EPISODE 1107

[INTRODUCTION]

[00:00:00] JM: Imagine the codebase of Uber. Uber has a gigantic codebase and it cannot be searched with naïve indexing algorithms. In order to search through a codebase the size of Uber's, it's necessary to build a much more sophisticated indexing system than a simple, pure text search. Sourcegraph is a system for universal code search. It allows developers to more easily onboard to a new codebase, make large refactors and perform other tasks. Sourcegraph can integrate with source control systems and IDEs and other tools to fit comfortably into an engineer's workflow.

Beyang Liu is a cofounder of Sourcegraph and he joins the show to talk about how codebases can become large and unwieldy and the tooling that Sourcegraph offers to make these codebases easier to work with.

If you are building a company in the developer tooling space or something related to infrastructure, send me an email, jeff@softwareengineeringdaily.com. I'm looking for companies to invest in and I'm also always looking for interesting stories for companies and interesting open source projects. Send me an email to jeff@softwareengineeringdaily.com.

[SPONSOR MESSAGE]

[00:01:13] JM: Join GitLab on August 26th, 2020 for GitLab Virtual Commit. It's an immersive 24-hour day of practical DevOps strategies shared by developers, operations professionals, engineers, managers and leaders. The attendees will hear from US Airforce and Army, the GNOME Foundation, State Farm, Northwestern Mutual, Google, more companies. Many more companies about problem solved, cultures changed, release times that have been reduced. You can come and be part of the community that is just as passionate as you are about DevOps. You can register today at softwareengineeringdaily.com/gitlabcommit. Register for GitLab Virtual Commit by going to softwareengineeringdaily.com/gitlabcommit.

Thank for listening, and thank you to GitLab.

[INTERVIEW]

[00:02:10] JM: Beyang, welcome to the show.

[00:02:12] BL: Hey, Jeff. Thanks for having me on.

[00:02:15] JM: I want to talk to you today about Sourcegraph, which is a company that you cofounded. And it's around code search and intelligence. Code search and intelligence, what does that mean?

[00:02:27] BL: It's basically a combination of two different things. Code search I think is self-explanatory, but it's basically the idea of you have this universal code that you work in. Maybe that's the corporates of code that's inside of your company. Maybe it could be as big as all of what's on github.com or gitlab.com. Regardless, it's unlikely that all the code that you care about using is on your local machine. And so in order to find the bits of pieces of code that are relevant to you at any given point in time, maybe you're looking into a bug report, maybe you're evaluating a new library you might want to use. You have to go and discover that piece of code.

And so what codes are just all about is making it so that you can type a query into a query box and instantly find the snippets or pieces of code that you're trying to get to. Once you click in to one of those results, that's where code intelligence, code navigation come into play. Then it's about you're reading through a code file, trying to build a mental model of what's going on in that code. And sort of the basic primitives of doing that are often go to definition, find references, hovering over certain symbols to get there, documentation.

And so code intelligence starts with that set of functionality. Providing those kind of basic navigational primitives so you can build that mental model quickly. And then it also kind of bleeds into more than just that. So think of all the additional context, contextual information that might be relevant to you while you're reading a piece of code. It can be things like a data that's pulled in that tells you the code coverage, the test coverage of a particular file. It could be annotations that link you to production incidents or outages that could be traced to a particular line and source. It really covers a lot of things. But the basic idea is that all the information that

might be useful to help you understand and build that mental model to the piece of code. That's what code intelligence is, and that's what kind of Sourcegraph is all about, search and discovery and then understanding.

[00:04:30] JM: Isn't this all accomplished by the IDE?

[00:04:35] BL: That is a great question, and that's a question that we've gotten a lot, especially in the early years of the company. The way I like to tackle that is, first of all, I think IDEs are great. They're extremely powerful tools with a lot of intelligence built into them. I think the trouble with IDEs arises when the set of code that you care about, that you might want to search and discover expands beyond just your current repository or what you have checked out locally on your machine.

Commonly, inside large companies, you have these giant codebases where no one is going to check out the entire codebase on to their local machine that would untenable. But also even more so, outside those larger companies, more and more people are relying on the universe of open source code. So the likelihood that you're going to want to find some piece of code that might be relevant to you that actually exists outside your IDE is kind of growing overtime.

The other kind of aspect of this is context switching. So developers, I think focus matters a lot to us in terms of our day-to-day workflow. And at least if you're like me, I use my IDE primarily for writing code. And when I'm in the mode of writing code, I don't like to get distracted. And the worst thing is when I'm writing code and all of a sudden I'm like, "Hmm, is there a library function that I could call to do this one particular thing?" Or let's say I get interrupted by some incident that happens in production. There's like an error message. I have to go investigate. The worst thing is like kind of blowing away my existing IDE state, like I was in the middle of writing something and now I have to like open up a new tab and then an IDE and go down this rabbit hole of figuring out what this other part of the codebase does. And then after I'm done with that, I'm kind of like completely lost. Where was I? I have to go retrace my steps and kind of rebuild that working state.

And so having code understanding tool that's kind of a separate application that's distinct from your editor, your IDE, is a way to kind of compartmentalize the part of the job that is like building

a mental model of a piece of code that doesn't happen to be what your – You have opened your IDE right now.

