Modular Architecture for Code and Metadata Sharing

Tomas Tauber Bruno C. d. S. Oliveira
The University of Hong Kong, Pok Fu Lam Road, Hong Kong, China
{ttauber,bruno}@cs.hku.hk

Abstract

Every fragment of code we write has dependencies and associated metadata. Code dependencies range from local references and standard library definitions to external third party libraries. Metadata spans from within source code files (hierarchical names and code comments) to external files and database servers (package-level dependency configurations, build and test results, code reviews etc.). This scattered storage and non-uniform access limits our programming environments in their functionality and extensibility.

In this paper, we propose a modular system architecture, **Haknam**, better suited for code and related metadata sharing. **Haknam** precisely tracks code interdependencies, allows flexible naming and querying of code references, and collects code fragments and their related metadata as messages in a distributed log-centric pipeline. We argue that this setting brings considerable advantages. In particular, we focus on modular development of tools and services that can assist in programming-related tasks. Every new functionality can be simply added by creating and processing messages from the distributed pipeline.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques—Computer-aided software engineering (CASE); D.2.6 [*Software Engineering*]: Programming Environments—Integrated environments; D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules, packages

Keywords Haknam, code slice, Haskell, fragmented code distribution, evidence, tags, event log

1. Introduction

Software development evolved rather spontaneously and organically from different scientific and engineering communities [25]. Its early days started with very "localized" environments. Before the internet or even personal workstations, collaborative development mostly happened within individual organizations. Code sharing implied shared physical access or physical media exchange. "External" code dependencies would most likely come from within the same organization. How well this worked or not depended on internal coordination. With the advent of portable operating system and programming environments, there has been an explosion of various productivity tools, from libraries, version control to build systems reused across different organizations.

The internet adoption later provided new means of code sharing on a global scale and this trend accelerated. Instead of developing code from scratch within a single organization, software development relies on external code dependencies from many different sources. Despite that, some of our conventions and unconscious assumptions remain almost as if nothing changed from the early days.

Common programming languages and their compilers implicitly treat all source code as something originating from the local filesystem. This convention then requires using external tools for separately storing and managing shared source code and metadata about it (where it came from, what version it is, etc.). Even with that, problems with name clashes or versioning arise, because the world would not agree on a global filesystem structure. Again, different tools and techniques exist to bypass this problem. One common approach is to use "stable distributions" (e.g. Anaconda [7]) where community maintainers compile a vetted and tested list of compatible packages. Another common approach is "version pinning" (e.g. Gradle [18]) of packages. Both of these approaches, however, ad-hoc emulate a centralized "single organization" environment. In that sense, this implies a fixed topology of inter-project dependencies. Changing it (e.g. merging or splitting out projects) would imply refactoring and a detached version history.

Another point is that commonly used imprecise indirect versioning may still cause problems. Firstly, versioning is a form of metadata outside language specifications and compilers generally ignore it. Therefore, programming language environments often cannot deal with depending on multiple versions of the same library. Secondly, using a single release number and coarse-grained dependencies may obfuscate and

MODULARITY'16, March 14–17, 2016, Málaga, Spain
© 2016 ACM. 978-1-4503-3995-7/16/03...
http://dx.doi.org/10.1145/2889443.2889455

complicate upgrades. Some package maintainers suffer from *Hauptversionsnummernerhöhungsangst*[1] [27] and introduce API-breaking changes as non-breaking. On the other hand, some API-breaking changes may be non-breaking to people who depend on a subset of a particular library. One possibility to mitigate these problems is to enforce semantic versioning. Unless community guidelines suggest otherwise, this often assumes compatible API signatures are the only external dependency requirement. It may be, but we may have other requirements on the external code: e.g. that it compiles with a specific compiler version on a specific platform, it has a good performance or passes all relevant tests. These requirements vary individually from one project to another; and the external code's author may not have all of them in mind.

Overall, a lot of metadata surrounds today's source code. Unlike source code, which tends to be in text files, metadata is scattered in different formats and places. We embed some metadata directly in the source code (e.g. code comments). At the same time, we place other metadata in different files in the same repository (e.g. versioning information) or in various databases and services (e.g. test results). Combining and processing this scattered metadata may be non-trivial and non-modular. Building and extending development tools is tied to a local environment and limited by what metadata they have access to. In other words, *the lack of a modular integrated infrastructure* limits what can be done in a programming language environment.

To address this issue, we propose **Haknam**, a modular architecture designed for the today's distributed software development. **Haknam** builds on three core ideas:

1. **Haknam** does not store shared source code in text files, but in uniquely identifiable parts of reusable functionality. Each part precisely describes its dependencies and can be used independently of irrelevant parts of a particular project and their dependencies.
2. **Haknam** does not impose a rigid filesystem-like structure, but allows flexible labeling of source code parts. There are two types of labels: tags added by users, and evidence which denote machine-verifiable properties. With these flexible labels, source code parts can be searched and work together without globally imposing any particular development workflow, conventions or naming structures.
3. Source code and metadata are uniformly stored and accessed as messages in a distributed append-only sequence. This provides a clear separation between writing new shared data and processing existing data into specialized services (e.g. for fast access of certain queries). Combining and processing metadata is a matter of filtering certain messages. We can extend functionality by writing new types of messages (that existing services may ignore) and services that will process them.

[1] German word for the fear of increasing the major version number

We describe the overall architecture and these three ideas of **Haknam** in Section 2. In Section 3, we outline the prototype implementation of **Haknam**, including an experimental web-based development environment. Then in Section 4, we present several prospective applications on top of **Haknam**. We discuss related work in Section 5. Finally, we expand on limitations of our prototype and overview future research directions in Section 6, and add a few concluding remarks in Section 7.

2. Vision Overview

This section shows the overall architecture and describes the main concepts behind **Haknam**: fragmented code distribution, code organization with tags and evidence, and unified data collection in the event log.

2.1 Once Upon a Programming Environment Design

In this section, we look at the extensible design intuition behind **Haknam** that underpins tool development. The overall architecture based on this intuition is shown in Figure 1. This Figure refers to three building blocks described later: code slices (Section 2.2), flexible and reliable code-related metadata (Section 2.3), and log-based storage (Section 2.4). Nonetheless, in an analogy to the Unix programming environment, development tools and services follow certain design principles:

- Each service (program) does one thing well.
- Each service is a "filter" in the sense that it subscribes to different log partitions (distributed log systems are generally divided into different partitions or topics) and processes certain input messages from it. Unlike Unix pipes, log partitions are distributed and allow multiple publishers and subscribers.
- Each service consumes and produces event log messages. This acts similarly to programs reading and producing textual stream in Unix.

