McCabe metric); (iv) removed complexity. We had to disregard the metrics regarding tickets and pull requests, since the studied organization did not keep track of these metrics for changes originating between teams.

We used the OCAM model in the second stage of our research, comparing and contrasting its results with the results from our model, as perceived by the development teams and architects.

3.1.3 Predicting Code Clone Introduction

In the design of the statistical model for clone introduction, we follow the method outlined by McElreath [27], which starts with stating your assumptions and the domain knowledge in the form of a Directed Acyclic Graph (DAG) [23, 26].

Figure 1 shows a causal DAG containing the variables and their dependencies in our model of clone introductions. Two variables are unknown (and therefore unmeasured) and are marked as latent—`CLEAN` and `KNOW`. In this DAG,[1] every change to each source code file is modeled according to the following variables:

[1] DAG motivation is available at `https://docs.google.com/spreadsheets/d/1vV6utjEEgmELib9ihplflOjhjVLPDXvHWbUG49cb-qQ`

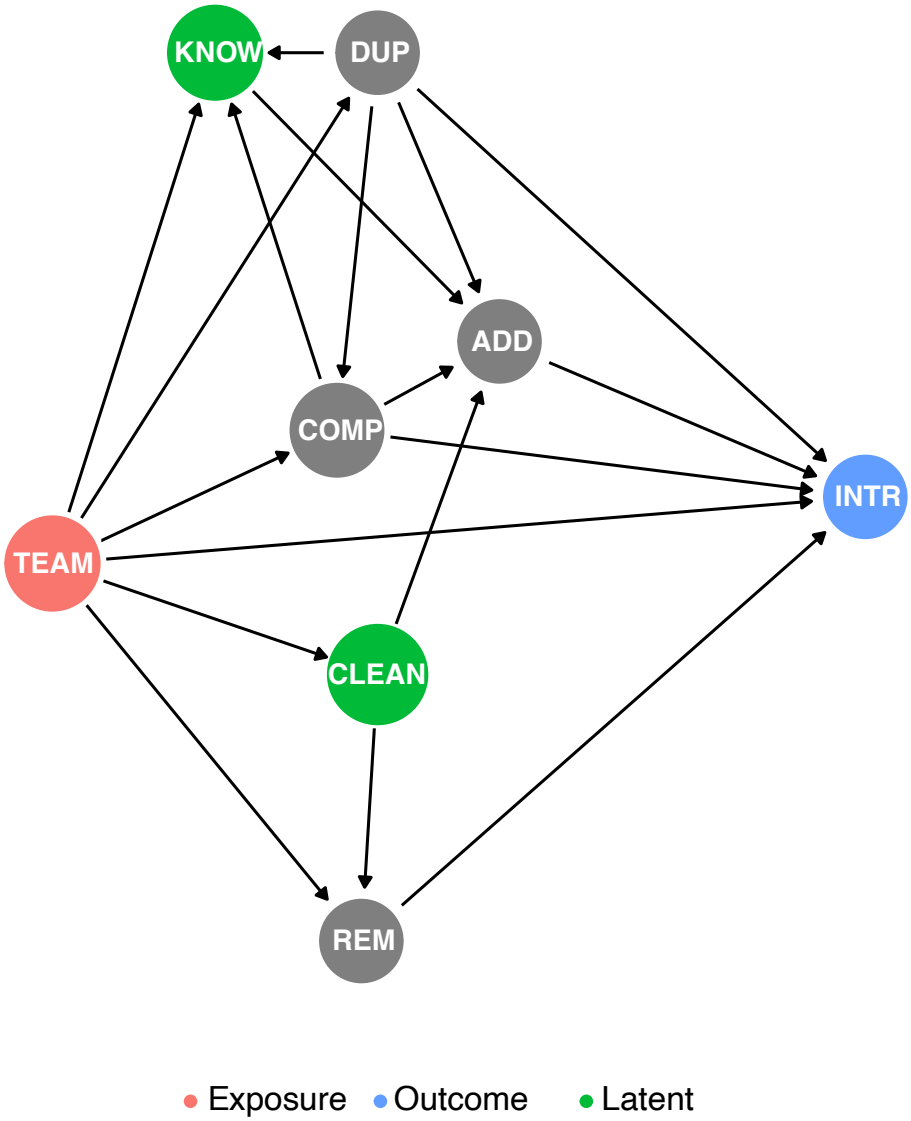


Figure 1: Assumed causal model of code clone introduction.

INTR (outcome) measures how many duplicates were introduced by the change to the file. This is the outcome variable of the model. The lower bound is zero, and the upper limit is bounded by the size of the changed file (though the duplicate detection tool will consider several adjacent identical lines a single duplicate).

TEAM (exposure) indicates the team that committed the change to the master branch, as a categorical predictor. Furthermore, we assume that behavior will vary between teams and that a single team will behave differently in different components (repositories). We map the committer to the team at the time of the commit, based on sampled organization charts from a team wiki page, which contains current and historical team compositions.

ADD (adjustment set) measures the number of lines that were added to the file in this particular change. The lower bound is zero, and there is no theoretical upper bound (although in practice, the used compiler will impose some fixed limit).

REM (adjustment set) measures the number of lines that were removed from the file in this particular change. The lower bound is zero, and the upper bound is limited by the size of the file before the change.

DUP (adjustment set) measures the number of existing duplicates in the file before the change, as measured by SonarQube. The lower bound is zero, and the theoretical upper bound is limited by the size of the changed file (based on how the tool calculates duplicates).

COMP (adjustment set) measures the complexity of the file according to McCabe [179], as measured by SonarQube, after the change. The lower bound is zero, and there is no theoretical upper bound.

`KNOW` is a latent (unmeasured and unknown) metric, representing the knowledge the team committing the change has in the file, and in the component to which the changed file belongs. As complex files—possibly with many existing duplicates—are likely to be harder to comprehend, the prior knowledge is affected by the DUP and COMP metrics. Knowledge affects the number of added lines, as we posit that developers make more copy-paste changes in components where they lack domain knowledge.

`CLEAN` is a latent (unmeasured and unknown) metric, representing the tendency of the team committing the change to clean up the code they touch or come across when they make changes. A team being formally responsible for the quality of—or with a high ownership stake in—a given component would be expected to clean up more (e.g., via refactorings) than a sporadically contributing team.

The main causal effect we want to study is how team knowledge and cleanliness affects the number of duplicates introduced for a given number of added lines. As we cannot measure these metrics objectively, we instead measure how the team and repository combination affect the influence that the other predictors (i.e., the number of duplicates in the file, the complexity, and the number of lines added or removed) have on the amount of duplicates being introduced.

The specified DAG implies the following conditional independencies: $\mathrm{REM} \perp\!\!\!\perp \mathrm{COMP} \mid \mathrm{TEAM}$ and $\mathrm{REM} \perp\!\!\!\perp \mathrm{DUP} \mid \mathrm{TEAM}$ ($\perp\!\!\!\perp$ is the symbol for independence, and $\mid$ is the symbol for 'given').

These independencies can be tested from the collected data, and while the number of removed lines does seem to be independent of the existing complexity, there might be a weak association between REM and DUP ($\mu = 0.13, 95\%\,\mathrm{CI} = [0.11; 0.14]$). This might indicate that (some) teams are more likely to remove lines in files with many duplicates. Thus, we do not rule out a causal path between DUP and REM. However, the existence of this path would not change our model or conclusions since, most likely, this path might only influence how the duplicates are removed, i.e., a file with many duplicates might receive changes that mainly focus on removing lines of code, or at least modifying them, aiming at removing duplicates.

3.1.4 Data Collection

We used Algorithm 2 to calculate the required metrics for each changed file, in each commit, per repository. List C contains each commit, in order of application to the master branch for each studied repository. Commit data is retrieved via the functions `AuthoredBy`(C), `AuthoredDate`(C), `CommittedBy`(C) and `CommittedDate`(C). In the git version control system, the `author` is normally the person initiating the change, and the `committer` is the person triggering the commit which merges the change to the main branch.

We used historical organization charts to determine the developer–team affiliation at a given time. This information is returned by the `FindTeam(A, D)` function. `Files`(C) are the files changed in commit C. `Added`($F_{j,i}$) and `Removed`($F_{j,i}$) are the number of added and removed lines of file F_j, in commit C_i, as counted by the git command. `Complex`(F) and `Duplicates`(F) are the McCabe complexity and number of duplicated code blocks, as measured by the SonarQube tool. We use the $F_{j,i-1}$ notation to denote the file state in commit C_{i-1}, regardless of whether the file changed in that commit or not. The number of introduced duplicated code blocks is given by Δ, in case it is positive. The complete anonymized data set is available in the replication package [180].

3.1.5 Simulation and Initial Model Design

Based on our prior domain knowledge, and supported by empirical findings such as Zabardast et al. [109], who found that most file changes neither introduced nor

Algorithm 2: Calculating introduced clones for changed files in a repository.

Data: C commits: C_i precedes C_{i+1}
Result: List of data for changed files
filestate $\leftarrow$ ()
for $C_i \in C$ **do**
  author $\leftarrow$ `AuthoredBy`(C_i)
  $\mathsf{d}_{auth} \leftarrow$ `AuthoredDate`(C_i)
  committer $\leftarrow$ `CommittedBy`(C_i)
  $\mathsf{d}_{comm} \leftarrow$ `CommittedDate`(C_i)
  $\mathsf{team}_a \leftarrow$ `FindTeam`(author, d_{auth})
  $\mathsf{team}_c \leftarrow$ `FindTeam`(committer, d_{comm})
  for $F_{j,i} \in$ `Files`(C_i) **do**
    add $\leftarrow$ `Added`$(F_{j,i})$
    rem $\leftarrow$ `Removed`$(F_{j,i})$
    comp $\leftarrow$ `Complexity`$(F_{j,i})$
    $\mathsf{dup}_{prev} \leftarrow$ `Duplicates`$(F_{j,i-i})$
    $\mathsf{dup}_{curr} \leftarrow$ `Duplicates`$(F_{j,i})$
    $\Delta \leftarrow \mathsf{dup}_{curr} - \mathsf{dup}_{prev}$
    if $\Delta > 0$ **then** introd $\leftarrow \Delta$
    else introd $\leftarrow 0$
    filestate $\leftarrow$
     filestate $+ (F_{j,i}, \mathsf{team}_a, \mathsf{team}_c, \mathsf{add}, \mathsf{rem}, \mathsf{comp}, \mathsf{dup}_{prev}, \mathsf{introd})$
  end
end
return filestate

removed Technical Debt Items, we assumed that most file changes would not introduce any additional duplicates. As we were modeling a count (≥ 0) of added duplicates, our initial model used the Zero-Inflated Poisson distribution, which is the maximum entropy distribution for counting independent events, having a known expected value.

However, after simulating and validating with collected data from the first repository, we found the constraints of the Poisson distribution, i.e., equal variance and expected value ($\sigma^2 = \lambda$), to be unsuitable for our domain. The conventional next choice of model is the Negative-Binomial distribution, which is parameterized via two parameters: μ, the expected value (mean), and ϕ, the shape parameter that adjusts the variance via the formula $\sigma^2 = \mu + \frac{\mu^2}{\phi}$. As recommended by McElreath [27], we used a logarithmic link function, $\log(\mu) = \beta_0 + \sum_i \beta_i P_i$ to tie our linear predictors to the parameters of the Negative-Binomial. The zero-inflation part is realized

via a Bernoulli trial, with the probability of 'success' (zero-inflation active) tied to the linear predictor via a logit (log-odds) link function, $\text{logit}(p) = \log(\frac{p}{1-p})$.

Following the recommendations by McElreath [27] and Gelman,[2] we transformed our data to fit within the useful range of the link functions. In theory, the logit function is unbounded, but in practice, the useful values lie within ± 10, as $\text{logit}(4.5 \cdot 10^{-5}) \approx -10$ and $\text{logit}(0.99995) \approx 10$. A similar argument can be made for the log link function, which causes the mean value of the Negative-Binomial likelihood to grow with the exponent of the linear predictors.

Based on our causal assumptions (Fig. 1), besides the team and repository categorical predictors, we considered adding four numerical predictors: added lines (ADD), removed lines (REM), existing duplicates (DUP) and existing complexity (COMP). Following the DAG, all numerical predictors were bounded by zero, without any obvious upper bound. Furthermore, exploratory analysis of collected data showed that all four predictors were right-skewed,[3] which caused us to base the linear model based on the logarithm of the predictor value plus one (as $\log(0+1) = 0$). We scaled and centered the resulting logarithm by subtracting the mean and dividing by the standard deviation of the logarithm, to aid prior selection and improve model fitting, as suggested in [27]. This means that a change of one unit in the scaled predictor corresponds to a change in the magnitude of the unscaled predictor equal to the observed standard deviation of the magnitude.

This leaves us with four potential numerical predictors:

A is the scaled and centered natural logarithm of the number of added lines (ADD).

R is the scaled and centered natural logarithm of the number of removed lines (REM).

C is the scaled and centered natural logarithm of the existing McCabe complexity of the changed file, as calculated by SonarQube (COMPLEX).

D is the scaled and centered natural logarithm of the existing number of duplicates in a changed file, as calculated by SonarQube (DUP).

Depending on the number of predictors we include, we will get models of varying complexity.

A crucial step in model design is conducting prior predictive checks [27]. For Bayesian inference to work efficiently, the selected priors should be wide enough to allow for reasonable outcomes, but not too wide, as this would force the model to explore outcomes not consistent with domain-specific knowledge. In our context,

[2] `https://statmodeling.stat.columbia.edu/2019/08/21/you-should-usually-log-transform-your-positive-data/`

[3] Most changes add a few lines of code, remove a few lines of code, and the complexity of the files is in general low, with fewer data points with high levels for any of these variables.

we know, based on the design of the duplicate detection tool,[4] that the number of introduced duplicates must be less than the number of existing lines in a file. Thus, regardless of the parameter values fed to the model, we would find outcomes ranging in the tens of thousands to be implausible.

Both domain knowledge simulation and prior predictive checks were conducted before actual data collection, and both are available in the replication package [180].

3.1.6 Model Design and Quality Assessment

We built three models of increasing complexity to compare and assess their results. We used the R programming language[5] and the framework BRMS [181, 182] as a front-end for the Hamiltonian Monte Carlo (HMC) framework provided by Stan [183].

3.1.7 Model Structure

All three models use a Zero-Inflated Negative-Binomial distribution, and differ in complexity only in the number of parameters used in the linear predictors.

$$p(y|\mu,\phi,\xi) = \begin{cases} \xi + (1-\xi)\cdot \mathrm{NB}(0|\mu,\phi) & \text{if } y = 0 \\ (1-\xi)\cdot \mathrm{NB}(y|\mu,\phi) & \text{if } y \neq 0 \end{cases} \tag{III.1}$$

A Zero-Inflated Negative-Binomial model is defined by Equation III.1, where NB is the Negative-Binomial distribution with mean (i.e., location) parameter μ and a shape parameter ϕ, which adjusts the variance via the formula $\sigma^2 = \mu + \frac{\mu^2}{\phi}$. The parameters y, μ, ϕ, and ξ can either be modeled as population-level parameters (i.e., the same distribution for all values in the input data set), or may depend on observation i.

We will adopt the notation favored by McElreath [27], where an i suffix indicates that the parameter is associated with observation i from the observed data. That is: μ_i means the μ value for observation i; $\beta_{0,\tau[i]}$ is the intercept parameter offset associated with the team in observation i; σ_τ means the standard deviation for team τ; and $\sigma_{\tau:\chi}$ is the standard deviation associated with team τ in component (repository) χ.

In all our models, we leave the shape parameter, ϕ, independent of observations.

3.1.8 Intercept-Only Model $\mathcal{M}_0$

The first, simplest model $\mathcal{M}_0$, ignores all predictors except the team and repository categories. This is a common choice as a baseline model, for comparison with models of higher complexity. It is specified as:

$\mathcal{M}_0$

$$\begin{aligned} \log(\mu_i) &= \beta_{0,i} \\ \mathrm{logit}(\xi_i) &= \gamma_{0,i} \end{aligned} \tag{III.2}$$

[4] SonarQube, `https://www.sonarsource.com/products/sonarqube`

[5] `https://www.r-project.org/`

where

$$\begin{aligned}\beta_{0,i} &= \beta_0 + \beta_{0,\tau[i]} + \beta_{0,\tau:\chi[i]} \\ \gamma_{0,i} &= \gamma_0 + \gamma_{0,\tau[i]} + \gamma_{0,\tau:\chi[i]}\end{aligned} \tag{III.3}$$

with priors

$$\begin{aligned}\beta_0 &\sim \text{Normal}(0, 0.5) \\ \sigma_\tau &\sim \text{Weibull}(2, 0.25) \\ \sigma_{\tau:\chi} &\sim \text{Weibull}(2, 0.25) \\ \phi &\sim \text{Gamma}(0.5, 0.1)\end{aligned} \tag{III.4}$$

Model $\mathcal{M}_0$ assumes that the logarithm of the mean (μ_i) of the Negative-Binomial for observation i is composed of one population-level parameter β_0, plus two adjustments to this value; $\beta_{0,\tau[i]}$, adjusting the intercept per team (regardless of the repository), and $\beta_{0,\tau:\chi[i]}$, additionally adjusting the intercept depending on the team–repository combination.

The same structure is used for the probability (ξ_i) of seeing inflated zeros; in this case, γ_0 represents the population-level intercept, adjusted by $\gamma_{0,\tau[i]}$ and $\gamma_{0,\tau:\chi[i]}$. The model assumes that the shape parameter is independent of observations.

3.1.9 A More Complex Model $\mathcal{M}_1$

Following our DAG (Fig. 1), we posit that the causal model of introduced clones would at least include the added and removed lines predictors, whose scaled magnitude is represented by A and R, with team- and repository-level variation only on the intercept and the removed lines (R) predictor.

Our second model then becomes:

$\mathcal{M}_1$

$$\begin{aligned}\log(\mu_i) &= \beta_{0,i} + \beta_A A_i + \beta_{R,i} R_i \\ \text{logit}(\xi_i) &= \gamma_{0,i} + \gamma_A A_i + \gamma_{R,i} R_i\end{aligned} \tag{III.5}$$

Here, the intercept ($\beta_{0,i}$) and the slope for the R predictor ($\beta_{R,i}$) are composed of three components, incorporating the team- and team-per-repository-level differences:

$$\begin{aligned}\beta_{0,i} &= \beta_0 + \beta_{0,\tau[i]} + \beta_{0,\tau:\chi[i]} \\ \beta_{R,i} &= \beta_R + \beta_{R,\tau[i]} + \beta_{R,\tau:\chi[i]}\end{aligned} \tag{III.6}$$

In contrast, the slope β_A of the added lines predictor is assumed to be constant between teams and repositories, meaning that as the number of added lines grow, the rate of introducing duplicates increases by the same amount for all teams and repositories, if the other predictors are kept constant. However, as the model allows the intercepts to vary between teams and repositories, each team might still have a different base rate of clone introduction in each repository.

As we let both the slope and the intercepts vary, we need to incorporate prior information for the correlation between these parameters. The offsets per team ($\beta_{R,\tau[i]}$) and team–repository ($\beta_{R,\tau:\chi[i]}$) are modeled as multi-variate-normal (MVN) models, with a zero mean—because they are offsets—and standard deviation matrices ($\Sigma_{R,\tau}$, $\Sigma_{R,\tau:\chi}$) decomposed by diagonal matrices and correlation matrices ($\Omega_{R,\tau}$, $\Omega_{R,\tau:\chi}$). We use the recommended Lewandowski-Kurowicka-Joe distribution for the correlation matrices; our choice of the $\mathrm{LKJ}(2)$ prior ensures that our model is mildly skeptical of extreme correlations between intercepts and slopes.

This means that our priors are defined as:

$$
\begin{aligned}
\beta_0 &\sim \text{Normal}(0, 0.5) \\
\beta_A &\sim \text{Normal}(0, 0.25) \\
\beta_R &\sim \text{Normal}(0, 0.25) \\
\beta_{R,\tau[i]} &\sim \text{MVN}(0, \Sigma_{R,\tau}) \\
\Sigma_{R,\tau} &\sim \text{diag}(\sigma_\tau)\Omega_\tau\text{diag}(\sigma_\tau) \\
\sigma_\tau &\sim \text{Weibull}(2, 0.25) \\
\Omega_\tau &\sim \text{LKJ}(2) \\
\beta_{R,\tau:\chi[i]} &\sim \text{MVN}(0, \Sigma_{R,\tau:\chi}) \\
\Sigma_{R,\tau:\chi} &\sim \text{diag}(\sigma_{\tau:\chi})\Omega_{\tau:\chi}\text{diag}(\sigma_{\tau:\chi}) \\
\sigma_{\tau:\chi} &\sim \text{Weibull}(2, 0.25) \\
\Omega_{\tau:\chi} &\sim \text{LKJ}(2) \\
\phi &\sim \text{Gamma}(0.5, 0.1)
\end{aligned}
\tag{III.7}
$$

The zero-inflation predictors (γ) use identical structures and priors:

$$
\begin{aligned}
\gamma_{0,i} &= \gamma_0 + \gamma_{0,\tau[i]} + \gamma_{0,\tau:\chi[i]} \\
\gamma_{R,i} &= \gamma_R + \gamma_{R,\tau[i]} + \gamma_{R,\tau:\chi[i]}
\end{aligned}
\tag{III.8}
$$

3.1.10 Full Causal Model $\mathcal{M}_2$

Following the complete DAG in Fig. 1, we posit that the existing complexity and number of duplicates in a file impact the number of duplicates that a particular team introduces, especially if the team is unfamiliar with the component where the file resides.

The fact that the existing number of duplicates (DUP, part of code technical debt) in a file will impact the number of duplicates that developers introduce while changing the same file is consistent with the *Broken Window Theory* phenomenon, first described in a Software Engineering context by Hunt et al. [43] and validated by Levén et al. [184].

With this reasoning in mind, we decided to also incorporate the existing complexity and number of duplicates into our model:

$\mathcal{M}_2$

$$\begin{aligned}\log(\mu_i) &= \beta_{0,i} + \beta_A A_i + \sum_{P\in\{R,C,D\}} \beta_{P,i} P_i \\ \text{logit}(\xi_i) &= \gamma_{0,i} + \gamma_A A_i + \sum_{P\in\{R,C,D\}} \gamma_{P,i} P_i\end{aligned} \tag{III.9}$$

Compared with $\mathcal{M}_1$, our model now has two additional predictors (C and D), representing the scaled magnitude of the complexity (COMPLEX) and existing duplicates (DUP) in the file. The $\beta_{C,i}$ and $\beta_{D,i}$ coefficients, just like $\beta_{0,i}$ and $\beta_{R,i}$ in model $\mathcal{M}_1$, are composed of three components, as is $\gamma_{C,i}$ and $\gamma_{D,i}$.

$$\begin{aligned}\beta_{0,i} &= \beta_0 + \beta_{0,\tau[i]} + \beta_{0,\tau:\chi[i]} \\ \beta_{R,i} &= \beta_R + \beta_{R,\tau[i]} + \beta_{R,\tau:\chi[i]} \\ \beta_{C,i} &= \beta_C + \beta_{C,\tau[i]} + \beta_{C,\tau:\chi[i]} \\ \beta_{D,i} &= \beta_D + \beta_{D,\tau[i]} + \beta_{D,\tau:\chi[i]} \\ \gamma_{0,i} &= \gamma_0 + \gamma_{0,\tau[i]} + \gamma_{0,\tau:\chi[i]} \\ \gamma_{R,i} &= \gamma_R + \gamma_{R,\tau[i]} + \gamma_{R,\tau:\chi[i]} \\ \gamma_{C,i} &= \gamma_C + \gamma_{C,\tau[i]} + \gamma_{C,\tau:\chi[i]} \\ \gamma_{D,i} &= \gamma_D + \gamma_{D,\tau[i]} + \gamma_{D,\tau:\chi[i]}\end{aligned} \tag{III.10}$$

We used the same priors as in model $\mathcal{M}_1$, but also included C and D:

$$\begin{aligned}\beta_0 &\sim \text{Normal}(0, 0.5) \\ \forall P \in \{A,R,C,D\} : \beta_P &\sim \text{Normal}(0, 0.25) \\ \forall P \in \{R,C,D\} : \beta_{P,\tau[i]} &\sim \text{MVN}(0, \Sigma_{P,\tau}) \\ \forall P \in \{R,C,D\} : \Sigma_{P,\tau} &\sim \text{diag}(\sigma_\tau)\Omega_\tau\text{diag}(\sigma_\tau) \\ \sigma_\tau &\sim \text{Weibull}(2, 0.25) \\ \Omega_\tau &\sim \text{LKJ}(2) \\ \forall P \in \{R,C,D\} : \beta_{P,\tau:\chi[i]} &\sim \text{MVN}(0, \Sigma_{P,\tau:\chi}) \\ \forall P \in \{R,C,D\} : \Sigma_{P,\tau:\chi} &\sim \text{diag}(\sigma_{\tau:\chi})\Omega_{\tau:\chi}\text{diag}(\sigma_{\tau:\chi}) \\ \sigma_{\tau:\chi} &\sim \text{Weibull}(2, 0.25) \\ \Omega_{\tau:\chi} &\sim \text{LKJ}(2) \\ \phi &\sim \text{Gamma}(0.5, 0.1)\end{aligned} \tag{III.11}$$

As for model $\mathcal{M}_1$, we used identical structure and priors for the predictors for the zero-inflation component (γ).

We used visualizations such as the plot in Fig. 2 to aid prior selection for all our models. Gelman et al. [185] recommends avoiding bounded priors such as the uniform distribution, and instead use priors consistent with domain knowledge. In Fig. 2,

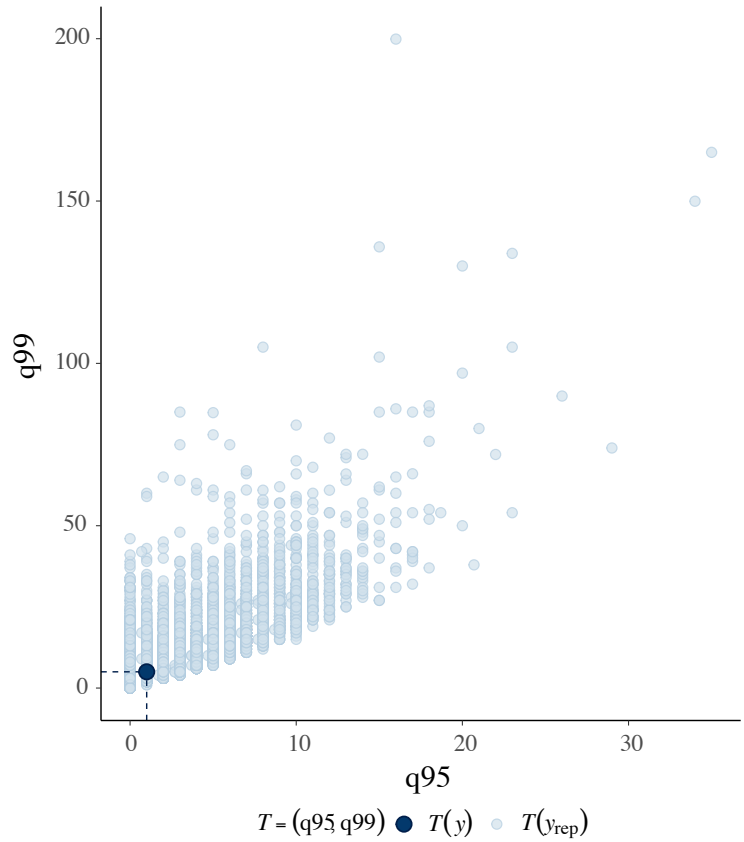


Figure 2: Prior predictive check plot of 95th vs. 99th percentile of introduced code clones. $T(y_{rep})$: Prior predictions. $T(y)$: Observations ($Q_{95} = 1, Q_{99} = 5$). Prior predictions are consistent with observations, and not unreasonable.

the x-axis depicts the 95th percentile and the y-axis depicts the 99th percentile of predictions made using only prior information, with predictor values used from our data. Based on the figure, we conclude that using these priors, the model would expect 95% of the observations to range between 0 and about 20 introduced duplicates (the observed count is 1, indicated by $T(y)$ in the figure). Likewise, the model would expect 99% of the observations to range between 0 and about 80 introduced duplicates, with the more likely outcomes being between 0–40.