[00:06:40] JM: Sorry. Is there an integration point with the IDE? Or is this just a way to search code that is external to the IDE?

[00:06:50] BL: So there is an integration point with the IDE. We do have integrations that let you kind of seamlessly jump from what you're viewing in your IDE to Sourcegraph and vice versa. And that's because often times you'll be exploring something in your IDE and it's going to lead to this rabbit hole and you'd want to open up in Sourcegraph so you can start exploring that code. And often times, the opposite happens. When one of our users is diving into a piece of code in Sourcegraph, they're not sure whether it's something they want to edit. But they find the piece of code they're looking for. They build confidence in kind of relevance to what they're doing right now and then they want to open that file directly in the IDE. That's kind of how we integrate with editors.

We do have integrations with other developer tools as well, namely code hosts and code review tools. I guess that also gets to another aspect of this beyond the IDE attitude, which is a lot of times when you're reading through code, it doesn't happen to be in your editor. It happens to be in like a code review process. In that case, like by definition, it's kind of new and unfamiliar code. Code that you want to build a mental model of so you can evaluate in the basis of quality and not having those basic navigational primitives, go to dev, find references can be a huge handicap in those situations.

[00:08:05] JM: So what is the workflow for somebody who is using Sourcegraph? Describe that in more detail. How am I integrating with points outside of my own codebase that I have loaded into my IDE?

[00:08:22] BL: Yeah. There's kind of a variety of different kind of workflow, I would guess. Because it's essentially like – Sourcegraph is really two, maybe three things right now. There's a search bar. There's a code explorer. Those are in your web browser. And then there's an integration, a Chrome extension that integrates into your code host. So that would be kind of like a thing that enhances the code review experience.

So when we talk about workflows, it really depends on which one of those three things that you kind of interact with first. So maybe I'll just go through a couple of like individual workflows just to give not a representative sample, but maybe just like a sampling of how people use the product. One use case is you are on call, and let's say there's a production outage and you go in and find that there's a stack trace in the logs, and that stack trace points to a particular error message. There's a distinct error message that you're trying to track down in source code.

And so one use case there with Sourcegraph is you copy that error message. You paste it into the Sourcegraph search bar, and you don't know where this error message might be coming from, because it could – Especially if you're in kind of a multi-repository type environment, or a multi-service type environment, it could be coming from one of many services. You're not sure where. It could be coming from like an open source dependency that you're pulling in. It's likely not going to be in the code that's on your local machine. That's why you would go to Sourcegraph, you drop the error message into the search bar and then it would show you a list of all the places where that error message or that error message pattern appears in the source. And so then you'd click into those results, pinpoint the place that you care about, build a mental model of how that surrounding logic and code works. And then when you're ready to fix the bug, you would use the IDE integration to say like, "Okay. This file that I have opened in Sourcegraph, let's open that up directly in my editor and let's go and kind of edit the code and push up the patch." Does that make sense?

[00:10:22] JM: It does make sense. And we could talk a little bit about the golden ideal here, I think. Like the code navigation tools in Google and Facebook are probably far beyond what almost anybody else has access to. There's just such a level of integration with the code and their workflows that even the hot startup with 200 people is not going to hold the candle to. I'm sure you've seen those code navigation workflows. How do those compare to what somebody can get with Sourcegraph plus a modern IDE?

[00:11:06] BL: Yeah, totally. You mentioned Google and Facebook, and those are two companies that have built kind of Sourcegraph-like things internally for their own developers. And I actually have kind of a firsthand experience with Google's tool, because I work there. But

it was only an internship. But I got to kind of experience all the fantastic developer tools that Google has built internally.

And just to share some color around what that looks like for people who might not have worked inside of Google previously, so Google has this internal tool called code search. There is a Google code search that's public, but it's a completely different application. The internal one is a lot more powerful, but also a lot more specific to Google's internal mono repo.

And what I always like to tell people is that this is probably the most popular developer tool inside Google other than someone's primary editor. So when I was at Google, a lot of people had kind of the standard two monitor setup as a developer. Your first monitor is almost always open to your editor. And then the second monitor happens to be kind of like a reference tool, or a reference monitor. And at least on the team that I was, that second monitor was open to a code search tab probably 80% to 90% of the day. Because people would just use it as their go-to tool for diving into an unfamiliar part of Google's code.

The reason they did that was because, first of all, you have a giant codebase. You're not going to have it checked out locally on your machine. Second of all, you don't want to blow away the working state in your editor. It was so powerful, because you really felt that you had – All the code in Google felt accessible to you. It wasn't just available. It was actually accessible. You could dive into it and quickly build a familiarity with it.

And it had a lot of same features that Sourcegraph now offers. Kind of trigam-based regular expression search, go to definition, find references, things like that. But it also had some certain things that were limiting that Sourcegraph kind of works past. And there's also a certain set of features that Sourcegraph now has that Google code search never had.

So to start with, Sourcegraph supports I think basically every major programming language now. Whereas the Google internal tool was – At that time, I think there were three major languages. I think that's now around to 4 or 5 inside of Google. But at that time, it only supported the languages that Google kind of had officially endorsed.