Other storage mechanisms can back different services, depending on their purpose (full-text index for API lookups, deductive databases for code dependencies, etc.). The data they contain is eventually consistent with the event log. Overall, **Haknam** utilizes two forms of extensibility:

1. We can add new forms of metadata, without affecting or changing existing metadata and services. The content of log messages can be arbitrary (any structure and any serialization format), so existing services only subscribe to the ones they can process and ignore the rest. New services can simply read or produce new metadata as messages that adhere to their processing conventions.
2. We can add new services, without affecting or changing existing metadata and services. New services can use and combine existing services, code slices and metadata about them.

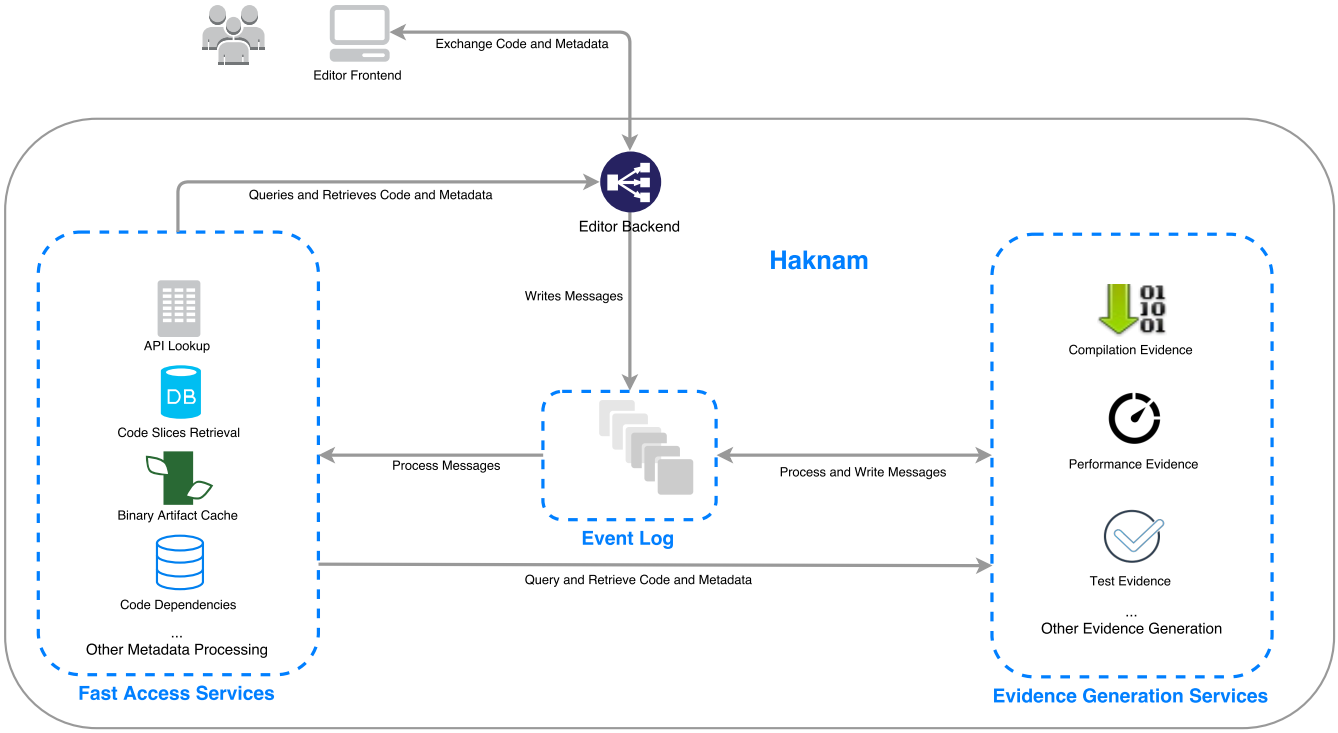

Figure 1. Overall architecture: code slices and related metadata are appended to a persistent log. The log is processed by specialized services that provide fast access to some queries and services that verify requested evidence (e.g. if a given code slice compiles) and write the result back to the log.

In Sections 2.2, 2.3, and 2.4, we describe individual concepts that contribute to the overall modular architecture. In Section 4, we explore various applications on top of **Haknam** that benefit from its extensible design.

2.2 Enter the Fragmented Code Distribution

In this section, we describe how **Haknam** handles code dependency metadata.

File Bundle Code Distribution. When discussing metadata about code, it is useful to look at the code itself. If we omit line and block comments, code contains (usually imprecise) metadata about its dependencies. In plain programming languages, the entry point for using external code is by writing (or generating) a statement similar to this one:

```
import Path.To.External.Code.File (externalNameReference)
```

There could be other keywords, such as for renaming or using qualified names. In any case, this is a form of metadata about the code embedded in its textual representation. Different tools may be interested in different portions of the stored code file. If a tool tries to process it with the respect to its corresponding abstract syntax (compilation, analysis, testing etc.), it only matters that: the tool can go to the given path (be it a local filesystem-related path or URL), fetch the given file (be it a source file or an object file with interface) which should contain the given name.

For this code-related processing, human-readable names, filesystem paths and the traditional file bundle distribution do not matter. In fact, they rather complicate the task:

1. Name or path conflicts may arise.
2. Referenced files may contain code which is irrelevant to code dependent on these files.
3. Irrelevant code may fail to compile or increase processing time.

Fragmented Code Distribution. To extract the precise dependency metadata, we move away from the traditional file bundle code distribution. Instead, we turn to the fragmented code distribution (FCD). In FCD, each individual declaration is exposed in its single encapsulated compilation unit with a unique name. Single compilation units import from other single compilation units only if they are true dependencies. The immediate advantage is that FCD does not introduce complications for processing that we had with the traditional file bundle code distribution. The potential disadvantage is that there may be more code duplicity, i.e. different units may share their dependencies. We, however, do not assume these units are written manually by programmers, but rather assisted and generated. Note that the traditional file bundle code distribution, despite files being written manually, still requires external tools to manage versioning, builds and external dependencies.

Before looking at an example of FCD features, we establish some terminology and background. Due to our prototype implementation (described in Section 3.1), we show code examples in Haskell in this section. We borrow the FCD concept from Fragnix [31], a new Haskell code/package manager, and will use its terminology. In Fragnix, a single declaration (of a function, a datatype, or a type class) is called a code *fragment*. An example fragment is shown in the left column of Table 1. This fragment is a function that takes a credit card number, checks whether it is valid according to the Luhn algorithm and either returns a value representing the card issuer or an error. Fragments themselves do not capture dependencies. For that, Fragnix operates with code *slices*. Each slice consists of four parts: a fragment, all its dependencies, any language extensions, and a unique ID. A unique ID is generated as a hash of the slice's content. Slices can then be represented as ordinary Haskell modules. For example, the slice that includes the example fragment is shown in the right column of Table 1. Code slices have two properties desirable for code-related processing: 1) name conflicts do not arise; 2) referenced code slices only contain code which is relevant to code slices depending on them.

