Dead Code Removal at Meta: Automatically Deleting Millions of Lines of Code and Petabytes of Deprecated Data

Will Shackleton
Katriel Cohn-Gordon
Peter C. Rigby*
Rui Abreu
wshackleton@meta.com
katriel@meta.com
pcr@meta.com
ruiabreu@meta.com
Meta Platforms, Inc.
Menlo Park, CA, USA

James Gill
Nachiappan Nagappan
Karim Nakad
Ioannis Papagiannis
jagill@meta.com
nnachi@meta.com
knakad@meta.com
yiannis@meta.com
Meta Platforms, Inc.
Menlo Park, CA, USA

Luke Petre†
Giorgi Megreli
Patrick Riggs
James Saindon
lpetre@meta.com
gmeg@meta.com
riggspc@meta.com
jsaindon@meta.com
Meta Platforms, Inc.
Menlo Park, CA, USA

ABSTRACT

Software constantly evolves in response to user needs: new features are built, deployed, mature and grow old, and eventually their usage drops enough to merit switching them off. In any large codebase, this feature lifecycle can naturally lead to retaining unnecessary code and data. Removing these respects users' privacy expectations, as well as helping engineers to work efficiently. In prior software engineering research, we have found little evidence of code deprecation or dead-code removal at industrial scale. We describe Systematic Code and Asset Removal Framework (SCARF), a product deprecation system to assist engineers working in large codebases. SCARF identifies unused code and data assets and safely removes them. It operates fully automatically, including committing code and dropping database tables. It also gathers developer input where it cannot take automated actions, leading to further removals. Dead code removal increases the quality and consistency of large codebases, aids with knowledge management and improves reliability. SCARF has had an important impact at Meta. In the last year alone, it has removed petabytes of data across 12.8 million distinct assets, and deleted over 104 million lines of code.

KEYWORDS

Code transformation, Automated refactoring, Data purging, Data cleanup

ACM Reference Format:
Will Shackleton, Katriel Cohn-Gordon, Peter C. Rigby, Rui Abreu, James Gill, Nachiappan Nagappan, Karim Nakad, Ioannis Papagiannis, Luke Petre, Giorgi Megreli, Patrick Riggs, and James Saindon. 2023. Dead Code Removal at Meta: Automatically Deleting Millions of Lines of Code and Petabytes of Deprecated Data. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Rigby is also a professor at Concordia University in Montreal, QC, Canada.
†This work was done while Petre was at Meta

1 INTRODUCTION

Software rapidly evolves to meet users' changing needs. As it does, some features become unnecessary, and the associated *code* and *data* need to be removed. In this paper, we introduce SCARF, a system which safely removes both dead code and data at scale.

Users expect organisations to only store their data when there is a clear need and purpose, and achieving this goal is necessary for every product that respects users' privacy expectations. One important aspect of this expectation is to prevent storing data for which no purpose exists at all. At first, storing unused data for which there is no clear need or purpose seems counter-intuitive and unlikely to happen in practice. We observe however that in the modern product development lifecycle new product features are constantly prototyped and assessed, collecting and persisting data in the process. For example, when prototyped features are sunset, any associated data collected during their operation should be proactively deleted. If not, data would reside in product databases for no particular purpose and remain unused in perpetuity.

Dead code is a common problem for all software engineers: almost all have personally experienced a class or method that is bloated with code that appears to be unnecessary. Removing this code is often non-trivial, and manually deleting it can lead to missed dependencies. For example, while linters [13] can catch simple cases of unreachable code such as unused private methods or code after a return statement, they can easily miss more complex or dynamic usage such as reflection or disabled feature flags [20]. As a concrete example, it is unclear whether a static initializer in C++ has side-effects, and whether those side-effects have a meaningful business impact (such as, registering an HTTP endpoint in a web request framework, even if that HTTP endpoint is not actually used). This in turn means that it is not easy for an engineer to decide whether it is safe to delete a seemingly-unused static initializer. In a survey of Meta developers, we found that of engineers who said, "I find it challenging to work in the codebase", 30% of them cited dead code as the reason.

More generally, manual removal of code or data by engineers does not necessarily follow standard processes to avoid breakages. For example, codebase maintainers might wish to enforce that data is *quarantined* before being deleted, for instance by applying a strict Access Control List (ACL) to prevent reads and writes. This allows any production issues to be detected while it is still easy to revert the quarantine, instead of having to rely on restoration from backups. Similarly, manually removing data from a large and complex graph may not be an instantaneous operation: database scans that iterate over millions or billions of rows can take significant time to complete (not including the quarantine time), and the engineer running the removal needs to remember to monitor the scan.

1.1 Overview and Example

Before we discuss the technical details, we provide an overview of SCARF. We have broken down the automated deletion process into three stages shown in Figure 1: Data Collection, Processing, and Deprecation. We describe each stage using a simple running example: a MariaDB [16] table which is used in several different codebases and specified in a data schema.

Collect Data. We identify dead code and data using two main data sources: source dependencies (*e.g.,* method calls) and runtime usage (*e.g.,* how much traffic a method handles). We create graphs based on these dependencies and connect code and data assets into subgraphs. For the example of our MariaDB table, we would collect the following information: a set of all tables which currently exist, metadata about how many reads and writes each table receives in production, and then pointers to the code which queries the table, as well as the location of the data schema defining the table.

Process. The data collected is passed through data quality checks and converted to a unified format. This standardisation is important since we require several systems to understand each of the heterogeneous data sources which comprise SCARF's data collection. Once standardised, the data are exported to a graph database for consumption by both SCARF and a suite of other internal tooling. From there, SCARF leverages this graph to select candidates for automated deprecation. It also presents insights into the subgraphs of this graph to engineers, to allow them to inspect a given subgraph and understand both its runtime usage as well as how to isolate it from the rest of the graph to unblock deprecation. For the example of our MariaDB table, our collected data can be analysed to locate tables which have no usages in code, no defining data schemas, and no production reads or writes.

Deprecate. Once identified as safe-for-deletion, some assets require approval for deprecation (*e.g.,* a database table), while others (*e.g.,* an unused method in code) can have a patch (or a diff at Meta) automatically generated and sent for code review. This occurs with a four step process: SCARF can alert the asset owner that deprecation is proceeding, obtain approval from that owner if necessary, then quarantine the asset (make it inaccessible), and finally enact the deprecation. For the example of our MariaDB table, SCARF can start by filing an internal ticket for the table's owner. If the table has no rows, it may skip obtaining approval for the deprecation. Next, SCARF can apply an ACL on the table to disallow all non-infrastructure reads and writes, and then finally it can issue a DROP TABLE command against the table.