Google's tool was actually quite kind of specific to the way that Google's mono repo functioned

as well. So Google had a very particular way of building its codebase and even invented a tool to build that codebase specifically, and code search was kind of tethered tightly to that way of building their code. Whereas Sourcegraph, we attempt to basically provide code search and code navigation for every repository out there. Now, obviously, there's a long tail of things that often are difficult to support, but we do our best to try to capture the most common cases you see in the wild.

And then kind of moving beyond a feature set of what Google do, there's been a variety of like additional features that we found are amazingly useful to our customers and our users, things like save searches. So save searches is a feature in Sourcegraph where you can basically create a search query that searches for like a particular pattern or anti-pattern that you want to keep track of and it will notify you anytime a new instance of that pattern pops up in the codebase.

There's also the extension API, which allows third-party plug-in makers to build essentially like the analog of IDE plugins, but for Sourcegraph. Integrating third-party sources of information on the kind of error monitoring data or the code coverage data that I've referenced earlier. That's all provided by kind of third-party plug-ins.

And then lastly, it's kind of the integration with your code review tool, which is amazing, because it kind of takes the code intelligence outside to your IDE and brings it to essentially every place that you might want to like read through and understand code.

[00:15:10] JM: So what you mentioned around the programming language support, so you gave the example that inside Google there is this blessed set of languages. I think it C++, Python, Java, Go maybe.

[00:15:24] BL: Yeah.

[00:15:24] JM: Or what they have today. So they only need to have parsing and indexing tools around that set of languages. If you have a much wider set of languages, what do you need to build around the parsing and indexing of these different programming languages?

[00:15:42] BL: Yeah, totally. It is a challenging problem. When you're trying to support things like go to definition, find references across every major programming language, and also build system, that kind of – There's almost like a combinatorial complexity that arises. And so the way we've approached that is this kind of approach that we've evolved over the years through witnessing what issues that our users and customers encounter trying to get code intelligence set up, is we have multiple layers of kind of the code intelligence backend. And these layers have different levels of precision or accuracy.

So there's kind of a basic layer, what we call basic code intelligence, that I think the way to think about it is it's kind of like C tags ++. So if you're familiar with C tags, it's this tool it's that's integrated into editor plugins that provides basic go to definition and find references functionality across many different languages. But it doesn't do that at a compiler level. It actually operates on kind of the – I don't know if they use regular expressions, but it's a pattern matching syntax that's similar in power to regular expressions or maybe a little bit beyond that.

It is, at the end of the day, kind of textual matching that doesn't fully understand the semantics of the language. And it turns out that for 80%, 90% use cases, this works quite well actually. So when I was at Google, incidentally, I was coding it mainly in C++ and everyone on my team was using Emacs. And everyone had C tags set up. And it turned out for C++, for the code that we were working on, this was kind of enough to get by for a jump to definition in the editor.

But obviously, that's often – It works 80% of the time, 90% of the time, but there's still the 10% to 20% time where it doesn't work. And those situations are situations where you have a function that's like name something like list, or get, or something really generic. In those cases, purely textual-based tool, because it doesn't understand the type system and structure of the language can't disambiguate between kind of semantically different functions or symbols that happen to have the same textual name.

And for things like that, we have to build kind of fancier support. And we call that next layer precise code intelligence. And so this is when you get into more of a compiler level understanding of the code. In some cases beyond the compiler, because for dynamically-typed languages like Python and JavaScript, you might actually have to do some amount of type inference beyond what the compiler and the interpreter is going to do, because you have to kind

of statically infer what the type or something is in order to be able to jump the user to the correct definition. This is a whole can of worms. I mean, there're a lot of different approaches that we use beneath the hood to deal with this. And it gets even more complicated when you think beyond compilers to build systems.

It is basically a standard compiler now in every major language, which is really nice from a code analysis point of view, because standard compiler – Like a single standard compiler also means there's kind of a one tool that will correctly build 99% of the code in a particular language. And you can hook into that tool to extract information you need to provide code intelligence.

The issue of build systems though introduces a complication into this, because many language ecosystems have more than one standard build system, right? In the Java world, which I used to work in quite a bit, you got people using Gradle. You got people using Maven. There are still people using Ant. There is people using Bazel now. It's just like a whole kind of buffet of options.

Within each of those options, there's a lot different configuration points. Within a single build.gradle script, there're a lot of things that you can do. And hooking into that, if you think about the information we need to extract from a codebase in order to provide code intelligence, it essentially reduces to the like symbol table that a compiler constructs internally when it's doing compilation of the code. Because from that simple table, we can then go and – Well, I guess it's simple table plus like the type annotated to AST.

From those two data structures, we can then go and extract the information we need to support any kind jump to def or find references. But in order to get to those data structures, you have to properly build the code, and that means properly integrating into the build system to actually build the code so that we can extract that data.

So far, the best approach that we found and the approach that we're kind of increasingly adapting is this protocol format, serialization format called, LSIF, L-S-I-F, and that stands for the language server index format. It's kind of the sister protocol to the language server protocol, which I can get into if that's of interest. but basically the idea it's this standard language-agnostic protocol that what you do is you write a tool that hooks into the build system and then spits out data in this LSIF format. And that is going to include things like all the definitions and references

that are contained in the code, so structural date about the code. You upload that to Sourcegraph, and then we have a backend that interprets that data. And when the user hovers over a particular reference or a definition, we use the data that we have stored and index to respond to that request.