The unique IDs guarantee that there are no conflicts – e.g. two different versions of the same function end up in two different unique code slices. The exact dependencies reduce processing workload and prevent other problems – e.g. when code slices fail to compile, it does not affect code slices that do not depend on them. This is not always the case in the traditional file bundle code distribution.

2.3 Crouching Tag, Hidden Evidence

In Section 2.2, we focused on the code processing aspect of external dependencies. In this section, we describe how **Haknam** handles code organization or naming-related metadata.

Filesystem-like Code Organization. The file bundle code distribution naturally ties with the filesystem-like namespace organization. This code namespace organization works with the fragmented code distribution as well – Fragnix defines *environments* to be modules that re-export code slices under saner names. In this setting, let us now look at an example task: searching for a new API. For instance, we can look for a credit card validation function. In the current development workflows, we will follow something along these steps:

Step 1: We will search with a query "credit card validation" either in a general purpose web search engine (and include the programming language name in the query) or in an API-specialized search service.

Step 2: After browsing results, reading through documentation and discovering a package name, e.g. "validation-tools", we will write it in some external dependency configuration file and get it downloaded.

Step 3: Finally, we will write something like this in our source file: $\mathbf{import}\ Data.Validate\ (luhnValid)$.

We see that the names we ended up with have not much in common with the original keywords in the query. It may be the case that Steps 1 and 2 take some time and that most developers search for this function with similar keywords. Step 2 may involve checking build and test results or other metadata.

We would then desire to create a tool, integrated in some development environment, that would help with such API searching tasks. We would face different obstacles. One major obstacle would be relating different metadata (original names, queries, build results etc.) spread across different sources – some metadata (such as query history) may not even be publicly available.

Label-based Code Organization. If we look at all the metadata involved (from names and queries to build results), we can think of it as programmer-written or machine-generated labels attached to some code slices. In other words, **Haknam** offers a flexible and direct scheme for code organization: every code slice can be described with an arbitrary number of user-defined labels (*tags*), and external dependencies can be looked up using those labels. Users can attach labels to any existing code slices. When writing a code fragment, queries specify all its external dependencies.

The example of that is shown in the middle column of Table 1 – each external name is accompanied by a query with tags denoted by #. The other possibility for lookups is to use *evidence* (denoted by @). Evidence labels are labels which imply stronger machine-verified guarantees about external dependencies. In other words, tags can be arbitrarily inserted, but evidence needs to be accompanied by a machine-verifiable justification.

The basic example of an evidence is publication. Once the code slice is added to some user repository that is shared online, it implicitly gets evidence for that. User repositories could be similar to project source code repositories as well as to various merged channels, such as channels that take and republish code slices from different projects. In that sense, stable distributions and other centralized solutions that exist today are special cases that can emerge in **Haknam**.

We can, however, make evidence as precise as we wish to: for example, we may require evidence for an exact version of the compiler (since ABI may be different between two compiler versions) on a particular platform. In Table 1, @ghc7.8 and @linux86 request evidence that the external code should be successfully compiled with the compiler GHC version 7.8 on the x86 Linux platform. Other examples of evidence include property-based randomized testing or unit tests where a justification is in the form of machine readable report of successful test passes. Evidence generation can be extended to any propositions that can be automatically verified. This includes using external verification tools and specifications. Beyond verification, we may, for ex-

Table 1. This table shows three main parts that we operate with in our prototype: 1) the user writes a code fragment, 2) the editor pre-fills queries for unresolved names in the code fragment and the user verifies them and may change them (as described in Section 3.5), 3) from the code fragment and relevant queries, the code slice is generated. Later, various tags or evidence may be added about this code slice.

Code Fragment (*written*)	**External Queries (*assisted*)**	**Code Slice (*generated*)**
$convertCard$ :: $\quad CardNumber$ $\quad\quad \rightarrow CardType$ $convertCard\ no =$ **if** $luhnValid\ no$ **then** $getCardType\ no$ **else** $cardError$	**CardNumber** : $\#credit\ card$ $\quad \#\ payment \#\ bank$ **CardType** : $\#credit\ card$ $\quad \#\ Visa \#\ MasterCard$ **luhnValid** : $\#credit\ card$ $\quad \#\ validation \#\ Luhn$ $\quad @ghc7.8@linux86$ **getCardType** : $\#credit\ card$ $\#\ detection@ghc7.8@linux86$ **cardError** : $\#invalid \#\ card$ $\#\ error@ghc7.8@linux86$	{-# LANGUAGE NoImplicitPrelude, (...) #-} **module** $F6438570461503997890$ **where** **import** $F2895876643766170726\ (luhnValid)$ **import** $F2982558241273070393\ (getCardType)$ **import** $F4230983858141319800\ (cardError)$ **import** $F8490776344619887876\ (CardNumber)$ **import** $F7326499101269886236\ (CardType)$ $convertCard :: CardNumber \rightarrow CardType$ $convertCard\ no =$ **if** $luhnValid\ no$ $\quad$ **then** $getCardType\ no$ $\quad$ **else** $cardError$

ample, require evidence for rigorous performance measurements below certain thresholds or for code style adherence.

If we revisit the example task of searching for a new API, we do Steps 1-3 in one go. The flexible code organization brings us a possibility to reach the external code in different ways: some people may look it up with $\#Data\#Validate\#Luhn$, others with $\#Validate\#Data\#Luhn$, or something different such as $\#credit\ card\#validation$. Some may also include extra keyword annotations denoting some desirable properties (e.g. that the external code was tested). Queries with evidence labels then bring some guarantees when a user chooses to recompile a fragment with different dependencies (i.e. when new slices matching the query appeared). In any case, there are different ways how to reach the external code and all of them are recorded.

If we were to create a tool that assists with API searching, we have all relevant metadata at our fingertips. We can create a simple tool, processing all query and label metadata into an inverted index mapping keywords to external code. Given a query, the tool can suggest an external code by ranking it against previously stored keywords from queries. If $\#credit\ card\#validation$ is a fairly common query among all users, we would get to that external code immediately with the original query of Step 1.

2.4 The Way of the Event Log

In Sections 2.2 and 2.3, we described how **Haknam** separates and handles different kinds of code-related metadata. In this section, we look at how metadata is stored and processed.

Heterogenous Non-uniform Source Storage. Traditionally, we do not find metadata in a uniform source. Code with some of its metadata is usually stored in text files. A history of changes is kept separately in internal version control files. Other metadata (such as continuos integration results) reside in remote databases. They may use different ways of access – from HTTP requests, custom communication protocols, to various middleware APIs. This has unfortunate consequences on metadata processing scenarios.

