The Engineering Implications of Code Maintenance in Practice

Noah Lee
Meta Platforms, Inc.
Menlo Park, USA
noahlee@fb.com

Rui Abreu
Meta Platforms, Inc.
Menlo Park, USA
rui@computer.org

Mehmet Yatbaz
Meta Platforms, Inc.
Menlo Park, USA
memo@fb.com

Hang Qu
Meta Platforms, Inc.
Menlo Park, USA
quhang@fb.com

Nachiappan Nagappan
Meta Platforms, Inc.
Seattle, USA
nnachi@fb.com

***Abstract*—Allowing developers to move fast when evolving and maintaining low-latency, large-scale distributed systems is a challenging problem due to i) sheer system complexity and scale, ii) degrading code quality, and iii) difficulty of performing reliable rapid change management while the system is in production. Addressing these problems has many benefits to increase system developer efficiency, reliability, performance, as well as code maintenance. In this paper, we present a real-world case study of an architectural refactoring project within an industrial setting. The system in scope is our codenamed *ItemIndexer* delivery system (*I2DS*), which is responsible for processing and delivering a large number of *items* at rapid speed to billions of users in real time. *I2DS* is running in production, refactored live over a period of 9 months, and assessed through impact validation studies that show a 42% improvement in developer efficiency, 87% improvement in reliability, 20% increase in item scoring, a 10% increase in item matching, and 14% CPU savings.**

***Index Terms*—Architectural Refactoring, Code Quality, Performance, Reliability, Developer Efficiency.**

I. Introduction

Code is increasingly getting more complex and difficult to evolve and maintain [1], [4], [9], [11]. As software systems evolve over time, complexity growth is a major challenge and, if not proactively managed, can lead to failed projects [27], low engineer sentiment [10], slow and unreliable systems [3], reduced engineering efficiency [7], increased maintenance costs [18], and lost opportunities impacting the company's bottom line.

This is especially true for low latency distributed systems that serve millions of items at a billion user scale due to the sheer complexity at the data, code, and system level. As more engineers contribute to the code over time, ensuring high-quality code is difficult due to the need to balance multiple objectives when refactoring code, such as developer efficiency, reliability, and performance.

Code quality refers to the internal and external characteristics of code and the degree of conformance to a given design from a functional and non-functional perspective. Poor software quality in the US alone is estimated to be worth 2 trillion dollars[1]. The ISO/IEC 25010 standard [24] defines code quality through the following factors: i) functional suitability, ii) reliability, iii) performance, iv) operability, v) security, vi) compatibility, vii) maintainability, and viii) transferability.

[1]https://www.it-cisq.org/cost-of-poor-software-quality-in-the-us/index.htm (accessed July 13, 2022)

Architectural refactoring is the process of restructuring and optimizing the internal structure of the code without altering its external behavior [1]. Amongst the benefits of architectural refactoring are i) clean code, ii) improved performance, iii) increased developer efficiency, iv) reduced technical debt, v) fewer bugs, vi) improved readability, and vii) easier maintainability.

The process of refactoring code is often complicated when dealing with legacy code, such as monolithic data and compute classes, as well as complex interactions between new and legacy code. Optimizing multiple aspects of code quality requires a cautious balancing act to improve one without regressing the other aspects [13]. As an example implementing performance optimizations easily can lead to more complex code, which in turn would adversely affect code quality from a maintainability or reliability perspective. Embarking on large scope architectural refactoring for production systems poses risks and is further made difficult as the code is constantly changing with 200 change requests/month submitted by thousands of engineers.

Lastly, measuring the multifaceted aspects of code quality requires investments into proper tooling and metrics to measure productivity concepts such as developer efficiency, code quality, and operational system metrics to motivate, justify, and incentivize the software maintenance work. The existing ItemIndexer codebase suffered various code quality issues, such as low cohesion, tight coupling, high code complexity, lack of testability, and the need to perform repetitive and risky performance optimizations, to name a few.

In this paper, we share our experience on a real-world architectural refactoring project at Meta. The system in scope is our codenamed *ItemIndexer* delivery system (*I2DS*), which is responsible for processing and delivering a large number of items at low latency to a large number of users in real-time. We pursue a modularization strategy with the goal of simplifying the codebase to allow our engineers to easily understand, develop, optimize, test, and maintain code. *I2DS* is running in production, refactored live over a period of 9 months, and assessed through impact validation studies that show a 42% improvement in developer efficiency, 87% improvement in reliability, 20% increase in item scoring, a 10% increase in item matching, and 14% CPU savings.

The contributions of the paper are as follows:

- We share a practical, real-world study of an architectural refactoring project of a codebase that performs large-scale item indexing for billions of users.
- We present a multifaceted impact validation study in terms of three aspects: i) developer efficiency, ii) reliability, and iii) performance gains.
- Our measure of developer efficiency is based on the notion of a cycle time that measures the end-to-end time it takes for an engineer to make a code change from conception to production.
- Within the confines of confidentiality and internal policies, we share our experience, success cases, challenges, and lessons learned.

The paper is organized as follows. Section II presents related work on software refactoring. Section III presents relevant background for some of the metrics we use and additional context on our code review system and codebase. Section IV outlines our platformization strategy, abstraction, and measurement framework. Section V presents impact validation studies and results, followed by a discussion in Section VI where we share challenges faced, insights, and best practices. We conclude the paper with threats to validity in Section VII and final remarks in Section VIII.

II. Related work

Code maintenance through refactoring is gaining more and more interest from academia and the industry with a growing portfolio of refactoring use cases such as optimal scheduling [29], recommendations [16], opportunity detection [12], and correctness testing [28]. Recent trends point towards refactoring at higher levels of abstraction (e.g. at the architectural level) [20], [21], [25] and to leverage machine learning to automate and simplify the refactoring life cycle [5].

Abid *et al.* [1] performed a systematic literature review covering 30 years of software refactoring research. They studied 3,183 papers on software refactoring and created a taxonomy to classify the prior art into structured themes. They also identified key trends, gaps, and avenues for future work and distinguished between industrial and open source related papers. Out of the 3K papers, only ten papers included a validation study in an industrial setting. The authors emphasized the need for refactoring techniques to be validated and checked for quality and reliability using industrial systems. They called out the need for industrial collaborations to bridge the gap between academic research and industry needs to produce groundbreaking research and innovation that solves complex real-world problems. In support of this work, our study reports on a large-scale software system refactoring that is used by billions of users every day. We also study the impact of our refactoring on developer productivity, reliability, and performance improvements. The scope and nature of our refactoring also differ from the reported industrial validation studies.

