EPISODE 1166

[INTRODUCTION]

[00:00:00] JM: Static analysis allows for the discovery of issues in a code base without compiling. There have been many generations of static analysis tools. A newer static analysis tool is DeepSource, which automates code reviews, identifies bug risks and generates pull requests to fix them. Jai Pradeesh and Sanket Saurav are founders of DeepSource and join the show to talk through the creation of static analysis tooling and their work on DeepSource.

If you want to support Software Engineering Daily more intimately, go to softwaredaily.com and become a paid subscriber. You can get access to all the episodes without ads. Go to softwaredaily.com to learn more.

[INTERVIEW]

[00:00:44] JM: Guys, welcome to the show.

[00:00:46] JP: Thank you so much for having us.

[00:00:49] SS: Hey.

[00:00:49] JM: The common problems that exist across codebases, the common bugs and anti-patterns and security flaws, what are those common flaws? What are the most common flaws and security bugs that appear?

[00:01:03] SS: Right. So there are anti-patterns, there are performance problems, there are security vulnerabilities, and these are highly specific to programming languages. Say, to give an example, in Python for example, if you're using the assert statement, you're not supposed to use the assert statement outside of your test files, because if you are using the asset statement to validate client input, like user input, and if you're optimizing your byte code, these assert statements actually get stripped off and the validation never happens. So this is one instance. So if you are using something like that, that opens up a security validity in your application code.

[00:01:47] JM: And that's a very general issue that could exist across tons and tons of codebases. I think the goal of what you're doing with DeepSource is, to some extent, find commonalities across codebases that can be analyzed and potentially alleviated. Is that correct?

[00:02:06] SS: Yup.

[00:02:07] JM: Talk in a little more detail about what DeepSource does.

[00:02:11] SS: Sure. So DeepSource is an automated code review tool that helps developers find and fix issues in their code. So we use static analysis and a bunch of other related technologies and integrate with your GitHub account or your GitLab or your Bitbucket account. And after you have enabled DeepSource on a particular project, whenever you make a pull request, DeepSource will analyze those changes and show you issues that you need to fix. And these issues are, again, performance issues, style issues, anti-patterns, security vulnerabilities. And also helps you automatically fix these issues in like a couple of clicks. So in essence, DeepSource helps you write better code and you can think of it like Grammarly, but for your code. Yeah.

[00:03:05] JM: So I would run DeepSource at the time of a pull request. This is a classic static code analysis type of tool. There are lots of static code analyzers. What's new about DeepSource?

[00:03:17] JP: Sure so the reason why we started DeepSource was like even as part of our previous company, we try to use all these static analysis tools. There are like two types of tools. One are these open source linters and tools that are available for programming languages, and the other one on the side are commercial products, right? So we tried all of them, and we ended up disabling all of them after like a few weeks in general. Then we figured out like there were some of the reasons. One was a high rate of false positives. With the commercial products, we found really high rate of false positives. That was kind of annoying. And none of those analyzers were really maintained. Like they were not really up-to-date. These issues were like the same for years and years. And on the other end, the problem with open source linters was it was very hard to configure. You have to have someone in the team to actually maintain it. And then the integration was not very smooth. In general, these open source tools are run as part of your CI systems, right? Which generally like messes up your other builds and everything. Sso that's the reason.

Like if you look at, say, in terms of CI systems, like before 2010, like primarily there was Jenkins and it was like hard to configure. Not a lot of people were like adopting it. The adoption was like very slow. But on the other end, in early like 2011, ’12, these days companies like Circle and Travis they came and they made it super easy for anybody to set up a CI system. And now nowadays like if you set up a project the first thing you do is like probably set up, write some test and set up a CI system. And we want to do the same for static analysis, like a simple configuration file where you can like manage everything. Like use any static analysis tool you want like as long as the language

is supported by us and integrate natively with like your daily code review workflows, like with native pull requests, or whenever you run a commit, it should run. And like most importantly less than 5% false positives is something which we guarantee. Yeah.

