EPISODE 1490

[INTRODUCTION]

[00:00:00] SPEAKER: Build automation tools automate the process of building code, including steps such as compiling, packaging binary code, and running automated test. Because of this, build automation tools are considered a key part of a continuous delivery pipeline. Build automation tools, read build scripts to define how they should perform a build. Common build scripts include Makefile, Dockerfile, and Bash.

Earthly is a build automation tool that allows you to execute all your bills in containers. Earthly uses Earthfiles, which draws from the best features of Makefile and Dockerfile and provides a common layer between language specific tooling and the CI base spec. Earthly the builds are repeatable, isolated, and self-contained, and will run the same way across different environments such as a CI system, or a developer's laptop.

In this episode, we spoke to Adam Gordon Bell, who works in Developer Relations at Earthly. Adam is also the host of CoRecursive Podcast.

[INTERVIEW]

[00:01:03] JM: Adam, welcome back to the show.

[00:01:05] AGB: Yeah, thanks for having me here, Jeff. I'm excited.

[00:01:08] JM: So, you are a frequent podcaster, been a podcaster for a while, and I knew that's how you got into your current line of work at Earthly. I want to start by just talking about podcasting a little bit. My personal experience with podcasting is for some reason, my amount of listened, podcasts declined during the pandemic. I became less of a podcast addict and I found that they're becoming less integral to my research and evolution process as an engineer. I'm not quite sure why, but are podcasts to you as dominant and relevant to your career as an engineer as they ever have been?

[00:01:57] AGB: I think so. I find that podcasts are a great way to kind of hear somebody's experience and like to get to know them. So, maybe it's not where I'm going to pick up the technical details of how to use Terraform or something. But you get the kind of personal story about how somebody, where Terraform came from, or something. So, that kind of story aspect of it, I get a lot from that. The thing that I've declined in watching is conference talks. I think they took a big hit during COVID and then a lot of people went to YouTube, or prerecorded. And so, I find myself, like watching a conference talk on YouTube a lot less. I used to do that a lot.

[00:02:39] JM: Would you say that when you're learning a new subject, does your workflow – is it predominantly spent in the code? Or do you think there's a necessity for when you're learning a new technology to surrounding yourself with other forms of educational sources, like podcasts or like conference talks?

[00:03:04] AGB: Yeah. I think like, if you're doing something very specific, it's helpful just being in the code, or looking up this like one specific error. But a lot of times, there's like a larger idea and that's harder to get if you're just like focused on this one problem, right? If you want to understand that whatever – I was trying to figure out how to how Terraform works today. So, it's kind of top of mind. But you need to understand that, it kind of has this Kubernetes like declarative model where it's trying to describe like here's the end result. If I'm just hitting errors, and looking things up, I might miss this bigger picture of like declarative configuration of servers, or whatever. That's the type of thing that I really learn from a talk or from a conversation, as opposed to just like in the dock somewhere.

[00:03:53] JM: So, speaking of where you actually work, which is Earthly. The primary product from earthly is CI/CD tools, and we've done a show on Earthly, we've done a show on a lot of different CI/CD technologies. When you look at the bottlenecks in getting a piece of code into production, and you think about the average CI/CD pipeline, where are the most frequent bottlenecks?

[00:04:24] AGB: I mean, I think it's like developer time, right? So, one thing that's frustrating is like you're working on something, and everything's working, all the tests are passing, and then you push it to Jenkins or some CI server and then if it fails, maybe just something had some sort of timing dependency or something happens and it stops working. It's flaky. Flaky builds are such a huge problem. I think that's one of the big ones.

[00:04:53] JM: So, it's not like a problem with the actual CI/CD tooling. It's more a problem with developers just having manual stages or manual work that they need to do to get the code to production?

