On the Efficacy of Keyword Searches to Find Meaningful Architectural Knowledge in Open-Source Software Mailing Lists

Bachelor Thesis for Computing Science

Andrew Lalis andrewlalisofficial@gmail.com

Supervised by **Dr. Mohamed Soliman** m.a.m.soliman@rug.nl

July 19, 2022

university of groningen / faculty of science and engineering

Abstract

Google and other web-based search engines give users immediate access to a plethora of knowledge. Mailing lists however, go unnoticed, and in searching for knowledge about software architecture for making important design decisions, they present an ideal source of information. In this study, we prepare the tools necessary to fetch, process, and categorize the architectural design decisions found in the mailing lists of several open-source projects from the Apache Software Foundation. An analysis will present information on the types of decisions, their prevalence, and the patterns these decisions form through a thread of emails. Several keyword-based search queries will be evaluated against the dataset to determine their effectiveness at retrieving the most relevant information.

Contents

Introduction

In this paper, we'll explore the efficacy of using targeted keyword search queries to find architectural knowledge in mailing lists for open-source software projects. More simply put, we'll build tools, collect data, and analyze that data to qualitatively determine how effective certain keyword-based search queries are at finding useful information in large sets of emails, sent in mailing lists for developers to communicate about large open-source software projects.

Software Architecture

Behind every software system you interact with, including the one used to read this paper, is a set of elements, relationships, and rationale that define the software's architecture. While on the surface, you see the *implementation* of the software, the *architecture* is concerned with what elements, their interactions in order to provide a framework for satisfying the requirements of the system [17]. Continuing our example, the software you're using to read this paper probably uses an architecture that includes elements like a PDF renderer, file reader, and a controller to manage your button clicks and mouse scrolling. Collectively, this set of components, design decisions, rationale, assumptions, and context together are defined as the *architectural knowledge* of the system[4].

In Perry's 1992 paper, "Foundations for the Study of Software Architecture", he posits that "we have not yet arrived at the stage where we have a standard set of architectural styles with their accompanying design elements and formal arrangements," and that each system is "a new architecture" [17]. While we are perhaps nearer now, 30 years later, to that mythical stage of architectural standardization, indeed many new systems are genuinely new architectures, with new elements or entirely novel approaches to communicating between components, where engineers must consult both their acquired skills and the collective knowledge of their peers, to make a best-effort to build a system to satisfy their requirements.

Despite the plethora of resources available to the modern software engineer, we still often see *architectural knowledge vaporization*[13] because of undocumented decisions engineers took when designing the architecture of a system, or because it's impossible or practically infeasible to extract useful information. This dissipation of knowledge is a function of time and architecture size, and can lead to some serious problems that have already established themselves as hallmarks of poor software design in the industry, such as an increasing cost to improve or upgrade a system [13, 17], and a lack of reusability [13], and an increased cost of maintenance [16] even when no new features are added. These costs can, and often do, reach a point at which it is simply more effective to start over, abandoning completely any established work on an architecture. Through the years, this evolved from reprogramming low-level batch programs for early computing installations[16], to today's distributed systems. Succinctly, a loss of architectural knowledge, by conjecture, leads to a loss of value and a loss of efficiency, and this paper aims to add additional tools to our collective defense against such regression.

Searching

It is important to explore different avenues for acquiring knowledge, especially as the body of information grows exponentially with time. It is difficult for developers and software architects to make informed decisions about their own projects, because the source of their knowledge is distributed in a variety of disparate sources. If we can reliably glean information about software architecture and the successful (and unsuccessful) decisions that other field experts have made, we can make this knowledge more accessible for all.

Most people are familiar with "googling" to find solutions to their questions, but Soliman et al. has shown that using web search engines is not effective for mining knowledge from mailing lists[19]. This probably is a symptom of the fact that most search engines do not index emails sent in mailing lists (as it generally doesn't make sense to do so), and that search engines can't provide a friendly user experience for navigating results that would be shown from mailing lists.

Therefore, we'll need to index and search over emails independently. Of course, this detaches the research from what the majority of people are doing in their everyday lives, but presents more opportunities for automated architectural knowledge scraping, and as Bhat et al. has shown, an effective machine learning model is well within the realm of possibility[3].

Defining Types of Design Decisions

Before we can begin searching for architectural knowledge in earnest, we must first exhaustively define the types of distinct architectural design decisions that will be categorized.

We start by evaluating Kruchten's original set of design decisions in section 2.1 of "Building Up and Reasoning About Architectural Knowledge"[15], which lists four main kinds of decisions: **existence**, **ban**, **property**, and **executive**.

These decisions were conceived in a domain that must consider architectural knowledge from a large variety of sources, and so we also consult den Boon[4] and Faroghi[7] for more information that's specific to open-source issue/ticket boards and mailing lists. From their more specific lists, as well as some cursory categorization attempts, the following architectural design decisions were adopted.

Existence

Existence decisions are those that are concerned with the creation of new elements of a system's conceptual or implementation architecture, or the creation of behavioral connections between existing components. Generally, existence decisions manifest themselves in discussions about whether to add a new component to a system, or to link components together.

Structural decisions are a subset of existence decisions that focus solely on the creation or modification of components of an architecture; most often the addition of new components to a system.

On the other hand, *behavioral* existence decisions are concerned with, well, the behavior of the components of a system, namely how multiple components in a system interact with each other.

For example,

- The system shall incorporate an additional process manager to orchestrate other high-level tasks.
- The system shall send periodic ”heartbeat” packets to connected subsystems.

Property

This category of design decision is taken almost verbatim from Kruchten's originally proposed list of architectural design decisions[15], and is defined as one which states an enduring, overarching trait or quality of the system. This is not so much related to any particular element or behavior because they often necessarily affect many components with cross-cutting concerns.

For example,

- The system shall be able to process at least 10,000 incoming connections concurrently.
- The system shall allow for flexible third-party plugin integration.

Process

Again, this design decision is taken without much need for revision, from previous papers on the subject. A process decision is concerned less with the architecture itself, and more the internal development process. This includes things like software quality and testing strategies, code review procedures, and enhancement proposal protocols.

For example,

- We as developers should define a strict suite of tests that must pass in order for any new release to be made available to the public.
- We should create a formal procedure for submitting patch requests.

Technology

These decisions are made regarding the comparison, evaluation, and choice of third-party technologies for use in or with an architecture. This could be as simple as discussing a choice of programming language, to choosing a set of supported Open-ID Connect API providers for a user authentication micro-service. Generally, an email shall be considered to contain a technology design ”decision”, even if no discrete decision is made in that email. This is because a large part of technology discussions focuses on evaluating the properties of various alternative solutions, and such discussions have been found in practice to end with a rather simple, abrupt choice based on a vote.

For example,

- We would like to use the *D* programming language to write startup scripts for the system, because of its flexible syntax and high performance[5].
- Our development team should make a choice between using Jira or GitHub Issues for our ticket board. Here are their pros and cons: ...

Research Questions

The main research question that this paper attempts to answer is summarized in the following question:

What architectural knowledge exists in open-source software development mailing lists?

This broad main question can be further segmented into a series of more specific questions on the nature of the knowledge that exists in mailing lists:

1. What types of architectural design decisions are present in mailing lists?
2. How often do architectural design decisions appear in mailing lists?
3. What patterns exist in the order in which architectural design decisions appear? Furthermore, why do such patterns, if any, exist?
4. How effective is using Apache Lucene to index and search for architectural design decisions in mailing lists?
5. How can we use data gathered in this research to improve our search queries?

Related Work

In the field of architectural knowledge acquisition (granted, a rather niche field), the work of this research builds directly on the results obtained in den Boon's paper that explored the effectiveness of using the Apache Lucene search engine for finding this knowledge[4]. This research found that Lucene can indeed be used for finding knowledge effectively, and also provided a basis for the four queries that this study uses for searching. It also concluded that, when using these queries to search through both issue/ticket boards and mailing lists, that mailing lists were more suitable to finding "macro architectural design decisions".

The primary supervisor of this paper, Mohamed Soliman, also prepared a study in 2021 that evaluated the effectiveness of common web search engines (i.e. Google) to find architectural knowledge, and from this paper we can learn that there are some key drawbacks to such a "lazy" approach to searching for architectural knowledge using the world's most ubiquitous search engine: design decisions are underrepresented, issue trackers and mailing lists are not indexed by search engines, and that most actual open-source software architecture on websites like GitHub is not easily accessed. Furthermore, we see from this paper that while Google *does* make it easy to compare alternative solutions for architectural problems, with blogs and tutorials being the most common result, finding information on instantiating *new architectures* is not as easily accessible. These drawbacks provide motivation to search for alternative, more targeted searches that go beyond Google's general purpose indexing patterns[19].

Xiong et al. worked in 2018 on analyzing the content of open-source Hibernate mailing lists to search for "assumptions regarding requirements, environment, design decisions, etc." and found that the Hibernate developer mailing list contained 832 such relevant results from a sample size of 9006 posts. This provides an indication that developer mailing lists can be a valid source of knowledge about architectural design decisions[21]. As stated in that paper, this study attempts to address the need to "explore the feasibility of this study in other OSS projects". Similar to this, work done by Ding et al. shows definitively that discussions in OSS developer mailing lists *do* lead to measurable changes in the software's architecture, which provides further indication that mailing lists have a high potential for finding legitimate, practical, and usable architectural knowledge[6].

Kruchten's paper "Building UP and Reasoning About Architectural Knowledge" provided a basis for the *types* of architectural knowledge that this paper will discuss, particularly his discussion on the "Ontology of Design Decisions"[15]. This information is combined with the research of den Boon[4] and Faroghi[7] to arrive at the formalized set of design decisions that is discussed in the Design Decisions section of the methodology discussion.

With regards to the development of the practical tools used for this study, Tang's comparative study of existing architectural management tools[20] was a useful indicator of what the current (well, as of 2008) functionalities and expectations are for such tools. However, the different solutions examined under that paper focused mainly on larger-scale commercial ventures, which this study is not. Those tools therefore had a plethora of additional requirements, like privileged access to architectural knowledge and concurrent user content locking. These features, while they can be useful when deployed in a large-scale setting, are not necessary for a simpler passive study of existing knowledge in mailing lists.

To extend the discussion on tools for extracting knowledge, Bhat et al. managed to produce a machine learning model for the identification and classification of architectural design decisions in OSS issues, and managed to achieve 91.29% accuracy in detection, and 82.79% accuracy in classification[3]. Granted, this model applies only to issues and not mailing lists, and mailing lists are generally more verbose, so the applicability of this study's findings to mailing list searching are rather limited. In any case, our study may help to provide data and guidance to future work on developing a machine learning model for the classification of emails, similar to Bhat's study.

Tools

To facilitate the research described in this document, a suite of tools was developed to aid in the fetching, processing, categorization, and analysis of emails from various open-source mailing lists. This section contains a detailed overview of each of these tools, and a high-level overview of the system architecture and how the entire suite can be used from start to finish.

The entire collection of tools and their source code is available online at the Architectural Knowledge Analysis GitHub organization. Each of the tools discussed in this section is organized under a separate repository. In general, any projects written in *Java* require the user to have version 17 or greater installed to build the project or run any executables/apps produced by it. You can get the latest OpenJDK release here. Any projects written in *D* should use a fairly recent version of D. The DMD compiler, version 2.099.1, was used to build and run all D projects/scripts. You can get the latest D compiler here.

Note that, especially for D projects, development was done in a Linux environment, and no serious attempt was made to provide additional compatibility and support for the Windows or Mac operating systems. Any compatibility is simply a consequence of the cross-platform nature of the technologies used. File path separators, process management, permissions, and other functionalities are handled differently in other operating systems, thus it is encouraged that you use a Linux-based OS to make use of these tools.

Overview

While each of the tools described in this section exist as a standalone project that can be incorporated into any third-party application, the majority of these focus on providing auxiliary services to the Email Dataset Browser application.

The general workflow that these tools enable, is described as follows:

1. Raw `mbox` files (files containing many emails) are downloaded from a mailing list archive's API.
2. The mbox files are parsed to extract the individual emails and their metadata, and this is used to build a relational database and Lucene search index.
3. A researcher categorizes the emails in the dataset according to a set of pre-defined criteria. In this case, we use the architectural design decisions covered in the introduction.
4. The categorized emails and Lucene search results are exported from the dataset to a JSON format.
5. Analysis is performed on the data and visualizations are generated.

Email Downloader

The email downloader component helps us to solve the first step of the workflow: fetching `mbox` files from a mailing list archive. It does this by introducing the `MailingListFetcher` that provides a common interface for fetching mailing list archives that is designed to be compatible with most online mbox file repositories. It simply defines a single `download` method that downloads a mailing list (identified by its domain and list name) and saves the mbox files to a given directory. The caller provides a start and end timestamp to limit the download to only those archives pertaining to emails sent within the specified timeframe, and finally a message consumer that can be used for downloader to report messages as it performs the download.

```
CompletableFuture<Collection<Path>> download(
  Path dir,
  String domain,
  String listName,
  ZonedDateTime start,
  ZonedDateTime end,
  Consumer<String> messageConsumer
);
```

The download happens asynchronously, and the returned `CompletableFuture` is completed when the download is complete.

Because this research will focus on mailing lists from the Apache Software Foundation, we provide an `ApacheMailingListFetcher` implementation that implements the above interface. It uses Apache's (undocumented) API endpoint at https://lists.apache.org/api/mbox.lua to download emails via HTTP, using a rate-limited strategy with a delay of 1 second, and a short-circuiting behavior where the fetcher will give up after 10 consecutive months of no emails.

For example, we could request the following from the API:

```
curl https://lists.apache.org/api/mbox.lua?domain=nemo.apache.org&list=dev&d=2007-12
```

Here, we fetch all emails (in a single mbox file format) from the *dev* list in the Apache Nemo project from the month of December, in 2007. The API will respond with a file download, which our library saves to the system's drive (in the desired directory).

CLI

Besides the programmatic interface that the email downloader provides, it also provides a command-line interface when running the project's JAR file as an executable. The JAR file can be obtained from the project's releases page, or built from source using maven:

```
mvn clean package assembly:single
```

The CLI provides access to, at the time of writing, just the Apache implementation for the mailing list fetcher, and essentially allows one to invoke the `download` method from the command-line. Its detailed usage information is provided below.

```
Usage: EmailDownloader [-hV] -d=<domainName> [--from=<fromDate>] -l=<listName>
                       [-o=<outputDir>] [--until=<untilDate>]
  -d=<domainName>            The domain to download from. For example, "hadoop.
                               apache.org"
  --from=<fromDate>          Earliest period to download from, inclusive.
                               Formatted as yyyy-mm.
  -h, --help                 Show this help message and exit.
  -l=<listName>              Name of the mailing list to download from.
  -o, --output=<outputDir>   Directory to place downloaded files in. Will
                               create it if it doesn't exist.
  --until=<untilDate>        Latest period to download from, inclusive.
                               Formatted as yyyy-mm.
-V, --version                Print version information and exit.
```

This information is also available via the standard `--help` flag.

MBox Parser

With the raw mbox files obtained by the email downloader, it's not yet possible to do anything productive until we extract the individual emails and their data. To do this, we introduce the MBox Parser library. This library provides a programmatic method for extracting and pre-processing emails from all mbox files in one or more directories or files.

The main interface is the `MBoxParser` class, from which objects can be instantiated with a provided `EmailHandler` implementation that executes a callback for each email parsed from the set of mbox files. Specifically, the format for emails obtained by the parser is as follows:

```java
public class Email {
  public String messageId;
  public String inReplyTo;
  public String sentFrom;
  public String subject;
  public ZonedDateTime date;
  public String mimeType;
  public String charset;
  public String transferEncoding;
  public byte[] body;
}
```

Additionally, a utility method is provided to read the body of the email as a simple string. Internally, the parser makes use of Apache's Mime4j library, and a custom email content handler that performs some extra processing on email dates and message ids to improve uniformity:

- We expect dates in the RFC-1123 format. For example, `Sun, 21 Oct 2018 12:16:24 GMT`, and if we get something slightly different, attempts are made to transform it into an acceptable format.
- We strip the angle brackets (< and >) from the email's unique message id. These brackets exist on every email, and do not contribute to the uniqueness of the id. Stripping them makes the message ids more readable as plain text when embedded in HTML or any XML, and makes searching over them easier.

The Mbox Parser provides users with the ability to provide their own filter functions to further refine the results, but this will be discussed in more detail when explaining the Email Indexer component. No command-line interface is provided for this library; it is only available as a programmatic Java library, meant to be used in concert with the email downloader and any other programs that consume individual emails.

Email Indexer

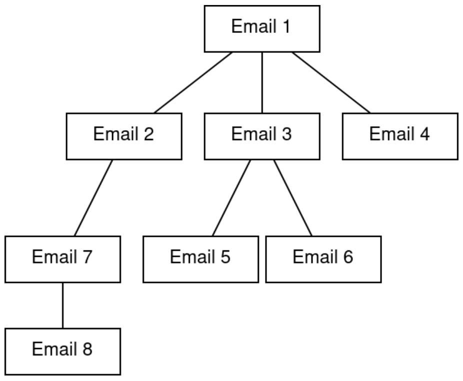
While the downloader and parser can be thought of as auxiliary to the goals of this research, the Email Indexer project forms the core of the email dataset's structure, and by implication, the research in this paper. The email indexer is a Java library that offers a programmatic interface for creating, managing, and exporting categorized email data.

