EPISODE 1461

[INTRODUCTION]

[00:00:00] JM: The terminal is a necessary tool for any software engineer. In order to work quickly, developers have always customized their terminals to work for their specific application workflow. Fig is a reimagined terminal product that adds autocomplete and an entire app ecosystem to the existing terminal that you're familiar with. In a previous show, we talked through the basics of Fig. In today's show, we take a deeper dive into the functionality and engineering behind the product with the founders of Fig, Brendan and Matt.

[INTERVIEW]

[00:00:27] JM: Guys, welcome to the show.

[00:00:30] BF: Good morning. Thanks for having us on.

[00:00:31] MS: Excited to be here.

[00:00:33] JM: We previously did a show about Fig. And I think it's worth starting off with just an explanation for Fig, which is a way to extend your terminal with added functionality. Give a brief description for some of the functionality that Fig adds on top of your terminal.

[00:00:59] MS: Yeah. Our big hypothesis around the terminal is that it's this tool that people use every day, but the interface has remained the same for the last 50 years. And so we started by thinking, “What's the kind of baseline thing that we can do to improve terminal workflows every day?” And we started off with autocomplete. So when you download Fig, currently, you can get set up with autocomplete, and it'll give you IntelliSense style suggestions the same way you would get code suggestions in VS Code, or another IDE.

[00:01:29] BF: So that was the first product we built. Really, that's what we tell users. And really under the hood, we've built an API on top of the terminal. We’re thinking, “Let's say we were building the terminal from scratch today, what would it have?” And we were like, “Well, it'd be extensible.” Literally, every editor and every other tool out there is incredibly extensible. Has an

API that you can build on top of. And you can build on top of the terminal on the shell. But it's really, really hard.

And so what we've built with Fig is this API that allows us to build – Essentially, allows any developer to build a web app that can read and write to the local shell, that can position itself, or in and around your terminal emulator. So it feels like you have the powers of a VS Code extension store. But it's attached to your existing terminal.

So the first product we built was autocomplete. And that was just like a proof of concept that turned out to actually be incredibly value to user workflows in the terminal. But the next thing is we're going to build around sharing environments, sharing workflows across teams. Basically, we think of Fig as the sort of source of truth for your developer environment and your developer workflows in the terminal. And all of the products we're going to build, and we hope the community builds, are around like making you faster, or like helping you sort of replicate your environment onto a new machine.

[00:02:49] JM: The first feature of autocomplete – So if I have an autocomplete for my shell, let's talk a little bit about the engineering behind that. So I imagine it's actually a pretty complex syntax data structure, that you would need to build to have autocomplete work. Can you describe that data structure?

[00:03:13] MS: Yeah. I mean, so there's two sides of it. One is all of the infrastructure that gets information out of the shell and passes it through this API and exposes it to the autocomplete engine. And then the other side of it is the actual completion specifications that the autocomplete engine uses to generate suggestions.

So I'll start off with the completion specs, as we call them. These are open source TypeScript files that define the structure of a CLI tool. So you go in and you say, “Here are all the sub-commands. Here are all the options. Here are all the arguments.” And you can declaratively say, “This is the structure of a CLI tool.”

But what's really cool about Fig is we let you go even further and write TypeScript generators. So functions that can run shell commands, or make web requests that can expand the list of suggestions and customize it based on the user's local device.

[00:04:03] BF: So to like hone in on some specific examples, let's take Git or NPM. So in our TypeScript schemas, what Matt was saying, called completion specs, we just think of it as a tree structure. It's a super simple tree, actually. And every CLI tool either has sub-commands, has options, and has arguments. That's basically it. And so if you look at Git, what are the sub commands there? Check out, and commit, and push, and pull? All right? And we just have a tree, Git is at the highest level, and nested underneath that there are sub commands. And then underneath sub-commands and some options as well, you have arguments.

And so an example for checkout is I type Git checkout and then a branch name, or commit hash or something like that. And what Matt was just saying is, for certain arguments, we can actually provide not just check out and the sort of the things that we call part of the skeleton, the stuff that you already know is going to be there. For arguments, we can actually run a command on the device, get the output and show a list of suggestions.