[00:05:07] AGB: Let me rephrase it, because I mean, that's valid. What I was thinking is, with containers, we've gotten to this world where your development environment kind of mirrors production, right? Or production mirrors your developer environment, depending on the way you want to look at it. If you're doing things with a container locally, you ship that same container to production. But builds are like this weird world where that's not the case. People, we go to build something in Jenkins, and it doesn't work the same way as it works on your machine. That's strange. Why don't we use the same technology, so that we can run the exact same build that happens in Jenkins or in GitHub Actions like on our local machine. So, this has happened, we've brought prod, and development together. Let's bring it all together. Let’s do our builds the same no matter where we are. That's what I was thinking.

[00:05:58] JM: Got you. Have you done much research or looked into the build and release processes of like Google and Facebook?

[00:06:07] AGB: Yeah. I mean, not specifically. I have not worked at either place. So, Vlad, the founder of Earthly, he worked at Google. And Google has a lot of specific ways. They develop software, internally, they have a very large mono repo, and they have a lot of great tooling built up around that. They have a build process. They have a build solution called Blaze as open source as Bazel. So, the story is that Vlad, when he left Google, he missed having these really fast and distributed builds, so he was looking around for some way to do that. And of course, Bazel is open source. You can use it, but it kind of has all these assumptions about the way that – it expects you to be Google or Google like, that you're using these certain languages that you really invest in the technology. A lot of the way that open source development happens right now, open source development, there's not one giant repo like people are pulling things in from different places. So, the way that Google does their builds is very powerful, but very specific to Google, I guess, is what I would say. I don't know if that's what you were getting at with the question. But I could talk about that for a while for sure.

[00:07:21] JM: Yeah, well, I think, the Google infrastructure for every one movement, I'd say, one of the biggest problems with making that a reality has been that Google infrastructure is very specific to Google and generalizing it has required building new solutions, therefore to, I think, to kind of mimic the

Google build process, Earthly had to build its own technologies and build a new company around that. Maybe you could dive a little deeper into the specifics of the Earthly build process, or the way that earthly looks at a build, and how it compares to the Google build process.

[00:08:05] AGB: Yeah, so Bazel is open sourcing of how Google builds things. It has a lot of great ideas in it, about like specifically describing the various inputs and outputs to various build stages, so that then you can cache them and distributed compute them. But it's a heavy lift, right? It’s built for an organization where they can invest a lot of resources into setting this up. And if you normally use when you're working in a small project, if you usually use your Scala command line, SPT thing to build or NPM or whatever, like, you're going to have to toss those aside and move to Bazel, which people don't want to do, right.

So, the idea that you could build up this big graph of what the computations look like, and that you could make sure that the build is always reproducible, those are great ideas. But can we do that in a way that works in a world where maybe we have a mono repo? Maybe we have a poly repo, maybe we have a combination of both. So, the way that Earthly approaches this is using containers for isolation. We have something called the Earthfile. In it, you specify the build steps, and you can use within it, whatever tools you normally use to build your software locally.

Let's say, you have two services, and they communicate via gRPC, using Protobufs. So, you may have a step where you generate the code based on these Protobufs, then you may have a follow-on step that build service A, and a follow-on step that builds a service B. You would just put those in the Earthfile with those three labels. The syntax would be very similar to Bash, and then you can use Earthly to build those, and when you do, it will do so inside of a container, like an OCI Docker container. And because of the way that you've laid these out with these three steps, it would be able to intuit, for instance, that once it builds the first one, it can build the other two in parallel. And also, because it's inside of containers, you can run this on your local machine, you can run this inside Jenkins, you can run this inside GitHub Actions, or whatever. I mean, you could also distribute it.

So, some of these steps are dependent upon others, some are independent and can be run in parallel. Just by specifying things in this format, there's a lot of power, right? So, you get the parallelism that I described, it's more approachable than some of these larger tools, that larger orgs might use, because it's similar to a Makefile or Bash Script. And you can get caching, so we know, for instance, if the proto files didn't change, we don't have to redo that step. So, we can skip that. We can kind of tell by what the inputs and outputs are to the various build steps, what steps can be cached, and that gives you kind of that power of fast builds that Vlad was looking for that he missed from Google.