Golubev *et al.* [9] conducted a survey ($n = 1,183$) of IntelliJ users to ask questions about software refactoring behaviors and preferences. They found that developers spend more than 1 hour in a single session to refactor code and that developers don't prefer to use IDE-based refactoring tools. In our case study we also confirm that engineers prefer to use their domain expertise and intuition to perform software refactoring rather than rely on tools. In our case, due to the complexity of our codebase and system, appropriate refactoring tools were not readily available. In contrast, whereas their work reports on incremental small-scale (floss) refactoring our work targets an architectural refactoring spanning over multiple months.

Ivers *et al.* [11] reported on a survey (n=107) with developers to understand large-scale refactoring, its prevalence, and how tools support it. They found that developers use several categories of tools to support large-scale refactoring and rely more heavily on general-purpose tools like IDEs than on refactoring specific tools. We can confirm the findings as our engineers also mainly relied on their domain expertise to perform the refactoring.

Moser *et al.* [17] reported on a refactoring case study to assess the impact of refactoring in a close-to industrial setting. They found that refactoring can improve aspects of code quality and productivity. The refactoring team consisted of 4 developers who refactored a Java application as part of the study. They assessed the quality of the code using complexity, coupling, and cohesion metrics. Productivity was defined as lines of code (LOC) divided by effort. In comparison, our study involves similar code quality metrics but differs in how we define productivity. Our measure of productivity does not rely on LOC, but on the notion of a cycle time that measures how long it takes to bring a change from conception to production. We believe that cycle time in combination with quality metrics such as guard rails is a more appropriate metric than lines of code [22].

Wahler *et al.* [26] conducted a case study in which software engineers consulted researchers to refactor their simulation software. The Java application had grown to 30K lines of code and was deemed unmaintainable. They used a combination of static analysis with software metrics to devise a refactoring strategy. Our codebase in 5X larger in size and shares a set of different lessons learned given the scale and nature of our codebase and system.

Szoeke *et al.* [23] performed a study in which they collected a large amount of data during a refactoring phase where the developers used a (semi)automatic refactoring tool. By measuring the maintainability of the involved subject systems before and after the refactorings, they extracted valuable insights into the effect of these refactorings on large-scale industrial projects.

While the literature is rich around survey and review papers, empirical case studies and industrial experiences for large scale software systems are scarce. Most studies report empirical investigations in an open-source setting. We argue that our work brings substantial value to the community by providing additional insights and lessons learned on refactoring large-scale software systems and their validation in an industrial setting.

III. Background

In this section, we provide context on our code quality review process, the various code quality metrics, and our *I2DS* codebase.

A. The code quality review process

At Meta, we use Phabricator as the backbone of our Continuous Integration (CI) system[2] and code quality assurance process. Phabricator is a standalone application that sits on top of our CI infrastructure and is used for modern code review, through which developers submit pull requests (which we internally call diffs) and comment on each other's diffs, before they ultimately become accepted into the codebase (or are discarded) to ensure that code changes are of high quality. More than $+100K$ diffs are committed to the central repository every week, using Phabricator as a central gatekeeper, reporting, curating, and testing system. The author uploads the diff and, after checking it and potentially adding comments to guide reviewers, "publishes" the diff. The author can assign individual reviewers and/or groups of reviewers, both before and after publishing. Phabricator's UI contains specific contents of the diff (see Figure 1). This includes a title, summary, and the code changes itself.

The benefit of Phabricator is the ability to distinguish between different events that take place during the code review process. Each event and action taken during the life cycle of a code review process and interaction with the Phabricator UI is associated with an engineers's identity and associated timestamp. Events such as when a code review was accepted, abandoned, or taken over by someone else, when a reviewer resigned, and when some attributes of a diff (e.g., title, summary, and test plan) were updated are available. To the best of our knowledge, we are not aware of any other code review system that provides this level of granularity. We leverage the Phabricator data to measure aspects of developer efficiency. In 2011, Meta released Phabricator as open source as explained in subsequent sections.

B. Meta's codebase

Meta stores all of its code in a monorepo[3] (akin to other companies [2], [19]). Storing all our code in a single repository has a number of advantages. First, simplification of dependency management, such as automatic prevention of the dreaded diamond dependency problem. Second, easier and more systematic version management for libraries. Third, the possibility to perform large-scale software refactorings and account for all API interfaces and call sites. And lastly, guiding engineers to make small, incremental changes often and early in the development cycle. A disadvantage of the monorepo model is that small changes can have a large impact radius [14] and therefore requires careful understanding by engineers to prevent regressions.

C. ItemIndexer system and codebase

ItemIndexer is an in-memory indexing sharded system and part of a broader service delivery ecosystem. It hosts and serves a universe of distributed eligible items for a given user at low latency. The system works with *ItemFinder*, another service, to find the best eligible items for a user by identifying the top candidates. The *ItemIndexer* performs item processing and delivery. The codebase of *ItemIndexer* is hot, i.e. if we only consider the activity on the folder *.../itemindexer*, there are about 200 diffs being landed every month and at least $1,500$ engineers reading and working with the code every month. The codebase under has roughly about $+173$K lines of C/C++ code, $+1.5$K files and $+1.4$K functions.

D. Measuring developer efficiency

Developer efficiency refers to the highest level of performance that uses the least amount of inputs to achieve the highest amount of output. Efficiency is quantitatively determined by the ratio of useful output to total input. We map code authoring activity such as code commits to a particular diff and thereby can measure and quantify the time it takes from making the first commit of a code change to the time the associated diff lands in production. At Meta, we refer to this cycle time as *Code Gestation Time* (CGT). A diff corresponds to a code change in the system and represents a unit of work being performed by engineers. We use the notion of a cycle time to quantify and reason about developer efficiency to get an understanding of how fast developers can land a change in production. In our case, the output corresponds to a single diff, and total input the time the engineer spent on producing that diff.

E. Measuring reliability

To measure code reliability, we leverage an internal incident management system that allows us to map an incident to a particular diff in question. Our reliability metric is a simple count of production incidences over time due to a particular code, e.g., n incidences per code per time period.

