Design Choices in Building an MSR Tool: The Case of Kaiaulu

Carlos Paradis[1], Rick Kazman[1]

[1]University of Hawaii, Honolulu, HI 86822, United States

Abstract

Background: Since Alitheia Core was proposed and subsequently retired, tools that support empirical studies of software continue to be proposed, such as Codeface, Codeface4Smells, GrimoireLab and SmartSHARK, but they all make different design choices with overlapping functionality. Aims: We seek to understand the design decisions adopted on these tools–good and bad–and their consequences to understand why their authors reinvented functionality already present in other tools, and to help inform the design of future tools. Method: We used action research to evaluate the tools, and determine principles and anti-patterns to motivate a new tool design. Results: We identified 7 major design choices among the tools: 1) Abstraction Debt, 2) the use of Project Configuration Files, 3) the choice of Batch or Interactive Mode, 4) Minimal Paths to Data, 5) Familiar Software Abstractions, 6) Licensing and 7) the Perils of Code Reuse. Building on the observed good and bad design decisions, we created our own architecture and implemented it as an R package. Conclusions: Tools should not require onerous setup for users to obtain data. Authors should consider the conventions and abstractions used by their chosen language and build upon these instead of redefining them. Tools should encourage best practices in experiment reproducibility by leveraging self-contained and readable schemas that are used for tool automation, and reuse must be done with care to avoid depending on dead code.

Keywords

mining software repositories, design choices, action research

1. Introduction

Research into software architecture and quality requires the analysis of large quantities of data. For researchers this often means mining data from multiple open source software projects. Pre-processing data, calculating metrics and flaws, and synthesizing composite results from a large corpus of project artefacts is a tedious and error prone task lacking immediate scientific value [1]—it is seen merely as a means to an end. This was the motivation for the Alitheia Core [1], which was made available in 2009 for the software engineering community. It provided features for data collection, integration and analysis services and emphasized an easy to use extension mechanism. Yet, as of today, Alitheia Core is a dormant (read-only) project in GitHub[1] and several other tools replicate at least some of its functionality.

What went wrong? Why have many tools re-implemented the same "tedious and error prone" tasks? And do the current tools live up to the promise of Alitheia Core? In this work, we revisit lessons learned by the Alitheia Core authors and the design choices made by the other more recent tools using an action research [2] approach.

Our contributions in this paper are twofold: first, we present a set of key design decisions derived from the aforementioned tools which either facilitated or hindered reusability, reproducibility, interoperability and extension of functionality. Second, we present our tool, Kaiaulu[2]., which builds upon the design decisions made from these tools, and which we believe fills a gap in the existing mining software repositories ecosystem.

2. Studied Tools and Lessons Learned

The tools that we studied are Codeface [3], Codeface4Smells [4], GrimoireLab [5, 6] SmartSHARK, [7, 8] and PyDriller [9]. We now present our observations regarding the strengths and weaknesses of these tools in terms of their design choices and note, throughout the work, lessons learned by the authors of Alitheia Core [1] presented in [10]. Many of these lessons are applicable and worthy of consideration in new tools with similar intents. We employed an action research methodology in studying these tools, but do not describe the details of that research here, due to space limitations.

MSR4SA Workshop at ECSA 2021

cvas@hawaii.edu (C. Paradis); kazman@hawaii.edu (R. Kazman)
https://carlosparadis.github.io (C. Paradis); https://shidler.hawaii.edu/itm/directory/rick-kazman (R. Kazman)
0000-0002-3062-7547 (C. Paradis); 0000-0003-0392-2783 (R. Kazman)

[1]https://github.com/istlab/Alitheia-Core

[2]The documentation for the R package can be found at https://github.com/sailuh/kaiaulu

2.1. Abstraction Debt

We have observed different levels of abstraction employed in the surveyed tools, ranging from applications that are built as monoliths to those built from smaller components. This is consistent with what has been noted in machine learning systems as abstraction debt [11], i.e. a lack of key abstractions to support the functions and growth of MSR tools.

Codeface was created as a monolithic application, in which an entire project's Git log or mailing list is analyzed. It abstracts a complete end-to-end pipeline, implemented by a command line interface (CLI), and outputs a database dump of a project. It is therefore difficult for other applications to build on some of its unique features, for example, using its Git log parser that parses at function (rather than file) granularity.