[SPONSOR MESSAGE]

[00:21:34] JM: If you listen to this show, you are probably a software engineer or a data scientist. If you want to develop skills to build machine learning models, check out Springboard. Springboard is an online education program that gives you hands-on experience with creating and deploying machine learning models into production, and every student who goes through Springboard is paired with a mentor, a machine learning expert who gives that student one-on-one mentorship support over video.

The Springboard program offers a job guarantee in its career tracks, meaning that you do not have to pay until you secure a job in machine learning. If you're curious about transitioning into machine learning, go to softwareengineeringdaily.com/springboard. Listeners can get $500 in scholarship if they use the code AI Springboard. This scholarship is for 20 students who enroll by going to softwareengineeringdaily.com/springboard and enter the code AI springboard. It takes about 10 minutes to apply. It's free and it's awarded on a first-come first-served basis. If you're interested in transitioning into machine learning, go to softwareengineeringdaily.com/springboard.

Anyone who is interested and likes the idea of building and deploying machine learning models, deep learning models, you might like Springboard. Go to softwareengineeringdaily.com/springboard, and thank you to Springboard for being a sponsor.

[INTERVIEW CONTINUED]

[00:23:10] JM: So what you're describing where you basically have to interlope into the build process. So that's because if you have something like typescript, the typescript has to be compiled down to JavaScript first, because you need – Of you give an example of Java where you might have a build tool that would need to be interjected. Like a build tool for – I don't know,

building – I think in Spring, there's probably some kinds of situations where you're going to be making some meta-classes. Am I understanding that correctly?

[00:23:10] BL: Yeah. You got it. It's stuff like – At the end of the day, what we need to do is we need to extract the information from the type annotated AST and the symbol table. And so the question is how do you get those data structures? In order to get those data structures, you kind of have to – Well, you have to compile the code, right? And if there is kind of a missing dependency or there is a code file where you pass the wrong compiler flag so it doesn't find the code file where it expects to be. You're going to get a compilation error.

Some compilers and more fault-tolerant than others, but the vast majority of compilers I would say are kind of designed with this like all or nothing approach in mind, which is if there's a file missing, it's just not going to generate a binary, right? Because it's not going to generate like a half working program.

And so if the compiler breaks, then there're kind of two approaches. One is you can kind of rewrite your own compiler, which some people have done. Like the eclipse presentation compiler in the Java world essentially does this. It's like a fault-tolerant compiler whose goal is not to generate an executable, but to just like do enough to get the same symbol table and AST so they can respond to these co-intelligence requests. But writing your own compiler is a bit of an undertaking.

So the other approach is you run the build script in a way that doesn't – Avoids any kind like turning complete computations that the build might need to run, but in a way that passes enough information to the compiler that it can successfully generate these data structures to give you the information that you need to complete code intelligence actions.

[00:25:27] JM: Understood. Can we talk more about the integration point? So how – Sourcegraph gets integrated, and what kinds of work needs to be done in order to set up that integration?

[00:25:43] BL: Yeah, totally. The term integration is a bit overloaded, and arguably we could do a better job of disambiguating between the types of integration. So we have a few types of

integration. So the integrations with the editors that you kind of touched upon earlier, those are actually pretty basic. So maybe we'll skip those.

There is the integration with code hosts, which is like how do you set up the connection between Sourcegraph and whatever is hosting the code that you'd like to index? So github.com, or GitLab, or Bitbucket for instance, those are code hosts integrations. And we integrate with basically every major git-based code hosting solution and we also have kind of a way to support non-git-based code repositories as well.

That's one set of integrations. Then there is the integrations with code review tools, which is hooking into that kind of – It hooks into the UI of code review and code host tools. So this is different from the indexing integration, which is just about kind of ingesting data into source graph. This is now like, “Okay. Let's go and modify the UI of GitHub PRs or GitLab MRs to kind of inject kind of source graph magic in there. Go to definition, find references and other code understanding functionality.

And then there is a third set of integrations, which we call extensions, which are integrations with third-party developer tools ranging from code coverage tools, like Codecov, to things like error monitoring, reporting tools like Sentry and things like that, that take data from the APIs of those other services or those tools and then annotate the source code in Sourcegraph with that data. So it’d be like line-based test coverage highlighting. That’d be an example of a thing that you could do with the extensions, Sourcegraph extensions.

And the cool thing about Sourcegraph extensions is that it actually hooks into the integrations with code host UIs. So if you write like a testing coverage extension for Sourcegraph that gives these line-by-line test coverage highlights and a user also has the code host UI integration installed, they will be able to get these code coverage annotations in a GitHub PR without any kind of extra effort or set up. Does that makes sense? And I can kind of dive in to anyone of these three sets of integrations that are interesting.

[00:28:12] JM: Something I’m actually more – I mean, that was great explanation. But something I'm more curious about is the index itself that you have to build, and basically the data structures that you have to build, the architecture of Sourcegraph, where it sits. Is this just

like a big index that's sitting in-memory on AWS somewhere? Do I need to load something into my laptop? Where is it and what's the architecture of the data structure that you're building?

