EPISODE 1450

[INTRODUCTION]

[00:00:00] JM: Terminals are a gateway to building and running software, but they have not been reimagined since their initial creation. Warp is a new kind of terminal with visual aids, autocomplete, documentation, customization and other features. It includes GPU acceleration, its own UI framework, and other engineering systems to increase speed and performance. Zack Lloyd joins the show to talk through the creation of Warp and the future of developer tools.

[INTERVIEW]

[00:00:25] JM: Zack, welcome to the show.

[00:00:26] ZL: Thanks for having me, Jeff. It's nice to be here.

[00:00:28] JM: You are working on Warp, which is a new kind of terminal. Can you start by explaining how are terminals typically built?

[00:00:37] ZL: Sure. So terminals has been around for a really long time. What you're actually using when you use a terminal on your Mac these days is a terminal emulator. And so what it's doing is it's sort of copying the behavior of a piece of physical hardware that hasn't been in wide use since, I don't know, the 80s. And what that piece of hardware was, was a very simple sort of device for sending keyboard input into a shell and then receiving characters out from that shell and sort of rendering them on a screen. So when you're building a terminal, you're building like a native app. Like Warp is a native Mac app. Essentially, you're building a UI for inputting characters and then taking characters back from the shell and rendering them on the screen. That's like super high-level what a terminal is.

[00:01:29] JM: Tell me more about how the terminal UI interfaces with the actual contents of your computer.

[00:01:37] ZL: Sure. So when you start a terminal, what it's doing is it creates something called, at least in the sort of Unix Mac world, it creates something called a pseudo terminal, which is a kernel concept, where, essentially, it's simulating talking to a – It’s like, basically, you sent

characters into this thing and you read characters out. And then sitting on the other side of this pseudo terminal is a shell process that you start. And so the shell is something like Bash, or Zsh, or fish.

And when you type into the terminal, so the key events are received by your normal Mac app. And they get sent through this PTY into the shell. And the shell is the character interpreter. It sort of reads in those characters. Eventually, when you hit enter, the shell executes a command. That command runs. And if it's sort of a command that isn't like Vim or Emacs or something, but something that produces a one-time result, it sends characters back via standard out and standard error. And they go over the PTY, and they come into the terminal.

And then on the terminal side, what you have is a bunch of parsing code, which is essentially it's like a big state machine that takes the characters that come back. And some of them are sort of plain text, and some of them are control codes. And you can think of control codes kind of like being like a very primitive sort of HTML, if you want to make an analogy. And the terminal parser looks at those control code and says, “Okay, we need to update the display to match whatever has been sent back.” And so it could be like the most basic thing is. We need to draw a character on the screen at a particular position. We need to move the cursor in the screen to someplace else and draw some more characters. And so what we're doing in the terminal is like we take the results of that parse. And then we have a hope sort of graphics framework where we end up putting that text on the screen.

[00:03:25] JM: What are the flaws of conventional terminals?

[00:03:30] ZL: It's a great question. So you can view these things as flaws or strengths. But there's this very strict sort of separation between terminal and shell. And the terminal really doesn't have any knowledge of what is being run behind the scenes. So this is one of the things that we're trying to address in Warp.

And what that means is like the characters that come back, when the terminal displays them, it displays them as just one big buffer of text. And so, for instance, you don't know – You have no like delineation of when one command started and ended and what output it produced. And it's just a big buffer of text. So if you want to do things, like, for instance, scroll around your terminal command by command, you can't do that very easily. If you want to just copy paste the output

of one command, you have to sort of like find it and then highlight it particularly well. There's just no structure is one big problem with it.

Second thing that traditional terminal is not really set up to do is the way that the input works, is basically when you type into a traditional terminal, every character immediately goes to the shell and comes back. And the terminal doesn't even really know much about what's being inputted. Doesn't know sort of like – It just has more characters that are being rendered on the screen as they come back from the shell. And that has resulted, I think, in like a pretty non-standard input experience. So like if you're new to the terminal, for instance, and you try to sort of click to put your cursor someplace, there's nothing that handles that in the traditional terminal. You can't really like select, and delete, and insert. You can kind of do some of these things if you get like advanced Vim or Emacs bindings in there, but it's a very unintuitive experience. And it's a very different experience than every other app that someone is using. And a lot of this has to do with the fact that the terminal and the shell are very separate.

