EPISODE 1215

[INTRODUCTION]

[00:00:00] JM: Build automation tools automate the process of building code including steps such as compiling, packaging binary code and running automated tests. 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 of your builds in containers. Earthly uses Earth files which draw from the best features of Makefile and Dockerfile and provides a common layer between language-specific tooling and the CI build spec. Earthly 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.

Vlad Ionescu is a founder and CEO of Earthly Technologies and he joins the show today to talk about why reproducible builds are important and how Earthly simplifies build scripts as well as with the long-term vision for Earthly looks like.

[INTERVIEW]

[00:01:00] JM: Vlad, welcome to the show.

[00:01:01] VI: Thanks for having me, Jeff.

[00:01:04] JM: Let's say I’ve got a software project and that project is getting really, really big. How does the build for that software project start to have problems?

[00:01:14] VI: Well, there are many ways. Usually if you don't maintain it quite well the build can take quite a lot of time. And so if you go over maybe 20, 25 minutes, you're starting to hurt in the productivity area and you're starting to optimize the way the build works. So the way you might do this is split it in like parallel builds or find ways to cache as much of stuff as you can.

Definitely you're hitting some scale issues and there aren't many products out there that are available either for free or easily deployable that can really magically take care of this for you. So really you have to kind of invest early on in some way to make your build scale with your project. And this could be like good engineering practices like splitting your project into components that build independently, are maybe tested independently before they are tested together, or use a bunch of tools that help you parallelize your build. A lot of the CI’s out there allow you to perform some of the steps in parallel. You have to kind of configure them to work in the right way. It's a lot of do-it-yourself kind of work. But with the right tools, you should be able to make this work over time.

Of course at a certain scale, I don't know, I’m not sure what part of question I should answer better. But if you get to like, I don't know, 50 plus engineers on a single project or even more, you're starting to have yet another set of issues where maybe traditional tools aren't really helping you all that much and you need like some kind of specialized build tools to help your team stay productive in a very active kind of project with a lot of contributors.

[00:03:14] JM: You work on Earthly which is a project based around builds. Can you explain what the inspiration for earthly was?

[00:03:22] VI: Yes, absolutely. So in my past I’ve worked at companies like Google, and one of the things we had at Google was this build system. It's called Bazel in the open source world, but in Google it's called Blaze. And one of the things that was interesting to me with this build system was that everything that you would build would use some kind of compute that would be in the cloud and it would parallelize everything for you, would take care of a lot of caching almost magically, and it had this very different way of working than you have out in the open source world. And that different way is more like a constraint resolver. A set of things that lead to an end result and the system can compute like a graph of dependencies and automatically figure out how to get there and how to cache things along the way. Versus what's in the open source world, which is more like a pipeline set of steps that go one after the other and they don't necessarily are aware of each other. They just know that they need to happen in a certain order.

So this system at Google was really, really efficient. And at the time I really thought of it as something quite alien, right? Alien in two ways, in a good way and in a bad way. And the good way in that, the build system would be like very advanced. I don't think there is anything more advanced on planet earth today. It would be like magical to every engineer working at Google. It would just work be super-efficient, super-fast and reuse as much as possible of the cache. But it's also bad in the sense that once you go outside of Google in the environment they have set it up, it can be very tricky to make it work with a regular project.

So one of the things you have to do, for example, let's say you have a Java project. You have to take your existing build files, your, I don't know, palm.xml or build.gradle or what have you and translate that into Blaze. And for a lot of people that's depriving them from the tools they've always known and always loved. And for this reason it feels very alien to an open source person or a person that's used to consuming open source and doesn't quite fit to the traditional tools that you're already used to. And the inspiration for Earthly was what if you could make these two worlds work together? What if you had most of the benefits of what Blaze can do, but not be constrained so much by the open source world and not working with open source? What if you could just leverage the open source and all the specific language ecosystem tools that you are already used to and take those and leverage those in the build system? And at that level, really, the build system becomes like the glue layer. It's doesn't go down to the compiler level. It doesn't have like deep integration into the compiler necessarily like Blaze has, but it is providing a bunch of very good things you can do with it like still maintaining some kind of reproducibility. If you maintain the same inputs, you would have the same outputs.

And so through the means of containers, this is what Earthly does really. It uses binaries that you're already familiar with and takes those, puts them in containers, runs them and gives you outputs, and those outputs can be like Docker images or artifacts, regular files, binaries and so on, and basically get the value of reproducibility that Blaze has and of some sort of caching that you get along the way similarly to how Blaze does it, but not at the same fine grain level.