[00:05:13] JM: Could you go into the stack a little bit? What happens when I actually run DeepSource?

[00:05:17] JP: Sure. So the way it works is, say, first if you have a project that is on either GitHub or GitLab or Bitbucket, all you need to do is add a file called deepsource.toml and you say that, "Hey, I want to enable these analyzers." And we support Python, Go, Ruby and JavaScript. So you say that enable Python analyzer. Give some metadata. And that's it. And you go to deepsource.io and then you sign up. And after you do that, we have native integrations with these three providers, platform, like source code providers at the moment. The moment you do that, every repository gets a dashboard that will show you like this is the current state of the repo. And when the first time you integrate this, the initial analysis runs and it will show you issues that are categorized as anti-patterns, bug risks, performance issues, security vulnerability, style and documentation. All of them have like detailed issues and where that issue is and a helpful description of how exactly can a developer fix it. And this is one part.

And after you do that, you don't need to configure anything else. Like whenever anyone contributes to that repository like send a commit or a pull request, DeepSource automatically runs and then shows you on how many bugs are you introducing and how many are you fixing. And yeah, it's a continuous process.

[00:06:34] JM: How do you assemble the static analysis, the code paths that you're regarding as buggy or problematic that the things that DeepSource is actually studying? H do you formalize those in the DeepSource scanner?

[00:06:47] SS: Right. So we maintain our own analyzers. We build our own analyzers. And to give an example, and let's take a programming language to illustrate this better. So for example in Python, we aggregate. First thing is we aggregate all the popular open source tools that exist. Say, for example, in Python you have PyLint, you have flake8, you have bandit and a couple of other tools. And in addition to that, we write our own custom checkers, right?

So the way that each issued, when we say that – So DeepSource detects over 650 issues in Python today. So when we say that we detect 650 issues, what it means is our Python analyzers has 650 definitions of these issues. And these individual checkers for each of these issues are

formalized in the form of actual code. And to go a little bit into how DeepSource detects these issues is we parse ASTs. So when we get your code, we convert that. The analyzer converts that, converts your code into an AST. Your entire project into an AST form, and our analyzers walk on that AST to find these patterns. So each of these checkers runs against each of your files to find whether any of these patterns are actually existing. And that's how we define an issue on DeepSource, that, “Hey. Okay, if this is a problem, this is how this is defined here as part of code.”

[00:08:25] JM: Are there particular languages that are easy to do static analysis for?

[00:08:31] SS: Well, a lot of languages do come with an inbuilt toolkit or maybe a standard library, which provides an easy way to kind of walk over the ASTs. So for example, Python has a really good way of doing that. Go, for example, Go has a standard library that allows you to do that. So static analysis generally happens via AST using ASTs, but there are a bunch of other tools as well. The bunch of other different kind of techniques that you can use that, for example, you can use regex as well to find patterns in the code. But in any popular language that has a rich tool kit to operate on ASTs, they are very easy to do static analysis on.

[00:09:19] JP: And yeah to add to that, one other thing is there is this misconception that, for example, like static analysis might be very helpful for dynamic languages like Python or JavaScript or something and you don't really need it for languages like Rust. On the other end, with the help of static analysis, you can actually find – Like in Rust, there is this tool called Clippy that actually finds like around 150 types of issues. I'm not sure about the accurate number, but something like that, like very large number of issues just using static analysis. Yeah.

[00:09:51] JM: How do you crawl the AST? Are there standard libraries for doing that or do you have to write your own code to crawl the AST?

[00:10:00] SS: So, generally, popular programming languages like Python and Go and JavaScript generally have something in their standard library that enables you to transform source code into ASTs and then gives a toolkit to write crawlers, walkers on ASTs. In most cases, for example, in Python, the inbuilt toolkit is kind of rudimentary. So what we do is we kind of build on top of. Like we built something on top of the standard library and we use a third-party library called Asteroid. And then we build something on top of that that suits us. We have an internal framework for writing these analyzers, for writing these checkers. So that kind of takes inspiration. Builds on top of that so that we can do more sophisticated analysis. But generally, yeah, most programming languages do give some toolkit. But to do anything sophisticated, you will need to build things on top of that.

