practice

DOI:10.1145/3486594

Article development led by acmqueue
queue.acm.org

An experience report.

BY TIMOTHY CLEM AND PATRICK THOMSON

Static Analysis at GitHub

GITHUB, A CODE-HOSTING website built atop the Git version-control system, hosts hundreds of millions of repositories of code uploaded by more than 65 million developers. The Semantic Code team at GitHub builds and operates a suite of technologies that power symbolic code navigation on github.com. Symbolic code navigation lets developers click on a named identifier in source code to navigate to the definition of that entity, as well as the reverse: given an identifier, they can list all the uses of that identifier within the project.

This system is backed by a cloud object-storage service, having migrated from a multi-terabyte sharded relational database, and serves more than 40,000 requests per minute, across both read and write operations. The static analysis stage itself is built on an open source parsing toolkit called Tree-sitter, implements some well-known computer science research, and integrates with the github.com infrastructure in order to extract name-binding information from source code.

The system supports nine popular programming languages across six million repositories. Scaling even the most trivial of program analyses to this level entailed significant engineering effort, which is recounted here in the hope that it will serve as a useful guide for those scaling static analysis to large and rapidly changing codebases.

Motivation: Seeing the Forest for the (Parse) Trees

Navigating code is a fundamental part of reading, writing, and understanding programs. Unix tools such as `grep(1)` allow developers to search for patterns of text, but programmers' needs are larger in scope: What the are most interested in is how the pieces of a program stitch together—given a function, where is it invoked, and where is it defined? Quick and quality answers to these queries allow a programmer to build up a mental model of a program's structure; that, in turn, allows effective modification or troubleshooting. Tools such as