1.2 Structure

This paper has the following structure. In Section 2, we provide background on Meta including the software development process and tools. In Section 3, we discuss the data we collect and the graphs that we create to facilitate dead code and data removal. In Section 4, we discuss the data processing stages which includes data cleaning and identifying subgraphs that can be safely removed. In Section 5, we present our approach to safe deprecation which includes quarantining data before final removal. In Section 7, we describe the scale and impact of dead code and data deletion at Meta. In Section 8, we position our work in the context of the literature. In Section 9, we describe our contributions and conclude the paper.

2 BACKGROUND

Meta is a large online services company that works across the spectrum in the communications, retail, and entertainment industries. Meta has tens of thousands of employees with offices spread across the globe (North America, Europe, Middle East, and Asia). Meta has its own dedicated infrastructure teams where the tools used are a mix of commercial and in-house developed systems. Like any large scale system, outdated code and data must be removed to avoid technical debt or unsupported features.

2.1 Software Development Process at Meta

A good resource for understanding software development process at Meta is provided by Feitelson, Frachtenberg, and Beck [12]. Here, we elaborate on the key aspects that have either undergone some changes or need more detail to better understand the context of our study..

Meta, like other Internet companies, builds software for their own servers as well as client software deployed on mobile devices or specialized hardware such as Virtual Reality headsets. This enables rapid updates to the software and allows fine-grained control over versions and configurations. At Meta, this deployment has led to a practice of regular "push"es of new code to production. Before being pushed, code is subject to peer review, internal user testing, and extensive automated testing. After being pushed, engineers monitor logs to identify potential issues.

As with Open Source Software (OSS), Meta developers are also users, and have first-hand knowledge of what the system does and what services it provides. Engineers continuously develop new features and make them available to users because of the need to constantly evolve to satisfy not only changing user needs but also a developing product landscape. As in many other Internet companies, Meta's new code is deployed as a series of small changes as soon as they are ucc ready. Since most of the functionality is on the server side, deploying new software to the servers immediately makes it available to all users, without any need for downloads and local installation. The ability to deploy code quickly in small increments behind feature toggles enables rapid innovation [19].

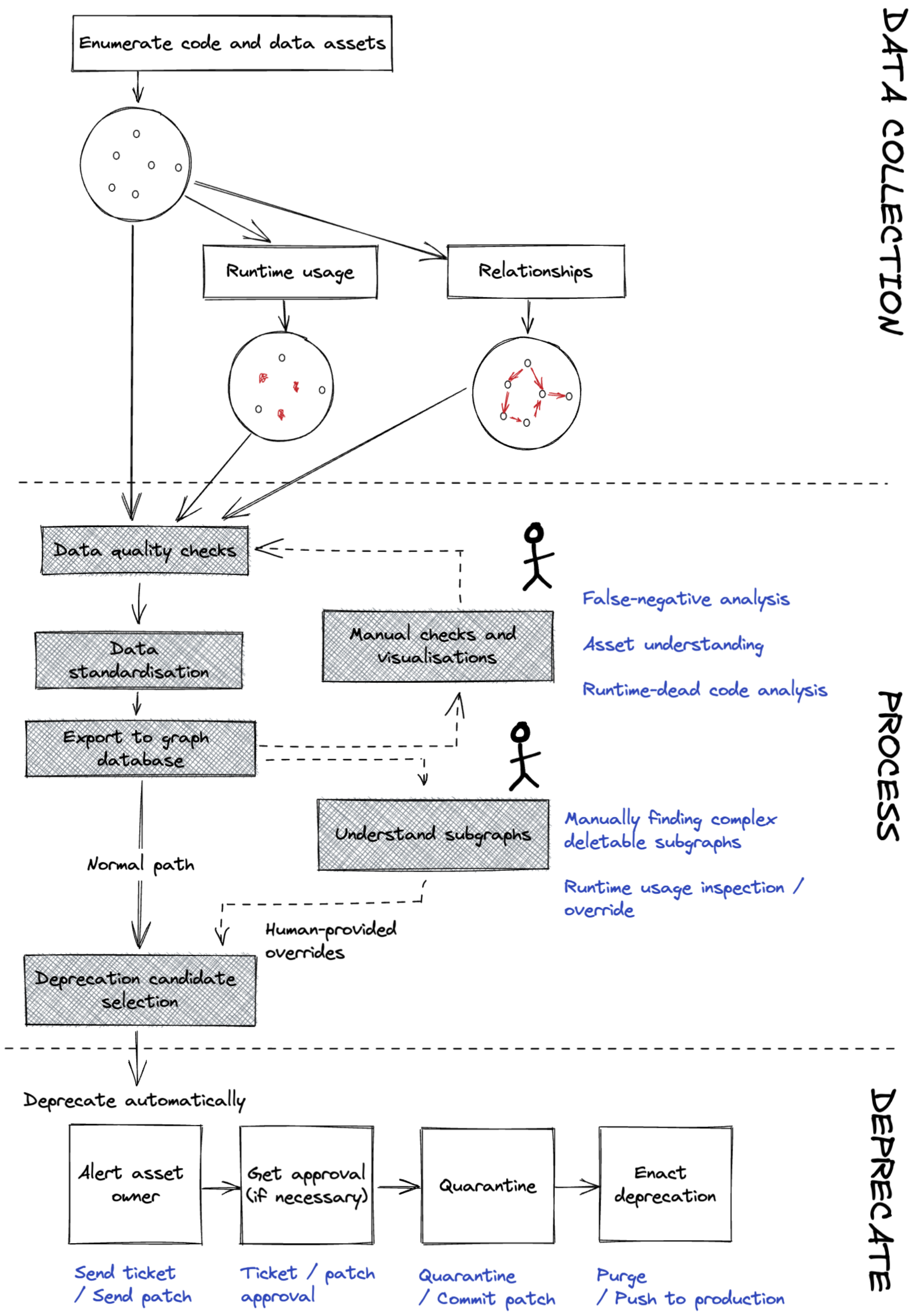

Figure 1: Architecture of SCARF, which shows its three stages. The data collection stage (Section 3) gathers metadata about code and data assets at Meta; the processing stage (Section 4) analyses, standardises and exports that data to other systems, and also determines candidates for automated deprecation; and the deprecation stage (Section 5) handles the safe, automated removal of assets. Components highlighted in gray are shared between instances of SCARF; components in white are implemented on a per-instance basis.