So for Git checkout, when the user types Git checkout space, we're running at the current working directory the user is in. We run Git branch. We list out all of the branches that are available to the user. We pass them, and then we render them as suggestions for the user. So although it sounds like, “Oh, this is some crazy data structure,” we've deliberately spent a ton of time making it ridiculously easy, because we want to build these completion specs for every CLI tool out there. In fact, we didn't even want to build it. We want to automate the construction of them and then make it easy to enrich. So yeah, simple tree structure, which maps out a CLI tool and allows you to enrich the suggestions for arguments as well.

[00:05:43] JM: So with the example of Git, if you wanted to automate the creation of autocomplete functionality for Git, how would you automate that system?

[00:05:58] MS: So Git is a bit of a tricky case, because as far as I understand it, the Git CLI tool is written kind of in in C with no CLI framework or anything like that. But where automation can come in, is if you're using like a CLI framework like Cobra, or oclif, or Commander.js, we can integrate at the framework level and use the information there to automatically produce completion spec.

[00:06:22] JM: Got it. So can you give maybe another example or two about how you might be able to auto generate the autocomplete functionality over-time?

[00:06:33] BF: Sure. So, Figs, CLI tool is built in a framework called clap in Rust. And basically, a framework for building a CLI tool is you shouldn't have to deal with all of the parsing infrastructure yourself. A framework is coming in and it's able to – You essentially say, "Here's a sub-command. Here's an option. Here's an argument." The framework does all the parsing for you.

So we integrate with cloud. Fig has built in integration with cloud. And every time we compile our CLI tool, we then have a command that is just part of the clap integration. It's called Fig Gen Fix Spec. I know this is getting a little bit meta. So maybe we can use like Docker as another example. Docker, we add a command. They use Cobra, which is a Go lang framework for building CLI tools. We add a command, Docker Gen Fig Spec, and it just spits out the full Fig completion spec for it. Like we do all of the passing for you. That's it. And you're ready to go. Like suddenly you have completions for the entire CLI tool.

What it doesn't give you is some of the sort of argument completions I was talking about. So it can't tell you, "We search over the Docker – Like, registry for Docker images." It can't do that. Because like someone has to be able to say, "Alright, this specific argument is a Docker image, or whatever, or Docker container." Someone has to manually add that. But at least we get 80% to 90% of the way there.

[00:07:56] JM: Tell me about the runtime of Fig in the case of autocomplete. So as I'm typing, what is happening behind the scenes and what's being generated in-memory? And I guess, yeah, what's being queried on disk?

[00:08:10] MS: Yeah. There's a lot of moving parts to make the seamless autocomplete experience that we hope users get. It starts off in the shell. We have different integrations for zsh, fish and bash, that will send the working directory the text that the user has typed in their shell, the process IDs, information like that, and that gets passed to a MacOS app that's responsible for hosting a web view that is the broker for this JavaScript API that we've built autocomplete on top of. So all of this information is getting passed to the autocomplete engine.

And then on every keystroke, the autocomplete engine is responsible for providing a new set of suggestions.

[00:08:52] BF: One thing I'll add quickly is everyone thinks of the terminal as something that's fast. And so Fig needs to be fast as well. So as Matt said, like, we need to know what you've typed, what your current working directory is, what the process ID is, where the cursor is. We need to get all of this information. We need to do that super quickly and efficiently. And then we need to send it to our autocomplete app and then generate the suggestions. So we've spent a ton of time making sure this isn't a lag on your terminal experience. This isn't a lag on your computer. Like we're not consuming a ton of memory. We like really, really want to make sure this is fast and just not in your way, like an Electron app is. We're deliberately built natively to avoid performance issues.

[00:09:36] JM: And remind me what this autocomplete functionality is written in?

[00:09:42] MS: We have kind of two components. There is a native component that is written in Rust, a mix of Rust and Swift. And then there is the user interface kind of application layer that's written in TypeScript.