Both GrimoireLab and SmartSHARK define several components, each with its own CLI, but the component abstractions they employ are not the same. To provide a point of comparison, Grimoire's Lab Perceval provides a CLI to obtain data from many data sources (e.g. GitHub, Git, Bugzilla, Jira, mailing lists, etc), serving as a single interface for data collection. In contrast, SmartSHARK defines its abstraction per data source type and, in the case of data acquisition, at a more fine-grained level than Perceval. For example, consider issueShark and vcsShark, two components of SmartShark. IssueShark defines abstractions for different types of issues tracker sources, and vcsShark for different types of version control systems. SmartSHARK's abstractions facilitate defining additional features specific to a data source type, such as separating static vs. dynamic data in issue trackers (e.g. creation time of the issue vs. comments), regardless of its underlying implementation (e.g. Jira or Bugzilla)[3].

Pydriller is a single component and is smaller in scope as it only abstracts Git repositories. However it is different from the other tools in that it provides an API instead of a CLI. Its motivation is also different: it wraps around PythonGit, which in itself provides a Pythonic API to nearly all features of Git, to provide an API catered towards mining software repositories only. In providing just a subset of Git functionality, it exposes functionality catering specifically to the needs of mining repositories.

The decision between choosing a CLI or API has tradeoffs. An issue with command line only interfaces occurs when an end-user may be interested in a different abstraction of the data not preconceived by the authors. However an API requires the user to be familiar with the programming language the tool was built on top of, whereas a CLI does not.

From the above we derive the following lessons learned: End-to-end pipelines such as Codeface's limit the ability of other researchers to build on top of them. Defining more specific abstractions per data type, whether via CLI or API as issueShark and PyDriller do, facilitates building additional functionality specific to a particular data type, or audience. Moreover, CLIs can be built on top of a well-defined API, providing the benefit of both interfaces, as we do in Kaiaulu.

[3]https://github.com/smartshark/issueSHARK#introduction

2.2. Tool Configuration Files vs Project Configuration Files

In [8], the authors of SmartSHARK noted that one of their goals was to support replication through the storage of data in a single harmonized schema. Replication, it is argued, is supported by a common dataset. However, we have observed that replication is also being done within configuration files in Codeface.

Codeface uses a concept we named project configuration files. These files provide a single compact source where parameters associated with the acquisition and manipulation of a dataset can be stored. Project configuration file parameters are required for tool execution, and they are a pragmatic, lightweight and human-readable way to specify reproducible results. Project configuration files also save time when a project is re-analyzed in other studies, as some project-specific information may not be obvious from the dataset alone.

Of all the tools we have reviewed, only Codeface provides users with a means to specify project configuration files. This led to a large collection of project configurations that have been versioned in Codeface over time [4]. This information, which supports repeatability, may otherwise not have been possible (or at least easy) to reconstruct if all that was shared was the data.

We note that externalizing parameter choices in data acquisition and manipulation tasks has been more prominent in machine learning frameworks, for example to define experiments in configuration files, [5] which include machine learning model selection and choice of model hyper-parameters [12].

From the above, we derive the following lessons: integrating configuration files that are human-readable and leveraged by the tool can enable reproducibility, without the hurdles of sharing large quantities of primary data.

2.3. Batch Mode, Interactive Mode, and Literate Programming

As we noted before, with the exception of PyDriller, every tool defines a CLI, but not an API. This means the only

[4]See https://github.com/siemens/codeface/tree/master/conf and https://github.com/maelstromdat/codeface4smells_TR/tree/master/Configurations for Codeface and Codeface4Smells respectively

[5]https://xnmt.readthedocs.io/en/latest/experiment_config_files.html

way to interact with these tools is batch mode. Meanwhile, PyDriller does not offer a CLI, only an API, which confers its users the ability to leverage Python's interactive mode to *explore* the data. However, it does not include a CLI for batch mode processing, for out-of-the-box data acquisition, processing or data analysis. What we observe then is that existing tools decide on either CLI or API, but not both. We believe, however, that the mining of software repositories requires a tool capable of both, supporting an iterative process of data exploration, and when concluded, a way to enact batch processing to scale up.