We used the following criteria to select appropriate priors:

- Based on empirical findings [109] and experience, we would find it implausible that the majority of file changes would result in new duplicates. Our priors put the highest probability at about 80% zeros, but expect anything from 40% to 100% zeros (with smaller probabilities at the extremes).
- The maximum estimated value (i.e., the number of introduced clones the model would expect, based only on prior information) for these priors range from

≈ 100 to ≈ 20000 duplicates, with higher probabilities at the lower end of the scale, and very little probability over 1000.

- As shown in Fig. 2, the 95th percentile is expected to range from 0 to ≈ 20 and the 99th percentile is expected to range from about 0 to ≈ 80, though more extreme values are also possible, albeit less likely.

The replication package contains more prior predictive plots, including predictions on group-level data, such as teams and repositories.

Based on our exploratory data analysis, we found that our data contained an excess amount of zeros (around 93%), so traditional measures of central tendency (mean and median) would not be of much use. We opted instead to use Q95 and Q99 metric—that is the 95th and 99th percentile metrics.

After fitting the model, and performing standard sampling checks,[6] we assessed the model fit by approximating leave-one-out cross-validation (LOO-CV) using Pareto-smoothed importance sampling (PSIS), as recommended by Vehtari et al. [178]. Instead of doing full leave-one-out cross-validation, which would require refitting the model once for each data point, the LOO-CV algorithm calculates the leave-one-out posterior distribution using Pareto-smoothed importance sampling (PSIS) by approximating the importance of each data point on the final posterior distribution. The result is reported for each data point as the Pareto k-value. Data points with Pareto k-values higher than 0.7 are considered outliers and should be investigated further. In our case, model $\mathcal{M}_2$ contained 12 data points with Pareto k values exceeding 0.7. Using the reloo function,[7] the model was refit once per suspicious data point, concluding that the real Pareto k values for all data points were below the recommended threshold. This completed our diagnostics of the models.

However, a well-diagnosed model might still produce bad inferences, in case it does not fit the data well. The conventional tool to assess model fit is to plot and quantify residuals (the difference between predicted and observed values) but for zero-inflated models like ours, the recommended way is instead to use the rootogram method [186]. In a rootogram, the y-axis contains the square root of the expected frequency and the x-axis contains the predicted outcome count. In a suspended rootogram, the difference between the expected and actual frequency is shown as the light-blue *Observed* histogram, while the *Expected* line (and surrounding credible interval) contains the expected frequency.

Figure 3 contains two subplots of rootograms in the suspended style, for model $\mathcal{M}_2$. Subplot **a)** contains predictions up to $y = 150$, and all of the predicted zeros (as $\sqrt{30000} \approx 173$). At this scale, it is hard to draw any conclusions. Therefore, we created subplot **b)**, which zooms in on predictions between 1 and 50. At this scale, we see that the model slightly underestimates introductions of 2 duplicates, and slightly

[6] See the replication package for details.

[7] https://discourse.mc-stan.org/t/clarification-about-the-purpose-of-reloo-in-the-loo-function/8742

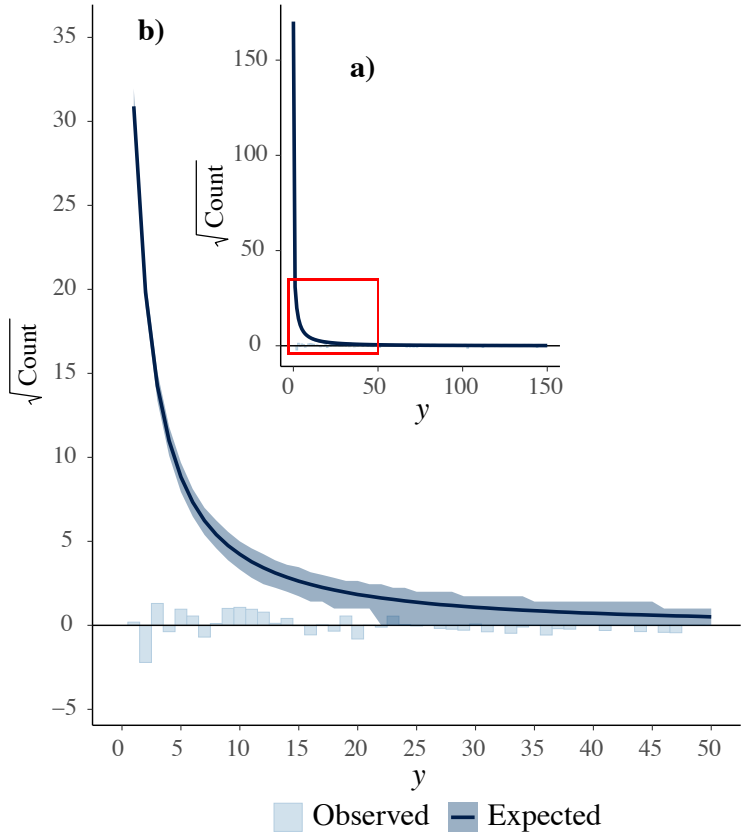


Figure 3: Suspended rootogram for model $\mathcal{M}_2$.
Subplot **a)**: expected count 0-150
Subplot **b)**: expected count 1-50; enlargement of the red square

less overestimates 3 duplicates. The rest of the expected counts fit progressively better. Overall, the model seems to fit the data well—this is also indicated by the narrow prediction interval around the expected frequency.

3.1.11 Model Comparison

The LOO-CV algorithm can also be used to compare and rank the models, as shown in Table 2. This indicates that model $\mathcal{M}_2$ outperforms the simpler model $\mathcal{M}_1$ by about eight standard deviations ($\frac{205.9}{24.7}$), which is a significant amount. The intercept-only model, $\mathcal{M}_0$, is a distant third.

Based on the DAG and the LOO-CV results, we decided to use model $\mathcal{M}_2$ for future exploration of the posterior predictions. Worth noting is that $\mathcal{M}_2$ also follows the causal model and, hence, from a causal perspective is the correct model given the assumptions concerning causality in the studied phenomenon.

Table 2: Comparison of LOO-CV expected log posterior. The first column contains the model name, the second column is the difference in expected log-predictive density, and the final column lists the difference in standard error.

Model	elpd_diff	se_diff
$\mathcal{M}_2$	0.0	0.0
$\mathcal{M}_1$	-205.9	24.7
$\mathcal{M}_0$	-2217.5	77.0

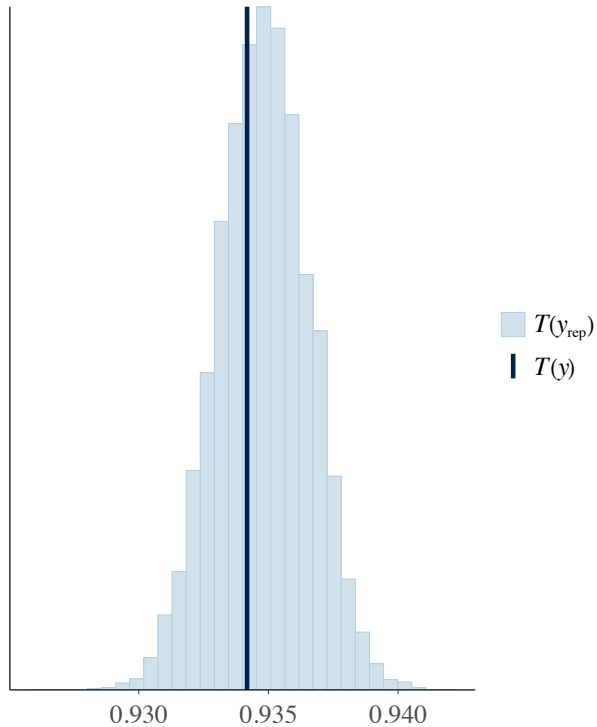


Figure 4: $\mathcal{M}_2$ posterior predicted proportion of 0. $T(y_{rep})$: Model predicted proportions. $T(y)$: Observed proportion of zeros. The observed value lies well within the predicted proportion (93%–94%).

3.1.12 Posterior Predictions

The final step in building a trustworthy model is to visualize posterior predictions and compare them with the collected data points. The model should fit the trained data, but should also allow some variations in predictions, to avoid overfitting.

Figure 4 shows the proportion of zeros (no clones added) predicted by model $\mathcal{M}_2$ when given the training data set. The blue histogram is the predicted proportion of zeros for various samples of the model, and the observed data is the solid dark blue vertical line. As can be seen in the figure, when seeing the training data set, the model expects between 93% and 94% zeros. This fits the observed data (black line $T(y)$) well. Further posterior predictive checks are available in the replication

package.

Once we have a robust model, we can use it to make inferences and visualizations. Because the model is multi-dimensional, we typically fixate all but one or two predictors, generate predictions, and then plot the inferences (e.g., expected values and credible intervals) as they vary across the varying predictors.

Further predictions and visualizations are available in Section 4.

3.2 Empirical Validation of $\mathcal{M}_2$

Following the methodology from Runeson et al. [40], we conducted an embedded case study, where the unit of analysis is a set of components belonging to a particular subsystem developed by a sub-organization in a large software development organization. We selected components based on availability and organizational structure—the organization was growing during the studied period, and wanted to know how changes to their existing business flow would affect their accumulation of technical debt.

Data collection was initially done via archival analysis (of code repositories and organizational charts), and later augmented by qualitative data from focus group interviews, which was transcribed and subject to open coding.

We performed four kinds of verification and validation of the model and its results: (i) we used synthetic data and analyzed whether the different models that we built could recover the used predictors; (ii) our choice of Hamiltonian Monte-Carlo means that the model execution itself will perform fundamental consistency checking (e.g. flagging divergent transitions [27]), in addition to the standard way to assess model fitting (e.g., rootograms [186], for our zero-inflated model); (iii) our choice of using PSIS-LOO (Pareto-Smoothed Importance Sampling, Vehtari et al. [178]) indicates that our model is free from highly influential points that might skew the model results, and finally (iv) we confronted three development teams, plus line and architect teams, with the results, which allowed us to evaluate whether the model inferences aligned with what they expected.

3.2.1 Organization and System Characteristics

The studied system consists of eight components belonging to a subsystem in a large-scale software system. All components are written in Java and are used in a scalable business processing application that combines event-driven processing with batch-processing tasks. We studied the components from January 2020 until November 2022, as development ramped up after having been paused for a few years. During the period under study, the number of contributors grew from 18 in three teams to, at most, 63 in ten teams. Two teams were disbanded during the study, leaving eight active teams at the end of the study.

The organization initially emphasized “collective ownership” of the code repos-

itories, meaning that teams were expected to contribute to all components, and make improvements wherever a developer sees the opportunity. However, some distribution of repository responsibilities between teams took place during 2021, whereby all development teams were expected to monitor the SonarQube issues and metrics for one or more allocated repositories. They were also expected to improve the test coverage in the allotted repositories, unless they were busy working on new features.

As the organization grew, it formed dedicated architect and quality assurance (QA) teams during late 2020. While the architect team was quite active, both in the production and the integration test code bases, the QA team did not contribute code but instead focused on planning and executing end-to-end tests.

3.2.2 Data Collection and Exploratory Data Analysis

The algorithm used for producing the quantitative data is described in Algorithm 2. We collected commit data from the Main branch of the studied repositories, as well as organizational data from the team wiki, which contained team composition at different points in time. In total, ten organization charts were identified for the studied period. Based on this data, we inferred which team a particular developer belonged to at a particular date. We used this affiliation data, together with the author and committer information, to identify the responsible author and the team they belonged at the time of the commit. Authors lacking known team affiliation (i.e., missing from the organization charts) were modeled as belonging to an "Unknown" team—we could have opted to remove or impute these data points, but chose to keep them, as they concerned only 2.0% of the input data, and had a low impact on the final model.

The git log provided data for each commit, such as date, author, committer, added lines of code (ADD) and removed lines of code (REM). By merging this with data from the SonarQube analysis, such as McCabe's cyclomatic complexity (COMPLEX), and the number of code clones before (DUP) and after the commit, we constructed a history over which author, and which committer, changed which file, at each point in time. This was then merged with the organizational data to attribute the change to the team that the author (or committer) belonged to at the time of the authoring (or committing).

In most cases, the author-team and committer-team are the same, and selecting one or the other does not change our conclusions. But we make the case that, in this organization, having "free-for-all commit rights," it is fairer to attribute the change to the committing team, rather than to the authoring team, as it is the committers team that merges the final change into the master branch. In organizations with more stringent merge rules or where some automated function merges the change, using the authoring team as the basis of analysis might make more sense.

After completing data collection, we performed exploratory data analysis, to ascertain data quality, and to identify possible patterns.

3.2.3 Feedback to Organization

We presented our findings to the studied organization, initially in a meeting with the architect team and the responsible line manager. In this meeting, we decided to present team-specific findings in meetings with the core teams: Red, Green, and Blue[8]. In the team-specific meetings, we discussed findings particular to the specific team.

We concluded with a summary presentation for the architect team, where we presented the findings together with comments from each core team. Both the architects and the development teams agreed that the metrics would be useful to gain an insight into team behavior related to clone introductions, but also stated that metrics related to the removal of code clones would be needed to give a more correct picture of the evolution of the code base.

3.2.4 Modeling Removal of Duplicates

We concluded the study by making a simple model for the removal of code clones, to see if this model changed some conclusions.

Algorithm 3: Calculating removed clones for changed files in a repository.

Data: C commits: C_i precedes C_{i+1}
Result: List of data for changed files
filestate $\leftarrow$ ()
for $C_i \in C$ **do**
  author $\leftarrow$ `AuthoredBy`(C_i)
  $\mathsf{d}_{auth} \leftarrow$ `AuthoredDate`(C_i)
  committer $\leftarrow$ `CommittedBy`(C_i)
  $\mathsf{d}_{comm} \leftarrow$ `CommittedDate`(C_i)
  $\mathsf{team}_a \leftarrow$ `FindTeam`$(\mathsf{author}, \mathsf{d}_{auth})$
  $\mathsf{team}_c \leftarrow$ `FindTeam`$(\mathsf{committer}, \mathsf{d}_{comm})$
  for $F_{j,i} \in$ `Files`(C_i) **do**
    $\mathsf{dup}_{prev} \leftarrow$ `Duplicates`$(F_{j,i-i})$
    $\mathsf{dup}_{curr} \leftarrow$ `Duplicates`$(F_{j,i})$
    $\Delta \leftarrow \mathsf{dup}_{curr} - \mathsf{dup}_{prev}$
    if $\Delta < 0$ **then** fixed $\leftarrow$ `abs`(Δ)
    else fixed $\leftarrow$ 0
    filestate $\leftarrow$ filestate $+ (F_{j,i}, \mathsf{team}_a, \mathsf{team}_c, \mathsf{fixed})$
  end
end
return filestate

[8] At least one developer in each of these three teams were contributing to the product throughout the whole studied period.

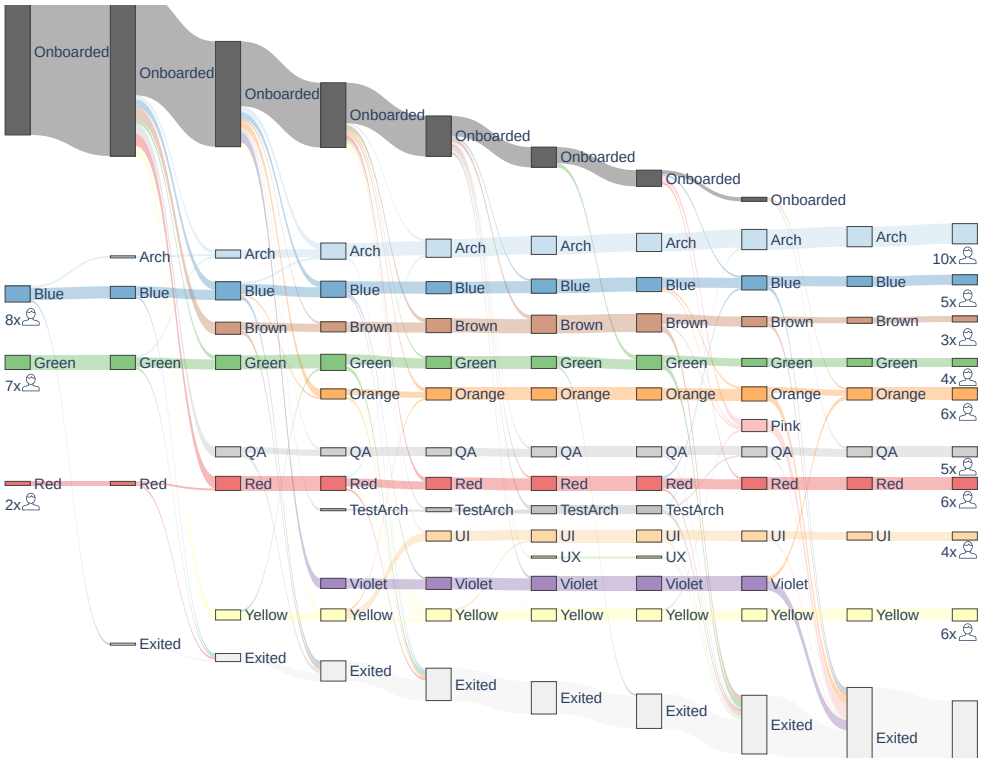


Figure 5: Flow of developers in and out of the organization, and between the different teams. Study start: Dec. 2019; Study end: Nov. 2022

We collected data on clone removals using Algorithm 3, and used an intercept model, structured like model $\mathcal{M}_0$, with y set to the number of removed clones (fixed), with identical priors as model $\mathcal{M}_0$, to illustrate the tendency of each team to remove clones. Results from the model were reported to the architect team in summary format, and are available in the replication package.

Further studies and model designs for clone removals are left as subjects of follow-up studies.

4 Results

4.1 Exploratory Data Analysis

All eight studied components existed at the start of the study and were being developed by three teams (Red, Green, and Blue) at that time.

At least one developer from each of these teams was part of the studied organi-

Table 3: Summary code statistics per repository. F_s/LOC_s: initial number of Java files/lines of code. F_e/LOC_e: closing number of Java files/lines of code. Chg: number of file change events during the study.

Repository	F_s	LOC_s	F_e	LOC_e	Chg
IntTest	243	99310	347	154637	3999
Jupiter	1103	151628	1768	219063	9413
Mars	413	64351	729	75050	2215
Mercury	166	23437	291	35597	1137
Neptune	288	38240	468	66616	1801
Saturn	1267	149820	2157	208294	6969
Uranus	227	26256	577	60423	3083
Venus	198	22703	482	55391	2390

zation during the whole study period, but all teams changed staffing, to some extent, during the study. The number of teams grew both organically (developers switching team affiliation) and via external recruitment.

Figure 5 shows the flow of developers into (**Onboarded**) and out of (**Exited**) the studied organization, as well as transfers between teams throughout the studied period. At the peak, 63 contributors formed eleven teams—including one architect team (Arch), one end-to-end testing team (QA, not contributing to the production-code base), and one UI team (mostly working in non-Java components). As two of the formed teams were disbanded during the study, at the end of the study eight developer teams, totaling 45 developers, were contributing to the code base, and 5 QA testers were validating the components end-to-end. In total, 82 developers were onboarded, and 50 left the organization during the study.

Table 3 shows the initial (F_s) and final (F_e) number of Java files, the initial (LOC_s) and final (LOC_e) lines of code, as well as the number of change events in each repository (Chg). The components were of different sizes, and changed to varying degrees. In the largest repository, Jupiter, almost nine times more files were changed than in the smallest, Mercury.

Figure 6 shows the proportion of file changes, per team and repository, that contain at least one new code clone. The size of the point is proportional to the number of file changes the team has performed in the given repository. We note that the architect team ("Arch"), though they make fewer changes than the more active teams (e.g., Blue, Green, and Red), are less likely to introduce duplicates. Furthermore, we note that the repositories have different likelihood of having duplicates introduced. In the integration test repository, most teams—except Architects—introduce at least one duplicate in every tenth file change, regardless of change size or other predictors. Some teams make very few changes, in particular Pink, which was active only a few months, and UI, which does not normally work in Java-based repositories. To avoid selection bias, we chose to keep these data points for model fitting, even though our analysis focused on the more active core teams.

Figure 7 shows the OCAM rank for the various teams in the Jupiter and Uranus repositories. As lower ranks imply higher contributions, we note that in Jupiter, Team Red leads in the number of commits, the total size of changed code (churn), and

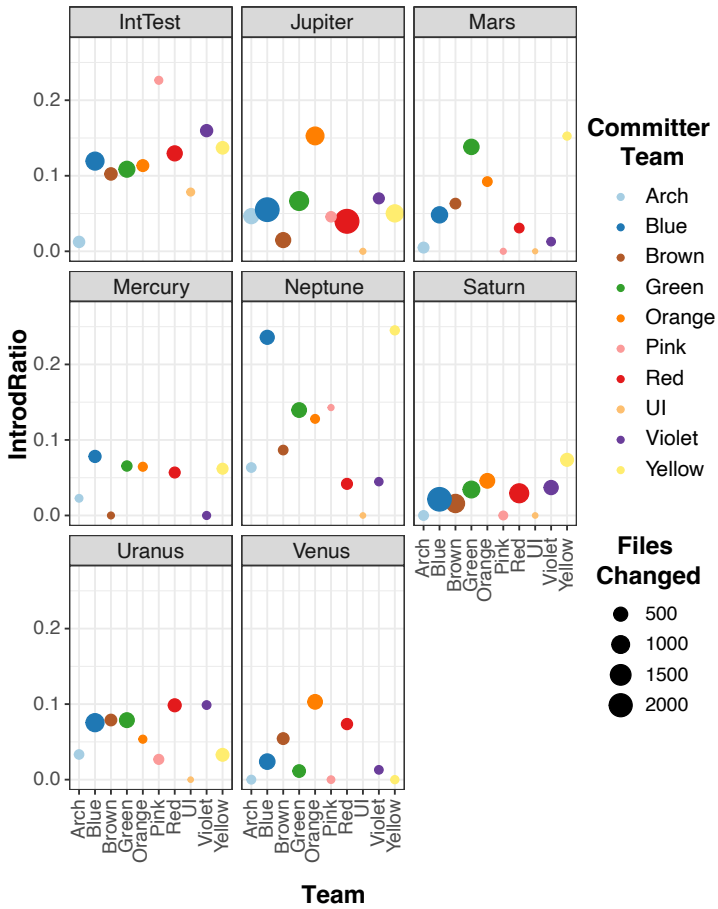


Figure 6: Observed proportion of file changes containing one or more new duplicates, per team and repository. The size of the point is proportional to the number of files changed by the team in the repository.

the added complexity. Team Blue also ranks high, indicating they have been highly active in this repository. In Uranus, the roles are practically reversed between Team Red and Blue, and we note that some teams (e.g., Pink, UI, Unknown, Violet) are consistently ranked low. We used OCAM plots like these to discuss ownership with the teams and contrast these findings with the visualizations from our model.

4.1.1 Core Teams in the Largest Repository

Three development teams were part of the organization during the entire studied period, and we also considered the Architect team, which was formed by developers from the Blue and Green teams in early 2020. Summary statistics for the contributions of the four core teams to the Jupiter repository, the largest in the product, is shown in Table 4. Team Red made over 2000 changes to files in the repository, and

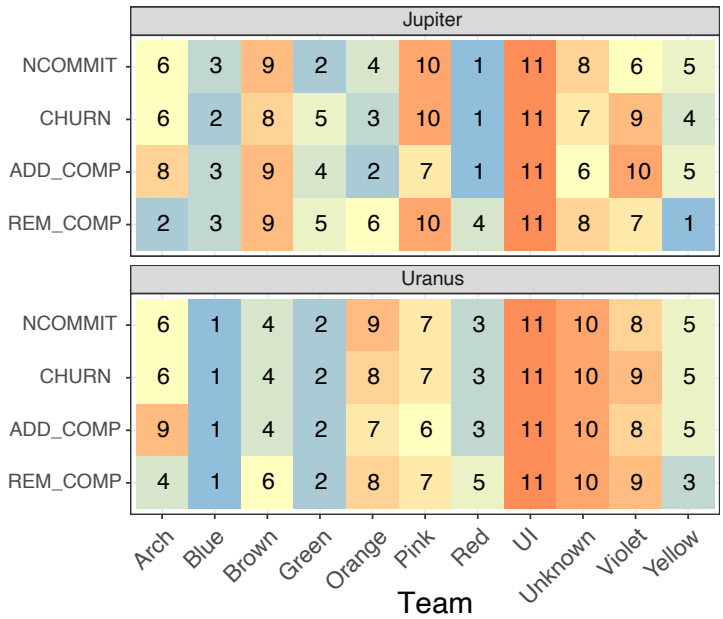


Figure 7: OCAM model rank of team contributions to the Jupiter and Uranus repositories. NCOMMIT: Number of commits. CHURN: Sum of `max(added, removed)` LOC. ADD_COMP: Sum of added McCabe complexity. REM_COMP: Sum of removed complexity. All metrics increase as contributions increase. Rank 1 means the highest contribution amongst the studied teams.

Table 4: Summary statistics for core teams in the Jupiter repository. `n`: number of changes to files. `%dup`: percentage of changes that introduce a clone. C_{q50}: median complexity of the changed file. JLINE_s: initial ratio of authored lines. JLINE_e: closing ratio of authored lines. The last row is the total metrics for all teams (including the absolute number of all lines in the Java files).

Team	n	%dup	C_{q50}	JLINE_s	JLINE_e
Red	2166	4.0%	18	10.5%	17.9%
Arch	623	4.6%	16	---	6.4%
Green	1172	6.7%	21	12.2%	3.8%
Blue	2123	5.5%	10	15.0%	5.0%
Other				62.3%	73.4%
Total	9413	5.9%	13	204 kL	282 kL

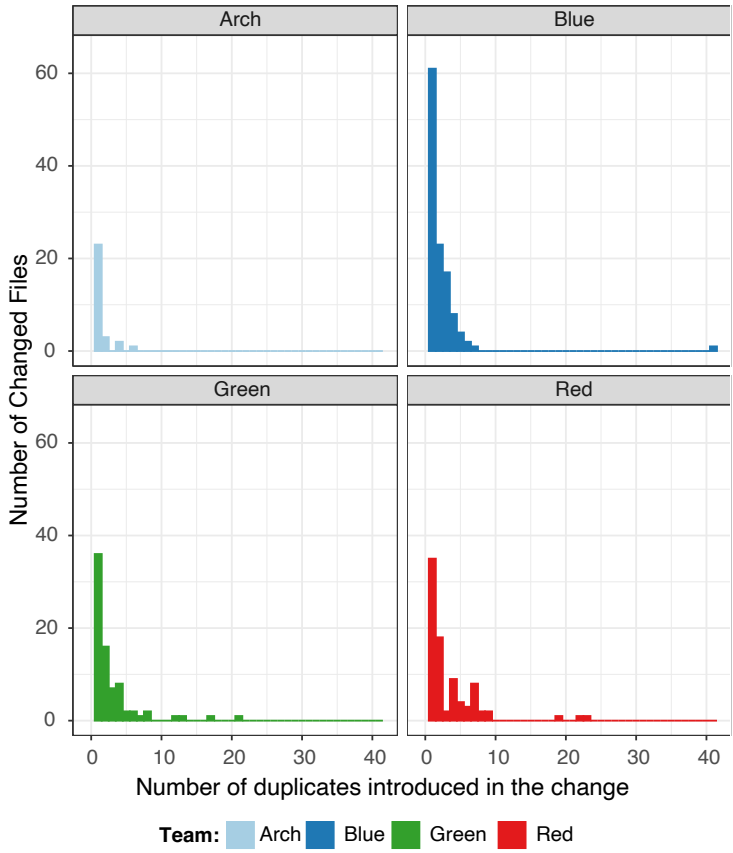


Figure 8: Histogram of the observed number of introduced duplicates in the largest repository (Jupiter) by four teams.

Team Green about half as many. As part of these changes, the proportion of code authored by Team Red grew from 10.5% to almost 18%, while the proportion authored by Team Green declined from 12% to barely 4%. Although Team Blue made over 2000 changes, the proportion of code authored by them declined from 15% to 5%, meaning that they either did significantly smaller changes and/or were consistently changing the same portion of the code in that particular component. At the end of the study, the four additional development teams had contributed between 0.6% to 6.9%—summarized in the *Other* row in the table. About 20% of the code in the repository had been authored by developers who had left the organization. About 5.9% of the file changes introduce duplicates in this repository.