[00:11:09] JM: In that build process, can you describe the different programming languages that are being used to parse and execute the build file? Maybe take me through the engineering and the runtime of actually executing an Earthly file?

[00:11:26] AGB: Yeah. So, you have an Earth file, and you put it in your repo, say, these services that I was describing, two services that talk to each other. So, you put an Earth file in that repo that kind of describes these three steps. It's written, it looks, maybe, similar to a Dockerfile or a Makefile. And then you need to install the Earthly CLI, the command line tool, so that's written in Go. And then you could type Earthly, space, plus service one, and then it will start building service one.

The way it will do that is – so yeah, it's written in Go, and it uses Build Kit to actually do the building. So, Build Kit is something that was spun out of Docker. And then it uses run C, basically like Linux namespaces, containers, to create an isolated environment in which it builds the steps of your build process. That's how you're able to know that – like a classic build problem is like, "Oh, there's something on my machine, but it's not on the build server." So, by using containers, we can make sure that you hit that problem before you get there. The early CLI is written in Go, and then it communicates with the back end that does the building, that would be running as a Docker container on your machine, and is also written in Go, and that build engine is called Build Kit, and it's what we'll be doing the build. Is that helpful?

[00:12:57] JM: Absolutely. What if something goes wrong during the build?

[00:13:03] AGB: Yeah. So, I mean, you'll get an error, right? So, say, you put in a test and a test fails, then you'll get an error back at the command line, like, such and such failed. Or do you mean something went wrong with the software?

[00:13:17] JM: No, no. I was just speaking about the debugging process of hitting some build with errors.

[00:13:24] AGB: Yeah. So, I mean, it's basically you're getting back red light, green light. This went well, or it didn't.

[00:13:29] JM: Gotcha. So, if I'm in a typical development environment, I'm probably, you know, building my software and running it and trying it many, many times a day, and in order to make some of that happen faster, ideally, parts of the build are going to be cached over time. How complicated is the caching infrastructure for the build tooling?

[00:14:00] AGB: It's a great question. So, the way I was describing the steps earlier, like where you have your proto step, and then your two services, you're specifying that, hey, these files kind of lead to the – they are what is used to generate the proto code, and then these things depend on it. So, by you outlining those steps, you've actually kind of given us a structure for how we can cache your build, right? So, we know that, hey, if the profile didn't change, we don't need to regenerate the code off of it. If service one, if none of the code that is used to build service one has been built, then we don't need to change that.

When Earthly builds up this graph of how to execute your build, it knows what the inputs and outputs to those stages are, and it can calculate a hash of those, kind of like a file, change hash and then it knows, hey, the last time I calculated the hash of all these proto files, it was x and it’s still x. So, I don't need to run those steps. I can just use the cached result from the previous time. That is how Earthly handles caching. It's able to tell for each step, what's changed, and it actually does it line by line. So, as it's going through each result, it knows the inputs to it, and then it caches those on disk, and the next time it needs to hit that line, if nothing changed, then it can just use the result of that. If something did change, then it's got to invalidate all the previous steps.

Here's the thing, caching is super important for builds, right? If you think about the safest way to build something, if you didn't trust yourself to do caching, would be to build everything from scratch every time, right? But how far do you go back? There are all these dependencies, all this source code being pulled in, from all these places. If you want to fast build, being able to know when you can cache steps and not rerun them is like the key. That saves developer time and that's one of the most expensive things for a lot of tech companies.

[00:16:05] JM: If I'm adopting a build tool, I mean, I've already got probably lots of build infrastructure in place. What's the adoption path for Earthly?