F. Measuring performance

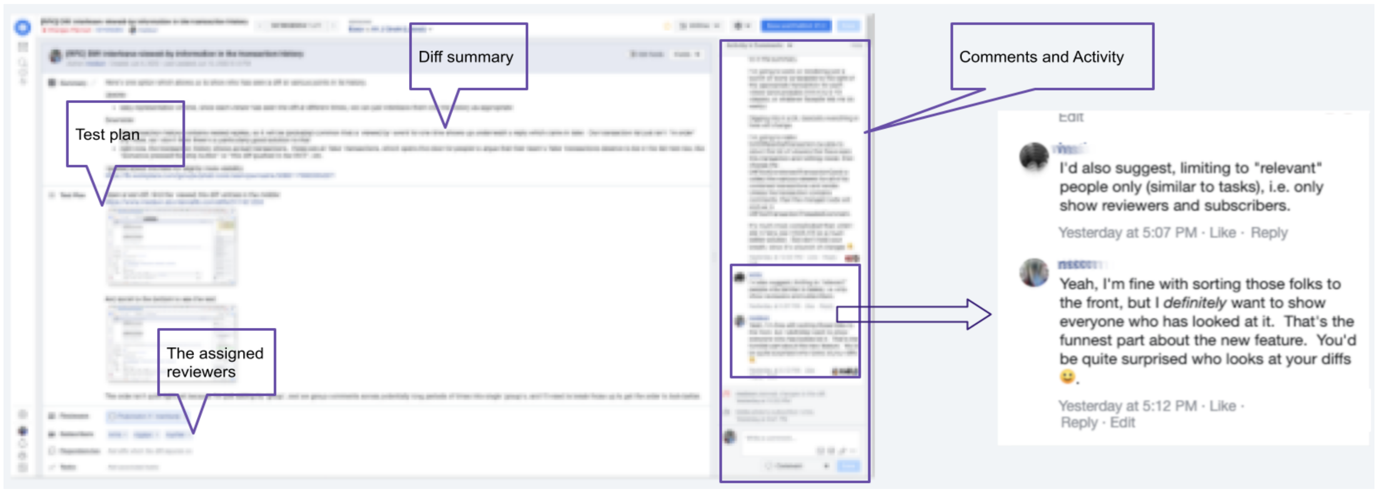
To measure performance, we mine internal performance data across various stages of the development cycle, as well as dynamic execution times of the system and the service.

G. Code quality metrics

Rather than using a single metric to quantify code quality, we prefer to have a portfolio of code quality metrics to get a multifaceted view of code quality. For our purposes, the primary metrics of interest relate to the complexity of code, the coupling, churn, and how many engineers modify the code. Our internal metric catalog has many more metrics, but for brevity and exemplary purposes, we only list a subset of the metrics below.

[2]http://phabricator.org (accessed July 13, 2022)

[3]Durham Goode. 2014. Scaling Mercurial at Facebook. https://code.fb.com/core-data/scaling-mercurial-atfacebook/ (accessed July 13, 2022)



Edit

I'd also suggest, limiting to "relevant" people only (similar to tasks), i.e. only show reviewers and subscribers.

Yesterday at 5:07 PM · Like · Reply

Yeah, I'm fine with sorting those folks to the front, but I *definitely* want to show everyone who has looked at it. That's the funnest part about the new feature. You'd be quite surprised who looks at your diffs 😀.

Yesterday at 5:12 PM · Like · Reply · Edit

Fig. 1. A redacted view of a pull request (also called diff) under code quality review in Phabricator. Authors and reviewers can interact via the diff review page. The diff under review has several sections including the diff summary, the actual changes that happened, the assigned code reviewers, the interactions between the various reviewers and author, the test case information and status, the results of static analysis, historical information, amongst other info.

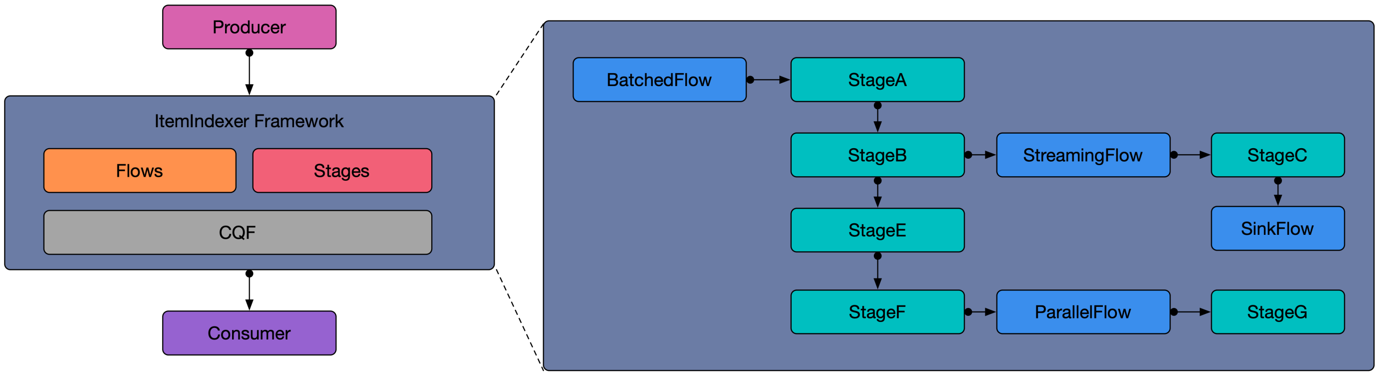


Fig. 2. The conceptual view of the code-as-platform platform strategy to refactor our *ItemIndexer* delivery system (*I2DS*). Producers are engineers who provide foundational abstractions and building blocks into the *ItemIndexer* codebase. Consumers are engineers who leverage and build upon these foundations. The more engineers consume the framework the more incentives the engineers have who produce, optimize, and maintain the Flow and Stages functionality. Therefore, we distinguish between framework producers and consumers. The main abstraction principles of the architectural refactoring are the concept of Flows and Stages. The code quality framework (CQF) provides an interface to Meta's infrastructure for measurement and monitoring purposes. Flows consists of stages, where a stage can point to a flow itself. The type of flow determines the type of concurrency such as Batched, Streaming, or Parallel Flow executions.

- **Cyclomatic Complexity (CCN)** — CCN [15] indicates the complexity of a software program. It measures the number of linearly independent paths through the source code of a program [6]. The more complex the code and the more branches, the harder it is to test the code, comprehend, maintain, and innovate on it. We use CCN at the file level by measuring the number of condition and exit points $\pi - s + 2$, where π is the number of decision points in the file, and s is the number of exit points.
- **Fanout Internal/External (FI/E)** — FI/E indicates how many different modules are used by a certain module to measure interdependency or coupling between modules. A high fanout makes code less modifiable. External imports concern imports from outside the software system (e.g., standard libraries), while internal imports concern references within the codebase itself. Fanout gives us a notion how large the potential impact radius is for a file. The higher the coupling the more risk is introduced when a change to a file is made.
- **Code Churn (last 28 days) CC (L28)** — CC(L28) measures how often a file has been changed in the last 28 days. We also examine L7, L90, and L180. For the sake of brevity, we only demonstrate L28 as the time horizon captures a clean 4 weeks of signal. Code churn gives us a signal of how often code is changing with the assumption that the more often code changes, the higher the risk for bugs and production issues.
- **Authors Count (last 28 days) AC (L28)** — AC(L28) measures the number of unique authors who changed the file in the last 28 days. See our remark about Code Churn. We follow the same principle to compute different time horizons of the metric. The author count gives us a signal of how many different engineers touch the code with the assumption that the more people touch the code, the more

risk is introduced.