[00:28:39] BL: Got it. So there are kind of two ways to use Sourcegraph right now. One is you go to sourcegraph.com and you sign up using Github, or GitLab, or you create an account, and then you can search all the code on sourcegraph.com that covers most of the open source world. The other way is you run a Sourcegraph instance yourself. What that means is you deploy a server, the Sourcegraph service, on to – Typically it's like an AWS node, or a GCP node, or an Azure node, and there are multiple ways of doing that, because we have different deployment environments that we target. So you can deploy it as a single Docker container. You can deploy it as a set of Docker containers via Docker Compose. Or you can deploy it as a Kubernetes cluster. There're kind of different deployment models, but they all kind of try to do the same thing, just in different packaging. They all kind of involve the same different services that comprise a Sourcegraph backend. To get at your specific question about kind of the index format, there are a variety of different indexes that we build in the backend to support different requests.

For a search, for instance, we use an open source library called zoekt, that was created by developers at Google. It itself was based on I think an earlier implementation of code search written by Russ Cox, and then open sourced, which that inclination was also based off an earlier version of code search that was internal only to Google. So there's kind of the succession of open source code search libraries terminating in zoekt which we used to kind of construct this trigram index that we use to make code search really fast. So that's one kind of backend index that we use.

The other I think big index that we use is the LSIF index. Earlier, I was telling you about the language server index format. We kind of accept that data via uploads to the Sourcegraph API. And then we convert that LSIF data into this index format, and we use a SQLite light backend to do that. And that makes things like looking up definitions and references by their symbol ID or their moniker really quick. And that's essential to optimizing the request response cycle of things like go to definition request or a find references request. Does that make sense? I feel like I'm having to go into more detail here. I feel like this is – I'm not doing a great job of explaining it, but those are kind of like the two main indexes in our backend.

[00:31:09] JM: And so when I'm typing in a search in Sourcegraph, this is going to be federated to multiple indexes.

[00:31:20] BL: So when you're typing in a search, that typically exercises the trigram index mainly. The LSIF index is used mainly to serve kind of code navigation requests. When you're typing in a search, a couple things happen. So there is kind of like a piece, I'll call it middleware, like search middleware that accepts the GraphQL search API request. And then it kind of federates that to multiple search backends. And there's only one kind of search backend that's indexed, and that is the trigram index. We use that to serve kind of these global search requests. Like when you're trying it search across an entire universe of code, whether that's like all the code inside your big enterprise company or all the open source code in the world. We use that index to make that fast.

There are couple other backends that are tailored to other use cases. So there is in-memory search backend, which handles search requests that are scoped to a specific repository or a small set of repositories, but at a different version that's not the master or default version of the code. And that obviously comes in handy when you're trying to explore a different branch or a particular commit.

There's another backend that is specifically tailored to symbol search. So a lot of major use case of code search is you don't want just any text result matching a particular query. You want to see only like function definitions that match it. You don't want to match stuff in like the comment string or like string literals or things like that. So there's a symbol search backend that also hooks into kind of the semantic understanding part of Sourcegraph. Knows what a symbol is and then searches symbols specifically.

And then there’s a couple other more minor search backends. Like there's one for kind of dif search, where you’re trying to search across like commit messages or difs that get exercised in specific user queries. So there's a really rich set of, I guess, like backend that are all federated into this single search query box that support a variety of different search use cases. Because what we found is code search is not just a single feature. It's actually a multitude of different use cases and features kind of bundled up into a single search box and a query language.

[00:33:30] JM: Is it hard to keep that search performant or does it matter? Because I guess you type in a search query and the work can be done somewhat asynchronously. You can probably do some naïve parallelism to keep it performant. Tell me about keeping that search performant over a really big index.

[00:33:50] BL: Yeah. I mean, that's a great question. I guess it is a challenge. It's a major challenge, especially when you're trying to search over all the code that happens to exist in the world and make it fast. It's particularly important to make it fast, I think, because I think I as a developer and many other developers, we value speed a lot, right? Any sort of wait time that we experience in the UI, that becomes an opportunity to lose our focus or fall out of the state of the flow. So we've invested a lot of effort into optimizing the performance of our search. And I think this is also kind of the case for other search engines, like Google. There's kind of like a couple of key ideas that give you like big gains in terms of performance and what performance is possible. And then there is just like a long tail of hacks and specific optimizations that you implement to target specific bottlenecks.

And so as far as like the big ideas in the code and indexing backend that make it fast, a number of them I've kind of already touched upon. One is the use of a trigram index to make a fast. So that's one key insight. It turns out there's a lot more you have to do to actually make that scale in practice. We have started backend. So when you're hitting the trigram index backend of Sourcegraph, from the point of the like search middleware, it looks like you're just hitting a single service. But really, that services is sharding that request out to – Sorry. It's distributing that request to multiple shards of the index and then collects those results and refers the kind of combined results back to the search middleware, which then combines them with all the other responses from the various search backends and sends it back to the user.

The index is a big important thing. There is also various ways of making in-memory search fast. So we've spent a lot of time thinking about optimizing regular expression, making regular expression search fast by – Making regular expression search fast for kind of like in-memory data. So there's a variety of different regular expression. What's the word? Like matchers, libraries out there. We've tried a lot of them and finally settled on one in particular that seems to have the best performance, and that's the one that we use for kind of in-memory search.