Now, one really big advantage of having the terminal and the shell separate actually is that you could have a terminal running on one machine and a shell running on another machine. And it's just a text-based interface for communicating with them. So the terminal shell architecture enables something like SSH, which is super, super useful, because it’s just sending characters back and forth over the network. But the actual sort of UI of the terminal is not ideal.

[00:05:52] JM: And is alleviating those issues in the terminal the right approach? I mean, there's so many different environments where code is written and code is displayed. You got GitHub and your text editor. Why is pushing more functionality into the terminal a wise developer workflow decision?

[00:06:18] ZL: Yeah, it's a great question. I guess, the way that I think about this is we're actually trying to push more work into the terminal. The terminal, sort of for better or for worse, is a tool that is super widely used. Probably open on most developers’ desktops if you walk by. And the reason it's widely used is because command line apps are so ubiquitous. And they're so ubiquitous because they're kind of the easiest apps to write. Like it's just a lot easier to write an app with a text-based interface than it is to write a GUI app. Like most of the servers that are running the Internet are basically running command line apps. Most of like the sort of DevOps work that goes on interacting with different clouds and stuff uses command line apps.

And so we're sort of taking the approach that these command line apps and text-based interfaces have like an inherent sort of usefulness to developers. And that what we're trying to do is make the interface around using those command line apps as sort of humane, and intuitive, and accessible as possible.

Code editing is interesting. Like, I use VS Code. I don't use Vim and Emacs. And I don't have like a sort of desire with Warp to push people out of ideas into the terminal. It seems like I actually like the functionality that you get in VS Code.

And sort of what we're doing, to some extent, is taking some of those more modern UI ideas and making them available in the terminal where it makes sense so that people can actually be – When they have to use command line apps, or where they want to, they can be more productive using it.

[00:07:57] JM: To tell me the mental process you went through to decide to build a brand-new terminal and why you took the approach to building from scratch as opposed to augmenting some existing open source terminal.

[00:08:14] ZL: Great question. A big part of the reason that we did Warp is just sort of personal. Like my background is I've been an engineer for the last 20 years. And I've actually never been a terminal power user. I've always thought that the terminal was like a useful tool with a very weird interface that could often be frustrating. But I also always worked with engineers who are really good at the terminal, really good at using the command line. And they could do things that were almost like magical. When we'd be in the middle of a firefight and they'd be able to like look through the logs in a way that I could never do. Or like they just had all these productivity shortcuts, which, when I saw them working, I was like, “Wow! They're actually being much more productive in this environment than I am.”

And a big part of the goal with Warp and the inception of it was can you build something that unlocks that power for everyone? If you assume everyone's going to be doing something with command line apps, how do you make the experience that the power user has, which it often really configured their terminal. They've learned a bunch of keyboard shortcuts. They've read

the man pages. How do you make that experience more readily available to the rest of developers? So that was a big part of the idea.

And then the other part of the idea was like, when I was looking at this app, I was like, “It's seems very archaic from the perspective of like it doesn't work the way that most of your other developer apps work these days, which is to say it doesn't like leverage the fact that the Internet exists.” It doesn't really have any primitives to help teams work better together. It doesn't really have any sort of facility for sharing, or collaboration, or an easy – Like even the whole view of the terminal as like platform seems a little hacky to me. It's like you basically go to a GitHub project and then curl something through bash and cross your fingers and hope that the conflict doesn't conflict with what you have already. And so I just saw it as like an interesting opportunity to take one of these apps that developers really spent a lot of time in and modernize it. So that was the problem space.

And then as far as like the approach of building a new one, rather than sort of trying to have an open source project or something that ups the power of the terminal within the current framework, there were a few things that we wanted to do that we're sort of, I think, too foundational as far as changing out of the terminal works in order to do them without building the terminal.

And so, for Warp, like those things are – We've really fundamentally changed the way that terminal input and output work. And we couldn't do that without actually writing a terminal. So that was the sort of the biggest thing. And like, Warp, what that means is the input works via like a normal text editor. We take over all the input from the shell and we've built something that has a code editing experience, kind of like what you would get in VS Code. And then the output is aware of when a command is run and what it produces. And so we have this block construct. And these are things that you can kind of bolt on to an existing framework. You really need to build it from scratch.