[00:16:19] AGB: Yeah. So Earthly, it's build system agnostic, in a way. You can write your Earthfile. And then where previously, you had a bunch of steps. If you're using Jenkins, maybe your build for the services had a bunch of steps written in Groovy, like in a Jenkins file. You could just instead write an Earthfile. The Earthfile you can run locally, but you can also just call that in Jenkins. That's a nice adoption, because you don't have to actually change your CI thing. It's just like, one day, when you get tired of fighting with Jenkins, trying to push something, like I think this works, maybe you can just rewrite it into an Earth file, and then start doing this.

The nice thing about that is like later, when you're like, never mind Jenkins, I'm going to GitHub Actions, because you have that stuff all in this neutral format, this Earthly that you can run wherever, you can move to GitHub Actions by just now calling Earthly within GitHub Actions. So, that's what we see as the adoption path, like, migrate your build to Earthly or just start a new build there, and then use it in whatever system you're using. It's kind of like a neutral format that you can bring with you. Does that make sense?

[00:17:29] JM: Totally. And I guess the portability, if it becomes kind of an integral system for anybody that's using Earthly, that seems like a sticky element of building a good company. I guess, when I was first looking at Earthly as a company, I guess I kind of didn't realize that it's in a separate state from just CI/CD tooling. It's kind of a differentiated category of software. There's not really a comparable system to Earthly I mean, it's kind of a one of a kind, if you look at the software market, right?

[00:18:08] AGB: Yeah, I think so. And I think that people seem to like it, and then we have ideas where we can offer more value to kind of grow into a commercial entity. So, everything I've described so far is just open source. But I mean, we plan to have value adds on top of that. But I agree. It's kind of like we're trying to make builds nicer, like CI and builds, is one of the areas that really needs an upgrade for developers. It's not a fun area. We're trying to make it more fun. If we can make a developer's life a little bit nicer, I mean, I think there's value in that.

[00:18:44] JM: Now, we've kind of gotten an overview of the system. Let's go a little bit deeper into the context of an actual Earthfile, and I guess how the syntax and the execution differs from other systems.

So, can you, I guess, walk me through the syntax and the typical, like, if I'm – let's say, I already have all my existing build tools. I've already got a CI/CD system in place, and I'm just onboarding with Earthly, what does the process look like? What does my typical first Earthfile look like?

[00:19:25] AGB: Yeah, that's a great question. So, the first thing you're going to do is, yeah, just create an Earthfile. And let's say, when I normally build my service, I type like, "Go". I just do go, do like, go, build, main.go, and that produces main, which is like my executable, right? So, if I make an Earthfile, and because it's containerized, I need to specify what container I should run it in, so I'll just put like at the top, it's Dockerfile, I put from Golang Alpine latest, and then I'll put build, colon, just describing, I'm going to call this stage build. And then I'm going to copy in my Go file. So, I will put like copymain.go. So, it's just saying, okay, I'm going to transfer in to this context, I'm going to do my build in a clean slate Docker context that I've just created, so I can copy in that main files, just copy main.go, and then I'm going to do run, go build main.go.

So, the steps that I've done there are basically the same I would do if I were doing a build locally, like the main.go, except first, I need to copy things in, and I need to specify like a container for that to run it. Actually, you can skip. You can skip specifying the container, and it's just going to be Alpine, but then I would need to install the Go compiler into that. So, that will be step one.

[00:20:54] JM: If I have a really complicated infrastructure, am I going to eventually need multiple Earthfiles, like a chain of Earthfiles?

[00:21:03] AGB: Yeah, you probably will want that, especially if you have like a mono repo setup. I was describing before those two services and the proto files, you can imagine them being in a repository with kind of like three folders, one for each. In each one, maybe I have an Earthfile. This is a possible way you could split it up. You could also do this all in one Earthfile. But three will become 30. So, eventually, it makes sense. Maybe in my folder for my proto file, I have the steps that just generate the code, like run ProTalk, dah, dah, dah. In the one for the first service, I have, how to build the first service and the second one, how to build the second one. And then at a top level, I might have an Earthfile that just has a step called all, and all just builds my two services. And then in whatever CI, I use, I can call that all step. Because of the nice caching, maybe if I make a change to one service, it won't have to rebuild the second service, because using the caching, it can know that, okay, the second service hasn't changed.