[00:09:55] JM: Got it. So I assume that the choice of rust is just for high-performance.

[00:10:02] MS: Exactly. Yeah. I mean, we're doing kind of low-level systems type of stuff. And this is an area where Rust really excels. Because, originally, we'd written this systems integration in C. And it worked. But it was a pain to maintain. There were weird issues that the memory safety of Rest really helped resolve. So we're happy that we did that rewrite.

[00:10:22] JM: If we broaden the conversation to talk about the API – So you have an API that allows people to build their own applications on top of Fig. Using the autocomplete as an example, can you talk about how the API works?

[00:10:42] MS: So our goal is to abstract away a lot of the complexity of the terminal and the different shells that a user might be using and expose a very simple clean interface that people can build on top of. And the way it works is pretty much the way that any NPM or TypeScript package would be installed. So you just Yarn install kind of our Fig, the autocomplete – Or

sorry. Our Fig JS API. And then there's different functionalities that are exposed, like the ability to run a local shell command, the ability to receive events from the shell. You can position the window next to the cursor or as part of the sidebar. And you can also insert text into the active terminal session.

[00:11:22] BF: It sounds so strange saying this over like a podcast. If you go to fig.io, you can – If you've never seen Fig before, it's just not something you can describe. It's literally a floating window on top of your existing terminal. Like we haven't built a new terminal here. We integrate with iTerm and your native MacOS terminal, with VS Code, with Kitty, with Alacritty. We integrate with all of your terminals. But we do it in a way that's at the operating system level, as opposed to like being our own window.

And so what our API is doing is effectively abstracting away all of this operating system integration that we've done, and terminal emulator integration, and shell integration, and CLI integration, and just exposing it as this very nice JavaScript API essentially, which as Matt said, can let you run a shell, command can position the window, can insert text, and can like receive these hooks or events from the shell as users are making changes.

But if you literally just go to fig.io and you'll see the demo on like the front of our website, even the first blog post we've written, the like launching Fig blog posts, shows you all of the capabilities of the API. It's like super weird to explain over a podcast. It will make a lot more sense when you like look at this demo video. And it's like really cool. Honestly, we're obviously a little biased, but we think it’s pretty cool.

[00:12:41] JM: Let's say I wanted to build my own Fig API app and I wanted it to publish collections of my terminal commands, like sequences of my terminal commands to – I don't know. Maybe take sequence – But that’s a stupid example. But take sequences of my terminal commands and, at random intervals, push them to another terminal and execute them in that terminal. Can you talk about how I would build that?

[00:13:19] MS: So right now, just to clarify, this Fig JS API is not public yet. We're focusing on building some first party applications that use it so we can stabilize it and make sure it works for a bunch of different use cases like this.

But what it sounds like kind of long term, like the app you've sort of defined is like a way of saving a list of commands that you can execute anywhere. So it's sort of like a workflow. And this is something we actually experimented with ages ago when we were in YC. We had this idea of Runbooks that lets – Say you have a deployment script or a readme to set up the development environment. These types of collections, like we want to be able to host them through Fig so you can share them across your team or, again, build your own app that uses them as a standalone thing.

[00:14:08] JM: Okay. Got it. Well, maybe we can move on to talking about more recent functionality, specifically the stuff we were talking about before the show. So you have some new ideas around how you're going to change Fig into more of a platform for basically terminal-based development. Can you explain where you're going with the project and the work around configuration files?

[00:14:42] BF: Sure. So let's take a step back. Let's come back to super high-level. Like what is Fig? What are we trying to do here? And essentially, Fig makes you more productive in the terminal. That's our end goal. And just what's so fascinating is the terminals barely changed since the 70s. And yet every single developer, hardware engineer, software engineer, data scientist, uses it like every day, every week. Like it's a core part of their workflow. And yet, it's really hard. Like it's this – Everyone remembers the first time they use the terminal. And it is just ridiculously hard. Key bindings are different. It's this black screen. You can't use the mouse. Like most of the time, you can't use the mouse. It's just really intimidating. And so what we're trying to do with Fig is make the terminal easier and more accessible for beginners, make it more efficient for advanced engineers, and then make it more collaborative for teams.