And then there's just kind of like the long tail of hacks, like playing around with pagination effects, thinking seriously about like whether you need a precise total account, total count of results for a particular query versus just a rough estimate. It turns out rough estimate could be a lot faster if you're surveying like an end-user request where they don't really care about the total number of counts. I would say it's kind of roughly analogous to Google search, where like the kind of initial insight was like page rank. And that was a step function increase. But then there was a long tail of hacks that they implemented to make their search really fast. And now if you're to dive into that search codebase, it'd be – I feel like it'd be a lot of special cases you'd find in the code just as you'd find in our code.

[00:36:49] JM: You mentioned a term called a trigram index several times. Is that T-R-I-E gram or T-R-I gram?

[00:36:57] BL: T-R-I-E. Well, I guess the spelling is T-R-I –

[00:37:02] JM: Like the trie data structure.

[00:37:04] BL: Sorr. It's not T-R-I-E. It's T-R-I, and in three, trigram. And basically the idea of the index is you store, you index things based on n-grams. In this case, three grams. And it turns out that this is a particularly efficient way of looking up specific like pieces of text in code, if that makes sense. The ideas is that like for code search, you're handed either like a string literal or a regex. In either case, it's kind of full text index. It's probably not what you're looking for, because the words that you encounter in source code are very different than the words that you encounter in plain text, right? You have fewer like whole words and more these like camel case concatenations or state case concatenations of various words. Or it might be like some long string of text that you want to treat as like a string literal, like an error message, right? And if you use kind of a standard plaintext indexer, it would find all these like fuzzy matches, which is not what you want.

And so one of the insights, and like, honestly, we can't claim to be the discoverer of this insight. I think it really goes back to – Well, I don't actually know who the original originator, but like where I learned about it was a blog post from Russ Cox where he talks about implementing a trigram

index for Google's internal code search. It's all about finding these like sequences of three characters that are pretty –And it turns out those are pretty distinct, and those become kind of a fingerprint for a particular piece of code that you're looking for. And so you end up building an index of these like trigrams and then searching for those, and that gives you a set of search results that works really well for both full text and regular expression code searches.

[00:38:50] JM: So it's three characters or three words?

[00:38:53] BL: I believe it's three characters.

[00:38:55] JM: Okay. And so that is just one of the indexes, the trigram index. And then there's other indexes you mentioned? What were the other indexes?

[00:39:06] BL: Yes. Sorry. I feel like I'm not doing a great job of explaining all these. But the trigram index is the primary index for code search. Basically, it's the only index that we hit for code searches. And then the other index I mentioned was the else if index, and that's for serving code navigation requests. When you click into a search result and you're trying to go to a definition or find the references of a particular symbol, then we would hit the else if index.

[00:39:35] JM: Got it. So the else if index, this is the language server index format.

[00:39:40] BL: Exactly.

[00:39:41] JM: Right. Okay. Tell me more about how these two search methods compare. You have that the language server index format and then you have the basic trigram search. How do these two compare?

[00:39:51] BL: They are extremely different. So the trigram index is optimized for serving request or the form like I type in a search query. It's like a particular pattern that I'm trying to find in a code. Now, go find me all instances of that pattern across the entire codebase. Whereas the else if index, the request that it's supporting is I have my cursor over a particular symbol. It's a function name for instance. And now I want you to find me the single definition where that

symbol is defined, or I want you to find me all the different places in which that symbol is referenced in a code.

The former, the trigram index, starts with a user query and is fundamentally about like finding a token of text or tokens of texts that match the pattern that you're looking for. And then the else if index is all about I'm hovering, I have my cursor over a particular point in a code file and now find me the forward and backward references of that vertex in the graph of references and dependencies in the source code. Does that make sense?

[00:41:04] JM: Yeah, it does. So you build both of these indexes for every company that uses Sourcegraph.

[00:41:13] BL: Correct.

[00:41:14] JM: Okay. And have you noticed, are there particular types of codebases the work better for LSIF and particular ones that work more for the search-based trigram format?

[00:41:26] BL: I would say the challenge here is not actually in the building of the indexes. I would say the – I'll treat the two different indexes separately, because they behave very differently. The trigram index is not really – There's nothing like language-specific about that index. It's kind of like a text-based index. It doesn't really vary by language. Now, there are kind of certain languages that are more verbose than others, and maybe that has some kind of like second order effect on the overall accuracy or the number of results that you have to sift through. But by and large, we don't really think about language specificity when it comes to the trigram index.

The else of index is interesting, because the index format itself is language-agnostic, but the way that you generate the index data is highly, highly specific to a particular language and a particular build system. Yeah, that's kind of the difference. And then for the else if index, in full transparency, like our support for languages using the else if index, it's not 100% across-the-board. We actually fall back on the kind of basic code intelligence method for a lot of languages. And a major challenge for us in the next couple months or so is building out support for many more languages using LSIF. And what that entails is kind of writing or adapting existing

indexers, else if indexers, to all the different build systems that we want to support in a particular language ecosystem and then documenting them such that it's easy for users and customers to set that up in their build system or CI pipeline. And we've also kind of invested in automating this whole process. So there are certain languages, like Go for instance, where for I'd say like 90%, maybe 95% of projects, we don't actually require you to do any manual set up in your CI service. We can kind of infer all the parameters we need to go and construct that index ourselves.