So, definitely you can organize things by having many Earthfiles, or you can have one very larger file, depends on how you structure your code, I suppose what would work best.

[00:22:21] JM: Gotcha. So, okay, now we've talked through the basics of how Earthly works. If you look at the company, as need to build, obviously, products that are going to make money, what is that business model going to look like?

[00:22:41] AGB: Yeah. It's important, right? We have investors. I have to get paid. So, we're going to build like CI. We're going to build a cloud-hosted CI solution that takes advantage of the Earthfile to do parallel and very highly cached and very user-friendly builds.

[00:23:02] JM: How has that product development process gone thus far?

[00:23:06] AGB: It's super interesting, right? So, we haven't announced a product yet. But yeah, we're building CI. We have all these people using Earthly, the open source product, and then we can talk to them and see what problems they have. It's super interesting. Builds are just the thorny area. It's like, there are people at every place with more than six developers. There's somebody who's just spent some time, just pulling out their hair over builds. So, there's a long list of things to improve. Yeah, like a lot of our process is talking to these people and trying to fine tune the best thing to build. But also, we have some sort of innate advantages for people who use the Earthfile. The way I described, the dependencies and the graph and the caching, just mean that we're going to be able to offer some pretty cool features as a SaaS CI service.

[00:24:02] JM: Anything that stands out in terms of the development process as far as being interesting? If you want to build a massively parallel to hosted CI/CD system? I guess, I'm not sure if you're directly involved in that, because you're more on the developer relations side. But maybe you have any context for the development thus far building CI/CD?

[00:24:24] AGB: I mean, the interesting thing about CI is like, a lot of times when people are building cloud infrastructure, the worst thing that can happen is some sort of remote code, execution exploit, right? Somebody running, whatever, arbitrary code on your servers or the servers that you're renting. The funny thing about CI, is CI, like it's remote code execution as a service. That's exactly what it is.

So, I think, if you look at the varying Cis there, that's a thing they all struggled with is like, how do we make it easy for people? Give them the fastest server so that we can run our code, but not, open things up too much? Or how do we – I mean, we don't have this problem yet. But if you look at a lot of the big providers, how do we prevent Bitcoin miners, right? Just give away free compute. This is what people are going to do with them.

So, that's a big challenge, just securing things, I guess. Another one is like just securing things in terms of making sure that one person's –people giving you their code to build, that's like one of their most important assets. So, how do you make sure that there's no way for one customer to get access to somebody else's code? Is that something I think that we take very seriously, and that all build providers hopefully do.

[00:25:46] JM: Yeah, definitely worse if person A could put Bitcoin miners into person B's code.

[00:25:54] AGB: Well, I was just thinking more about, like, what if they could exfiltrate people's code? What if there was some cloud build provider, and somebody could run something malicious, that could get access to other people's payloads on it. You have to be – you have to think very carefully about these things. Corporate espionage, or something like that.

[00:26:16] JM: So, your work is developer relations. This is kind of an emerging role, or I guess it's been emerging over the last 10 years. What does the job of a developer relations person look like, for this company in particular? I guess, maybe you could probably highlight some of the things or the frequently asked questions that you encounter when you speak with developers working with Earthly.

[00:26:43] AGB: Yeah, developer relations is super interesting. And one thing I should mention, Jeff, is like, I'm in developer relations because of you. I listened to this podcast. And then at one point, you were looking for guest people, and I started – I did a couple of guest episodes for you, and learned that I really liked talking about technology. So, I started my own podcast, and then eventually that led to this job, where, yeah, I just talk about tech all the time and I really love it. I don't think it's for everybody. So, thank you. I guess that's one thing I wanted to say.