And that's what we have done. We have done that for Go. We have done that for JavaScript. And the latest general availabilities that we did for Ruby, we have done that for Ruby as well.

[00:11:09] JM: So I'd like to go into a little bit deeper of an example. So let's say, again, I've got a Ruby codebase and I make a pull request and I'm on GitHub. What is the static analysis tool doing when I make that pull request?

[00:11:25] JP: Sure. So the first thing that happens is GitHub – Since we have an integration, GitHub notifies us that, "Hey, via webhooks, that there is a pull request that is made to this repository." So what we do is our analyzers know that, okay, the change set is from, say, this commit to this command. We first pull the complete repository, because though the issues would be shown only on the diff, we still need the whole codebase for us to understand the full tree of the code. And after we pull the repository, our analyzers works with the AST, and we have predefined rules that are being continuously updated. For example, Python Analyzer [inaudible 00:12:02] 3 – supports somewhere around 570 different types of issues. And each of these are like different rules written. And what happens is the AST first will be parsed and then these rules will be matched against. And these are not very simple rules, but complex rules for every single issue. And we keep adding new rules once a while.

So the moment if there is a match that, "Hey, we think that this kind of pattern, if we see in an AST, it could be an anti-pattern. Or this could be a performance issue." And the analyzer reports the same back to our Django server and then it will be displayed in the UI. And we immediately notify back GitHub as well, say, "Hey we did this and our analyzers found these many issues." And that link is generally posted in the PR. You click that link. You'll be taken to the DeepSource dashboard for that specific run and will show that, "Hey, you made change to, say, two files, of which we found these five, six issues."

[00:12:58] JM: And so what's the process of remediation in a little more detail? How does the pull request get – How does a fix request get generated and how do I accept that?

[00:13:10] SS: Right. So DeepSource has a feature called Autofix. So some of these issues that we detect, and right now our coverage of Autofix is like around 25%. So we can automatically fix 25 of the issues that we detect in your code. So if we can detect an issue in your code, then you'll see a button called Autofix on the DeepSource UI. And that UI is directly – You can navigate to the DeepSource UI for a pull request from your GitHub pull request. So we send a check for each analyzer that you have enabled. So when you go to the DeepSource UI and you see all the issues that we have detected in a pull request, if an issue can be automatically fixed, you see an Autofix

button. You click on Autofix. You select all the files that you want to run Autofix on in the pull request and DeepSource takes like a few seconds and generates the fixes for you, right? And shows you a diff with each fix as a hunk. And what you can do is you can go ahead, you can triage that diff and you can select or deselect some hunk if you don't want to apply that fix. And once you have reviewed that, in one more click, you can just make a comment in that pull request with the fixes. This is what happens when DeepSource can fix an issue. If DeepSource cannot fix an issue – So it will show you how to fix that issue with a very helpful description. So then in, general, as a developer, what you do is you go and look at the description. You know how to fix this issue, and then you make a comment fixing that issue, which will trigger DeepSource to analyze your entire chain set again. And in this case, DeepSource won't find that issue again because you have already fixed it. And that's how the workflow happens. So this is how you remediate either automatically using DeepSource's Autofix feature or manually by actually changing the code and pushing another commit.

[00:15:20] JM: It seems like there's some subjectivity in what qualifies as a bug or an error. How do I configure DeepSource to decide what is a bug and what is not?