Each dataset produced by the email indexer exists nominally as a directory, or as a ZIP archive of a directory. Each dataset directory contains the following components:

1. An embedded relational H2 database file named `database.mv.db`, which stores all emails, tags, and other key information for the dataset. A schema diagram is provided in this section, and the full schema definition can be found in the appendix.
2. An `index` directory in which the indexes generated by Apache Lucene are stored.
3. A `metadata.properties` file that holds some meta information about the dataset. As of the time of writing, the only metadata stored in this file is the dataset version, which, again at the time of writing, is version 2.

Email Thread Structure

Emails have a self-referencing *many-to-one* relationship defined by their "In-Reply-To" field. It is important to understand the consequences of this structure, and how it impacts the dynamics of processing data. In any information system, a self-referencing, many-to-one relationship that an entity participates in, can be interpreted as a *tree* data structure.



Each email can be thought of as a node in the tree, where each reply to that email forms a branch. There is no limit to how wide or deep the tree can become, because there is no universal hard limit on the number of replies to an email, nor on how many replies may be sent in a thread.

In previous research by den Boon, this natural tree structure of email threads was artificially flattened into a sequential list, presumably to facilitate more efficient methods of working with the data[4]. However, in this research, the tree structure of the emails is preserved in the dataset, because it more accurately depicts how real users would navigate a set of emails using any number of existing programs, and therefore can provide a better model for determining the effectiveness of searches. This choice does come with some drawbacks, however.

1. Because emails form a tree structure, navigating the dataset is more computationally intensive, and more complex for a user, than if one could simply traverse a flat list.
2. Analysis is more complex, due to the fact that most algorithms will need to incorporate tree traversal, either iteratively or recursively.
3. Relational databases generally do not offer facilities for recursive fetching, which can lead to performance losses when processing large sets of emails.

Despite the drawbacks, it is important to analyze and categorize emails in a format that does not lose the latent structural information contained in the email thread's tree itself. Furthermore, for this particular research, the volume of data and methods of analysis do not present any difficulties in terms of computational performance.

Database Schema

The relational schema is the core of the email dataset, and defines how we operate on and organize the data. It is purposefully designed to be as simple as possible, to make it easy to use the dataset in a variety of applications and environments.

The schema consists of three entities: *emails*, *tags*, and *mutations* (although mutations are not strictly required).

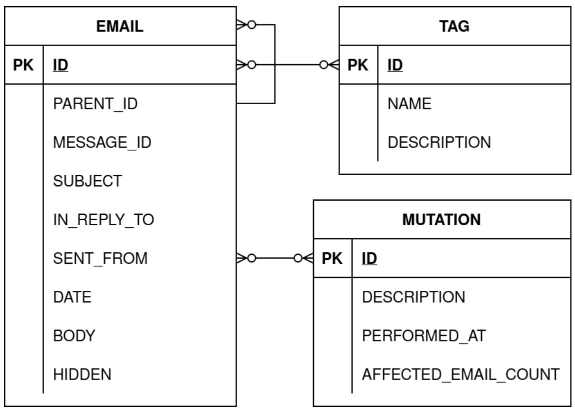


Figure 1: A diagrammatic representation of the email dataset's relational schema.

Emails are, as you'd probably expect, the entity that represents a single email message, sent by someone, in a mailing list. While emails *do* have their own unique `MESSAGE_ID` attribute[18], they are identified by an 8-byte integer primary key. This reduces the dataset size, improves search performance due to the monotonic nature of incrementing primary keys, and allows for easier referencing of emails by their id, instead of a long, randomly-generated message id.

Also of note is an email's `PARENT_ID`, which is a self-referencing foreign key that references another email entity by its id. This foreign key is the manifestation of the *many-to-one* relationship between an email and its parent, where many "child" emails may refer to one parent email. The parent id is derived, when processing emails, from the email's "In-Reply-To" field. According to RFC 5322, the in-reply-to field, if present, must reference another email's message-id, and this format of replies forms the basis of an email thread's structure[18].

Tags are entities which can be attached to any number of emails, to offer a method of categorizing emails. Originally in version 1 of the dataset format, tags were not entities but simple texts that were attached to each email, but it was discovered during the development of the tools for this research that there are several benefits to having tags as their own entities: the ability to edit a tag's name, provide a description, and reduce the total dataset size by referencing an 4-byte integer id instead of keeping a copy of a string for each tag applied to an email.

There exists a *many-to-many* relationship between **emails** and **tags**, which one could verbalize as, "A tag may be applied to many emails, and an email may have many tags applied to it."

Mutations are entities that record some change to the dataset which is used as a historical record of how a the emails in a dataset are filtered or modified. Mutations are used at the discretion of any third-party application which interfaces with the EmailIndexer API, and are not required to be used.

There exists a *many-to-many* relationship between **mutations** and **emails**, which one could verbalize as, "A mutation may involve many emails, and an email may be involved in many mutations." It is not required that a mutation be linked to any emails.

Indexing with Lucene

Alongside the relational database resides the `index` directory, as mentioned at the start of this section. This contains the indexes produced by the Apache Lucene indexing library when we build an index using the emails in the database. This index is then used for executing query searches over the database by users.

For each email in the dataset, we index the following fields as non-stored string-type fields using the `DOCS_AND_FREQS_AND_POSITIONS` index options provided by Lucene[1]:

- The `SUBJECT` field.
- The `BODY` field.

In addition to these indexed fields, we store the email's id and pre-compute the *root id* (id of the first email in the current email's thread), as these are essential for producing actionable results from any query search on the index.

Dataset Generation

Probably the most critical part of the email indexer is its workflow for taking in mbox files, using the Mbox Parser to extract emails, and build the dataset and its various components. The following snippet illustrates an abridged version of the dataset generation algorithm (with trivial log messages and typical Java verbosity removed).

```java
 1  public CompletableFuture<Void> generate(Collection<Path> mboxFileDirs, Path dsDir) {
 2    return Async.run(() -> {
 3      // Parse files and create database
 4      Files.createDirectories(dsDir);
 5      DatabaseGenerator dbGen = new DatabaseGenerator(dsDir.resolve("database"));
 6      List<Path> mboxFiles = new ArrayList<>();
 7      for (var dir : mboxFileDirs) {
 8        mboxFiles.addAll(findMboxFiles(dir));
 9      }
10      MBoxParser parser = new MBoxParser(new SanitizingEmailHandler(dbGen));
11      for (var file : mboxFiles) {
12        parser.parse(file);
13      }
14      dbGen.postProcess(status);
15      dbGen.close();
16
17      // Generate Lucene Index
18      EmailDataset dataset = new EmailDataset(dsDir);
19      new EmailIndexGenerator().generateIndex(dataset);
20      dataset.close().join();
21
22      // Generate Metadata
23      Properties props = new Properties();
24      props.setProperty("version", "3");
25      props.store(Files.newBufferedWriter(dataset.getMetadataFile()), null);
26    });
27  }
```

Notice on line 8 that we use a `SanitizingEmailHandler` to handle all parsed emails. This handler applies some extra sanitization to further reduce the amount of useless junk that ends up in the final dataset. It removes any emails which:

- Have an unknown charset (we're not able to decode them).
- Have a missing or empty body.
- Have a missing or empty message id.

Additionally, it also performs the following modifications to emails:

- Removes reply texts which start with ”—–Original Message—–”, or whose lines begin with ”>”. This greatly reduces the volume of text for large email threads, and helps to reduce the number of false-positives returned by searches.
- Strips ”<” and ”>” from the beginning and end of any email's message id, in-reply-to, or sent-from attributes.

Data Access API

In order for the email indexer API to be used by third-party applications, we provided a set of *repositories* that can be used to find, create, update, and delete entities from the dataset, as well as a set of *searchers* that provide more advanced retrieval mechanisms with SQL and Lucene searching.

1. The `EmailRepository` provides access to emails, including methods to find both the full email entity, and a *preview* which omits the body and in-reply-to fields to make it feasible for applications in search results or other scenarios where the full email information isn't needed.
2. The `TagRepository` provides access to tags, and various methods for adding and removing tags from emails.
3. The `EmailSearcher` has methods for finding emails using a collection of `SearchFilters`, which are used to build SQL queries over the set of emails. The methods in the EmailSearcher generally return an `EmailSearchResult` which contains the actual email results, as well as some information that's useful for pagination implementations in third-party applications.
4. The `EmailIndexSearcher` has methods for finding emails (or email threads) using an Apache Lucene search query.

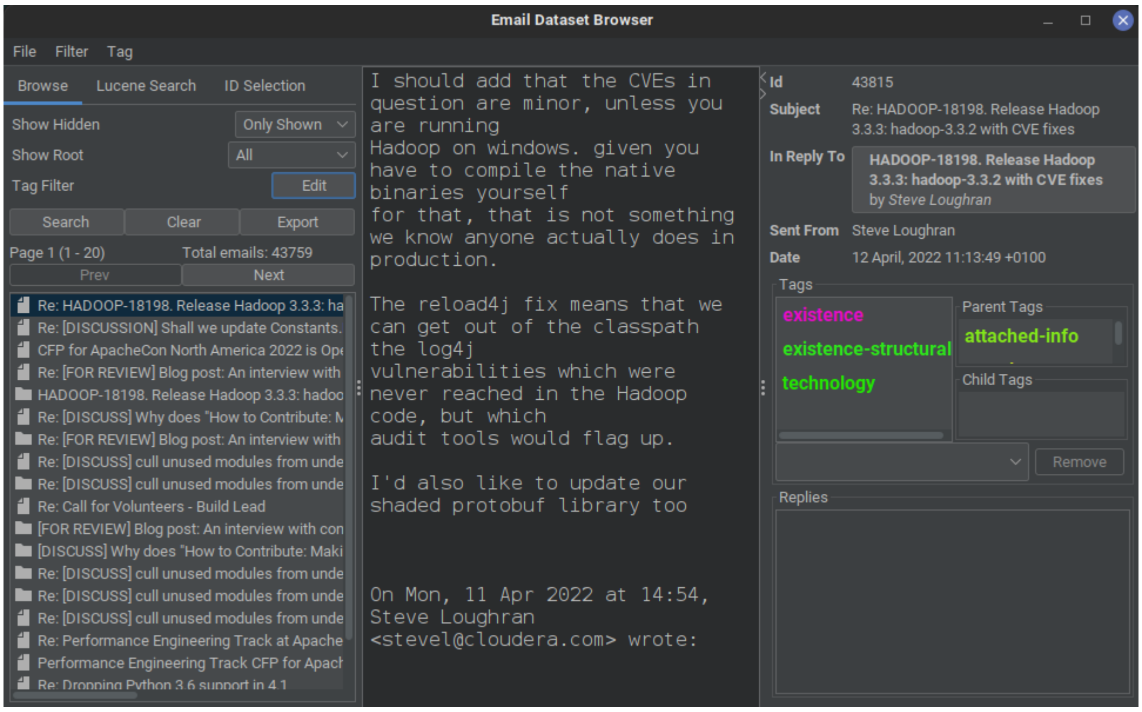
For developers' convenience, all data-access objects are full documented with accurate Javadoc so that the usage of the API is abundantly clear.

Email Dataset Browser

The main part of the research, categorizing emails based on the types of architectural design decisions they contain, is accomplished by the Email Dataset Browser application. It is a desktop application built with Java and the Swing user-interface component library.

While the original design is based on the *ArchDetector* tool designed by den Boon[4], it was built anew to better support the structure of email threads, and provide a more extendable platform for analysis without needing to deploy a web server or PostgreSQL database like the ArchDetector did. Additionally, since den Boon's research, the Apache Software Foundation has updated their infrastructure for mailing list management, which broke the ArchDetector's automatic mailing list download functionality. Our Email Downloader tool addresses that.

The browser is essentially a front-end for the Email Indexer's functionality, and provides some other conveniences. The remainder of this section will discuss all the features of the app, and how it is typically used for categorizing emails.



Show Hidden: Only Shown
Show Root: All
Tag Filter

Page 1 (1 - 20) Total emails: 43759

- Re: HADOOP-18198. Release Hadoop 3.3.3: ha
- Re: [DISCUSSION] Shall we update Constants.
- CFP for ApacheCon North America 2022 is Ope
- Re: [FOR REVIEW] Blog post: An interview with
- HADOOP-18198. Release Hadoop 3.3.3: hadoo
- Re: [DISCUSS] Why does "How to Contribute: M
- Re: [FOR REVIEW] Blog post: An interview with
- Re: [DISCUSS] cull unused modules from unde
- Re: [DISCUSS] cull unused modules from unde
- Re: Call for Volunteers - Build Lead
- [FOR REVIEW] Blog post: An interview with con
- [DISCUSS] Why does "How to Contribute: Maki
- Re: [DISCUSS] cull unused modules from unde
- Re: [DISCUSS] cull unused modules from unde
- Re: [DISCUSS] cull unused modules from unde
- Re: Performance Engineering Track at Apache
- Performance Engineering Track CFP for Apach
- Re: Dropping Python 3.6 support in 4.1

```
I should add that the CVEs in
question are minor, unless you
are running
Hadoop on windows. given you
have to compile the native
binaries yourself
for that, that is not something
we know anyone actually does in
production.

The reload4j fix means that we
can get out of the classpath
the log4j
vulnerabilities which were
never reached in the Hadoop
code, but which
audit tools would flag up.

I'd also like to update our
shaded protobuf library too



On Mon, 11 Apr 2022 at 14:54,
Steve Loughran
<stevel@cloudera.com> wrote:
```

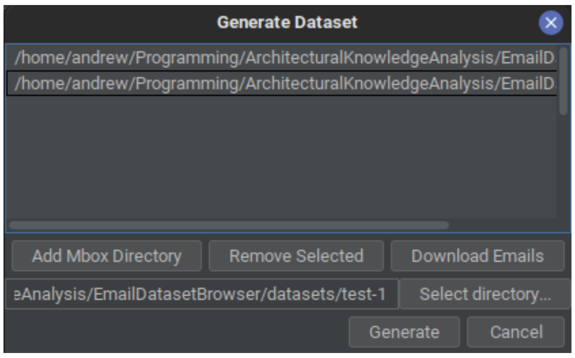
Id: 43815
Subject: Re: HADOOP-18198. Release Hadoop 3.3.3: hadoop-3.3.2 with CVE fixes
In Reply To: **HADOOP-18198. Release Hadoop 3.3.3: hadoop-3.3.2 with CVE fixes** by *Steve Loughran*
Sent From: Steve Loughran
Date: 12 April, 2022 11:13:49 +0100

Tags: **existence**, **existence-structural**, **technology**
Parent Tags: **attached-info**
Child Tags
Replies

Figure 2: A screenshot of the email dataset browser app's main user interface, showing an email that has some tags.

Generating a Dataset

The browser app allows users to generate a new email dataset from mbox files on the user's file system, and allows users to download mbox files on-the-fly from Apache's mailing lists. Navigate in the app's menu to **File** and then **Generate Dataset**, and you should see a popup open, as shown in figure 3. The main interface consists of a list of directories to read mbox files from. You can add and remove directories from this list.

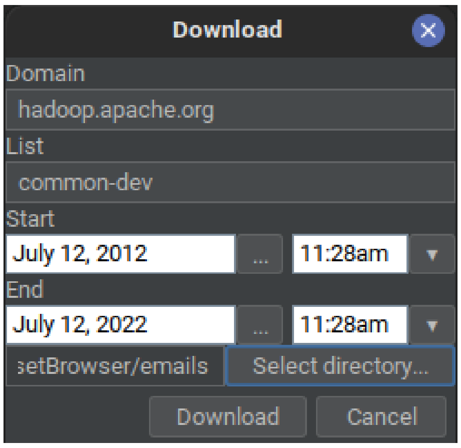


/home/andrew/Programming/ArchitecturalKnowledgeAnalysis/EmailD
/home/andrew/Programming/ArchitecturalKnowledgeAnalysis/EmailD
eAnalysis/EmailDatasetBrowser/datasets/test-1

Figure 3: The popup for generating a new dataset.

If you'd like to download mbox files from Apache's mailing lists, you can use the **Download Emails** button to bring up another popup, as shown in figure 4. In this interface, the user can provide the domain and list names, as discussed in the Email Indexer's section, as well as a start and end date and directory to download to. Do note that download can take a significant amount of time (up to 10 or 20 minutes for large mailing lists), so be patient with this step.

Once you've downloaded and/or located all the directories containing mbox files you'd like to include in the dataset, you must select a directory in which to place the dataset. This directory will be populated with the dataset's database file, metadata file, and Lucene index directory. When you've done this, click **Generate** to generate the dataset.