At Meta, code is generally reviewed by other engineers. This serves multiple purposes. First, the original engineer is motivated to ensure that the code is of high quality. Second, the reviewer comes with a fresh mind and might find defects or suggest alternatives. Third, knowledge about coding practices and the code itself spreads throughout the company. Phabricator [18] is the backbone of Meta's Continuous Integration (CI) system and is used for modern code review, through which developers submit changes (diffs) and comment on each others' diffs, before they ultimately become accepted into the code base or are discarded.

Phabricator and the version control systems are also used to measure the development process where both events associated with the code change and developer and reviewer actions and the current state of diffs are recorded. When developers submit their code for review, they make a commit (create a version of the code, also known as *patch*) representing the initial version of the so-called diff. If reviewers notice any issues or suggest improvements, they may make additional revisions until the diff is either approved and incorporated into the code base ("landed"), or abandoned. The diff is a collection of patches representing the initial version of a code change, together with revisions that were needed to make it "land."

2.2 Tooling

Static analysis tools are widely deployed at Meta. To identify source code dependencies and find unused code, we use Glean[1] for C/C++ code and for Hack. Another relevant tool for the approach presented in this paper is Apache Hive [25]. Hive is an open-source data warehousing and analytics solution built on top of the Hadoop Distributed File System. It was developed by Meta and released as an open-source project in 2008. Hive allows users to query and manage large datasets residing in distributed storage by using HiveQL, a SQL-like language. It provides a way to project structure onto the data and query it using SQL-like syntax, making it easier for users familiar with relational databases to analyse big data.

2.3 Sources of Deprecation Candidates

We observe three major sources of unused data likely to occur in most products and applications.

First, *engineers need to perform thorough cleanup of a feature* after turning it off. For example, Ramanathan, Clapp, Barik, and Sridharan [21] describe "feature flags" which determine which users are able to see particular product features. Disabling the feature flags of a deprecated feature is not sufficient to ensure the eventual deletion of underlying data schemas and associated data.

Second, *engineers identify relevant scope of deletion.* Often, dead code cleanup is performed after conducting experiments, or when a new feature has replaced old features. Since the deprecated feature's code may be interlinked with still-live parts of the product, engineers must identify which specific sections of code are deprecated, potentially down to the function- or method-level within a class.

Third, *engineers need to delete it safely and completely.* For example, one common definition of "unused" may be "database logs indicate no queries to the table." However, it is common for large databases to have periodic scans over all data, for example from Data Loss Prevention (DLP) software or operational metrics. Another potential definition of "unused" is "no code references to the database exist", but large, interconnected systems often do not provide strong boundaries between the code and data for one feature and another within a larger product. A large feature can reach many different storage systems, and manually finding all the individual pieces of associated data can be very time consuming. In practice, even when engineers attempt to manually delete any stored data that is no longer necessary, the process is technically challenging.

[1]Glean is an open-source system for working with facts about source code. Available at https://glean.software/

3 DATA COLLECTION STAGE

The major components of the system are described in Figure 1. We have deployed SCARF widely at Meta across many teams, code repositories and data systems. There are over 50 instances across two main implementations: automated removal of *code* and of *data.*

We discuss the data collection stage in this section. SCARF gathers information about code and data assets at Meta. These assets can be a symbol of code, such as a class or a method, or a logical type of data in a database, such as a table. The list of assets is stored in a key-value store, and for each one we gather additional information:

(1) *Basic data:* the asset such as size or creation time
(2) *Runtime usage:* how often is this asset invoked, read from, or written to
(3) *Relationships*: what else refers to this asset?

3.1 Collecting Information about Code

We start by describing the information that SCARF gathers about code assets.

3.1.1 Basic data. SCARF gathers metadata directly about each asset. At minimum, this consists of a unique name or identifier, and the asset class and location (e.g. function in repository X or table in database Y). Where possible, it also gathers the size of the asset (e.g. lines of code or number of rows), creation metadata such as original author and timestamp, and data classification where relevant.

We use existing data catalog tools at Meta to collect this information about code and data assets. Within that catalog is a set of domain-specific implementations for each codebase or data system.

3.1.2 Runtime usage. SCARF gathers information about how each asset is used in production, mainly by instrumenting production frameworks and systems with logging that records all usage. For example, the runtime usage of a data table consists of its read and write traffic volume: how many bytes per second are being added or updated to the table, and how many are being queried by other systems. Runtime usage of code consists of usage logging: for instance, in an Model-View-Controller (MVC) framework the usage might be the Queries Per Second (QPS) being served by each controller, or in an Object Relational Mapping (ORM) it might be the frequency which with its load/store methods are called.

To implement this instrumentation for Meta's codebases, we integrate SCARF-specific logic into a variety of systems, scanning access logs or counters for individual frameworks within the codebase. For example, our primary web codebase contains an MVC request handling framework to which we added access logging to record how often each controller is loaded. Similarly, we wrote extensions for MySQL to record usage metrics by incrementing a per-table counter each time a client queries it.

This granularity of logging can present problems with some of Meta's largest data systems such as TAO [4]: their read volume can be too high to count every hit without causing performance regressions. To solve this, SCARF leverages *dynamic sampling*: once we identify a read that we consider as valid runtime usage, we pause logging for a certain amount of time. We then aggregate counters in Operational Data Store (ODS), a high-performance global counter service [5, 24]. This provides good system performance and ensures that we can detect even one single valid read to each data asset. Because it throttles expensive logging it also enables us to run more computationally-expensive classification checks at logging time: we can filter out system infrastructure traffic and log only "true" application traffic.

3.1.3 Relationships. SCARF gathers a directed graph representing the dependencies between assets. Each node in the graph is an asset and each edge a dependency. The edges have one of three semantic flags: a deprecation blocker, a transitive deprecation, or no-op.

A deprecation-blocker edge indicates that the destination node cannot be deprecated until the source node is deprecated, or until the edge is removed. For example, if a class `Foo` contains a function call to `bar()`, we cannot remove the function definition without first changing the callsite in the class. To reflect this dependency, we create a deprecation-blocking edge from `Foo` → `bar()`. More complex examples include class inheritance (`Foo extends Bar`), type aliases (`type Foo = Bar`) or build artifacts (`Foo` generates `class Bar` at build time); all of which create edges `Foo` → `Bar`, or in some cases `Foo` ↔ `Bar`. Most edges fall into this category.