[00:11:23] JM: And as you were deciding to build it from scratch, probably the most critical decision you had to make was what language to write it in. And you chose Rust. And to me, that's intuitive, because Rust is the most modern systems level programming language, I'd say. I don't really know why. Whenever I ask people about Go versus Rust, I guess the best answer I usually get is – And I don't know how accurate this is. But my recollection is that Rust is often

really good for memory management, low-level memory management. And then Go is really good for networking primitives. And I can imagine memory management, or I guess low-level operations being more important than networking in a terminal.

But yeah, I guess you could just write it in Node for all I know. But then you'd have latency issues, probably, because that's slower than Rust. So I don't know. I'd love to know about the programming language choice.

[00:12:24] ZL: Sure. I mean, what you said is pretty accurate. So we actually prototyped it first in Electron and using web tech. We were all sort of from web backgrounds. And that was most comfortable. The performance became an issue very, very quickly. Like what a terminal is doing is basically like rendering a lot of text potentially very fast. And so the web-based approach – I don’t know. We pretty quickly became convinced that from a performance perspective, it wasn't going to work that well. And so we looked around for an alternative. And Rust was sort of, I think, like you said, the natural choice. We wanted something that would give us maximum performance and maximum control. And we'd be relatively cross-platform, and wouldn't be like a huge sort of burden in terms of developer ergonomics and productivity.

And what we found with Rust is like even though it is low-level, and it's got a learning curve when it comes to the memory manager model. Like, in Rust, you have to think about who owns what memory. Who's borrowing a reference to it? How long does that memory live for? You get the hang of it relatively quickly. And then the cool thing is that your app is pretty fast by default. Whereas if you're doing it with a web tech, it's not really fast by default. And you can hit walls around what you can and can't do.

And, for instance, for me, so I used to be the engineering lead on Google Sheets and run that team. And I was an engineer. I used to lead the – The tech lead for Google Docs. And when we built Google Sheets, we hit very fundamental performance bottlenecks with the browser and the sort of memory model around JavaScript where we’re like we couldn't really control the memory layout of a spreadsheet cell, which was very frustrating to hit that kind of limit. Whereas with Rust – And we're even doing not just Rust. We're doing the full rendering sex. We're going all the way up through the shader code. We have complete control over the performance. And it's let us build an app that is very, very fast by default. So that was the choice.

[00:14:30] JM: So how have you found the – It has been mostly like a refactoring process of when going from that Electron prototype to the Rust implementation. What's that process of the reimplementation been like?

[00:14:47] ZL: So the Electron one was a demo. And we just basically throw it out. When we did the Rust one, we started off using as a basis for our code, this project called Alacritty, which is another Rust-based terminal. It's a very totally different product philosophy from Warp, but very good codebase. And so each one of our sort of blocks in Warp is almost like a mini terminal rendering.

And so we started with that. And then the rest of it we sort of developed around that. And it took a fair amount of work. Like we were working on – From the time we started the company to when we put out a sort of private beta with external users, it took us about a year. So it was a fair amount of work.

[00:15:33] JM: What has surprised you about the engineering problems inherent in building a terminal?

[00:15:41] ZL: It's interesting. The trickiest part for us is how to interact with the shells. So to interact with like Bash and Zsh. And specifically, the trickiest part is how do you innovate on a terminal and make a better product experience while still being 90 plus percent backwards compatible? We try to be as backwards compatible as we can, which means like we want, when someone uses Warp, them to be able to sort of download it, open it and work with someone's existing configuration, and work with their existing shell.

And what we're trying to do from a technical standpoint is something that's not really natively supported by shells. And so we've had to sort of configure them, and hack around a bit, and then in order to get the right metadata coming back for them so that we can implement the features that we think make the terminal better.

So, for instance, in like Bash and Zsh, we need to hook into when are those shells running commands? When are they done running commands? We need to understand, like, are they echoing characters back? Or are they actually in the process of running a command? And all of

that like Shell integration stuff is sort of the hairiest and diciest like technical piece that we've had to deal with so far.

[00:17:02] JM: And what is hairy about that? I mean, is like race conditions, or somehow lowering the memory footprint, or latency issues? What is difficult?

[00:17:13] ZL: So what's difficult is that the shell doesn't know anything about the terminal. It's like expecting it's running in like a normal terminal environment. But we're running an environment where we've taken over the input to give a really concrete example. And so in a normal terminal environment, as you type the character, that character gets sent into the shell, and the shell echoes it back.