And so what we realized early on is there are so many pain points in the terminal. Like, we cannot solve them all ourselves. So Matt mentioned like the API is in public right now that we've built. But we want to make that public as soon as possible. I would say that in the next couple of months, it'll be like – Definitely, within the next couple of months, it'll be live and open. And people can start building it. Maybe by the time this podcast is announced, like it'll almost be live. So we've built this API. And now we want to start solving problems with it. And we also want to enable the community to solve problems with it. So let's look at some of the problems.

And then let's look at how we think about solving them. The first problem is, I have no confidence when I'm typing in the terminal. It's this black box. I'm like literally typing. The cursor isn't even blinking. It's just like this block cursor. Really intimidating. We build autocomplete to solve that. It gives you the confidence that what you are typing is correct. And for advanced engineers, it's purely a typing shortcut. It's like you can type faster.

Cool. Okay. So that's one problem. What are some of the other problems out there? And the next big problem we've identified is around your developer environment. And so these are what people call your dotfiles. It's like your bash RC or your Z shell RC. If you need to customize your terminal, which you have to do when you join a new company, you often have to add environment variables, or you have to – Like you need to download certain scripts and source them in your terminal before it opens up. Like that all happens in this thing called the bash RC or the Z shell RC. Think of it as just your configuration for your terminal. Most people are probably somewhat familiar with it.

And the thing is like that's also really hard. Editing your dotfiles is like really difficult. Most of the time when you edit settings in a web app, you have this nice GUI and it has little suggestions and descriptions. In bash RC and Z shell RC, it's a plain text file that has to exist in a specific location on your device that has no instructions around it. It is just like it exists.

And so what we want to do is sort of think through what are the pain points developers face around their dotfiles. And we've sort of isolated four pain points. One is the actual editing experience. Like it's, as I said, plaintext file. It's really, really tough. How do I make that easier? Two is how do we make the discovery of things to add to your dotfiles easier? What does this mean? Well, a lot of people want to customize their prompts. A lot of people want to customize their color scheme. People want like specific aliases and other workflows to be embedded. Like there's no real centralized place to go and do that. There are some tools like Oh My Z shell. But even then, as I just said, editing your dotfiles is cumbersome. And Oh My Z shell, it's like the easiest of them all. But it's still like a little strange. And so how can we make the discovery aspect of adding new tools that you just randomly find on Hacker News to your terminal environment and like develop a workflow easier?

Then three is, once you've made these changes, what happens is you forget to like sync it to some centralized place. So you go to a new computer, and you just have to set it up manually

from scratch. So it's how can we sort of sync the changes that you make on one device to a new device, to another device you have, or to at least some source of truth in the cloud? And then finally, there's the replication process of, “When I go to a new machine, how can I just set up my environment, like my dotfiles, exactly as I had them set up on my other device?”

And we can talk about a little bit more. But basically, we're making it ridiculously easy to edit your dotfiles. Like, we do it in this nice keyboard-driven GUI that's as fast, if not faster than before, but gives you those instructions that you expect from a sort of web-based like app. Discover – Like we're going to build a plug install. Like we have actually built this plug install, we're going to launch it within a week, around discovering new additions to your terminal. Three is syncing. It's like we will handle the sync. We’ll automatically sync it to your devices. We’ll automatically sync it to the cloud. And four is replication. Well, if Fig sort of open source cloud hosting of your dotfiles is the source of truth, then replicating your dotfiles onto a new machine is as easy as downloading Fig and just logging in. And it just suddenly syncs.

[00:19:47] JM: Could you put more context around why some of those features are useful? Like why do I want multidevice syncing for my terminal data? Why is that so important?