But you get this whole package in an easy way to adapt. Not as complicated as Blaze would be.

[00:07:31] JM: Can you explain how Earthly changes the workflow of a developer in a little more detail?

[00:07:41] VI: Sure. Sure. So our goal is not to change the workflow all that much. Really, as a developer, you work in your IDE and you iterate on a certain code that you're working on running unit tests locally. Maybe some integration tests and so on. And then you put that into a Git commit, for example. Push that to maybe GitHub and then the CI picks it up and executes a number of scripts that somebody has written and verifies that your project works.

Where Earthly comes in is a lot of these scripts that you would execute only in CI, Earthly gives you the ability to also execute them on your local machine. So essentially you write some instructions for the system to understand how to build your project and Earthly containerizes those instructions and runs them for you, and you can put that in CI and it will run typical on top of existing CI's like Circle, Travis, GitHub Actions, Jenkins and so on. But you can also take that same script and run it locally. And what that gives you is if something breaks in CI, the way to iterate on that often takes like a lot of time to recommit everything to Git and re-run it and so on, re-trigger it, wait for it to kick up. It takes maybe five to ten minutes to iterate on every single change if you've got a breakage in CI. Whereas with Earthly, you take that same script that is failing in CI, you run it locally. It's containerized so it runs the same way. And you can now iterate on it much, much faster. Like of the order maybe minutes or seconds for each iteration. And once you've fixed it, then you commit it to GitHub and the CI again picks it up, verifies that it actually works exactly the same way as on your laptop. And your day is that much more productive.

[00:09:44] JM: Just to revisit what you described earlier, can you talk in more detail how Earthly compares to Blaze and Bazel? I know you described it earlier. I just like to double down on this because I think it's important.

[00:10:01] VI: Sure. Sure. Absolutely. So I guess there's multiple components to this really. So one key difference is that Blaze and Bazel is more like a constraint solver. So it picks up all the inputs that you have available like source files and so on and it works its way to a certain output in the end. Earthly is a mix between a constraint solver and an imperative language. So like at each step of your build your instructions are imperative. So it's very easy for a developer to reason about them. They just happen step by step. And at the high-level, all these different targets that work together they're sold with a constraint solver, which essentially is taking all the inputs that you have, like all your source files, maybe input images, Docker images and so on and then solves towards the end result that you're trying to build, which may be like a binary or a Docker image. So that's one key difference.

Another is Blaze has this very deep compiler level integration which allows Blaze to be very fine-grained about what exactly has changed since your last build and reuse a lot of that cache. So the end result of that is improved efficiency of the way you compile things and build buildings. For Earthly, there is no such a fine-grained level of detection. It works very much like a Docker file-based cache. Every instruction is a layer on top of another layer. So if one copy command detects that one of the files has changed, the execution restarts from that point onwards. It's not quite as fine-grained, but the benefit you get with Earthly is that it's a bit loose on this level of determinism and reproducibility that Blaze has, but it allows you to use all the tools you have today. So it makes this tradeoff, this sacrifice of correctness for the benefit of interoperability with any other tool. And for us we think it's much more important. The level of reproducibility you get with containers is still very high, and for most people that's all they need. And I guess in this day and age where open source exploded and there's so many tools out there for every single language, it's that much more important for your build tool to actually work with the existing ecosystem out there, and that was a high priority for us.

[00:12:47] JM: Tell me about the syntax that is used to describe a build in Earthly.

[00:12:52] VI: Sure. The syntax is we describe it typically as kind of like Makefile and Dockerfile had a baby. And the reason for this was that in the past we've written a bunch of – We always wanted containerized builds like in in many environments I've worked in, and we could never get there. And what we had to do was combine some kind of Docker file with a Make file and a lot of glue code in between and so on. And we realized there is a pattern here. We keep writing these Makefiles and Dockerfile pairs in every single project. And this is kind of where Earthy took off in the beginning where we said, “What if the Makefile and Docker file were actually merged together and so every instruction you would have in the makefile would actually also run in some kind of a container?”

So like typical problems we had in the past was that if you ran like a Makefile on a Mac versus on a Linux machine, like some instructions would run differently like said or find. Typical Unix tools that you're used to they have like very small differences between them that caused the build to fail. And we said, “What if you just took every single instruction in a Makefile and you run it in a container?” And really when you have this level of isolation between targets in a Makefile, you are freer as to the side effects of what different targets can do to each other. A typical side effect of a Makefile running is one target trembles up on another or like the order of execution is wrong or like you can never enable parallel mode in Makefile because it's just so brittle. Nobody really tests that. It's just so difficult to work with it.