[00:15:33] SS: Right. So by default, DeepSource has these definitions inbuilt. For each analyzer that you enable on DeepSource. We have a set of issues that we detected. These issues are categorized as these different categories, right? So when your team, let's say you start using DeepSource and then when the first time that you run DeepSource and you see that hey, "DeepSource has detected these issues." Your team can go ahead and triage these issues and, say, disable some issues if you don't want to or not do that, right? So you can create these sophisticated ignore rules. So you can say that, "Hey, ignore this issue in only my test files. Or ignore this issue in only this particular file pattern." Or you can say that, "Hey, ignore this particular issue across my entire repository." So that's how you tell DeepSource that, "Hey, which issues or which kinds of issues are important to you?"

And in each repository settings you can explicitly declare that, "Hey, which categories of the issues do you feel are important?" You can turn off style issues, for example. Like if you're using an automated code formatter, you can turn off style issues so DeepSource won't report any stylistic issues, because it then understands that, "Hey, you are using an automated code formatter." So that is how you tell DeepSource what do you care about.

In terms of the issues itself, we don't have the ability for you to define a new issue at the moment, but that is going to be in the future soon so that you can define your own issues, which you care about and start using DeepSource to analyze those.

[00:17:31] JP: Yeah. And to quickly add to that, though we don't allow you to create custom issues primarily because it is very difficult for like someone to write a rule that has very low rate of false positive in all the edge cases. But that's said, we do have a community where we our analyzer team is very active. And in case if you find some issues that DeepSource is not detecting, you can just let us know in the forum and, yeah, and we will take care of it. And if it makes sense, we'll implement it.

[00:18:02] JM: What's your practice for keeping up with all the integrations and different analyzation, analysis paths that you have to keep up with in order to give coverage to all these different languages and frameworks?

[00:18:15] JP: Sure. So one thing which we consciously did was when we started DeepSource around one and a half years ago, we started with only Python. We didn't want to support all the languages in a shallow way. Rather just get one programming language and go really deep into it. So we started with Python. And like six months down the line we added support for Go. And even now DeepSource supports like Python, Go, Ruby and JavaScript. That's it, like we don't really support a lot of languages.

On the other end, the reason again is that we want to detect as many issues as possible and we want to improve some coverage for Autofix and all these things. So the way we keep up with each of these languages is that I'll give an example. For example, say, when Python 3.8 was released, like somewhere around a year ago, we added like around eight new rules that was like specific to that language. And yeah, our analyzer team is like continuously evolving. And we are like very active in the language communities and forums. Whenever we find something like that's interesting, we probably implement it.

And one of the great ways of finding these issues are GitHub issues of these static analysis tools that are open source. So they can't easily fish for like some of the things which the tool or the maintenance things that probably they shouldn't make it as a part of that tool. Our team, if they find it valuable and if more developers really want it, we'd go ahead and implement it.

[00:19:38] JM: You have a pretty large engineering team at this point for just a two-year-old company. Tell me about the process of scaling up the team.

[00:19:46] SS: Right. So the engineering team is divided into two parts. One is the language engineering team that explicitly works on – That only works on the analyzers. And the other is the

platform team that works on the platform. They build the APIs. They build the frontend apps, the web apps that DeepSource uses, the DeepSource serves.

So the way that we add new people in the team is for – And the way that we hire people for the team is different for these two teams. For the language engineering team, we look at people who have worked with programming languages before. So currently in the team, we have people who have contributed to, say, the core Python, C Python itself, or they have contributed to popular open source static analysis tools, which is where we kind of go and look at people who have contributed. And then we reach out. We do a lot of outbound reach out for the language team.

For the platform team, we look at people who have worked on developer tools, because the kind of product – Our users are developers. So we look for people who care about developer tools who have worked on developer tools before and have some evidence that they really care about the developer ecosystem itself, right?

Generally, what we do is most of our hiring happens inbound and through reference, because both Jai and I, we have been part of the startup community for quite some time now and we have our own network from people who we have previously worked with. So far, a lot of hiring has happened inbound through referrals. But mostly since we are a developer tool, we have a lot of visibility in the developer community. And a lot of hiring comes – A lot of new leads in hiring comes from inbound that people who are our users, and they see that, “Hey, DeepSource has these five open positions.” They reach back and they say that, “Hey, I'd like to work with you.”