Domain: hadoop.apache.org
List: common-dev
Start: July 12, 2012 11:28am
End: July 12, 2022 11:28am
setBrowser/emails

Figure 4: The popup for downloading mbox files from Apache mailing lists.

Again, this process may take a while, depending on the size of the dataset and your system's hardware performance, so please be patient. A popup will appear that shows messages from the email indexer's dataset generation system to indicate the current status as the process progresses. Once the generation is complete, you can click the **Done** button to exit the popup. The dataset now exists on your system, in the directory you specified in the popup, and you can open it like you would any other dataset using the app.

Opening and Browsing Datasets

To open a dataset, navigate to the **File** menu, and select **Open Dataset**. In this file selection dialog, select the a dataset's directory to open it. You may also select a ZIP archive file instead, in which case the app will first unzip the archive to a directory of the same name, and open that. Once you open a dataset, you'll see on your left a panel with three tabs labelled *Browse*, *Lucene Search*, and *ID Selection*. These are the three different ways in which you can navigate the dataset.

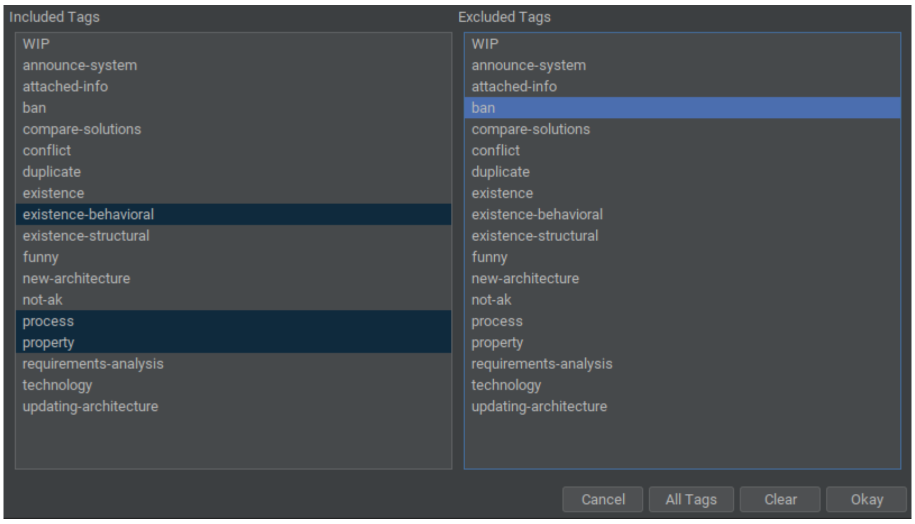
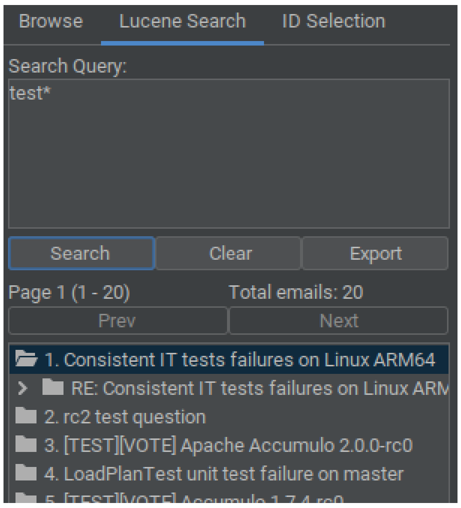


Figure 5: Use this popup to filter emails in the standard browse panel based on the tags that they do or do not have.



1. Consistent IT tests failures on Linux ARM64
RE: Consistent IT tests failures on Linux ARM
2. rc2 test question
3. [TEST][VOTE] Apache Accumulo 2.0.0-rc0
4. LoadPlanTest unit test failure on master
5. [TEST][VOTE] Accumulo 1.7.4-rc0

Figure 6: The view for searching the dataset using Lucene queries.

In the *Browse* view, you can browse the emails using a traditional paginated, filterable view. You can choose to show only hidden emails (those that have been removed by a user because they're irrelevant), only non-hidden emails (the default option), or both. You can also choose to show only *root* emails, (emails that are at the root of an email thread, i.e. the first email in the thread), only child emails, or both (the default option). Finally, you can also filter the emails based on the tags they have, using the tag filter shown in figure 5. If you select tags under the *Included Tags* list, only emails that have those tags will be shown. Conversely, if you select tags under the *Excluded Tags* list, only emails which **do not** have those tags will be shown.

In the *Lucene Search* view, you can search the dataset using a provided Lucene query. You can read about the acceptable syntax for such queries in the Lucene 9.1 documentation. Figure 6 shows an example where we search over a dataset using the query "test*". The results show the list of all email threads which match the query, ranked from most to least relevant, according to Lucene. By default, only 20 results are shown, but this can be changed in the **Settings** under the **File** menu; look for the `Browse page size` option.

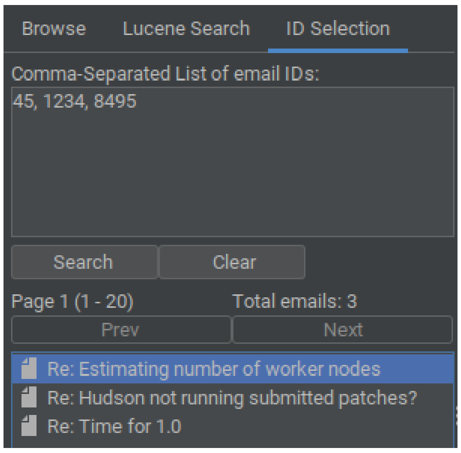

Browse
Lucene Search
ID Selection
Comma-Separated List of email IDs:
45, 1234, 8495
Search
Clear
Page 1 (1 - 20)
Total emails: 3
Prev
Next
Re: Estimating number of worker nodes
Re: Hudson not running submitted patches?
Re: Time for 1.0

Figure 7: The view for searching the dataset using specific email ids.

Finally, in the *ID Selection* view, you may enter a comma-separated list of email ids, to view only those emails. Take note that by *id*, we mean the id which the email indexer has assigned to the email when inserting it into the database, **not** the email's message id. As shown in figure 2, the id can be found at the top of the email's information panel on the right-hand side of the app. This view is especially useful for the iterative process of familiarizing oneself with the types of architectural design decisions, and how they appear in various emails in practice.

In each of the three views, a list of emails is shown in the *email tree view* below the view-specific options panel. As shown in figure 8, this tree view allows the user to expand an email to show its replies, if any exist. With this type of view, it's easy to navigate large email threads, and get an overview of the high-level structure of the thread. Each email is listed in the view using its subject, similar to how many email client programs already do.

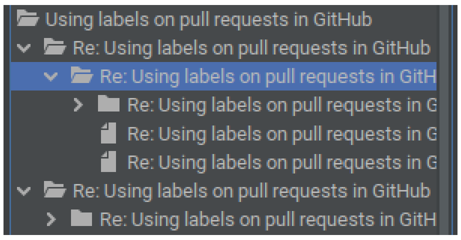

Using labels on pull requests in GitHub
Re: Using labels on pull requests in GitHub
Re: Using labels on pull requests in GitH
Re: Using labels on pull requests in G
Re: Using labels on pull requests in G
Re: Using labels on pull requests in G
Re: Using labels on pull requests in GitHub
Re: Using labels on pull requests in GitH

Figure 8: The tree view used to navigate lists of emails.

Beside each email is shown an *info panel* with detailed information about the selected email, as shown in figure 9. This panel is shown beside each selected email, and can be expanded or hidden using the controls on the vertical divider between it and the email's main content view. Here you can see the following information, listed in the order in which it appears:

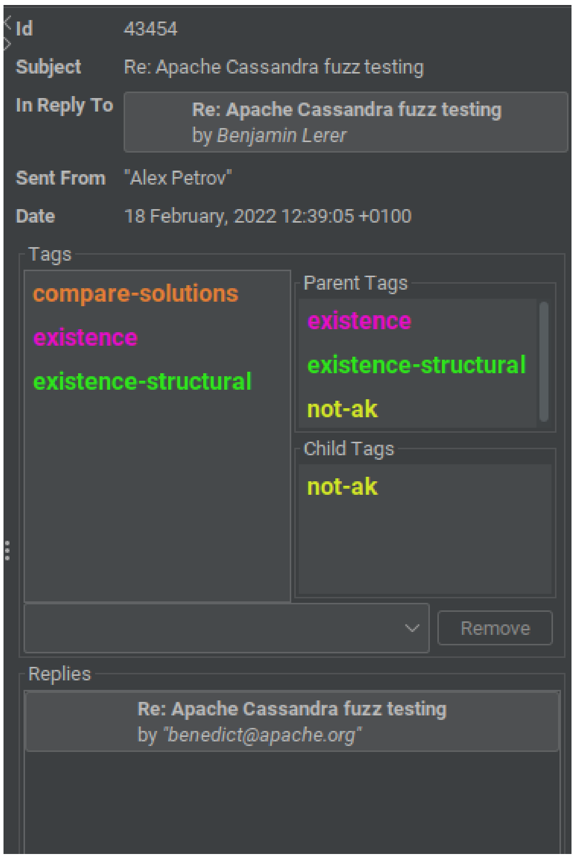
1. The email's assigned id. This is used in the ID Selection view.
2. The email's full subject field.
3. A link to the email that this one is a reply to, if any.
4. The name of the person who sent the

email.

5. The date at which the email was sent.
6. A panel which shows all the tags that have been assigned to the email. This will be covered more later.
7. A list of replies to the email. You can click on a reply to navigate to it.

In the email's *tag panel*, you can see all the tags that have been assigned to the email. On the right, you can see the *parent tags* (any tags that belong to any parent email of this one) and the *child tags* (any tags that belong to any child email of this one). Each tag is colored using a random color derived from a hash of the tag's name, to improve readability and easy identification of tags.

At the bottom of the tag panel, there's a select box where you can select tags to apply to the selected email. If you select any of the tags in the email's main list of tags, you can also click the **Remove** button to remove those tags. You can double-click on any of the tag items in any of the lists (main, parent, child) to edit the tag's name or description.



Id 43454
Subject Re: Apache Cassandra fuzz testing
In Reply To Re: Apache Cassandra fuzz testing
by *Benjamin Lerer*
Sent From "Alex Petrov"
Date 18 February, 2022 12:39:05 +0100

Tags
compare-solutions
existence
existence-structural

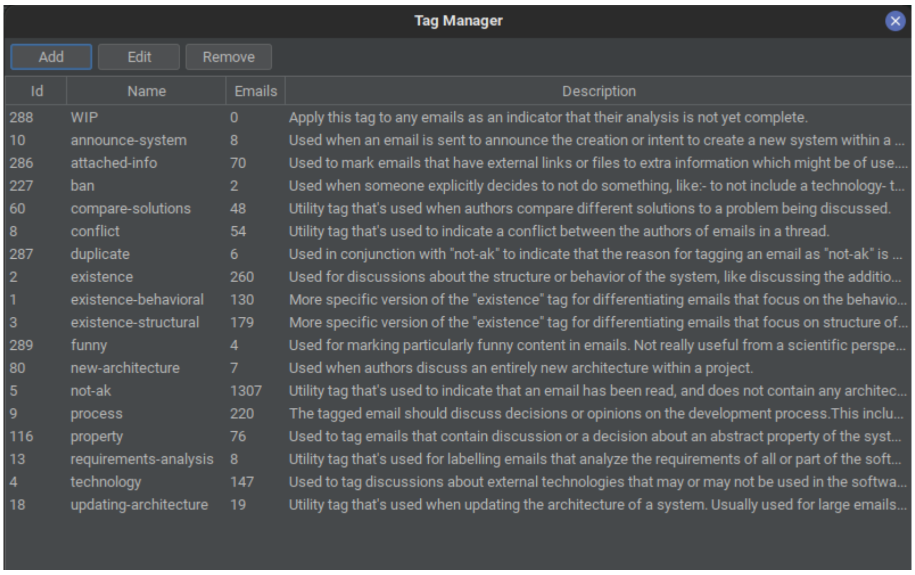
Parent Tags
existence
existence-structural
not-ak

Child Tags
not-ak

Replies
Re: Apache Cassandra fuzz testing
by *"benedict@apache.org"*

Figure 9: The information panel that's shown beside each email.

Managing Tags



Id	Name	Emails	Description
288	WIP	0	Apply this tag to any emails as an indicator that their analysis is not yet complete.
10	announce-system	8	Used when an email is sent to announce the creation or intent to create a new system within a ...
286	attached-info	70	Used to mark emails that have external links or files to extra information which might be of use....
227	ban	2	Used when someone explicitly decides to not do something, like:- to not include a technology- t...
60	compare-solutions	48	Utility tag that's used when authors compare different solutions to a problem being discussed.
8	conflict	54	Utility tag that's used to indicate a conflict between the authors of emails in a thread.
287	duplicate	6	Used in conjunction with "not-ak" to indicate that the reason for tagging an email as "not-ak" is ...
2	existence	260	Used for discussions about the structure or behavior of the system, like discussing the additio...
1	existence-behavioral	130	More specific version of the "existence" tag for differentiating emails that focus on the behavio...
3	existence-structural	179	More specific version of the "existence" tag for differentiating emails that focus on structure of...
289	funny	4	Used for marking particularly funny content in emails. Not really useful from a scientific perspe...
80	new-architecture	7	Used when authors discuss an entirely new architecture within a project.
5	not-ak	1307	Utility tag that's used to indicate that an email has been read, and does not contain any architec...
9	process	220	The tagged email should discuss decisions or opinions on the development process.This inclu...
116	property	76	Used to tag emails that contain discussion or a decision about an abstract property of the syst...
13	requirements-analysis	8	Utility tag that's used for labelling emails that analyze the requirements of all or part of the soft...
4	technology	147	Used to tag discussions about external technologies that may or may not be used in the softwa...
18	updating-architecture	19	Utility tag that's used when updating the architecture of a system. Usually used for large emails...

Figure 10: The tag manager popup, which shows a list of all tags and some information about them.

To create, edit, or remove the set of tags that are available for applying to emails, you can use the *tag manager*. Open this popup by navigating to the **Tag** menu, and select **Manage Tags**. You should see the popup shown in figure 10. Here, you can see a table showing all tags currently in the dataset. You can select a tag and click **Edit**, or just double-click it to edit. New tags can be added via the **Add** button, and selected tags can be removed via the **Remove** button.

Do take care with removal though, as removing a tag is a permanent and irreversible operation that will permanently remove information from your dataset. Consider renaming a tag to have an ”unused” prefix first, until you're absolutely certain that a tag can be removed without losing any valuable information.

Filtering Emails

While our APIs make a best attempt to remove some of the most egregious ill-formatted and useless emails, a large amount of noise does make it through to the dataset. Therefore, we provide a suite of tools to filter out emails using a few different methods. These are available in the **Filter** menu of the app.

First, any email can be individually hidden with the **Hide** and **Show** buttons. Besides that, you can also hide all emails sent by the same author as the one who sent the selected email, or hide all emails that have exactly the same body content as the selected email, with **Hide by Author** and **Hide all by body** respectively. Finally, **Hide by SQL** can be used to manually define an SQL clause to hide all emails that match that clause.

Email Dataset Report Generator

When enough data has been collected through the categorization of emails in a dataset, the dataset report generator pipeline is used to extract, analyze, and visualize the data. This pipeline is orchestrated by a `run_pipeline.d` script that invokes each of the sub-processes in the correct order.

This pipeline uses Java 17 with Maven, and D 2.099.1 with the Dub package manager. Additionally, it makes use of the DSH library for D. It's not required to have it installed, but it does make it easier to make changes to the pipeline script. You will need both Java and D, and their respective package managers installed to run the pipeline successfully. Additionally, the pipeline has not been validated on Windows or Mac operating systems. Use these at your own risk.

`run_pipeline.d`, or the *pipeline runner* as it'll be referred to from now on, is an executable file that, if given the proper executable permissions, will run as-is. Otherwise, you can build the file using Dub as a single-file project. If using DSH, you can run the following command to compile a binary executable:

```
dshutil compile run_pipeline.d
```

The pipeline runner, when run, expects a single argument, which determines the behavior of the program.

- If the **help** argument is passed, then a simple help message is shown.
- If the **clean** argument is passed, then files generated by previous runs are removed.
- If the **rebuild** argument is passed, then executable files for the various steps of the pipeline are rebuilt from source. Use this if you make changes to the source. Note that executables are also built automatically if they're missing at runtime.
- Otherwise, the argument is interpreted as the path to a **dataset directory** to run the pipeline on.

To summarize, the report generation pipeline is as follows:

1. Extract data from the dataset.
2. Perform analysis on the data.
3. Generate visualizations of the data.

The remainder of this section will discuss each step of the pipeline in more detail.