For CCN and Fanout (FI/FE) we use the Multimetric[4] open source library. CC and AC are computed from internal logging data. Due to internal security policies leveraging closed-source commercial code quality frameworks is not an option for this initiative.

IV. Methodology

In this section, we describe our code platformization strategy, the code abstractions that we used to perform architectural refactoring and our code quality framework.

This initiative is motivated by the following aspirations: i) to re-architect the core components of *I2DS* to be simple to understand and innovate and ii) to build the core libraries that enable robust, observable and high-performant execution. Our approach is guided by Gall's Law [8], which states that "a complex system designed from scratch never works and cannot be made to work." One has to start over and begin with a simple working system or a set of subsystems first. Therefore our aim is to identify and implement strategies that seek simplicity rather than adding to the complexity of the current system.

The conceptual view of our architectural refactoring consists of the *ItemIndexer* Framework, engineers who produce foundational abstractions (Producers), and engineers who consume and build upon these (Consumers). The key abstractions consist of Flow and Stages. Each Flow consists of multiple Stages and a Stage can refer to other Flows recursively (see Figure 2).

A. Code platformization

By code platformization, we refer to the strategy to view and treat code as a platform. For this project a few senior engineers embarked on the architectural refactoring to produce foundational abstractions that other engineers can consume, built-upon, and leverage to build more complex business logic. This creates a network effect where senior engineers are incentivized to produce more abstractions as more and more engineers consume and benefit from it. So rather than enforcing hard guardrails and guidance on how code should be written and designed, we employed a soft guidance approach by providing and leading with the right code abstractions that other engineers can use to follow best practices and write higher quality code.

B. Status quo

The existing *ItemIndexer* codebase suffered from various code quality issues, such as low cohesion, tight coupling, high code complexity, lack of testability, and the need to perform repetitive and risky performance optimizations, to name a few. Understanding a unit of code was difficult due to low cohesion and unrelated code being intermixed with no clear responsibility boundaries. Changing code was error-prone due to tight coupling and opaque side effects, making debugging time-consuming and landing production changes risky. Code in general had high code complexity with many execution paths, which further made writing test cases more difficult and intractable. Compute logic was mixed in monolithic functions. Data logic were mixed in monolithic classes. Infra logic were coupled with business logic. The existing code had no general mechanisms to address performance optimizations for new changes and required reinventing the wheel for each new local optimization, which introduced further reliability risks. Optimization ideas like multi-phase retrieval, interruption-based timeout, latency hiding, value-based prioritization and streaming processing have great performance potential. However, these optimizations would have added complexity to the system. If not managed correctly, the complexities would have been difficult to understand, prototype, and maintain. The lack of clean, simple, and high quality code prevented our engineers from writing diffs quickly, debugging quickly, and introducing new technologies quickly. We needed a strategy that trades off developer efficiency, reliability, and performance.

C. Architectural refactoring

One consideration was to make the codebase more modular and pursue a thoughtful, staged approach to refactor *I2DS* live over a period of many months. We describe a refactoring approach that unifies the before mentioned optimization ideas. The abstraction encapsulates the concurrency logic, so the actual business logic is decoupled and remains easy to understand. At the same time, such unification simplifies implementing and maintaining aforementioned performance optimizations.

The proposed abstraction applies the actor model[5] and the Pipe and Filter pattern to decouple infra vs. business logic. It specifies concurrency as actors called "Flows" that exchange "Messages". For example, parallel ranking can be a Flow and sequential truncation of items can be a Flow. By specifying how Flows exchange Messages, we program performance optimization ideas in a unified manner. Flows handle the infra logic, specify execution modes, stopping conditions, enable configuration driven management, reusability, and allow for the generation of arbitrary compute pipelines. They provide a mechanism to unify various optimization paradigms such as latency hiding, value-based prioritization, and stream processing, which enables us to balance developer efficiency, reliability, and performance by decoupling concurrency logic from business logic in a reusable way. An example of the Flow and Stage abstraction is shown in Figure 3.

The conciseness of the code is due to the existing *futures* algorithms of our Folly library[6] (see Figure 4). We can use < 30 lines to specify batched execution and < 100 lines to specify streaming execution. Moreover, a Stage can be used to run either in a batched, streaming, or parallel mode, depending on what Flows are used.

Programmatically, a request execution is a linear chain of Flows. Each Flow runs Stages with specific execution

[4]https://github.com/priv-kweihmann/multimetric (accessed July 13, 2022)

[5]https://en.wikipedia.org/wiki/Actor_model (accessed July 13, 2022)

[6]https://github.com/facebook/folly (accessed July 13, 2022)

Flow Implementation
Flow implementation encapsulate infra logic

```
class ParallelFlow: public FlowInterf {
  public:
   ItemHandles exec(ItemHandles itemHandles) override {
      … // Infra logic
   }
}
```

Flow Specification
Gives business logic of execution policies

```
ParallelFlow indexingFlow(timeout, numThreads);
indexingFlow.add<CollectFeaturesStage>();
indexingFlow.add<ProcessFeatures>();
indexingFlow.add<Evaluate>();
itemHandles = indexingFlow.exec(itemHandles);
```

Stage
A stage model the business logic of a compute model

```
struct CollectFeaturesStage {
  static ItemHandles proc(ItemHandles itemHandles) {
    for (auto itemHandle: itemHandles)
        fetch (itemHandle.getRef<Features>());
    return sty::filter(itemHandles, isSuccess);
  }
}
```

Field
A field models the business logic of a data module

```
Using ItemHandles = std::vector<DataAPIHandle<Features, Feature, Score>>;
```