[00:21:48] JP: And in terms of our personal experience as well, in the beginning it was Sanket and I. Both of us were technical. We were writing the code. And like as I said, we start with just Python. So we had to – Like as Sanket mentioned, in our language engineering team, we want like people who are really, really specific in the language and we want someone working on every language all the time. We don't want like one person working on three analyzers at a time. So that's the reason, depending on the number of languages we support, we’ll need to add bandwidth there.

In terms of the platform engineering team, to be honest, in the beginning, we actually had like one frontend developer, one backend developer, one person to take care of infrastructure. What started happening was in case like the users were growing and every day there were like many feature requests and everything coming up, and what we realized was we had to – We value support a lot. So whenever somebody raises an issue, we generally try to reply back as soon as possible. And at this time what we realized is that probably we should add more people to the team. So in case,

say, if one frontend developer is not available, every other thing does not get stuck. So that's the reason. And at the moment we are like around 18 people. So yeah, that was an experience. Yeah.

[00:22:56] JM: Have there been any interesting scalability problems as you've grown the company and grown the project?

[00:23:03] SS: Right. One of the problems, one of the challenges that we had was due to the nature of the product, we started seeing a lot of customers, a lot of potential customers asking for us for an on-premise version of the product. So we actually haven't had any scalability problems for our on cloud version, because we kind of built the infrastructure in a way and the orchestration, the analysis orchestration layer in a way that it is auto scale and we kind of have some intelligent algorithms, which kind of predict how many analysis are we running parallel and are we expecting and then scale the scalar compute like that. But in terms of product scalability, I think that is one of the challenges that we saw that we suddenly saw a surge of people asking for an on-premise version of DeepSource. And we had to make a decision that we had to go on-premise like very, very quickly. And that is what we are working on.

And then, again, even if we had started, like we had kept in mind that we had to go on-premise someday. So most of the services that we use internally, we don't use a lot of external services except for maybe Amazon S3 or SES for sending emails. But still, packaging on-premise has been one of the challenges that we have been facing so that we can scale to larger companies. We can reach out to larger companies and get them to use DeepSource. Yeah.

[00:24:35] JM: And what has been tough about going on-prem?

[00:24:39] SS: Right. So we've kind of figured out most of these things. But probably the toughest thing has been that it needs to work exactly the same way that it works on cloud. And a lot of things, when you build a product from scratch, you don't really – Even if you build your infrastructure in a way that you have to go on-prem someday, which is what we did. There a lot of application dependencies which need to be figured out. You need to figure out how you're going to sync your data, for example. How you're going to manage analytics. How you're going to send that analytics event or dump of those analytics events back to you. Because in some environments, that might not be allowed, right? And there can be some unexpected things that we take for granted. For example, if you're running a SaaS company, you might just take SES or Mandrill for sending emails for granted, or S3 for granted. But when you're moving to on-premise, you can't use that. So you need to have a solution on-prem, something like MinIO maybe for S3. And then you need to figure out, “Okay, how do you send emails without hitting Amazon SES?”

So most of the things have still been figured out, like we use Kubernetes internally for our application orchestration. So it's very easy for us to package the application. But then some of these things, the point-based solutions that we internally use, we need to figure that out. And in the end, when you ship your software to on-premise, it kind of becomes a black box, because you don't get access as easily to the on-premise setup as you get access – As you see that you get access to your own cloud. So it might become operationally very heavy to provide support or to debug things. So we will end – You need to end up building things to get that data out or get, say, logs out or figure out, "Okay, how to debug when some things are going on?" So these are like the major pain points when you ship some software on-premise like we are doing right now.

[00:26:52] JM: Did you look at any of these companies that help you with packaging, like Replicated?