Now, we have to sort of not do anything as it's echoing that character back. And we have to understand, is it in a mode where it's echoing that character to us? Or is the command actually producing output that's running? If it's echoing it, we need to like just basically ignore it. If it's running, it needs to be printed on the screen.

And so I think one particular technical thing that's caused us a lot of headache is like people who run terminal commands, you'll often like run some commands. So you'll run like, I don't know, git-diff, or git-log. Or maybe those aren't good examples. But git status would be a good example. And then before that git status finishes running, there are terminal users who are so fast on the command line that they'll start running their next command. And we have to handle the case where the shell is like basically sending those characters back. Because what the shell does when you run a command like that is it like it runs the current command, it buffers the characters that come in, and then it sends them back. And we have to know that, "Hey, this isn't part of the output of the current command. It's actually meant to be the input for the next command." And sells just like aren't set up at all for us to do that. And so it's like we start using weird heuristics about like, "Well, what state is the shell in while it's running this command?" in order to figure this stuff out?

[00:18:48] JM: Interesting. And so do you have good introspection tools for being able to assess like what's slowing you down or bottlenecks? Or like, what's the process of doing diagnostics?

[00:19:02] ZL: It's a good question. So we've built into Warp a sort of debug mode that it's only something that's currently enabled in our developer builds. But what it does is it sort of gives you – It prints out everything that the shell is coming back in. And so we've had to build a little bit of sort of self-diagnostics around this.

And then the other diagnostic tools that we sort of regularly use for at least like for performance issues – And since it's the native Mac app, we're using the tools that come with Xcode. So like Instruments is a super useful tool for us if we want to do profiling. Since we're doing all of like the rendering ourselves using Metal and writing directly to the GPU, Instruments has a really good sort of GPU snapshot and debugging tool that we use to debug rendering issues.

And then on the sort of testing side, we've also built a bunch of, I think, pretty cool tools where you can run Warp in a sort of headless mode and write integration tests against it, so that if you want to simulate, “Hey, a user is typing XYZ, and that goes into the shell, we want it then like assert that the terminal is in a particular state after executing it. Kind of like Selenium for web tests or something like that. We have the ability now to do that, and to do it against sort of different shell configurations, right? That's another thing that is complicated. It’s like there's all different versions of these shells. Kind of like in the browser world, it's like there's Edge, and Firefox, and Chrome, and different versions. And so we've had to deal with something similar around accounting for all these different configurations as we sort of debug and test.

[00:20:44] JM: Tell me more about that debugging and testing process.

[00:20:49] ZL: Sure. So some of it is just the normal Rust unit testing. So Rust has got a great facility where if you run cargo test, it will find all of your tests. And so our code is like pretty significantly unit-tested. If we want to write an integration tests, what we do is – Actually, we just added this very cool thing, where you can sort of grab a snapshot state of Warp. So basically, if you want your integration tests to start from a particular terminal state, so you want – Like these five commands had been run. And then what you're going to do is you're going to test it. Like right clicking on a block brings up the appropriate context menu. And choosing an item puts the right thing on the clipboard. So you can basically bootstrap the integration test to run from a particular state.

And then you write a test which sort of says, “Hey, simulate a mouse click here. Simulate a key event here.” And then you allow like a framework to pass. And then you can assert that the model and the viewer in the correct state. And you can do this locally. And actually, when you do it locally, you can actually sort of pull up Warp and see it running as though a user was doing the thing. And then when we commit it, what we do is we have GitHub actions that run these integration tests sort of through GitHub’s CI platform on native Mac's in the cloud. And we are able to actually do that against different like Bash configurations and Zsh configurations. So I don't know if anything like it exists or not. But it's been super useful for us for preventing UI regressions.

[00:22:25] JM: Have you had situations where the UI lies to you and you can't get the UI synced to what is actually going on?

[00:22:32] ZL: That's interesting. I mean, we've definitely had flakiness in these tests, as you are apt to get with any kind of tests that tests a full application and a full system. The way that something like what you're saying would happen is if our sort of view model, like the model was out of sync with the UI, which is totally possible.