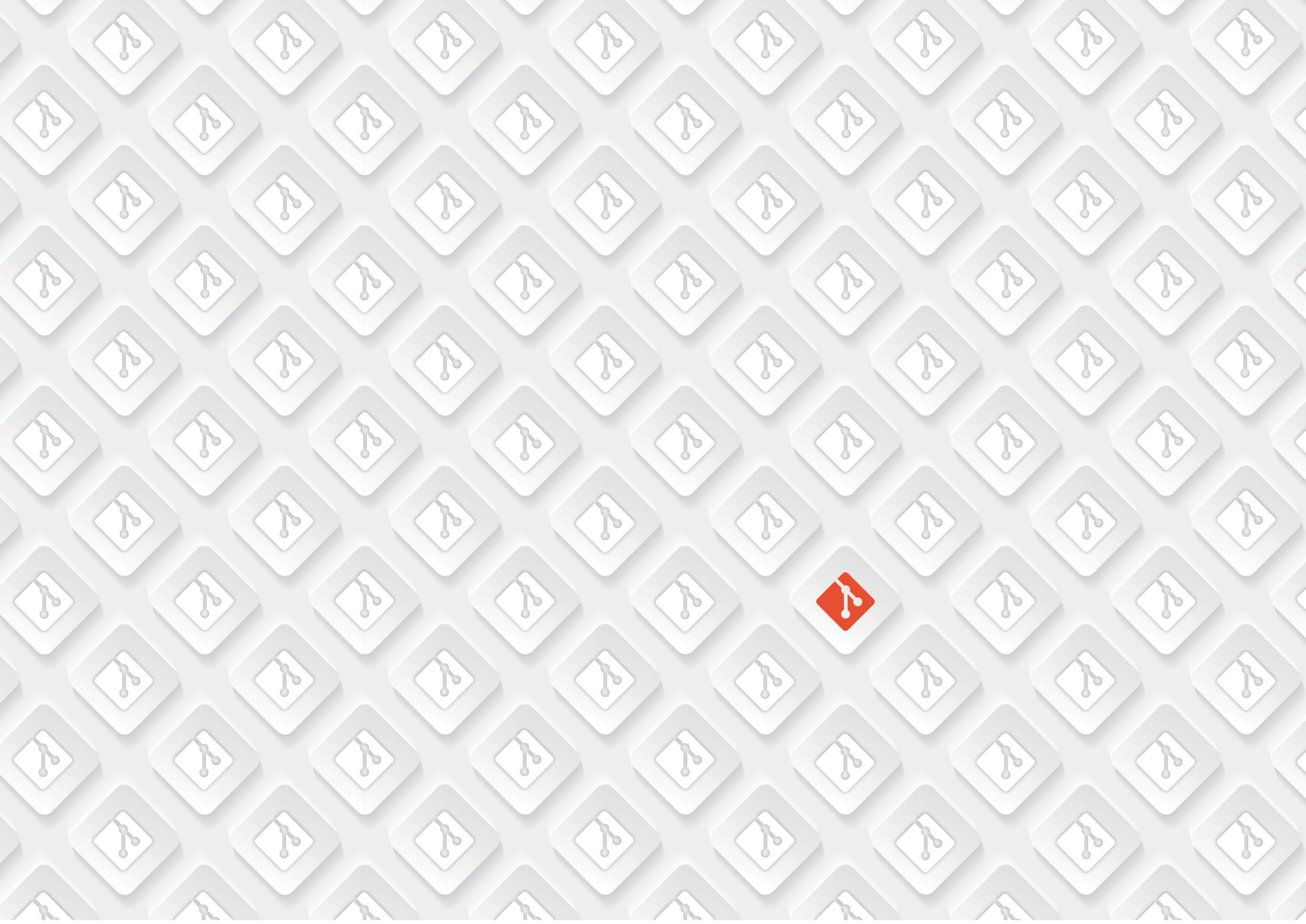


IMAGE BY ANDRIJ BORYS ASSOCIATES; GIT LOGO COURTESY OF GIT-SCM.COM (CC BY 3.0)

grep that are restricted to text matching and have no knowledge of program structure often provide too little or too much information.

Fluent code navigation is also an invaluable tool for researching bugs. The stack trace in an error-reporting system starts a journey of trying to understand the state of the program that caused that error; navigating code symbolically eases the burden of understanding code in context. As such, most integrated development environments (IDEs) have extensive support for code navigation and other such static analyses that ease the user's burden.

The Semantic Code team wanted to bring this IDE-style symbolic code navigation to the Web on github.com. The team was inspired by single-purpose sites such as source.dot.net, Mozilla, and Chromium Code Search that provide comprehensive in-browser code-navigation. The question was how to do that at scale: GitHub serves more than 65 million developers contributing to over 200 million repositories across 370 programming languages. In the last 12 months alone, there were 2.2 billion contributions on github.com. That's a lot of code and a lot of changes.

Philosophy: To Tree or Not to Tree

The Semantic Code team's approach to implementing code navigation centers around the following core ideas.

1. Zero configuration. The end user doesn't have to do any setup or configuration to take advantage of code navigation, beyond pushing code to GitHub. There are no settings or customizations or opt-in features—if a repository's language is supported, it should just work. This is critical for this particular use case, since if you view a source-code file on GitHub in a supported language, the expectation is that code navigation should *just work*. If every open source project had to do even a little extra work to configure its repo or set up a build to publish this information, the experience of browsing code on GitHub would vary dramatically from project to project, and the time between push and being able to use code navigation might depend on slow and complex build processes.

It's not sufficient to require that developers clone and spin up their own IDEs (or wait for an in-browser IDE such as GitHub Codespaces to load); developers are expected to be able to read and browse code quickly *without* having to download that code and its associated tooling. For this feature to scale and serve all of GitHub, it has to be available everywhere and in every project. The goal is for developers to focus on their programs and the problems they are trying to solve, not on configuring GitHub to work properly with their projects or convincing another project owner to get the settings right.

2. Incrementality. For each change pushed to a repository, the back-end processing should have to do work only on the files that changed. This is different from instituting a continuous-integration workflow, in which a user might specifically want a fresh environment for repeatable builds. It also hints that results will be available more quickly after push—on the order of seconds, not minutes. Waiting an entire build cycle for code-navigation data to show up isn't

Building Parsers

Every programming language must have a parser to convert its textual representation to one that can be executed by a machine. Instead of employing the canonical parsers for the supported languages, however, the Semantic Code team reimplemented these parsers atop Tree-sitter. This provides a number of advantages over using a language's built-in tooling:

- **There are many versions of common programming languages with various levels of syntactic compatibility. There is often no information that distinguishes a Python 2 file from a Python 3 file, so using a language's built-in parser would require heuristics or guesswork to find which parser is appropriate.**
- **Some language tooling requires external information or project-level configuration not present in the code itself. To fully analyze projects of this nature, the analysis must, in essence, run a build of the software. GitHub provides build infrastructure with GitHub Actions and post-build analysis with CodeQL, but users must opt into this functionality, and results are generally available in minutes, not the seconds that code navigation relies upon.**
- **Running individual language-specific tools is too complex to be operationally feasible. Not only do you end up with mixed-platform VMs (C# tooling would have to run on Windows, Swift on macOS, most other languages would require a Unix system), you would need a way to switch between various versions of the language and its core components. GitHub Actions tackles this out of necessity for the CI (continuous integration) space, but it requires effort from both users and implementers to select the right versions of languages, frameworks, compilers, and operating systems.**

By developing a common parsing framework, the Semantic Code team created standard machine-readable grammars that parse a superset of all versions of a language. This has interesting consequences such as being able to easily support the Embedded Ruby templating languages by composing the HTML and Ruby grammars. Additionally, this work yielded the ability to support multiple languages through one tool. Tree-sitter parsers are zero-dependency C libraries, making them easy to embed and bind to other languages.

tenable for the desired user experience; developers expect the navigation feature to keep pace with their changes.