Fig. 3. An example of the high-level design of the Flow and Stage abstractions. Flows capture infra logic whereas Stages capture business logic to have a clear separation boundaries. A Flow specification orchestrates the realization of infra and business logic being executed to perform a task within I2DS with minimal code effort. Fields model the data aspect of implementing business logic.

```
class FlowInterf {
 public:
  virtual ~FlowInterf() {}
  virtual folly::SemiFuture<FeedbackSignal> onRec(
      MsgVec msgVector) = 0;
  virtual folly::SemiFuture<FeedbackSignal> onNotifyMsgEx() = 0;
};

class FlowKernel {
 public:
  FlowKernel(
      FlowIntf& nextFlow,
      std::function<MsgVec(MsgVec)> flowFunc, int64_t batchSize = 0)
      : nextFlow_(nextFlow), batchSize_(std::max(1l, batchSize)) {
    procBatchFunc_ = [flowFunc,&nextFlow](MsgVec msgVec) mutable {
      return folly::makeSemiFuture(std::move(msgVec))
          .deferVal(flowFunc)
          .deferVal(std::bind(&FlowInterf::onRec, &nextFlow, std::placeholders::_1));
    };
  }

  folly::SemiFuture<FeedbackSignal> process(
      MessageVector msgVec) noexcept {
    auto semiFutures = msgVec | ranges::views::chunk(batchSize_) |
        ranges::views::transform(ranges::to_vector) |
        ranges::views::transform(procBatchFunc_) | ranges::to_vector;
    return folly::collectAll(std::move(semiFutures))
        .deferVal(internal::mergeSignals);
  }

  folly::SemiFuture<FeedbackSignal> gen() noexcept {
    return folly::mkSemiFuture(MsgVec())
        .deferVal(procBatchFunc_);
  }

  FlowInterf& getNextFlow() { return nextFlow_; }

 private:
  FlowInterf& nextFlow_;
  const int64_t batchSize_;

  std::function<folly::SemiFuture<FeedbackSignal>(MsgVec)>
      procBatchFunc_;
};
```

Fig. 4. An example of the FlowInterface and FlowKernel implementation. The FlowInterface specifies the execution interface of a flow and the FlowKernel the common facilities of a Flow. We use our internal library Folly[8] to implement SemiFutures, which simplifies the Flow logic into a handful of lines. Folly is a library of C++14 components designed with practicality and efficiency in mind. Folly contains a variety of core library components used extensively at Meta.

options. Here, Stages are modular compute and data units and purely model business logic: It is a single-threaded function that inputs and outputs Messages; it is fully decoupled from concurrency. Each Flow can run multiple Stages with varying execution options. Flows have different types to specify different execution modes such as batched vs. streamed processing vs. parallel for low-latency parallelization. Stages can run Flows enabling flexible recursion logic. These are the only key abstractions to keep the framework simple. Different types of Flows specify different execution modes, for example, a BatchedFlow does not process Messages until all Messages have been received, while a StreamingFlow processes Messages as they are received. By configuring what Flow to run what Stages, we specify the execution behavior. And a specific execution mode is a Flow class that can be reused.

A more detailed implementation of FlowInterface and FlowKernel can be seen in Figure 4.

The framework allows for an incremental adoption model so that architectural refactoring can be accomplished by refactoring code gradually to derisk the effort over time with respect to performance or reliability regressions. At the beginning, we started with only a handful of Stages and have grown to currently 80+ Stages in total. The various colors of the area chart refer to different phases of the I2DS system (see Fig. 5).

We did not employ specific external refactoring tools, but rather relied on the domain expertise of our engineers to analyze the code and design the proper abstractions.

D. The code quality framework (CQF)

The Code Quality Framework (CQF) is an internal effort to build up the logging and measurement infrastructure to compute various metrics and insights to gain an understanding of code quality and its impact on developer efficiency, reliability, and performance. The framework uses various internal infrastructure systems, platforms, and tools to provide a simplified and consolidated interface to data, code, and execution related metrics. The phabricator, for example, is one sub-component of that framework. Where possible we also leverage open source packages for some of the metrics as

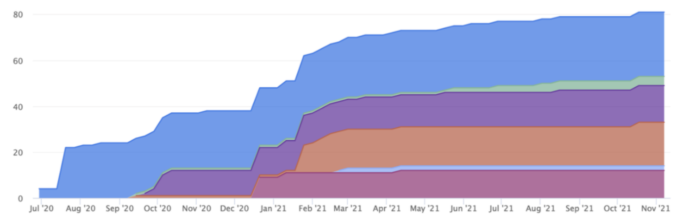


Fig. 5. An area chart of the different phases of the ItemIndexer framework and gradual introduction of associated Stages. We currently have more than 80 Stages for various business logic use cases. From top to bottom, the phases correspond to item-ranker, post-ranker, post-return, pre-ranking, stage-one-ranking, and item-retriever phases, which are internal labels for the different phases of the ItemIndexer process and system. The chart shows a high level break down of how the 80 Stages are attributed to the different phases of ItemIndexer.

described in the prior section. Through a Code Commit Cache (CCC) mechanism we are able to rapidly obtain arbitrary source code snapshots allowing us to calculate various code metrics at fine grained time resolutions for arbitrary histories. For our purposes, we obtained daily snapshots (see Figure 6). The Metric Generator is a distributed process that shards the codebase into chunks to compute the metrics in parallel across a cluster of machines. Due to internal policies, security concerns, and the scale of our infrastructure, the standard commercial frameworks do not work for us due to the scale we operate and the need to synergize alignment with our internal infrastructure.

V. Outcome assessments and results

In this section, we report our data collection process, how we analyzed the impact on developer efficiency, reliability, code quality metrics, and performance, as well as the actual results.

A. Data collection

We obtained data from various internal sources such as our internal Version Control and CI systems to get access to the I2DS codebase. We use CCC to obtain the latest commit of the day to get the latest status of the codebase for that day. Due to our monorepo design the latest commit of the master branch represents all code changes that have been made by engineers. We needed to build CCC due to the scale of our codebase and the volume of commits being performed to retrieve commits quickly for arbitrary time periods. Once we had access to the codebase, the CQF Metric Generator computed a portfolio of daily code quality metrics (see Figure 6).

For developer efficiency metrics we leveraged the internal logging data of Phabricator, which provides start and end times for the different phases of the life cycle of a diff (e.g. creation, reviewing, publishing, production). All diffs were taken into account, which includes new feature work as well as bug fixes. *I2DS* is a constantly evolving system and actively developed to improve efficiency, performance, reliability, and evolves based on internal requirements. In addition, we combined logging data on commits that engineers perform prior to creating a diff and used the time of the first commit that can be traced back to a particular diff. Each diff has a list of the files that changed, which we used to map a diff back to the *I2DS* codebase. For reliability metrics, we leveraged the internal incident management logging data, which tracks for each incidence the relevant root cause diffs. Performance metrics are readily available due to our continuous system and infrastructure monitoring tools. For the code quality metrics our analysis period matched the time period of when the refactoring started and ended, namely from 2020-05-11 till 2021-02-26. For the developer efficiency metrics, we had to choose a longer time period to understand the status quo prior to the refactoring and leave room for a sufficiently long time period after the refactoring. We chose 2021-01-01 to 2021-04-11 as our time period for analysis. For reliability metrics, our time period ranged from 4 months before and after the refactoring period.