To illustrate our claim—as no existing tool provides both capabilities—we provide a few examples: in a recent socio-technical study, we needed to do identity matching, applying heuristics that have been published by other authors (e.g. [13, 14]) to assign identities to developers who use different names and e-mails in version control systems and mailing lists. Consider the case where we chose the simplest method, where developers whose name or e-mail match are assigned the same id. At first glance, this seems like a reasonable assumption. However, it was due to experimenting interactively with the identity matching API that we discovered that all core developers, due to the use of an issue tracking system, ended up sharing the same e-mail address. We noted this case as a unit test until a better heuristic could be found, and then examined the data for other cases until we were satisfied with the results. We then saved the observed parameters in a project configuration file, and used it to deploy a batch process to collect various computationally intensive architectural metrics.

We have had similar experience in determining and testing heuristics to filter files in a repository, or determining the method that developers adopt to annotate issue numbers in commit messages. Because each project may apply its own conventions, tools that offer an experimentation capability, and then defer mass data processing to batch more efficiently support the full workflow of a researcher in mining software repositories.

The described interactive data explorations could certainly have been done in a Python or R session, but it is better to leverage literate programming using, for example, Python or R Notebooks, so that the rationale of the design experiment is not lost. However, care must be taken to not extensively rely on notebooks without further refactoring functionality into the code base, leading to dead experimental code paths [11].

Our learned lessons here were: existing tools choose either APIs or CLIs (supporting batch or interactive modes). However, making both interfaces available will better support users in their various research efforts in mining software repositories. The use of Notebooks to illustrate and explain the API complements the API, provided functionality is not entirely written in Notebooks. In Kaiaulu, we leverage both APIs and Notebooks, which is a common practice in R packages, therefore avoiding abstraction debt.

2.4. Minimal Paths to Data

According to [10, p.233], the effort required to learn how infrastructure code works has to be proportional to the gains and account for deprecation. We agree with this observation. Let us look at how existing tools manage this concern.

When using GrimoireLab components (in particular Perceval) the minimal path to data is surprisingly short. Provided with a Git repository URL, or a local copy, it will output a JSON file to stdout. Likewise, provided with a URL to a website mbox or local file, it will also provide a JSON file to stdout. A developer can easily integrate wrappers to its CLI, and users can easily obtain data for a project of interest. In this ecosystem, a database is available, but it is optional: users need not to concern themselves with learning GrimoireLab's Elastic Search database to obtain data.

This is in contrast to Codeface and SmartSHARK, both of which require user familiarity with MySQL and MongoDB respectively, along with their data model schemas to obtain the equivalent version control system and mailing list data. The minimal path to data in these cases is much longer, including the setup overhead and integration with other tools.

When data integration is sought in the database, GrimoireLab retains its approach of keeping the data closest to source, and not harmonizing it in a schema that facilitates integration [8]. Codeface's MySQL and SmartSHARK's MongoDB provide a harmonized schema, which makes it easier for users to store the various types of data.

In the case of PyDriller, which provides an API, the minimal path to data requires familiarity with the Python programming language. This offers the convenience of reshaping the data to the user's final need, but adds an overhead to the user for familiarization with the API, instead of just the raw data schema from the source of interest (which the user is likely already familiar with for their research purposes). One researcher [15, p.39] who extended Codeface4Smells identified a problem of Pipeline Jungles [11], due to heavy reliance on a folder hierarchy and file name conventions.

Our lessons learned here were: databases need not be a requirement to provide users with various data sources. This also simplifies component reuse by other tools and decreases the likelihood of reinventing the wheel. Providing a minimal path does not exclude providing a database for researchers, as Perceval shows. However providing a harmonized schema can save researchers from having to reimplement code to integrate the same kinds of infrastructure over and over. Lastly, providing an API gives some *flexibility to users to reshape the data with the tool. But user familiarity with the programming language and API is a kind of overhead and this does not seem ideal, as the data could be provided directly via a CLI leaving a task for the researcher to adapt it in their own programming language. As such, we believe having available a CLI that outputs the data as Perceval does, and a harmonized schema as in Codeface and SmartSHARK, provides the best combination.*

2.5. Other Design Decisions

We briefly mention here other (more minor) design decisions that we believe may cause difficulties in adoption.

Familiar Software Abstractions. Both Perceval and PyDriller leverage a common interface for end-users. They are both Python libraries, and provide the expected interactions for CLI and API respectively. In Perceval's CLI, provided with a list of parameters and flags, data is output to stdout. PyDriller exposes an API, an extension to a programmer's familiar programming paradigm. This is in contrast to ecosystems that define a different abstraction, such as SmartSHARK, where detailed instructions must be followed to extend its functionality [6]. Extension instructions are also not available for Perceval or Codeface.