It becomes non-trivial and non-portable to manage, combine, and extend metadata from different sources. An example where complications happen is with inter-project topological changes (such as splitting one project into two). In this example, some relevant metadata before changes may become "lost" (disassociated) after changes – for instance, if split project code is copied to a fresh new repository without its history of changes. Even with its history of changes, some metadata, such as old test results, may not be carried over.

Log-based Uniform Source Storage. **Haknam** stores all data in the event log. The event log is a simple data storage abstraction: append-only (previously written items are immutable) and totally-ordered sequence of records. The current state can be thought of as a result of selective aggregation over this sequence. For example, if we have three events about slice with id 2895876643766170726:

1. Add tags $\#credit\ card$ and $\#validation$ to this slice.
2. Add a tag $\#Luhn$ to this slice.
3. Add evidence labels @ghc7.8 and @linux86 to this slice.

If we aggregate events for this slice, we can deduce its current set of tags contains $\#credit\ card$, $\#validation$, and $\#Luhn$; and its evidence contains @ghc7.8 and @linux86. The ultimate purpose of this abstraction is to keep a single source of what happened and when. This is sometimes called

Event Sourcing or atomic broadcast [9]. Event logs tend to be central points in many distributed system architectures for many reasons. Firstly, they form one reliable source of all data, so tools do not need to worry about explicit synchronization and fault tolerance. This is in contrast with the traditional non-uniform source storage where tools need to worry about explicit synchronization and may not access some sources simultaneously. In this setting, inter-project changes do not cause any complications, since all old data remain accessible for processing. For example, project splitting would correspond to writing events representing the creation of the new project and moving split code slices to it from the old project. Secondly, they provide a clear separation between writing and reading the data, which guides implementation. Thirdly, several distributed high-availability log systems exist in production [15, 22]. Finally, these systems allow processing in realtime, so all tools can start processing new data as soon as it arrives.

3. Prototype Implementation

This section contains implementation details and discusses limitations of our prototype for **Haknam**. In addition to that, it also describes an experimental web-based programming editor environment. Development tools and services on top of **Haknam** interact with this environment.

We chose Haskell for the prototype implementation. We favored Haskell over other languages for two reasons: 1) Haskell is a purely functional language; 2) Haskell's module system is relatively simple, i.e. it mostly does name resolution. Both cases make code reorganization for precise dependencies easier, e.g. we do not need to worry about global mutable variables. In Section 6.4, we discuss how our work could extend to other languages.

3.1 Code Slices

We reused Fragnix's internal implementation in our prototype. A few technical details may change in later implementation. Firstly, builtin definitions from the Haskell standard library (Prelude) are not sliced, but imported directly. Ideally, this may change and builtin definitions will not need to be treated in a different way.

Secondly, code slice hashes are computed from the textual representation of their code fragments, dependencies, and language extensions using the FNV-1 algorithm. This approach suffices for our prototype, but a real system would benefit from two changes. One change would be to use abstract syntax trees rather than textual representations in hash computations. This change would make code slices more robust to local name refactoring. The other change would be to store metadata about the used hash algorithm. This would strengthen the implementation against system changes.

Thirdly, single name declarations export multiple pieces: functions export their type signatures and their definitions, datatypes export their names and constructors, and type classes export their names and methods. Our prototype implementation does not handle type class instances well and does not support advanced language features, such as metaprogramming language extensions. Again, a full system implementation would need to address these issues.

3.2 Name Queries

Section 2.3 described two types of labels used for flexible code organization. The middle column of Table 1 shows examples of external name queries to generate the code slice in the right column from the code fragment in the left column. In our prototype, these name queries are resolved as boolean queries of the standard Boolean information retrieval (BIR) model [23]:

$$matched = id \wedge (\bigwedge_{e \in evidence} e) \wedge (\bigvee_{t \in tags} t)$$

There may be no matched code slices. In that case, an error is shown. There may also be multiple matched slices. In that case, we rank the matches by the number of overlapping tags and by their age: the one with the most overlapping tags is selected; in case of draws, the latest one is selected. Other ranking schemes are possible; for example ranking by the total number of external code slices that are importing the ranked slice. However, the ranking with overlapping tags is fairly flexible – since the programmer could see intermediate search results in the editor (described in Section 3.5), he/she could always refine the query and add more specific tags or fix it with some specific evidence.

For example, if one needs to disambiguate between *convertCard* and *getCardType* which may share similar metadata, one could use their identifiers to do so. A more reliable approach would be using an evidence label denoting that a function passed a test suite which feeds it with valid and invalid card numbers. On the other hand, one may only need the latter function and want to do a custom validation, for instance, for performance reasons, because $luhnValid$ may be inefficient. In that situation, there could be an evidence label denoting passing a strict certainty threshold on rigorous performance measurement of repeated function invokations with randomized card numbers. In any case, the simple ranking scheme and query language were a prototype implementation choice. We discuss the possibility of a more expressive query language in Section 6.4.

3.3 Storage

There are different possibilities how to implement ideas described in Section 2. Given the event log, one natural choice is to use a log-centric distributed system. In our prototype, we used Apache Kafka [22], but other systems (e.g. Event Store [15]) would be equally good. The event log drives the overall architecture design shown in Figure 1, since the main communication happens via appending and reading messages from it. In Kafka, the log is partitioned into topics. We create topics for the following two purposes:

1. "System" topics: these are single word topics for short system-wide announcement messages, such as registration of new users or creation of new repositories.
2. Project topics: for each user project, a topic "user.project" is created and gathers all messages related to changes in that project.

There are different message types. All of them are written as "Keyed Messages" in Kafka. The key denotes the message type, e.g. "add-tags-<slice-id>", and its content is in JSON which depends on the message type, e.g. a JS array of tags to be added. For example, the body of the first add tags event from Section 2.4 would look as follows:

```
{"sliceID":2895876643766170726,
 "tags":["credit card", "validation"]}
```

This approach is suitable for our prototype, but two changes may improve a full system implementation. Firstly, key names could contain information about their content serialization format. Other serialization formats may later evolve and gain usage in some communities. Secondly, schemas for message content would be recorded in a dedicated system topic. These would be messages that may refer to other previous schemas and would contain a hash code computed from the schema structure. Key names would refer to these hash codes. This change would account for potential evolution in exchanged information (e.g. extra fields).

3.4 Basic Functionality

The implementation of services is language-independent. For our prototype, we used Haskell.

Publication Request. The publication request's message contains a code fragment, its external name queries, and an optional back-reference to a code slice. The back-reference is used when the generated code slice is supposed to replace an existing code slice.