3. Language agnosticism. The same back-end processing code should be run and operated regardless of the language under analysis. Consequently, the team decided not to run language-specific tooling, such as the Roslyn project for C# or Jedi for Python, as that would require operating a different technology stack for each language (and sometimes for each version of a language).

Though this means the language grammars may accept a superset of a given language, this philosophy yields the ability to scale and deliver results much faster. The infrastructure can run a single code stack, there's no management of multiple containers and associated resource costs, there's no cold start time for bringing up tooling, and there's no attempt to detect a project's structure, configuration, or target language version.

This also means that major aspects of analysis can be implemented in one place, providing abstractions such as machine-readable grammars and a tree query language. Only certain things unique to a programming language must be developed before that language can be enabled in the rest of the technology stack.

4. Progressive fidelity. Results that are *good enough* should never wait for ones that are *perfect*. The system has been designed so that over time the results it yields can be refined and improved. Instead of trying to build and reveal the perfect system for all languages across all of GitHub, the team has chosen to ship incremental improvements: adding languages one by one and improving navigation results with further technology investment.

5. Human partnering. Great software tools augment and complement human abilities. In this case, GitHub's Semantic Code team wants to facilitate the process of reading, writing, and understanding software and to do so in such a way that developers can contribute to the tooling in the languages they care about the most. The system should, to the greatest extent possible, avoid the walled-garden model of software development, as language communities are often ready and willing to fix bugs and oversights with respect to their language of choice.

Methodology: Tagging the ASTs

The static analysis that the GitHub code-navigation feature is built upon is called a *tag analysis*. A tag analysis looks at the definitions and the usages of functions, variables, and data types, collating them into a format suitable for viewing the definition of a given entity, as well as querying the codebase in full to determine all the places where that entity is used.

This analysis was introduced by a program named `ctags`, developed by Ken Arnold and released in 1979 as part of BSD Unix 3.0. As its name indicates, `ctags` supported only the C programming language; despite this, it was an immediate success, eventually incorporated into the Single Unix Specification. Further descendants of `ctags` added support for more programming languages, and the vast majority of text editors and IDEs now provide `ctags` integration. Though a tag analysis is trivial, relevant to the state of the art in static program analysis, implementing such an analysis at GitHub scale and within GitHub's distributed architecture was not.

Tree-sitter. The first step in any sort of static analysis is to parse textual source code into a machine-readable data structure. This is done by producing concrete syntax trees with Tree-sitter, a parser generator tool and incremental parsing library.

Tree-sitter enables specifying a grammar for a programming language and then generating a parser, which is a dependency-free C program. Given a parser and some source code, the system yields a syntax tree of that source code. The act of tagging that code entails walking the tree and performing a filter map operation. Using an s-expression-like DSL (domain-specific language), Tree-sitter provides tree queries that allow specifying which nodes to match in the syntax tree (for example, an identifier that represents the name of a function). The tooling in the Tree-sitter ecosystem allows fast iteration for grammar development and tree matching, making it possible to support new language syntax quickly or identify new constructs for code navigation.

Tree-sitter pulls from several areas of academic research, but the most prominent is generalized LR (GLR) parsing. An extension of the well-known LR (left-to-right derivation) parsing algorithm, the GLR algorithm can handle ambiguous or nondeterministic grammars, which are common in modern programming languages. Real-world languages use many different parsing algorithms: Python uses the LL(1) algorithm, Ruby uses LALR, and GCC (GNU Compiler Collection) projects use custom recursive descent parsers. The GLR algorithm, introduced in 1974 by Bernard Lang and implemented in 1984 by Masaru Tomita, is flexible enough to parse all these languages; relying on a less sophisticated algorithm such as LR(1) would be too restrictive, given that such parsers are limited to looking ahead only one token in the input (see sidebar for more information).

Tagging the trees. The next challenge is to do some basic analysis of the parse trees. To start, the team decided to provide naive name-binding information based just on identifiers for known syntactic constructs. This was originally designed as a prototype, but it worked so well it was shipped while work was beginning on further refinements.

The basic idea is to:

1. Parse a source file.
2. Walk the parse tree, capturing identifiers for certain syntactic constructions such as function definitions.
3. Keep a database table of these tags, along with information such as the line/col where they appear in the original source code and whether the identifier is the definition of something or a reference (for example, a function call would be a reference).

It turns out that for many languages, this name-binding information alone provides a compelling product experience. It's not perfect, but it's such an improvement over not having any information at all, that the team used this prototype to flesh out the production-scale system, while simultaneously researching how to refine the precision of the results.