Figure 8 show four histograms, one per team, where the x-axis represents the number of introduced duplicates, in the Jupiter repository, and the y-axis is the number of file changes that add this number of duplicates. For these teams, eight changes introduce more than ten duplicates (four by Green, three by Red, and one by Blue).

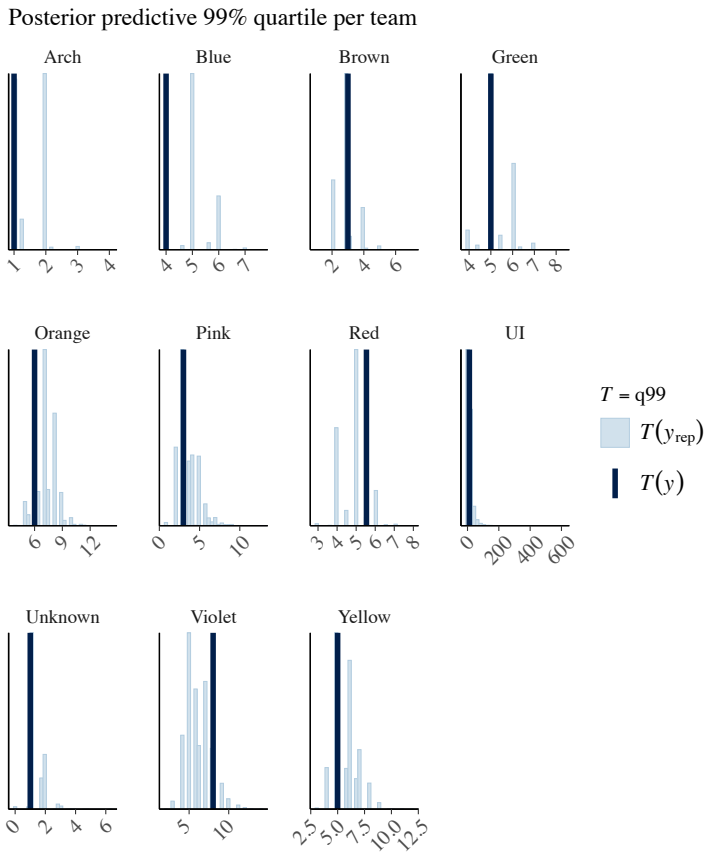


Figure 9: Posterior predicted 99th percentile per team, across all repositories. Note the different x-axis scales.

We also note that the architect team introduced fewer duplicates (23 single-duplicate-introducing changes, and only a few changes with 2–6 duplicates). The Blue team made more than 60 changes introducing a single duplicate, and they committed one change with more than 40 duplicates, which also was the maximum observed value for this repository across all teams.

The summary data alone shows that teams have different rates of introducing duplicates in the Jupiter repository. We will explore the differences more using the model in the following section.

4.2 Fitness of the Model

Figure 9 shows a posterior predictive visualization displaying the 99th percentile prediction for the various teams across all repositories. As seen on the respective x-axis,[9] most teams are expected to introduce less than 16 duplicates at the 99th percentile.

[9] Note the different scales on the x-axes.

Table 5: Summary statistics for file changes in all repositories.

Metric	Q_{50}	Q_{95}	Q_{99}	Max
ADD	6	143	370	3772
REM	2	92	311	3413
COMP	16	282	633	1244
DUP	0	36	99	664

The exception is the UI team, where some predictions range up into the hundreds. This is because the UI team only contributed a few changes to a few repositories, causing the model to be more uncertain. In general, the expected 99th percentile values (light blue histograms) align well with the observed values (solid $T(y)$ line), indicating a good model fit.

More posterior prediction visualizations are available in the replication package.

4.3 Model predictions

Given that small changes rarely introduce any duplicates and that all metrics are highly right-skewed, we focus our predictions on choosing either the median (indicated by Q_{50}), 95th percentile (indicated by Q_{95}), or 99th percentile (indicated by Q_{99}) as predictor values. The observed values for these metrics across all repositories are shown in Table 5.

Figure 10 shows the predicted probability of introducing at least one duplicate for a large change (added and removed lines at their 99th percentile) to a complex file with many duplicates (complexity and existing duplicates at their 95th percentile). The model predictions are made for the teams active at the end of the studied period, plus a simulated "Average" team, which pools information from all teams and repositories. The different behaviors of the teams are clearly visible in the repositories. The figure indicates that in many repositories (e.g., IntTest, Saturn, Uranus, Venus), the architect team has a low probability of introducing duplicates—in some cases this behavior is shared with other teams (e.g., Red in IntTest and Green in Uranus).

The largest repository, Jupiter, has the highest probability of teams introducing code clones, in particular by the Blue, Brown, and Orange teams. For these teams, the model expects more than a 50% probability that a change with these characteristics will introduce at least one duplicate, whereas the probability for the Green team is around 30%. In the second largest repository, Saturn, the only teams that deviate from the norm are the architects, with a *lower-than-average probability* of introducing clones, and Team Yellow, with a slightly *higher-than-average* probability.

4.4 Model Evaluation---RQ 1, RQ 2

Table 2 shows that the model that best fits our data, using approximated leave-one-out cross-validation (LOO-CV), is $\mathcal{M}_2$, described in Eqs. III.9–III.11. This is a

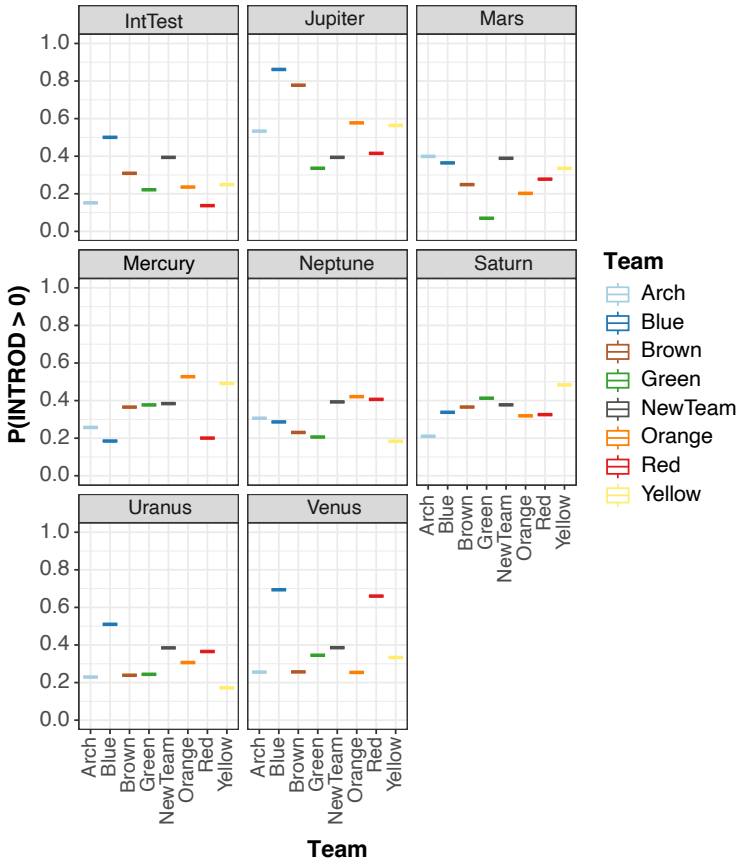


Figure 10: Probability of introducing at least one duplicate for a large change in a complex file. ADD = 370 (Q_{99}); REM = 311 (Q_{99}), COMP = 282 (Q_{95}); DUP = 36 (Q_{95})

Bayesian MLGLM, with four numerical and two categorical predictors, with varying slopes and intercepts. To assess model convergence, we used standard diagnostic checks [27, 178, 185], such as $\widehat{R}$, n_{eff}-ratio and Pareto-k value approximation, and to assess model fit, we used the rootogram method recommended by Kleiber et al. [186]. Detailed diagnostics are available in the replication package [180].

4.5 Predictions of the Estimated Number of Introduced Duplicates

Using $\mathcal{M}_2$, we can make predictions using the posterior distribution, to visualize and summarize results.

Figure 11 shows how the estimated number of introduced duplicates varies for a large change, made by two teams (Blue and Red) in two repositories (Jupiter and Uranus). The x-axis represents the complexity of the changed file, and the differ-

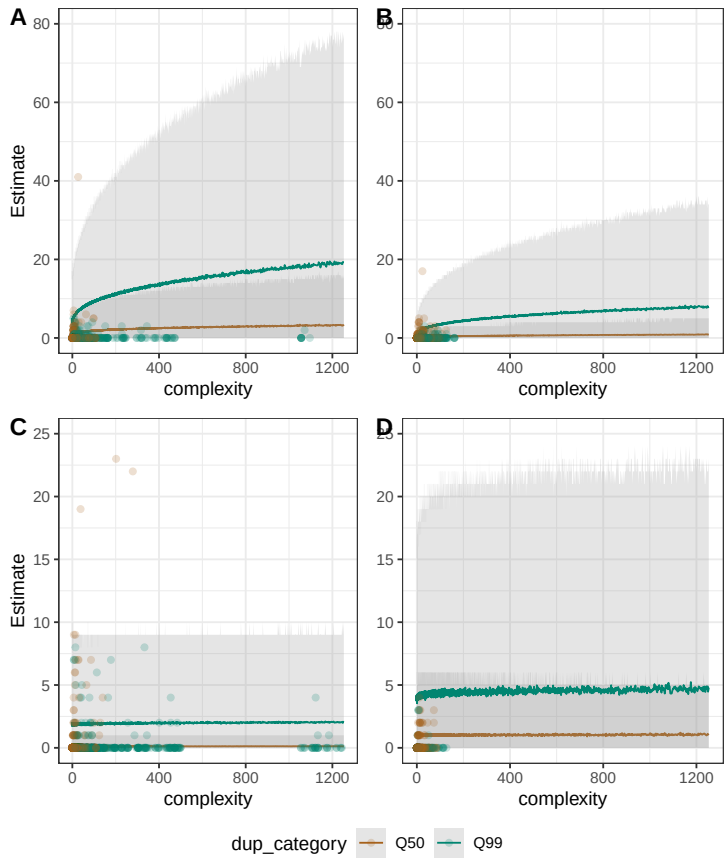


Figure 11: Estimated number of introduced clones for a large change, by complexity and existing duplicates. Q_{50}: DUP = 0; Q_{99}: DUP = 99 A) Team Blue in Jupiter; B) Team Blue in Uranus; C) Team Red in Jupiter; D) Team Red in Uranus; ADD = 143 (Q_{95}); REM = 92 (Q_{95}); observations as points, regardless of change size. The shaded area represents the 89% credible interval. Team Red is largely indifferent to existing complexity. Both teams are sensitive to existing duplicates, and are expected to introduce fewer clones in repositories that they have chosen to be responsible for.

ent colored lines represent the expected number of introduced clones, depending on whether the file had any existing duplicates or not. Both added and removed lines are set to their respective 95th-percentile value. The shaded areas represent the 89% credible interval for predictions—all areas start at 0, and the larger the number of existing duplicates, the larger the credible interval. The figure also contains observed data points for the respective teams, regardless of the size of the change, but colored by whether the file had duplicates or not.

Complexity matters—but not for all
As is shown in Fig. 11, the expected number of duplicates for Team Red (bottom row) does not depend on the existing complexity of the changed file. The other teams show varying strengths in the association between existing complexity in the file and the expected number of introduced clones. The preexisting number of clones plays a role in all teams—files without any existing clones are unlikely to see a large number of introduced clones, and the larger the number of existing clones, the larger the expected number of newly introduced clones.

During 2021, Team Blue (top row) chose to be responsible for Uranus (B), while Team Red (bottom row) chose Jupiter (C). As the figure shows, both teams are expected to add around half as many duplicates in the repository they assumed ownership of, relative to the other.

(Some) owners are different

As shown in Fig. 11, assumed ownership of a component does seem to impact the rate of clone introductions. In this study, teams were allowed to self-select which components to be responsible for, from the perspective of reducing SonarQube violations and increase test coverage. There was no formal team ownership (such as requiring sign-off on commits or patch sets) in place. As illustrated in the figure, both Team Blue and Team Red appear to introduce about half as many duplicates in the components they chose to be responsible for, relative to the other.

Given what Dietz et al. [155] say about the importance of *monitoring of the common resource* to efficiently govern its usage, we postulate that our model, and in particular the posterior predictions and visualizations would be a useful tool for architects and team leaders to judge the adherence of individual teams to the agreed code duplication policies. Unlike standard statistical summaries (such as Fig. 6 and Fig. 8), which does not take complexity and existing number of clones into account, our model adjusts its predictions to the circumstances of each file change. Thus, it has the potential to be more precise, and therefore, more trustworthy, for developers and architects alike.

4.6 Comparing with the Average

Using the model, we can simulate how a new team (i.e., a team that has not yet produced any data points) is likely to behave, based on the behavior of existing teams, using a concept called “partial pooling” in Bayesian contexts. This can give interesting

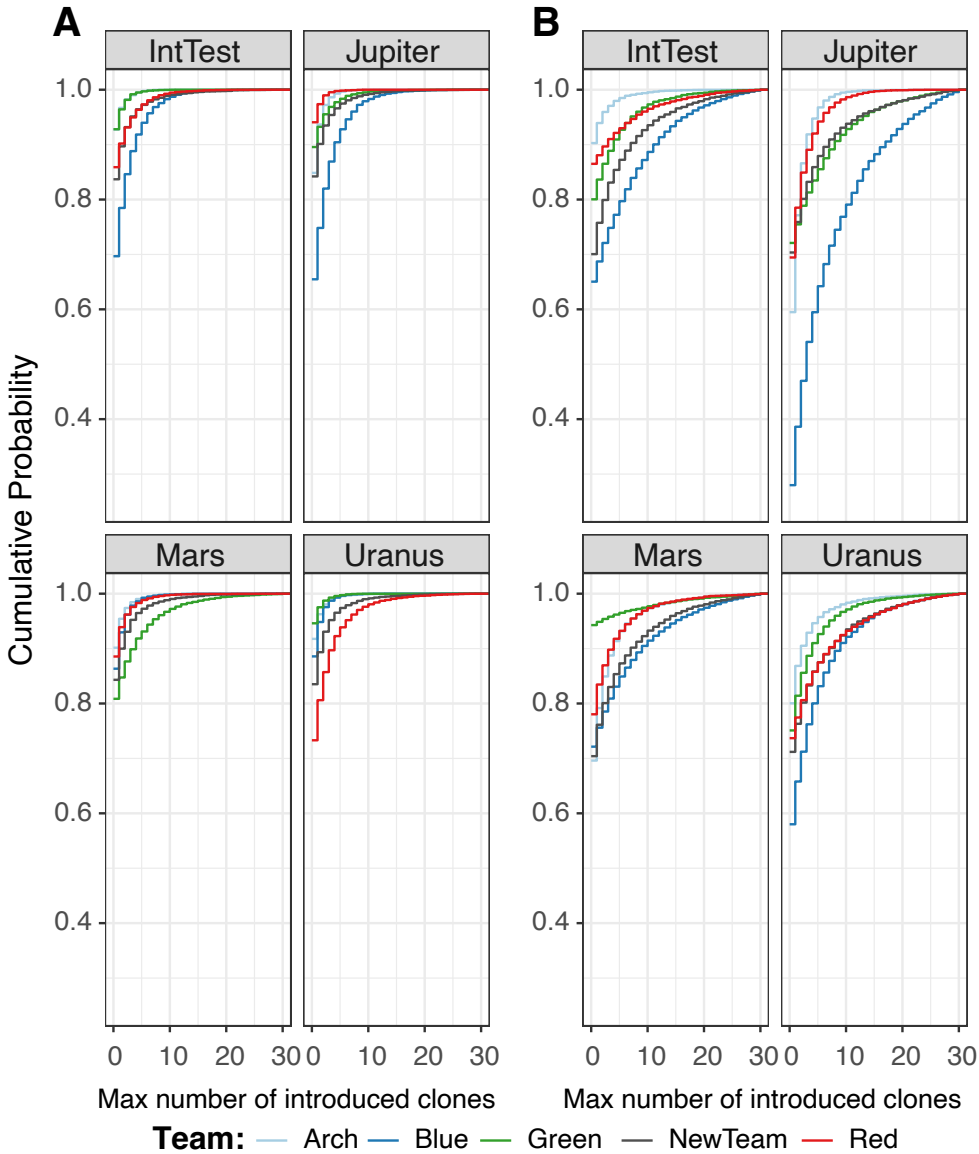


Figure 12: Cumulative probability of introduced code clones for four existing and a simulated new team, making a large change; ADD = 143 (Q_{95}); REM = 92 (Q_{95}); in four repositories; **a)** in a file of median complexity lacking existing duplicates; COMP = 16 (Q_{50}); DUP = 0 (Q_{50}), and **b)** in a complex file with many existing duplicates; COMP = 282 (Q_{95}); DUP = 36 (Q_{95}).

comparisons. Figure 12 shows the expected cumulative probability of introducing duplicates for four existing teams, as well as the population average ("NewTeam") in four repositories, for a fairly large change (added and removed lines to their 95th percentile). The x-axis shows the maximum number of introduced duplicates and the y-axis shows the probability of seeing the corresponding x number of introduced clones, or less. In Figure 12, part A, the complexity and existing number of duplicates are set to their median values (16 and 0, respectively), indicating a large change in a median complex file. In Figure 12, part B, all values are instead set to their 95th percentile values (complexity 282 and 36 existing duplicates), indicating an equally sized change, in a more complex file. Based on the figure, we can draw several conclusions:

- "NewTeam", the population average—being the average—can be used to indicate whether a team is more or less likely to introduce duplicates than the average team. We see that the placement of teams, relative to the population average, varies across repositories, and depending on complexity. For a change in a median file in Uranus, Team Red is below average, whereas they are close to the average for a complex change. The opposite is true for Team Blue, who are more affected by complexity.

- Team Blue, as stated, behaves below average for both integration tests and Jupiter, the largest repository, where they have slightly less than 30% probability of avoiding introducing duplicates in a complex file. In Mars and Uranus, changing median complex files lacking duplicates, they behave better than the average team.

- Team Red, in general, behaves like the average team, or better, with the exception of changing median complex files in Uranus. The model expects the probability of introducing a duplicate in Uranus to be similar regardless of complexity and whether there are duplicates or not (though the slope is different, so the number of duplicates introduced would be expected to be higher in a more complex file)

- Team Green in the Mars repository, is very unlikely to introduce clones in the complex file, but in the median file, they behave slightly worse than average (though the probability of avoiding clones is still around 80%). The reason for this is to be found in the data—the majority of cases where Team Green has introduced clones in Mars have been in files lacking any existing duplicates, and they have never introduced a clone in a file with more than two existing duplicates. Thus, the model infers that when Team Green makes a change in a complex file with existing duplicates, they are unlikely to introduce any clone.

- Architects are, in general, less likely to introduce clones, particularly in the integration test repository, where the model infers around a 90% chance of

avoiding clones in a complex file. One exception is the probability (around 60%) of keeping zero clones for a change in a complex Jupiter file—but the probability of more than one clone drops off more quickly than the average team.

Pathbreakers vs. caretakers
When confronted with the findings from the collected data, both the Architect team and Team Blue stated that the reason for the aberrant behavior of Team Blue in the Jupiter repository was that they handled "the most difficult tasks", and during the studied period they were actively working on rearchitecting the main business flow, largely encapsulated inside the Jupiter repository. Thus, they could be seen as "pathbreakers", versus the more careful and deliberate "caretakers", i.e., the Red and Green teams.

4.7 Teams' Feedback---RQ 3

While the organization lacked any formal "gatekeeper" form of ownership of the repositories, the teams did state that they, during 2021, had distributed the repositories between themselves to manage SonarQube violations and improve test coverage. Teams were allowed to self-select repositories to care for, and based on the (LOC) size of each repository, one or more repositories were chosen. While Team Red had responsibility for one repository—Jupiter, the largest, Team Green had seven, among them Mars, and Team Blue had two, one of which was Uranus. The Architect team was not allocated any repository, as they had the overall product structure responsibility.

Figure 12 shows that the Architect team in the integration test repository has a very low probability of introducing clones, relative to the normal development teams. They attributed this to the fact that "we make changes to this repository only when we change various interfaces of the product". Looking at the data, this statement is partially correct—the Architect team does introduce fewer new files (14 of 241, $\approx 5.8\%$) than the Blue (93/1030, $\approx 9.0\%$) and Brown (24/332, $\approx 7.2\%$) teams. However, many teams are less active, and Team Red (another core team) introduced 30 new files out of 617 changes ($\approx 4.9\%$).

The Blue team in the Jupiter repository shows a stark difference from the other teams. This repository is the largest in the product, where most of the business logic is contained. When we discussed this with the team, we got a partial explanation: "We were in Proof-of-Concept mode, no process was followed. Lots of code was copied across repositories... And after that, there was a reset, where 1.5 years of work was almost null and void..." The Architect team backs this up: "It was a huge feature, a twelve–fourteen-kind-of-sprints feature, changing almost every aspect of the two main business flows. Because of this, they might have been replicating parts

of it, causing duplicates".

As depicted in Fig. 12, Team Green behaves close to the population average in most repositories. However, in the Mars repository, they behave significantly differently, having a 90% chance of avoiding introducing a duplicate when changing a complex file, compared to the population average of around 75% (a figure that is also close to the Red and Blue team behavior). Team Green responds to this fact with: "In Mars, we have achieved kind of a standard of code, whereas in Jupiter we are introducing diverse code, which results in bad coding practices sometimes". This indicates that this team is familiar with Mars, while being more foreign to Jupiter. The other teams do not appear to be as familiar with Mars as Team Green.

All teams state that they think the statistics and visualizations give important insights into team behavior, but also state that a fuller picture would be gained by also modeling how teams improve code by removing duplicates.

4.8 Model for Removal of Code Clones

As the teams indicated that the removal of clones by teams in different components should also be considered, we tried to model clone removals via the same method as the introductions. We collected data on clone removals according to Algorithm 3, and then fit the simplest, intercept-based model onto the collected data to get the average behavior of the team in each repository without considering any predictors.

To construct model $\mathcal{M}_3$, we used Equation III.1, with y representing the number of (zero or more) removed code clones. The parameters of the zero-inflated negative Binomial likelihood are defined according to Eqs. III.2–III.3. As the prior predictive checks showed reasonable values, we used the same priors (defined by Eq. III.4) for model $\mathcal{M}_3$ as for model $\mathcal{M}_0$.

Figure 13 shows the probability of removing one or more duplicates for the eight currently active teams in the various repositories. A few teams and repositories stand out, such as the Architects and Team Green in Neptune, Team Yellow in integration tests, and Team Brown in Mars. But the general pattern is that the teams show quite similar behavior, and the simple intercept model (not considering any predictors) expects, at most, a 5–10% chance of a team removing a clone, regardless of which repository.

5 Discussion

In this section, we separate the discussion of the general findings from the case-specific ones.

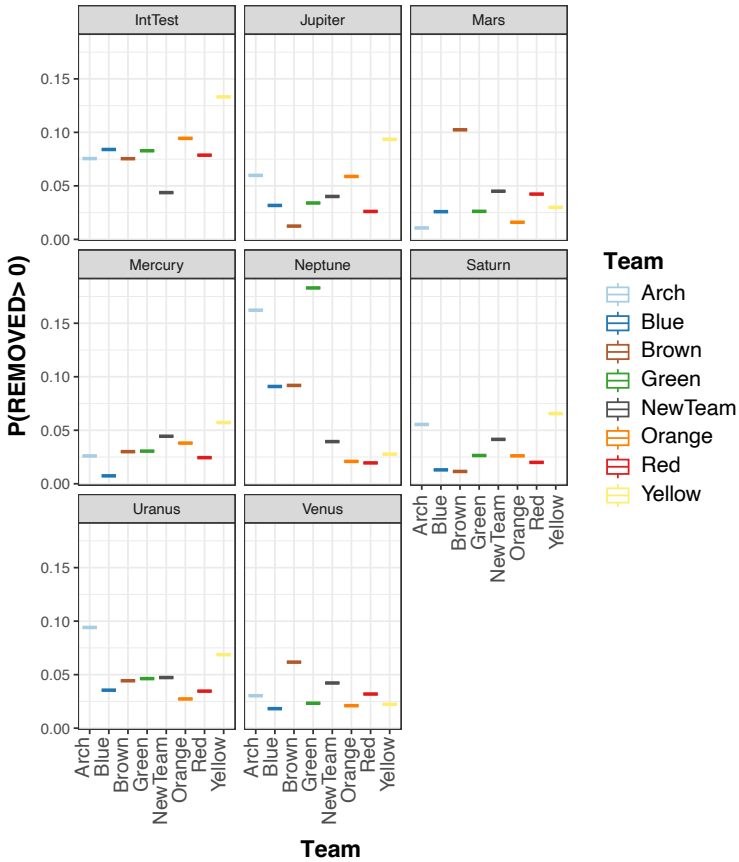


Figure 13: Probability of removed clones, based on an intercept-only model.

5.1 Monitoring Code Clone Introduction Trends

Relative to summary statistics, such as the proportional data in Fig. 6 and Table 4, our Bayesian model allows instant "grading" of a contribution. Using the tools that generated Fig. 12, each file change in a merge request can be automatically rated, both in comparison to the average team and relative to the typical behavior of the developing team. Given input data (i.e., the number of added and removed lines, the existing complexity and number of duplicates in the file, and the author team), the model would output the probability of seeing y new clones. This could be integrated into automated review tools, such as SonarQube or Gitlab pipelines. Providing this sort of feedback *before* merging to the main branch would allow the team to proactively affect their "clone introduction probability," before the next iteration of the model is built.

Statistical summaries such as the ones in Fig. 11 and Fig. 12 could be used by architects or experienced developers to ascertain how newly formed or onboarded

teams fare, relative to "the average team" and detect when they might have a different behavior than what could be expected. This should allow an objective assessment of how teams behave from a clone-introducing perspective, relative to the average team, in different components. It is important to note that, as the averaging occurs across the entire population of developer teams, there will always be *some* teams that are better, and *some* that are worse than the average. But the model also allows an objective assessment (with uncertainty estimates) of *how much* better or worse, each team is, relative to the average team. This should allow the architects to gain insights that can be used to initiate discussions about how each team are performing in the different components of a subsystem, which in turn could be used to identify components where more architectural support is warranted.

Choice of Grouping Level for Code Clone Introduction Modeling Relative to earlier software engineering studies, which usually mapped out ownership based on the number of changes individuals make to distinct files [e.g. 157, 159, 160], our study aligns more with the OCAM [165] way of inferring teams based on the author or committer of a particular file change. In the studied organization, teams were responsible for what they committed, including reviewing and testing the change, and we validated this inference with the studied organization. We attribute the team affiliation based on where the committer was located at the time of the change. This means that the ownership of components will fluctuate over time, as people change teams (as seen in Fig. 5).

In other organizations, like Open-Source projects with loosely affiliated teams and occasional corporate contributors, different methods might be applied to assign team labels to file changes. In a typical successful Open-Source system[10], committing to active branches is often reserved to a select set of *Committers*, while *Reviewers* are responsible for reviewing incoming code changes, and *Maintainers* are responsible for the coordination of contributions, releases, and the overall project direction. The Linux kernel project[11] allows different headers in its patches (e.g., `Co-developed-by:`, `Acked-by:`, `Signed-off-by:`, `Reviewed-by:`), which should be used to identify contributing individuals. Depending on the question you are trying to answer (e.g., "In general, how do component *C* patches, originating from Organization *X*, compare from a clone perspective to those written by *Y*"), one or more of these attributes could be used to determine the contributing team, in addition to the git standard `Author:` or `Committer:` attributes. This might also require updates to the causal model underlying the Bayesian model.

The Bayesian model is agnostic to the choice of grouping level; it will use whatever groups (labels) the data contains. However, too wide groups (e.g., sub-

[10] `https://docs.linuxfoundation.org/lfx/project-control-center/v2-latest-version/reports/health-metrics/code-contributions`

[11] `https://docs.kernel.org/process/submitting-patches.html`

organizations, geographical sites, or countries) might produce more diffuse, less-actionable results, since the findings might be harder to connect to team practices and team behavior. Conversely, groupings that are too fine-grained (e.g., on individual author level) might produce groups with too few data points to make reliable predictions. Additionally, attributing a change to a single author would introduce bias by disregarding contributions made by others, for instance, due to pair programming, reviewing, or peer testing.