So on the other side, if you make every target isolated in a container and every instruction being isolated from another, it doesn't really matter that they run in parallel. There is no way for them to interact with each other. And then on this other side we have Docker files, which are super popular, super successful. An excellent language that is able to produce Docker images. And we took these two, put them together, and what you have is basically a bunch of targets that look like Makefile targets where the instructions within are actually Docker file instructions. And if one target depends on another, there's like clear ways in which they do. So you have to declare files that are passed from one target to another. You have to declare images that are passed from one target to another. It's like there is this inner dependencies that can happen. And the system, if you give it like a final target that you're interested in building, the system can figure out automatically which other targets to build along the way and builds all of those in parallel in order for you to get your final result that you're interested in. So far, it's working really well. We have really good feedback on it. It's really very powerful. You can do just about anything with it really.

[00:16:05] JM: Tell me about the engineering of building Earthly. What have been some of the difficult engineering problems?

[00:16:12] VI: Sure thing. So we use a lot of the Docker internals actually to execute everything that Earthly does. So some people may be aware that the Docker builder that is used for building Docker files is actually very separate from what Docker uses to run things, to run images. And when you do like a run in a Docker file, it's actually very different from a Docker run outside of a Docker file. Like you don't have the same mounts, you don't have the same, I don't know, ports being opened and so on. In Docker files you have this system of OverlayFS, which is this underlying file system that allows that layer-by-layer caching that takes place in a Docker file.

And over the last maybe two, three years or so, the Docker team has been working on replacing their old Docker builder system with something called BuildKit. It's something they've been working on for a while. And BuildKit is super powerful, super interesting. A lot of new features are coming to both Docker files and Earthly as a result, and this is what we're adopting ourselves. So I don't think BuildKit is GA just yet. I think it's just about to get GA'd or may have been very recently. But essentially we use BuildKit underneath. We operate all our independent steps as BuildKit steps within our builds. There essentially underneath is a series of runC instructions actually. So underneath BuildKit, it uses a runC container executor, which is able to run the independent steps that you need to execute. And that's together with the OverlayFS file system it creates basically a container for every single line that you have in a Docker file or an Earth file. So every run instruction is its own container with its own layer and the system works together to create every single step in the build process.

That together with a graph, a direct acyclic graph, or a DAG as people call it, which keeps track of which part of the build depends on another, and that creates maybe a potentially vast graph that allows you to create dependencies between the different parts of the system even between projects. In the case of Earthly, you can like import other GitHub repositories within your build or execute builds of another repository and so on. And putting this all together in a single graph allows then BuildKit to understand what are the ways in which you can parallelize the execution of this. So this is where it becomes kind of that constrained solver I was talking about where you have a set of inputs and a final place you want to get to and the system can figure out on its own what's the fastest way to get there by traversing every single necessary node of the graph, discarding what's not necessary and paralyzing as much as possible.

[00:19:24] JM: So how does my usage of Earthly evolve over time? Do I write additional build scripts or do I just have like a single build script and I’m just editing that single script over time?

[00:19:37] VI: Well, it depends on how your source code may be laid out. So typically we want people to work in isolated like component-based builds where every component of your system has its own earth file. And then there's other earth files that may be referencing another. So like a great example is actually where you might have a complex microservice architecture with maybe, say, protocol buffers in between. And a common problem is how do you get those protocol buffers from one project to another, because typically you want one protocol buffer definition to be available both to the client and the server.

So just a quick overview, protocol buffers is this serialization format by Google. It's underneath GRPC. GRPC is basically protocol buffers plus HTTP2. And the interesting part of it is that you define a schema and that schema represents what the messages look like. And this schema is shared between the client and the server and that's how they know how to communicate with each other. But the schema needs to produce some generated files for every language. Like if you want to use protocol buffers in Java, you have to generate the Java classes for it. Or if you want to use it in JavaScript, you generate some JavaScript source files that create the necessary objects for you to work with protocol buffers within that language. And the challenge here is often the protocol buffer definitions is in one repository, whereas your client and server may be in different repositories. And it's very difficult for you to like work with protocol buffers consistently across varying languages, varying code bases and so on because it's just hard to import with traditional tools just files, regular files from another repository or build artifacts from another repository.