To walk the parse tree, it suffices to use Tree-sitter's support for queries. By specifying the structure, using an s-expression syntax, of the types of syntax nodes that contain useful identifier information, a walk of the parse tree can yield only the nodes specified. As shown in Figure 1, given the use case of identifying method definitions in Ruby, a small snippet of Ruby code is parsed that defines a method add.

In more complex situations, the Tree-sitter tree queries can maintain custom state during the query's operation. This is important for a language such as Ruby, which makes no syntactic distinction between variables and functions called without arguments. To yield useful results, you must record the names of any local variables in a given Ruby scope, which allows that syntactic ambiguity to be eliminated.

Implementation: Architecture

GitHub's code-navigation pipeline is built atop open source software and standards:

▸ *Apache Kafka.* A platform for handling high-throughput streams of data such as commits to repositories. An individual datum in a stream is a message.

▸ *Git.* A distributed version-control system. Programmers upload changes to repositories of code hosted on GitHub. Each change affects one or more files—called blobs—in that repository. A unit of change to one or more blobs is called a commit, and the act of uploading one or more commits is a push. Once commits are pushed, other programmers can see and incorporate those commits into the copy of the repository present on their computers. Programmers can upload their code without affecting others by targeting a branch, a named collection of commits. Git projects start out with a single branch, usually called "main," which is then changed either by pushing commits directly or by reviewing and integrating others' changes (pull requests). Entities in a Git repository are given a unique tag based on the SHA-1 hashing algorithm.

▸ *Kubernetes.* An orchestration system for the deployment and operation of application services, Kubernetes provides pods, isolated units of computation on which one or many copies of a given application can be deployed.

▸ *Semantic.* An open source program-analysis tool from GitHub.

▸ *Tree-sitter.* This parsing toolkit, built atop the GLR algorithm, generates code that efficiently parses source code: The act of parsing transforms a human-readable, textual representation of source code into a tree data structure, usually referred to as a syntax tree, suitable for consumption and analysis by machines.

▸ *Twirp.* An RPC (remote-procedure call) standard, Twirp defines the manner in which entities in the system can communicate.

The system is made up of three independent services deployed on Kubernetes that integrate with the main github.com Ruby on Rails application (affectionately called "the monolith" because of its size and complexity): The *indexer* consumes Kafka messages for git pushes and orchestrates the processes of parsing, tagging, and saving the results for a particular repository at

Figure 1. An example of parsing and tagging Ruby code.

```
# test.rb
def add(a, b)
  a + b
end
```

Using Tree-sitter, the parse tree looks like this, in s-expression form, containing the node name and the line/column/span information associated with that node:

```
> tree-sitter parse test.rb
(program [0, 0] - [3, 0]
  (method [0, 0] - [2, 3]
    name: (identifier [0, 4] - [0, 7])
    parameters: (method_parameters [0, 7] - [0, 13]
      (identifier [0, 8] - [0, 9])
      (identifier [0, 11] - [0, 12]))
    (binary [1, 2] - [1, 7]
      left: (identifier [1, 2] - [1, 3])
      right: (identifier [1, 6] - [1, 7]))))
```