Capturing the Diversity of Team Behaviors The visualizations of different teams' probability of introducing one or more duplicates (e.g. Fig. 12) can be used to discuss the causes of clone introductions with the team. Team Blue was, by architects and themselves, seen as "pathbreakers," who were often tasked to perform the more complex tasks (such as architectural changes), relative to the more "caretaking" Team Red (who, in turn, were standing out in that they were, in most components, unaffected by existing complexity when introducing clones). Thus, the visualizations can guide architects to understand, and initiate discussions with, different teams, in a language that they understand. Both Team Blue and the Architect team agreed that neither a formal process nor SonarQube rules were followed while Team Blue was changing the architecture as a "Proof of Concept." This is the likely reason for them being much more likely to introduce clones in the largest repository in the product (and also, to a lesser extent, in the other repositories).

Comparing With the Average, Rather Than Between Teams Figure 12 illustrates how a Bayesian model can simulate a new team, based on the pooled population average. The behavior of the "generic new team" is based on the partial pooling of existing teams, with the influence of individual teams proportional to how much they have contributed to the various repositories. These predictions can be used to monitor the progress of both new and existing teams and detect behaviors that could be improved, or competencies that should be widened.

Relating to the findings made by Dietz et al. [155] to avoid the "Tragedy of the Commons", we postulate that our model, and its visualizations can serve as a useful tool to *monitor* (or act as a *quality gateway* for) both newly onboarded, as well as more experienced, teams making contributions to existing components. The architect team also indicated that the visualizations could improve *communication* about how and why teams introduced code clones in different components. Comparing with the average (rather than pitting one team against another) might also improve the *social networks* between architects and development teams. However, seeing any long-term effects requires more longitudinal observations, which we defer to future studies on this subject.

5.2 Case-specific findings

In the following sections, we discuss the major findings highlighted by the model, as well as how the teams reacted to these findings.

Architects are Active and Also Behave Differently In this organization, the architect team is active in all repositories, and employs a different behavior than the regular development teams. They have the highest median number of removed lines (7), which is higher than their median number of added lines (3). Most other teams have higher median added lines (ranging from 3 to 20) than median removed lines (ranging from 0 to 4). Figure 12 shows that the behavior is especially pronounced in the integration test repository, where the architect team is between 5–25% less likely to introduce duplicates in a complex file than the other teams. The architect team claims that this is because they are mainly making changes due to added or changed external interfaces.

Teams React Differently to Existing Complexity As visualized in Fig. 11, Team Red is largely indifferent to existing complexity with regard to the rate of clone introduction. For all repositories but one (Venus), this team seems to be unaffected by complexity, although the rate of clone introduction will vary across repositories (as shown in Fig. 11, parts C and D). Team Blue, on the other hand employs the more common behavior, where the likelihood of introducing code clones increases, as the complexity of the file grows. Team Red stated that they are careful to avoid introducing new clones, and that they have started fixing old SonarQube violations as they go along with their contributions. They have not contributed much code to the Venus repository, which could be one reason for why the model assumes they behave like an average team there.

Ownership Matters In Fig. 11, Team Blue is responsible for Uranus (B) and is less likely to introduce clones here than in Jupiter (A). The reverse is true for Team Red, who is responsible for Jupiter (C), but not for Uranus (D). In Fig. 12, Team Green is responsible for Mars, and has the lowest likelihood (about 10%) of introducing clones of all teams, when changing complex files. Initially, the studied organization employed total collective ownership. However, this changed in 2021, when the teams were allowed to *self-select* which components they should care for by fixing SonarQube issues and improving test coverage. To some extent, this can explain the behavior—teams would likely select components where they "felt at home" when they allocated team ownership. Because of this, we cannot claim causality, but we can state that for these three teams, ownership allocation was reflected in how they behaved in the respective components.

Given that most Open-Source projects also employ self-selection, it would be interesting to evaluate whether this finding applies also to such organizations, and how

it varies for corporate contributors. To make such an assessment, in addition to the Bayesian model and a tool such as SonarQube, the following would be needed: (i) the full source code, including commit history (to build the historical data needed for predictions) of the project you want to analyze; (ii) how to map contributions using labels such as `Author:`, `Committer:` or any of the ones mentioned in Chapter 5.1, into the team (or other grouping level of interest) responsible for the contribution; (iii) how the responsibilities of these groups are mapped to the *Contributor*, *Committer*, *Reviewer*, and *Maintainer* roles for the various components of the system under study. We would assume that committers (and possibly maintainers) would show the strongest ownership behaviors, followed by reviewers, and last contributors. If our assumption holds, for a given component, we would see groups with the committer or maintainer role behave similarly to Team Green in Mars or Team Red in Jupiter, relative to how they behave in other components, where they are merely contributors. Unfortunately, we have to defer the studies of such Open-Source systems to a follow-up study.

Do Not Forget Removals When discussing our findings with the teams, they often asked about clone removals. To illustrate this, we also developed a simple (intercept-based) model for how teams remove clones in the various repositories. This is illustrated in Fig. 13, where we see the overall clone removal probability for various teams, in the various repositories. There are some teams that "stand out", such as Team Yellow in integration tests, and the Architects and Team Green (and to a lesser extent Team Blue and Team Brown) in Neptune. Overall, this model illustrates that most teams behave quite similarly regarding clone removals. Further research in this area is deferred to a follow-up study to explicitly model removals with a different DAG and possibly different predictors.

6 Threats to validity

We structure our threats to validity according to the four different angles recommended by Wohlin et al. [139].

Construct validity concerns whether the studied measures reflect what the researcher had in mind, and what is stated in the research questions. We base our study on a causal DAG and attribute latent constructs (such as team cleanliness and knowledge of components) to quantitative data (such as introductions of code clones). To aid interpretation of the results, and improve construct validity, we presented the findings to four teams, and included their feedback in this paper.

Changes to source code are made by individuals, based on more or less feedback from other individuals. In this study, we attribute the *committing* individual, at the time of the commit, to the associated team according to the organization chart, and use this information to make inferences about general team behavior. Several possi-

ble threats arise from this attribution: (i) modern software development methods (e.g, pair or mob programming), and software version control systems (e.g., Gerrit[12] and GitLab[13]) allow multiple developers to collaborate on a *patch set*, before one person finally merges it, as one or more commits, into the master branch; (ii) individuals do not necessarily represent the team they are working in—misrepresentations are possible, for instance by "lone wolf development," or by individuals working closer to other teams than their officially assigned; (iii) the committing individual might have no relation at all to the actual changes—in some organizations, the commit function is outsourced to an automated function, that merges the code once all required tests pass. Regarding threat (i), we compared inferences based on both original author of the commit and the committer, and found that they were virtually identical; regarding threat (ii), we validated the organization charts with the studied organization, and found them to be reasonably accurate and up-to-date; regarding threat (iii), the organization did not use automated merge tools and was not rebasing (i.e., re-writing) commits to any large extent.

The Git logs contained a small number of authors that could not be associated with a team. As this concerned only a few data points (less than 2% of the data), we modeled these as belonging to a separate "Unknown" team. Although not correct from an organizational point of view, this allowed these data points to influence the population average, which would not have been possible if these data points had been excluded from the analysis.

We based our model on the detection of exact clones; Type 1 as defined by Bellon et al. [187] and Koschke [140]. We used the default configuration of SonarQube to detect clones. Evaluating how the model performs for the identification of Type 2 (renamed/parameterized), Type 3 (near miss), and Type 4 (semantic) clones remains a subject for future studies.

Internal validity deals with whether there might be other, non-studied factors that could explain some of the findings.

One complicating factor is that the study took place during the Covid-19 remote work period, where developers in their daily work had to work and collaborate remotely. This possibly impacted both the onboarding of new teams and team members, as well as intra- and inter-team collaboration. However, all the studied teams were operating under the same remote-work rules, so the same confounding factor applies to all teams in this study. Still, teams might react differently to the remote-work mandate, but we have to defer this factor to a future study. We found during the focus groups that even with the remote work mandate lifted, the studied organization was continuing to use a hybrid work mode, where teams would come to the office one or two days per week, and work the other days from the home office.

We focused our qualitative work on the core teams, with deep knowledge of

[12] `https://www.gerritcodereview.com/`

[13] `https://about.gitlab.com/`

the product. This was partly for availability reasons, and partly because these teams were the most experienced, and had members that had been part of the product for a long time. The same method could have been used to study less contributing teams, but then considerable effort would have to be spent to find the people who were part of these teams during the study period.

External validity concerns to what extent it is possible to generalize the findings, and to what extent the findings are of interest to other people outside of the investigated case.

This paper contains data from a particular system, developed by a particular organization, which limits generalizability. However, as Flyvbjerg [42] states, cases play an important role in human learning, and it is, in fact, possible to learn from a single case.

We have tried to describe characteristics that might enable others to judge whether our findings apply to other systems, but we cannot claim generalizability across all possible systems or organizations. To aid this judgment, we provide the full anonymized data set, including all models and graphs described in the paper, and invite others to replicate the study in other contexts.

The constructs in our DAG and model are all generic and are not specific to the studied organization. Hence, it should be straightforward to use the same model in other contexts, to see whether the findings can be replicated.

Reliability concerns whether or not the data and analysis are dependent on the specific researchers.

Most of the data in this article are collected from quantitative sources, and processed and visualized using standard statistical tools. We deliberately chose industry-standard tools (git, SonarQube) for data collection, to avoid bias specific to particular tools. We provide a replication package, including the full anonymized data set, that can be used to replicate our Bayesian model-building process.

Interpretation of the processed data runs the risk of introducing reliability threats. We strove to reduce these threats by interacting with the studied organization via mail and by setting up focus groups to discuss our findings. We elicited feedback from four teams, including architects, one line manager, and three core development teams. Anonymized transcripts and codes are available upon request.

7 Conclusions and Further Work

This paper introduces a model and replication package [180] designed to help visualize and understand how the team behavior related to introducing code clones varies across different components. The model is based on an industrial case study and has been validated through five focus-group sessions involving four teams.

The general pattern is that the number of introduced clones is related to both added and removed lines (i.e., the change size), and the existing complexity and

number of duplicates, following the *Broken Window Theory*, as described by Hunt et al. [43].

However, in some components, some teams are not affected by the existing complexity in the file. Furthermore, teams that have chosen ownership of a component appear to introduce fewer duplicates, relative to how they behave in other components. This suggests that (self-selected) ownership does matter in getting the teams to introduce fewer code clones.

An important factor to consider when analyzing data is *why* teams behave as they do. In the study, one team that *stood out*, relative to the other teams, was tasked to rearchitect the main business flow of the application, and this is a probable explanation for their behavior. Thus, the visualizations are important, but the *insights* that the visualizations lead to are probably more important (e.g., the importance of removing old code, once the new architecture is in place).

All interviewed teams stated that the visualizations of the clone introduction probabilities were useful for understanding clone introduction behavior across teams and repositories. However, most teams also felt that the model should be complemented with a corresponding visualization of the amount of removed clones. We developed the simplest possible model in this regard and found no significant outliers on a team-level stratification. In most repositories, most teams were acting similarly regarding clone removals.

This suggests that these two models could make a useful addition to a toolbox to govern the commons, according to the principles outlined by Dietz et al. [155]. We intend to explore the clone removal model in follow-up studies and invite others to do the same.

Understanding how much historical data to incorporate into models like these, and how often to resample them remains to be studied in follow-up studies.

In addition, replications in other organizations are needed to understand whether the observed behaviors differ in other contexts, and whether the models can be useful when analyzing other units of analysis (such as geographical sites or sub-organizations).

Paper IV

Learning Observability Tracing Through Experiential Learning

Anders Sundelin.
In: International Conference on Product-Focused Software Process Improvement (PROFES 2025). (December 2025).
Springer International Publishing, Cham, Switzerland.
Accepted, peer-reviewed conference contribution, reproduced with permission from Springer Nature.

Abstract

In a large-scale software development product development organization, we found that most developers, although experienced, were lacking architectural knowledge of the specific developed product.

As a remedy, we evaluated whether we could stimulate learning the product architecture by conducting training in how to use the product's distributed tracing platform, built on the OpenTelemetry standard and the open-source Jaeger Tracing visualization tool.

We planned and participated in a training event, where parts of the organization explored, using experiential learning, how to set up and use tracing to troubleshoot a realistic fault scenario we prepared. Respondents were asked to rate the tool according to the Technology Adoption Model (TAM), and responses were collected on Likert-type scales, analyzed, and summarized using a Bayesian workflow.

Even as tool usage post-training was low, respondents still had a positive attitude toward using the tool, valued the experiential training, and expressed a strong intent to use the tool for program comprehension.

1 Introduction

Even with comprehensive design documentation, becoming fluent in a codebase with millions of lines is difficult. Feature location involves static (source-code-based) and dynamic (logging, tracing) methods [188], but trace-based call-flow visualizations require setup and usage skills. With the advent of distributed microservices, where

one request passes through multiple services before returning, the problem of locating features is even more complex, and distributed tracing tools have emerged to facilitate program comprehension and fault localization [189].

Over six months, we conducted a three-cycle action research study in a large-scale software development organization, encompassing around 100 developers (15 teams) on multiple sites in two time zones (Europe and India). As the majority of developers were relatively new to the large code base (more than 12 MLOC Java code), dating back 15 years, we wanted to understand if a distributed tracing tool could be used to help developers understand the program flow. To test this theory, we planned and executed a learning session in which 19 developers, on two sites, learned how to set up and use the Jaeger tracing tool in a realistic problem scenario. Throughout the study, we collected feedback via surveys from the whole organization on how they perceived their competence and their experience with the training.

This paper is structured as follows: This section contains the overall problem formulation and the research questions. Section 2 contains background and related work, while Section 3 includes the research design, studied context, and methodology. Section 4 presents the results of our action cycles, which are condensed into learnings in Section 5, before we end the paper in Section 6 with conclusions.

1.1 Problem Statement and Research Questions

In our studied case, even as the product had supported tracing for over six months, few developers used it to learn about the product, which leads us to our problem formulation:

Can we use a tracing tool, like Jaeger, to increase architectural knowledge of a large product in a development organization where most developers lack specific product experience?

To investigate this problem, we formulated the following research questions:

(RQ1): To what extent do developers perceive that tracing tools help them understand the system architecture of a large-scale, multi-service system, most of which has been written by other developers?

(RQ2): How do developers perceive learning Jaeger tracing via experiential learning, using the system where they usually work?

(RQ3): To what extent do developers intend to continue using the tracing tools?

2 Background and Related Work

The Open-Source standard and framework OpenTelemetry[1] was formed to enable practical observability of microservices in an easy, universal, vendor-neutral, loosely coupled, and built-in way. To meet these goals, it provides standards and reference implementations that generate, collect, and export telemetry data in traces, metrics, and logs. Jaeger Tracing[2] is an open-source tool that produces web visualizations of collected OpenTelemetry data.

Gortney et al. [190] conducted a systematic mapping study and found that dynamic tracing, log analysis, and metrics collection were the three main practices used to visualize microservice architectures.

To assess the state of observability tools in industry, Li et al. [191] conducted an interview study, where 25 interviewees from ten companies of different sizes and in five domains were interviewed about their tracing practices and tools. All but the smallest of the studied companies (≈ 10 services and ≈ 20 kLOC) were employing, or considering using, some trace and log processing pipeline. The trace function was most commonly used for timeline and root cause analyses.

3 Research Design

3.1 Context

The organization consists of 15 cross-functional teams of $6 - 8$ developers, equally split between Europe and India. The product runs on Kubernetes in the cloud, and includes both in-house and externally sourced services, mostly Java (3.01 MLOC production, 2.22 MLOC unit tests, 6.49 MLOC functional tests), plus smaller Vue.JS (181 kLOC) and TypeScript (817 kLOC) front ends.

Half of the 15-year-old code base is older than five years, and only 30% is written by current employees, with the most experienced team contributing 8.8%, and average team contributions under 1%. A survey (55% response rate) shows both European and Indian developers average over ten years in general software development experience, but those in Europe have more product knowledge (average 4.2, max 15 years) than those in India (average 2.5, max 4 years).

3.2 Summary of Research Cycles

The organization asked us to investigate why OpenTelemetry and Jaeger were not used, even though architects saw them as valuable for exploring product architecture and unfamiliar features.

[1] https://opentelemetry.io/

[2] https://www.jaegertracing.io/

Cycle 1: Problem identification and formulation assessed the organization's development practices by collecting system statistics and administering a survey[3], asking respondents to rate their skills in six development areas, using a $0 - 100$ scale with descriptive anchors for five levels.
Cycle 2: Planning and executing the training session involved developing and delivering Jaeger training simultaneously at the European and Indian development centers. The session was one of six optional tracks during an organization-wide training day, with instructor-led, in-office participation to promote skill sharing and cross-team networking, while organizers allocated attendees based on budget and preferences.
Cycle 3: Follow-up of the training examined whether developers used Jaeger after the training by monitoring internal wiki page views. After one month, a follow-up survey was distributed to gather feedback on the training experience.

3.3 Data Collection and Analysis Methods

All our surveys used 7-level symmetric Likert scales with mandatory items and optional qualitative questions to assess agreement with survey statements. We applied Bayesian data analysis [27, 192] under the TAM framework to model constructs like usability, ease-of-use, and intent-to-use, comparing our treatment group to two control groups from unrelated training sessions. Models were ranked using PSIS-LOO [178] and built with the brms framework [181], interfacing with the Stan language, with full methodology available in the replication package [193].

We compared training sessions and sites by analyzing distributions, means, and credible intervals of the expected rating (defined as $\mathbb{E} \equiv \sum_{k=1}^{7} p(k) \times k$, where $p(k)$ is the probability of rating k) for the different constructs. We also summarized the qualitative feedback and presented this to the organization.

IV

4 Results and interpretation

We follow recommendations from Staron [194] and report the results per cycle.

4.1 Cycle 1: Problem identification and formulation

focused on assessing the current situation, from a tracing and observability perspective. We administered a survey to assess the self-rated competence level in the organization from various development perspectives. While the majority of the 55 respondents, in both Europe and India, are confident in writing and debugging Java code, fewer were confident in Vue.JS, and only four individuals (all based in Europe)

[3] Available in the replication package: `https://doi.org/10.5281/zenodo.15678405`

reported that they were confident in Jaeger tracing.

We presented these findings, together with the proposed training, to the organization, which agreed to continue the study.

4.2 Cycle 2: Planning and executing the training session

started with collecting background knowledge and planning an interactive training session. We wanted to involve participants and use a realistic scenario recognizable to developers. The cycle lasted six weeks, with regular follow-up meetings, comprising four phases: (i) planning and scoping the tutorial, (ii) preparing the setup instructions and developing the scenarios the students would face, (iii) validating the instructions (also including fixing issues found in existing documentation), and finally (iv) executing the training sessions, and collecting direct feedback.

Our planning phase focused on how to introduce a condition that would trigger a suitable anomaly (e.g., prolonged response time).

We developed one mandatory "seeded" problem and four "overflow" tasks, freely chosen by each team. To aid teams, we developed four hints for our problem, where the first hint indicated the product area, and the last one pointed to a complete functional test case reproducing the problem. Depending on the team's problem-solving progress, teachers distributed hints as text on printed paper. All teams needed the first hint, and one team used all four.

After one experienced developer formulated and implemented the seeded problem, including hints, we used two validation phases. First, one developer on each site independently followed and commented on the instructions. After adjustments and clarifications of instructions and hints, one junior developer performed final validation, including measuring the required time.

The training day comprised five one-day training sessions, from requirements engineering to security analysis. Participants ranked their preferences, and organizers allocated spots based on these choices and available capacity. All participants in the *Generative AI* and Vue.JS tracks had chosen these as their primary alternative. All architect and developer teams were represented in the *Jaeger tracing* track, but only 70% had ranked it as their primary choice.

After the training day, we collected opinions on the training from participants on both sites via a prepared survey. The tool-centric training sessions shared the same TAM-based structure of the questions, but had different characteristics:

Generative AI Participants in this track would learn principles behind Retrieval-Augmented Generation (RAG), use and partially develop a Python-based RAG solution in an existing Jupyter Notebook. Developers currently use none of these tools in their daily work.

Jaeger tracing This is our treatment group, where we explain how to set up Open-

Table 1: Respondents per training session, their reported years of professional experience, and the percentage of participants responding to the survey.

			Years of prof. experience					Response
	Site	N	Mean	σ	Median	Min	Max	Rate
GenAI	Europe	3	14.0	14.2	9	3	30	50%
	India	6	13.7	3.6	13	10	20	75%
Jaeger	Europe	10	17.5	8.7	17.5	8	30	83%
	India	9	12.9	3.4	14	5	17	100%
Vue.JS	Europe	5	14.8	10.2	19	0	25	83%
	India	5	7.8	4.0	10	3	12	71%

Telemetry and Jaeger in an experiential learning setting, and use these tools to solve a realistic, prepared problem in a special product branch.

Vue.JS Participants in this track would learn how to leverage the (for the product) new Vue.JS component pattern[4]. They were expected to learn the pattern and refactor existing views by developing and using common components in their normal development environment.

We asked the participants to rate the tools they used in their training from the angles of: (i) usability, (ii) ease-of-use, (iii) ease-of-access, and (iv) intent to use the tool in the future. Overall, the ratings were positive and similar for both sites, indicating that the training was well received.

Table 1 shows the number of respondents per session and site, and summary statistics on their (self-reported) professional developer experience. All sessions involved experienced professional developers, with a median experience ranging between 9 and 19 years. The response rate varied between 50% and 100%.

Figure 1a) shows the expected average usability rating, distribution, point estimate, and 95% credible interval. The figure shows that Vue.JS participants perceived the strongest usability, especially the Indian cohort. However, the European participants are *Quite Likely* to agree that Vue.JS components are usable. Both cohorts of Generative AI participants agree that the RAG tool is *Slightly Likely* to be usable, though the European cohort also tends towards *Neutral*. The Jaeger participants take the middle ground, rating the tool between *Quite Likely* and *Slightly Likely* to agree on its usability. The Indian cohort leans more towards *Slightly Likely*, though the difference is negligible.

Concerning ease-of-use, Figure 1b) show that the Vue.JS participants are more skeptical, particularly the European cohort, which rates ease-of-use *Slightly Likely*, while the Indian cohort leans more towards *Quite Likely*. Both the other tracks are very similar between sites. However, the Jaeger distribution is slightly narrower, and estimates, with 95% probability, that the expected value for ease-of-use lies between *Slightly Likely* and *Quite Likely*. Although the point estimate of the Generative AI

[4]https://vueschool.io/articles/vuejs-tutorials/5-component-design-patterns-to-boost-your-vue-js-applications/

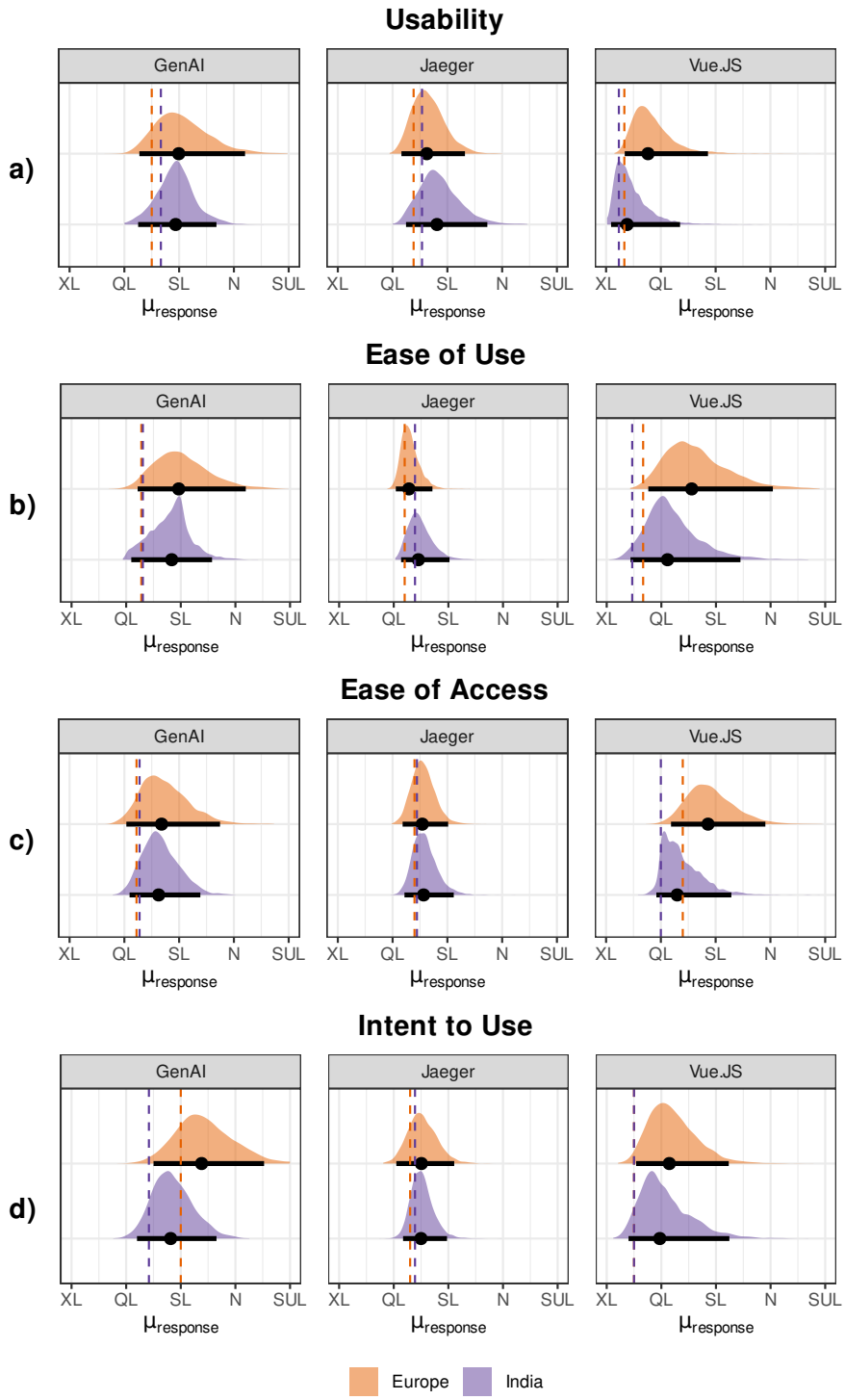


Figure 1: The distributions, by tool and training site, of the posterior expected rating for the four constructs: usability, ease-of-use, ease-of-access, and intent-to-use. Likert levels on the x-axis represent *Extremely Likely*, *Quite Likely*, *Slightly Likely*, *Neutral*, and *Slightly Unlikely* agreement, omitting the last two levels. The point estimate of the expected rating is shown as a dot, surrounded by the 95% credible interval. Sample (response) means are shown as dashed lines.

track is similar, the distribution is wider, meaning there is more uncertainty about where the expected value lies, particularly towards the *Neutral* level.

Figure 1c) shows the perception of how easy the tools are accessed in day-to-day use. All tools rate between *Quite Likely* and *Slightly Likely*, though the European Vue.JS participants lean closer towards *Slightly Likely*, while their Indian counterparts lean more towards *Quite Likely*.

Finally, as shown in Figure 1d), we measured the intent to use the tool in the future. Not surprisingly, the Vue.JS participants were most certain that they would use Vue components, with the expected value close to *Quite Likely* for both European and Indian cohorts. The Generative AI participants were more skeptical, particularly the European cohort, where the expected rating was estimated to lie between *Slightly Likely* and *Neutral*. The Indian cohort was more positive, though close to *Slightly Likely*. Both Jaeger groups were equally certain that they lean between *Slightly Likely* and *Quite Likely* to continue using the tracing tool in the future.

Fifteen of the 19 participants in the Jaeger training also provided qualitative responses. The thirteen positive responses range from: "Knowing that the tool exists" to the more explicit: "Save significant debugging time when you have visibility on complete code flow across components along with their respective process time." Five respondents found things to improve related to the training (e.g., "Docker logins should be fully working"), and six respondents commented on the Jaeger GUI (e.g., "I found the comparison feature difficult to understand").