And a typical way that Earthly helps here is it allows you to reference other builds from other repositories or other directories if you have a monorepo. So either works just as well. And essentially you could say import the very proto definition, the schema of the communication protocol, or the generated files for your language that may have been produced by another build. It's super easy to just copy over an artifact that was produced by another build to your build and just use it right away, because it's all containerized. The way they are generated doesn't really matter where it executes as long as it has the same inputs. So that's one benefit. You can just grab stuff from other places, put them all together and then make them work.

So if you have like a multi-repo set up, that's one way to go. You might have like each repository with its own Earth file and those evolve independently. They serve the different use cases that you might have in those local specific repositories. But also when you want to tie things together maybe in some kind of integration test or this kind of protobuf use case I mentioned or for whatever other reason, you can always have another Earth file reference these different repositories to put everything together and maybe put like a Docker compose up type of setup where you do integration tests and so on.

In a monorepo, it’s very similar. You might have each directory being their own like project, and those projects can evolve independently they have their own Earth files. Whereas when you want to put everything together, you can have yet another directory or yet another file somewhere else that references these other directories and put everything together for you.

[00:23:37] JM: You've mentioned a few architectural features like protocol buffers, for example, and how that can affect your build strategies. Are there any other architectural features you can discuss and how that affects builds?

[00:23:52] VI: Yeah. I guess sterilization formats is a big one. Another is really, I guess, double clicking into integration tests. Oftentimes people do integration tests in various ways. So you might be running all your integration tests in containers, or for whatever reason you might have some parts running natively on your computer, other parts in containers. In fact, talking about workflow for developers on their computers – We've been interviewing maybe 50 or so engineers since we started about how they build, how they work together, how they have their local workflow and so on. And really the thing what we were surprised about was just how diverse it can be especially when you go to the local workflow setup. We all know there is some kind of variance. We know people argue about Veeam and Emacs and Gui and Visual Studio Code or IntelliJ and so on, but really we didn't imagine yet another level of complexity. Even within each of these environments people have their own style and ways of being productive on their own.

And this level of complexity and variance between individual developers that may be on the same team creates a challenge for us creating the best environment that allows you to be productive both locally and in CI in the same way. So what we're allowing more and more now is integrating as much of your integration test within Earthly to create some kind of uniformity between people. So if you have like a Docker compose stack, for example, we now allow you to isolate even that stack in an integration test within Earthly. Reuse as much cache as possible and run that in an isolated manner so that even if you have modified your Docker daemon in some strange way, whatever runs in this integration test is still isolated and would provide the same result as if it were running in a CI. And that was pretty important to us as we were architecting how this might work. I guess as a summary, there's a lot of variance in builds for various reasons between organizations of course, but even within the style of each developer. And creating this uniformity across environments was a significant architectural challenge for us to figure it out and to put it together.

[00:26:39] JM: Are there any outstanding problems in builds that you haven't been able to address even with Earthly? Like problems that that seem solvable but you have not been able to solve?

[00:26:52] VI: There's always something. So one of the challenges is once you put everything in containers, you have limitations as to the platform, the target platform you have working towards. So if you want to build something on a Mac, there is no Mac containers available, and that can be a problem. It's like it may be not the best fit if you are building iOS applications, for example. So if you're doing that, the stack that Apple provides is probably your best bet. We're not sure where we will tackle that soon or at what point in the future. For now we're focusing on cloud engineering specifically. Our team has been –We have extensive experience in cloud engineering and this is kind of the use case we're targeting specifically. And a lot of cloud engineering is Linux, and that is the most conducive for us to pursue at least in the beginning. Never say never. It could be interesting if this were provided even for Mac type of targets. But you can of course use Mac laptops to build Linux containers. That's absolutely fine.

And another area is windows containers. Not many people know, but it is actually possible to run a Windows container that runs your typical windows things, like PowerShell and so on, and these are not running on a Linux container. Like the guest is Windows, which is like a significant departure from how people are aware of how containers work. I've not seen this being super popular. In fact, over the years I've been seeing being less supported by Docker. So for example, the new BuildKit builder doesn't yet support Wwindows containers, although the old builder does. It's up in the air whether this is like a thing of the future where people have just decided Linux is the operating system of the cloud and that's it. So I guess, yeah, anything that's outside of cloud builds for now, it's not quite conducive for Earthly.

[00:29:08] JM: Now there's a workflow that can often happen where like I'm working on my software and then I try to build it and the build fails, and then I try to do what I can to fix it and the build fails again, then I try to do – I fix it and the build fails and I can't I can't seem to fix it. And as I'm fixing it I want the cycle time to be as quick as possible. How does Earthly help with that short cycle time?