Code Slice Generation. The input message is the publication request. At first, external name queries are evaluated. For that, a label database is used to lookup external code slice IDs. If that succeeds, a new code slice is generated. From Fragnix's momentarily implementation, the hash ID is generated from the source code text (as described in Section 3.1). The output message with the newly generated code slice is then written. If the request contained a back-reference, a removal message for the old code slice is also written.

Tag Insertion. The tag insertion message contains a sequence of tags to be added to a given code slice.

Code Slice Display and Removal from Projects. The service processes code slice generation messages into a key-value store that maps code slice IDs to the actual code. When a code slice is "removed" from a project (i.e. the log contains an event indicating removal), it is still processed and other code can depend on it. Project attachment, hence, serves more as a special label for project-based display in the editor. Even if a code slice becomes "removed" from all existing projects, its original creation event persists in the log. A user could still potentially find this code slice and copy it.

Code Slice Transitive Dependencies. This service also processes code slice generation messages, but into a deductive database where it captures dependency relationships among code slices. This is then used for retrieving transitive dependencies of a particular code slice (the input message contains a slice ID, and the output is a sequence of IDs).

Code Slice Label Attachment. Tag and evidence insertion messages are processed into a relational database. This database is then used for resolving external name queries (the input message is a sequence of queries, and the output is a sequence of IDs).

Evidence Request. The evidence request contains the slice ID and the desired evidence label.

Evidence Generation. This service processes the evidence requests. In our prototype, we assumed a trusted execution service and only implemented an ad-hoc compilation evidence. After reading the request with certain evidence labels, it queries the deductive database for transitive dependencies of a compiled code slice. It then tries to compile the slice with its dependencies. If compilation succeeds, it writes a message with a sequence of desired evidence labels. We plan to extend the prototype to handle general evidence generation by reusing existing continous integration software.

3.5 Experimental Editor

The editor frontend is independent of the underlying architecture (Figure 1) and potentially any existing editor, such as Emacs, could take this role. The concepts we introduce in Section 2, however, move away from a traditional text file-oriented source code storage with scattered metadata. Thus, adapting traditional text editors to this scenario may require a lot of engineering effort. For this reason, we decided to implement a prototype of an experimental "fileless" development environment. Our motivation for developing this editor prototype was to have an easy control over its user interface for rapid prototyping. This allows us to illustrate ideas from Section 2, connect it with basic functionality described in Section 3.4, and possibly later with some applications described in Section 4.

After logging in and selecting or creating a project, a user enters the interactive editor environment, which is shown in Figure 2. In the left panel, the user sees project organization by "queries". Users are free to specify queries according to their personal preferences. The project organization then may resemble traditional filesystem-like structure or not, i.e. same code slices may appear under different queries if the user wishes to. In the middle panel, the user sees the results of "organization queries". This, again, may resemble

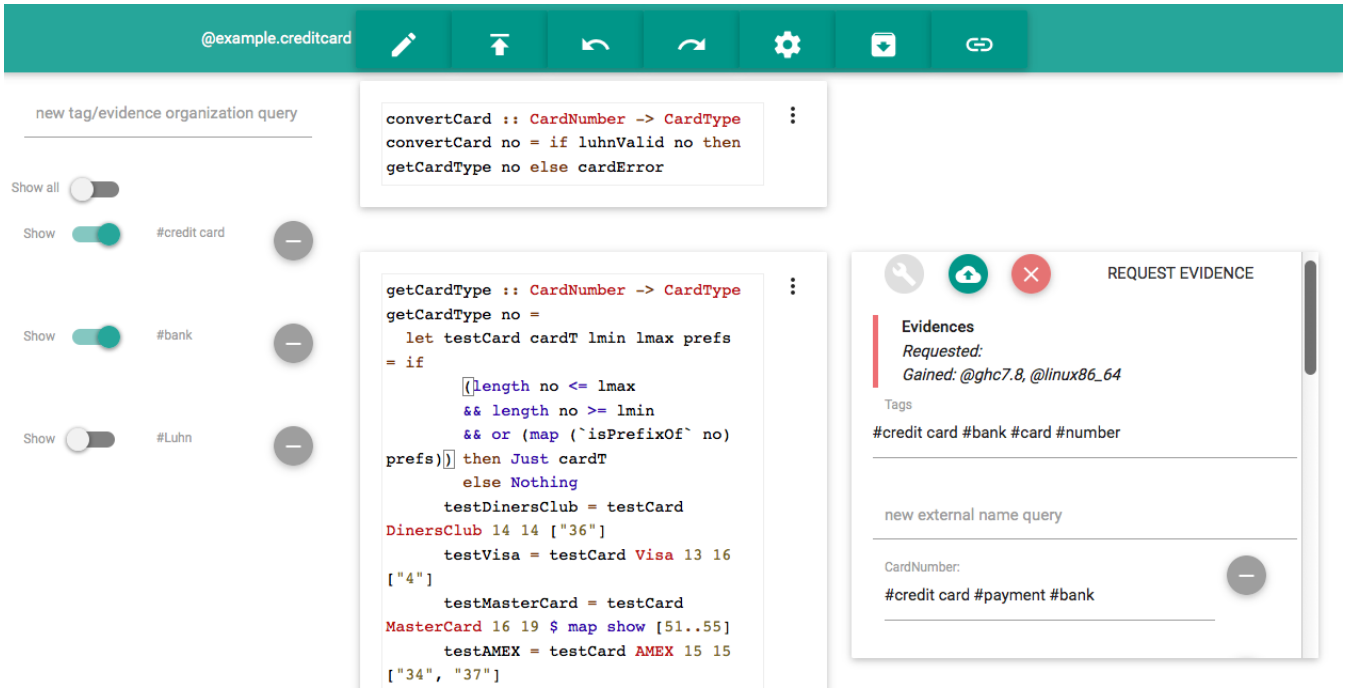

Figure 2. A screenshot of the experimental web-based programming environment: the left panel is used for query-based source repository navigation, the middle panel shows the resulting code fragments, and the right panel shows opened extra metadata information.

an opened text file. The difference is that the ordering of individual code fragments does not matter. The user can inspect code fragments and click to expand relevant metadata about them. The metadata appears in the right panel and the user can modify it.

In Figure 1, the editor is split into two parts. The editor backend communicates with the event log and specialized services. This part was written in Haskell using the Snap web framework. The editor frontend displays the user interface and communicates with the backend. The frontend was written in Elm and uses the Materialize framework and the CodeMirror editor. Network communication, i.e. querying and retrieval, happens via Socket.io. Local code is stored in HTML5 Web Storage.

4. Prospective Applications

In this section, we overview several examples of potential applications that can be built on top of **Haknam**. Implementation of these extensions follows the same guideline as the core implementation: they produce, filter and process certain messages.