We also surveyed the teachers, achieving between 50% and 100% response rates for the tool sessions. The most positive teachers were found in the Jaeger track, where teachers indicated "it went generally well," and "pretty much smooth." The primary Vue.JS track teacher indicated that the session probably had "too much freedom," and that "letting people choose what they want to do can be overwhelming." The teacher indicated that more structure and options would be imposed if the track were held again. The single (out of two) AI teacher respondent indicated that developers requested "more post-practices," indicating that the participants wanted more experience with the tool they partly developed. The Jaeger tracing and Generative AI teachers were *Extremely* or *Quite* agreeing that they had enough time and resources for preparations and that the tasks were adequate for the participants. In contrast, one Vue.JS teacher was only *Slightly* agreeing to those statements. However, all teachers enjoyed being mentors and agreed that participants learned something useful.

4.3 Cycle 3: Follow-up of the training

investigated Jaeger tool usage after the training by analyzing page views of the setup instructions on the corporate wiki[5]. Due to low page traffic, we conducted a follow-up survey one month later to assess how participants applied their learnings and iden-

[5] `https://www.atlassian.com/software/confluence`

tify barriers to tool adoption.

The survey had a low response rate, with only 23 replies, including six Jaeger training participants, none of whom reported using the tool. However, they expressed motivation to use tracing for learning program flow, and slightly agree that it would improve their understanding and development speed. All six respondents provide qualitative feedback (e.g., "I will use it when needing to trace something that is hard to track down in some other way").

4.4 Conclusion

Related to **RQ1**, respondents have a favorable view of the utility of a tracing toolchain like OpenTelemetry and Jaeger to aid program comprehension. The expected intent to use the tools is close to "*Quite Likely*" and our model is 95% sure that the intent to use is higher than "*Slightly Likely*". This is also confirmed by the qualitative data collected via the survey.

Related to **RQ2**, we find from qualitative survey responses that the respondents view the training session positively, though two respondents indicate they had to overcome troubles with the used environment. No respondents stated that the problem was too complex or that the hints gave away too much. The participants appear engaged, and they seem to appreciate the preparations.

Related to **RQ3**, web access logs indicate that the setup instructions were rarely accessed, and survey responses, although limited, state that no respondent had used the Jaeger tracing tool one month after training. However, all six respondents reported that they were positive and would use it when they needed to learn more about the product architecture. This might indicate that the tool's utility is real, but the need to use it is infrequent. We intend to follow up on how the tool is perceived and how the competence spreads in the organization.

5 Learnings

5.1 Contribution to Theory

Our study indicates that tracing tools, such as OpenTelemetry and Jaeger are valuable for visualizing system architecture and localizing features. However, despite developers recognizing their usefulness, usage remains limited six months after introduction and one month after training, suggesting these tools are used sparingly, mainly when exploring new domains or unfamiliar features.

The experiential learning event, featuring one structured track and four "open-ended" problems, was well received, with all respondents providing positive feedback on their learning. Teachers also noted that they enjoyed guiding participants and making new social connections.

5.2 Recommendations to Other Companies

We structure our recommendations according to the study phases:

Prepare problem(s) well in advance, and validate (using "fresh" developers) that the problem is attainable, but still challenging. If possible, frame tasks in the regular code base, not a synthetic toy problem. **Problems** can be "open" or "closed", and participants appreciated the "closed path" in the beginning, while enjoying the "open tasks" when they had learned more. To gain confidence in problem complexity, we used a representative developer to solve the "closed" problem, reviewing the number of hints needed and their detail.

Hints should be at different levels of detail, and we used paper slips rather than digital web pages. Teachers need to be prepared to disseminate these among the training groups, according to where they are in the learning journey.

6 Conclusions and Future Work

Our study adds to the findings by Li et al. [191] and Gortney et al. [190] that tracing can be an important tool to discover and learn about the product architecture. However, even as training participants perceived the tool and training positively, we found that these tools appear to be used relatively infrequently, possibly because developers rarely venture into unknown components.

We created a "Goldilocks problem," that was challenging yet attainable to help students focus on learning the tool chain, and used three developers to validate that the instructions were clear and unambiguous. Both students and teachers appreciated the initial structured approach, along with the four open-ended problems that allowed students to explore the tool freely.

Experiential learning [195] relies on attentive, self-directed students, so we provided hints of varying explicitness to participants as needed during the training. Survey responses show students found training tied to their daily work more usable, with higher scores for the Vue.JS component and tracing tool tracks than for Generative AI.

Despite developers expressing positive intent, one month after the training, none of our respondents had used the tracing tool in their daily work. This suggests that longer studies are needed to properly assess its adoption. We plan to continue surveying developers over time to monitor their work practices.

Paper V

Reducing Friction in Cloud Migration of Services

Anders Sundelin, Javier Gonzalez-Huerta, and Krzysztof Wnuk.
Submitted to The Journal of Systems and Software (In Practice track). First major revision resubmitted 29 July 2025.

Abstract

Public cloud services are integral to modern software development, offering scalability and flexibility to organizations. Based on customer requests, a large product development organization considered migrating the microservice-based product deployments of a large customer to a public cloud provider.

We conducted an exploratory single-case study, utilizing quantitative and qualitative data analysis to understand how and why deployment costs would change when transitioning the product from a private to a public cloud environment while preserving the software architecture. We also isolated the major factors driving the changes in deployment costs.

We found that switching to the customer-chosen public cloud provider would increase costs by up to 50%, even when sharing some resources between deployments, and limiting the use of expensive cloud services such as security log analyzers. A large part of the cost was related to the sizing and license costs of the existing relational database, which was running on Virtual Machines in the cloud. We also found that existing system integrators, using the product via its API, were likely to use the product inefficiently, in many cases causing at least 10% more load to the system than needed.

From a deployment cost perspective, successful migration to a public cloud requires considering the entire system architecture, including services like relational databases, value-added cloud services, and enabled product features. Our study highlights the importance of leveraging end-to-end usage data to assess and manage these cost drivers effectively, especially in environments with elastic costs, such as public cloud deployments.

1 Introduction

Developing large software products and services brings many challenges to software development organizations. These challenges include (i) splitting the product into behaviorally isolated components, allowing individual teams to be in charge of their maintenance and evolution (e.g., by using Service-Oriented Architecture or microservices), and (ii) provisioning just enough compute and storage resources in time to give clients acceptable quality of service. In the early 2010s, the eruption of the cloud paradigm [196], with its promise of elasticity and the ability to scale software and hardware systems to match the customers' actual demands, caused many development organizations to consider deploying their software solutions to the cloud.

However, migrating large monoliths to cloud-deployed microservices can pose challenges and risks [197, 198]. Taibi et al. [199] found that 38% of interview participants manage migration risks by stepwise migration, whereby new or changed features are introduced as separate microservices around the existing monolith, which is gradually replaced. This is sometimes referred to as "the Strangler Fig Pattern" [200]. The migrated software could then be deployed on virtual machines running on bought hardware, functioning like a private cloud. In later transformation steps, the organization could migrate services to a public cloud provider, where resources are bought on demand.

Deploying on a public cloud might require adapting the design of some services to the characteristics of such a platform. A service that executes on virtual machines running on owned compute hardware might tolerate some inefficiencies as long as the total demand for the service does not exceed the existing hardware's capacity. However, migrating such a service to a pay-per-use cloud provider, where the organization pays per used virtual CPU core and possibly per database access, will expose these inefficiencies as unnecessary costs. If brought to the extreme, changing deployment options might impact the whole business case for the service [201].

In today's highly competitive software market, successful organizations cannot risk taking decisions solely based on expert opinions but need to systematically collect and process data from their systems[1] to make informed decisions related to their processes and products [202]. Reverse design [203] is one way in which usage data can inform decisions about the design and architecture of software solutions.

This paper presents a case study that examines how deployment cost efficiency changes when migrating a large-scale software product to a different deployment platform. The Product Development Organization (PDO) and a large customer considered migrating product deployments to a public cloud service provider, and wanted to understand the economic consequences of such a migration. To assess these consequences, the organization launched a proof-of-concept project where the complete service (including operations and maintenance functions) was migrated to a large

[1] Such data collection and processing must comply both with relevant legislation and existing customer terms and conditions.

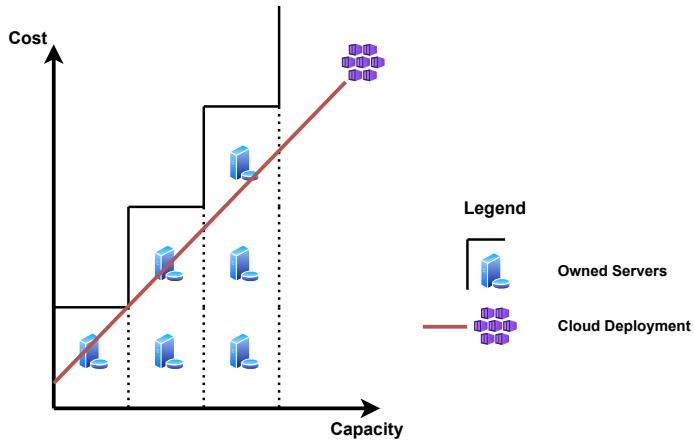


Figure 1: The theory behind cloud economy [196]: With owned hardware, users always pay for ``the next server'', no matter how little extra capacity you need (black, zig-zag line), and peak usage will determine the entire system cost. With a cloud solution, you should be able to ``pay only what you actually use'' (red line).

public cloud provider.

The customer expected to reduce deployment costs, as reported by Villamizar et al., Villamizar et al. [204, 205], but the project results contradicted these assumptions. To understand why this was the case and suggest possible remedies, we analyzed 90 days of product usage data and interviewed six project members with different roles. We validated our findings through a focus group with eight participants having different roles in the organization. To the best of our knowledge, this paper is the first to systematically describe the cost implications of migrating a large legacy product to a public cloud provider.

The remainder of the paper is structured as follows: in Section 2, we report the background and related work, and in Section 3, we describe the case and our research methodology. Section 4 contains our results, and in Section 5, we discuss these. Finally, in Section 6, we conclude the paper and outline future work.

2 Background and Related Work

Figure 1 illustrates the theory behind the cloud economy, as described by Armbrust et al. [196]. Owning dedicated servers (illustrated by the black line) requires an up-front investment in hardware based on projected usage and plans for expansions well in advance (to manage hardware delivery and installation lead times). When using a cloud provider (illustrated by the red line), the hardware and infrastructure work is outsourced to the cloud provider, who would manage these resources for several systems. The foundation for sharing resources is the capability to divide large

hardware resources (e.g., server-class computers) into smaller, shareable pieces. The most common strategy relies on virtualization software, which divides one server into many Virtual Machines (VMs).

We follow the terminology of Armbrust et al. [196] and use the term *public cloud* to denote compute (and storage) services being made available to the general public via some on-demand subscription service, called *utility computing*. In contrast, a *private cloud* refers to a data center that might use cloud technologies, though the services are only available to a select set of users, who typically share some organizational belonging (such as departments or subsidiary companies), that own and operate the underlying hardware and network services. According to Armbrust et al. [196], utility computing is preferable to a private cloud in either of the following cases: i) the demand for the service *varies over time*, or ii) the demand is *unknown in advance*, or iii) the work can be *highly parallelized*, as is the case when processing independent work batches.

The three main drivers that motivate legacy software to cloud migration are maintainability [206, 207], scalability, and flexibility [208]. Andrikopoulos et al. [209] defines four different migration types of increasing ambition level: (i) replacing individual architectural components (e.g., a database) with a corresponding cloud offering, (ii) partially migrating some of the application functionality to the cloud, (iii) migrating the application's whole software stack, with the architecture intact (sometimes referred to as "lift-and-shift"), and (iv) rewriting the application as a composition of services running in the cloud. The "lift-and-shift" migration strategy does not benefit the user much, as it is hard to scale, and because of the comparatively long startup time of large VMs [210].

Chen et al. [211] outlines two cloud usage scenarios: the "unified client" model, where applications are accessed by a single customer or a small group, and the "multi-client" model, where diverse third parties access shared resources, similar to public content distribution. They suggest that for "unified clients," outsourcing is cost-effective when computation outweighs communication. The multi-client setting is generally better for mid-sized enterprises due to the high cost of networking.

Many companies that migrate complex legacy systems prefer to rewrite them using current technologies over splitting up existing code bases [206]. To take advantage of cloud deployment, applications should have the following five properties: Isolated state, Distribution, Elasticity, Automated management, and Loose coupling (IDEAL) [212].

The financial and business aspects of cloud migration were studied by several authors. Among them, Walker [213] compared the cost of purchasing a cluster with the cost of leasing CPU time and storage capacity from an open market by considering the aggregated cost of the resources from a Net Present Value (NPV) perspective, taken over the expected lifetime of the cluster (typically 3-5 years). Tak et al. [201] also used the NPV approach but also considers networking costs, arguing that cloud migration is appealing for small or stagnant businesses. They found that partitioning

components is expensive due to the increased networking costs, and that a mix of in-house and cloud deployment can be helpful for certain applications (e.g., to scale out during high traffic).

Konstantinou et al. [214] compared the Total Cost of Ownership (TCO) of a private cloud (StratusLab) with that of a commercial provider. Their findings indicate that even small-scale private clouds can become more cost-effective within 2–3 years, well before hardware reaches its end-of-life. They also highlight that compute-intensive applications are more economical when organizations maintain full control over the virtualization layer. Villamizar et al. [204, 205] report significant infrastructure cost reductions when transitioning from monolithic architectures to microservices deployed on public cloud platforms like AWS Lambda. By implementing three versions of a reader and a writer service, they demonstrate cost savings of up to 77%, measured per million requests.

Various approaches have been proposed for auto-scaling cloud resources. Mao et al. [215] describes how cloud providers enable users to define triggers to scale up or down the cloud resources based on some defined metric (such as CPU utilization, disk or bandwidth usage, etc.) They postulate using simulations to assess cloud budgets properly. Boza et al. [216] describe how to use a set of $M(t)/M/*$ queuing theory models to predict load (hence also the cloud budget) for a system over time. In such a model, arrivals are governed by a Poisson process where the arrivals at time t are a function of t: $\lambda(t)$. With reasonable resolution (e.g., hours), this allows for modeling situations where requests vary by the time of day, as is often the case for services that interact with end users in the same time zones. In another study, Zhu et al. [217] represents the system behavior via an auto-regressive-moving-average with exogenous inputs (ARMAX) model, supporting both a fixed time limit and a resource budget for the task at hand. They develop a controller that uses the ARMAX model and a Proportional-Integral (PI) control, combined with a reinforcement learning component to minimize costs, subject to time and budget constraints, achieving 200% improvement over a static provisioning scheme, with a controller overhead of less than 10%.

Cloud cost calculations [196, 213] and cloud migration of services [206, 208] have been researched before. However, to our knowledge, no study has analyzed product usage data when migrating a large (millions of lines of code) legacy system with millions of end-users to a public cloud provider. Likewise, we have not seen any studies on the actual bottom-line impact of autoscaling services in such legacy systems. Thus, we believe this paper can provide feedback to the research community on the problems faced by practitioners tasked to migrate existing large-scale software systems to the big public cloud providers.

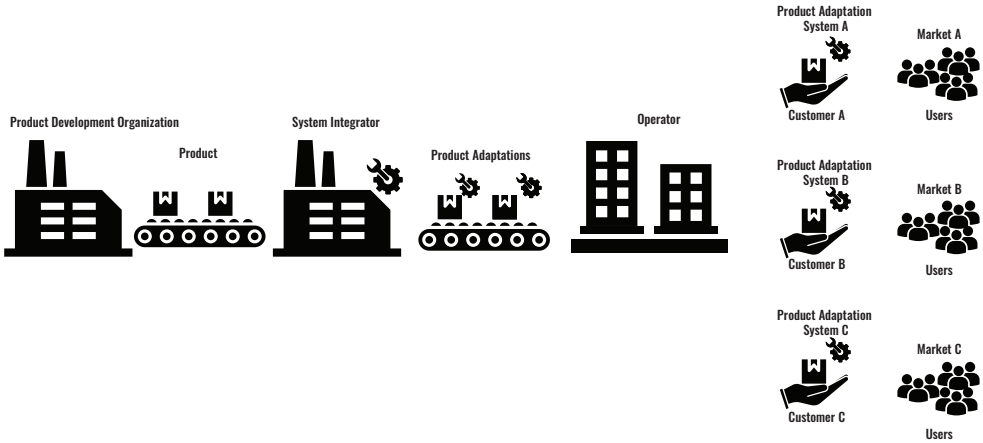


Figure 2: Schematic view of the product and its stakeholders. The Product is developed by the Product Development Organization, customized by System Integrators, made available by Operator(s) who own the product, and realized as installed Systems A, B, C, etc., which are used by Users in different Markets. In our case, the individual Systems are operated by different Customers (A, B, C, etc.), who are owned by a common Operator.

3 Research Methodology

In this paper, we study the potential consequences of a strategic decision to migrate a large, existing product offering to a particular public cloud provider. To this end, we formulate the following research questions:

(RQ1): How does the projected cost for the used system resources change when changing deployment from owned hardware to public cloud?

(RQ2): What factors drive the change in cost, and what is their estimated influence?

(RQ3): To what extent can we use service usage data to explain resource usage in deployments?

To answer these research questions, we conducted an exploratory single-case study following the guidelines by Runeson et al. [40]. The goal of the case study, described using the GQM framework [31] is: To analyze the usage of the (fully integrated) software product by its end-users ***from the perspective of*** the organization developing the software product, ***with the objective*** of understanding the impact of its architecture on deployment cost ***in the context of*** a potential public cloud migration.

V

3.1 The Case and Unit of Analysis

The case under analysis, shown in Figure 2, is a global FinTech software product developed by a Product Development Organization (PDO) that exposes financial services via an Application Programming Interface (API). The product contains some

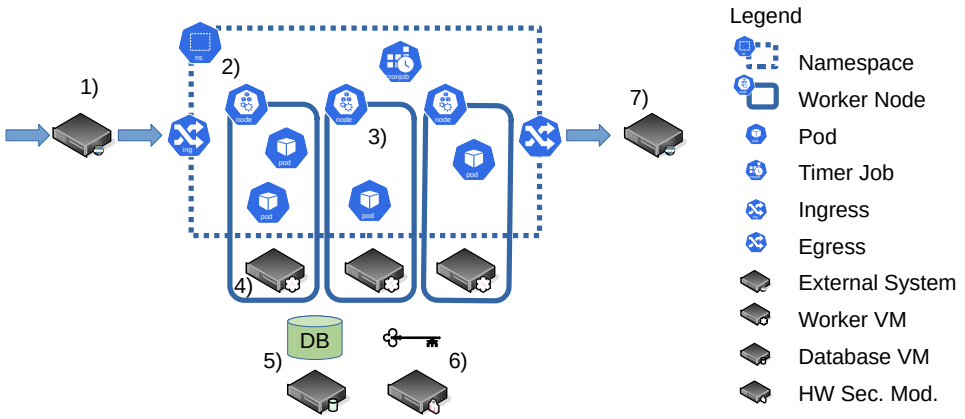


Figure 3: High-level deployment view mixed with Component and Connector view of the current product. Incoming requests (1) are processed by services running as pods inside a Kubernetes namespace (2). Pods execute on worker nodes (3), which are realized via Virtual Machines (VMs, 4) on top of owned hardware. Outside the namespace, the database service (5) runs on separate VMs, and cryptographic keys are stored in a Hardware Security Module (HSM, 6). Some requests cause interactions with external systems (7).

core functions, such as Graphical User Interfaces, but the API allows System Integrators to build end-user Use Cases and adaptations unique to each product installation. At present, the Product Development Organization sells the product as an integrated software-hardware stack, where each Customer operating the service (Operator) owns the whole solution, including hardware, reselling the financial services to end-users (Users), who typically pay agreed fees (depending on the Use Case) for the usage of the system. The Users are individual consumers, small merchants, and large companies, such as banks. In each market, the product is adapted and customized by System Integrators, companies outside of the PDO, that configure and build the complete end-to-end Use Cases, via the Software Product API and their own code.

Due to a change of strategy, both the Product Development Organization and the largest Operator wanted to change from the capital-heavy "owning" model [213] to a more elastic deployment, using a public cloud provider. To evaluate the consequences of such a shift, the Product Development Organization developed a deployment solution using a public cloud provider selected by the Operator. Following the conclusion of the evaluation project, we conducted this case study to elicit recommendations for future adjustments to the software product and the organizational setup of the cloud economy.

Figure 3 shows a high-level picture of the product. A request (1) is sent to a Kubernetes service running in a namespace (2). In the namespace, each service is implemented by one or more pods running on worker nodes (3). A minimal configuration contains about 20 Java-based microservices (composed of $\approx$ 3 MLOC), plus over 80 third-party services. Each worker node is realized as a Virtual Machine (VM, 4),

running on top of owned physical hardware, using static configuration (e.g., memory and vCPU characteristics). Some application services, such as the audit log and the main application relational database (RDBMS), are also running as separate VMs (5) outside the Kubernetes namespace. A separate, physical, Hardware Security Module (HSM, 6) manages and protects the cryptographic keys used by the product. In a few special use cases, the application service communicates with an external system (7) before returning a response. Timers trigger some use cases (such as reports or maintenance tasks), but most services are initiated by incoming requests from external actors.

In production systems, the product is deployed redundantly, where each microservice exists in multiple copies within the same namespace. For higher availability, each deployment might be replicated in multiple namespaces, interconnected via public or private networks, running on geographically separated hardware.

Overall, the architecture of the system follows the following structure:

i) The system exposes operations (services) of different types (320 in total), where each operation performs a well-defined task. Most operations are accessed through an HTTP REST API [218].

ii) Some operations access services external to the system, waiting for these services to respond before returning a response.

iii) Generally, external communication occurs at well-defined places, from a defined subset of operations, meaning that incoming and outgoing requests can be correlated using their operation types.

The product has existed for over ten years and was first deployed, as a monolith, on Linux, running on dedicated physical computers (referred to as "bare metal" deployment). Over time, as the product has grown in capabilities, size, complexity, and number of micro-services, its architecture has evolved, first by adapting to virtualization (running on VMs), then containerization, by adopting the Kubernetes deployment style, as advocated by Leymann et al. [210]. Currently, it is deployed in over 20 different markets across several continents.

The context of this case study can be summarized as: (i) the Operator initiating the change to public cloud deployment deployed the product in several markets (countries), having sub-companies (Customer A, B, C, etc.) in each country. (ii) the Operator unilaterally selected deployment to a particular public cloud provider, (iii) the existing Operator deployments already shared resources (such as hardware, but also some expensive services), and (iv) the product kept its original architecture, based on an industry-standard relational database (RDBMS), which scaled only vertically by adding more CPU, memory, and storage to an existing physical or virtual computer.

Table 1: Interviewees, how many years they have worked (in general and in the product), and overall job description relevant to the cloud migration project.

Role	Experience (y) Work	Product	Major tasks
Cloud Architect	17	12	Conceptualizing, dimensioning
Cloud Developer	16	13	Developing cloud solution
Operations Eng.	18	8	Implementing cloud solution
Project Manager	27	14	Managing people resources
Requirements Eng.	30	5	Eliciting requirements
Sales Architect	12	12	Technical customer interactions

3.2 Planning

We selected the case based on convenience and availability due to the existing partnerships between the studied organization and Blekinge Institute of Technology.

3.2.1 Methods for Data Collection

We employed a mixed-methods case study, collecting both quantitative and qualitative data. In the quantitative part, we employed archival analysis [33], collecting documents and budgets from the migration project and 90 days of aggregated usage data from two of the largest system deployments. In the qualitative part, we employed semi-structured interviews with different roles in the development organization and a focus group to validate our conclusions and give feedback to the studied organization. We used investigator triangulation when conducting and analyzing interviews and focus groups, and data triangulation to confirm conclusions from a qualitative and a quantitative angle.

We interviewed six PDO employees working in different roles, selected using convenience sampling. Details about the participants are available in Table 1. Through time-boxed (30-minute) interviews, we tried to understand (i) the drivers and the context of the migration to the public cloud provider, (ii) their subjective experiences during the project, and (iii) their perceptions about using usage data to assess the deployment strategy of the system.

Having explained our data anonymization and reporting protocol, we recorded and transcribed the interviews using Microsoft Teams©. The automatic transcriptions were then revised and anonymized by one of the researchers in the research team, and the resulting transcription was presented to the interviewee for feedback and possible clarifications. The anonymized transcripts are available upon request.

To validate our findings, we also conducted a focus group session with eight participants (including roles such as Technical Sales, System Architect, Cloud Architect, Developer, Tester, and Operations/DevOps Engineer), who commented on our conclusions. The focus group was also recorded, transcribed, and anonymized.

3.2.2 Methods for Data Analysis

The data analysis has been iterative, although some activities have also been intertwined. Quantitative data was analyzed using visualizations, descriptive statistics, and correlation and regression models. Qualitative data has been analyzed using open coding [219], and transcripts and coding scheme are available upon request. As an example, the quote "*It was not so clear what they were trying to solve [with the move to cloud deployment].*" was coded with [Weak and unclear requirements], which, together with the code [Limits of lift-and-shift], also was applied to the quote (from a different respondent): "*We were just thinking of how we can run [the product] in the cloud. We did not think about the impact […] in terms of cost or in terms of operational efficiency.*"

3.3 Generalizability

Although this paper contains data and findings from a particular system developed and deployed by specific organizations, we have strived to describe the surrounding context in adequate detail while still respecting the need for confidentiality required by commercial enterprises. As Flyvbjerg [42] states, cases play an essential role in human learning, and it is, in fact, possible to learn from a single case.

The characteristics and context we describe are far from unique, and we hope the case description enables others to judge whether our findings apply to other systems. However, we cannot claim generalizability across all possible organizations, systems, or cloud providers, although we believe the study brings interesting insights and lessons for practitioners and researchers.

4 Results

4.1 Projected cost change due to change of deployment platform (RQ1)

The first public cloud estimate turned out to be four times more expensive than the existing private cloud due to four key reasons: (i) using a single estimate (for the smallest deployment) as the base of the cost estimation for the entire Operator group of companies, (ii) the lack of cost requirements in the initial specification, (iii) the lack of relevant feedback loops—i.e., the requirement to come up with the complete five-year cost before ever deploying anything to the cloud, and (iv) the failure to consider that some—crucial, but lightly used—cloud services could be shared among all Operator companies.

The Project Manager emphasized the intent to replace existing Open-Source bundled services (e.g., log visualization, intrusion detection and other services not directly tied to product-specific features) with commercial cloud services: "*If it is in*

the cloud, as a service from the Cloud Provider, or in the Marketplace, then we will use that service. […] OK, so then we can finally get a nice security monitoring tool, instead of Tripwire."

The Cloud Developer stated the lack of feedback as a prime reason for the increased cost estimate: "*When you deploy a solution to the cloud, it takes a while to come to the optimal solution[…] You can never get an optimal level on paper[…] You need some time to get some feedback from live data to make an application.*" He also stated the lack of feedback as a hindrance to choosing suitable external services: "*Log analytics is responsible for collecting all the logs from the infrastructure. It is a very expensive service in the Cloud Provider. Before I have tested with a real live installation, how can I determine how much storage I should use for it?*"

Sharing Used Services

Once the initial cost estimate was realized, the Sales Architect started to get more involved: "*My first reaction was, no one has looked into this architecture from the point of optimization. I went into challenging the teams[…] 'OK, why do we need to ingest all that data into [the Log Analytics Service] in Cloud-Provider"*?" This intervention reduced the estimated deployment cost considerably—from four times (300%), down to 50% extra cost, compared to the existing "private cloud" solution.

Thus, while the initial cost estimate was unrealistic, even after optimizations suggested by the Sales Architect were considered, the deployment cost for the system in the public cloud would still be 50% higher than the existing private cloud costs.

4.2 Factors influencing public cloud deployment costs (RQ2)

Based on feedback from the Cloud Developer and the Operations team, we also looked into how the required resources varied over time to estimate whether the used resources could be sized accordingly.