[00:29:35] VI: Yeah, sure. So often this is exaggerated by the fact that your cycle may be dependent upon a CI. So the underlying problem here is that it's really hard to reproduce what the CI is doing on your local machine. There have been attempts. So there is, for example, somebody built a Docker container that mimics the GitHub actions environment, for example, and you can submit the same YAML files that GitHub actions consumes. But it's like very slow. Not super consistent. It's like an 18 gig image. So people have attempted to do this. And really most of the CI's that have been built to date haven't been really considering the story of the user on their own computer. And I think that there's a gap, and this gap is what we're trying to address how to really empower the developer to be productive on their own machine when they work on builds that run in the CI. So Earthly is that gap filler in a way that if you write your scripts to be on top of Earthly, then earthly would be able to run the same containers in CI and on your local system the same way. And that's a big part of the way, of course. It allows you to say you have a problem at step number 15, right? If you iterate on that step 15 with every time doing Git commit try again, it's going to take a while until it goes through all the steps yet again. A lot of the CI's don't have proper caching or it's caching that you have to figure out yourself, a lot of waiting even for the VM to kick up to start your build.

What earthy is able to do in this context is before step number 15, Earthly already cached steps number 1 to 14 and you don't have to re-execute those every time. Not to mention the fact that you don't have to commit to Git every time. So iterating in such a situation is so much easier with Earthly because everything is just at your fingertips. You iterate in your ID the same place you're always used to. Resubmit the build in your terminal and Earthly picks it up and you know executes it much, much faster than the CI can.

And people ask us often, especially less technical people who are not really aware of how this works exactly, but they ask us, “What's the time saving difference? How often can CI fails happen?” And it's really hard to give a good number for this or like a good estimate, but compounded with the fact that when a CI build fails, sometimes the entire team is blocked. They cannot make progress, and that can be an issue. Like if you have like, I don't know, seven people in that team and maybe half of them at that point they want to commit stuff in the master branch or main branch, maybe they should wait if the build is broken not to cause more problems to the build person, and that's definitely something that impacts the productivity of the team every single time. If you do it locally, this can be much faster to address. So something that may take half a day maybe takes half an hour. It was a very rough estimate, but of course every failure is its own story, its own journey, and there is nothing quite similar between these failures necessarily to draw a clear estimate for this. But this is what I usually tell people. It’s often the difference between half a day of hair pulling and half an hour of just doing it.

[00:33:29] JM: A topic that is related to builds, monorepos versus polyrepos. Tell me about how repository management works at your ideal company.

[00:33:45] VI: Well, it is a topic of a lot of debate. Sometimes it gets very religious maybe. It's sometimes more like tabs versus spaces. But underneath there is actually a bunch of things to consider really when you decide between a polyrepo and a monorepo. So I've written at length about this in our blog, but essentially one of the things you have to consider is how the team makes progress how they work together especially when they interact between components of the system, between say microservices or different libraries and so on. Often this dividing gap between projects is the gap that gets the least attention. And I think for a lot of teams that can be the cause of many bugs or inconsistencies or outages in production. And really the reason is that often teams are concentrated in areas that are the projects themselves, not the intersection between projects.

What I usually try to do in my engineering teams is to create as many bridges as possible between these different aspects of your infrastructure or your components of software and allow the team to contribute across as easily as possible. Just the simple fact that another project can have a different language that it is written in can be a significant gap for another user to contribute across. Add on top of that being able to install all the necessary tool chain and Java 1.6 or whatever specific Python version 2.7 or the specific version of the tool chain together with the right extension, the right plugins, the right version of I guess Proto-C or whatever type of tooling you might have in your environment that everyone on that team is very familiar with. They already have all their environments set up. But someone else coming to that team and trying to make the simplest integration may have a lot of trouble just simply compiling that project really.

And the fact that it is hard to compile and run the tests of another project and run integration tests and so on can be a big deterrent between teams not being to collaborate with each other. So for this reason I've always tried to encourage monorepos wherever possible in my teams that I've worked with just to help with this mental gap of how do you build this other project?

How do you contribute across? How do you make it as easy as possible for people to collaborate with each other even though they're not on the same team? But there are of course a lot of challenges to making this work. A lot of the open source tooling out there today is not really well-suited for monorepos. And you start with, say, GitHub itself. Although it works really well, we see very large repositories on GitHub. But there's like how do you see differences for a specific project and not others? Like the pull requests can be much more intimidating. If all the pull requests are mixed in, you have to kind of learn to be productive in that environment. Or if you go to CI systems, I think no popular CI is very well-suited for mono repos today. Like you would have to be able to trigger the build based on a subdirectory potentially so that if you're working on your little project you don't trigger the build for every other project as well in a monorepo. So that's a big challenge when it comes to CI.