[00:26:58] JP: Sure. So we are actually working with Replicated. We started around a month ago. So again, like when you look at – So how we started was like we have the site called enterpriseready.io., so which kinds of lists. Like these are the 10 things you should have if you're like taking your product on-prem. So yeah, Replicated was a pretty good solution. I think it was used by companies like NPM and Travis. So yeah, we're working with them. And on our end as well, since we deal with source code for these large companies, security is very important. That's the only reason why they're looking for an on-prem, nothing else. So they don't want to ship their source code outside.

So from the beginning, like we use like things like Kubernetes. All of our infra is on Kubernetes. So we can basically reuse whatever we have and just package it with Replicated. So we just started the process. Ideally, we should be able to like package something in the next one month or so. But yeah, it's a pretty good tool.

[00:27:50] JM: Let's say I've already got my engineering workflow built out. I've already got my continuous integration tools. I've already got my static analysis tools. How am I going to fit in another tool? What's the process of getting DeepSource into an existing workflow?

[00:28:05] SS: Right. So this is something that we had foreseen when we started the company, that in addition to being very easy to use, that was a requirement. It should also be very easy to set up initially, right? So let's say you are already using – And this is this is a general case. This is how people use some open source static analysis tools today. You have a CI setup. You are either using CircleCI or GitHub Actions or Travis, and if you are using, say, something like Python or

JavaScript – In JavaScript you are using ESlint, then you are either using ESlint as a build step so that ESlint or, say, something in Python for example, PyLint, that tool would fail if it finds an issue during – As a build step. Or you use pre-commit hooks. That before someone commits, you run these checkers, right?

So if you're using DeepSource, you actually don't need to do any of these things. You can just take these steps out and DeepSource itself is very easy to configure. So all you need to do if you are hosting your code on GitHub, you sign up using GitHub on DeepSource. The second step is you enable DeepSource on your organization if it's a team or your own personal account. And that's pretty much about it. The next step is you go to deepsource.io, you look at your repositories will be automatically synced on DeepSource. So you click on a repository. We have a wizard that automatically generates configuration as a single file. So you enable this and that's it. Aand the next step that will happen is the next pull request that you make, DeepSource will start analyzing that pull request.

So in terms of the workflow, your workflow does not change at all, because you're anyways using a CI tool. DeepSource runs in parallel to your CI. We have our own orchestration. We have our own run time. So you don't have to set up anything. You don't have to install anything anywhere. You don't have to do anything on your CI. Just in a couple of clicks you integrate and deep source will start running in parallel to your CI. So if you already have something set up and if you don't want to throw your existing things out, you can just start using DeepSource without doing anything, without changing anything in your workflow. And then later when you're seeing value, and that's how a lot of teams that use DeepSource do it, later when you have seen value out of DeepSource, you don't need to use any static analysis tools, because DeepSource anyways covers everything that you anyways used before. Yeah.

[00:30:50] JM: There are these different generations of static analysis tools that have existed over the years. It seems like every two years a new static code analysis tool emerges. What are the advantages you have of starting in 2018? Are there some technological advance advances that have come that have made it easier or better to build a static analysis tool today?

[00:31:13] SS: The technologies that we are using, and it might sound weird, but the technologies that we are using today like AST parsing, abstract syntax trees and CST, concrete syntax trees for Autofix, these have existed for quite some time, right? The technologies that we did benefit was, say, using things like Kubernetes, which allows us to segregate the analysis environment so that the clients that we have or the customers, our customers have complete security, right? In a previous generation if we had built this we probably would have used some shared runtime

environment. So that is one. But the primary thing that we did was we looked at the user experience. And there are a few things that, for example, things like machine learning. So we do use some machine learning to learn from your behavior and prioritize issues based on what your previous behavior was. So there have been some new technologies that we are leveraging, of course. But the core to deep source has been there for quite some time. Just that what we believe is existing companies who try to solve this or existing people who try to solve this, they did not push the limits of these technologies far enough. And that is something that we believe we are doing. We are trying to push the limits of static analysis. We are trying to push the limits of what can be done with technologies like CSTs to do Autofil. And I think that's what has enabled us to do what we are doing today.