B. Impact on developer efficiency

To compare CGT before and after the finalization of the architectural refactoring, we chose two time periods of a week in length, but 4 months apart. The first time period (in red, left box) captured CGT at the beginning of January 2021, and the second time period (in green, right box) captured CGT after the refactoring had been completed for at least a month. This is to measure the impact for the current half of the year. Note that we did not directly use the first week of January due to holidays. Rather than picking a point estimate we computed the trend and averaged the trend values (orange smooth curve) for the two mentioned comparison periods to account for noise in the metric (blue noisy curve). So, let $cgt_b = 1/n \sum_i^n x_i$ be the mean of the CGT trend before refactoring and $cgt_a = 1/n \sum_i^n x_i$ the mean of the CGT trend after refactoring. We quantify the improvement in CGT as $(cgt_a - cgt_b)/cgt_b$. We found that the refactoring effort led to a 42% improvement in developer efficiency (i.e., CGT decreased by 42%). The red vertical line indicates the end of *ItemIndexer* framework refactoring (2021-02-26) (see Figure 7 and Table I).

TABLE I
Impact of refactoring on developer efficiency measured by CGT (in hours).

Before	After	Delta	% change
298.4	172.9	-125.5	-42.1

C. Impact on reliability

For reliability, we compared two 4-month periods before and after the refactoring period (2020-05-11 to 2021-02-26) and measured the number of *ItemIndexer*-specific production issues. So let $issue_b = \sum_i^n x_i$ be the sum of the issue counts before the refactoring and $issue_a = \sum_i^n x_i$ the sum of the issue counts after the refactoring. We quantify the improvement in reliability as $(issue_a - issue_b)/issue_b$. We found a 87% improvement in the reduction of reliability issue events (see Table II).

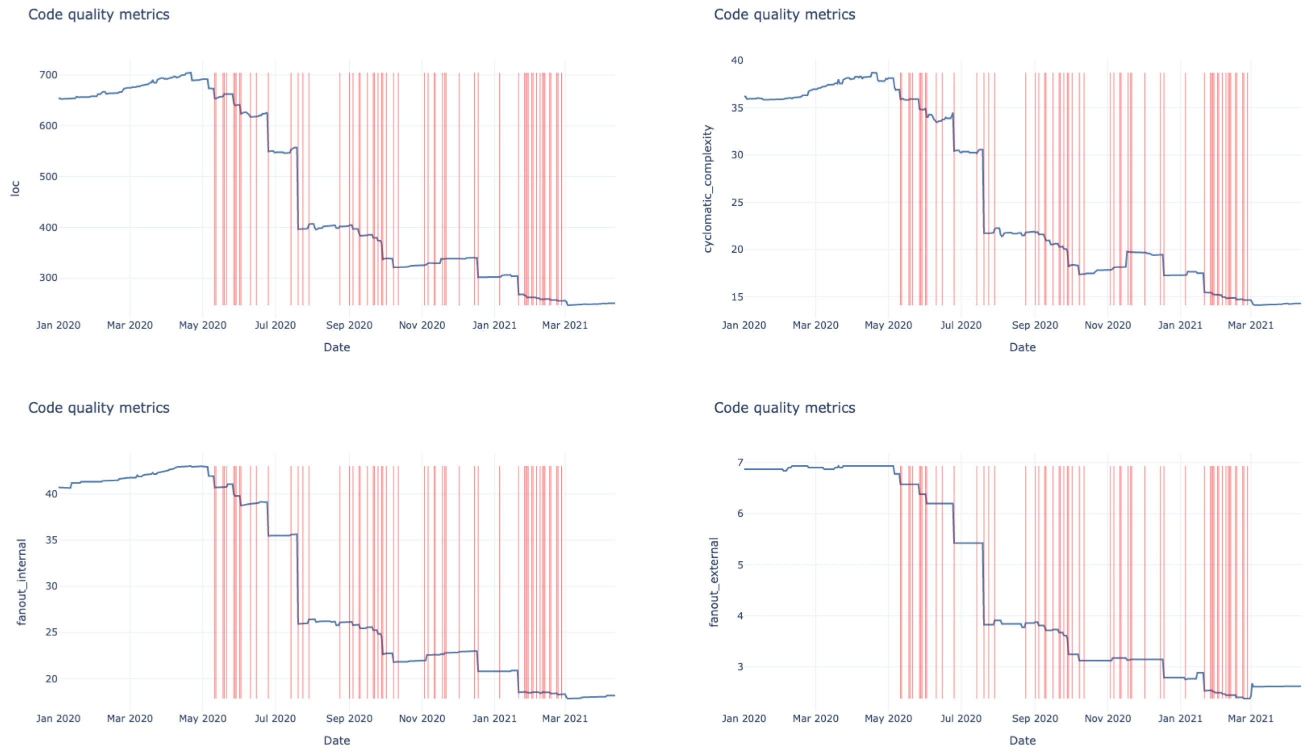


Fig. 6. A subset of internal code metrics computed in the ItemIndexer codebase over a period from 2020-01-01 till 2021-04-11. Red vertical lines indicate the date when architectural refactoring diffs were landed in production. The first refactoring diff was landed on 2020-05-11 and the last on 2021-02-26. The top-left graph shows the daily trend of the average number of lines of code (LOC) per file in the codebase. The top-right graph shows the daily trend of the average cyclomatic complexity of a file in the codebase. The bottom-left graph shows the average daily trend of fanout_internal and the bottom-right graph shows the average fanout_external of a file in the codebase. For all examples a lower average is preferred, i.e. the goal of the refactoring is to reduce the complexity (size, branching structure, and coupling) an engineer has to deal with. Header files have been excluded. One can also observe that prior to the refactoring period, the loc, cyclomatic complexity and fanout_internal metrics have been trending upward.

TABLE II
IMPACT OF REFACTORING ON RELIABILITY (ISSUE COUNTS).

Before	After	Delta	% change
49	6	-43	-87.7

TABLE III
IMPACT OF REFACTORING ON MULTIFACETED CODE QUALITY METRICS (2020-05-11 - 2021-02-26). LOC=LINES OF CODE, CCN=CYCLOMATIC COMPLEXITY, FI=FANOUT INTERNAL, FE=FANOUT EXTERNAL, CC(L28)=CODE CHURN (LAST 28 DAYS), AC(L28)=AUTHOR COUNT (LAST 28 DAYS). NEGATIVE PERCENTAGES DENOTE A DECREASE IN THE METRIC, E.G. CCN DECREASED BY 59.2%.