Autoscaling Worker Nodes

Figure 4 shows, using a grouped boxplot, how the number of incoming requests per second varies throughout the day for a representative deployment. The figure shows a flat daytime load, followed by a stable peak, starting around 17:00 and receding around 21:00, which reaches about two-thirds (median $\approx \frac{3000}{1800}$) more than the daytime load. During night-time, incoming requests are much lower than during daytime, with traffic increasing around 06:00. The figure also shows a tentative adjustment to the needed resources. The observed daily variability is consistent with feedback from the Cloud Developer: "*At night, we hardly have any traffic. During the day, we have plenty of traffic, but we are paying for the same resources day and night.*"

Based on the existing traffic pattern, the Cloud Developer suggested adding auto-scaling to the product to adjust the deployment size and needed resources accord-

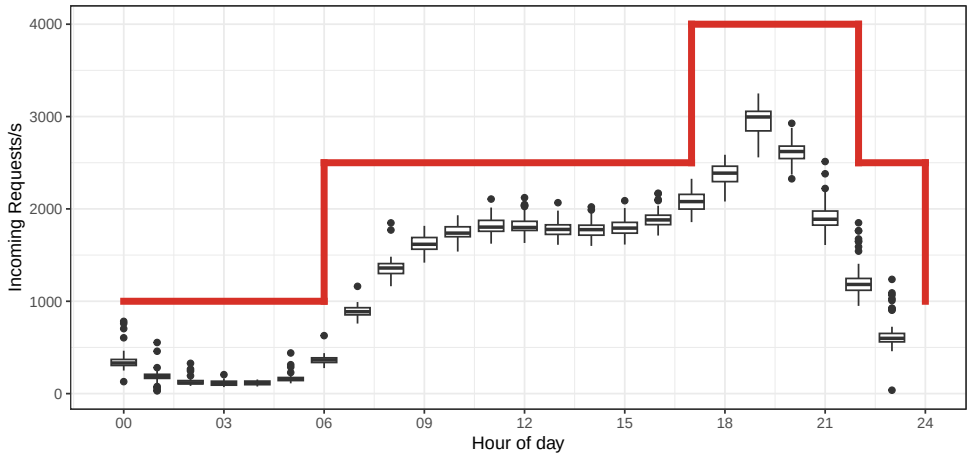


Figure 4: A grouped boxplot showing how the number of incoming requests per second varies by the hour of day, measured during 81 days in late 2023. Daytime traffic is flat, followed by an evening peak and drop-off after midnight. The red line indicates a possible adaptation of needed resources to the incoming load.

ing to incoming load. Converting the red line in Figure 4 into needed resources (assuming the maximum resources required corresponds to the peak usage—i.e., 4000 requests per second), finds that, based on this traffic pattern, the required resources over time can be expressed as a fraction of the peak resource usage, as per expression V.1.

$$\frac{6 \cdot \frac{1000}{4000} + 11 \cdot \frac{2500}{4000} + 5 + 2 * \frac{2500}{4000}}{24} \approx 61\% \tag{V.1}$$

As the dimensioning factor is the peak resource usage, in a private-cloud scenario—where the hardware cost is fixed—it makes little sense to reduce resources via autoscaling.

The chosen public cloud provider charges resource usage per started VM, regardless of the vCPU load, meaning that the number of Kubernetes worker nodes are the only resources that could be autoscaled. For redundancy reasons, all deployments require at least two worker nodes.

Cloud Cost Break-Down

We analyzed the Bill of Materials[2] (BoM) used by the project, which specifies contents and their costs (initial investment plus five-year total license and maintenance costs) for Small, Medium, and Large deployments. Costs for the three alternative solutions are estimated for each deployment size:

Unoptimized refers to the initial estimate, where each deployment is separate, with no shared resources. This solution also heavily uses the AI-based log analytics

[2]Anonymized data found in: `https://tinyurl.com/Sundelin-Cloud-Migration`

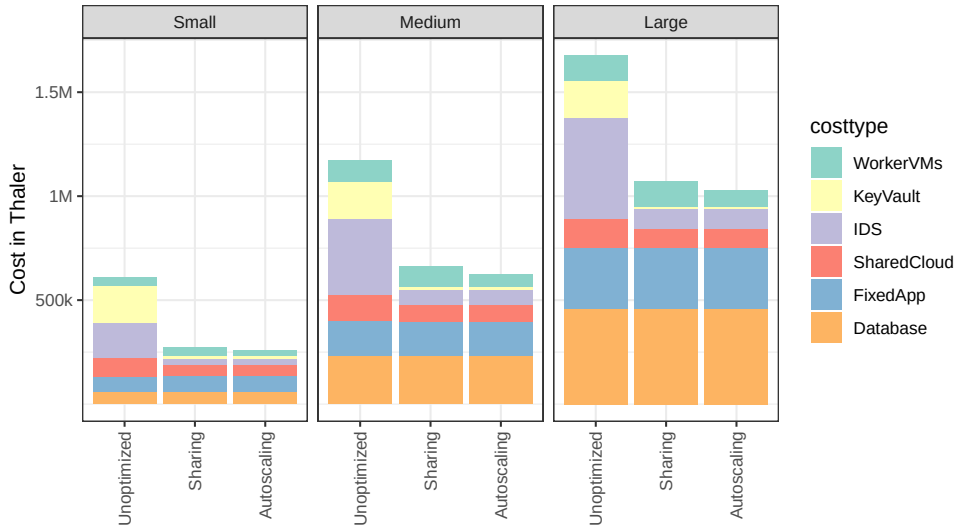


Figure 5: Estimated costs, per cost type and deployment size and type. Costs are expressed in the fictional *Thaler* currency, having a fixed, hidden, exchange rate to USD.
Cost types are: **WorkerVMs** the Kubernetes worker nodes (Virtual Machines); **KeyVault** the Cloud Provider Key Management solution for secure key storage; **IDS** the Cloud Provider AI-based Intrusion Detection System; **SharedCloud** services potentially shared across deployments (e.g., firewalls, observability tools, syslog infrastructure, and the associated network connectivity); **FixedApp** services needed by each deployment (e.g., storage, backup, network connectivity); and **Database** license and support costs.

solution from the Cloud Provider.

Sharing refers to the estimate in which some expensive cloud resources are shared among the deployments, and logs sent to the AI-based log analytics solution were trimmed.

Autoscaling contains all improvements from the **Sharing** solutions, but also adds dynamic scaling of the Virtual Machines used as Kubernetes worker nodes.

The original BoM specified the costs in US dollars. However, for business confidentiality reasons, we converted the costs to a fictional currency, called the *Thaler* ($\mathcal{T}$), which converts to USD via a fixed and hidden exchange rate.

Figure 5 contains the cost per cost type for the three deployment sizes and solutions. The *Unoptimized* costs grow from $\approx 609\ \text{k}\mathcal{T}$ for the Small deployment, to $\approx 1.68\ \text{M}\mathcal{T}$ for the *Large*. The figure also shows the dominant cost of the Intrusion Detection System (IDS) and the Key Vault for all three deployment sizes. The relative database cost grows with the deployment size, more than doubling from Small ($\approx 10\%$) to Large ($> 25\%$) deployments.

Table 2 contains the total and the worker node costs expressed in *Thaler*, as well as the expected savings realized when adding *Autoscaling* to the *Sharing* deployment variant. The table shows that we could expect total savings of $\approx 4 - 6\%$, if worker nodes were scaled according to the traffic pattern shown in Figure 4. The Cloud

Table 2: Total and worker node cost for three deployment variants in the fictional *Thaler* currency, and the savings realized by adding autoscaling, as a percentage of the total **Sharing** cost.

	Total cost in $k\mathcal{T}$		Worker node cost		Relative
Size	**Sharing**	**Autoscaling**	**Sharing**	**Autoscaling**	**Savings**
Small	272	257	42	26	5.8%
Medium	664	627	102	65	5.5%
Large	1070	1026	122	77	4.2%

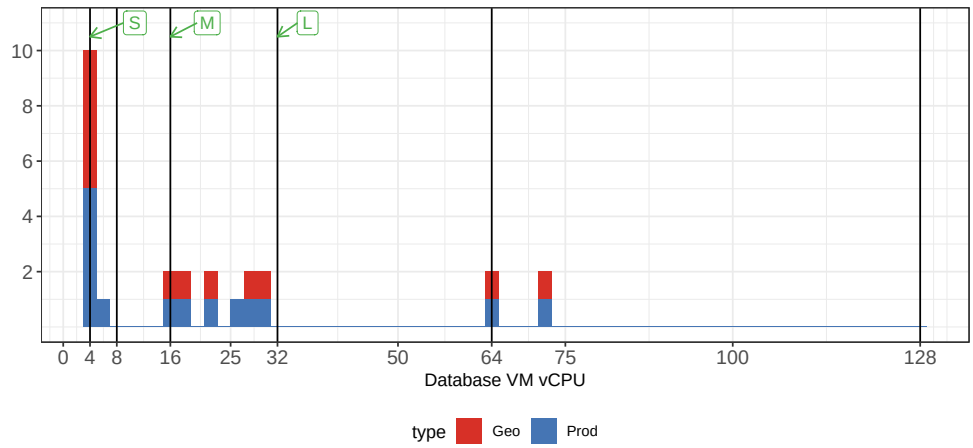


Figure 6: Number of deployments with a particular number of vCPUs in the database Virtual Machines. Five deployments have 4 vCPUs, and the biggest deployment has 72. All but two deployments are geographically redundant. The sizes of the database VM in Small, Medium, and Large deployments are marked with S, M, and L. The available public cloud VM sizes are marked with black vertical lines.

Architect expected more savings to materialize from autoscaling: "*We need to work more on that concept of autoscaling, and actually use exactly what you need at this moment.*".

In contrast, the database service, running on statically configured Virtual Machines (VMs), forms a much larger part of the overall cost, and its importance increases as the system grows. The reasons for the cost growth are both that the Cloud Provider provides VMs of fixed, exponentially spaced sizes (i.e., 4, 8, 16,…vCPU), and that the license costs for the database are proportional to the number of available virtual CPUs on the database VMs.

Figure 6 shows the size of the database VMs used by some deployments (both production and geographically redundant systems). The five smallest systems require only four vCPUs, but the largest requires 72, implying that the corresponding available public cloud VM would need 128 vCPUs. Running the database on such a VM would increase license costs by 78% for the largest deployment. Taken together, should all these systems deploy on fixed-size public cloud VMs (whose available sizes are indicated by the vertical lines in the figure), the license costs would increase by 31%. The exponential sizing of available VMs implies avoiding database expansions will significantly impact operating costs, especially for larger systems.

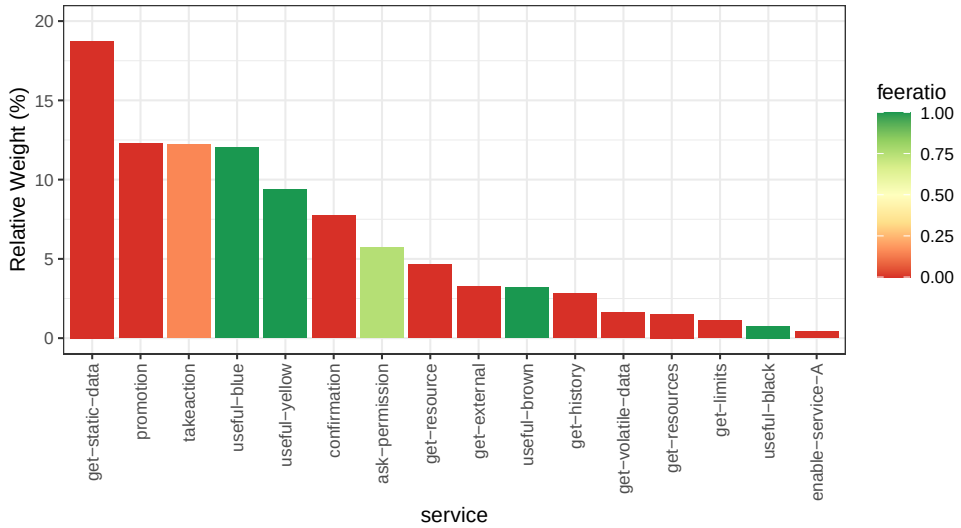


Figure 7: Relative resource weight of the top 16 (out of 320) different services being used, comprising 97.5% of the used resources. The color indicates the extent to which the service was generating revenue.

We discussed the database solution and its cost with our interviewees, and it is clear that: (i) the product requires a particular brand of Database; (ii) while the chosen Cloud Vendor provides other databases "as-a-Service," none of these databases are fully compatible with the required brand: "*No one is using Database in Cloud-Provider. Everyone is using other databases like SQL Server or Postgres, which are available natively there.*"; (iii) using a third-party Database service would introduce unwanted dependencies on other vendors (e.g., in troubleshooting, fault isolation, and other contractual obligations); (iv) the Cloud-Provider recommends using Virtual Machines to provide the required Database: "*They don't have Database 'as-a-Service.' But they have like a recipe how to do it, and then you have to do it yourself.*"

4.3 Using service data to explain resource usage (RQ3)

Given that the database service constituted a significant factor driving public cloud deployment costs, we decided to investigate *how* the product was being used for two of the major deployments. With accurate usage data, it would be possible to pinpoint which parts of the product contributed to the cost and relate these to the revenue-generating parts.

Budgeting Resources

To assess resource usage by the different services, we define the metric $\mathcal{W}[S]$, called *weight* for service S, calculated over some time of interest:

$$\mathcal{W}[S] = t_i[S] \cdot n_i[S] - t_o[S] \cdot n_o[S] \tag{V.2}$$

Using equation V.2, we define $\mathcal{W}[S]$ in terms of: $t_i[S]$, the average service time (latency) for service S; $n_i[S]$, the number of service invocations; $t_o[S]$, the average latency of external services used by service S; and $n_o[S]$, the number of requests to external services (≥ 0) made by service S. Importantly, in this model, a database request is considered an internal service, as the database runs as an integrated part of the system.

The current product allows clients unlimited requests without considering the cost of each used service. Figure 7 shows the most resource-intensive services in the system, ranked by relative weight in relation to the combined weight of all services. The figure shows that only four of the top 16 services are directly generating revenue. The most used service ("get-static-data") is reading static data, which would have been easy to cache.

The services "takeaction" and "askpermission" only partially generate revenue. In "takeaction" about 87.5% of requests are for free, while "askpermission" requests may be declined by end users. In total, services using about 30% of resources generate direct revenue, considering that two services only partially do so. Based on usage data analysis of two systems, we find that a moderately sized cache would reduce the number of requests to the most used service "get-static-data" by between 10% and 50%, which corresponds to a 2%-10% reduction of overall resource usage.

Focusing on Database Statements

With moderate effort, it is possible to look into what statements the product is executing towards the database, and given application traces and source code, these statements can be attributed to services.

Figure 8 shows the top 20 SQL statements for the two largest product deployments. We found five statements related to the above-mentioned "get-static-data" service, and could be cut between 10% and 50% if caching had been used. The second most used statement was related to a feature neither customer used, but the product assumed to be active due to misconfiguration.

Summary---Usage Data to Explain Resource Usage (RQ3)

We can summarize the results with a quote from the Sales Architect: "*[Solution Integrators] look at our API specification and their focus is to get the functionality [the Use Cases] to work. Their focus is not to develop something performance-oriented. I have never seen someone questioning 'this design, is it efficient from a performance perspective?*'" As shown in Figure 6, this approach was adequate as long as the system was executing on fixed, owned, and paid-for hardware. Selecting the chosen Cloud Provider, with its fixed-size VMs, shows that the costs become painfully obvious, at least for the larger systems.

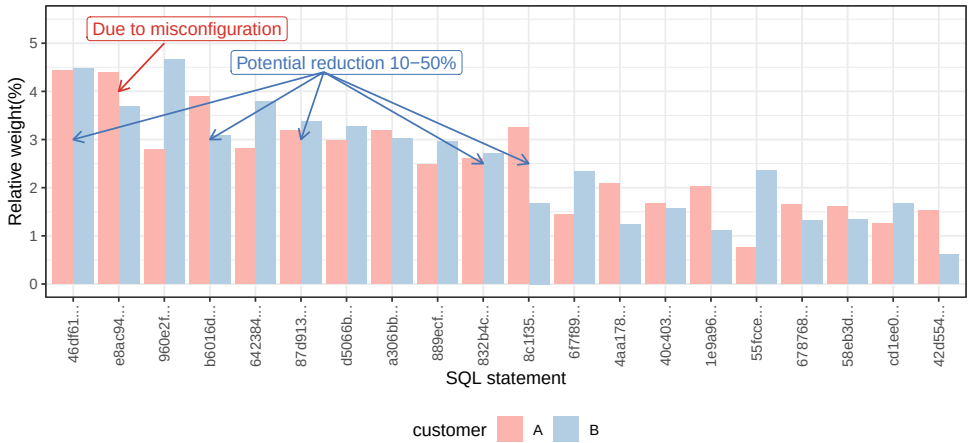


Figure 8: Relative weights for the top SQL statements for the two largest deployments. A misconfiguration (an unused, partially enabled feature) causes the second heaviest statement, and five of the eleven heaviest could have been reduced by 10%-50% if ``get-static-data'' calls had been cached. Statements are arranged by average relative weight, and the graph includes statements having over 1.5% weight in either A or B.

Figure 7 shows that only six of the top 16 used services generate revenue, and that the most used service, consuming about $\approx$ 18% of the database resources, is reading mostly static data, which would have been easy to cache. We see in Figure 8 that this service affects five of the most costly database statements, and this figure also reveals that the second most resource-intensive statement, using between $3.5 - 4\%$ of the database capacity, was entirely unnecessary, caused by misconfiguration of the product.

We see that usage data is critical to understanding current "hot spots" and highlighting inefficiencies that would go unnoticed. Our respondents agree with this statement: "*You need real-life usage data to determine the chips, the size, and which services to use.*"

5 Discussion

Unlike previous studies [204, 205], our study considered moving a complete application stack, including security, operations, and maintenance services, from an existing owned hardware platform to a large public cloud provider. The product had evolved over ten years from "bare metal" deployment to containerization with Kubernetes, but still relied on a classical, monolithic, relational database backend, with no plans to change this architecture.

How would deployment cost change when moving to a public cloud? We found that, for the initial estimate, specific cloud provider *value-added services*, such as a custom Log Analysis service and a Key Vault (managing cryptographic keys for

the application), together with the assumption that these were required for each deployment caused the cost to increase by 300%, as a five-year total, compared to the existing deployment. Even if these assumptions were relaxed by sharing services across deployments and reducing the volume of logs processed by the Log Analysis service, we found that the deployment cost in the public cloud would exceed the existing deployment cost by 50%.

What factors caused this increase in cost? We found that the existing relational database was the major cause of the projected increase in cost when deploying to the chosen cloud provider. Had contracts been renegotiated, another Cloud Provider been chosen, or the database replaced by an Open-Source alternative, this cost driver might have been reduced or disappeared.

We found from interview and usage data that the product's workload was heavily time-dependent, with predictable evening peaks, flat daytime load, and very low nighttime load. Due to the Cloud Provider's fixed pricing model based on VM size regardless of vCPU load, changing the number of Kubernetes worker nodes was the only way to scale automatically. However, such scaling would yield comparatively small savings of around 5%.

Significant cost savings materialized when our Sales Architect questioned whether all requested Cloud Provider services were needed and whether they provided enough value to match their cost. This is a classic requirements engineering problem: although a service might appear extremely useful, its value must exceed the cost it incurs to make economic sense.

Clearly, the Cloud Providers know how to price and market their services to users of their systems. In our case, it was obvious to the Sales Architect that some Cloud Provider services could be shared across deployments and that the new, expensive, AI-based Intrusion Detection System should be used with care, and only be fed data where it could efficiently use its AI capabilities, such as standard system and access logs, but not product-specific application log files.

Larger—but more complex to implement—savings would have been realized by being more careful when accessing product services and, indirectly, the database. For small systems (4-vCPU VMs), the database license and support costs represent about 10%, but for larger systems using 32 vCPUs, these correspond to over 25% of the deployment cost. In a cloud scenario, trying to avoid increasing the database VM size as long as possible makes sense, which is precisely the opposite of the linearly scalable situation depicted in Figure 1. Our respondents found it counterintuitive that autoscaling represented small savings relative to the database and shared services costs.

How can we use service usage data to explain resource usage? We found that service usage data of incoming and outgoing requests and database usage statistics are crucial to finding the parts of the system that drive the resource usage and, therefore, the costs. Using such data, we quickly found several potential improvements that would reduce peak time load on the database by $10 - 20\%$, which translates to

substantial savings for the larger systems (e.g., up to 10 fewer vCPUs). Importantly, these savings also apply—albeit to a lesser degree—in the private cloud scenario, as reduced database license costs.

Summary: Enforcing budgets is a common way to make users aware of economic constraints. With the system currently deployed in a private cloud, no constraints were imposed on system integrators using the product API. This led to a situation where a significant portion (between 10%-50%) of calls to the most used service were unnecessary and could have been cached in an appropriate place, with substantial savings in resource utilization. Likewise, we found an unwanted feature that was partly enabled, causing at least 4% unnecessary database load. In a public cloud scenario, the importance of caching and performance evaluations of features will increase.

Given our traffic pattern, autoscaling would generate marginal cost savings at best. The most significant cost savings would come from carefully choosing cloud provider services, reducing unnecessary API calls, and reviewing the used product features. We also found that API users, such as system integrators, seem to focus more on functionality than on measuring the performance implications of their implemented use case. The product development organization could add budgeting tools based on live system data, allowing customers and system integrators to assess and predict the resource usage of individual use cases. This would allow customers to detect and react to unnecessary API calls by system integrators building frequently used use cases. Furthermore, such budgets would allow end-to-end use case performance testing to verify compliance with the allocated resource budget.

Although complex autoscaling models, such as those envisioned by Boza et al. [216] and Zhu et al. [217], might apply to this system, other factors, such as use case optimization and database scalability, play a more significant role than these models.

6 Conclusions and Further Work

The early works on cloud computing [196, 213] envisioned that computing in a public cloud would be used as a utility, and researchers such as Villamizar et al. [204, 205] have claimed deployment cost savings of up to 77%. We find that the opposite would be the case for a real-life, non-trivial system serving millions of users via hundreds of use cases for over ten years. Even if obvious optimizations were adopted (sharing services across deployments, restricting the use of expensive security services), the deployment cost would increase by around 50% if the suggested public cloud provider were used and the system architecture were unchanged. This suggests that the IDEAL properties of Fehling et al. [212] need to apply to *all* relevant product components and services for the benefits to materialize.

Similar to findings by Taibi et al. [199], our product had evolved using the

"Strangler Fig Pattern" [200], whereby the number of microservices had increased to around 100, of which about 80 were open-source tools. However, most microservices still used a commercial relational database (separating the storage via private schemas), whose license costs were directly tied to the number of virtual CPUs used for the database Virtual Machines. At a minimum, the product or service owner needs to be prepared to buy a "Database-as-a-Service" or ensure that the chosen cloud provider supports Virtual Machines (VMs) of adequate size granularity. Compared to the TOSCA vision of Leymann et al. [210], old relational databases are still heavily reliant on VMs, as the database products have accumulated many features over the years.

In our study, the cloud provider's new, advanced, AI-based Intrusion Detection System represented a large part of the projected cost. Care had to be taken to keep costs down so as not to overload it with data that was unlikely to yield much benefit for the system's security. Sharing costs across deployments also lowered the public cloud deployment cost, but could not match the existing private cloud costs. Such new commercial services, or sharing costs across deployments, are not considered in existing models of cloud cost effectiveness [211], suggesting a need for new, updated models considering the new cloud deployment offerings.

Regarding auto-scaling [215], simple or more complex algorithms, in the vein of Boza et al. [216] and Zhu et al. [217], might be effective to some extent, but in our case, even though the peak-hour traffic exceeded the median daytime load with $\approx 66\%$, and night-time load was even lower, the net savings would be marginal, around 5%.

More savings would be realized using budgeting tools when describing API usage to system integrations to avoid unoptimized use cases. However, these savings would also apply to private cloud deployments, in which case they would postpone expansions of the existing deployment (i.e., reducing the need for additional server hardware). This implies that we agree with the conclusions of [214], at least for systems like ours that keep their original architecture when migrating. This adds context to the studies of migrating legacy systems to the cloud [208, 211], and suggests a potential necessary condition (i.e., cloud-aligning the architecture), before it is economically viable to migrate to a public cloud provider. We find that the data collected by Auer et al. [207] contain some of the metrics (e.g., response time, number of requests per time unit) that can be used to build budgeting tools, provided that these metrics are measured on a level granular enough to assess which services are driving the resource usage.

Our research suggests that the economic benefits of migrating a large legacy system to the present-day commercial public cloud providers are highly dependent on: (i) the architecture of the migrated system—in particular, how dependent it is on traditional relational database systems, (ii) the use of *additional value-added* services provided by the cloud provider—in particular, services used for operation and maintenance (including security monitoring) of the migrated system and (iii) to what

extent it is being “over-used” by clients who might not realize that they are utilizing the system more than needed—the marginal cost of an extraneous invocation leading to an additional required vCPU will be immediately visible in the monthly bill from a public cloud provider. In contrast, owned hardware is expanded much less often; therefore, the cost is much more obscured.

We intend to continue studying the impact of visualizations and budgeting tools on how system integrators can realize the actual deployment cost of their solutions, regardless of whether the system is deployed in a public or private cloud scenario, and invite others to do the same.

Appendix A

On Analyzing Likert Scales With Bayesian Methods

1 Background

In Paper IV, we used 7-level symmetric Likert scales to judge survey repondents' agreement with the respective TAM statement. The survey instruments are available, as Markdown[1] files, in the replication package associated with the paper [193].

We relied on Bayesian data analysis to summarize the responses for the TAM questions. Under TAM, multiple questions are assumed to measure the same construct (usability, ease-of-use, and intent-to-use), and the answers from each respondent form a distribution of ratings used in our Bayesian models.

As control groups, we used two other training sessions, whose participants used tools unrelated to Jaeger tracing. This allows us to compare the expected score (according to the metrics in TAM) between these groups and our subjects.

We summarized survey responses using Bayesian models of various complexity, using PSIS-LOO [178] to rank the models. All models utilized the brms framework [181], which is a front-end for the Stan statistical programming language.

Model construction and evaluation followed a Bayesian work-flow [185], where we:

(i) determine which phenomenon that we were interested in—such as which training session that were investigated;

(ii) determine the factors that might have a causal influence on this phenomenon, such as on which site the training was held;

(iii) determine which priors to use, reflecting the knowledge we have about the organization, and validate that simulations from these priors match our expectations, before the model sees the data;

(iv) run the model, using the collected data and our priors, and check for model convergence, using standard tools (possibly iterate and adjust hyperparameters, such as step size);

[1] `https://daringfireball.net/projects/markdown/`

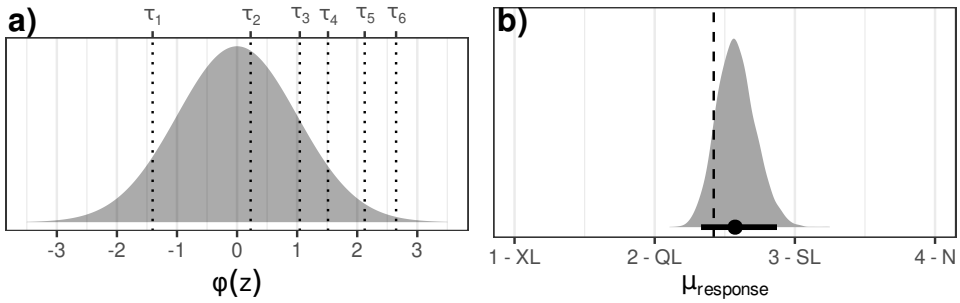


Figure 1: A 7-level cumulative probit ordinal regression partitions the probability under the standard normal distribution $\mathcal{N}(0,1)$. Part **a)** shows 99.7% ($z \in [-3,3]$) of the probability as well as six cutpoints, τ_1 to τ_6. Part **b)** show the associated distribution, point estimate, and 95% credible interval of the expected response $\mathbb{E}(r)$. The dashed line is the arithmetic sample mean. Unlikely levels ($5-7$) are excluded from the x-axis.

(v) use the obtained posterior distribution, and construct appropriate summary statistics that help us learn about the phenomenon.

1.1 Details on the Cumulative Probit

All our models used a *cumulative probit* model, which assumes that the underlying construct being measured by the (discrete, but ordered) Likert responses follows the standard normal distribution function. The *k* response categories to the Likert questions (e.g., *Neutral*, *Slightly Likely*) divide this distribution, using *k-1* thresholds, so that each category occurs with different probability, and the data seen by the model determine these thresholds.