[00:20:02] MS: This is important because having a consistent developer environment means that you can get up to – You can be productive on any machine that you're using. Developers have this tendency to be power users. They customize their workflow. They have special key bindings and aliases. And having to set that up on every device that they go to, especially if it's not replicated across devices, means that there's inconsistencies from one machine to another. So something that works on your local device might not work on your remote machine, or if you use a Cloud IDE. And so we think that consolidating this information, and making it really easy to edit, but also to sync everywhere will supercharge people's terminal experience.

[00:20:43] BF: An example is if you go to a new computer – This is like not even a technical example. But say you're used to using a specific shortcut that you have on your phone, or you have in your computer, and you go to another device, and suddenly that shortcut is missing. You feel it. It feels weird being on that new device. And so how can we literally take your environment and workflows everywhere you go? Like, it is actually a painful thing.

And with cloud IDEs and SSH-ing into remote machines, you're not switching physical devices all the time, but you aren't going to new machines every time you SSH every time you set up a like an IDE in the cloud with GitHub Code Spaces, for instance. So with that trend, continuing, like we think this is going to continue to be a big problem.

[00:21:26] MS: Also, you can see that this is a pain point for developers if you just go to github.com and search dotfiles. It's very common to see developers create an open source dotfiles repo where they sync all this stuff manually using Git. But then you have to remember to pull down the changes on your machines. And there's a little bit more overhead when it comes to replicating and syncing this stuff.

[00:21:51] JM: What have been the biggest engineering problems in developing the syncing functionality?

[00:22:00] MS: I think to start off, like a tricky thing, in general, when dealing with the terminal is backwards compatibility. We are trying to create new interfaces and affordances on top of a technology that hasn't been updated in a long time. And so we need to be really flexible, because our schema needs to support all sorts of different CLI tools, and plugins, and configuration languages. So I think that that was kind of an early challenge of like how can we create a schema that represents plugins and dotfiles that's flexible enough for basically all of the complexity that people put in their dotfiles currently?

But then kind of from a specific implementation perspective, we wanted syncing to be seamless and instantaneous. So the way this works currently is you can edit your dotfiles anywhere using fig. And then changes are automatically reflected both in every terminal session on your local device. But in any terminal session or – Yeah, in any terminal session that's on a remote machine or another computer as well. And we do that using WebSockets. So anytime that there is an update, we send out this ping to all of the devices you have set up with Fig, and then they can automatically be updated without you needing to manually pulldown changes or configure anything.

[00:23:09] JM: Gotcha. So the engineering around the syncing process, are you moving that config data to like a database in the cloud? Or are you just copying the configuration files and

storing them in an S3 bucket and then replicating them elsewhere? Or what's the engineering behind that?

[00:23:34] MS: That’s a great question. So what we've done is we have this higher level abstraction over what ends up in your dotfiles. We call them blocks. And so a block could be an environment variable. It could be an alias. It could be a function. It can be a custom script. And when the user edits their dotfiles using Fig, they're ultimately creating these blocks.

And what's cool about this kind of higher-level abstraction is it means you can define a block once, and depending on differences between shells, or environments, those blocks actually might be compiled down to the shell script differently because there's some syntactic differences between zsh, or fish, or whatever.

And so what we do is we store this higher-level representation. And then whenever you need to download your dotfiles to a new computer, we compile this representation down to the shell script based on certain parameters you might send. So say, you only want certain blocks to be sourced in an interactive login shell, or if you only want them to be sourced on Linux, we can dynamically generate the dotfiles depending on these different variables.

[00:24:36] JM: So when you think about how this functionality changes the workflow of a developer in the terminal, what's the biggest value add?

[00:24:48] BF: So in the really early days of Fig, actually, when we did our first podcast, we were building a ton of tools, and they're all separate. And so you can think of this today as like there are hundreds of CLI tools out there. And what we found is users would download fig in early days and they would open up one of the apps and they'd be like, “Oh, this is really cool.” And then they'd never go back and use it.