	LOC	CCN	FI	FE	CC(L28)	AC(L28)
change	-60.9%	-59.2%	-55.1%	-63.8%	-70.7%	-69.0%
delta	-398.4	-21.2	-22.4	-4.2	-2.5	-1.6
before	653.64	35.88	40.7	6.58	3.66	2.41
after	255.2	14.64	18.29	2.38	1.07	0.74

D. *Change in code quality metrics*

We have also computed a set of code quality metrics to provide a multifaceted view on how the refactoring effort has impacted code quality. We have chosen to highlight Cyclomatic Complexity (CCN), Fanout Internal (FI), Fanout External (FE), Code Churn (CC(L28)), and Author Count (AC(L28)). We found that the refactoring led to the following improvements. A 60% reduction in Cyclomatic Complexity, a 55% reduction in Fanout Internal, a 64% reduction in Fanout External, a 70.7% reduction in Code Churn (CC(L28)), and 69% reduction in author count (AC(L28)) (see Table III).

E. *Impact on system performance*

Lastly, we report on various performance wins as part of the refactoring project. We found a +20% increase in item scoring and a +10% increase in item matching through C++ pointer optimizations that allowed, e.g., the use of *std::move* instead of copy. This further allowed a 91% reduction in time spent for certain Heap and Iterator operations such as pop and next. Through optimized timing of object destruction calls, reusable optimization mechanisms, and automatic asynchronous destruction we found a +14% improvement in runtime reduction.

VI. DISCUSSION

This investigation demonstrates the benefits of addressing complexity growth by prioritizing code quality improvements through refactoring. Pursuing a code platformization strategy

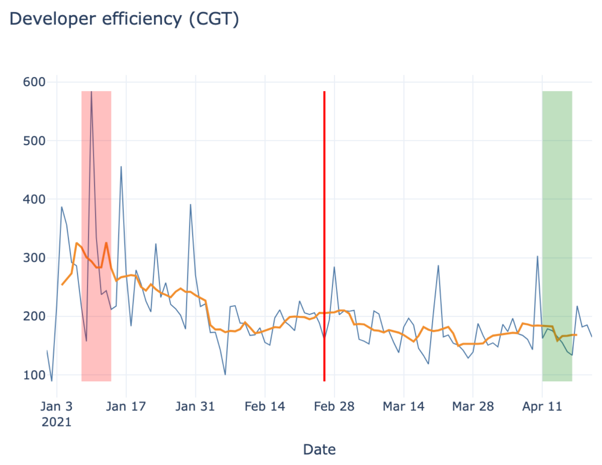


Fig. 7. Improvement in CGT for H1 2021. Red line (vertical line) indicates the end of the refactoring project (2021-02-26). The red area (left rectangle) indicates the start window to measure CGT at the beginning of 2021. The green area (right rectangle) indicates the end window to measure the CGT as of 2021-04-18 leaving enough room for the metric to evolve after the refactoring had concluded. The blue line (noisy curve) represents the raw daily CGT trends and the orange line (smoothed curve) is the extract time series trend to smooth out daily fluctuations and noise. Note that the trend is decreasing, since CGT is a measure of the cycle time for a diff. The smaller the CGT, the faster engineers can bring a change to production.

and designing and implementing the right abstractions help to make the code simpler to work with, which in turn unlocks improvements in developer efficiency, reliability, and system performance. We demonstrate that architectural refactoring of a codebase can be achieved via modular code structure, proper abstractions, and measurement frameworks.

We take a multifaceted view towards the impact of refactoring considering developer efficiency, reliability, and performance. Improvements in code quality to changes in developer efficiency have not received much attention as part of refactoring research for large-scale distributed systems. The measurement framework allows us to efficiently compute arbitrary code metrics from arbitrary time periods and resolutions obtained through a parallelized commit cache.

A systemic challenge that we would like to propound to academic research community is that many of the refactoring solutions came from seasoned experts and manual analysis of the codebase. Having semi- or automatic ways to assist this process would be of high interest, yet usable solutions or methods to easily apply in an industry setting are scarce.

A. Lessons learned

The refactoring of our *ID2D* codebase was a challenging and risky undertaking. Many months of preparation, design, and alignment discussions were performed to develop our strategy and approach. In this section, we would like to share some of the lessons learned.

L1: Large-scale distributed system — Embarking on an architectural refactoring of a large-scale distributed system that is live $24/7$ is a big undertaking with many risks involved. Especially for complex parallel code constructs, changing is risky. Not to mention that refactoring had to be performed gradually where engineers needed to cope with an intermediate mixed-state of legacy and new refactoring code. Careful, parallel test paradigms and supporting the notion of a partial API to gradually role out the refactoring helped in managing this risk.

L2: View code as a platform — The notion of code platformization helps to identify and incentivize the right abstractions, which enabled us to create modular simplified building blocks, clear separation of concern, better ownership attribution, and to transform repetitive and risky performance optimization into reusable generic optimization principles that consumers can easily leverage and build upon.

L3: Core abstractions — Focus on the engineering quality of the core abstractions, making it minimal and simple to use and adopt.

L4: Performance regression — Minimization of performance regressions was on the one hand important and, on the other hand, rather challenging due to the user-facing nature of the system, which required careful implementation of abstractions through template meta-programming and close monitoring of performance with constant performance mining and tuning.

L5: Early feedback — Early involvement of the framework's consumers and iteration of their feedback was essential to fail fast, choose the right trade-offs, and optimize design and architecture. Learning from our users what worked and what did not helped us further prioritize to align immediate needs with longer-term refactoring milestones.

L6: Tooling and measurement — The complexity of the initiative and time horizon required for every phase the availability of multifaceted measurement signals to guide the effort, validate changes, measure effectiveness, and ultimately the outcomes, which did not exists. A lot of the tooling and measurement had to be build in parallel and was instrumental to guide, inform, and justify the effort. Define metrics to track the abstraction onboarding progress and quality, and at the same time, keep retrospecting its impact on developer efficiency.

L7: Incentives — Architectural refactoring alone is only half the story, and investing, motivating, incentivizing code quality improvement initiatives requires a tightly coupled multifaceted measurement framework to validate, proactively identify and quantitatively measure the impact of refactoring opportunities. As has been reported in prior literature motivating engineers to embark on refactoring projects is a challenge, which requires additional investments to build out the code quality framework to make impact identification, prioritization, and measurement easier for engineers.