Extraction

To extract the necessary data, a small Java program uses the Email Indexer as a dependency, and exports all emails that have any tag, into a large JSON array. Each email in the export is structured like so:

```json
{
  "id": 43815,
  "parent_id": 43810,
  "message_id": "CAL7CpJx7sSVZR+idVmdvwgDGX5DOD31L\u003d+fw@mail.gmail.com",
  "subject": "Re: HADOOP-18198. Release Hadoop 3.3.3: hadoop-3.3.2 with CVE fixes",
  "in_reply_to": "CAL7CpJydXGTxkgzV903dc2HR2yQgXpacs_iJa7MirKJQ@mail.gmail.com",
  "sent_from": "Steve Loughran \u003cstevel@cloudera.com.INVALID\u003e",
  "date": "2022-04-12T11:13:49+01:00",
  "body": "I should add that the CVEs...",
  "tags": [
    "existence",
    "existence-structural",
    "technology"
  ]
}
```

In addition to the emails, this program also exports information about various Lucene searches, because Lucene is inherently a Java library and this is our last opportunity to extract such data before the next steps of the pipeline. A list of queries is provided in the classpath of the program, in a `queries.properties` file. For each query, we get the list of individual emails and email threads that are returned as results, and compile this data into a JSON array.

```json
{
  "name": "decision_factors",
  "query": "actor* availab* budget* business case* client* concern* conform*",
  "threads": [39719, 42370, 42375],
  "emails": [43269, 42370, 42375]
}
```

Finally, the program exports information about all known mutations to the dataset, and puts these into a JSON array. Below is an example of a mutation object.

```json
{
  "id": 6,
  "affected_email_count": 458,
  "description": "Hiding all emails sent by email addresses like:
↪  %hudson@lucene.zones.apache.or%"
}
```

Thus the end result of this step is an `emails.json` containing the list of all tagged emails, a `searches.json` containing a list of all Lucene queries and the results they returned, and a `mutations.json` containing a list of all recorded mutations to the dataset.

As an aside, we use the extraction step to opportunistically export and perform any additional steps besides core data extraction which can easily be done in Java. To that end, the extraction program also performs a minimalistic natural-language-processing pipeline on the set of tagged emails using Stanford's CoreNLP library. With CoreNLP, we annotate every tagged email in the dataset to find the most common lemmas (simple canonical or dictionary form of a word[8]) for the following groups of emails:

- Emails with any architectural design decision (existence, technology, process, or property).
- Emails that contain any existence decision.
- Emails that contain any technology decision.
- Emails that contain any process decision.
- Emails that contain any property decision.
- Emails that are categorized as non-architectural.

This data is exported as a JSON object whose keys refer to named lemmatization group objects, each of which has its own ordered list of lemmas, starting with the most frequent. You can find it in `lemmas.json` after running the analysis pipeline.

While the lemmatization data of emails will not be used further in this research, we present this feature so that it can be used by future efforts to pinpoint the best keywords to use when using traditional index-based keyword search approaches. The logic also serves as a foundation for more extensive natural-language analysis of email contents, for any future research in that area.

After the data is extracted by the previous step, we use a D program to run a suite of analyses on the data. The program can be run using the following arguments:

```
./analysis -e emails.json -s searches.json --minifyJson
```

Where **-e** defines the required `emails.json` input file, and **-s** defines the required `searches.json`. **–minifyJson** is an optional flag that, if present, minifies the JSON output of the analysis.

Internally, the set of emails is parsed into a data structure that allows us to rapidly process all analytics by keeping the emails in memory for the duration of the program. Then, we introduce two types of analyses: those which operate on emails, and those which operate on search results separately. This allows us to define two generic interfaces that all independent analyses can conform to.

```
/**
* A set of emails as obtained from a dataset, with some pre-computed arrays
* for faster data lookup.
*/
class EmailSet {
  Email[] emails;
  Email[] rootEmails;
  Email[long] emailsById;
  Email[][long] repliesById;
  string[] tags;
  Email[][string] emailsByTag;
}

/**
* Represents an analysis operation that consumes emails to add to its data,
* and produces at the end some data which can be added to a JSON object.
*/
interface Analysis {
  void initialize(EmailSet set, string[] akTags);
  void accept(Email email, EmailSet set, string[] akTags);
  void addToJson(ref JSONValue obj);
}

/**
* Represents an analysis operation to be performed on dataset search results.
*/
interface SearchQueryAnalysis {
  void accept(EmailSet set, SearchQueryData data, string[] akTags, ref JSONValue obj);
}
```

The actual analyses that are performed by this program will be discussed in more detail in the Methodology section.

Ultimately, both types of analyses append their results to a referenced JSON object that will be written to the program's standard output stream. In the context of the pipeline runner, the output is sent to a `analysis_results.json` file, so that it can be read both by users, and by the visualization program that runs in the next step.

Visualization

To visualize the data in the charts and diagrams you see in this report, we use the JFreeChart library in conjunction with a simple Java program that reads the JSON output from the analysis step.

The program defines a simple interface so that visualization can be subdivided into a set of independent rendering operations.

```java
public interface ChartRenderer {
  void renderCharts(JsonObject data) throws Exception;
}
```

This way, the rendering operations can be parallelized. The speedup from this is minimal, but should provide clear benefits with larger datasets (of more than 5,000 tagged emails). The resultant images are placed in the working directory of the program. This is managed by the pipeline runner that correctly adapts its working directory to place the visualizations into a `visual` directory alongside the other JSON files.

Methodology

For the purposes of this research, we will focus on analyzing the contents of mailing lists from three major open-source projects from the Apache Software Foundation: Hadoop, Cassandra, and Tajo. Mailing list data will be obtained from lists.apache.org, and this data will be indexed and searched over using Apache Lucene. A subset of emails from these mailing lists will be categorized based on the type of architectural design decisions they contain, in order to conjecture about the efficacy of searching.

We'll first discuss the process of preparing the datasets, followed by an overview of the methods of categorization and analysis.

Choosing Sources

While previous work by den Boon chose to analyze data from both issue tracking boards and mailing lists from open-source software projects[4], this research will limit the scope of sources to just mailing lists, since that is the focus of our research questions. More specifically, a *mailing list* is a "mechanism whereby a message may be distributed to multiple recipients by sending to one address."[10] Put more plainly, someone may *subscribe* to a mailing list, and in doing so, they will receive any email which is addressed to that list. Likewise, that person may also send an email to the list's address, and it will be broadcast to all other subscribed recipients. In the context of software development, mailing lists have been used extensively since their inception as a way to collectively discuss and make design decisions regarding large open-source projects. Because of this, we can expect mailing lists for software development to be particularly rich in architectural knowledge, and they make a good candidate for analyzing the effectiveness of various search approaches.

Within the domain of mailing list communications, we can further narrow our search down to software projects which we can reasonably expect to contain a large amount of, and large variety of architectural decisions. These would generally be large projects with many moving parts, which have a broad range of applications and societal uses. The Apache Software Foundation (ASF) is a leader in these sorts of projects, with many open-source systems for managing, processing, and searching through big data, which find their usage in various state-of-the-art enterprises and research endeavors, notably NASA, Facebook, Twitter, Netflix, and so on[2]. From the ASF, we chose mailing lists from the following three projects for analysis:

1. Cassandra - An open source NoSQL distributed database trusted by thousands of companies for scalability and high availability without compromising performance.
2. Hadoop - A framework that allows for the distributed processing of large data sets across clusters of computers using simple programming mod-

els. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage.

3. Tajo - A robust big data relational and distributed data warehouse system for Apache Hadoop. Tajo is designed for low-latency and scalable ad-hoc queries, online aggregation, and ETL (extract-transform-load process) on large-data sets stored on HDFS (Hadoop Distributed File System) and other data sources.

Project descriptions obtained from the respective projects' homepages.

For each of the above projects, we obtained one or more mailing lists dedicated to the discussion of the project's internal development, thus excluding irrelevant discussions about usage and user issue reports.

For Cassandra, we chose the following mailing lists:

- dev@cassandra.apache.org

For Hadoop, we chose the following mailing lists:

- common-dev@hadoop.apache.org
- hdfs-dev@hadoop.apache.org
- mapreduce-dev@hadoop.apache.org
- yarn-dev@hadoop.apache.org

For Tajo, we chose the following mailing lists:

- dev@tajo.apache.org

Fetching and Processing Sources

The first step to being able to analyze and identify architectural knowledge in mailing lists is, of course, to obtain the emails from the mailing lists in the first place. For this purpose, the Email Downloader utility library was developed, and for parsing and preparing data for use in our datasets, the MBox Parser utility library was developed.

Downloading

The EmailDownloader utility offers an asynchronous interface for downloading a series of MBox (email archive format) files to a directory. Because all of the mailing lists used by this research come from the Apache Software Foundation, the library includes an implementation for downloading from the ASF's mailing list internal API at https://lists.apache.org/api/mbox.lua, with support for rate-limited downloading and the ability to detect long periods of inactivity and short-circuit prematurely to save time.

The end result is that we are able to download a large archive of all emails sent in a mailing list, since its inception, to today. Specifically, from the Hadoop, Cassandra, and Tajo mailing lists, we obtained all emails between the dates of 31-01-2006 and 12-04-2022, which totaled 246,673 emails.

- 20,489 emails from Cassandra's mailing list.
- 220,001 emails from Hadoop's mailing lists (this makes up the majority of the dataset).
- 6,183 emails from Tajo's mailing list.

Processing

Once we have downloaded a large amount of MBox files, we must parse their contents to extract the individual emails for use in our analysis. This is done by the MBox Parser utility library, which offers an interface for parsing a collection of MBox files and issuing a callback handler for each email that was obtained.

The end result is a collection of many Email objects that are ready for further use, whose format is described in the code snippet below:

```java
public class Email {
  public String messageId;
  public String inReplyTo;
  public String sentFrom;
  public String subject;
  public ZonedDateTime date;

  public String mimeType;
  public String charset;
  public String transferEncoding;
  public byte[] body;
}
```

We use the Email Indexer's set of preliminary filtering efforts to remove junk emails, and then used a manual approach to remove any further emails which we deemed to be irrelevant (usually, advertisements, off-topic messages, and automated messages). Therefore, all emails sent from the following list of addresses have been pruned from the dataset:

- `hudson@lucene.zones.apache.org`
- `hudson@hudson.zones.apache.org`
- `hudson@hudson.apache.org`
- `git@apache.org`
- `jenkins@builds.apache.org`
- `jira@apache.org`

This list of email addresses which send irrelevant automated messages is limited to only those which a human researcher encountered and identified, so it must be noted that there is no guarantee that this is an exhaustive list; it is simply a list compiled as emails were categorized.

After a review of the quality of the dataset with the supervisor, it was determined that we would further filter the emails to exclude those which contained the texts ”call for papers” or ”workshop” in their subject, or which contained ”call for papers” within the body of the email. This allows us to remove a large amount of irrelevant emails that will almost certainly not contain a lot of architectural knowledge. These ”call for papers” emails also usually attempt to garner more traffic by including large amounts of SEO-friendly keywords, which tends to place them high in the Lucene search results; something we want to avoid.

Categorization Process

This study categorized the architectural knowledge contained in 1919 individual emails, spread across 188 different email threads from Hadoop, Cassandra, and Tajo developer mailing lists. This section provides an overview of how this was achieved.

Email categorization happened in *iterations*, where a selection of emails would be analyzed for the architectural knowledge they potentially contain, and a subset of this section was presented to the supervisor for verification. With each iteration, feedback and a broadening collection of examples helped to build the final list of design decisions included in this paper, and already-categorized emails were revised under the new definitions. More specifically,

1. The researcher categorizes a set of emails (usually between 100 and 200), and selects a sample from that set, and sends the ids of the sample set to the supervisor, along with a copy of the dataset at that point in time.
2. The supervisor evaluates the accuracy of tag annotations on the emails in the sample set, and discusses the motivations for why certain tags do or do not apply to specific emails.
3. The researcher makes note of each of the emails which were incorrectly categorized, and their reason for the erroneous categorization.
4. The researcher rectifies categorization on all emails which were incorrectly categorized, and revisits all emails from the original set whose tags match in any way with the erroneous emails, so that they can be re-evaluated.

In total, 9 iterations were performed over the course of this research. With the first iteration, nearly every single email was found to have errors in its classification. However, towards the 7th, 8th, and 9th iterations of review, consensus was reached on nearly every email in the review set.

The rate at which email categorization can happen is a clear limiting factor to the scale of this study, because of the time requirements to develop an intuition for accurately identifying the design decisions in emails, and the time required to simply read such a volume of emails. This will be discussed further in the paper's conclusion.

To better understand the process of categorizing emails based on the types of design decisions they contain, the following sections will describe, in detail, how to identify instances of those design decisions in practice.

Identifying **Existence** Decisions

To identify existence decisions, we look for specific cases where components and their behaviors are discussed.

```
... There has already been some Slack discussion around this, but for anyone who
↪  doesn't follow that closely, I'd like to lobby more widely for my proposal
↪  in CASSANDRA-17292 to eventually move cassandra.yaml toward a more nested
↪  structure. ...
```

In the above example, we see a discussion regarding changes to the structure of the `cassandra.yaml` file (a cassandra node's configuration file). This is an example where a developer is advising to make a structural change to a key component in the Cassandra architecture.

In another example, we see that existence decisions can be found when a proposal is made to add a new component, or merge an external component into the main system architecture.

```
So, on Orange side, we propose to discuss with Datastax how to best merge Casskop's
↪  features in Cass-operator.
These features are:
- nodes labelling to map any internal architecture (including network specific labels
↪  to muti-dc setup)
- volumes & sidecars management (possibly linked to PodTemplateSpec)
- backup & restore (we ruled out velero and can share why we went with Instaclustr but
↪  Medusa could work too)
- kubectl plugin integration (quite useful on the ops side without an admin UI)
- multiCassKop evolution to drive multiple cass-operators instead of multiple casskops
↪  (this could remain Orange internal if too specific)
```

In addition to adding/modifying/removing components from a system's architecture, we can find existence decisions in cases where the behavior and coupling of components and subsystems in an architecture occurs.

```
                        +-------+
+------+ credentials 1 | SSO   |
|CLIENT|-------------->|SERVER|
+-------+  :tokens      +-------+
2 |
  | access token
  V :requested resource
```

```
+--------+
|HADOOP  |
|SERVICE|
+--------+

The above diagram represents the simplest interaction model for an SSO service in
↪  Hadoop.
1. client authenticates to SSO service and acquires an access token
a. client presents credentials to an authentication service endpoint exposed by the SSO
↪  server (AS) and receives a token representing the authentication event and verified
↪  identity
b. client then presents the identity token from 1.a. to the token endpoint exposed by
↪  the SSO server (TGS) to request an access token to a particular Hadoop service and
↪  receives an access token
2. client presents the Hadoop access token to the Hadoop service for which the access
↪  token has been granted and requests the desired resource or services
a. access token is presented as appropriate for the service endpoint protocol being
↪  used
b. Hadoop service token validation handler validates the token and verifies its
↪  integrity and the identity of the issuer
```

Here, we see a prime example of an existence design decision which focuses on how different parts of the system interact, and this description of the behavior of Hadoop's single-sign-on authentication flow is a key example of behavioral-existence architectural knowledge.

In another example, we see a more strictly behavioral form of existence knowledge, which appears more in the form of a high-level algorithm or workflow, than an explicit description of component interconnections.

```
One failure scenario: Node A, B, and C replicate some data.  C goes
down.  The data is deleted.  A and B delete it and later GC it.  C
comes back up.  C now has the only copy of the data so on read repair
the stale data will be sent to A and B.

A solution: pick a number N such that we are confident that no node
will be down (and catch up on hinted handoffs) for longer than N days.
(Default value: 10?)  Then, no node may GC tombstones before N days
have elapsed.  Also, after N days, tombstones will no longer be read
repaired.  (This prevents a node which has not yet GC'd from sending a
new tombstone copy to a node that has already GC'd.)
```

Identifying **Technology** Decisions

In contrast with existence decisions, technology decisions are some of the most trivial to identify, because it's just naturally quite easy to identify discussions about third-party technologies as separate from the system architecture itself.