Stable Code Vetting. The architecture described in Sections 2 and 3 is flexible and allows extensions beyond the originally envisioned scope (unlike many current PL environments). In this case, we consider a feature for automated processing of community vetting on stable code. Different organizations, communities or even individual projects have different definitions of code stability. We may define stability based on the time since last API changes, the number of stable external code slices depending on it and other features. Even though we may process these features, code maintainers with good reputation generally decide on when something becomes stable. Hence, for the purposes of demonstration, we would consider code to be stable if a certain threshold of whitelisted users marked it as such.

The implementation of this feature is fairly straightforward. We would add a new type of message that denotes an individual vote for code slice stability; and include this action in the editor. Then, we would create a service that collects and processes these votes from the event log. After reaching a certain threshold on votes, this service would trigger an action that represents stability. That could be in the form of adding certain evidence labels, such as publishing given code slices in a dedicated *stable* project repository. Unlike in conventional repositories, stable and development code coexist in the same environment. Creating stable distributions is just a matter of labeling code by a trusted party.

Collaborative Code Editing and Replay. In the prototype, we slice and store source code at a granularity of "checkpointed" compilation units. An alternative or complementing approach would be storing individual small incremental changes, such as UI events. This granularity would not be very useful for the basic functionality we described in Section 3.4, but other applications could make use of it. One immediate application is collaborative editing where, given

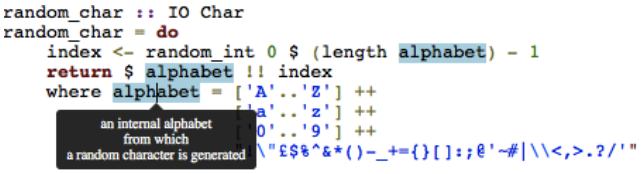

Figure 3. Comment metadata decoupled from source code: In this example, an explanation comment is attached to a local expression.

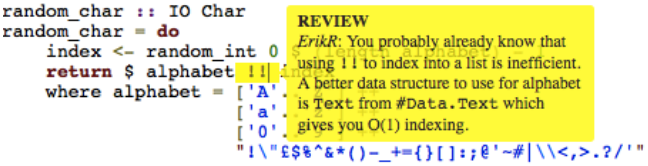

Figure 4. Comment metadata decoupled from source code: In this example, a different type of comment (code review) is used.

change information is annotated with its author, different programmers could share a coding session by listening on and contributing to the same stream (e.g. the same partition of the event log). Replaying edit changes could be another usage, e.g. for code reviewers to get an idea how code under review came about.

It would be impractical storing information at this granularity forever. In Section 6.1, we describe a mechanism that could solve this issue. Just like the current set of tags is an aggregation of tag addition messages, the final slice could be generated as an aggregation of all small incremental changes. The messages with small changes could expire after a certain time or action, e.g. after their resulting code has been marked as stable, and be compacted into the final code slice.

Version Control. All VCS functionality could be easily added. Commits are just metadata messages written to the event log after some changes. Forking and branching is done by reinserting messages from one project to a different project. The VCS functionality could go beyond its traditional form. For example, we could also fork individual code slices by dragging them from one project and dropping them in another one in the editor. Merge notifications can be shown immediately: 1) if a forked project updated some parts, it can be shown in the original project whether to synchronize those changes, 2) if an external dependency was updated and the most recent update still satisfies all evidence constraints and tags, the user can confirm to replace it. Pull requests could be implemented, like everything else, as messages written to the event log.

API Searching. One interesting area is API searching. Code searching services can utilize available code and all its metadata. They could combine code and metadata for ranking of results. From the core data, we can rank by the number of depending projects; or given a set of tags, we can see which ones are used most in the queries. Code searching services could also utilize the metadata from other applications. For example, using the information about project bookmarking, code searches can be personalized by ranking code from users' bookmarked projects higher.

Decoupled Enriched Code Comments. Finally, the uniform storage may enrich common functionality. Traditionally, source code is entangled with textual comments. In **Haknam**, we could decouple comments from code: we could store code comments as separately written messages with references to existing code slices. This allows greater flexibility in displaying code comments. For example, separate comments could be made by different users and refer to different segments. In the editor, one could highlight arbitrary parts of code slices and write comments about them. Unlike traditional textual comments embedded in source code, decoupled comments could be attached to a specific expression – an example of that is shown in Figure 3. The decoupled comments could be visualized as collapsable overlay bubbles.

Other possibility for enriched code comments is to store different types of comments separately. Comments may contain high level descriptions, example usage or detailed explanation. Each comment type can be visualized differently. For instance, comments with example usage would render code samples and clicking on a code sample would offer an option to copy it into an edited code slice. Another example is that code comments could capture conversations and code reviews. We show an example of that in Figure 4. All of this could be easily achieved through uniformly storing and processing metadata.

5. Related Work

System Software. Nix [13] or Guix [8] are OS-level language-agnostic package managers that aim at reproducibility. The idea from Fragnix [31] of hashing module dependencies is inspired by Nix, where cryptographic hashes capture all build dependencies and configurations of packages. TagFS [4] or hFAD [32] are filesystems that allow more flexible file organization through attaching labels to files and querying using these labels. Novel programming systems, such as Tunes OS [34] or Awelon Blue [3], track changes and separate human-readable names from linking. GitTorrent [2] project aims at creating a decentralized network of Git repositories with additional information stored in a distributed hash table. Since Git implements similar semantics to what we described in Section 2.4, GitTorrent could potentially serve as an alternative implementation of the storage layer of **Haknam**. Eidetic computer systems [12] are the ones that can recall any past state that existed in them. This is one of the motivations behind **Haknam**'s storage layer which could partially be considered eidetic. In **Haknam**'s

prototype, we assumed services as trusted black boxes. One potential extension would be to require logging their internal state changes as well.

Experimental Languages. The idea of having a single expression as a unit of compilation is explored in Annah [17]. In Annah, all expressions are encoded in calculus of constructions and stored in a textual representation on a web server or a local filesystem. After reading an input expression, Annah's compiler downloads and parses external expressions, and super-optimizes the whole program. It, however, does not provide any notion of versioning or additional metadata. Nava [29] used semantic queries to lookup functionality. In **Haknam**, tags do not refer to any concepts, hence resemble folksonomies rather than the semantic web. Logic Metaprogramming [11] stores fragmented program representations as predicates in a deductive database and uses them for flexible and declarative form of metaprogramming. **Haknam** slices and stores source code at a granularity of compilation units, but as discussed in Section 4, finer granularity could benefit some applications – declarative metaprogramming would be an interesting direction to explore.