[SPONSOR MESSAGE]

[00:43:48] JM: When the New Yorker magazine asked Mark Zuckerberg how he gets his news. He said the one news source he definitively follows is Techmeme. For more than two years and nearly 700 episodes, the Techmeme Ride Home Podcast has been one of Silicon Valley's favorites. The Techmeme Ride Home Podcast is daily. It's only 15 to 20 minutes long and it's every day. By 5 PM Eastern, it has all the latest tech news, but it's more than just headlines. You could get a robot to read you the headlines.

The Techmeme Ride Home Podcast is all about the context around the latest news of the day. It's top stories, the top posts and tweets and conversations about those stories as well as behind-the-scenes analysis. Techmeme Ride Home is like TL DR as a service. The folks at Techmeme are online all day reading everything so they can catch you up. Search your podcast player today for Ride Home and subscribe to the Techmeme Ride Home Podcast.

[INTERVIEW CONTINUED]

[00:44:51] JM: Else if, you're saying that whenever somebody ships new code, the index needs to be updated, right?

[00:45:00] BL: Correct.

[00:45:01] JM: What does that update look like?

[00:45:04] BL: Yeah, that's a great question. In the naïve case, it is you re-index the entire repository. And that is kind of still the case for a lot of code that we index. But lately, we've also invested in kind of incremental indexing, which is you examine the diff of the files that changed since last time you indexed. Determine what files need to be recompiled, and then you just extract the index information from those files and then merge it with the index information you have from the last run of the indexer. And this is obviously can be quite a lot faster especially for large codebases, but it is also more difficult from a technical standpoint, because you have to figure how to merge those indexes correctly.

[00:45:54] JM: And how does this fit into a CICD workflow?

[00:45:58] BL: Yeah. The idea is build systems and build configurations are highly idiosyncratic and often times highly specific to a particular organization or even a particular repository. And so instead of trying to build a generic backend that can go and understand all the different exponentially variable build configurations there are, which I would argue is an intractable problem. What we do is we say, “Okay. You as the owner or maintainer of this codebase, you understand the ins and outs of your build system better than we do and probably better than anyone else in the world by a wide margin.”

And so what we’re going to do is instead of trying to like magically infer what this configuration is for you, we’re going to say, “Okay. Here's this tool. We’ll give you a well-documented tool with various configuration points and explain how this hooks into a build system. But then we’re going to rely on you to actually add a step in your build pipeline and pass the proper parameters to this tool to generate the else if data.” Does that makes sense?

[00:47:13] JM: Yeah. Yeah. Definitely. And the thing I wonder is what have you seen around usage of Sourcegraph that has surprised you? How have people that are using Sourcegraph surprised you with their usage?

[00:47:27] BL: I think one of those surprising things about Sourcegraph usage is just the multitude of ways in which people find Sourcegraph useful. When we originally wrote Sourcegraph, the use case that I kind of had in mind was this use case of exploring new libraries and building a mental model of how those libraries function, how to call their APIs and

things like that. It turns out that there is actually like a lot more use cases for searching code beyond just that that actually overwhelmed that in terms of frequencies.

One particular case was the case I mentioned earlier, which is investigating production outages. We've heard from customers that like, “Hey, Sourcegraph cut down the time it took to resolve our production issue from like hours to essentially minutes, and that essentially allowed us to bring our ecommerce website back online and continue serving payments requests.” And that was certainly like surprising, because I had not anticipated that kind of use case.

There are other features that we’ve built in response to customer use cases that had been surprising. So like that save search as functionality that I mentioned briefly earlier was kind of built in response to a customer need that we saw where people were kind of saving queries in Sourcegraph and using them to track like anti-patterns that might get added to the code. So instead of like, “Okay. Let me start out with a query that I want to find at a given point in time.” It’s more like, “Run this query on a recurring basis in the background and alert me when the results set changes.”

I don't know. There is kind of like a lot of things. I guess another thing is – So when we first started Sourcegraph, we kind of focus it primarily on like search and discovery use case. But as we talk to more and more of our customers, we found that actually there's a lot of companies out there that have this problem of I want to make some simple change to a library that is shared across many services or many parts of code in my codebase. And if my codebase is sufficiently large, making this simple change actually becomes extremely nontrivial. If I want modify the public interface of a function that’s dependent on by a dozen or so additional services, I not only have to make a pull request against that specific library repository, I have to go and open a PR against a dozen or so different repository and then I have to interact with a dozen or so different teams potentially.

And so we’re seeing people use Sourcegraph for this kind a like large-scale find and replace procedure, and they are using Sourcegraph mainly for the kind a like find part of that. And then they would have some, like hacky shell script or a homegrown tool to go and actually initiate that a set of changes across many different repositories and monitor the progress and kind of shepherd that change through.

And in many cases, what's very surprising to me is that like inside larger companies, this can be a process that takes like months, or even in one case, like over a year. And that just seems crazy to me. That's one of things that like – I guess like it's one of those things when people type in like codebase ossification, like this kind a like as your codebase grows in complexity, it becomes harder and harder to change, change it and improve it significantly. I think this like feeds right into that.