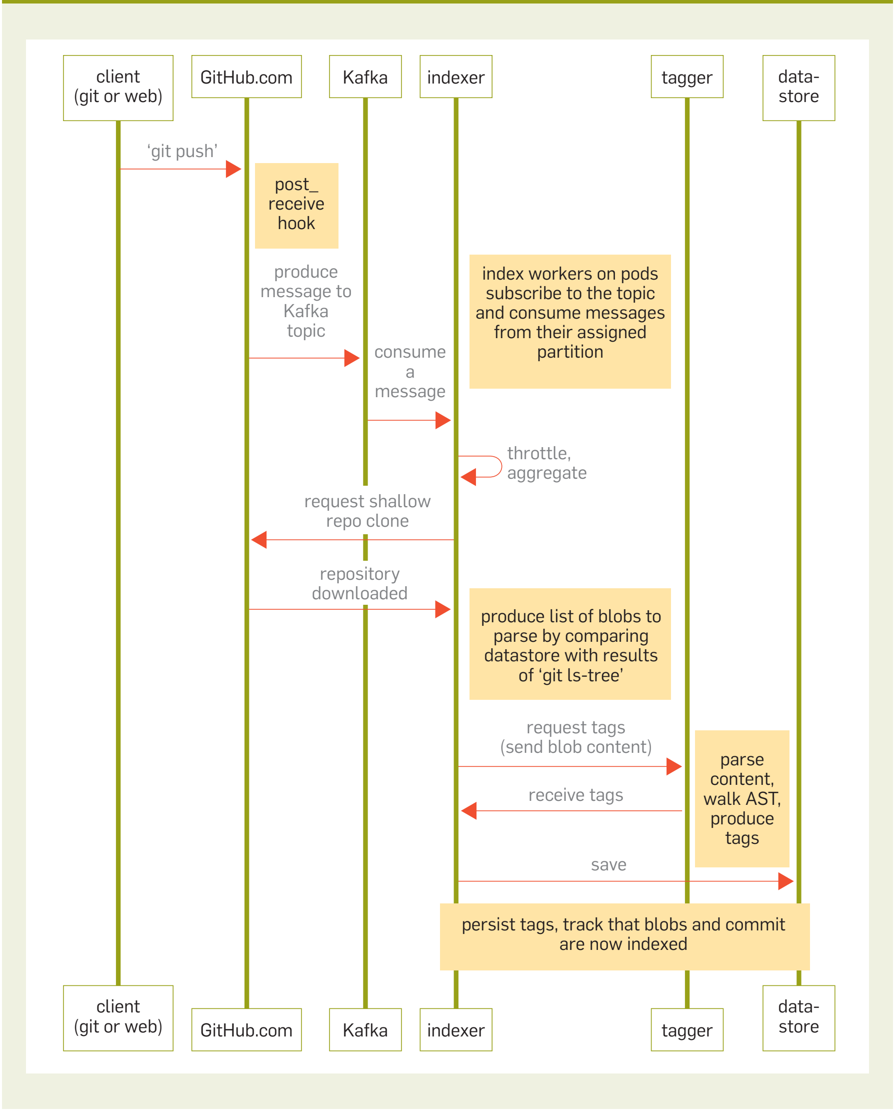
A tagging operation for Ruby methods such as add is encoded with the Tree-sitter query syntax:

```
(method
      name: (_) @name) @definition.method
```

The @ captures and tags parts of the tree that match, whereas the _ means we don't care about the structure of the tree beyond this point.
Tree-sitter can now use this query to provide the desired match:

```
> tree-sitter tags test.rb
        add | method   def (0, 4) - (0, 7) 'def add(a, b)'
```

Figure 2. Indexer sequence.


client (git or web)
GitHub.com
Kafka
indexer
tagger
data-store
'git push'
post_receive hook
produce message to Kafka topic
consume a message
index workers on pods subscribe to the topic and consume messages from their assigned partition
throttle, aggregate
request shallow repo clone
repository downloaded
produce list of blobs to parse by comparing datastore with results of 'git ls-tree'
request tags (send blob content)
receive tags
parse content, walk AST, produce tags
save
persist tags, track that blobs and commit are now indexed

a commit SHA; the *tagger* is a service that accepts raw source-code content, parses and tags this code, and returns the results; the *query service* provides a Twirp RPC interface to the symbols database, allowing github.com's front end to look up definitions and references.

Figure 2 shows the relationships among the various indexer sequence components.

Indexing. Developers upload code to GitHub via pushing through Git or as a result of editing a file through the website's interface. Upon receiving a new push, GitHub servers send a message to Kafka describing metadata (project location, change author, target branch) about the push. The system runs a pool of Kubernetes pods, spread across multiple clusters and physical data centers, that listen to these messages. Because messages are distributed across the pool of pods according to a repository's unique identifier, a given pod typically consumes messages for the same repositories over and over again, allowing effective local caching in the indexing pipelines.

As messages are consumed, the indexer process aggregates them, ensuring that pushes to the same repository within a time window are processed as one request. It subsequently filters out invalid messages; because of reasons of scale, pushes are indexed only to the default branch; they must involve only programming languages that the system supports. It then *throttles* the rate of indexing, so that initial indexes are allowed to complete fully before subsequent pushes are processed. Kafka records when a given message is passed off for processing, and a cap on simultaneous processing puts *backpressure* on Kafka. This means that in the case of capacity overload, the indexer will not attempt to consume too many messages, providing resiliency and elasticity in the system.

When processing a push, the indexer does a shallow clone of the repository to a temporary directory. The aggregation stage means a single download can be reused in future indexes.

Indexing involves determining which git blobs in the repository at the change under analysis have not been indexed. The OID (object identifier) of a blob is a SHA-1 hash of its contents; this identifier can easily be used to keep track of files that need to be processed. Blobs that haven't changed are shared between commits and require no additional work, though the set of paths and blobs reachable in a particular commit is recorded.

For the blobs that do require parsing and tagging, the indexer calls out to a separate service running on another set of pods. The reason for this separation is that not every push represents the same amount of parsing work. There could be anywhere between one and 20,000 files to process. The indexer workload is primarily I/O-bound: waiting on sockets (for Kafka, the datastore, repo cloning) and reading/writing to the file system. Parsing, however, is primarily compute-bound, so having a separate pool of pods across which requests are load-balanced means resources can be appropriately scaled, and the system won't run into the case where a single indexer pod gets *hot* as a result of processing a large or active repository.