A transitive deprecation edge indicates that if the source node is deprecated, the destination node must also be deprecated. Transitive deprecation edges also block deprecation. For example, we create a transitive deprecation edge from a data schema to a data asset if the latter is the schema for the former, since we cannot deprecate a data asset until its schema is removed from code. This encodes the fact that once we deprecate the schema we should continue and deprecate the underlying data asset.

If an edge does not have either of these flags we consider it a *no-op* edge. We include these edges because they can be useful to explain the source of data in a data asset, even if that source does not block its deprecation. For example, a code reference of the form `if foo instanceof Bar` is a no-op edge, since we can automatically remove that reference by replacing the instance check with a constant `false`, if we know that there are no other usages of the class `Bar`.

For nodes representing code, these edges primarily come from static analysis of source code. For nodes representing data, these edges come from analysing data infrastructure dependencies, as well as scanning code for mentions of the name of a given type of data. As with the other types of metadata collected by SCARF, the implementation of relationship gathering is domain-specific to each supported use-case.

3.2 Collecting Code Relationships

We use two main technologies to collect code relationships at Meta: Glean [10] for syntactic references, and BigGrep [26] for others.

Glean is a system for working with facts about source code. It efficiently stores facts about the code structure (such as "function `foo` is called in the following eight files") and answers queries such as "what are all the call sites of `bar`".

In practice, Glean's large indexing jobs can take multiple hours, and produce a relatively large and unwieldy database. In order to handle large repositories, Glean also supports incremental indexing [17]: instead of regenerating the index on every source control change, it periodically performs a full rebuild of the index, then updates it to take into account recent code commits.

3.2.1 Complex runtime usage. We found that, in some cases, relying on Glean's static information to provide sufficient coverage across the data system or language is difficult. That is, it is often challenging to find *all* sources of runtime usage or relationships. Complex examples include:

- bulk internal traffic such as backup or indexing services, which create traffic to many data assets;
- unclear external traffic such as web crawlers, which may load large numbers of endpoints;
- code using string concatenation to construct table names;
- reflection used to call methods on classes without static references to function names;
- language support for automated factory registration (i.e., enumerating all subclasses of a given class);
- compilation artefacts which reference classes but are not committed to a repository;
- table names stored in a database instead of in code.

Our implementations of SCARF must be resilient to such usage: SCARF should not cause errors in production by deprecating assets with such references. Where feasible, we design our code and data systems to prevent non-recommended usage. For example, we could mandate that references to TAO assets should be through an ORM, and thus increase our confidence that there are no dynamic references to such assets. Similarly, we can offer a type-safe API to construct the Uniform Resource Identifier (URI) for a given controller, and surface lint warnings to developers where the API is not used.

Where this approach is insufficient we directly codify complex usage patterns into our collection infrastructure. We implement this as a decision engine which detects a number of different patterns. For example, if a certain class hierarchy is known to use reflection, we can explicitly detect that usage and add deprecation-blocking edges pointing towards its subclasses in order to reflect the dynamic dependencies. There are over three hundred of these pattern detectors in the largest instance of SCARF as of this writing.

We have found the combination of these two approaches (discouraging non-recommended usage, but allowing business logic to understand the reality of a large codebase or data system) to be sufficient to maintain strong production safety guarantees in practice.

3.2.2 BigGrep. We implemented a specific detector based on textual references: we perform a code search through all repositories for all symbols and data assets that may be deprecated. To do this, we use BigGrep, Meta's large-scale code search tool.

We remark on some interesting subtleties while integrating SCARF with BigGrep. By default, BigGrep returns three possible types of value: (a) no matches found, (b) a complete list of matches, or (c) a truncated list of matches to prevent system overload for very generic queries. A naïve implementation of text reference search which searched for the name of all potential assets turned out to be computationally expensive for the indexing service. We therefore construct a prefix tree of asset names and submit queries in depth-first order. If BigGrep truncates a query for an intermediate node, we query for each of its children in order to reconstruct the full set of references to each node in the prefix tree.

In some cases, we know more structured information. For example, certain types of data asset can only be referenced in code in a form like `Schema.typeID=[0-9]+`. In these cases we can submit a single regex query to BigGrep to return all such results. Optimisations such as these enabled us to reduce the QPS of SCARF to BigGrep to under 1.

While using a text reference search as well as information from the language itself can prevent incorrect deprecations, it also increases the rate of incorrect edges i.e., reported dependencies between parts of the code base which are not in fact true dependencies. While this can reduce the efficacy of automated deprecation, we found that the overall benefits in production safety outweighed any costs. Furthermore, some of the incorrect references can be mitigated using business-specific heuristics, such as to ignore references from certain files. For example, while deploying this subsystem we found an example of a file containing a list of particularly frequently-invoked functions. We suppressed any references from this file.

4 PROCESSING STAGE

The second stage in Figure 1 is the processing stage. In this stage, we write data quality checks, standardise the data, and select potential candidates for deprecation. Data processing at Meta is orchestrated using existing data warehouse infrastructure.

Most of the processing is conducted in daily batch jobs running in Dataswarm [23], a data workflow automation and scheduling platform, and data stored in Apache Hive [25] tables. Dataswarm and Hive are designed for large data workloads, and thus suitable for the scale of analysis required given the size and complexity of codebases and data systems at Meta. Each day jobs are scheduled for the various components of each implementation of SCARF: Dataswarm maintains the graph of dependencies between these jobs. Failures are highlighted to on-call engineering teams. SCARF is designed such that a failure in a single implementation of the system does not block the progression of other parts of the system.

SCARF then performs data quality checks and standardises its datasets, before exporting data to two systems: Glean [10] for code implementations of SCARF, and RocksDB [8, 9] for data implementations. Both of these systems permit the efficient query operations required by SCARF.

4.1 Implementation of Processing Stage

In large, this stage of SCARF looks like many other data processing pipelines: it leverages idempotency, retries, health monitoring and common data analysis techniques to accomplish its goal.

4.1.1 *Data quality checks.*

The data gathered in the data collection stage needs to be checked for quality, consistency and regressions. We run both generic and domain-specific checks on the data from all instances of SCARF. For example, a generic check ensures that the size of each dataset is greater than zero, and a domain-specific check ensures that collected MariaDB table names do not contain forbidden characters. SCARF also performs regression testing, comparing each dataset to the previous day's data to measure the day-by-day change. Differences above a certain threshold require engineer approval to ensure that they are not caused by errors or system failures but instead are deliberate changes (for example, when making a significant modification to the system).