Licensing. Another important consideration in reusing a code component is how permissive its license is. For example, stringr, an R package to manipulate strings used by XGBoost, a popular machine learning algorithm, was replaced by stringi, another R package to manipulate strings, solely based on the difference in licenses.[7] Similar reasoning also led an R package that represents data tables efficiently to adopt a different license because the existing license "could be interpreted as preventing closed-source products from using data.table"[8]. Lack of clarity on interactions of open source licenses has been reported by [16]. Among the tools we studied, we have observed the following licenses: Codeface adopts GPL 2.0, PyDriller Apache 2.0, SmartSHARK Apache 2.0, and Grimoire's Lab GPL 3.0 and LGPL 3.0.

Perils of Code Reuse. With the availability of package managers such as CRAN and PyPi which greatly facilitate code reuse, you can declare dependencies on others' code instead of copying it into your own project, taking advantage of their functionality without assuming the burden of maintenance. However code interdependence also poses risks [17], such as dependencies going extinct [18]. Hence, care has to be taken to avoid dependencies to non-maintained third-party code.

An interesting example occurs in mecoSHARK[9] through a chain of dependencies which exemplifies the concern posed here. mecoSHARK is a component that serves as a wrapper for OpenStaticAnalyzer[10], with a last commit date of July 13, 2018. In turn, OpenStaticAnalyzer also wraps several other dependencies, including FindBugs [11], last released in March 15, 2015. In its bug tracker[12], FindBugs requests for bugs to no longer be reported, noting that SpotBugs[13], FindBugs' successor, should be used instead. This confirms that the mecoSHARK wrapper, which provides OpenStaticAnalyzer functionality to SmartSHARK,is now dependent on dead code, further increasing the burden of the SmartSHARK ecosystem maintainers. Nonetheless, SmartSHARK's approach to wrap black-box packages into common APIs is considered good practice [11].

As a means to mitigate this risk, relying on and contributing work to open source communities that more carefully assess the health of projects and try to maintain them, such as the Apache Software Foundation, ROpenSci[14], and CHAOSS[15] may be an important consideration. For example, ROpenSci accepts R packages via a streamlined peer review process and, for accepted packages, provides community support, package promotion, and fast-track publication to journals[16].

3. The Kaiaulu R Package

Based on the above observations and lessons learned, we now describe the design decisions behind the Kaiaulu R package that we created. We created this in R, as we believe the R ecosystem provides a reasonable approach to implementing the lessons learned from existing tools a single architecture.

Batch Mode, Interactive Mode, and Literate Programming in Kaiaulu. We chose to use the R language[17], in contrast to existing packages that use Python (with the exception of Codeface, which includes both Python and R, but not as a package) due to the familiarity of the authors with the language and a preference for its package architecture.

Minimally, the structure of an R package consists of the package metadata and its API. In addition, the R ecosystem promotes best practices to include vignettes, which leads R users to expect an API and R Notebooks when installing packages from CRAN (The Comprehensive R

[6]https://smartshark.github.io/plugin/tutorial/python
[7]https://github.com/dmlc/xgboost/issues/1338
[8]https://github.com/Rdatatable/data.table/pull/2456
[9]https://github.com/smartshark/mecoSHARK

[10]https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer
[11]http://findbugs.sourceforge.net/
[12]https://sourceforge.net/p/findbugs/bugs/1487/
[13]https://github.com/spotbugs/spotbugs
[14]https://ropensci.org/about/
[15]https://chaoss.community/
[16]https://devguide.ropensci.org/softwarereviewintro.html#whysubmit
[17]https://www.r-project.org/

Archive Network).[18] CRAN also treats R Notebooks as first class citizens in an R package[19] showing on each package's website any R Notebooks available. Because of R package structure, complying with familiar software abstractions (see Section 2.5) automatically brings the benefits of literate programming (see Section 2.3).

Abstraction Debt in Kaiaulu. R natively supports tables and vectors as data types, which is a familiar abstraction for data analysts. To capitalize on this, Kaiaulu's *parse_* functions map most data sources (Git logs, mailing lists, file dependencies, software vulnerability feeds, metrics, etc.) as tables with standardized column naming, which allows for quick identification of what data can be combined. Kaiaulu also offers various *transform_to_network_* functions to represent and interactively visualize these networks[20] which in turn enable more complex socio-technical analyses at different granularities: functions, files, classes, etc.