[00:32:48] JM: What's the hardest engineering problem you're working on right now? Is it on-prem?

[00:32:52] JP: [inaudible 00:32:52], yes. And one other thing is improving coverage for all these issues, because though zero percent false positive is not quite possible yet, like at least that's the consensus. But yeah, trying to handle edge cases for all these specific special rules, all these rules we have. That is something that's like challenging. That's something which we are continuously working on to guarantee the less than five percent false positive rate.

[00:33:22] JM: How do you measure the health of a codebase?

[00:33:25] SS: Right. So there are a few factors. When you say good code, when you say the health of the codebase, right? There are two things. One, does your codebase have these obvious problems? And these obvious problems could be performance issues, security vulnerabilities. And then there are some non-obvious problems, like anti-patterns, structural problems or the way that you have written code. The way that you have designed code. The way that you have organized your code. So that contributes to technical debt. And again, technical debt can be subjective and objective. We mostly deal with objective technical debt. And the second part is your key metrics of code.

Again, these key metrics depend on teams. But some of these are generic. So things like test coverage. Things like documentation coverage. Things like dependency hell. How many external dependencies do you have? And then there could be some specific things. For example, if you're shipping a front-end library or a CLI tool, you might want to ensure that the size is capped. Like you can't ship a 1MB JavaScript file, because nobody's going to use it. So it generally depends. So

some of these things are objective, like, “Okay, are there any obvious security vulnerabilities? Are there any obvious anti-patterns or performance issues?”

The second part would be objective metrics. And the third part would be subjective kind of contextual metrics depending on what you are building. And your code health then kind of becomes an aggregate of all these things. Do you have readability problems in your code? Do you have lack of documentation? Do you have lack of test coverage? And depending on your use case, do you have a super heavy CLI that you're shipping?

[00:35:18] JM: Tell me about what's in the future for DeepSource. Where do you expect the company to grow into and expand into?

[00:35:25] SS: Sure. So we like to think about this as what we are doing at DeepSource. So we like to think about it in three phases. The first phase is, again, the mission of the company is to help developers write good code. So how we think about it is in three phases. The phase one was to build a tool that helps developers find the issues in their code. And that's what we started with two years back. Phase two is automatically fixing most of these issues with human approval. And that is where we are today. We have an Autofix that we launched like six months back that helps you automatically fix issues in your code in a couple of clicks, right?

Where we want to be in the future in the next two or three years is our phase three, which we call Autofix on autopilot, right? What that looks like is if you're using DeepSource, DeepSource sits in your development workflow. Integrates with all these systems that you have. You have GitHub for your source code hosting. You have tools like Sentry for your bug tracking or crash reporting and other tools, right? Whenever you make changes to your code, DeepSource automatically figures out which issues that you're introducing can be fixed with very high reliability. And it will just automatically go ahead and fix it without your approval. If it can't fix any issues automatically, then it will show you that, “Hey, you need to fix this, and this is how.” And the third thing is it will automatically figure out which of these issues are of super high priority. So think about this, we integrate with something like Sentry and we know that, “Hey, this is where your code is breaking right now, right? And then we tell you that, “Hey, by the way, your code is breaking here. These are the few issues that we think can cause this. We can automatically fix two of these issues.” So DeepSource goes ahead and fixes those two issues and tells you that, “Hey, why don't you fix these three issues?”

So in the future, we believe that computers should help developers write good code. Like when you write English today on, say, Google Docs or Gmail and use things like Grammarly, it's very easy for

you to write good English today with the help of the computer. In the future, we want DeepSource to enable developers to write good code as easily as it is right now, like today, to write English. So that's kind of the long term vision of the company.

[00:38:13] JM: Well, that sounds like a good place to close off. So you think you'll be there in what? Three to five years?

[00:38:19] SS: Oh absolutely.

[00:38:21] JM: Well, guys. Thanks for coming on the show. It's been real pleasure talking to you.

[00:38:23] SS: Thanks a lot for having us.

[END]