Like one of the things that we do in order to improve performance is that all of the parsing of the text that comes back from the shell happens on a separate thread from the main thread. And so it's totally possible that we would have like a concurrency bug or something like that, where when we go to render the view, we aren't rendering like the correct state, or the state is somehow changed in between rendering a frame. So we've had weird bugs like that. But it hasn't been that common.

[00:23:25] JM: Let’s about some of the implementation of features. So collaboration is one feature you're trying to implement, collaborative terminal. That seems like a pretty good step change from traditional terminals. What are the difficulties of implementing collaboration?

[00:23:44] ZL: Yeah. So we haven't done real time collaboration yet. The only collaboration feature we have right now is for any command you run, you can basically say, “I want a shareable snippet,” which gives you a permalink to that command that you can then like Slack to people or share posts wherever and they'll be able to see what command you ran or what its output was.

For the real-time sharing, the way that we want to do it – I mean, we can talk about the challenges because we've looked at it quite a bit. We want that, basically, for any terminal session that you do in Warp to be able to get a link to it that someone can open in sort of like Google Docs or Figma. Be able to see a live view of what's going on and maybe even participate in the session.

In order to make that work, like one thing we need to do, which is very technically challenging, is to do a web rendering of Warp. And we want to do it using the same Rust code base, which means, essentially, the Rust code needs to compile to WebAssembly. And the platform-specific pieces, so like the event inputs and the rendering primitives, we need to port those from being Mac-specific to being web-specific. So for instance, like we to switch from using Metal to using WebGL. And so the web piece of it is one big piece of it.

A second piece of it that will be challenging is how do you sort of do the server coordination of it? And specifically, if you're running like a local version of Warp, how do you sort of poke a hole into that in a way that's like safe and secure, potentially using something like an SSH tunnel, or like what ngrok does. I don't know if you're familiar with that tool. Where you would let someone else sort of see what's going on, on your machine. That's going to be challenging.

And then if you really want to get into like the collaboration primitives, sort of having like either operational transforms – Or our code editor is actually a CRDT. So it's like if we wanted to get into real-multi cursor collaboration, that's even another big project. I don't think that that's necessarily the right way for a terminal to work. But it's going to be a pretty significant technical lift when we do decide to do it.

[00:26:01] JM: What about integration with Git? Are there opportunities to build better tools for Git workflows?

[00:26:10] ZL: I think for sure. Like when we talk to Warp users and just terminal users in general, like Git is always one of the most frequently used tools. One thing we have right now is we ship Warp out of the box with like a pretty robust set of Git completion. So you don't have to configure that with Warp. And the UI that we have shows documentation around what you're

doing with Git as well. So that's one way that we've sort of helped people. But I think it's actually pretty minimal.

A new feature that we're building right now, I think, will be even more useful. It's a feature called workflows. And the idea is that, instead of assembling a command word-by-word and using completions to build something up, what you'll be able to do is sort of save and share an entire command. And like the example that I always use with Git is like I always forget like how do I undo my last commit? And it's like – I forget. It's like Git reset--soft, hard, tilde, head. Or something like that. Whereas we want to create a sort of searchable, shareable library of little workflows that help you find the whole command rather than doing it on a word-by-word basis.

And then part of that can expand in even doing something that's like almost like little like internal UIs or internal like workflow apps to make some of these more common things that you might do in Git. Like how do I check out a file from another branch? Or how do I do some complicated rebase? All those things, having better documentation, and being easier to like find and execute in the terminal. So I definitely think there's a lot of scope for improvement.

[00:27:55] JM: I'd love to know more about what you said about the GPU interface versus the Metal interface. I don't know much about either of those. Can you just tell me about what those interfaces are? And like what coding to them looks like?

[00:28:13] ZL: Sure. So Metal is the is max, like, kind of Shader Language. So it's like the interface into if you want the GPU to do your rendering. And it's similar to OpenGL or WebGL, if you're familiar with those. But what you're essentially doing is you're writing code in something called a Shader Language. And what that Shader Language does is it sort of lets you – There's usually a couple of different phases in it. So one phase is like figuring out what shapes you want to draw. And typically, you're passing information into these shaders in the form of like sets of vertices.

So imagine you're trying to draw a rectangle on the screen, you would pass in, but simplify in say four vertices. And the shader programs have basically code that let you sort of interpolate between those vertices what the value should be at every pixel. And so you pass in these vertices. You say, “I want to draw a rectangle at a specific spot.” And then you can write

something called a fragment shader, which gets called essentially on every pixel in like a massively parallel way. And what its job is, is to output a color at that pixel.