Here's an example of an email discussing the tradeoffs of python and jdk-based cassandra multi-node testing.

```
The primary tradeoffs as I understand them for moving from python-based multi-node
↪  testing to jdk-based are:
Pros:
  1. Better debugging functionality (breakpoints, IDE integration, etc)
  2. Integration with simulator
  3. More deterministic runtime (anecdotally; python dtests _should_ be deterministic
  ↪  but in practice they prove to be very prone to environmental disruption)
  4. Test time visibility to internals of cassandra
Cons:
  1. The framework is not as mature as the python dtest framework (some functionality
  ↪  missing)
  2. Labor and process around revving new releases of the in-jvm dtest API
  3. People aren't familiar with it yet and there's a learning curve
```

As the above example and the next one illustrate, most technology discussions tend to contain some form of informal comparison between competing technologies to accomplish a goal. We see this again in another conversation on comparing the Jira ticket system to GitHub and Apache's own in-house system for improving the code-review workflow.

```
We had a pretty long conversation about this very topic on the dev list
awhile ago (search for "Discussion: reviewing larger tickets" on the
mailing list). I think the final conclusion was that having the
back-and-forth via JIRA helped codify some of the design decisions that
took place during implementation and review that could be lost using an
external tool.

So while it's extra overhead and very raw from a tooling perspective, the
pros outweighed the cons.
```

And as one final example, technology knowledge is not limited to just comparisons, but also assertions about the quality or properties of a third-party technology in the context of a larger discussion. Take for example this email on the Thrift serialization API's `TRecordStream` component.

```
Yes. TRecordStream's fundamtental use case is to be a robust file format for
storing records (in our case thrift or ctrl delimited log data) and that
they/it be self describing.

This means fixed sized frames that can be skipped over in case of corruption
and providing transparent checksums and/or compression if needed.  And a way
to put the serializer/deserializer information in each header.

And of course cross platform/languages - Java, Python, Perl and C++.
```

Identifying **Process** Decisions

To identify process decisions, we look for conversations around the testing and development workflows that developers follow, since these often lead to actual concrete process decisions. For example, a common theme is the reiteration that a project needs more or better testing, which then spawns a discussion around how to adjust the development process to achieve that. We see that in this example below.

```
Hi Dev,

What principles do we have? How do we implement them?

Our team has been evaluating 3.0.x and 3.x for a large production deployment.  We have
↪  noticed broken tests and have been working on several patches.  However, large
↪  parts of the code base are wildly untested, which makes new contributions more
↪  delicate.
```

```
All of this ultimately reduces our confidence in the new releases and slows down our
↪  adoption of the 3.0 / 3.x and future 4.0 releases.

So, I'd like to have a constructive discussion around 2 questions:

1. What principles should the community have in place about code quality and ensuring
↪  its long term productivity?
2. What are good implementationg (as in rules) of these principles?

To get this started, here is an initial proposal:

Principles:

1. Tests always pass.  This is the starting point. If we don't care about test
↪  failures, then we should stop writing tests. A recurring failing test carries no
↪  signal and is better deleted.
2. The code is tested.

Assuming we can align on these principles, here is a proposal for their implementation.

Rules:

1. Each new release passes all tests (no flakinesss).
2. If a patch has a failing test (test touching the same code path), the code or test
↪  should be fixed prior to being accepted.
3. Bugs fixes should have one test that fails prior to the fix and passes after fix.
4. New code should have at least 90% test coverage.
```

Other process discussions can be more limited in scope, simply making an assertion about what *should* be the case for how the project is handled. This next example contains a simple process decision about how an `experimental` flag should be applied to features, and when to remove it.

```
Reviewers should be able to suggest when experimental is warranted, and conversation on
↪  dev+jira to justify when it’s transitioned from experimental to stable?

We should remove the flag as soon as we’re (collectively) confident in a feature’s
↪  behavior - at least correctness, if not performance.
```

Additionally, we identify quite a few process decisions which are structured more concretely as assertions about the current state of the development process, such as this one about how Cassandra’s major releases are structured.

```
We are moving away from designating major releases like 3.0 as "special,"
other than as a marker of compatibility.  In fact we are moving away from
major releases entirely, with each release being a much smaller, digestible
unit of change, and the ultimate goal of every even release being
production-quality.
```

Identifying **Property** Decisions

Finally, property decisions are, in practice, the most difficult to properly identify, since they are first of all quite a bit more rare than the others as we’ll see in the analysis section, and the very nature of property decisions is that they are broad, abstract suppositions about the qualitative attributes of the software architecture and its goals. In the first example, we present a case where properties of a new in-memory table implementation are established for Cassandra.

```
We would like to contribute our TrieMemtable to Cassandra.

https://cwiki.apache.org/confluence/display/CASSANDRA/CEP-19%3A+Trie+memtable+implementation

This is a new memtable solution aimed to replace the legacy implementation, developed
↪  with the following objectives:
- lowering the on-heap complexity and the ability to store memtable indexing structures
↪  off-heap,
- leveraging byte order and a trie structure to lower the memory footprint and improve
↪  mutation and lookup performance.
```

In another example, we see that a software architecture's property decisions don't necessarily need to focus on physical qualities like performance and memory usage, but also things like the quality of the system's security promises, and support for third-party integrations.

```
I've been recently looking into how we could improve security in
Cassandra by integrating external solutions. There are very interesting
projects out there, such as Vault[0], ...
... Wouldn't it be cool to have
automated, build-in certificate management instead? That's what got me
started to work on CASSANDRA-13971.
```

For a third example, we will show the first email sent by Michael Cafarella which started the HBase project in Hadoop, back in 2006. It's a long email, but even in just the opening three paragraphs, we see the establishment of several key properties of the system.

```
I've written up a design that I've been working on for a little bit, for
a project I'll call "HBase".  The idea is for Hadoop to implement something
similar in spirit to BigTable.  That is, a distributed data store that
places a greater emphasis on scalability than on SQL compatibility
or traditional transactional correctness.

BigTable is neither completely described anywhere, nor is it
necessarily exactly what we want.  So I'm not trying to clone BigTable,
but I am going to draw on it a lot.

My personal view is that BigTable is a great "physical layer" but not yet
a great database system.  A major thing it lacks is a good query language.
Another, freely admitted by the Google people, is any kind of inter-row
locking.  I'm not going to try to solve all these problems, but I would
like HBase to be extendible enough that it's easy to add new query
languages or primitives.
```

In contrast, we can find property decisions in some of the smallest emails that are made in rebuttal to claims about what a system should or shouldn't do, like in the following example.

```
Hadoop 3693 is, btw, how archives got implemented for 18.  As the spec at
the beginning says, compression is not a goal.
```

Analysis

Once a reasonable sufficient number of emails were categorized, a set of analyses was developed to help illustrate the results, and how they pertain to the research questions. As mentioned above, the dataset we operate on contains 1919 unique categorized emails, from 188 different email threads. For more information on how these analyses are implemented, and how to run it yourself, see the section on the Email Dataset Report Generator.

Relevance

For multiple metrics in the analysis, we discuss an email or email thread's *relevance*. This is a measure of how valuable such a result is, where those with high amounts of architectural knowledge should have a higher relevance than those who don't. And any email or thread containing no architectural knowledge should certainly have a relevance value of 0.

For individual emails, we simply take an email's relevance value as 1 if it is categorized as containing at least one type of architectural design decision, or a value of 0 otherwise.

Let e be an email, and $|e|$ be the number of architectural tags applied to email e. Then, we can compute the relevance $rel(e)$ as follows:

$$rel(e) = \begin{cases} 1 & |e| > 0 \\ 0 & \text{otherwise} \end{cases} \qquad \text{(Email Relevance)}$$

For an entire email thread, the approach is more involved, as threads can contain wildly different amounts of architectural knowledge. We use a nuanced approach to computing relevance which incorporates both the thread's total amount of architectural knowledge, and the *density* of knowledge in the thread. A thread's (Email Thread Density) is simply the ratio of architectural emails in the thread, to the total number of emails. This way, we avoid the pitfalls of overvaluing large threads with a lot of sparse knowledge, while providing extra motivation for finding dense threads with a large amount of architectural design decisions. Let $|t_{ak}|$ be the total number of emails in a thread which contain at least one architectural tag, and $|t|$ be the total number of all emails in the thread, architectural or not.

$$d_t = \frac{|t_{ak}|}{|t|} \qquad \text{(Email Thread Density)}$$

When incorporating a measure describing the thread's total amount of architectural knowledge, we compute first the total number of tagged emails in every thread in the dataset, and then take the third-quartile value as our t_{Q3} against which each thread's total amount of tagged emails can be normalized. Then we can define the thread's normalized ”total knowledge” value.

$$t_n = min\Big(1, \frac{|t_{ak}|}{t_{Q3}}\Big) \qquad \text{(Email Thread Total Knowledge)}$$

We use both the thread's density d_t, and its normalized total knowledge t_n as a linear combination to produce a relevance metric for a thread.

$$rel_t = \frac{d_t + t_n}{2} \qquad \text{(Email Thread Relevance)}$$

Lucene Search Precision

In order to answer the question of *how effective* Lucene indexing and searching is to find relevant architectural knowledge, two metrics were used.

First, a simple iterative precision metric is used to determine a search query's precision P_n for a search to find n emails or threads where rel_i is the relevance of the i^{th} email or thread:

$$P_n = \sum_{i=1}^{n} \frac{p_i}{n} \text{ where } p_i = \begin{cases} 1 & rel_i > 0 \\ 0 & rel_i <= 0 \end{cases} \qquad \text{(Search Precision)}$$

This metric tells what ratio of results returned by a query contain some architectural knowledge, so it can be of use to describe how effective a search is at returning *anything* of value, but does not account for the rank of elements in the search, nor the true relevance of email threads. This equation works for analyzing searches over individual emails, and searches over threads by obtaining rel_i using either (Email Relevance) or (Email Thread Relevance), respectively.

Normalized Discounted Cumulative Gain

A better metric that accounts for a result's *rank* in addition to its relevance is the Discounted Cumulative Gain, and this helps us to model real-world user persistence in examining large result sets[14].

$$DCG_n = \sum_{i=1}^{n} \frac{rel_i}{log_2(i+1)} \qquad \text{(DCG)}$$

However, this metric may be wildly different depending on the query that's used. To compensate for this, we *normalize* the cumulative gain by comparing the DCG to the best possible result that could be obtained, also known as the *ideal* DCG[12]. Let REL_n be the list of the n most relevant documents in the dataset, ordered in descending order. Then we can compute the ideal DCG as follows:

$$IDCG_n = \sum_{i=1}^{|REL_n|} \frac{rel_i}{log_2(i+1)} \tag{IDCG}$$

And once the IDCG is obtained, we can simply use this to normalize the DCG result produced by any query over the dataset.

$$NDCG_n = \frac{DCG_n}{IDCG_n} \tag{NDCG}$$

In the end, we can use NDCG as a general metric to evaluate the effectiveness of a search query to produce *highly relevant* results, with the *most relevant results first*. In theory, we would like a perfect query for which the NDCG is computed as 1 for all n[4].

N-Gram Patterns

Besides evaluating the effectiveness of the various query searches, we'd also like to answer questions about the content of architectural knowledge, and identifying patterns in how such knowledge appears can be key to understanding this. To that end, inspiration was taken from the N-Gram method for analyzing natural language. N-gram models have been used successfully in machine translation, speech recognition, entity detection, and information extraction[9]. The n-gram approach can be applied to an email thread to find the most common ordered n length sequences of tags that appear in the dataset. Furthermore, Guthrie et al. demonstrated that *skip-grams* (a technique where unmarked elements can be skipped) are effective at finding meaningful value from datasets without needing to increase the size of the dataset[11].

Therefore, we present a recursive algorithm for finding a list of all known sequences of email ids in a thread that correspond to one of many possible tag patterns, which can optionally be configured to skip non-architectural emails (generating a skip-gram model).

First, the `patternSearchRecursive` method is defined to iterate over all tags on a given email, and for each tag, find any patterns that start with that tag. For any of those patterns, call the `findMatchingSequence` method and store the results in our `result` instance variable.

The `findMatchingSequence` method will then recursively traverse an email thread from a starting email, until it finds a sequence that matches the given pattern of tags.

Co-Occurrence

N-grams are excellent at giving insight into the ordered patterns of architectural knowledge discussion, but a simpler co-occurrence metric can tell us what types of architectural knowledge most often appear together. For this, we just make a pass over all categorized emails, and increment a counter for each combination of tags that is encountered.

Aggregate Data

In addition to the above analyses, we will also provide a suite of simple aggregate analyses to be able to better visualize the data. This will include the following properties for each individual email:

- Body size: the total number of characters of the email's body.
- Word count: the total number of words in the email's body. Words are any sequence of text separated by one or more whitespace characters.

And for email threads, the following additional properties will be examined:

- Thread size: the total number of emails in a thread.
- Participant count: the total number of all unique participants in a thread.

Results

In this study, 1919 individual emails, in 188 unique email threads, were categorized according to the architectural design decisions they contained. Furthermore, this set contains the top 75 email threads from each of the four provided Lucene search queries. We will now illustrate the results of this effort, and attempt to substantiate answers to the posed research questions.

Note that all data, plots, and other graphics that were programmatically generated, have their source code available on GitHub.com.

Kinds of Architectural Design Decisions

To answer the first and second research questions, we can look at the total number of emails (and threads) that have been categorized under each type of architectural design decision.

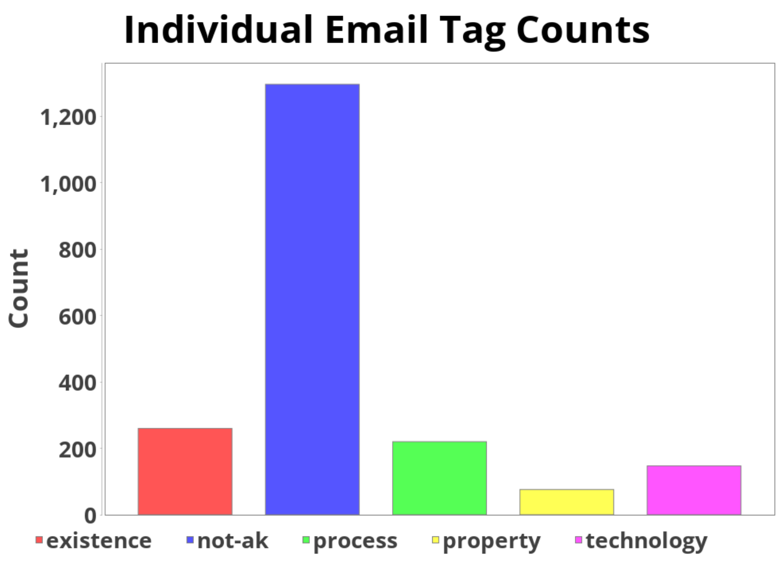


Figure 11: The total number of emails with each type of tag.

From figure 11 we can clearly see that the majority of emails contain no architectural knowledge at all, and by a large margin. However, we also see some interesting variation in what kinds of architectural design decisions do exist, with existence being the most common, followed closely by process decisions.

Looking at the results for threads in figure 12, the outlook isn't quite so depressing. Despite the fact that yet again, *most* threads generally don't have architectural knowledge, a large portion of threads *do* contain a significant number of identified design decisions. Again, existence decisions are the most popular, but thereafter it is evident that there is a relatively large number of threads discussing technology decisions, while fewer process discussions than one would expect if extrapolating the individual email data.

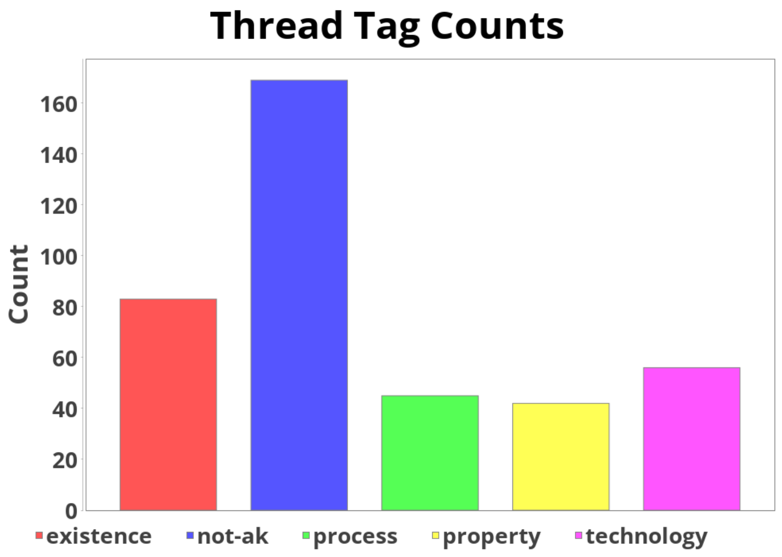


Figure 12: The total number of threads that contain each at least one email with a given tag.

The reduced frequency of process discussions at the thread level, juxtaposed with the increased frequency at the individual email level, indicates that there exists a fewer number of threads discussing development processes, *but* those threads which *do* discuss processes, generally contain more emails. We can see this difference illustrated further in figure 14, which plots the distribution of thread sizes for threads containing various tags.

Conversely, there are proportionately more threads discussing technology than there are individual emails, and again this can be explained by the nature of such discussions. Conversations about technologies tend to be shorter than other conversations about the actual architecture of a system, probably because there is a lesser need to reason about and qualify decisions compared to the system's own components.