[00:27:20] JM: Yeah, very welcome. I mean, I stole my passion from SE radio, of course.

[00:27:28] AGB: Yeah. So, developer relations, at Earthly, I mean, I think it's unique everywhere, probably. It's a varied role, right? I guess. But one thing that I do a lot, is actually just try to teach people how to use things that aren't even necessarily Earthly. So, we've written tutorials on how to speed up your Jenkins build, or best ways to write Makefiles, or how to use AWK. So, I spent a lot of time writing. When I started in this role, I wasn't totally sure, like, it wasn't going to be that. But Vlad and I, when I started, we had met with some of the investors in Earthly and one of them was this guy, Mitch Wiener, who's one of the founders of Digital Ocean. I don't know, you're probably aware. Digital Ocean, just, they put a lot of work into documenting how to do things on Linux servers. They just invest a ton of money into that.

So, he recommended we do that. Don't just teach people about Earthly, teach people about everything. People who might be potential users of Earthly like, what problems do they have? And how can you solve them? That's what we spend a lot of time doing and it's great. I might run out of things to talk about Earthly, even though it's amazing. But I can always take a new tool, teach you how to use Terraform, or Make or whatever. Yeah, so developer relations at earthly involves a lot of written teaching to other developers.

[00:29:01] JM: So, are there some frequently asked questions that you run into when you're communicating with people?

[00:29:07] AGB: Yeah. If you only ask question is the one you asked how are you guys going to make money? So, we're building a pretty cool CI. The thing that people often ask is, "Hey, you're building things inside containers, can we like volume mount our code inside and it will be faster that way?" Because sometimes people have a lot of code to copy and it slows it down. So, people ask about, in general, can you implement this performance tweak to Earthly that would make it faster for my specific use case? We thought about this a lot, right? We have kind of a soft rule that we won't introduce features that will make the builds flaky, that will make the builds possibly unstable, even if they would make them faster. We try to say that what's most important to us is a reproducible build, and then we work to make that fast.

So, a lot of questions and stuff from people currently using Earthly is around this idea, and how do we teach them like, listen, the most important thing is that your build is always 100% the same. Let's get that on point and then we'll work on performance. So, we talked about that a lot.

Another thing we talked about is, we try to make the Earthfiles very simple to understand it has some simple concepts, copy things in, run things. Sometimes, especially if you're really familiar with Earthly, you might want more complicated steps added. What we think is, it's important that however your build works is very understandable. If you added this complicated step that could make things easier to specify, but it was harder for somebody coming to it to the first time to understand, we prefer to rethink that, right? We try to put our focus on what we call approachability. People don't spend all day in the build every day. We want it to be like when you go to it, you understand it. So, we spend a lot of time teaching people about that.

In general, that is a thing. We have a certain approach to doing builds and we're trying to teach it to people. We put a lot of thought into it and there's some best practices. But we're trying to teach people the Earthly way, I guess. One other big question we used to get a lot was why are you using a BSL license? We used to be licensed under BSL. BSL was the Maria DB license, the business source license, and that was popular with open source databases. So, BSL is very much close to an open source license, except with one excluding principle that says, you cannot compete with us commercially. So, this is a license that was born out of Amazon, basically taking open source databases and running it on AWS and kind of leaving them no commercial –

[00:31:55] JM: Right. It’s the the thing that Kafka and Redis and Elasticsearch use.

[00:32:00] AGB: Yeah. So, there's a couple of varieties. I don't know which is which. But yeah, it's like open core model, right? Is it open core? I'm not sure if the definitions. But all these licenses were made with this idea, let's prevent Amazon from eating our lunch. Originally, we’re using that license and we actually got a lot of feedback from the community on it and we switched. So, now we're using the Mozilla public license. Basically, it's a standard open source license. We were worried about making the change. But I think it was a great decision. Do you want me to explain why we made that decision?