And so the point of all this – And this is like how like video games work. And like, basically, at the end of the day, pretty much everything on your screen right now is being rendered through the GPU. The point of it is it gives you massively parallel computation of what like the pixel should be at every position based on like passing in a few set primitives.

And so what we do with Warp and what those primitives are, it's kind of interesting. It's like if you're writing a terminal program, you actually don't need that many. You need the ability to draw a rectangle. And that rectangle needs to have things like borders and background colors and perhaps like rounded corners. So rectangles are one primitive that we've had to code into this. You need the ability to draw images. And so GPUs are actually really good at drawing images. You need the ability to draw a text. Those are probably the three biggest things.

So what happens in Warp is like we have all this sort of UI code that prepares a scene. And you can think of a scene as a collection of these primitives. And then, at that point, we hook into Apple-specific interfaces for passing data into these shaders. And then you basically say, “Okay, shaders, take this data and render a frame.” And what comes back from that is all the pixel data, which is what shows up on your screen. And we're doing this constantly at like 60 fps, or even like higher FPS than that, as someone scrolls around in the terminal, or as new text comes back.

[00:31:07] JM: Are there any engineering challenges you've encountered in making those scenes amenable to the human view? I mean, it gets more UI or display challenges? Like, how to actually present – I mean, when I think about like a terminal, it's very primitive in how it typically displays things to people. It's very utilitarian. How have you like improved the UI experience for building a terminal?

[00:31:33] ZL: Yeah. So one of the things that's really different about Warp is that we're able to do sort of arbitrary graphical sort of things within the terminal. We aren't just limited to the character grid. And so, for instance, if you play with warp and you hit tab, you get a completion menu, which is sort of similar to what you would get in VS Code. Or if you do Ctrl+R, you get a visual search through your history. Or if you do Command+P, you get a command palette. So

we have all these different UI elements, which are meant to make using the terminal more intuitive. And Warp and in another terminals, to support this, we've had to support sort of a richer set of UI primitives than just rendering characters.

Some of the things that are challenging if you're trying to do them on the GPU, like doing gradients is an example of something that takes like a bit of math. Doing like rounded corners. Doing drop shadows actually, is relatively sort of challenging to do in an efficient manner using just like the math primitives of the GPU. But it's all sort of doable. Text has been a pretty interesting challenge.

So the way that text works in Warp is we essentially ask the Mac – Some Mac ships with some platform frameworks around text layout called Core Text. And if we want to render a string of text or a paragraph of text, we ask Core Text essentially for like what's the layout. And the layout algorithms for text are pretty complicated. And like there's a lot of different languages, and there's different kerning for every fonts and different – If you want ligatures, you need something special.

And so you don't want to reimplement that in Warp. And we haven't really learned that. But what Mac tells us is like, “Hey, here's how the characters should be spaced relative to each other.” And then here's all the pixel data for the individual glyphs. And then we sort of take that all, assemble it, pass it into the GPU, pull out that pixel data and render the text on the screen. And so all of this is like challenging. But at the end of the day, the advantage is that we have sort of ultimate performance control over what we're doing. And actually, it's not that much code. Like we only have like 300 lines or so of code that's written in the Shader Language. So it's not as bad as it sounds.

[00:33:54] JM: I'd love to know the business direction for Warp. I think it's really hard to build a developer tools business that gets gigantic. There's a lot of like niche developer tools you can build. There'll be moderate-sized businesses. I'd love to know your vision for how to turn this into like a really big business.

[00:34:17] ZL: Sure. It's a very, very good question. So one thing I would say is like we specifically targeted a very horizontal tool. So we did this not just for business reasons, but just in terms of – I think the potential impact is greatest when you're working on a tool that's used by

most developers. And so we're not building a tool around a particular technology like Kubernetes, or React, or something. It's like we're building a horizontal productivity tool that all developers have a chance to use. So in terms of like the total addressable developer space, it's not as niche as similar developer tools. That's one starting point.