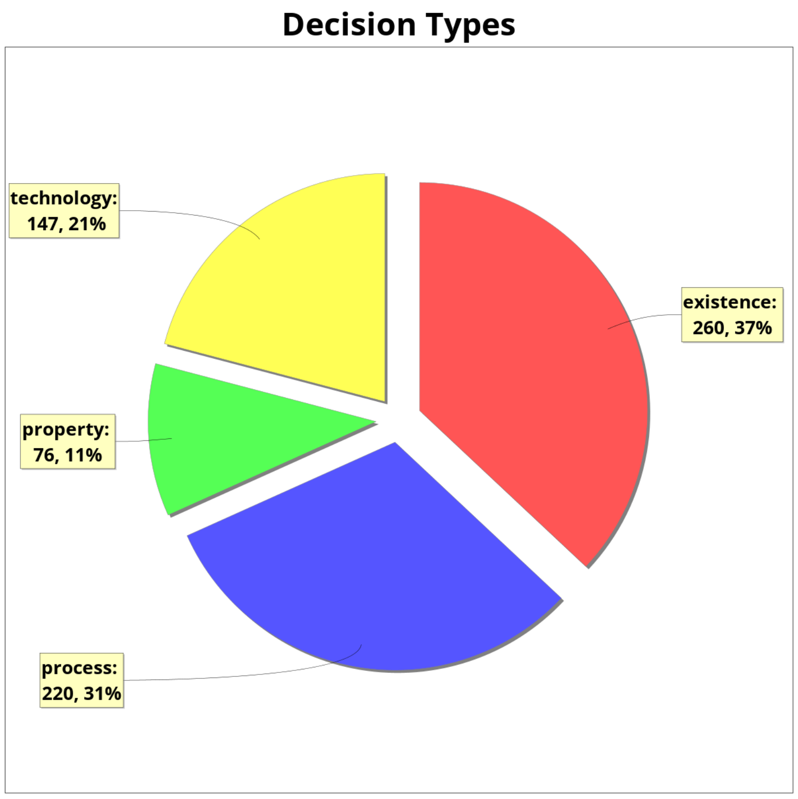


Figure 13: The relative frequency of the different types of design decisions.

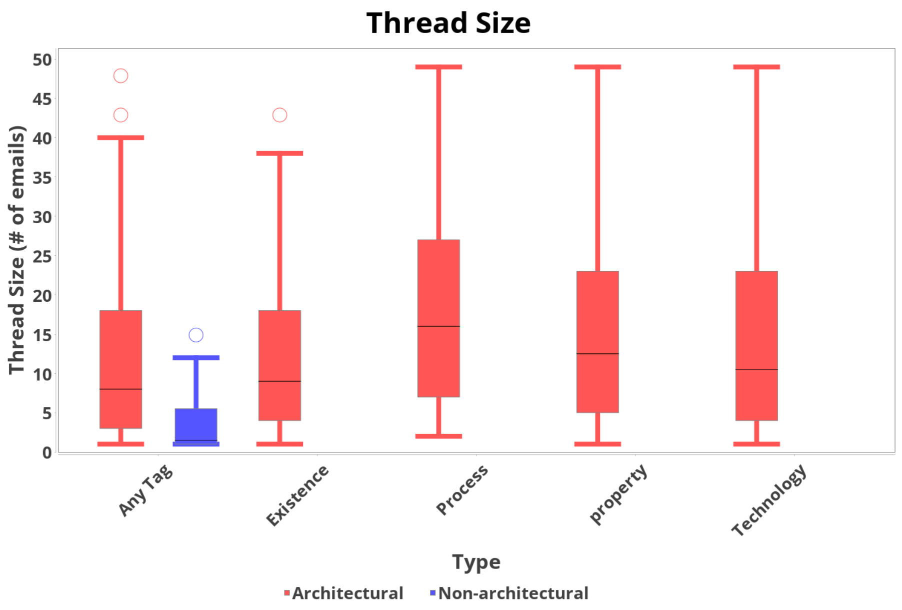


Figure 14: Boxplot showing the distribution of total thread size for threads containing different tags. A thread is placed into one of the categories if it contains at least one email with that tag.

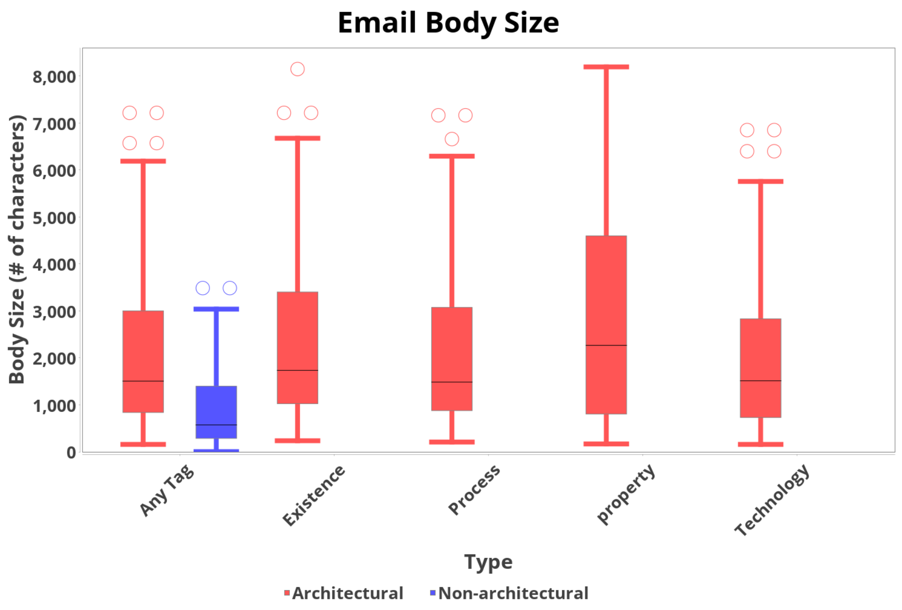


Figure 15: Boxplot showing the distribution of body sizes of emails for threads containing different tags. A thread is placed into one of the categories if it contains at least one email with that tag.

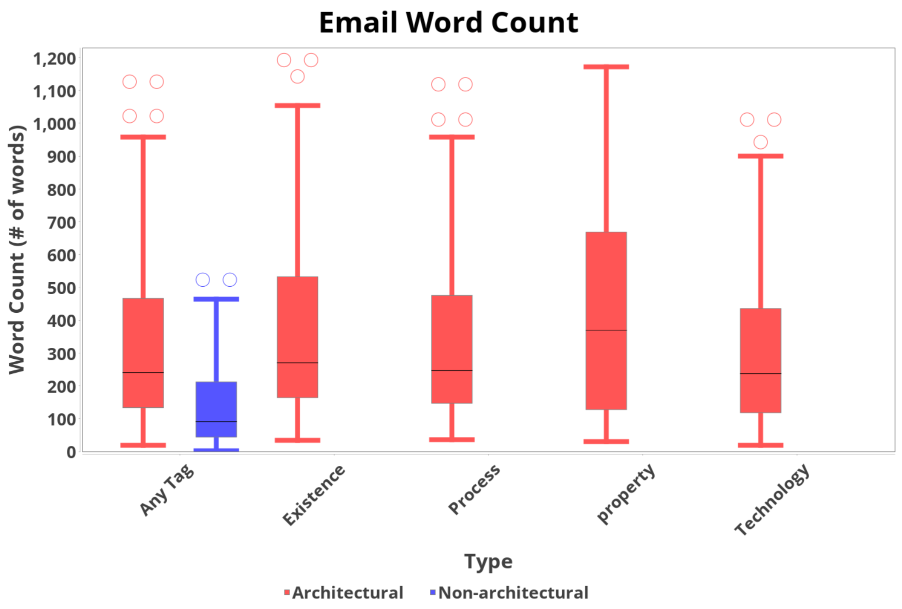


Figure 16: Boxplot showing the distribution of word counts of emails for different tags.

The (Email Thread Relevance) equation provides some insight into how often architectural knowledge appears. We observe a roughly linear gradient from a relevance of 0.1 to a relevance of 1.0, and again we see that a plurality of email threads are mostly irrelevant in terms of their architectural knowledge content.

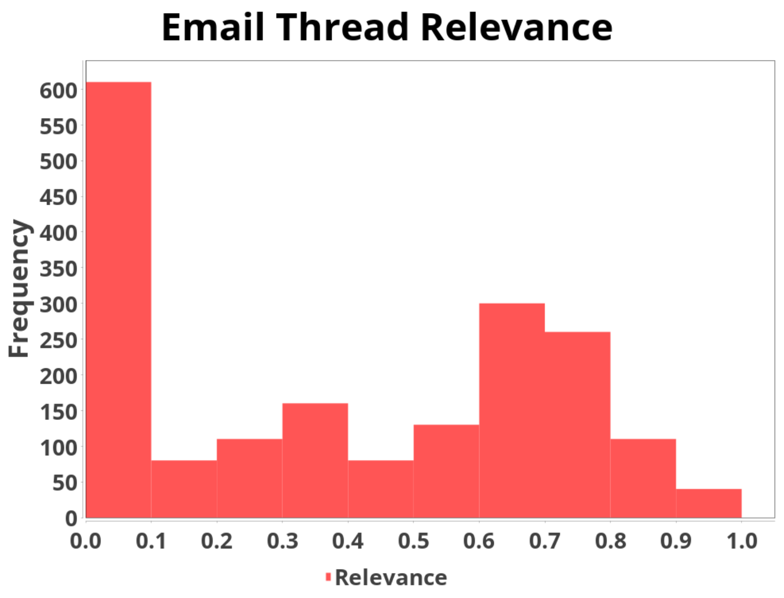


Figure 17: Histogram that shows the distribution of the relevance of all email threads.

Patterns of Design Decisions

To determine if and what sort of patterns exist in progression of architectural design decisions through an email thread, a series of n-gram and co-occurrence analyses were performed. We will first discuss what patterns exist, and then consult a few cases to try and understand why those patterns exist.

Table 1: All 2-gram sequences, ordered from most common to least common.

Pattern	**Count**
property → existence	18
existence → process	17
technology → existence	17
existence → technology	16
existence → property	15
property → technology	14
process → existence	11
process → technology	11
property → process	10
technology → process	9
technology → property	6
process → property	4

Table 2: All 2-gram sequences, ordered from most common to least common, skipping non-architectural emails between tags.

Pattern	**Count**
existence $\rightarrow$ process	58
existence $\rightarrow$ property	55
technology $\rightarrow$ existence	54
existence $\rightarrow$ technology	49
property $\rightarrow$ existence	43
technology $\rightarrow$ process	41
process $\rightarrow$ technology	40
process $\rightarrow$ existence	35
property $\rightarrow$ process	27
property $\rightarrow$ technology	25
technology $\rightarrow$ property	21
process $\rightarrow$ property	15

Already with just the 2-gram model, it is clear that there is a sort of bidirectional relationship between property and existence decisions, and between existence and technology decisions. Furthermore, the model indicates that there is not a strong directionality in the patterns, since most pairs have a similar frequency with both orientations. The skip-gram technique does lead to more abundant results, but there may be a slightly reduced accuracy[11], in that an arbitrary number of intermediate emails have been skipped, thus the application of a skip-gram model may not be as reliable as the non-skip, but seems to perform better without needing a larger dataset.

Table 3: Top 5 3-gram sequences, ordered from most common to least common. Other sequences omitted due to low frequency.

Pattern	**Count**
property $\rightarrow$ existence $\rightarrow$ process	4
existence $\rightarrow$ property $\rightarrow$ technology	3
technology $\rightarrow$ existence $\rightarrow$ process	2
property $\rightarrow$ technology $\rightarrow$ existence	2
property $\rightarrow$ technology $\rightarrow$ process	2

Table 4: All 2-gram sequences, ordered from most common to least common, skipping non-architectural emails between tags.

Pattern	**Count**
technology → existence → process	24
existence → process → property	21
existence → property → process	19
existence → technology → process	19
technology → process → existence	19
technology → property → existence	16
existence → technology → property	14
property → existence → process	14
existence → property → technology	13
property → technology → existence	12
technology → existence → property	11
process → technology → existence	10
property → existence → technology	9
existence → process → technology	9
property → technology → process	8
technology → property → process	7
process → existence → technology	7
technology → process → property	6
property → process → existence	6
property → process → technology	3
process → property → existence	3
process → existence → property	2
process → property → technology	1
process → technology → property	0

Given that the dataset consists of nearly 2,000 categorized emails, the results of the strict 3-gram model show that there is no abundant evidence for any highly structured patterns of discussion. The skip-gram model offers some more information, but again, the frequencies are quite low to be able to definitively state that a pattern exists. A much larger sample would be needed. Despite that, we do see that technology and existence decisions tend to occur at the beginning of email discussions, while process and property tend to follow.

Table 5: Co-occurrence frequency for pairs of tags.

Pattern	**Count**
existence, property	31
existence, technology	25
existence, process	14
process, technology	9
property, technology	8
process, property	8

The co-occurrence data shown in figure 5 reinforces the previous claim about a relationship between the existence and property decisions, and existence and technology decisions. Now that it is clear that such relationships do exist, we will discuss the possible motivation for the existence of these relationships.

Regarding the existence $\leftrightarrow$ property relationship, several cases indicate that this happens when a participant begins a high-level proposal or discussion on adding a new component, or integrating a subsystem into the main architecture. This tends to initiate a conversation with one or more property decisions, as users and developers debate on the existing and new properties of the system, and once this is resolved, concrete components or connections between components are proposed.

Example A

An initial email proposes a new implementation of a memtable for the Cassandra project.

```
We would like to contribute our TrieMemtable to Cassandra.

This is a new memtable solution aimed to replace the legacy implementation, developed
↪  with the following objectives:
- lowering the on-heap complexity and the ability to store memtable indexing structures
↪  off-heap,
- leveraging byte order and a trie structure to lower the memory footprint and improve
↪  mutation and lookup performance.

The new memtable relies on CASSANDRA-6936 to translate to and from byte-ordered
↪  representations of types, and CASSANDRA-17034 / CEP-11 to plug into Cassandra. The
↪  memtable is built on multiple shards of custom in-memory single-writer
↪  multiple-reader tries, whose implementation uses a combination of state-of-the-art
↪  and novel features for greater efficiency.
```

This first email is an example of one where existence and property decisions happen together: in this case, the new memtable implementation is a structural change to the architecture, and provides several features that allow better qualitative properties of Cassandra. In a response, we see more existence decisions as the details of the proposed implementation are explained (as these details often imply structural changes to the architecture).

```
The memtable pluggability API (CEP-11) is per-table to enable memtable selection
↪  that suits specific workflows. It also makes full sense to permit per-node
↪  configuration, both to be able to modify the configuration to suit heterogeneous
↪  deployments better, as well as to test changes for improvements such as this one.
Recognizing this, the patch comes with a modification to the API that defines memtable
↪  templates in cassandra.yaml (i.e. per node) and allows the schema to select a
↪  template (in addition to being able to specify the full memtable configuration).
↪  One could use this e.g. by adding:
memtable_templates:
  trie:
    class: TrieMemtable
    shards: 16
  skiplist:
    class: SkipListMemtable
memtable:
  template: skiplist
(which defines two templates and specifies the default memtable implementation to use)
↪  to cassandra.yaml and specifying  WITH memtable = {'template' : 'trie'} in the
↪  table schema.
```

Example B

In a thread titled *Hadoop Summit: Security Design Lounge Session*, the first email begins by summarizing the attendance and main takeaways from a recent Hadoop Summit meetup, and intends to continue discussion about Hadoop's security in the thread. The author guided the discussion primarily to how to improve Hadoop's security. Already, this identification of points of improvement for the system and preliminary discussion includes property and existence decisions.

```
...
In order to keep the scope of conversations manageable we tried to keep focused on
↪  authentication and the ideas around SSO and tokens.

We discussed kerberos as:
1. major pain point and barrier to entry for some
2. seemingly perfect for others
a. obviously requiring backward compatibility

It seemed to be consensus that:
1. user authentication should be easily integrated with alternative enterprise identity
↪  solutions
2. that service identity issues should not require thousands of service identities
↪  added to enterprise user repositories
3. that customers should not be forced to install/deploy and manage a KDC for services
↪  - this implies a couple options:
  a. alternatives to kerberos for service identities
  b. hadoop KDC implementation - ie. ApacheDS?
...
```

One response to this email provided more support for modifying Hadoop's authentication mechanisms to support pluggable authentication providers, and mentions a third-party authentication provider (ApacheDS) as one example of how it could work.

```
There was certainly discussions around the emerging work from Daryn related to
↪  pluggable authentication mechanisms within that layer and we will immediately have
↪  the options of kerberos, simple and plain. There was also talk of how this can be
↪  leveraged to introduce a Hadoop token mechanism as well.

At the same time, there was talk of the possibility of simply making kerberos easy and
↪  a non-issue for intra-cluster use. Certainly we need both of these approaches.
I believe someone used ApacheDS' KDC support as an example - if we could standup an
↪  ApacheDS based KDC and configure it and related keytabs easily than the end-to-end
↪  story is more palatable to a broader user base. That story being the choice of
↪  authentication mechanisms for user authentication and easy provisioning and
↪  management of kerberos for intra-cluster service authentication.
```

In the other common bidirectional case with existence ↔ technology, we see a similar pattern of a proposal followed by a rich discussion. For this relationship, however, proposals often directly or indirectly mention third-party technologies, and ensuing conversation includes a series of emails that evaluate the value of integrating such technologies into the architecture.