4.1.2 *Data standardisation.*

We then standardise the data into a uniform format which is the same across all instances of SCARF. This has several benefits. First, we can create asset identifiers which are unique across all SCARF instances, and hence avoid confusion between similarly-named assets of different types. This enables analysis of all SCARF data together, rather than of each individual implementation. Second, edges in the graph can be connected across implementations of SCARF. For example, if we have a reference to a MariaDB table in some Python code, then the MariaDB relationship collector may only be able to resolve that to a file name and line number. Inter-implementation standardisation allows further resolution of that edge to the exact symbol within the Python codebase, without the MariaDB relationship collector having to understand Python code specifically. Third, we store the data in a graph database to allow low-latency, high-query-load access to the data. This graph database is queried by SCARF itself in its deprecation candidate selection process. It is also queried by the subgraph understanding tooling, as well as a suite of other internal tooling at Meta.

4.2 SCARF data consumers

We then export the results of our analysis to three main consumers at Meta: deprecation candidate selection, subgraph understanding, and manual checks and verification.

4.2.1 *Deprecation candidate selection.*

We build a directed dependency graph with nodes representing individual instances of code or data assets and edges represent static and runtime dependencies. Each node is also annotated with its basic data and runtime usage. Once we have built this graph, we pass over the graph to determine which subgraphs are candidates for deprecation. Candidate subgraphs must:

(1) have no inbound deprecation-blocking edges, for example, a function cannot be removed if its return value is used in production code,
(2) have no runtime usage, for example, a table should not be deprecated if it is being actively queried by a downstream system, and
(3) be of the same type, for example, SCARF cannot atomically deprecate a graph consisting of some functions and some data tables in one pass.

These candidate subgraphs are exported to the automated deprecation pipeline described in Section 5. The subgraphs can also be analysed by engineers, using our internal tooling suite, to understand if complicated subgraphs can be deleted.

During deprecation candidate selection, we avoid selecting the full set of all assets for deprecation for two reasons: rate-limiting and production safety. We limit the speed of automated deprecation in order to balance progress through the full set of potential targets with available developer capacity. This also ensures that, even if SCARF incorrectly marks assets as deprecated, it will do so at a limited speed and thus give more time to detect and remediate the issue.

4.2.2 Subgraph understanding. SCARF can export non-candidate subgraphs for human analysis: if a developer believes that a certain subgraph *should* be a deprecation candidate, they can examine SCARF's decisions in an internal tool and take actions to adjust the subgraph or override its logic. This internal tool presents data about a subgraph with visualisations, aggregations and human-readable understanding of the data. This gives the engineer insight into how the subgraph is being used: they can reason about its combined runtime usage, as well as how the subgraph is connected to the rest of the overall graph. The engineer can isolate the subgraph from its external dependencies in the larger graph, and then understand the runtime usage of the subgraph's nodes. To remove the entire subgraph from production, the engineer needs to bring the runtime usage down to zero.

The internal tool for subgraph understanding imports its own copy of the dataset into its own database, for each subgraph that an engineer is inspecting. This allows for an additional layer of classification and visualisation not immediately offered by SCARF's datasets on their own: namely, the ability to clearly visualise the boundary of that subgraph, and to adjust it interactively. To do so, engineers classify edges which cross this boundary, to say if the boundary should be adjusted, and if so, how. This is an iterative process, and this internal tool continues to import further data from SCARF as they go. Once the boundary is set, the tool aggregates the metadata from SCARF and provides a high-level sequence to the steps the engineer should take to perform the deprecation. This ordering is influenced both by the ordering of the imported dependency graph, as well as with custom business rules. For example, the tool will recommend that mobile code is removed before server-side code, as this is a good rule-of-thumb for retaining stability of mobile apps.

The engineer will typically perform this work by committing code and configuration changes which are then subsequently picked up by the data collection process. Sometimes the runtime usage of a node may not drop to zero once its usage is logically removed from the system: there may be a remnant source of usage. For example, perhaps an engineer issued some test queries to a database themselves. In this situation, the engineer can instruct SCARF to ignore the usage signal: the engineer's authorisation is then fed into the deprecation candidate selection phase. Similarly, SCARF might detect a code reference which it was not able to remove automatically; in this case, it reports the reference to the developer, who can manually refactor the caller to remove the reference. For instances of SCARF that deprecate data assets, this causes SCARF to comment on the associated ticket for a deprecation indicating the engineer-driven action that was taken; for instances of SCARF that deprecate code, this causes a patch to be generated on behalf of the engineer who took the action, rather than as a service user. This clearly indicates that the patch is in a different category to the regular patches submitted by SCARF, and shows that a human was involved in the patch being generated.

Together these two modes of working (committing changes to production, and ignoring usage signals) create a feedback loop that brings their subgraph into isolation. As components of this subgraph become dead, SCARF will begin initiating their removal. Over time the engineer and SCARF work together to remove the entire subgraph.

4.2.3 Manual Checks and Visualization. We implemented tools to validate and improve SCARF, as well as exporting data to identify improvements that can be applied to many use cases. For example, if SCARF identifies a repeated pattern where nodes of a certain type have low runtime usage but no relationships, it may indicate that the method by which SCARF gathers that runtime usage metric needs adjusting. In one instance, we identified a piece of backup infrastructure causing runtime usage to be logged and thus preventing automated deprecation. This appeared in SCARF logs as a single source of usage connected to a large number of data assets, and thus was discoverable by SCARF engineers. Upon investigation we suppressed this usage traffic as a false positive.

We use deprecation data to power various systems to aid developers in improving code quality. For example, our dataset on runtime code usage is used to power a linter [13] which warns developers of unused functions and suggests their removal. This is a form of code maintainability metric [1]. At a higher level, we use the above subgraph understanding tooling to highlight to developers particular subgraphs which may be unused or have limited usage, and suggest their deprecation.

Finally, the data are also used to highlight opportunities for higher-level business decisions around whether to maintain or deprecate a given feature. Ownership of features may transition to different teams, and that represents a good opportunity to highlight the usage of the feature as a motivating factor for deprecating, rather than continuing, the feature.