I think what's really challenging though is that terminal has always been free. And like competing with free and trying to sell it I think is it's hard. And so we're not trying to do that. Like, basically, we're trying to give work away for free for individual developers. And our bet is that if we can get enough traction, that there's a set of features that belong in the terminal or around the terminal that are appealing to teams and to enterprises where you can get a company to pay for some of these things.

And for instance, like real-time collaboration. It's something that doesn't exist today. I think we would have some version of it, that's for free. But perhaps there are things that would be more useful in a team context that you could charge for. Or security in the terminal. So like making sure that people aren't sort of randomly copying keys or copying PII that shouldn't be in there I think is an interesting enterprise angle. Or auditability.

And so let's say you have a big outage at your company and you want to understand like what are all the things that developers did in the middle of that outage in the terminal. Is there a way that you can find that out? Or firefighting. Can you build something that's better for DevOps and SRE teams? Where when you need to be like jointly SSH into a machine to really debug something, can the terminal help you with that? Onboarding. Can you make it really easy for when someone joins a team that they have all of the terminal configuration and all of the commonly used commands available? These are all the ideas that we have around things that we could monetize. We haven't proven that you can do it yet. And our focus for now is very squarely on like can we just make individual developers more productive? And like I think that's really, at end of the day, like that absolute key thing. If we can do that, I think a lot of people will use Warp and will have a lot of opportunities to build a business. If we can't do that, I think, we won't.

And then one thing we are explicitly not doing to be clear is like building any kind of business around like sucking up people's data from the terminal and trying to monetize that. We want to

build something that people want to pay for, because it makes them more productive in a terminal.

[00:37:12] JM: How have you seen adoption of Warp versus other like newer terminal paradigms? Like there are a few others like Fig. And then there's web-based IDEs like Replit. Do you see these is like, complementary or mutually exclusive? And how do you see the preferences for different adoptions?

00:37:34 [] ZL: It's really interesting. So I think Fig is really cool. We actually use some of Fig's like completion repo, their open source repo, in building parts of Warp. I think it's solving a similar problem to us. It's just a very different product approach. So their approach is more like use your existing terminal. And then you have a UI component that sits on top of it. And there's definitely advantages to that approach as far as like go to market because you don't – If you're a terminal user, you don't need to change your terminal. You can install something on top of it that adds functionality. Our product bet is that, ultimately, building the whole terminal gives us the ability to sort of build the platform and not just build the thing that sits on top of it. And we think we can build a better product experience by doing that.

So to some extent, it's competitive. To some extent, it's like orthogonal. And then something like Replit, I think, is super cool. But also like somewhat orthogonal, in the sense that it feels like it's really heavily focused on the code editing experience, as opposed to the use case around like, "Hey, I'm using a bunch of command line apps, which is what the terminal is about. I do think the future is going to be some like – It's just going to increase. And we'd be like you're doing stuff in the cloud. And you're not doing stuff locally. But we decided, with Warp, that one of our product principles is to meet developers where they are. And still, today, I don't know the exact number, but it's got to be 95% plus of development is being done against people's local file systems. And so sort of we thought we would not have as good of a shot as helping developers today if we did something that was purely cloud-based to start.

[00:39:17] JM: Do you have any reason to add in other programming languages? Like, as we talked about a little bit, like Go is quite good for networking primitives. Are there any reasons why you would need more heterogeneity for programming languages? Or would rust just satisfy everything in the language?

[00:39:35] ZL: It's a good question. So our server is built in go right now. It's also a very minimal server. Like our server doesn't do much. We're primarily a client-side app. But we are using go on the server. And the reason that we made that choice – And I don't know if that choice will stick or not to be totally honest. But the reason we made that choice was because the sort of the cloud providers can to have better SDK support for Go right now. So it's like we're on like GCP. We're on Google's cloud. And so there's like native GCP, like API kits for go. Whereas for Rust, at least when we were looking before, there weren't.

And then it felt like a library maturity thing on the GO side. Go was like a little further along. The Rust community is great. Like the Rust community is super enthusiastic. And I think server-side Rust is something that is viable today and improving rapidly. And like, personally, I would rather not have multiple languages, unless there's like a really strong reason. It's like, every time I context switch between our client code and the server code, I have to be like, “What's the for loop look like in Go again?” And so I would rather sort of have a single language.

And also, for us, there's potential for sharing code across client server if we're going to do like a cloud-based terminal ever. And so I wouldn't be surprised if we revisited that decision. But we did start with that. So we have a Go server right now.