Another challenge is the caching of builds. If you are creating interdependencies between projects, you want as much as possible to be cached of the projects you have not changed so that you can start testing as quickly as possible in the project that have changed or the areas that need a retest. And today no CI system, no popular one anyway, can give you the answer to the question. Given that I’ve changed this file, what integration test do I need to rerun specifically? And this is where a lot of the big companies have been able to create productivity around such tools as Bazel, Pants and Buck, companies like Google, Facebook and Twitter. But it does require a team that is dedicated to maintaining the system for the engineering team and it requires also for everyone to be on board with this strategy and not go on the side and start creating Gradle files or other builds that are not compatible with Bazel or whatever tool you chose.

So ideally, yeah, in a perfect world, I would love for more repos to be more like monorepos because of the inter connection between engineers. Realistically, polyrepos are very popular in open source and there's a lot of support for small projects being built independently or being managed independently. And if you want some kind of monorepo, oftentimes it's really more like a hybrid kind of solution to the point that your CI no longer scales. You can have a monorepo up to that point, but certain use cases or certain situations or certain technologies will expect things to be separated either for performance reasons or just the way they were designed just because a lot of open source is polyrepo. So your mileage might vary when you make this decision.

Also, if you have open source in your organization, monorepo is definitely not the way to go. Like there is no way for you to open source only parts of your source code in that monorepo. So you have to decide which parts exist in other repositories and which parts are private to your organization. So overall I think there're a lot of factors that can go into this decision. It's hard to give a general advice. It's very dependent upon your specific situation, oftentimes the specific technologies you're using whether they support monorepos or not, they scale well with monorepos or not, and what kind of engineering culture you want to create in your organization as well. So it all depends on every team's requirements.

[00:40:42] JM: As we begin to wind down the conversation, I’d love to get any perspectives you have on the future predictions for how software will change. And given your close-up view on the world of builds, I guess the technology around builds, how that will change.

[00:40:59] VI: Sure, absolutely. There is a lot of ways to go into the future. I think we've seen more and more cloud adoption, and not just from the organization perspective using cloud as a production environment, but also for the development perspective. More and more we're switching from Jenkins to GitHub actions going towards the cloud and embracing these sandbox builders that have start from scratch every time they build. They don't have any cache. The cache they have needs to be downloaded and so on. There's more and more of that.

In parallel there have been attempts to create an IDE environment in the cloud as well, and I think that can be very powerful when done right. Certainly, I’ve seen it successful at Google as well. This is a tool they had as well internally. The ability to just fire up an IDE in the browser, make some changes, submit a PR, or change list as they call it and be very productive that way. There's GitHub code spaces that goes in that area, and I think it's super exciting the way it just picks up your visual studio extensions and settings and just drops them in the browser. It's really great.

One thing that's not great though is currently the performance of that machine that the IDE runs on is not great. So like when you use the terminal to execute builds and so on, it's kind of slow. But I see a lot of potential towards this area. More and more will go towards the cloud, and certainly one of the things we will do as well is offer Earthly as a service in the cloud for you to execute builds natively in Earthly without the use of another CI in your stack. It's definitely interesting to see the way things shape. Containers maybe 10 years ago were just starting up, not very popular at all. They were like experiments of certain companies adopting them and so on. But now they're beginning to be mainstream and it's very – Before maybe you had a few developers knowing what containers are. Now you have a few developers that don't know what containers are.

Definitely, the skills have tipped and a lot of the innovation that comes from, say, build environments or local workflow of the developer I think will be geared around containers somehow whether that even the build servers that people have for syntax highlighting or language servers, I mean, even those will be probably containerized at some point. And I think it's just a matter of time until IDEs are containerized maybe like entire operating systems in a way Android is kind of like every application has its own container. But I believe at some point there will be operating systems that have very strict restrictions between applications much like containers and there will be much more mainstream. Certainly, it's super interesting. I'm very excited for what things are coming up and, yeah, very excited for the future.

[00:44:18] JM: Well, Vlad, thank you so much for coming on the show. It's been a real pleasure talking to you.

[00:44:21] VI: Thanks for having me, Jeff. Pleasure to talk to you.

[END]