Example C

In this example, we look at a discussion surrounding the replacement of some old CLI framework used by various Cassandra tools. The current framework (airlift/airline) is deprecated and this email thread is continuing a conversation started in a Slack (instant messaging service) channel.

```
In CASSANDRA-17445 we’ve started discussing the options of replacing the deprecated
↪  airlift/airline framework used in CLI tools.

Considering the amount of commands this framework is used in, the impact this might
↪  cause and the future possibilities the operational aspects of Cassandra could
↪  leverage, first comments at slack revealed an in-depth discussion would be
↪  desirable.

Kind request for comments.
```

The initial message simply expresses a desire to replace the CLI framework, but doesn’t provide any specific solutions. This intention to refactor/replace the CLI framework is already an existence decision, as it will generally mean adding and removing some structural components to facilitate a new CLI dependency. Replies go into more detail about actual technologies that can be used to replace the airlift/airline framework.

```
Maintaining some other tools, originally inspired by nodetool, I started evaluating
↪  alternatives. Started with airline2 and picocli, my preliminary findings are

airline2:
+ low effort to replace, nearly unchanged API
+ capability to generate documentation

picocli:
- considerable effort for migration needed
+ more modern and lean API
+ capability to generate documentation
+ capability to generate shell-completion, including not just subcommands but all
↪  options and parameters
+ capability to generate man pages

In my opinion, there is room for improvement, especially on documentation and the
↪  usability of tools:

* more detailed descriptions on commands and subcommands
* better accessibility to help and docs, e.g. on the website, as man pages, etc.
* adding usage examples to docs and help
* doc pages on the website could be generated e.g. as a side-product of the release
↪  process
* the command completion script could be generated, without the need for manual
↪  maintenance and packaging
* coloured output in a terminal would be appreciated, especially by less experienced
↪  users, e.g. when looking at nodetool status on larger clusters
```

The above reply provides one of the most typical examples of an excellent comparison of technologies, in this case CLI frameworks to use for Cassandra's tools. It also contains a slight property decision, by advocating for the improvement of the tools' usability.

```
airline/airlift is deprecated. I suspect if there were any security issues they would
↪  not be fixed. Their project recommends moving to Airline 2 or picocli.

I share Stefan's concern about the stability of the CLI and output formatting. We
↪  should avoid any breakages resulting from this migration. Lots of automation
↪  depends on this "interface" being stable.
```

And finally, this reply simply provides a little bit more detail into the motivation behind the desired change.

Example D

In another example, we look at a proposal by a developer for opening Hadoop up to alternative authentication methods beyond just Kerberos (this relates to example B). The initial email provides a list of motivations for why developers should want to do this, and proceeds to ask for community feedback and ideas.

```
Kerberos was developed decade before web development becomes popular.
There are some Kerberos limitations which does not work well in Hadoop.  A
few examples of corner cases:

1. Kerberos principal doesn't encode port number, it is difficult to know
if the principal is coming from an authorized daemon or a hacker container
trying to forge service principal.
2. Hadoop Kerberos principals are used as high privileged principal, a form
of credential to impersonate end user.
3. Delegation token may allow expired users to continue to run jobs long
after they are gone, without rechecking if end user credentials is still
valid.
4.  Passing different form of tokens does not work well with cloud provider
security mechanism.  For example, passing AWS sts token for S3 bucket.
There is no renewal mechanism, nor good way to identify when the token
would expire.

There are companies that work on bridging security mechanism of different
types, but this is not primary goal for Hadoop.  Hadoop can benefit from
modernized security using open standards like OpenID Connect, which
proposes to unify web applications using SSO.   This ensure the client
credentials are transported in each stage of client servers interaction.
This may improve overall security, and provide more cloud native form
factor.  I wonder if there is any interested in the community to enable
Hadoop OpenID Connect integration work?
```

Another developer responds with more specific information about how they have experience with OpenID Connect (OIDC) authentication.

```
I'm interested in OpenID Connect (OIDC) integration.

In addition to the benefits (security, cloud native), operating costs may
be reduced in some companies.
We have our company-wide OIDC provider and enable SSO for Hadoop Web UIs
via Knox + OIDC in Yahoo! JAPAN.
On the other hand, Hadoop administrators have to manage our own KDC servers
only for Hadoop ecosystems.
```

```
If Hadoop and its ecosystem can support OIDC, we don't have to manage KDC
and that way operating costs will be reduced.
```

This reply provides another motivation for adding support for alternative authentication mechanisms, but mostly motivated by cost optimization rather than just intrinsic benefit of the architecture. This is a common occurrence in technology emails, where respondents qualify their opinions based on the costs of different choices on their industry.

```
We aren't very far from having OIDC support.  The pre-requisite RPC/TLS &
RPC/mTLS recently completed rollout to our entire production grid.
Majority of the past year was spent shaking out bugs and ensuring 100%
compatibility.  There are a few rough edges I need to clean up for a
community release.

A few weeks ago I created a rough POC to leverage RPC/mTLS with OIDC access
tokens.  Goal is a mTLS cert may be blessed to impersonate with an access
token.  A compromised service may only be abused to impersonate users that
have recently accessed said service.

Kerberos, mTLs, and OIDC may all be simultaneously supported.  Part of the
simplicity is regardless of the client's authn/authz, delegation tokens are
still acquired by jobs to avoid short-lived identity credential expiration.
```

And generally, we see multiple emails of a similar nature in technology discussions, where respondents provide their perspective on how they use a technology in their organization, and in the above case, some details about the implementation or how it can be used.

Search Effectiveness

To determine the effectiveness of our four queries at finding relevant emails and threads rich in architectural design decisions, we will use both the precision and NDCG as metrics to evaluate over each query.

From figure 18 we can see immediately that all queries generally perform quite well in terms of their precision. Recall that the precision is a measure of the average ratio of returned results that contain architectural knowledge, meaning that all four queries perform slightly above 50%. Initially, the *decision factors* query performs better, followed by the *rationale* query from $n = 7$ onward. The *reusable solutions* and *components and connectors* queries in contrast perform relatively poorly in the initial results, but precision improves as n approaches 20. However, it should be noted that the initial performance is highly unstable and slight variations in the content of emails may completely alter the first few results, and so practically we should only consider evaluating a query based on its total precision over a greater n.

In figure 19, the NDCG metric is visualized for the four queries. The graph is less chaotic than the precision graph, and we can see a more organized trend line for three of queries, while the *reusable solutions* performs consistently at

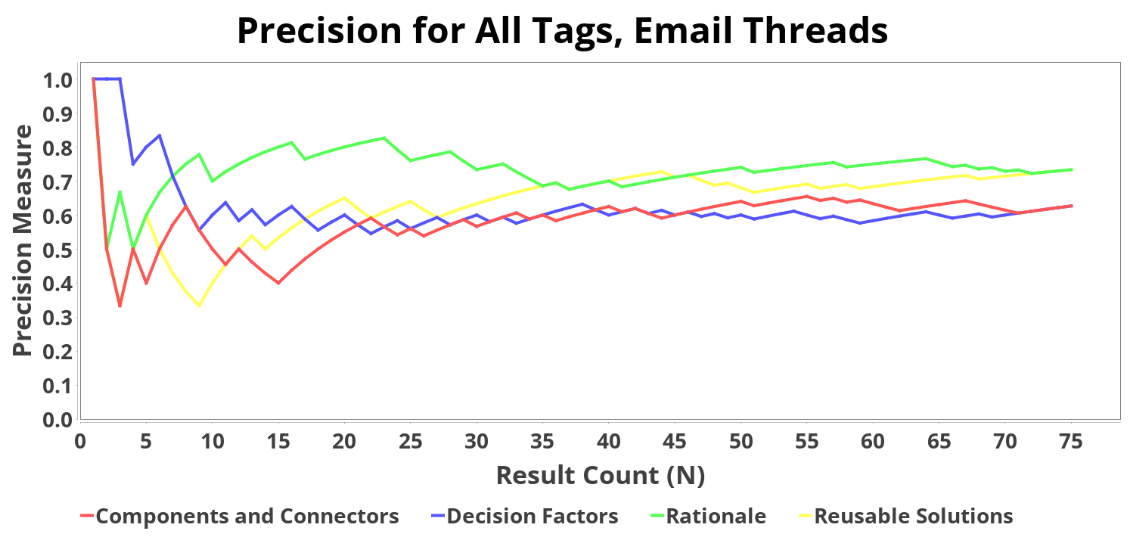


Figure 18: Precision metric with thread relevance computed using all architectural design decision tags, for all queries.

the lowest level. Again, *decision factors* and *rationale* perform well initially, and in this metric, they continue to provide the best performance, albeit by a tiny margin. Despite the fact that the NDCG metric ”discounts” results at lower ranks in the result, the queries appear to improve in performance slightly as the result set grows larger. This could indicate that the factor by which we discount lower-ranked results is too weak, or that the queries are simply not effective at ranking highly-relevant results appropriately.

While the focus in this study is on evaluating the effectiveness of queries in a real-world context that's similar to how a user would really search for knowledge, we also include precision and NDCG measures when using the queries to search over individual emails, to compare with the thread-based search. This type of search benefits from a larger dataset volume, as we have roughly 10 times as many fully categorized emails as threads, but does not accurately depict a typical human search, since it removes context from the search results by returning fragmented bits of information from many different conversations. Generally the searches perform worse over individual emails than they do over threads, with both precision and NDCG values remaining below 0.4 for almost the entirety of the query results, and degrading as n increases. However, it appears that the *reusable solutions* query, which performed most poorly in the thread-based precision and NDCG metrics, tends to perform at or near the best of all the queries for individual emails.

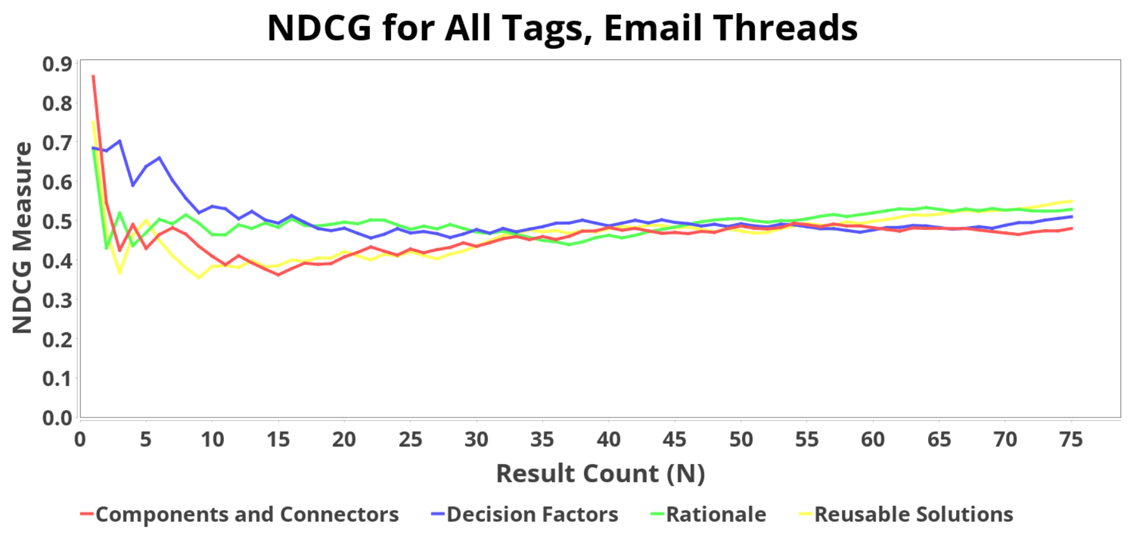


Figure 19: NDCG metric with thread relevance computed using all architectural design decision tags, for all queries.

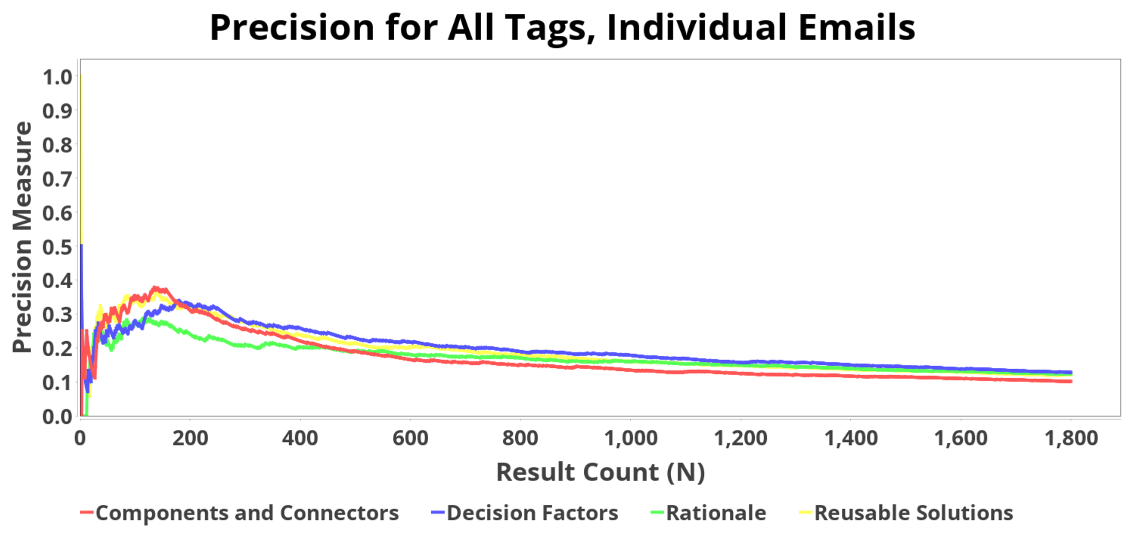


Figure 20: Precision metric with email relevance computed using all architectural design decision tags, for all queries.

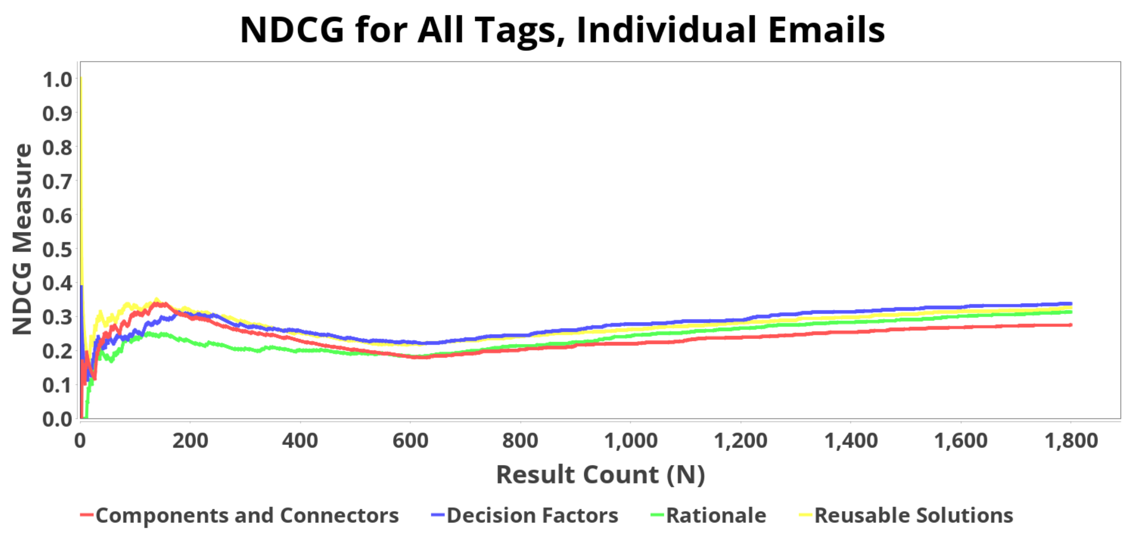


Figure 21: NDCG metric with email relevance computed using all architectural design decision tags, for all queries.

Conclusion

In this study, we prepared tools to fetch and process mailing lists, and prepared a categorized dataset containing 1919 emails in 188 unique email threads, to determine what types of architectural knowledge exists in open-source software development mailing lists. The results of a varied set of analyses over this dataset tell us first and foremost that *most* emails in such mailing lists do not contain useful architectural design decisions, but that a large proportion of email threads *do* contain architectural design decisions. There exists a strong relationship between existence decisions and both technology and property decisions, as a natural consequence of the flow of a conversation from proposal to an evaluation of alternative solutions, to choosing an accepted solution.

From applying Apache Lucene search queries to the dataset, the precision and NDCG metrics for both thread-based and individual email results indicate that the current search implementation is not effective for finding a sufficient amount of architectural knowledge, when compared to similar metrics performed by den Boon[4] where queries achieved NDCG measures of greater than 0.9 over 100 results.

Open-source software development mailing lists, as shown by this research, present ample opportunity for extracting architectural knowledge, and we have shown that contextual patterns in the types of design decisions can be used to further aid in searching. However, the search methods used in this study, or the metrics used to grade them, were insufficient to conclude that keyword-based Lucene queries are an effective tool for extracting that knowledge. More work is needed to improve how the queries operate over email threads, how to compute a more accurate relevance measure, and improved metrics are needed to be able to better differentiate the performance of different queries.