Tool Configuration Files vs Project Configuration Files in Kaiaulu. Following the design choice of Codeface (see Section 2.2), and building on best practices for machine learning configuration files [11] we implemented project configuration files using YAML. Because we externalize all parameters in project configuration files, an important concern is that the file does not grow overly complex, requiring documentation of its own. That is, we do not wish the minimal path to data to increase as new features are added, as we discuss next.

Minimal Path to Data in Kaiaulu. As discussed in Section 2.4, it is important that the path to data remains as small and easy as possible. How should users be introduced to project configuration files? Should a manual page be dedicated to explain its various parameters? We again build upon familiar concepts, specifically with the intent of applying the rule of least surprise [19, Ch.11][21] i.e. 'do the least surprising thing'. In an R package, it is expected that R Notebooks provide examples of how to leverage the API to accomplish a task by combining multiple functions, while individual functions provide self-contained examples, which can be obtained in the R environment at any time by preceding a function name with a question mark, e.g. '*?parse_gitlog*'.

To build upon this we: 1) *Do not* create any dependency between configuration files and the API: functions take as input parameters which are familiar to any programmer, not a filepath to the project configuration file; 2) *Use* of project configuration files only in the first code block in R Notebooks to load the variables required to use the functions of the API, similar to how best practices in static programming languages encourage variable definitions at the beginning of a program ; 3) *Do* create a dependency between the CLI and the project configuration files, to facilitate batch processing and reproducibility.

Our intent is that users will first observe the R Notebooks to get a better understanding of the API for a particular task of interest, and in doing so will familiarize themselves with both the relevant portion of the API and the project configuration file. If the interest is only, for example, to understand how to parse Git logs, using for example the Git log R notebook, then users should not be concerned with specifying the mailing list. When comfortable, users can then use their newfound understanding to scale the analysis to the entire project using the configuration file for the CLI, build their own analyses as vignettes, or define new CLI interfaces. This design is consistent with a mining software repositories workflow, in which a researcher should first explore the data qualitatively to assess threats to validity, before scaling up data processing in batch mode without clarity of what assumptions the tool is making using default parameters or arbitrary thresholds.

Kaiaulu also further decreases the minimal path to data in terms of how it handles third party dependencies. Users need only concern themselves with installing dependencies for their task of interest. For example, if the interest is only to parse Git logs, they need only set up Perceval, and provide its binary path as a parameter to Kaiaulu's *parse_gitlog* to obtain the parsed data. More generally, the *parse_* API minimizes effort to researchers by transforming various tool-specific data formats, if the researcher so desires, into tables, and performing minimal processing on potentially inconsistent fields, such as file paths, to make them internally consistent.

3.1. Kaiaulu's Features

In [10], Gousios notes a lesson he learned in the transition from Alitheia Core to GHTorrent: 'Open now trumps open when it is done'. Kaiaulu is a new tool, but it already includes substantial usable functionality: parsers for Git, mailing list mbox archives (leveraging Perceval), and identity matching which enables socio-technical studies. And it is easy to add new capabilities, building upon the infrastructure we have created. Of note, unique to Kaiaulu is an interface to visualize networks interactively[22]. Kaiaulu has, as a core abstraction, a network model. We use this to, for example, relate and analyze authors, committers, commits, files, CVEs and CWEs rather than merely reporting project-level metrics.

[18] https://cran.r-project.org/web/packages/

[19] See for example under Vignettes: https://cran.r-project.org/web/packages/ggplot2/index.html

[20] https://github.com/sailuh/kaiaulu/blob/master/R/network.R

[21] Also publicly available at: http://www.catb.org/~esr/writings/taoup/html/ch11s01.html

[22] For examples, see the various R Notebooks available in Kaiaulu documentation.

4. Conclusions and Future Work

In this paper, through an action research approach, we have determined a set of key design decisions observed in existing tools, and iteratively developed Kaiaulu, an R package for mining software repositories building on our observations and lessons learned.

In Kaiaulu we have implemented, and are currently in the process of openly releasing, a comprehensive set of capabilities to mine, analyze, and visualize software repositories, including social smells [4], architecture smells and metrics [20], and bug timelines based on prior work by other authors. Kaiaulu is licensed under MPL 2.0.