[00:41:08] JM: So we talked a little bit about security. What are the security issues in building a terminal? And how do you shore those up?

[00:41:16] ZL: Yeah. So I think as long as you're not talking much to the server, which right now, Warp, it doesn't talk to the server, except for like analytics and like crash reports, that kind of thing. But we're not sending any data to our servers. So we don't really have a concern on the server-side at the moment.

Our biggest concern, to be honest, security-wise, right now is probably just the fact that we're shipping desktop software. And that if you pull in a dependency that has some sort of malware or something like that, that's like the risk. As we get more advanced and start to build features that are more cloud-enabled, I think that we have to be super careful around how we treat user data. A lot of sensitive data flows through the terminal. And we have to be very explicit about what data is going to go off of someone's local machine, whether it's to our servers or to our collaborators. We have to be – As we build more of a platform, and there are times when we

execute third-party code. Like, for instance, if we're building an extension framework, or if we're shipping with someone else's completions, that type of thing, there's risk in that, too. So sandboxing anything that's not written by us is going to be a big security concern.

I think as we do these things, it's like we take it super seriously. And they're going to be super – We're going to audit it. We also plan on making the code publicly visible pretty soon. And I think having visibility into the code outside of our core team will also be sort of helpful on the security and privacy side.

[00:42:51] JM: Let's wrap up just by zooming out and getting your perspective on developer tools more broadly. How has your usage of modern developer tools changed over the last few years? What are you using these days? And what anticipations do you have about the near future?

[00:43:07] ZL: Yeah. So, like, my super broad perspective on sort of the developer world is something like I sort of think of it in terms of product and infrastructure. On the infrastructure side, my personal take is like it's always changing. And like there is a gigantic, sort of like long tail of ways that people do databases or languages that people use. And they're always changing. It's like, at one point, it was React, then it was Vue. Then, at some point, it was Angular. It's like, the technologies, I think, are rapidly changing. And same thing with like Docker, Kubernetes. I don't know what's coming next. But I am very confident that the infrastructure and language world will always be somewhat fragmented and always changing and not converging. That's my personal opinion. People may disagree. But I just feel like, empirically, that's been the case.

On the product side, I feel like it's a little bit less fragmented. And it seems to me that VS Code, on the coding side, has become – It's not like dominant-dominant, but it is like the de facto tool that people are using. It's what I use. And they've managed to build something that I think is like pretty good. But the really big benefit in it is that there's a sort of network effect around the extensions that are available for it. So it's like it has good support for Rust. And if I switch to Go, it has good support for Go. And if I need a JSON viewer, it has good support for that. So the community and ecosystem around it I feel like is super valuable.

That's like the main – Besides like the two tools that I use that I have open on my computer all day long at this point are Warp in VS Code. And then every once in a while, I'll use something like Postman for doing API debugging. There's a tool, Archetype, which is for like SQL stuff, which I think is kind of cool. But yeah, the two main tools that I live in are the code editor in the terminal. And for me, is VS Code and Warp.

[00:45:10] JM: Cool. Well, any other projections for the near future of Dev? What's your – Or Warp.dev. What you're working on?

[00:45:17] ZL: The current state is that we're in a private beta. We're going to go to a public beta sometime in the next few months where we get rid of our waitlist and we get rid of our invite mechanism. And the main things that we're waiting for there are just sort of shoring up some of the missing like basic functionality. Like you can't customize your key bindings all in Warp, which is kind of crazy given it's just a keyboard-centric app.

And then like our sort of roadmap and goal is to continue to build features that are useful for individuals around the daily workflows that people do in the terminal. So it's like how do you make command entry really, really streamlined? How do you make working with the outputs of your terminal commands much better? And then as we build some of those single user, like core UI improvements, we're going to start to focus a bit more on features that are around like how can you make the terminal work better in a team context? How can you make it work better by leveraging the cloud and the Internet and just like collaboration and those types of things? But in the short term, it's all about just like, "Hey, we want somebody to download Warp and be like, "This is a really awesome command line experience. And I'm more productive immediately by using it."

[00:46:24] JM: Cool. Well, Zach, thank you so much for coming on the show. It's been a real pleasure talking to you.

[00:46:28] ZL: Yeah, Jeff. It's been awesome. Thank you so much for having me.

[END]