VII. Threats to Validity

Internal validity — The metric measurements are derived from our internal tools, which are used daily by thousands of engineers. No sampling was applied. To get representative data we utilized various time periods to perform before and after comparisons. The before and after periods contained diffs that represented the typical engineering workload to evolve *I2DS*. A slight difference is that the improvement in reliability reduced the number of bug fix work. The developer efficiency measures were computed on the whole engineering population that consumed the *ItemIndexer* framework. For more robust estimates, we compared trends for two point estimates based on a 7 week average to account for noise. Running an A/B test was neither feasible nor practical due to cost and business constraints.

External validity — *I2DS* is an internal system with a unique architecture and scale. The results are only valid for the investigated system and may not be valid for other systems, yet we hope that the challenges and lessons learned can provide value to other practitioners and researchers.

VIII. Conclusion

In this paper we shared our experience on a code maintenance project that involved one of our *ItemIndexer* delivery systems (*I2DS*), a service that delivers millions of items to billions of users at low latency. We found that investing in code maintenance can have a positive impact on developer efficiency, reliability, and system performance, as well as improvements for various code quality metrics.

Future work is focused on scaling out the code quality framework to other areas of Meta's codebase. This will involve building out multi-language support for our code quality metrics, enriching our metric portfolio, creating features and code quality specific labels to drive code maintenance automation initiatives and tooling development.

References

[1] Chaima Abid, Vahid Alizadeh, Marouane Kessentini, Thiago do Nascimento Ferreira, and Danny Dig. 30 years of software refactoring research: A systematic literature review. *ArXiv*, abs/2007.02194, 2020.

[2] Nicolas Brousse. The issue of monorepo and polyrepo in large enterprises. In *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*, pages 1–4, 2019.

[3] Aloisio Cairo, Glauco Carneiro, and Miguel Monteiro. The impact of code smells on software bugs: A systematic literature review. *Information*, 9:273, 11 2018.

[4] L. Cruz and Rui Abreu. Using automatic refactoring to improve energy efficiency of Android apps. *ArXiv*, abs/1803.05889, 2018.

[5] Dario Di Nucci, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Andrea De Lucia. Detecting code smells using machine learning techniques: Are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 612–621, 2018.

[6] Christof Ebert, James Cain, Giuliano Antoniol, Steve Counsell, and Phillip Laplante. Cyclomatic complexity. *IEEE software*, 33(6):27–29, 2016.

[7] Nicole Forsgren, Margaret-Anne Storey, Chandra Maddila, Thomas Zimmermann, Brian Houck, and Jenna Butler. The space of developer productivity: There's more to it than you think. *Queue*, 19:20–48, 02 2021.

[8] John Gall. *General Systemantics: An Essay on how Systems Work*. General Systemantics Press, 1975.

[9] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. One thousand and one stories: A large-scale survey of software refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1303–1313, New York, NY, USA, 2021. Association for Computing Machinery.

[10] Daniel Graziotin, Fabian Fagerholm, Xiaofeng Wang, and Pekka Abrahamsson. What happens when software developers are (un)happy. *Journal of Systems and Software*, 140:32–47, 2018.

[11] James Ivers, Robert L. Nord, Ipek Ozkaya, Chris Seifried, Christopher Steven Timperley, and Marouane Kessentini. Industry experiences with large-scale refactoring. *CoRR*, abs/2202.00173, 2022.

[12] Satnam Kaur, Lalit Awasthi, and Amrit Sangal. A review on software refactoring opportunity identification and sequencing in object-oriented software. *Recent Advances in Electrical Electronic Engineering (Formerly Recent Patents on Electrical Electronic Engineering)*, 13, 07 2020.

[13] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. An empirical study of refactoring challenges and benefits at Microsoft. *IEEE Transactions on Software Engineering*, 40(7), July 2014.

[14] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. Predictive test selection. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 91–100, 2019.

[15] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[16] Mohamed Wiem Mkaouer, Marouane Kessentini, Slim Bechikh, Kalyanmoy Deb, and Mel Ó Cinnéide. Recommendation system for software refactoring using innovization and interactive dynamic optimization. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 331–336, New York, NY, USA, 2014. Association for Computing Machinery.

[17] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In Bertrand Meyer, Jerzy R. Nawrocki, and Bartosz Walter, editors, *Balancing Agility and Formalism in Software Engineering*, pages 252–266, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[18] Edward Ogheneovo. On the relationship between software complexity and maintenance costs. *Journal of Computer and Communications*, 02:1–16, 01 2014.

[19] Rachel Potvin and Josh Levenberg. Why Google stores billions of lines of code in a single repository. *Communications of the ACM*, 59(7):78–87, 2016.

[20] Ganesh Samarthyam, Girish Suryanarayana, and Tushar Sharma. Refactoring for software architecture smells. In *Proceedings of the 1st International Workshop on Software Refactoring*, IWoR 2016, page 1–4, New York, NY, USA, 2016. Association for Computing Machinery.

[21] Darius Sas, Paris Avgeriou, and Umut Uyumaz. On the evolution and impact of architectural smells – an industrial case study. *Empirical Software Engineering*, 27(86), 2022.

[22] Syed Muhammad Ali Shah, Efi Papatheocharous, and Jaana Nyfjord. Measuring productivity in agile software development process: A scoping study. In *Proceedings of the 2015 International Conference on Software and System Process*, ICSSP 2015, page 102–106, New York, NY, USA, 2015. Association for Computing Machinery.

[23] Gábor Szőke, Csaba Nagy, Péter Hegedűs, Rudolf Ferenc, and Tibor Gyimóthy. Do automatic refactorings improve maintainability? an industrial case study. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 429–438, 2015.

[24] Unknown. ISO/IEC 25010:2011. https://www.iso.org/standard/35733.html, 2011. [Online; accessed 11-Nov-2021].

[25] Roberto Verdecchia, Philippe Kruchten, Patricia Lago, and Ivano Malavolta. Building and evaluating a theory of architectural technical debt in software-intensive systems. *Journal of Systems and Software*, 176:110925, 2021.

[26] Michael Wahler, Uwe Drofenik, and Will Snipes. Improving code maintainability: A case study on the impact of refactoring. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 493–501, 2016.

[27] Kaitlynn M. Whitney and Charles B. Daniels. The root cause of failure in complex IT projects: Complexity itself. *Procedia Computer Science*, 20:325–330, 2013. Complex Adaptive Systems.

[28] Jifeng Xuan, Benoit Cornu, Matias Martinez, Benoit Baudry, Lionel Seinturier, and Martin Monperrus. B-Refactoring: Automatic Test Code Refactoring to Improve Dynamic Analysis. *Information and Software Technology*, 76:65–80, 2016.

[29] Minhaz F. Zibran and Chanchal K. Roy. Conflict-aware optimal scheduling of code clone refactoring: A constraint programming approach. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 266–269, 2011.