The system is deployed up to dozens of times a day, and it handles those deployments gracefully by stopping the consumption of any new Kafka messages, allowing inflight indexes a 30-second window to complete processing, and aborting and re-queueing any index that can't finish in that window. Since processing is incremental for each git blob, the new pod that picks up that message will have to parse and tag only those blobs that didn't finish during the original run.

Querying. When tag results have been fully processed, they are stored in a database, along with enough information to serve future queries: *For repository X at commit Y where is* `foo` *defined?* The system provides a few Twirp RPC (remote procedure call) endpoints that allow the github.com front end user interface to look up definitions and references based on the user's interaction and context. In the past, the team's database of choice was a large MySQL instance, partitioned across multiple machines by Vitess (a MySQL sharding technology). As of this writing, tag data is stored in Microsoft Azure Blob Storage, a cloud storage system that supports uploading files.

Figure 3 shows the relationships among the various query sequence components.

Operational Journey

The first prototype of this system used the `ctags` command-line tool directly: An invocation of `ctags` dumped the yielded tags into the Git storage associated with the tagged repository, and furthermore attempted to do this on every push to that repository. This was great for a demo, but infeasible to run at scale: CPU resources are at a premium on the Git storage servers; multiple pushes to multiple refs made managing the tags file challenging; and handling lookup requests on those servers wasn't ideal. `Ctags` didn't have the API needed to take advantage of Git's data structures, and it's not easy to use command-line programs in such a way that their operations can be served by Web services.

Enter Tree-sitter. When the Semantic Code team picked up the project, the next iteration used Tree-sitter instead of `ctags` but most of the indexing logic was still implemented in the main github.com Ruby on Rails application. Queued on push, indexing jobs written in Ruby served to fetch Git content over an internal RPC, detect the language of a source file, filter out irrelevant content (for example, binary files, code in languages not supported), call out to a service for parsing/tagging, and save the resulting tags. A dedicated MySQL database was used for storing and retrieving tag data and the query path was essentially direct to this database from the Rails controllers.

This solved a certain set of problems and proved that the Tree-sitter parsers and initial name-binding analyses could provide parsing/tagging as a service, with easy iteration on grammars and easy horizontal scaling of the parsing compute workload. Most importantly, this prototype convinced leadership that this was a valuable set of features to build and that it enhanced the GitHub experience in a meaningful way. It also showed that we had a plan for scaling, supporting all the repositories on GitHub, and supporting new languages.

Out of the monolith. The next iteration moved further out of the github.com Ruby on Rails application, for several reasons:

- The job architecture in GitHub wasn't overly resilient and had "try once, best effort" semantics, which led to unreliability. Additionally, these operations entailed significant resource contention on the job workers shared with myriad background jobs being processed at GitHub, such as webhooks, delivering emails, and updating pull requests.
- As more repositories, users, and languages were added, the growth of the tags database impaired the ability to scale. This quickly became one of the largest MySQL instances at GitHub.

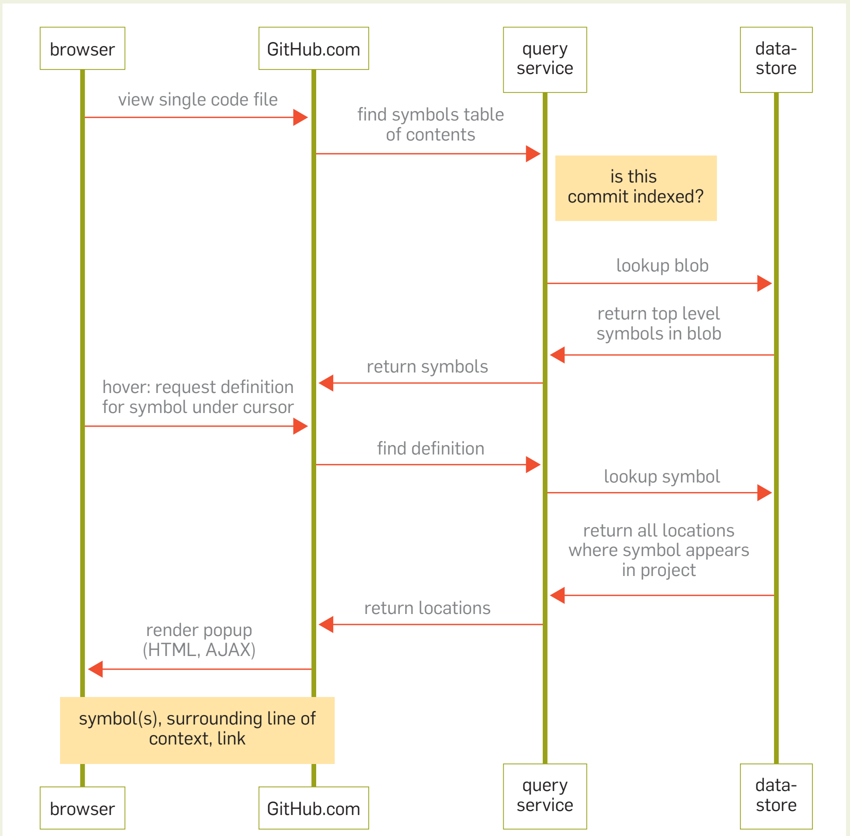
From an engineering organization perspective, working in the main Ruby on Rails application codebase was slow and came with a number of restrictions. Many parts of GitHub's main deployment pipeline are streamlined today, but at the time deploying a single pull request could take more than a day. To address these challenges, additional services were implemented:

- A rewrite of the parsing/tagging service, using GitHub's open source Semantic language toolkit.
- The indexer service, which consumes push information from Kafka, clones and parses repositories, and inserts results into a dedicated MySQL database.
- The query service that performs database lookups and returns results to the Ruby on Rails application.

These patterns served the system well for a significant period of time, allowing rapid growth, the release of several new languages for the code-navigation feature, and streamlined development and deployment.

Scaling MySQL. The next iteration required migrating the database from a single MySQL instance to a horizontally distributed cluster of partitioned MySQL instances, powered by the open-

Figure 3. Query sequence.


browser
GitHub.com
query service
data-store
view single code file
find symbols table of contents
is this commit indexed?
lookup blob
return top level symbols in blob
return symbols
hover: request definition for symbol under cursor
find definition
lookup symbol
return all locations where symbol appears in project
return locations
render popup (HTML, AJAX)
symbol(s), surrounding line of context, link
browser
GitHub.com
query service
data-store

source project Vitess. Moving to Vitess entailed engineering complexity: The team needed to write new schemas, choose new sharding strategies, and deploy both primary and secondary MySQL instances within this cluster. During this time, the team migrated from a Semantic-based tagging service to one that operated entirely with (newly developed) Tree-sitter queries. Though Semantic performed well, using the Tree-sitter query language allowed faster iteration and avoided the operational overhead of a program-analysis framework.

Moving to blob storage. The most recent iteration, and the one deployed to github.com as of this writing, entailed a dramatic change in database formats. While the Vitess-sharded MySQL database provided, in theory, significant runway for horizontal scaling (just add more shards), several new constraints became apparent:

▸ Money. With the Vitess cluster, there was technically no limit to horizontally scaling MySQL, but sufficiently powerful hardware entailed significant upfront and maintenance costs. While these costs were constrained by indexing only a repository's primary branch, projections for the costs of parsing all branches for all code on all of GitHub were simply prohibitive.

▸ Not all the features of MySQL could be used, especially given that SQL's capacities were limited by the constraints imposed by Vitess in order to shard the database correctly.

▸ The number of writes to the database was overwhelming MySQL: In order to keep MySQL and Vitess running smoothly, traffic had to pass through a throttling service, which entailed a significant bottleneck in indexer performance. Writes to MySQL are throttled based on replication lag. When the lag (time between a write to a primary being applied to a replica) exceeds some threshold, the indexer processes back off inserts. Usually these are temporary blips caused by unhealthy hardware, configuration differences, or just patterns of data (for example, developers are extra busy on Monday mornings and in January after the holidays).

Though this migration was primarily about cost structure, the change in data storage resulted in a few interesting performance benefits along the way. Architecturally though, everything

From an engineering organization perspective, working in the main Ruby on Rails application codebase was slow and came with a number of restrictions.

in the system stayed the same: Just swap the box that says "MySQL" with a different box that says "Azure"—that was the idea, at least.

In practice, this involved significant work. Not only did the team have to maintain the legacy MySQL and new Azure systems side by side, but it also had to design an Azure-compatible schema for tags data that would allow incremental indexing of repositories and querying of tags with roughly the same performance characteristics as the MySQL-based system. This meant optimizing for space and time and costs. (Storage costs money, but so does each operation: for example, read operations are less expensive than writes, which are less expensive than list-query operations.)

Datastore design. The storage structure the team ended up with is the following, expressed in a file-system-like form. The overall path structure includes a `<version>` prefix to allow major structural and formatting changes of the datastore. The data for each repository is then prefixed with its id (see Figure 4).

This in-database file hierarchy provides access to all the different data required to provide useful code-navigation information. During indexing, the appropriate files are uploaded to blob storage for the repo under analysis. At query time, the repository, a commit SHA, and the symbol name provide enough information to fetch the appropriate files and compute the results. All files are immutable.