[00:32:36] JM: I assume, so that Amazon couldn't just jack your product and make a CI/CD product based off of it?

[00:32:44] AGB: Well, yeah. So, that's why we originally chose BSL, but then we switched away from it. Now, we're using MPL. Basically, we think that the risk is low of AWS, like jacking our product, as you said, like databases are clearly within their wheelhouse, build tools are less clear. There's a cost to doing these things, right? If it's an impediment for adoption, like if the developer starts using Earthly and then realizes like, “Oh, I have to go talk to legal at my company”, you just lost the battle right there. So, there's a cost. It's friction to the adoption process, and for us, you have to weigh those things against the risk.

[00:33:23] JM: So, if someone comes up to you and they say, “Look, I like the idea of having a new build tool, but we really don't have the developer resources to invest in migrating to it and changing our whole build pipeline.” What would you say to them?

[00:33:41] AGB: I get it. Builds are not fun. So, migrate. If it's working, keep using it. But when you go to make a change to your Jenkins build, or your GitHub Actions build, and something doesn't work, and then you're like, “Okay, I think I fixed it”, and you commit that, and you push that and you wait for it to run, and then that fails. When that starts happening, you very quickly, the end of the day will happen, then the end of the week, like time just disappears into this huge feedback cycle.

I mean, when you get into that mode, just take a half day, look at Earthly, your time will be paid back. If you're just able to get to a build that you can run on your local machine without these big feedback loops, I think that you'll pay off that investment so fast. But I would also say, if you're working on something new, doesn't have a build yet. I mean, that's also a great opportunity. Look at those times. You can keep using Jenkins or whatever you're using, but maybe, look at when you're having these slow feedback cycles, if you can introduce something like Earthly to kind of just get off that push and pray cycle, which is so painful.

[00:34:55] JM: Yeah. Well, if you were to estimate how much developer resources it takes to migrate a build tool, or migrate to a build tool, and how you would propagate that through an organization, are there any playbooks you've seen for how a large organization would migrate entirely to a new build tool?

[00:35:17] AGB: Yeah, so a lot of times, there's like a build guru. There's somebody within the organization or within the team. I mean, if you have simple builds, and they don't do too much, then it doesn't matter. Everything's simple. But usually things get more complex over time, and so, not everybody knows how the build works. So, there's a build guru type of person who has to do these changes. And then if something goes wrong with the build, he becomes kind of like the bottleneck. So, a lot of times, the way a change gets made from one build system to another, is that build guru becomes a huge bottleneck. Somebody who's maybe a team lead, or an engineering manager, who's maybe still a little bit in the weeds, it gets on the radar. This is a problem. We have this bottleneck here with our builds, whatever it is, right? People are waiting days to get things merged in, because there's this backlog of builds, or something's down and this isn't happening, or the builds just take hours, and this bottleneck gets identified, and then bring together people to discuss about what's to be done.

A lot of times, the smart way to approach this is not to wholesale, try to change everything everywhere. But to pick a place where you know, you can kind of do a trial run of something. “Oh, maybe we should try Bazel. Maybe we should try Earthly. Maybe we should try GitHub Actions in this one project and see if it alleviates our problem.” “Oh, maybe we should try this caching strategy in this little area and see how it goes.” And then from there, things can spread. So that's, I think, the smart way to do adoption is like a slow rollout, where somebody's trying it out, let it bake a little bit, and then try it a couple other areas, and then slowly roll it out throughout an organization.

[00:37:01] JM: If you take a step back and look at the long term, direction of the company, what's the most direct comparator? Would it be Like CloudBees or like Circle Ci?

[00:37:14] AGB: Yeah, probably. Either of those. I mean, I guess CloudBees is more, like Jenkins existed, and they made a cloud version of it. So, I'm not sure that – actually, I don’t want to say that they're not innovative. I was just thinking, Circle CI seemed very innovative, so I think that's a comparison point, for sure.