Fig. 1 illustrates a simple model from our data. Part **a)** shows how seven areas under the distribution function $\phi(z) \equiv \mathcal{N}(0,1)$ are formed by six different cutpoints, τ_1 to τ_6, each representing a Likert response.

$$p(k) = \begin{cases} \Phi(\tau_1) = \int_{-\infty}^{\tau_1} \phi(z)\, dz & \text{when } k = 1 \\ \Phi(\tau_k) - \Phi(\tau_{k-1}) = \int_{-\infty}^{\tau_k} \phi(z)\, dz - \int_{-\infty}^{\tau_{k-1}} \phi(z)\, dz & \text{when } 1 < k < 7 \\ 1 - \Phi(\tau_6) = 1 - \int_{-\infty}^{\tau_6} \phi(z)\, dz & \text{when } k = 7 \end{cases} \tag{A.1}$$

The probability $p(k)$ of seeing response $k \in \{k : \mathbb{N}, 1 \leq k \leq 7\}$ in a 7-level Likert scale, given the set of six thresholds τ_k is defined by equation A.1. In Fig. 1, where $\tau_1 \approx -1.406$, the probability of seeing response 1 (*Extremely Likely*) is $p(1) \approx 7.99\%$, while $\tau_2 \approx 0.226$ implies $p(2) \approx 50.9\%$, and so on. Part **b)** shows the distribution of the model-predicted expected response, including the average and 95% credible intervals. The dashed line represents the arithmetic sample mean of the Likert responses, which assumes equidistant levels. In our analysis, we instead use the expected response, defined by:

$$\mathbb{E}(r) = \sum_{k=1}^{7} p(k) \times k \tag{A.2}$$

In Figure 1**b)**, the mean expected value point-estimate is 2.58, and the model estimates, with 95% probability, the expected value to lie between $2.33 - 2.87$. Thus, this model is robustly saying that the expected rating ranges between *Quite Likely* and *Somewhat Likely*, slightly closer to the latter.

Unlike classical frequentist approaches that treat Likert data arithmetically, this model *does not* assume equal spacing between response levels. Instead, the six cut-points ($\tau_1 - \tau_6$) are freely estimated (subject to being ordered) allowing for the possibility that some levels are inherently more likely than others. The model infers these probabilities from the observed data, combined with the specified priors, to produce posterior predictions. These predictions are then used, as shown above, to compute the expected response value, which essentially represents a probability-weighted average of the expected score.

1.2 Model construction

We constructed models for usability (6 questions per respondent), ease-of-use (6 questions), accessibility (3 questions), and intent-to-use (2 questions). Following recommendations by Gelman et al. [192] and McElreath [27], we constructed models of varying complexity and ranked their fitness to the data via the LOO-CV algorithm [178].

The simplest models for each construct ($\mathcal{M}_1$, $\mathcal{E}_1$, $\mathcal{A}_1$, $\mathcal{I}_1$) contain no predictors at all. They could be seen as "global averages," not considering any of the available predictors in the data.

The next level of models ($\mathcal{M}_2$, $\mathcal{E}_2$, $\mathcal{A}_2$, $\mathcal{I}_2$) contain the site (categorical, value *Europe* or *India*) predictor as a fixed effect. In this model, we acknowledge that the trainings took place in different sites, allowing for eventual changes due to this to appear in the data, as models that vary according to whether the respondent took place in the European or Indian training session.

The next level of models ($\mathcal{M}_3$, $\mathcal{E}_3$, $\mathcal{A}_3$, $\mathcal{I}_3$) adds a regularizing prior on the respondent. This means that we use *partial pooling*, which allows for different respondents (i.e., humans) to have different perceptions about how they perceive the Likert levels (e.g., respondent A's sense of *Extremely Likely* may not be the same as respondent B's). The reason we do this is that we are—typically—not interested in predicting the exact answer for these particular individuals, but we want to model and predict how the hypothetical *future respondents* behave. Partial pooling achieves this by "shrinking to the population average," to avoid being biased by extreme outliers [27].

The final set of models ($\mathcal{M}_4$, $\mathcal{E}_4$, $\mathcal{A}_4$, $\mathcal{I}_4$) are those we used to make predictions.

In these models, we also allow the *shape* of the probit curve to vary, by adjusting the model's *dispersion* parameter (α_i) to vary according to the site categorical predictor. They are specified with the equations:

$$
\begin{aligned}
p(\text{rating} = k|\{\tau_k\}, \mu_i, \alpha_i) &= \Phi(\alpha_i[\tau_k - \mu_i]) - \Phi(\alpha_i[\tau_{k-1} - \mu_i]) \\
\mu_i &= 0 + \beta_1 \text{siteIndia}_i + u_i \\
\log(\alpha_i) &= 0 + \gamma_1 \text{siteIndia}_i \\
u_i &\sim \mathcal{N}(0, \sigma_u) \\
\tau_1 &\sim \mathcal{N}(-1.068, 1) \\
\tau_2 &\sim \mathcal{N}(-0.566, 1) \\
\tau_3 &\sim \mathcal{N}(-0.180, 1) \\
\tau_4 &\sim \mathcal{N}(0.180, 1) \\
\tau_5 &\sim \mathcal{N}(0.566, 1) \\
\tau_6 &\sim \mathcal{N}(1.068, 1), \\
\beta_1 &\sim \mathcal{N}(0, 1) \\
\gamma_1 &\sim \mathcal{N}(0, \frac{\log(2)}{2}) \\
\sigma_u &\sim \text{Exponential}(5)
\end{aligned}
\tag{A.3}
$$

In all our models, we used priors for the intercepts τ_k that place similar probability on each response (e.g., $\int_{-\infty}^{-1.068} \phi(x)dx \approx \int_{-1.068}^{-0.566} \phi(x)dx \approx \frac{1}{7}$). However, as we used 1 as standard deviation for our priors, the aggregated priors will place slightly more probability on the middle responses (around *Slightly Likely*, *Neutral* and *Slightly Unlikely*).

Figure 2 show a prior predictive plot of one of our $\mathcal{A}_1$ models, having only the $\tau_1 - \tau_6$ priors. As is shown in the figure, the middle levels *Slightly Likely*, *Neutral* and *Slightly Unlikely* are more likely by the priors, but the extreme ratings (1 and 7) are also quite likely (as the error bars are quite high).

2 Validity Evaluation

We follow guidelines by Wohlin et al. [139] to evaluate the validity of the study described in Paper IV.

Construct validity concerns whether the studied measures reflect what the researcher had in mind, and what is stated in the research questions. The training evaluation survey contains questions modeled after TAM [220], a validated research model. We also checked question formulations by getting feedback from the other teachers before sending the invite. We used conditional branches to keep the survey short and tailored to the chosen training session.

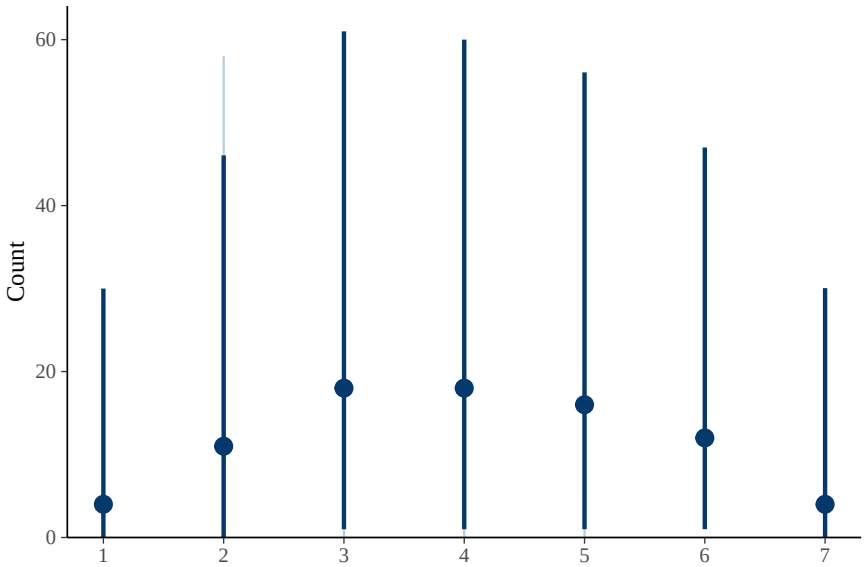


Figure 2: Prior predictive check plot of our priors. Level 1 corresponds to *Extremely Likely*, Level 4 corresponds to *Neutral*, and Level 7 corresponds to *Extremely Unlikely*. The dot represent the median expected probability and the error bars represent the 90% probability range. The figure shows that our model, when just using prior information, rates *Slightly Likely*, *Neutral* and *Slightly Unlikely* (3-5) more likely than the other responses. Still, the extreme ratings are also probable.

External validity concerns the extent to which the findings are generalizable and whether they interest other researchers. Although this study is set in a specific company, we have described the context in detail, and the training session used proven methods, like experiential learning. This should aid other researchers in assessing whether our learnings can be transferred to different companies and domains. We also provided our complete anonymized data set and analysis of the responses in the replication package.

Internal validity deals with whether other, non-studied factors could explain some of the findings. We augmented the quantitative Likert questions with optional, text-based feedback options, and participants on both sites agreed with our conclusions about the training, even though the developers' experience differed. We made clear, via an opt-out clause, that the surveys were anonymous, used for research purposes, and would not be used by the organization to grade participants in their daily work. Still, we cannot rule out that other factors (such as social pressure) would impact survey responses on the training, and we intend to study further how tracing tools are used in practice.

Reliability concerns whether the analysis depends on the specific researchers. We measured usability, ease-of-use, and intent to use the tools via the validated TAM model [220]. All our survey questions were reviewed for clarity by peers before being administered. To aid other researchers in reproducing our results, we have provided

the survey instruments, responses, and models in a replication package [193].

Bibliography

[1] A. Brand, L. Allen, M. Altman, M. Hlava, and J. Scott. “Beyond authorship: Attribution, contribution, collaboration, and credit.” In: *Learned Publishing* 28.2 (2015).

[2] P. Naur and B. Randell, eds. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1969.

[3] A. G. Oettinger. “President’s Letter to the ACM Membership”. In: *Commun. ACM* 9.8 (Aug. 1966), pp. 545–546. ISSN: 0001-0782. DOI: 10.1145/365758.3291288.

[4] P. McBreen. *Software Craftsmanship: The New Imperative*. Addison-Wesley, 2002. ISBN: 978-0201733860.

[5] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. ISBN: 978-0132350884.

[6] R. C. Martin. *The Clean Coder: A Code of Conduct for Professional Programmers*. Prentice Hall, 2011. ISBN: 978-0137081073.

[7] R. C. Martin. *Clean Architecture: A Craftsman’s Guide to Software Structure and Design*. Robert C. Martin Series. Boston, MA: Prentice Hall, 2017. ISBN: 978-0-13-449416-6.

[8] S. Mancuso. *The Software Craftsman: Professionalism, Pragmatism, Pride*. Pearson Education, 2014. ISBN: 978-0134052588.

[9] R. C. Martin. *We, Programmers: A Chronicle of Coders from Ada to AI*. Addison Wesley, 2024. ISBN: 978-0135344262.

[10] C. Thompson. *Coders: The Making of a New Tribe and the Remaking of the World*. New York: Penguin, 2019. ISBN: 978-0735220560.

[11] R. Hoda, N. Salleh, J. Grundy, and H. M. Tee. “Systematic literature reviews in agile software development: A tertiary study”. In: *Information and Software Technology* 85 (2017), pp. 60–70. ISSN: 09505849. DOI: 10.1016/j.infsof.2017.01.007.

[12] M. Neumann. “The integrated list of agile practices-a tertiary study”. In: *International Conference on Lean and Agile Software Development*. Springer. 2022, pp. 19–37.

[13] I. Nurdiani, J. Börstler, and S. A. Fricker. “The impacts of agile and lean practices on project constraints: A tertiary study”. In: *Journal of Systems and Software* 119 (2016), pp. 162–183.

[14] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Pearson Education, 1995. ISBN: 978-0132119160.

[15] H. Washizaki, ed. *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), Version 4.0a*. IEEE Computer Society, 2025.

[16] A. Endres and H. D. Rombach. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws, and Theories*. Pearson Education, 2003. ISBN: 0-321-15420-7.

[17] R. L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, 2003. ISBN: 0-321-11742-5.

[18] T. Winters, T. Manshreck, and H. Wright. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O’Reilly Media, Inc., 2020. ISBN: 978-1-492-08279-8.

[19] M. Lehman. “Programs, life cycles, and laws of software evolution”. In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076. DOI: 10.1109/PROC.1980.11805.

[20] C. A. Furia, R. Feldt, and R. Torkar. “Bayesian data analysis in empirical software engineering research”. In: *IEEE Transactions on Software Engineering* 47 (9 Sept. 2021), pp. 1786–1810. ISSN: 19393520. DOI: 10.1109/TSE.2019.2935974.

[21] A. Gelman. “Commentary: P: Values and Statistical Practice”. In: *Epidemiology* 24.1 (2013), pp. 69–72. DOI: 10.1097/EDE.0b013e31827886f7.

[22] A. Gelman and J. Carlin. “Some Natural Solutions to the p-Value Communication Problem—and Why They Won’t Work”. In: *Journal of the American Statistical Association* 112.519 (2017), pp. 899–901. DOI: 10.1080/01621459.2017.1311263.

[23] J. Pearl. *Causality*. 2nd ed. Cambridge, UK: Cambridge University Press, 2009. ISBN: 978-0521895606.

[24] C. Cinelli, A. Forney, and J. Pearl. “A Crash Course in Good and Bad Controls”. In: *Sociological Methods & Research* 53.3 (2024), pp. 1071–1104. DOI: 10.1177/00491241221099552.

[25] C. A. Furia, R. Torkar, and R. Feldt. “Applying Bayesian Analysis Guidelines to Empirical Software Engineering Data: The Case of Programming Languages and Code Quality”. In: *ACM Transactions on Software Engineering and Methodology* 31 (3 July 2022). ISSN: 15577392. DOI: 10.1145/3490953.

[26] C. A. Furia, R. Torkar, and R. Feldt. “Towards Causal Analysis of Empirical Software Engineering Data: The Impact of Programming Languages on Coding Competitions”. In: *ACM Transactions on Software Engineering and Methodology* 33.1 (Nov. 2023). ISSN: 1049-331X. DOI: 10.1145/3611667.

[27] R. McElreath. *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*. 2nd ed. Boca Raton, FL, USA: CRC press, 2020. ISBN: 978-0367139919.

[28] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. 2nd ed. PWS Publishing Company, 1997. ISBN: 0-534-95600-9.

[29] G. Norman. “Likert scales, levels of measurement and the “laws” of statistics”. In: *Advances in Health Sciences Education* 15.5 (2010), pp. 625–632.

[30] P.-C. Bürkner and M. Vuorre. “Ordinal Regression Models in Psychology: A Tutorial”. In: *Advances in Methods and Practices in Psychological Science* 2.1 (2019), pp. 77–101. DOI: 10.1177/2515245918823199.

[31] V. R. Basili, G. Caldiera, and H. D. Rombach. “The goal question metric approach”. In: *Encyclopedia of Software Engineering*. Vol. 2. Wiley, 1994, pp. 1–10.

[32] L. T. M. Blessing and A. Chakrabarti. *DRM, a Design Research Methodology*. English. 1st ed. London: Springer, 2009. ISBN: 978-1848825864.

[33] K.-J. Stol and B. Fitzgerald. “The ABC of Software Engineering Research”. In: *ACM Transactions on Software Engineering and Methodology* 27.3 (2018), pp. 1–51. DOI: 10.1145/3241743.

[34] T. Dingsøyr, T. E. Fægri, and J. Itkonen. “What is Large in Large-Scale? A Taxonomy of Scale for Agile Software Development”. In: *International Conference on Product-Focused Software Process Improvement*. Springer. 2014, pp. 273–276. ISBN: 978-3-319-13834-3.

[35] C. Robson and K. McCartan. *Real World Research*. 4th ed. Chichester, West Sussex, United Kingdom: Wiley, 2016. ISBN: 978-111-874-5236.

[36] M. Lipscomb. “Mixed method nursing studies: a critical realist critique”. In: *Nursing Philosophy* 9.1 (2008), pp. 32–45.

[37] P. McEvoy and D. Richards. “A critical realist rationale for using a combination of quantitative and qualitative methods”. In: *Journal of research in nursing* 11.1 (2006), pp. 66–78.

[38] R. Bhaskar. *A Realist Theory of Science*. Routledge, 2013. ISBN: 978-1134050864.

[39] A. Tornhill. *Software Design X-Rays - Fix Technical Debt with Behavioral Code Analysis*. The Pragmatic Programmers, LLC, 2018. ISBN: 987-1-68050-272-5.

[40] P. Runeson, M. Höst, A. Rainer, and B. Regnell. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012. ISBN: 978-1-118-10435-4.

[41] T. D. Cook and D. T. Campbell. *Quasi-Experimentation*. Houghton Mifflin Company, 1979. ISBN: 0-395-30790-2.

[42] B. Flyvbjerg. "Five Misunderstandings About Case-Study Research". In: *Qualitative Inquiry* 12.2 (2006), pp. 219–245. DOI: 10.1177/1077800405284363.

[43] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Pearson Education, 1999. ISBN: 978-0132119177.

[44] S. Jalali and C. Wohlin. "Global software engineering and agile practices: A systematic review". In: *Journal of software: Evolution and Process* 24.6 (2012), pp. 643–659.

[45] D. Salah, R. F. Paige, and P. Cairns. "A systematic literature review for Agile development processes and user centred design integration". In: *18th International Conference on Evaluation and Assessment in Software Engineering*. London, UK: Association for Computing Machinery, 2014. ISBN: 978-1450324762. DOI: 10.1145/2601248.2601276.

[46] R. Vallon, B. J. da Silva Estácio, R. Prikladnicki, and T. Grechenig. "Systematic literature review on agile practices in global software development". In: *Information and Software Technology* 96.April 2017 (2018), pp. 161–180. ISSN: 09505849. DOI: 10.1016/j.infsof.2017.12.004.

[47] P. Diebold and M. Dahlem. "Agile practices in practice - A mapping study". In: *18th International Conference on Evaluation and Assessment in Software Engineering*. Association for Computing Machinery, 2014. ISBN: 978-1450324762. DOI: 10.1145/2601248.2601254.

[48] D. Karlström and P. Runeson. "Combining Agile Methods with Stage-Gate Project Management". In: *IEEE Software* May/June (2005), pp. 43–49.

[49] T. Dogša and D. Batič. "The effectiveness of test-driven development: An industrial case study". In: *Software Quality Journal* 19.4 (2011), pp. 643–661. ISSN: 15731367. DOI: 10.1007/s11219-011-9130-2.

[50] P. Abrahamsson and J. Koskela. "Extreme programming: A survey of empirical data from a controlled case study". In: *Proceedings - 2004 International Symposium on Empirical Software Engineering, ISESE '04.* IEEE, 2004, pp. 73–82. DOI: 10.1109/ISESE.2004.1334895.

[51] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo. "A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?" In: *IEEE Transactions on Software Engineering* 43.7 (2017), pp. 597–614. DOI: 10.1109/TSE.2016.2616877.

[52] A. Tosun, O. Dieste, D. Fucci, S. Vegas, B. Turhan, H. Erdogmus, A. Santos, M. Oivo, K. Toro, J. Jarvinen, and N. Juristo. “An industry experiment on the effects of test-driven development on external quality and productivity”. In: *Empirical Software Engineering* 22.6 (Dec. 2017), pp. 2763–2805. ISSN: 15737616. DOI: 10.1007/s10664-016-9490-0.

[53] H. Munir, M. Moayyed, and K. Petersen. “Considering rigor and relevance when evaluating test driven development: A systematic review”. In: (2014). DOI: 10.1016/j.infsof.2014.01.002.

[54] M. Poppendieck and T. Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley, 2003. ISBN: 978-0321150783.

[55] K. Petersen and C. Wohlin. “Measuring the flow in lean software development”. In: *Software - Practice and Experience* 41.9 (2011), pp. 975–996. ISSN: 00380644. DOI: 10.1002/spe.975.

[56] K. Petersen. “A palette of lean indicators to detect waste in software maintenance: A case study”. In: *Lecture Notes in Business Information Processing*. Vol. 111 LNBIP. Springer Verlag, 2012, pp. 108–122. ISBN: 978-3642303494. DOI: 10.1007/978-3-642-30350-0_8.

[57] O. Cawley, X. Wang, and I. Richardson. “Lean/agile software development methodologies in regulated environments - State of the art”. In: *International Conference on Lean Enterprise Software and Systems*. Vol. 65 LNBIP. Springer Verlag, 2010, pp. 31–36. ISBN: 3642164153. DOI: 10.1007/978-3-642-16416-3_4.

[58] M. Feyh and K. Petersen. “Lean software development measures and indicators - A systematic mapping study”. In: *Lecture Notes in Business Information Processing*. Vol. 167. Springer Verlag, 2013, pp. 32–47. ISBN: 978-3642449291. DOI: 10.1007/978-3-642-44930-7_3.

[59] E. Kupiainen, M. V. Mäntylä, and J. Itkonen. *Using metrics in Agile and Lean software development - A systematic literature review of industrial studies*. 2015. DOI: 10.1016/j.infsof.2015.02.005.

[60] C. Wohlin. “Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering”. In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE ’14. London, England, United Kingdom: ACM, 2014, 38:1–38:10. ISBN: 978-1-4503-2476-2. DOI: 10.1145/2601248.2601268.

[61] E. Mourão, J. F. Pimentel, L. Murta, M. Kalinowski, E. Mendes, and C. Wohlin. “On the performance of hybrid search strategies for systematic literature reviews in software engineering”. In: *Information and Software Technology* 123 (2020), p. 106294. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2020.106294.

[62] V. Garousi, M. Felderer, and M. V. Mäntylä. “Guidelines for including grey literature and conducting multivocal literature reviews in software engineering”. In: *Information and Software Technology* 106.May 2018 (2019), pp. 101–121. ISSN: 09505849. DOI: 10.1016/j.infsof.2018.09.006. arXiv: 1707.02553.

[63] P. Taylor. “Vernacularism in Software Design Practice: does craftsmanship have a place in software engineering?” In: *Australasian Journal of Information Systems* 11.1 (2003). ISSN: 1449-8618. DOI: 10.3127/ajis.v11i1.143. URL: https://journal.acs.org.au/index.php/ajis/article/view/143.

[64] M. Paasivaara and C. Lassenius. “Communities of practice in a large distributed agile software development organization – Case Ericsson”. In: *Information and Software Technology* 56.12 (2014). Special issue: Human Factors in Software Development, pp. 1556–1577. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2014.06.008.

[65] P. Rodríguez, K. Mikkonen, P. Kuvaja, M. Oivo, and J. Garbajosa. “Building Lean Thinking in a Telecom Software Development Organization: Strengths and Challenges”. In: *Proceedings of the 2013 International Conference on Software and System Process*. ICSSP 2013. San Francisco, CA, USA: ACM, 2013, pp. 98–107. ISBN: 978-1-4503-2062-7. DOI: 10.1145/2486046.2486064.

[66] J. Lingel and T. Regan. “”it’s in Your Spinal Cord, It’s in Your Fingertips”: Practices of Tools and Craft in Building Software”. In: *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*. CSCW ’14. Baltimore, Maryland, USA: ACM, 2014, pp. 295–304. ISBN: 978-1-4503-2540-0. DOI: 10.1145/2531602.2531614.

[67] P. Lucena and L. P. Tizzei. “Applying Software Craftsmanship Practices to a Scrum Project: an Experience Report”. In: *I Workshop sobre Aspectos Sociais, Humanos e Econômicos de Software (WASHES 2016)*. Maceió, Alagoas, Brazil, 2016. arXiv: 1611.05789. URL: http://arxiv.org/abs/1611.05789.

[68] I. Jacobson and E. Seidewitz. “A New Software Engineering”. In: *Queue* 12.10 (Oct. 2014), 30:30–30:38. ISSN: 1542-7730. DOI: 10.1145/2685690.2693160.

[69] J. O. Coplien. “Borland Software Craftsmanship: A New Look at Process, Quality and Productivity”. In: *Proceedings of the 5th Annual Borland International Conference*. Orlando, FL, USA, 1994.

[70] B. Pyritz. “Craftsmanship versus engineering: Computer programming - An art or a science?” In: *Bell Labs Technical Journal* 8.3 (2003), pp. 101–104. DOI: 10.1002/bltj.10079.

[71] G. Marcionetti, F. Cannizzo, and P. Moser. “The Toolbox of a Successful Software Craftsman”. In: *Engineering of Computer-Based Systems, IEEE International Conference on the(ECBS)*. Vol. 00. Mar. 2008, pp. 389–397. DOI: 10.1109/ECBS.2008.48.

[72] D. Parsons, T. Susnjak, and A. Mathrani. “Design from detail: Analyzing data from a global day of coderetreat”. In: *Information and Software Technology* 75 (2016), pp. 39–55. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2016.03.005.

[73] R. Lindell. “Crafting Interaction: The Epistemology of Modern Programming”. In: *Personal Ubiquitous Comput.* 18.3 (Mar. 2014), pp. 613–624. ISSN: 1617-4909. DOI: 10.1007/s00779-013-0687-6.

[74] D. Thomas. “Professional Developers Practice their Kata to Stay Sharp.” In: *Journal of Object Technology* 9 (Mar. 2010), pp. 23–25. DOI: 10.5381/jot.2010.9.2.c3.

[75] D. Parsons, A. Mathrani, T. Susnjak, and A. Leist. “Coderetreats: Reflective Practice and the Game of Life”. In: *IEEE Software* 31.4 (July 2014), pp. 58–64. ISSN: 0740-7459. DOI: 10.1109/MS.2014.25.

[76] B. Boehm. “A View of 20th and 21st Century Software Engineering”. In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE ’06. Shanghai, China: ACM, 2006, pp. 12–29. ISBN: 1-59593-375-1. DOI: 10.1145/1134285.1134288.

[77] T. Sedano. “Towards Teaching Software Craftsmanship”. In: *2012 IEEE 25th Conference on Software Engineering Education and Training*. Apr. 2012, pp. 95–99. DOI: 10.1109/CSEET.2012.29.

[78] I. Bergström and A. F. Blackwell. “The practices of programming”. In: *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*. Vol. 2016-Novem. 2016, pp. 190–198. ISBN: 978-1509002528. DOI: 10.1109/VLHCC.2016.7739684.

[79] R. Lindell. “The Craft of Programming Interaction”. In: *Proceedings of International Workshop on the Interplay between User Experience Evaluation and Software Development (I-UxSED 2012)*. 2012, pp. 26–30.

[80] N. Wirth. “A Brief History of Software Engineering”. In: *IEEE Annals of the History of Computing* 30.3 (2008), pp. 32–39. ISSN: 10586180. DOI: 10.1109/MAHC.2008.33.

[81] C. Larman and B. Vodde. *Scaling Lean and Agile Development: Thinking and Organizational Tools for Large-Scale Scrum*. Agile Software Development Series. Pearson Education, 2008. ISBN: 978-0321617149.

[82] R. Sennett. *The Craftsman*. Yale University Press, 2008. ISBN: 978-0300149555.

[83] D. Hoover and A. Oshineye. *Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman*. Theory in practice. O'Reilly Media, 2009. ISBN: 978-1449379407.

[84] P. Seibel. *Coders at Work: Reflections on the Craft of Programming*. IT Pro. Apress, 2009. ISBN: 978-1430219491.

[85] C. Haines. *Understanding the 4 Rules of Simple Design*. Leanpub, 2014. URL: https://leanpub.com/4rulesofsimpledesign (visited on 04/19/2018).

[86] R. Winter. *Agile Performance Improvement: The New Synergy of Agile and Human Performance Technology*. Apress, 2015. ISBN: 978-1484208922.

[87] P. Chatzipetrou, D. Šmite, and R. van Solingen. "When and Who Leaves Matters: Emerging Results from an Empirical Study of Employee Turnover". In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '18. Oulu, Finland: Association for Computing Machinery, 2018. ISBN: 978-1450358231. DOI: 10.1145/3239235.3267431.