`The` symbol names by hash prefix file (Figure 5) contains all symbol names in the commit that share a two-character SHA1 hash prefix stored as TSV. This prefix allows speeding up database queries by filtering based on this two-character prefix. This is the first file read when doing a symbol lookup in order to get a list of blob SHAs where a symbol is defined (D) or referenced (R).

The index blobs for a commit file (Figure 6) denotes that a commit has been indexed and holds a map of all blob SHAs in the commit, along with paths for those blobs. This file is read, and intersecting the blob SHAs with those from the previous step gives a list of blobs to fetch in a later step.

This file contains a header and two TSV content sections. The header gives the byte offset (from the start of the file) of the start of each section. Each section contains one line for each path that was indexed. The first section contains `<path>\t<blob_sha>` fields and is sorted by path. This allows a binary search for a particular path to find its blob SHA in the commit. The lines in the second section contain a single `<byte_offset>` field that is the offset (from the start of the file) of one of the lines in section 1. The section in Figure 6 is sorted by blob SHA and allows a binary search for a set of blob SHAs to find the paths where they appear in the commit.

All definitions and references for a particular git blob (Figure 7) is the final step in symbol lookup, providing the information to return to the front end. Many definition lookups result in a single file read at this step. Since references can be spread out across multiple files, reads are done in parallel with an upper limit.

Definitions are stored first, so that find operations can return early once the first reference is reached. Note that since this is the SHA1 hash of the blob contents, once a blob is parsed, it never needs to be parsed again and can be shared across any commit that includes that blob.

The other files are used to quickly tell if a particular branch or tag has been indexed or not. `refs/(heads|tags)/<branch_name>.tsv` contains the single commit SHA for the tip of the branch that has been indexed. `DEFAULT.tsv` contains a commit SHA and `ref` of the default branch.

Each iteration of the system and even the migration of the datastore happened under the full production load with minimal to no impact on end users. The blob storage change resulted in significant cost savings, an increase in indexing throughput, and a slight decrease in query request times.

Conclusion

The resulting code-navigation system processes and indexes more than 1,000 pushes per minute and generates more than two million new identifier names per minute. The p99 (99th percentile; that is, the time it takes for the slowest 1% of indexes) for the first-time index of a repository is ˜60 seconds (the 50th percentile is ˜1.1 seconds). The p99 for an incremental index is ˜10 seconds (p50 is ˜1.3 seconds). The system serves 30,000 requests per minute for symbol lookup, with p99 request times on the order of 90 milliseconds. It supports nine languages: C#, CodeQL, Go, Java, JavaScript, PHP, Python, Ruby, and TypeScript, with grammars for many others under development.

We learned, though, that scale—as much as being about large numbers of users and repositories and HTTP requests and the daunting size of the corpus of code hosted on GitHub—is about adoption, user behavior, incremental improvement, and utility. Static analysis in particular is difficult to scale with respect to human behavior; we often think of complex analysis tools working to find potentially problematic patterns in code and then trying to convince the humans to fix them. Our approach took a different tack: use basic analysis techniques to quickly put information that augments our ability to understand programs in front of everyone reading code on GitHub with zero configuration required and almost immediate availability after code changes. The result is a delightful experience of reading, navigating, sharing, and comprehending code on GitHub.

Figure 4. Datastore file system layout.

```
/<version>/<repo_id>/
  blobs/
    <blob_sha>.tsv.gz
    ...
  commits/
    <commit_sha>/
      index.tsv.gz
      symbols/
        <hash_prefix>.tsv.gz (see figure 4.)
        ...
    ...
  refs/
    DEFAULT.tsv
    heads/
      <branch_name>.tsv
      ...
    tags/
      ...
```

Figure 5. Symbol names by hash prefix.

```
# Filename pattern: commits/<commit_sha>/symbols/<hash_prefix>.tsv

blob _ sha <D|R> <symbol _ name>
```

Figure 6. Indexing blobs for a commit.

```
#Filename pattern: commits/<commit_sha>/index.tsv.gz

# content start:    <byte offset of start of content>
# oid index start:  <byte offset of start of secondary index>
# columns: filepath blob_sha
#################################################

<path>  <blob_sha>
...

<byte_offset of blob_sha>
...
```

Figure 7. All definitions and references for a blob.

```
# Filename pattern: blobs/<blob_sha>.tsv.gz

<symbol _ name> <D|R> <syntax _ type> <row> <col> <end _ col> <line of source code>
```

Timothy Clem has been building GitHub since 2011, where he finds great delight in distributed systems, the design and application of programming languages, and bringing research ideas to scale.

Patrick Thomson is an engineer, functional programmer, and programming-language nerd at GitHub.