And so we kind of took that and said like, "Okay. They're already using Sourcegraph for kind of the fine part of this find and replace procedure. How can we better support the replace part of it?" And that actually led to a major new feature that we're now launching called Sourcegraph Campaigns, which is all about first expressing a pattern that you like to find in the source code and then specifying a replacement pattern to substitute that old pattern with a new replacement and then executing this in kind of a formal, easy-to-use and standard kind of way across your entire codebase.

[00:51:20] JM: Okay. Well, I think that sets us up for a bit of a conversation about the future. So Sourcegraph has a pretty detailed master plan. You have goals for 1 year, 3 years, 5 years, 10 years, 30 years into the future. Tell me about the master plan. What's the arc of the future of Sourcegraph?

[00:51:43] BL: Yeah. The master plan – And for those listing, you could just Google Sourcegraph Master Plan, that kind of lays out our long-term vision for the company. I think there're kind of two ends to the master plan. There's kind of the near term and then there the long-term. And then a lot of the plan is just talking about how we make those two meet in the middle. And so I would say in kind of like the near to medium term, next couple years, we are really focused on producing a tool that helps people kind of understand and grok the universe of code that's relevant to them. And I think, increasingly, for most developers, that universe that's relevant to them is becoming like the universe of all code in the world, because if you're not accessing and leveraging the universe of open source code, you're liable to fall behind in kind of the software world.

And so we're investing in code search, making it scale and making it superfast across an even broader range of repositories, supporting kind of new pattern matching syntaxes beyond regex, because reges is I think a lot of developers have a love-hate relationship with regex. And there's actually a new pattern matching syntax that was created by a teammate of ours in his PhD thesis that we think is awesome. And kind of also investing in like the UI of code search and making it accessible to both beginners and power users of Sourcegraph.

Also, the co-navigation part, supporting precise code navigation, code intelligence across all the different languages in the world, expanding the extension API so that we integrate more and more third-party tools into Sourcegraph. We think that could become a potentially fantastic channel for a new developer tools to get discovered in large part because the people who use Sourcegraph typically use it organization-wide and like every developer uses it every single day. So like that's the level of engagement and value that we're seeing. And so we want to use that to make it an avenue of discovery for like other great tools that you can use. And then also building out new features that are kind of adjacent to what we currently do, like campaigns. It's all around this task of like making sense, understanding and kind of managing the complexity inside large corpuses of code, large universes of code.

In kind of the longer term, our Northstar has always really been to make it so that everyone in the world has the opportunity to code, and not just to code, but also to make like a living off a code. To code as part of their job, or code as – Produce a code or a software product that other people use, get value from and like ultimately pay for. And the way we're going about that is we're first starting with a tool that understands the universe of code and makes that accessible to every professional software engineer. And then we want to slowly kind of work our way down. Like this summer, we actually have an intern who's working on standing up Sourcegraph for various universities and focusing on kind of a classroom use case. Ultimately, we also want to make it accessible for people in like kind of non-software engineering roles to also understand the code. Because guess what? Code is becoming kind of an essential part of every company and almost every organization side of company.

A salesperson or a person on the marketing team, there might be some piece of automation that they want to modify our understand that's relevant to their jobs. And right now, I think code is by and large a black box to people, even people with like a background on software engineering.

Like unless you have the technical chops and familiarity to like set up a local development environment for your codebase, it's a black box to you. And so we would like to kind of break that boundary and bridge the gap so that anyone out there whose job is kind of affected or adjacent to code, which I think increasingly is becoming more and more people can kind of dive into code and understand it and make a valuable update or change to it.

[00:55:49] JM: Well, that's great future, and I appreciate you sharing that with us. It's been great talking to you, Beyang, and I am excited about Sourcegraph in the present and in the future.

[00:56:00] BL: Awesome. Thanks so much for having me on the show, Jeff.

[END OF INTERVIEW]

[00:56:12] JM: You probably do not enjoy searching for a job. Engineers don't like sacrificing their time to do phone screens, and we don't like doing whiteboard problems and working on tedious take home projects. Everyone knows the software hiring process is not perfect. But what's the alternative? Triplebyte is the alternative.

Triplebyte is a platform for finding a great software job faster. Triplebyte works with 400+ tech companies, including Dropbox, Adobe, Coursera and Cruise Automation. Triplebyte improves the hiring process by saving you time and fast-tracking you to final interviews. At triplebyte.com/sedaily, you can start your process by taking a quiz, and after the quiz you get interviewed by Triplebyte if you pass that quiz. If you pass that interview, you make it straight to multiple onsite interviews. If you take a job, you get an additional $1,000 signing bonus from Triplebyte because you use the link triplebyte.com/sedaily.

That $1,000 is nice, but you might be making much more since those multiple onsite interviews would put you in a great position to potentially get multiple offers, and then you could figure out what your salary actually should be. Triplebyte does not look at candidate's backgrounds, like resumes and where they've worked and where they went to school. Triplebyte only cares about whether someone can code. So I'm a huge fan of that aspect of their model. This means that they work with lots of people from nontraditional and unusual backgrounds.

To get started, just go to triplebyte.com/sedaily and take a quiz to get started. There's very little risk and you might find yourself in a great position getting multiple onsite interviews from just one quiz and a Triplebyte interview. Go to triplebyte.com/sedaily to try it out.

Thank you to Triplebyte.

[END]