[88] V. Braun and V. Clarke. "Using thematic analysis in psychology". In: *Qualitative Research in Psychology* 3.2 (Jan. 2006), pp. 77–101. ISSN: 1478-0887. DOI: 10.1191/1478088706qp063oa. arXiv: 1011.1669.

[89] B. G. Glaser and A. L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Publishing Company, 1967. ISBN: 978-1138535169.

[90] K. J. Stol, P. Ralph, and B. Fitzgerald. "Grounded Theory in Software Engineering Research: A Critical Review and Guidelines". In: *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 120–131. ISBN: 978-1450339001. DOI: 10.1145/2884781.2884833.

[91] V. Braun and V. Clarke. "What can "thematic analysis" offer health and wellbeing researchers?" In: *International Journal of Qualitative Studies on Health and Well-being* 9 (2014), pp. 20–22. ISSN: 17482631. DOI: 10.3402/qhw.v9.26152.

[92] M. I. Alhojailan. "Thematic Analysis: A Critical Review of Its Process and Evaluation". In: *West East Journal of Social Sciences* 1 (2012), pp. 39–47.

[93] J. Saldana. *Coding Manual for Qualitative Researchers*. 3rd. Sage Publications, 2015, p. 223. ISBN: 1473902495.

[94] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley Professional, 2001. ISBN: 978-0201616415.

[95] B. Curtis, H. Krasner, and N. Iscoe. “A Field Study of the Software Design Process for Large Systems”. In: *Commun. ACM* 31.11 (Nov. 1988), pp. 1268–1287. ISSN: 0001-0782. DOI: 10.1145/50087.50089.

[96] M. Ivarsson and T. Gorschek. “A method for evaluating rigor and industrial relevance of technology evaluations”. In: *Empirical Software Engineering* 16.3 (2011), pp. 365–395. ISSN: 13823256. DOI: 10.1007/s10664-010-9146-4.

[97] M. Fowler. *BeckDesignRules*. 2015. URL: https://martinfowler.com/articles/BeckDesignRules.html (visited on 08/08/2020).

[98] T. Gilbert. *Human Competence: Engineering Worthy Performance*. McGraw-Hill, 1978. ISBN: 978-0070232174.

[99] M. Fowler, K. Beck, J. Brant, and W. Opdyke. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. ISBN: 978-0201485677.

[100] J. McCarthy. *Dynamics of Software Development*. Vol. 3. Redmond, WA, USA: Microsoft Press, 1995. ISBN: 978-1556158230.

[101] R. Britto, D. Šmite, and L.-O. Damm. “Software Architects in Large-Scale Distributed Projects: An Ericsson Case Study”. In: *IEEE Software* 33.6 (Nov. 2016), pp. 48–55. ISSN: 0740-7459. DOI: 10.1109/MS.2016.146.

[102] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Upper Saddle River, NJ, USA: Pearson Education, Inc., 2013. ISBN: 978-0321815736.

[103] E. W. Dijkstra. “On the role of scientific thought”. In: *Selected writings on computing: a personal perspective*. Springer, 1982, pp. 60–66.

[104] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns*. Vol. Vol. 1. Chichester: John Wiley & Sons, 1996, p. 476. ISBN: 0471958697.

[105] M. Richards. *Software Architecture Patterns*. O’Reilly Media, Inc., 2015. ISBN: 978-1491924242.

[106] P. C. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, J. Stafford, P. Merson, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. 2nd Editio. Pearson Education, 2010. ISBN: 978-0321552686.

[107] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994. ISBN: 978-0321700698.

[108] M. Kim, T. Zimmermann, and N. Nagappan. “A field study of refactoring challenges and benefits”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE ’12*. New York, New York, USA: ACM Press, 2012, p. 1. ISBN: 978-1450316149. DOI: 10.1145/2393596.2393655.

[109] E. Zabardast, J. Gonzalez-Huerta, and D. Šmite. “Refactoring , Bug Fixing , and New Development Effect on Technical Debt : An Industrial Case Study”. In: *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, Aug. 2020, pp. 376–384. ISBN: 978-1728195322. DOI: 10.1109/SEAA51224.2020.00068.

[110] L. Crispin and J. Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Signature Series (Cohn). Pearson Education, 2008. ISBN: 978-0321616937.

[111] M. Cohn. *Succeeding with Agile: Software Development Using Scrum*. Pearson Education, 2009, p. 504. ISBN: 0321660560.

[112] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007. ISBN: 978-0131495050.

[113] E. Bjarnason, M. Unterkalmsteiner, M. Borg, and E. Engström. “A multi-case study of agile requirements engineering and the use of test cases as requirements”. In: *Information and Software Technology* 77 (2016), pp. 61–79.

[114] D. E. Knuth. “Literate programming”. In: *The Computer Journal* 27.2 (1984), pp. 97–111.

[115] G. L. Kelling, J. Q. Wilson, et al. “Broken windows”. In: *Atlantic monthly* 249.3 (1982), pp. 29–38.

[116] A. Silva, T. Araújo, J. Nunes, M. Perkusich, E. Dilorenzo, H. Almeida, and A. Perkusich. “A Systematic Review on the Use of Definition of Done on Agile Software Development Projects”. In: *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. EASE’17. Karlskrona, Sweden: Association for Computing Machinery, 2017, pp. 364–373. ISBN: 978-1450348041. DOI: 10.1145/3084226.3084262.

[117] E. Wenger. *Communities of Practice: Learning, Meaning, and Identity*. Cambridge, UK: Cambridge University Press, 1999. ISBN: 978-0521663632.

[118] D. Šmite, N. B. Moe, G. Levinta, and M. Floryan. “Spotify Guilds: How to Succeed With Knowledge Sharing in Large-Scale Agile Organizations”. In: *IEEE Software* 36.2 (2019), pp. 51–57. DOI: 10.1109/MS.2018.2886178.

[119] D. Šmite, N. B. Moe, M. Floryan, G. Levinta, and P. Chatzipetrou. “Spotify Guilds”. In: *Commun. ACM* 63.3 (Feb. 2020), pp. 56–61. ISSN: 0001-0782. DOI: 10.1145/3343146.

[120] W. Cunningham. “The WyCash portfolio management system”. In: *ACM SIGPLAN OOPS Messenger* 4.2 (1992), pp. 29–30.

[121] R. Westrum. “A typology of organisational cultures”. In: *BMJ Quality & Safety* 13.suppl 2 (2004), pp. ii22–ii27.

[122] W. W. Royce. “Managing the development of large software systems”. In: *Proceedings, IEEE WESCON*. 1970, pp. 1–9. DOI: 10.1016/0378-4754(91)90107-E.

[123] M. Beedle, M. Devos, Y. Sharon, K. Schwaber, and J. Sutherland. “SCRUM: An extension pattern language for hyperproductive software development”. In: *Pattern languages of program design, 4*. Ed. by N. Harrison, B. Foote, and H. Rohnert. Vol. 4. Reading, MA: Addison Wesley, 2000. Chap. 28, pp. 637–651. ISBN: 978-0201433043.

[124] CollabNet VersionOne. *The 13th annual STATE OF AGILE Report - 2018*. Tech. rep. 2019. URL: https://stateofagile.com/#ufh-i-613553418-13th-annual-state-of-agile-report/7027494.

[125] R. K. Yin. *Case Study Research: Design and Methods*. 5th Ed. Sage Publications, Inc., 2014. ISBN: 978-1-4522-4256-9.

[126] P. Ralph and E. Tempero. “Construct Validity in Software Engineering Research and Software Metrics”. In: *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*. EASE’18. Christchurch, New Zealand: Association for Computing Machinery, 2018, pp. 13–23. ISBN: 978-1450364034. DOI: 10.1145/3210459.3210461.

[127] M. Fowler. *Microservices*. 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 01/08/2020).

[128] A. Balalaie, A. Heydarnoori, and P. Jamshidi. “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture”. In: *IEEE Software* 33.3 (May 2016), pp. 42–52. ISSN: 07407459. DOI: 10.1109/MS.2016.64.

[129] O. Zimmermann. “Microservices tenets: Agile approach to service development and deployment”. In: *Computer Science - Research and Development* 32.3-4 (2017), pp. 301–310. ISSN: 18652042. DOI: 10.1007/s00450-016-0337-0.

[130] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman. “Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162)”. In: *Dagstuhl Reports* 6.4 (2016), pp. 110–138. ISSN: 2192-5283. DOI: 10.4230/DagRep.6.4.110.

[131] A. A. Sawant, R. Robbes, and A. Bacchelli. “On the reaction to deprecation of 25,357 clients of 4+1 popular Java APIs”. In: *Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016* (2017), pp. 400–410. DOI: 10.1109/ICSME.2016.64.

[132] A. A. Sawant, R. Robbes, and A. Bacchelli. “To react, or not to react: Patterns of reaction to API deprecation”. In: *Empirical Software Engineering* (2019), pp. 3824–3870. ISSN: 15737616. DOI: 10.1007/s10664-019-09713-w.

[133] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue. “Do developers update their library dependencies?: An empirical study on the impact of security advisories on library migration”. In: *Empirical Software Engineering* 23.1 (2018), pp. 384–417. ISSN: 15737616. DOI: 10.1007/s10664-017-9521-5. arXiv: 1709.04621.

[134] W. Snipes and S. Ramaswamy. “A proposed sizing model for managing 3rd party code technical debt”. In: *Proceedings - International Conference on Software Engineering* (2018), pp. 72–75. ISSN: 02705257. DOI: 10.1145/3194164.3194179.

[135] B. Curtis, J. Sappidi, and A. Szynkarski. “Estimating the principal of an application’s technical debt”. In: *IEEE Software* 29.6 (2012), pp. 34–42.

[136] Z. Codabux and B. Williams. “Managing Technical Debt: An Industrial Case Study”. In: *Proceedings of the 4th International Workshop on Managing Technical Debt*. 2013, pp. 8–15. ISBN: 978-1467364430.

[137] P. Kruchten, R. L. Nord, and I. Ozkaya. *Managing Technical Debt: Reducing Friction in Software Development*. Addison-Wesley Professional, 2019. ISBN: 978-0135645932.

[138] E. Alégroth and J. Gonzalez-Huerta. “Towards a Mapping of Software Technical Debt onto Testware”. In: *43rd Euromicro Conference on Software Engineering and Advanced Applications*. Vienna, Austria, 2017, pp. 404–411.

[139] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Berlin: Springer, 2012. ISBN: 978-3-642-29043-5.

[140] R. Koschke. “Survey of research on software clones”. In: *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2007.

[141] C. K. Roy and J. R. Cordy. “A survey on software clone detection research”. In: *Queen’s School of computing TR* 541.115 (2007), pp. 64–68.

[142] D. Rattan, R. Bhatia, and M. Singh. “Software clone detection: A systematic review”. In: *Information and Software Technology* 55.7 (2013), pp. 1165–1199. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2013.01.008.

[143] J. R. Pate, R. Tairas, and N. A. Kraft. “Clone evolution: a systematic review”. In: *Journal of Software: Evolution and Process* 25.3 (2013), pp. 261–283. DOI: 10.1002/smr.579.

[144] E. V. d. P. Sobrinho, A. De Lucia, and M. d. A. Maia. “A Systematic Literature Review on Bad Smells–5 W’s: Which, When, What, Who, Where”. In: *IEEE Transactions on Software Engineering* 47.1 (2021), pp. 17–66. DOI: 10.1109/TSE.2018.2880977.

[145] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. “Do code clones matter?” In: *2009 IEEE 31st International Conference on Software Engineering*. IEEE. 2009, pp. 485–495.

[146] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman. “Identification and management of technical debt: A systematic mapping study”. In: *Information and Software Technology* 70 (2016), pp. 100–121. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2015.10.008.

[147] M. Fowler. *CodeOwnership*. [Online; accessed 11-July-2023]. 2006. URL: https://martinfowler.com/bliki/CodeOwnership.html.

[148] D. M. Ribeiro, F. Q. da Silva, D. Valença, E. L. Freitas, and C. França. “Advantages and disadvantages of using shared code from the developers perspective: a qualitative study”. In: *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 2016, pp. 1–6.

[149] M. Nordberg. “Managing code ownership”. In: *IEEE Software* 20.2 (2003), pp. 26–33. DOI: 10.1109/MS.2003.1184163.

[150] P. Avgeriou, P. Kruchten, R. L. Nord, I. Ozkaya, and C. Seaman. “Reducing friction in software development”. In: *IEEE Software* 33.1 (2016), pp. 66–72. ISSN: 0740-7459 VO - 33. DOI: 10.1109/MS.2016.13.

[151] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. O’Reilly Media, Inc., 2015. ISBN: 978-1491950357.

[152] S. Baškarada, V. Nguyen, and A. Koronios. “Architecting Microservices: Practical Opportunities and Challenges”. In: *Journal of Computer Information Systems* 60.5 (2020), pp. 428–436. DOI: 10.1080/08874417.2018.1520056.

[153] T. Sedano, P. Ralph, and C. Péraire. “Practice and perception of team code ownership”. In: *Proceedings of the 20th international conference on evaluation and assessment in software engineering*. 2016, pp. 1–6.

[154] G. Hardin. “The tragedy of the commons: the population problem has no technical solution; it requires a fundamental extension in morality.” In: *Science* 162.3859 (1968), pp. 1243–1248.

[155] T. Dietz, E. Ostrom, and P. C. Stern. “The struggle to govern the commons”. In: *Science* 302.5652 (2003), pp. 1907–1912.

[156] J. F. Abrantes and G. H. Travassos. “Common Agile Practices in Software Processes”. In: *2011 International Symposium on Empirical Software Engineering and Measurement*. 2011, pp. 355–358. DOI: 10.1109/ESEM.2011.47.

[157] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. “Don’t touch my code! Examining the effects of ownership on software quality”. In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011, pp. 4–14.

[158] M. Greiler, K. Herzig, and J. Czerwonka. “Code ownership and software quality: A replication study”. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE. 2015, pp. 2–12.

[159] G. Avelino, L. Passos, A. Hora, and M. T. Valente. “Measuring and analyzing code authorship in 1 + 118 open source projects”. In: *Science of Computer Programming* 176 (2019), pp. 14–32. ISSN: 0167-6423. DOI: 10.1016/j.scico.2019.03.001.

[160] M. Foucault, C. Teyton, D. Lo, X. Blanc, and J.-R. Falleri. “On the usefulness of ownership metrics in open-source software projects”. In: *Information and Software Technology* 64 (2015), pp. 102–112.

[161] L. M. Maruping, X. Zhang, and V. Venkatesh. “Role of collective ownership and coding standards in coordinating expertise in software project teams”. In: *European Journal of Information Systems* 18 (2009), pp. 355–371.

[162] C. Faragó, P. Hegedűs, and R. Ferenc. “Code ownership: Impact on maintainability”. In: *Computational Science and Its Applications–ICCSA 2015: 15th International Conference, Banff, AB, Canada, June 22-25, 2015, Proceedings, Part V 15*. Springer. 2015, pp. 3–19.

[163] M. Orrú and M. Marchesi. “A case study on the relationship between code ownership and refactoring activities in a Java software system”. In: *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics*. WETSoM ’16. Austin, Texas: Association for Computing Machinery, 2016, pp. 43–49. ISBN: 9781450341776. DOI: 10.1145/2897695.2897702.

[164] M. Borg, A. Tornhill, and E. Mones. “U Owns the Code That Changes and How Marginal Owners Resolve Issues Slower in Low-Quality Source Code”. In: *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. EASE ’23. Oulu, Finland: Association for Computing Machinery, 2023, pp. 368–377. ISBN: 9798400700446. DOI: 10.1145/3593434.3593480.

[165] E. Zabardast, J. Gonzalez-Huerta, and B. Tanveer. “Ownership vs Contribution: Investigating the Alignment Between Ownership and Contribution”. In: *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. IEEE. 2022, pp. 30–34.

[166] E. Zabardast, J. Gonzalez-Huerta, and F. Palma. “The Impact of Forced Working-From-Home on Code Technical Debt: An Industrial Case Study”. In: *48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2022, pp. 298–305. ISBN: 9781665461528. DOI: 10.1109/SEAA56994.2022.00054.

[167] K. Herzig and N. Nagappan. “The Impact of Test Ownership and Team Structure on the Reliability and Effectiveness of Quality Test Runs”. In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM ’14. Torino, Italy: Association for Computing Machinery, 2014. ISBN: 9781450327749. DOI: 10.1145/2652524.2652535.

[168] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool. “A Systematic Review on Code Clone Detection”. In: *IEEE Access* 7 (2019), pp. 86121–86144. DOI: 10.1109/ACCESS.2019.2918202.

[169] F. H. Quradaa, S. Shahzad, and R. S. Almoqbily. “A systematic literature review on the applications of recurrent neural networks in code clone research”. In: *PLOS ONE* 19.2 (Feb. 2024), pp. 1–40. DOI: 10.1371/journal.pone.0296858.

[170] M. Kaur and D. Rattan. “A systematic literature review on the use of machine learning in code clone research”. In: *Computer Science Review* 47 (2023), p. 100528.

[171] S. Rongrong, Z. Liping, and Z. Fengrong. “A Method for Identifying and Recommending Reconstructed Clones”. In: *Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences*. ICMSS 2019. Wuhan, China: Association for Computing Machinery, 2019, pp. 39–44. ISBN: 9781450361897. DOI: 10.1145/3312662.3312709.

[172] M. Zakeri-Nasrabadi, S. Parsa, M. Ramezani, C. Roy, and M. Ekhtiarzadeh. “A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges”. In: *Journal of Systems and Software* 204 (2023), p. 111796. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2023.111796.

[173] Y. Zhong, X. Zhang, W. Tao, and Y. Zhang. “A systematic literature review of clone evolution”. In: *Proceedings of the 5th International Conference on Computer Science and Software Engineering*. CSSE ’22. Guilin, China: Association for Computing Machinery, 2022, pp. 461–473. ISBN: 9781450397780. DOI: 10.1145/3569966.3570091.

[174] L. Yu, S. Ramaswamy, and A. Vaidyanathan. “Understanding the Effects of Code Clones on Modularity in Software Systems”. In: *2012 19th Asia-Pacific Software Engineering Conference*. Vol. 2. 2012, pp. 105–111. DOI: 10.1109/APSEC.2012.49.

[175] C. J. Kapser and M. W. Godfrey. ““Cloning considered harmful” considered harmful: Patterns of cloning in software”. In: *Empirical Software Engineering* 13 (2008), pp. 645–692. DOI: 10.1007/s10664-008-9076-6.

[176] C. K. Roy, J. R. Cordy, and R. Koschke. “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach”. In: *Science of Computer Programming* 74.7 (2009), pp. 470–495. ISSN: 0167-6423. DOI: 10.1016/j.scico.2009.02.007.

[177] K. Ljung and J. Gonzalez-Huerta. ““To Clean Code or Not to Clean Code” A Survey Among Practitioners”. In: *23rd International Conference on Product-Focused Software Process Improvement*. Elsevier, 2022, pp. 298–315. DOI: 10.1007/978-3-031-21388-5_21.

[178] A. Vehtari, A. Gelman, and J. Gabry. “Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC”. In: *Statistics and computing* 27 (2017), pp. 1413–1432. DOI: 10.1007/s11222-016-9696-4.

[179] T. J. McCabe. “A complexity measure”. In: *IEEE Transactions on software Engineering* (4 1976), pp. 308–320.

[180] A. Sundelin and A. Bauer. *epkanol/governing-commons-code-clones: Initial submission*. Version v0.0.1. May 2024. DOI: 10.5281/zenodo.11357296.

[181] P.-C. Bürkner. “brms: An R Package for Bayesian Multilevel Models Using Stan”. In: *Journal of Statistical Software* 80.1 (2017), pp. 1–28. DOI: 10.18637/jss.v080.i01.

[182] P.-C. Bürkner. “Advanced Bayesian Multilevel Modeling with the R Package brms”. In: *The R Journal* 10.1 (2018), pp. 395–411. DOI: 10.32614/RJ-2018-017.

[183] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. A. Brubaker, J. Guo, P. Li, and A. Riddell. “Stan: A probabilistic programming language”. In: *Journal of Statistical Software* 76 (2017).

[184] W. Levén, H. Broman, T. Besker, and R. Torkar. “The broken windows theory applies to technical debt”. In: *Empirical Software Engineering* 29.4 (May 2024), p. 73. ISSN: 1573-7616. DOI: 10.1007/s10664-024-10456-6.

[185] A. Gelman, A. Vehtari, D. Simpson, C. C. Margossian, B. Carpenter, Y. Yao, L. Kennedy, J. Gabry, P.-C. Bürkner, and M. Modrák. *Bayesian Workflow*. 2020. DOI: 10.48550/arXiv.2011.01808. arXiv: 2011.01808 [stat.ME].

[186] C. Kleiber and A. Zeileis. "Visualizing count data regressions using rootograms". In: *The American Statistician* 70.3 (2016), pp. 296–303.

[187] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. "Comparison and Evaluation of Clone Detection Tools". In: *IEEE Transactions on Software Engineering* 33.9 (2007), pp. 577–591. DOI: 10.1109/TSE.2007.70725.

[188] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. "Feature location in source code: a taxonomy and survey". In: *Journal of Software: Evolution and Process* 25.1 (2013), pp. 53–95.

[189] C. Cassé, P. Berthou, P. Owezarski, and S. Josset. "Using Distributed Tracing to Identify Inefficient Resources Composition in Cloud Applications". In: *2021 IEEE 10th International Conference on Cloud Networking (CloudNet)*. 2021, pp. 40–47. DOI: 10.1109/CloudNet53349.2021.9657140.

[190] M. E. Gortney, P. E. Harris, T. Cerny, A. A. Maruf, M. Bures, D. Taibi, and P. Tisnovsky. "Visualizing Microservice Architecture in the Dynamic Perspective: A Systematic Mapping Study". In: *IEEE Access* 10 (2022), pp. 119999–120012. DOI: 10.1109/ACCESS.2022.3221130.

[191] B. Li, X. Peng, Q. Xiang, H. Wang, T. Xie, J. Sun, and X. Liu. "Enjoy your observability: an industrial survey of microservice tracing and analysis". In: *Empirical Software Engineering* 27 (2022), pp. 1–28. DOI: 10.1007/s10664-021-10063-9.

[192] A. Gelman and J. Hill. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Analytical Methods for Social Research. Cambridge, UK: Cambridge University Press, 2006. ISBN: 978-0521686891.

[193] A. Sundelin. *epkanol/observability-tracing-experiential-learning: First formal release*. Version v1.0.0. Aug. 2025. DOI: 10.5281/zenodo.16790168.

[194] M. Staron. *Action Research in Software Engineering*. Cham, Switzerland: Springer, 2020. ISBN: 978-3-030-32609-8.

[195] D. A. Kolb. *Experiential Learning: Experience as the Source of Learning and Development*. 2nd ed. Upper Saddle River, NJ, USA: Pearson Education, 2014. ISBN: 978-0133892406.

[196] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. “Above the clouds: A berkeley view of cloud computing”. In: *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS* 28.13 (2009), pp. 1–23.

[197] P. Jamshidi, A. Ahmad, and C. Pahl. “Cloud Migration Research: A Systematic Review”. In: *IEEE Transactions on Cloud Computing* 1.2 (2013), pp. 142–157. DOI: 10.1109/TCC.2013.10.

[198] X. Zhou, S. Li, L. Cao, H. Zhang, Z. Jia, C. Zhong, Z. Shan, and M. A. Babar. “Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry”. In: *Journal of Systems and Software* 195 (2023). DOI: 10.1016/j.jss.2022.111521.

[199] D. Taibi, V. Lenarduzzi, and C. Pahl. “Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation”. In: *IEEE Cloud Computing* 4.5 (2017), pp. 22–32. DOI: 10.1109/MCC.2017.4250931.

[200] M. Fowler. *Strangler Fig*. [Online; accessed 5-Mar-2025]. 2024. URL: https://martinfowler.com/bliki/StranglerFigApplication.html.

[201] B. C. Tak, B. Urgaonkar, and A. Sivasubramaniam. “Cloudy with a chance of cost savings”. In: *IEEE Transactions on Parallel and Distributed Systems* 24.6 (2013), pp. 1223–1233. DOI: 10.1109/TPDS.2012.307.

[202] J. a. Bosch. “Speed, Data, and Ecosystems: The Future of Software Engineering”. In: *IEEE Software* 33.1 (2016), pp. 82–88. DOI: 10.1109/ms.2016.14.

[203] L. Hou and R. J. Jiao. “Data-informed inverse design by product usage information: a review, framework and outlook”. In: *Journal of Intelligent Manufacturing* 31.3 (2020), pp. 529–552. DOI: 10.1007/s10845-019-01463-2.

[204] M. Villamizar, O. Garces, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil. “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud”. In: *10th Colombian Computing Conference, 10CCC 2015*. 2015, pp. 583–590. DOI: 10.1109/ColumbianCC.2015.7333476.

[205] M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang. “Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures”. In: *16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2016*. 2016, pp. 179–182. DOI: 10.1109/CCGrid.2016.37.

[206] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann. “Microservices Migration in Industry: Intentions, Strategies, and Challenges”. In: *Proceedings – 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*. IEEE. 2019, pp. 481–490. DOI: 10.1109/ICSME.2019.00081.

[207] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi. “From monolithic systems to Microservices: An assessment framework”. In: *Information and Software Technology* 137 (2021). ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2021.106600.

[208] M. H. Hasan, M. H. Osman, N. I. Admodisastro, and M. S. Muhammad. “Legacy systems to cloud migration: a review from the architectural perspective”. In: *Journal of Systems and Software* 202 (2023), p. 111702. DOI: 10.1016/j.jss.2023.111702.

[209] V. Andrikopoulos, T. Binz, F. Leymann, and S. Strauch. “How to adapt applications for the Cloud environment: Challenges and solutions in migrating applications to the Cloud”. In: *Computing* 95.6 (2013), pp. 493–535. DOI: 10.1007/s00607-012-0248-2.

[210] F. Leymann, U. Breitenbücher, S. Wagner, and J. Wettinger. “Native cloud applications: Why monolithic virtualization is not their foundation”. In: *Communications in Computer and Information Science* 740 (2017), pp. 16–40. DOI: 10.1007/978-3-319-62594-2_2.

[211] Y. Chen and R. Sion. “To cloud or not to cloud? Musings on costs and viability”. In: *Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC 2011*. 2011, pp. 1–7. DOI: 10.1145/2038916.2038945.

[212] C. Fehling, P. Arbitter, W. Schupeck, R. Retter, and F. Leymann. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Wien, Austria: Springer, 2014. ISBN: 978-3709115671.

[213] E. Walker. “The real cost of a CPU hour”. In: *Computer* 42.4 (2009), pp. 35–41. DOI: 10.1109/MC.2009.135.

[214] I. Konstantinou, E. Floros, and N. Koziris. “Public vs private cloud usage costs: the StratusLab case”. In: *Proceedings of the 2nd International Workshop on Cloud Computing Platforms, CloudCP 2012 – Co-located with EuroSys 2012*. 2012, pp. 1–6. DOI: 10.1145/2168697.2168700.

[215] M. Mao, J. Li, and M. Humphrey. “Cloud auto-scaling with deadline and budget constraints”. In: *Proceedings - IEEE/ACM International Workshop on Grid Computing*. IEEE. 2010, pp. 41–48. DOI: 10.1109/GRID.2010.5697966.

[216] E. Boza, C. Abad, M. Villavicencio, S. Quimba, and J. A. Plaza. “Reserved, on demand or serverless: Model-based simulations for cloud budget planning”. In: *2017 IEEE 2nd Ecuador Technical Chapters Meeting, ETCM 2017*. Vol. 2017-January. IEEE. 2018, pp. 1–6. DOI: 10.1109/ETCM.2017.8247460.

[217] Q. Zhu and G. Agrawal. “Resource provisioning with budget constraints for adaptive applications in cloud environments”. In: *HPDC 2010 – Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. 2010, pp. 304–307. DOI: 10.1145/1851476.1851516.

[218] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[219] J. Saldaña. *The Coding Manual for Qualitative Researchers*. London, UK: SAGE Publications, 2016. ISBN: 978-1-4739-0248-0.

[220] F. D. Davis. “Perceived usefulness, perceived ease of use, and user acceptance of information technology”. In: *MIS quarterly* (1989), pp. 319–340.