Haskell Tools. Annex [21] is a dependency manager for Haskell in development. It abandons semantic versioning and manages dependencies through querying a deductive database about program artifacts. Skete [20] aims to represent software package databases as Git repositories. Programmatica Project [19] integrated Haskell code development and verification. Some of tools developed in the context of Programmatica are related to **Haknam**, in particular, its ideas of program slicing, tracking dependencies on a definition-level and evidence management. The differences are that Programmatica tools focused on local code development, there was no notion of shared code and versioning, and aimed at verification. "Evidence management", hence, dealt with property testing and theorem proving. In **Haknam**, evidence can be anything, including automated style checking or performance profiling.

Monolithic Codebases. Many large software companies, such as Google [26] or Facebook [14], published reports about their positive experience when using monolithic version control repositories. Despite common beliefs, monolithic repositories were chosen for simplified organization, dependency management, cross-project changes, and the ease for developing tools. This motivation is shared by our work. The difference is that **Haknam** does not impose any global repository semantics beyond writing and reading messages from the event log. Monorepos assume a consensus on the filesystem structure, which is achievable within a single organization, but not outside that. In **Haknam**, different organizations may publish their code and impose their workflows within their projects, while all their code can be accessed like any other code in other projects. For example, one workflow difference may be in handling changes. Automated scripts for some projects may watch for changes in transitive dependencies, actively request evidence and pull in latest results; some may only watch for direct dependencies; others may update dependencies manually, but automatically request all evidence that their dependants need.

CASE. Finally, the basic prototype functionality and experimental editor (Sections 3.4 and 3.5) as well as all prospective applications (Section 4) are examples of Computer-aided Software Engineering (CASE) [5] tools. **Haknam** can thus be seen as a CASE environment – the difference is that traditional CASE environments, even in a distributed setting [24], generally operate within a single organization. One related extensive research branch of automated software engineering is tool integration [1]. Processing of multiple sources of data plays a great role in improving code searching (e.g. SEXTANT [30]) where data integration is an important issue [10]. The idea of different tools needing a different access to related data was first explored in Garlan's *views* [16]. Related approaches appeared in, for instance, *virtual source files* in Stellation [6], *virtual files* in the Desert software engineering environment [28], or *intents* in intentional programming [33]. The idea is to separate usage of source code from its storage representation. In **Haknam**, this motivation is shared to some extent, especially in terms of storing precise dependencies, but there is no primary aim of recreating files.

6. Discussion & Future Directions

In this section, we first discuss how to tackle some issues that were not handled in our prototype, but would be important considerations in a real system implementation. We then outline possible future research directions.

6.1 Scalability

The event log systems, such as Apache Kafka [22] or EventStore [15], can process tens of billions unique message writes per day. In terms of daily workload, they proved to be horizontally scalable. This, however, does not assume all messages can be stored forever. If we stored every single message, we would eventually run out of storage space. Even if we assumed an infinite storage space, replaying the complete log would take longer and longer. The standard way to solve this problem is using *log compaction*. It means that we throw away some obsolete records and store their more recent updates. By doing so, we still have a complete backup and can exactly replay more recent states, but lose the ability to replay *all* states. For example, we can get all tags for a code slice in a certain project, but cannot replay all individual updates of tags. There are other possible optimizations to prevent "code and metadata pollution". For instance, we can delete and archive old code slices (and their related metadata) that do not belong to any project and are not imported by any other slices that belong to some projects. After this

"garbage collection", we will not have a complete log, but a log of everything that all current shared code depends on.

6.2 Backwards Compatibility

Existing Metadata and Environments. There are two directions when considering existing metadata and environments for backwards compatibility: 1) how to extract and integrate metadata from various sources into **Haknam**, 2) how to export data from **Haknam** into existing environments. The first direction is challenging due to heterogeneous storage, coarse-grained dependencies, and various conventions. The second direction should work by having an "export" service that would filter certain events from the event log, transform them to the required format and store the result in a desired location.

Existing Code. There are three directions when considering existing code for backwards compatibility: 1) how to use existing Haskell code in **Haknam**, 2) how to use code from **Haknam** in existing Haskell code, 3) how to interface with non-Haskell code. For 1), we can reuse Fragnix's feature that allows slicing an existing Haskell module into an environment and code slices. Thus, we can automate the process by having a service that reads updates from an old Haskell repository, slices all modules, writes messages with resulting slices and tags generated from module names (other metadata may be challenging). For 2), code slices in **Haknam** are still regular Haskell modules, so they can be exported as regular Haskell source files. One desirable thing, however, may be to have a human-readable interface in a Haskell source file. For that, we can create Fragnix's environment whose name could be generated from user's selected tags. For 3), Haskell's FFI can be used and external non-Haskell code can be bundled in a message with a reproducible package definition, e.g. a Nix [13] or Guix [8] expression, and built by special compilation services.

6.3 Privacy & Security

As described in Section 3, we gain a lot from having a single integrated source of truth, but we would need to consider more issues for the real system implementation. The first issue is the notion of trust for the external services. For example, how do we trust successful compilation evidence and relevant binary artifact caches? One possibility is running our own binary artifact cache and make it recompile and verify our code dependencies on-demand. Other possibility is publishing public keys, signing messages and including checksums. The second issue is how to restrict access to shared code. For example, a company has some proprietary code that cannot be made open source. We can resolve such issue either by running a private event log, or by encrypting sensitive messages. The final, perhaps hypothetical, issue is with the infrastructure ownership where the solution could be placing the event log on a content-addressable distributed storage with decentralized ownership (e.g. GitTorrent [2]).

6.4 Future Directions

We conclude this section with several open questions, outlying areas of potential future research directions.

Other Programming Languages. We focused on the standard Haskell, which eased our task in two ways: 1) it is a purely functional language, 2) it has a relatively simple module system. For example, simple information hiding can be achieved through using let-bindings for private definitions. Given a code fragment, it can generate different code slices at different times or with different queries for external names. "Separate compilation" as in separete typechecking of code fragments is not possible, but code slices can be separately pre-compiled as normal Haskell modules. This could be utilized for efficiency reasons – for development and testing, "stable" strongly-connected dependencies could be separetely compiled and used; for deployment, global compilation may be preferred. The question is how our work could extend to other languages. The core issue is how to create code slices when there are mutable global state interactions and when the module system needs a complex access-level name resolution or does metaprogramming. Dependencies could potentially be tracked at a coarser level. Depending on a language, it may not be clear what a minimal compilation unit should be and automated slicing may only yield approximate results. In OO languages, a natural unit for slices could be well-encapsulated classes, traits or mixins.

First-class Language and More Expressive Queries. We reused an existing text file-centric programming language and preserved its semantics. Instead of embedding existing languages, one alternative route would be to design and implement a new programming language with **Haknam** features in mind from scratch. Instead of plain text files, the primary abstraction would be the event log. In such language, there may not be any notion of module or identifier names and the lookup may need to be more expressive (e.g. Datalog queries). The ability to reason about name lookups and "time" can feed back into understanding how to extend to other languages – for example, given name queries and fragments as a form of light-weight metaprogramming, it can help with understanding what sort of separate type checking is possible. Other directions may include safe hot code swapping (re-evaluating lookups at runtime) or utilizing different granularity of compilation.