[00:37:33] JM: Yeah, well, I think, with Jenkins, it was just an older product and it was probably harder to iterate on. Circle CI, definitely more innovative, but it does seem to be really hard to mature out of your core product as a continuous integration company. But you know, it's one of those things, it's such a valuable piece of infrastructure that the churn is really low. So, it's at least a really good business. But when you look at the long-term direction of other problems that the company could tackle, are there other areas of infrastructure that you can imagine expanding laterally into? Or do you think just the CI process is so deep, that it will be something you can spend an eternity on?

[00:38:22] AGB: Well, I think that the directions of expansion are twofold in my mind. So, there's CI, but then you still need to deploy things. How do you actually get things into prod? Not just building them? So, that's a direction. So, going towards production, there's lots of that can be improved that way. And then the other direction is towards dev. How can you improve how the developer does their job? Why do they even have to commit to build something? Why can't I be doing some sort of distributed testing of all the services on some fancy machines in the cloud, while they're sitting there on their local machine? Just a random idea.

But I think that there's lots of that can be improved in both directions. The developer process on their local machine, as well as continuous deployment, and then these weird in between areas. People have these staging environments that everybody pushes their code to and tries to integrate. How do you test services without having like a whole replica of prod for each developer and so on? There's so much, honestly, I could spend forever on it.

This area, like needs even more companies. If developer time is valuable, then the more time we can spend polishing these things, and taking away these paper cuts from the development process, the more valuable. I think that, okay, slow build costs you 15 minutes, or whatever amount of time. But it really costs you more than that, because there's kind of like this interruption of flow, of whatever you're working on. When something gets like more than five minutes, then I'm going to like go grab coffee or something, and then I get sidetracked by Slack. It's just the time that's lost by distraction by slow processes that don't have to be. It's astronomical, I think.

[00:40:18] JM: If you look at your own history as a developer, are there places where you've worked where things would have been alleviated by having a better build system?

[00:40:31] AGB: Oh, yeah. A previous place I worked at, we had this build, and to prevent problems from occurring in production, we would keep adding things. We add some integration tests, and then some of the integration tests depend upon some third-party environment, and then it would fail. So, I just remember having these builds, and they would take 30 minutes, and then they would fail once in a while. Not even once in a while. It was like, for a while, it seemed like they would fail one out of three times. And then it was like, you could take the time that we were in – I mean, this is like, bad teamanship, right? But nobody wanted to take the week and figure out what was going on there, or, like, the couple days or whatever it was, so everybody, your build fails one or three times, you just hit rebuild, and then it'll work the next time.

Eventually, we got in there and we found out what was wrong with these integration tests. But the time we spent, it was like, nobody wanted to be the person to own that to get these tests working. It was just a huge problem. And then there was another team at this place, who had more of a mono repo approach and they had a long PR backlog and then the PRs, to merge them, after people reviewed them, they needed to be merged. And then meanwhile, there'd be other PRs made. So, they needed to like rebase them, and then a new build would be needed. They just started having this world where they were fighting to get code merged. People were racing to like merge PRs, before another PR got in, which triggered another build. It was insanity. Literally, a zero-sum game of a whole bunch of developers competing to who can get their thing merged, so they don't have to rebase and build again.

I've seen so many horrible practices where there's just like a build problem in people. You don't want to invest the time in it, it's not what you're measured on. You're measured on shipping your feature. You don't want to dig in and fix this problem for everybody. Of course, people do eventually, right? That's when a developer can really show leadership, but we shouldn't be dependent upon that. We need better systems.

[00:42:39] JM: Well, anything you want to close out with?

[00:42:41] AGB: No. Thanks for having me on. This is fantastic.

[00:42:44] JM: Cool. Pleasure, as always. Thanks, Adam.

[00:42:47] AGB: Thank you.

[END]