5 DEPRECATION STAGE

In the third stage in Figure 1, SCARF initiates a safe removal process for each deprecation candidate subgraph identified in Section 4.2.1. Deprecation begins by notifying the responsible engineering team that safe removal has been initiated. In some configurations, SCARF then waits for an engineer to approve the request, and in others, it waits for a fixed safety window. During this waiting time, the properties of the candidate subgraph that led to safe removal initiation must continue to hold true at every run of the system: if the subgraph stops being a candidate, then safe removal is aborted.

After this waiting period ends, or once the engineer's approval is obtained, where applicable, the instances are then quarantined: SCARF makes the instances inaccessible, such that other code, data and systems are not able to use the instances. For example, in the case of a data table, this could consist of applying a strict ACL to the table to block production traffic. For code, quarantine means landing a commit to remove the code. This quarantine is performed transactionally, since all nodes in the subgraph are part of the same production system. Once quarantine has been completed, a further waiting period is initiated to allow time for any errors caused by this stage to be detected and reported. In this case, engineers can revert the quarantine and report an issue to the SCARF team.

After this second waiting period, SCARF enacts the deprecation, by permanently removing it from the production system or data system. For example, this could consist of dropping a table, or deploying a new version of code to production. Finally, a verification phase ensures that the deprecation was completed successfully, and a record that the removal was completed is stored.

Chained removals. When a node has completed the safe removal process, a check is performed to see if it is the source of any transitive deprecation edges in the SCARF graph. For any identified destination edges, safe removal is also then initiated. If this deprecation cannot be completed due to deprecation-blocking edges, the engineering team responsible for the deprecation is notified and asked to resolve the remaining usage to unblock continued deprecation.

Transitive deprecation. When a subgraph of nodes of the same type are all collectively eligible for deprecation, SCARF removes them in one pass. However, when a subgraph comprises nodes of different types SCARF can still proceed with deprecation by removing each layer of the graph one iteration at a time (assuming the graph is in fact a Directed Acyclic Graph (DAG)). For example, SCARF might remove some code in one repository, which enables the removal of code in another repository. That code's removal may enable the deprecation of an ORM schema, which finally will enable the deprecation of a table of data. This ability to remove dependency graphs that cover code and data is a key property of SCARF.

5.1 CodemodService

We found that scalably generating a large volume of automated changes in code review required its own automated system, which we refer to as CodemodService. CodemodService provides a framework for implementing a *code automation Config*: each Config is scheduled to run every working day. When run, a Config determines the set of *Inputs* that it should be scheduled against: this is the universe of possible files, symbols or assets over which it operates. *Filters* are used to narrow down that set of *Inputs* to a manageable working size. *Inputs* are clustered into *Batches* (*e.g.,* one batch for each set of files owned by a team). CodemodService then fans out a job for each *Batch* to perform the *Config*'s *Transformation* (the point where the code modification occurs).

From here, CodemodService handles the lifecycle of turning this code change into a reviewable patch within Meta's existing code review infrastructure, which is published with the identity of a service user (a "bot"). It publishes the patch in the code review system, and identifies suitable reviewers. Accepted patches are landed during working hours, and rejected patches are abandoned. Patches that remain unreviewed are rebased to ensure up-to-date test coverage signals are present. CodemodService also ensures that no single *Config* generates too many patches, through a rate-limit system.

An open-source system similar to CodemodService called AutoTransform exists outside of Meta [22].

In most cases, the patches generated by CodemodService are treated exactly the same as any other patch produced by an engineer: they are reviewed by other engineers, approved if suitable, and then landed and processed by CI as normal. In a handful of cases, we can be highly confident that the changesets are safe to land without human review—for example, if they are removing a method which we are confident is never referenced or invoked. In these cases we still generate a patch for review to allow developers to provide feedback, but we allow the CI to commit the changes automatically after a suitable waiting period. The metadata prescribing whether a given symbol's removal would be safe to commit without human review is one example of an additional piece of metadata gathered in the collection stage of SCARF: this is derived from a hand-curated set of kinds of symbols, and symbols pertaining to specific frameworks within supported codebases, which are known to not leverage some of the dynamic features (such as reflection) of supported languages.

To ensure that automated patches remain of high quality, CodemodService monitors feedback through two main channels: rejected patches and an explicit feedback form. Developers of a CodemodService config monitor the rate of rejected patches; rejecting patches from a config with automatic commits generates an alert for the owner. All automated patches from CodemodService also include a explicit feedback flow, through which developers reviewing a patch can flag them as incorrect or difficult to understand, or provide freeform comments on the config.

SCARF configs for CodemodService generate hundreds of thousands of commits per year across Meta repositories.

6 DEPLOYMENT SEQUENCING

SCARF was rolled out at Meta over several years, progressively building up to its current state and coverage. The initial phase of deployment started with a proof-of-concept to remove unused HTTP endpoints from a large web codebase. This was the first time that large-scale automated code changes had been done in the company, and engineering culture had to adjust to support them. For example, before automated code removal it was common practice to commit unused code as an example or for future use, whereas now such code is automatically removed and engineers rely either on source control or on documentation. As another example, SCARF was the first system at Meta not geared to a single team's codebase to submit patches authored by automation rather than humans, meaning that code review needed to allow submission of feedback to the author of the automation instead of the patch.

The early systems for removal of data were solely initiated by a human, but with safety checks to prevent production incidents caused by removal of in-use data. In Phase 2, we built a proof of concept to trigger automated removal of data assets in one particular storage system. This highlighted the need for high-quality runtime usage metrics (Section 3); at this point these did not exist for all storage systems and were opt-in where present. The provision of high-quality runtime usage metrics allowed fully automated deprecation initiation to succeed with sufficiently low false-positive rates: automated deprecation was successful and we began rolling it out to more data systems.

Scaling challenges were hit during this stage: we found in the data system TAO [4] we had millions of types configured in production with little or no data, which had been created by an end-to-end test system: these accounted for up to 99% of the types configured in TAO (measured by number of types, not number of rows of data). With these scaling challenges solved we were able to deploy this early system to several other data systems.

In Phase 3, we abstracted the systems above into the architecture described in Figure 1. This allowed us to roll out to many more systems with minimal duplication of effort: adding support in a new system could reuse shared functionality and thus only required implementing the system-specific parts of the architecture. At this stage we introduced a dependency on Glean for code, and on a unified set of usage metrics; we also built a central user interface.