Applications and Evaluation. We listed examples of different applications in Section 4. There could potentially be other applications extending the functionality beyond what we described: a binary cache of most dependent on code slices, project bookmarking, etc. Different related questions may be worth investigating. One question is what this modular setting would imply for large open-source communities. For example, if one tried to convert some existing historical metadata related to API usage, it could be examined whether original interfaces are used as envisioned in their

static structure, or different structures emerged. Other question is, for example, how the fragmented code organization affects compilation times: on one hand the embedded compiler processes only relevant source code fragments, on the other hand, the linker gets more burden. It also depends on situations whether we consider compilation from scratch, or recompilation after small changes during development.

7. Conclusion

Programming tool development is limited by its environment. Despite the wide-spread code reuse and sharing over the internet, some of our conventions in programming environments remain in the "localized" pre-internet era. It is, hence, worth exploring a modular system architecture that is better suited for the today's programming environment needs. Our preliminary work serves as a foundation and opens up different prospective applications that could validate our hypothesis about its extensible design.

In turn, we expect it to inspire further research on modular architectures for code and metadata sharing, and on software development in this setting. This area spans from practical issues, such as how we can retrofit other existing programming languages with concepts from this work, to more exploratory matters, such as theoretical foundations for a language that embodies these concepts in its semantics.

References

[1] F. Asplund and M. Törngren. The discourse on tool integration beyond technology, a literature survey. *Journal of Systems and Software*, 106 (C):117–131, August 2015.

[2] C. Ball. GitTorrent, 2015. URL `https://github.com/cjb/GitTorrent`.

[3] D. Barbour. Awelon Blue, 2014. URL `https://awelonblue.wordpress.com/`.

[4] S. Bloehdorn, O. Görlitz, S. Schenk, M. Völkel, and F. I. Karlsruhe. TagFS – Tag Semantics for Hierarchical File Systems. In *I-KNOW 06: Proceedings of the Sixth International Conference on Knowledge Management*, pages 6–8, 2006.

[5] A. F. Case. Computer-aided Software Engineering (CASE): Technology for Improving Software Development Productivity. *SIGMIS Database*, 17(1):35–43, September 1985.

[6] M. C. Chu-Carroll, J. Wright, and D. Shields. Supporting aggregation in fine grained software configuration management. *SIGSOFT Software Engineering Notes*, 27(6):99–108, November 2002.

[7] Continuum Analytics. Anaconda Python Distribution, 2012. URL `http://continuum.io/downloads`.

[8] L. Courtès. Functional Package Management with Guix. In *European Lisp Symposium*, June 2013.

[9] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Information and Computation*, pages 200–206, 1985.

[10] B. de Alwis and G. C. Murphy. Answering Conceptual Queries with Ferret. In *Proceedings of the Thirtieth International Conference on Software Engineering*, ICSE '08, pages 21–30. ACM, 2008.

[11] K. De Volder. *Type-oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 1998.

[12] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen. Eidetic Systems. In *Proceedings of the Eleventh USENIX conference on Operating Systems Design and Implementation*, pages 525–540. USENIX Association, October 2014.

[13] E. Dolstra. *The Purely Functional Software Deployment Model*. PhD thesis, Faculty of Science, Utrecht University, 2006.

[14] G. Durham and S. Agarwal. Scaling Mercurial at Facebook, 2014. URL `https://code.facebook.com/posts/218678814984400/scaling-mercurial-at-facebook/`.

[15] Event Store LLP. Event Store, 2012. URL `https://geteventstore.com/`.

[16] D. Garlan. Views for tools in integrated environments. In *An International Workshop on Advanced Programming Environments*, pages 314–343. Springer-Verlag, 1986.

[17] G. Gonzalez. Annah, 2015. URL `https://github.com/Gabriel439/Haskell-Annah-Library`.

[18] Gradle Inc. Gradle, 2007. URL `http://gradle.org/`.

[19] T. Hallgren, J. Hook, M. P. Jones, and R. B. Kieburtz. An overview of the programatica toolset. In *High Confidence Software and Systems Conference (HCSS04)*, 2004.

[20] A. Heller and D. Scies. Skete, 2015. URL `http://code.xkrd.net/skete/skete`.

[21] M. Hibberd. Annex: A Fact Based Dependency System, 2014. URL `http://mth.io/posts/annex/`.

[22] J. Kreps, N. Narkhede, and J. Rao. Kafka: A Distributed Messaging System for Log Processing. In *Proceedings of ACM SIGMOD Workshop on Networking Meets Databases (NetDB'11)*. ACM, 2011.

[23] F. Lancaster and E. Fayen. *Information Retrieval: On-line*. Information Sciences Series. Melville Pub. Co., 1973.

[24] D. B. Leblang and R. P. Chase, Jr. Computer-aided software engineering in a distributed workstation environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 104–112. ACM, 1984.

[25] M. S. Mahoney. Finding a history for software engineering. *Annals of the History of Computing*, 26(1):8–19, January 2004.

[26] C. Metz. Google is 2 billion lines of code – and it's all in one place, 2015. URL `http://www.wired.com/2015/09/google-2-billion-lines-codeand-one-place/`.

[27] S. Raemaekers, A. van Deursen, and J. Visser. Semantic Versioning versus Breaking Changes: A Study of the Maven Repository. In *2014 IEEE Fourteenth International Working Conference on Source Code Analysis and Manipulation*, pages 215–224. IEEE, September 2014.

[28] S. P. Reiss. The Desert Environment. *ACM Transactions on Software Engineering and Methodology*, 8(4):297–342, October 1999.

[29] H. Samimi, C. Deaton, Y. Ohshima, A. Warth, and T. Millstein. Call by Meaning. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software - Onward! '14*, pages 11–28. ACM, October 2014.

[30] T. Schafer, M. Eichberg, M. Haupt, and M. Mezini. The SEXTANT Software Exploration Tool. *IEEE Transactions on Software Engineering*, 32(9):753–768, September 2006.

[31] P. Schuster. Fragnix, 2014. URL `https://github.com/phischu/fragnix`.

[32] M. Seltzer and N. Murphy. Hierarchical file systems are dead. In *Proceedings of the Twelfth Conference on Hot Topics in Operating Systems*, HotOS'09. USENIX Association, 2009.

[33] C. Simonyi. The death of computer languages, the birth of intentional programming. In *NATO Science Committee Conference*, pages 398–399, 1995.

[34] TUNES Project. TUNES OS/Language Project, 1992. URL `http://tunes.org/`.