Threats to Validity

Despite the researcher's best efforts, the data in this paper could be subject to a number of faults, and it is important that the reader acknowledge these before pressing onward with any research that depends on these results.

The researcher is a single student whose practical experience does not overlap much with the realm of architectural knowledge identification and extraction. The researcher did iteratively develop a sense of intuition for the design decisions through many weeks of supervised email categorization, but this is still prone to the occasional error. Fatigue may play a role in producing lower-quality results, as multiple hours of reading emails tends to drain the mind of all its vitality.

The indexing and searching implementation may not be optimal for email threads, as no general solution exists for this, given that mailing lists are usually navigated chronologically by their users. The Email Thread Relevance equation may not be optimal for computing a truly useful relevance value for an email thread.

Future Work

This study provides a solid basis for identifying the types and amount of architectural knowledge that can be found in open-source software development mailing lists, but is far from an exhaustive resource on the matter. Additional research is needed to further expand on the data with additional mailing lists from other open-source communities that focus on different types of software. While we explored the patterns in how architectural design decisions are made, our research was limited to exclusively the content of the emails themselves, and there are many examples where the discussion in an email is migrated to a Jira or GitHub issue tracker, and the reasons for these ”jumps” between media has not yet been explored, nor the patterns which might exist between discussions on either end.

While more work can be done to improve the indexing and searching implementation, and the queries used by it, research by Bhat et al.[3] shows that there is promise in designing machine learning models to automatically extract architectural knowledge. The data in this study can be incorporated into training data for such models.

The tools developed for this study are rudimentary in nature, but have been developed to the best of the researcher’s abilities to be future-proof and open to expansion and re-use, and we encourage others to build on their capabilities. Currently, the EmailDownloader utility supports only Apache Software Foundation mailing lists, but could be expanded easily to support others as well. The EmailIndexer utility currently only supports the H2 database implementation, which offers limited concurrency when operating in embedded mode, and this can be improved to allow for multi-user systems where many people can contribute to a single dataset’s information at once. And finally, the EmailDatasetBrowser application could be expanded to include real-time analytics and improved search interfaces.

References

[1] URL: https://lucene.apache.org/.

[2] Aleem Akhtar. "Role of Apache Software Foundation in Big Data Projects". In: *arXiv e-prints* (May 2020). DOI: 10.48550/arXiv.2005.02829.

[3] Manoj Bhat et al. "Automatic Extraction of Design Decisions from Issue Management Systems: A Machine Learning Based Approach". In: *Software Architecture*. Ed. by Antónia Lopes and Rogério de Lemos. Springer International Publishing, 2017, pp. 138–154. ISBN: 978-3-319-65831-5.

[4] Tom den Boon. "Exploring the effectiveness of search engines for finding architectural knowledge in open source repositories". In: *University of Groningen Student Theses* (2021), pp. 3–30. URL: https://fse.studenttheses.ub.rug.nl/25813/.

[5] Walter Bright, Andrei Alexandrescu, and Michael Parker. "Origins of the D Programming Language". In: *Proc. ACM Program. Lang.* 4.HOPL (2020). DOI: 10.1145/3386323. URL: https://doi.org/10.1145/3386323.

[6] Wei Ding et al. "Understanding the Causes of Architecture Changes Using OSS Mailing Lists". In: *International Journal of Software Engineering and Knowledge Engineering* (2015), pp. 1633–1651.

[7] Said Faroghi. "Mining architectural knowledge in issue tracking systems". In: *University of Groningen Student Theses* (Feb. 2022). URL: https://fse.studenttheses.ub.rug.nl/26603/.

[8] W.N. Francis and H. Kučera. "Frequency Analysis of English Usage: Lexicon and Usage". In: *Oxford University Press* (1982).

[9] Alex Franz and Thorsten Brants. "All Our N-gram are Belong to You". In: *Google AI Blog* (Aug. 2006). URL: https://ai.googleblog.com/2006/08/all-our-n-gram-are-belong-to-you.html.

[10] R. Gellens. *Mailing Lists and Internationalized Email Addresses*. Internet Engineering Task Force (IETF). Oct. 2010.

[11] David Guthrie et al. "A Closer Look at Skip-gram Modelling". In: (2006). URL: https://web.archive.org/web/20170517144625/http://homepages.inf.ed.ac.uk/ballison/pdf/lrec_skipgrams.pdf.

[12] *Introduction to Information Retrieval*. Apr. 2013. URL: https://web.stanford.edu/class/cs276/handouts/EvaluationNew-handout-6-per.pdf.

[13] Anto Jansen et al. "Software Architecture as a Set of Architectural Design Decisions". In: *Proceedings - 5th Working IEEE/IFIP Conference on Software Architecture, WICSA 2005* 2005 (Jan. 2005), pp. 109–120. DOI: 10.1109/WICSA.2005.61.

[14] Kalervo Järvelin and Jaana Kekäläinen. "Cumulated Gain-Based Evaluation of IR Techniques". In: *ACM Trans. Inf. Syst.* 20.4 (2002), pp. 422–446. ISSN: 1046-8188. DOI: 10.1145/582415.582418. URL: https://doi.org/10.1145/582415.582418.

[15] Philippe Kruchten, Patricia Lago, and Hans van Vliet. “Building Up and Reasoning About Architectural Knowledge”. In: *Quality of Software Architectures* (Dec. 2006), pp. 43–58. DOI: 10.1007/11921998_8.

[16] Peter Naur and Brian Randell. “Software Engineering: Report on a conference sponsored by the NATO Science Committee”. In: *Working Conference on Software Engineering* (1969).

[17] Dewayne E. Perry and Alexander L. Wolf. “Foundations for the Study of Software Architecture”. In: *ACM SIGSOFT Software Engineering Notes* 17 (4), pp. 40–52. DOI: 10.1.1.40.5174. URL: http://users.ece.utexas.edu/~perry/work/papers/swa-sen.pdf.

[18] Pete Resnick. *Internet Message Format*. RFC 5322. Oct. 2008. DOI: 10.17487/RFC5322. URL: https://www.rfc-editor.org/info/rfc5322.

[19] Mohamed Soliman et al. “Exploring Web Search Engines to Find Architectural Knowledge”. In: *Proceedings - IEEE 18th International Conference on Software Architecture, ICSA 2021* (), pp. 162–172. DOI: 10.1109/ICSA51549.2021.00023. URL: https://doi.org/10.1109/ICSA51549.2021.00023.

[20] Antony Tang et al. “A comparative study of architecture knowledge management tools”. In: *The Journal of Systems and Software* (July 2008).

[21] Zhuang Xiong et al. “Assumptions in OSS Development: An Exploratory Study through the Hibernate Developer Mailing List”. In: *25th Asia-Pacific Software Engineering Conference (APSEC)* (Dec. 2018). DOI: 10.1109/APSEC.2018.00060.

Appendix

This section will contain larger bits of text or code or figures that aren't well suited to being placed inside the body of the paper. They may be referenced throughout the body of the paper. Note that not all source code for the various tools and analyses are included here, due to the impracticality of distributing such a large amount of source code. For the complete suite of software used in this study, please visit the ArchitecturalKnowledgeAnalysis organization on GitHub.

Lucene Keyword Queries

This section contains four keyword-based Lucene search queries provided by Mohamed Soliman, as derived from previous research for use in this project. You may consult Apache Lucene's documentation for a reference of the syntax used in these queries.

Decision Factors

```
actor* availab* budget* business case* client* concern* conform*
consisten* constraint* context* cost* coupl* customer* domain*
driver* effort* enterprise* environment* experience* factor*
force* function* goal* integrity interop* issue* latenc*
maintain* manage* market* modifiab* objective* organization*
performance* portab* problem* purpose* qualit* reliab*
requirement* reus* safe* scal* scenario* secur* stakeholder*
testab* throughput* usab* user* variability limit* time cohesion
efficien* bandwidth speed* need* compat* complex* condition*
criteria* resource* accura* complet* suitab* complianc* operabl*
employabl* modular* analyz* readab* chang* encapsulat* transport*
transfer* migrat* mova* replac* adapt* resilienc* irresponsib*
stab* toleran* responsib* matur* accountab* vulnerab* trustworth*
verif* protect* certificat* law* flexib* configur* convent*
accessib* useful* learn* understand*
```

Reusable Solutions

```
action* adapt* alloc* alternativ* approach* asynch* audit*
authentic* authoriz* balanc* ballot* beat bridg* broker*
cach* capabilit* certificat* chain* challeng* characteristic*
checkpoint* choice* cloud composite concrete concurren*
confident* connect* credential* decorat* deliver* detect*
dual* echo encapsulat* encrypt* esb event* expos* facade factor*
FIFO filter* flyweight* framework* function* handl* heartbeat*
intermedia* layer* layoff* lazy load lock* mandator* measure*
mechanism* memento middleware minut* monitor* mvc observ*
offer* opinion* option* orchestrat* outbound* parallel passwords
pattern* peer* period* piggybacking ping pipe* platform* point*
pool principle* priorit* processor* profil* protect* protocol*
prototyp* provid* proxy publish* recover* redundan* refactor*
removal replicat* resist* restart restraint* revok* rollback*
routine* runtime sanity* schedul* sensor* separat* session*
shadow* singleton soa solution* spare* sparrow* specification*
stamp* standard* state stor* strap strateg* subscrib* suppl*
support* synch* tactic* task* technique* technolog* tier* timer*
timestamp* tool* trail transaction* uml unoccupied* view* visit*
vot* wizard* worker*
```

Components and Connectors

```
access* allocat* application* architecture* artifact* attribute*
behav* broker* call* cluster* communicat* component* compos*
concept* connect* consist* construct* consum* contain* control*
coordinat* core criteria* data database* decompos* depend*
design* diagram* dynamic element* engine* entit* event* exchang*
exist* external filter* function* hardware* independ* information
infrastructure input* instance* integr* interac* internal item*
job* layer* link* load* logic* machin* memor* messag* model*
modul* node* operat* outcom* output* part* peer* platform* port*
process* produc* program* project* propert* provid* publish*
read* relat* request* resourc* respon* scope separate server*
service* shar* source* stor* structur* subscrib* support* system*
target* transaction* trigger* runtime realtime network* thread*
parallel notif* distribut* backend* frontend* central* persist*
queue* concurren* middleware* provid* suppl*
```

Rationale

```
advantag* alternativ* appropriate assum* benefit* better best
caus* choic* choos* complex* condition* critical decid* decision*
eas* evaluat* hard* quick* rational* reason* risk* simpl* strong*
tradeoff weak* rational* disadvantag* comparison* pros cons good
differen* slow* lightweight overkill* recommend* suggest* propos*
outperform* important* versus vs contrast* distinct* fast* heav*
boost* drawback* option*
```

Email Dataset Schema

The following SQL snippet defines the relational schema which is used for email datasets in version 2 of EmailIndexer API.

This DDL script is written using the H2 dialect of SQL.

```sql
CREATE TABLE EMAIL (
  ID BIGINT PRIMARY KEY AUTO_INCREMENT,
  PARENT_ID BIGINT NULL DEFAULT NULL REFERENCES EMAIL(ID)
    ON UPDATE CASCADE ON DELETE SET NULL,
  MESSAGE_ID VARCHAR(255) UNIQUE,
  SUBJECT VARCHAR(1024),
  IN_REPLY_TO VARCHAR(255),
  SENT_FROM VARCHAR(255),
  DATE TIMESTAMP WITH TIME ZONE,
  BODY LONGTEXT,
  HIDDEN BOOL NOT NULL DEFAULT FALSE,
  CHECK (PARENT_ID IS NULL OR PARENT_ID <> ID)
);
CREATE INDEX IDX_EMAIL_DATE ON EMAIL(DATE);
CREATE INDEX IDX_EMAIL_HIDDEN ON EMAIL(HIDDEN);

CREATE TABLE TAG (
  ID INTEGER PRIMARY KEY AUTO_INCREMENT,
  NAME VARCHAR(255) UNIQUE,
  DESCRIPTION MEDIUMTEXT NULL DEFAULT NULL
);
CREATE INDEX IDX_TAG_NAME ON TAG(NAME);

CREATE TABLE EMAIL_TAG (
  EMAIL_ID BIGINT NOT NULL REFERENCES EMAIL(ID)
    ON UPDATE CASCADE ON DELETE CASCADE,
  TAG_ID INTEGER NOT NULL REFERENCES TAG(ID)
    ON UPDATE CASCADE ON DELETE CASCADE,
  PRIMARY KEY (EMAIL_ID, TAG_ID)
);

CREATE TABLE MUTATION (
  ID BIGINT PRIMARY KEY AUTO_INCREMENT,
  DESCRIPTION LONGTEXT NOT NULL,
  PERFORMED_AT TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT CURRENT_TIMESTAMP(0),
  AFFECTED_EMAIL_COUNT BIGINT NOT NULL DEFAULT 0
);
CREATE INDEX IDX_MUTATION_DATE ON MUTATION(PERFORMED_AT);

CREATE TABLE MUTATION_EMAIL (
  MUTATION_ID BIGINT NOT NULL REFERENCES MUTATION (ID)
    ON UPDATE CASCADE ON DELETE CASCADE,
  EMAIL_ID BIGINT NOT NULL REFERENCES EMAIL(ID)
    ON UPDATE CASCADE ON DELETE CASCADE,
  PRIMARY KEY (MUTATION_ID, EMAIL_ID)
);
```

Key Analysis Algorithms

This section contains a selection of algorithms from the analysis tool which merit their inclusion in this paper.

N-Gram Pattern Searching

```d
/**
 * Finds a list of sequences of email ids, where the emails in those
 * sequences have tags matching the given pattern of tags, in exactly the
 * same order.
 * Params:
 *   email = The root email to search in.
 *   set = The email set.
 *   pattern = The tag pattern to search for.
 * Returns: A list of email id sequences, where each sequence contains a
 * list of email ids corresponding to emails that have tags matching the
 * given pattern.
 */
private long[][] findMatchingSequences(Email email, EmailSet set, string[] pattern) {
  import std.algorithm;
  import std.array;
  if (pattern.length == 0) return []; // Failsafe exit if the patterns are empty.
  bool thisEmailMatches = email.tags.canFind(pattern[0]);
  long[][] sequences = [];
  // An appender for appending sequences to the list.
  auto sequenceAppender = appender(&sequences);
  if (thisEmailMatches) {
    // Our base case: the root email has a matching tag.
    if (pattern.length == 1) {
      sequenceAppender ~= [email.id];
    } else {
      // Recursive step: find sequences in replies that match the rest of the pattern.
      foreach (reply; set.repliesById[email.id]) {
        foreach (sequence; findMatchingSequences(reply, set, pattern[1 .. $])) {
          sequence.insertInPlace(0, email.id);
          sequenceAppender ~= sequence;
        }
      }
    }
  } else if (skip) {
    // If we allow skipping non-architectural emails, check all replies anyways.
    foreach (reply; set.repliesById[email.id]) {
      sequenceAppender ~= findMatchingSequences(reply, set, pattern);
    }
  }
  return sequences;
}
```

Thread Relevance

This set of functions is used to compute the *relevance* of an email thread, in terms of how valuable it is for the thread to appear in search results. See (Email Thread Density) and (Email Thread Relevance)

```d
/**
 * Computes the relevance of an email thread as a linear combination of the
 * density and relative tag count metrics for the thread.
 * Params:
 *   rootEmail = The root email of the thread.
 *   set = The email set.
 *   akTags = The list of tags which are considered architectural.
 *   maxRelevance = The pre-computed "max" relevance for the set.
 * Returns: The relevance of the thread.
 */
double threadRelevance(Email rootEmail, EmailSet set, string[] akTags, double
↪  maxRelevance) {
  import std.algorithm : min;
  uint tagCount = countTagsRecursive(rootEmail, set, akTags);
  return (threadDensity(rootEmail, set, akTags) + min(1.0, tagCount / maxRelevance)) /
↪  2.0;
}

/**
 * Computes the value that we consider to be the "max relevance" of all email
 * threads in a dataset. This is the third-quartile value of the count of all
 * tags in all email threads.
 * Params:
 *   set = The email set.
 *   akTags = The list of tags which are considered architectural.
 * Returns: The max relevance.
 */
double getMaxRelevance(EmailSet set, string[] akTags) {
  import std.algorithm : sort, mean;
  uint[] rootEmailTagCounts = new uint[set.rootEmails.length];
  foreach (i, rootEmail; set.rootEmails) {
    rootEmailTagCounts[i] = countTagsRecursive(rootEmail, set, akTags);
  }
  rootEmailTagCounts.sort();
  return rootEmailTagCounts[rootEmailTagCounts.length * 3 / 4];
}

/**
 * Computes the density of an email thread. This is the ratio of architectural
 * emails to the number of total emails in the thread.
 * Params:
 *   rootEmail = The root email.
 *   set = The email set.
 *   akTags = The list of tags which are considered architectural.
 * Returns: The density of the email thread.
 */
double threadDensity(Email rootEmail, EmailSet set, string[] akTags) {
  return cast(double) countAkEmails(rootEmail, set, akTags) / threadSize(rootEmail,
↪  set);
}
```