In Phase 4, we built the additional tooling described in Sections 4.2.2 and 4.2.3. This enabled SCARF to build a feedback loop: where it was unable to identify a deprecation candidate, it could surface deletion-blockers to relevant engineers via internal tooling. This reveals new deprecation candidates to SCARF, which removes them and in turn creates new deprecation candidates. At this stage we also integrated SCARF with company process: for example, when a decision is taken to deprecate a product, the relevant engineers can inform SCARF of the desired deprecation and use the internal tooling to track its progress.

6.0.1 *Early designs of SCARF.*

Earlier versions of SCARF did not collect all usage metadata ahead of time: these earlier versions were both less efficiently implemented, and were very conservative in the number of assets they would deprecate in one go. As such, earlier versions instead only collected the metadata for assets as those assets were being selected for deprecation: the metadata collection was invoked synchronously when needed. This allowed for faster iteration and tighter testing loops which did not involve waiting for batch data analysis jobs to complete.

6.0.2 *Adding automated deprecation to a new system.*

One specific goal in designing the SCARF architecture was to make it as easy as possible to add automated deprecation to a new system. The process to do so is as follows. First, engineers implement the system-specific parts of SCARF and integrate them with the new system. This generally consists of an asset enumeration (e.g., listing all the database tables or code files) and reference enumeration, which are provided to SCARF via a standard API. At this stage we configure SCARF to perform only manual deprecations at the behest of an engineer, and to expose a simple interface to start or stop deprecation for a given asset. We run SCARF in this configuration for a period of time to monitor the correctness of the implementation; for example, to confirm that requesting the deprecation of a table does eventually lead to its complete removal.

After confidence in correctness is reached, we configure SCARF to begin automatically selecting deprecation candidates at a small rate e.g., 5 assets per day. At this quantity, each candidate can be manually inspected by an engineer in a timeframe much shorter than the waiting periods configured in SCARF. Again, we run SCARF in this configuration for a period of time to monitor correctness, focusing this time on developer feedback received. For example, receiving feedback that a deprecated asset was still being used may indicate errors with the runtime usage metrics. The daily limit is gradually increased until it can be removed completely, and SCARF is able to deprecate the full backlog of candidates which it identifies.

At this stage few deprecation candidates remain in the system, and we generally configure SCARF to gather more information when an engineer requests to stop a deprecation. For example, we may ask for specific evidence that the asset is in use, or an impact assessment [27] of the risks of preserving it despite lack of usage.

7 IMPACT OF SCARF AT META

SCARF has had a multifaceted impact on software development and deployment at Meta. We have a set of dashboards that monitor the health of SCARF and these also track its impact. In this paper, we report summary statistics on the number of lines of code and bytes of data that have been removed. We also report success rates for automatically generated deletion diffs. Another important saving has been the reduction of compute power and we estimate those savings.

Code: In total, SCARF has deleted over 100 million lines of code, with over 46 million lines deleted in 2022 alone. These lines were deleted in over 300,000 diffs, with over 140,000 diffs in 2022 alone. SCARF is effective in multiple languages deleting code from Hack, Python, JavaScript, Objective C, CSS, Thrift schemas, and GraphQL schemas.

Code patches are automatically generated for deletion and we see a patch generation success rate for Hack of 97% in the last three months. While not all of these diffs land, the majority indeed are accepted and subsequently landed, some by automation and some by humans. One area of future work is finding a reviewer who is able to understand the removal of dead code that may no longer be owned as well as diffs that crosscut multiple parts of the codebase. Another issue is that of finding the balance between sending all diffs in a given part of the codebase to a single expert reviewer, versus balancing review load amongst a team responsible for a feature: leaning too much on a single person can create fatigue, however sharing review load between too many people creates duplicated context-gathering as each engineer learns the same set of information to determine how to review the diff.

Engineers are able to select subgraphs that are not completely dead and have complex dependencies. For example, a decision may be taken to deprecate a product with low but nonzero usage, and engineers must locate the sources of this usage and disable them. This feature is successful, with over 7000 completed complex subgraph removals. There are many ongoing projects, since the time-scale of these removals can be that of many months, sometimes even years: a single SCARF deprecation can take over a month of waiting time from start to finish, and this number compounds when there are dependencies between assets that take a long time to deprecate in a single subgraph. Finally, much cleanup work at Meta is done alongside other projects.

Data: SCARF to date has removed petabytes of data from Meta's data warehouse and online data systems, across the many different instances of SCARF. This corresponds to millions of types of data;

in some cases (such as TAO), SCARF was able to delete over 98% of its defined data types.

When SCARF deletes code, that enables the further deletion of data; SCARF has instances that target ORM schemas to handle this case in particular, however regular code removal can also enable further data deletion if the final relation to a data type is removed.

SCARF's code removal instances have used this type of "chained deprecation" to enable the deletion of terabytes of data. Through this process, SCARF has removed over 7000 data logging ORM schemas, over 6000 online data access ORM schemas, and over 40,000 data warehouse ORM schemas. By eliminating these data warehouse ORM schemas, the removal of the corresponding processing pipelines also saved over one megawatt of compute power.

8 RELATED WORK

The concept of dead code, also referred to as unused code, unreachable code, or lava flow, is a frequent topic in code-smell catalogs due to its alleged detrimental effect on the comprehensibility and maintainability of code and related assets [14]. Various dead code detection techniques have been suggested to aid developers in their refactoring efforts. As an example, Chen *et al.* [7] proposed a data model for C++ software repositories that supports reachability analysis and dead code detection. Fard and Mesbah introduced JSNOSE [11], a metric-based technique for detecting dead code in JavaScript. Other researchers have proposed dynamic techniques, static techniques, or a combination of both to detect and eliminate dead code [14, 15]. Most related work techniques are either language specific, are not automated, and have not been demonstrated to work at industrial scale. In contrast, we are proposing an automated technique that is capable of (i) dealing with distinct programming language and (ii) working at scale.

Many large companies use feature flags, toggles, or gates to allow new features to be quickly implemented and rolled out to a small group of users to determine that the feature works properly and performs well according to a predefined metric. However, as Rahman *et al.* [19] showed on Google Chrome, this technical debt quickly builds up as new features must be stable before old feature can be turned off. Chrome introduced a manual spreadsheet of outdated features, however, after an initial dip in outdated features, toggles continued to grow exponentially. These features no longer are run after the toggle is turned off. SCARF can target these experimental features and automatically change code to remove them.

