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.
```