Acknowledgments

The authors wish to thank Damian Tamburri, Corne Broere, and Massimo Manca for identifying bugs, providing use cases and suggesting features for Kaiaulu. This work was supported in part by the US National Science Foundation under grant CCF-1817267.

References

[1] G. Gousios, D. Spinellis, Alitheia core: An extensible software quality monitoring platform, in: 2009 IEEE 31st International Conference on Software Engineering, 2009, pp. 579–582.

[2] S. Easterbrook, J. Singer, M.-A. Storey, D. Damian, Selecting Empirical Methods for Software Engineering Research, Springer, 2008.

[3] M. Joblin, S. Apel, C. Hunsen, W. Mauerer, Classifying developers into core and peripheral: An empirical study on count and network metrics, in: IEEE/ACM 39th International Conference on Software Engineering, 2017, pp. 164–174.

[4] D. Tamburri, F. Palomba, R. Kazman, Exploring community smells in open-source: An automated approach, IEEE Transactions on Software Engineering (2019) 1–1.

[5] D. Moreno, S. Dueñas, V. Cosentino, M. A. Fernandez, A. Zerouali, G. Robles, J. M. Gonzalez-Barahona, Sortinghat: Wizardry on software project members, in: IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings, 2019, pp. 51–54.

[6] S. Dueñas, V. Cosentino, G. Robles, J. M. Gonzalez-Barahona, Perceval: software project data at your will, in: Proc. 40th International Conference on Software Engineering: Companion Proceeedings, ACM, 2018, pp. 1–4.

[7] F. Trautsch, S. Herbold, P. Makedonski, J. Grabowski, Addressing problems with replicability and validity of repository mining studies through a smart data platform, Empirical Software Engineering 23 (2018).

[8] A. Trautsch, F. Trautsch, S. Herbold, B. Ledel, J. Grabowski, The smartshark ecosystem for software repository mining, arXiv preprint arXiv:2001.01606 (2020).

[9] D. Spadini, M. Aniche, A. Bacchelli, PyDriller: Python framework for mining software repositories, in: Proc. 26th ACM Joint Proceedings of ESEC/FSE, ACM Press, 2018, pp. 908–911.

[10] T. Menzies, L. Williams, T. Zimmermann, Perspectives on Data Science for Software Engineering, 1st ed., Morgan Kaufmann Publishers Inc., 2016.

[11] D. Sculley, et al., Hidden technical debt in machine learning systems, in: Proc. 28th International Conference on Neural Information Processing Systems - Volume 2, NIPS'15, MIT Press, Cambridge, MA, USA, 2015, p. 2503–2511.

[12] G. Neubig, et al., XNMT: The extensible neural machine translation toolkit, in: Conference of the Association for Machine Translation in the Americas Open Source Software Showcase, 2018.

[13] C. Bird, A. Gourley, P. Devanbu, M. Gertz, A. Swaminathan, Mining email social networks, in: Proc. International Workshop on Mining Software Repositories, ACM, 2006, p. 137–143.

[14] J. Zhu, J. Wei, An empirical study of multiple names and email addresses in oss version control repositories, in: IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), 2019, pp. 409–420.

[15] F. Giarola, Detecting code and community smells in open-source: an automated approach, Master's thesis, Politecnico di Milano, 2016.

[16] D. A. Almeida, G. C. Murphy, G. Wilson, M. Hoye, Do software developers understand open source licenses?, in: IEEE/ACM 25th International Conference on Program Comprehension, 2017, pp. 1–11.

[17] M. Valiev, B. Vasilescu, J. Herbsleb, Ecosystem-level determinants of sustained activity in open-source projects: A case study of the pypi ecosystem, in: Proc. 26th ACM Joint Proceedings of ESEC/FSE, ACM, 2018, p. 644–655.

[18] J. Coelho, M. Valente, Why modern open source projects fail, in: Proc. 11th Joint Meeting on Foundations of Software Engineering, ACM, 2017, p. 186–196.

[19] E. S. Raymond, The Art of UNIX Programming, Pearson Education, 2003.

[20] R. Mo, Y. Cai, R. Kazman, L. Xiao, Q. Feng, Architecture anti-patterns: Automatically detectable violations of design principles, IEEE Transactions on Software Engineering (2019).