The work most closely related to ours is PIRANHA [20, 21] that has been successfully deployed by Uber to handle stale feature flags. PIRANHA identifies obsolete feature flags in Objective-C, Java, and Swift programs. By incorporating into developers' CI workflows, PIRANHA is able to do perform large-scale cleanup of extensive codebases with millions of lines of code and spanning multiple programming languages. PIRANHA is closely related to work focusing on eliminating code protected by outdated C preprocessor conditionals and libraries [2, 3, 6]. SCARF advances beyond the current state-of-the-art by analysing not just flags and call dependencies, but also identifying a lack of runtime usage.

AutoTransform [22] is a system similar to Meta's CodemodService, designed to produce code patches for developer review. In particular, AutoTransform is a tool — developed by Slack for automated code transformations — designed to help organizations make large-scale changes to their codebase efficiently, without requiring manual modifications of files. AutoTransform allows developers to define rules for how code should be transformed (e.g., renaming a function, changing an API endpoint, or updating a library version), and the tool will automatically apply those changes across your codebase. The tool also includes features for managing conflicts and handling errors that may arise during the transformation process. Similar to CodemodService, this can save a lot of time and effort compared to manually editing each file. This is particularly important when dealing with large-scale changes that would be difficult to handle manually.

9 CONTRIBUTIONS AND CONCLUDING REMARKS

We have introduced SCARF, our automated dead code and data deletion system. Our system was designed to work at scale and has deleted over 104 million lines of code and petabytes of data. Our major contributions are:

(1) Code deletion at scale is possible, even in non-typed, dynamic languages that have reflection and side-effects.
(2) Data deletion at scale is possible, even if there are complex, heterogeneous data systems and data processing pipelines.
(3) Understanding code and data usage from a business perspective, *i.e.* runtime usage, as well as a programming language perspective is important for successful deprecation.
(4) While most deletions happen automatically, complex deletions can unblock automation by allowing engineers to inspect subgraphs and remove dependencies.
(5) Building a standardised API for accessing metadata about code and data assets is key for integration with other business tools, and allows deprecation systems to be used across multiple languages.

The scope, scale, and diversity of programming languages used at Meta make our approach applicable to most software systems. We hope that the description of our system will inspire other companies to institute dead code and data deprecation, and allow researchers to understand the difficulties in dealing with technical debt at scale.

REFERENCES

[1] Luca Ardito, Riccardo Coppola, Luca Barbato, and Diego Verga. 2020. A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review. *Scientific Programming* 2020 (Aug. 2020), 1–26. https://doi.org/10.1155/2020/8840389

[2] I.D. Baxter and M. Mehlich. [n.d.]. Preprocessor conditional removal by simple partial evaluation. In *Proceedings Eighth Working Conference on Reverse Engineering*. IEEE Comput. Soc. https://doi.org/10.1109/wcre.2001.957833

[3] Ira D. Baxter. 2002. DMS. In *Proceedings of the International Workshop on Principles of Software Evolution*. ACM. https://doi.org/10.1145/512035.512047

[4] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. {TAO}: Facebook's distributed data store for the social graph. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*. 49–60.

[5] Nathan Bronson, Thomas Lento, and Janet L Wiener. 2015. Open data challenges at Facebook. In *2015 IEEE 31st international conference on data engineering*. IEEE, 1516–1519.

[6] Michael D Brown and Santosh Pande. 2019. Carve: Practical security-focused software debloating using simple feature set mappings. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*. 1–7.

[7] Yih-Fam Chen, Emden R Gansner, and Eleftherios Koutsofios. 1998. A C++ data model supporting reachability analysis and dead code detection. *IEEE Transactions on Software Engineering* 24, 9 (1998), 682–694.

[8] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *CIDR*, Vol. 3. 3.

[9] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Transactions on Storage* 17, 4 (Oct. 2021), 1–32. https://doi.org/10.1145/3483840

[10] facebookincubator. [n.d.]. *GitHub - facebookincubator/Glean: System for collecting, deriving and working with facts about source code.* Meta Research. https://github.com/facebookincubator/glean

[11] Amin Milani Fard and Ali Mesbah. 2013. Jsnose: Detecting javascript code smells. In *2013 IEEE 13th international working conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 116–125.

[12] Dror G. Feitelson, Eitan Frachtenberg, and Kent L. Beck. 2013. Development and Deployment at Facebook. *IEEE Internet Computing* 17, 4 (2013), 8–17.

[13] Stephen C Johnson. 1977. *Lint, a C program checker.* Bell Telephone Laboratories Murray Hill.

[14] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software* 167 (2020), 110610.

[15] Ivano Malavolta, Kishan Nirghin, Gian Luca Scoccia, Simone Romano, Salvatore Lombardi, Giuseppe Scanniello, and Patricia Lago. 2023. JavaScript Dead Code Identification, Elimination, and Empirical Assessment. *IEEE Transactions on Software Engineering* (2023).

[16] MariaDB Foundation. 2019. MariaDB. https://mariadb.org/

[17] Simon Marlow. [n.d.]. *Incremental indexing with Glean.* Meta Research. https://glean.software/blog/incremental/

[18] Phacility. 2011. Phabricator. https://www.phacility.com/phabricator/

[19] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C. Rigby, and Bram Adams. 2016. Feature Toggles: Practitioner Practices and a Case Study. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 201–211. https://doi.org/10.1145/2901739.2901745

[20] Murali Krishna Ramanathan. 2020. Introducing Piranha: An Open Source Tool to Automatically Delete Stale Code. https://eng.uber.com/piranha/. Accessed: 2022-03-30.

[21] Murali Krishna Ramanathan, Lazaro Clapp, Rajkishore Barik, and Manu Sridharan. 2020. Piranha. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. ACM. https://doi.org/10.1145/3377813.3381350

[22] Nathan Rockenbach. 2022. AutoTransform. https://slack.engineering/autotransform-efficient-codebase-modification/

[23] Mike Starr. [n.d.]. *Dataswarm.* Meta. https://www.youtube.com/watch?v=M0VCbhfQ3HQ

[24] Liyin Tang, Vinod Venkataraman, and Charles Thayer. 2012. Facebook's large scale monitoring system built on HBase. In *Strata Conference, New York*.

[25] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. 2010. Hive-a petabyte scale data warehouse using hadoop. In *2010 IEEE 26th international conference on data engineering (ICDE 2010)*. IEEE, 996–1005.

[26] Jeroen Vaelen. [n.d.]. *Searching through Code at Scale.* Meta. https://www.facebook.com/watch/?v=1911812842425144

[27] David Wright and Paul De Hert. 2012. *Privacy impact assessment.* Vol. 6. Springer.