And so what we're thinking with this sort of new reimagining of Fig and how it works is we're big on just simplicity. There's one thing that you need to know to do anything to do with Fig. And that's the Fig command. Literally, you just run fig. It's going to open up this nice UI. It's all keyboard-driven. And it can sort of speak to your terminal. It's a Fig app at the end of the day. So that's the one command we want you to know as like Fig.

And what we sort of want to train the behavior is, if you want to make a change to your configuration, you want to edit your dotfiles, you want to update the path, you want to install a new plug in, you want to add an alias. Like, literally, anything to do with your environment or your workflow, just type Fig. That's like all you need to do.

And so the cool thing about that is, well, once you started typing fig for editing your dot – Like your shell configurations, like what else can we build that would be exciting for individuals, or for teams, or for enterprises that you're already using Fig every single day. So what else can we do? So examples are around secrets. Let's say you have some secret that you want to host. You currently don't put secrets in your bash RC, or maybe you do and you'd be like very protective about it. But if you want to make a change to a secret, you often have to go to someplace in the cloud, or you have to change some other file. Why can't we host secrets? Why can we make it easy to not just edit your shell configurations, but your vim configuration, the CLI tools that you have installed on your computer? Like any other sort of – Your VS Code configuration, for instance. Like, those are all part of your developer environment.

And then same with workflows is like what if I want to – I've run the same command 50 times in a row. I want to make it faster. Like we can prompt you to go to Fig and sort of say, "Hey, you know what? Here's some bit of code that I need to reference in the future or that I want to make faster." Fig can just be the source of truth for that workflow.

And what's really interesting is this applies for individuals, but it also applies for teams. So you join a new company, and you're like, "Oh, crap, what are the CLI tools I need to have installed? What are the onboarding steps I need to do? What are the environment variables I need?" Like it can all be hosted in a team apart of just running the Figg command.

So we really see – As I've said a ton of times, do you think of Fig as this he source of truth for your developer environment or for your develop a workflows? the more stuff you put in there, the more it makes sense to keep putting more things in there? I know, that sounds a little weird. But yeah, the idea is like you start by editing your dotfiles, which is a really common workflow for a ton of developers. And then you can just keep layering on more functionality. Because no one wants to have 500 different CLI tools and 500 different apps. If we can just be this one thing that's really safe, really secure, and like you sort of associate your developer world with, like, we think that's a really compelling pitch.

[00:27:59] JM: Have there been certain workflows that you focused on, like you could think about data science workflows, or DevOps workflows? And I would think that by focusing on those prototypes, you could design, or think about, or ideate around what kind of added terminal functionality you should put in? Have you gone through any of those exercises?

[00:28:27] BF: So, yeah, I think we've got – Our entire life, the past a little while, has been going through every workflow. And the interesting thing is there are just so many, and they're so particular, two different types of developers. And we want to solve all of them. And I think in the early days of Fig, we tried to do everything. And we ended up doing 50% of a good job of everything. And so what we decided to do since is let's just focus on a few really cool workflows and absolutely nail them, and then go to the next, and then go to the next.

So some examples are the JavaScript community. NPM is really popular CLI tool and Yarn. Like, how can we make those – Just the experience of using NPM and Yarn as good as it possibly can be. But we have completion specs for NPM and Yarn. But like what are you actually trying to do? Most of the time you use NPM, you're installing a new package, and you are like running some scripts that you have in your package dot JSON. Maybe you're doing something with like the NPM package registry for like a private cloud thing. Maybe you're logging in. But most of the time, you're installing something, or you're running a script.

So for the running the script, we show you, what are all of the scripts that are available in the package dotJSON? You do NPM run space. We just list out all of those scripts. As I said earlier on, we give you the confidence that, yeah, when I type NPM run Dev, this specific repo has a script called Dev. Another cool example with NPM is installing packages. When I type NPM install, like, “Oh crap, I'm installing Redux Thunk. Okay, is it Redux-Thunk? Is it Redux Thunk with a capital T? What is it?”

We when you run NPM install space, we actually just search over the NPM registry, the package registry. So we just show you live. It’s as if you're searching in the browser like Redux Thunk and you're getting the name. But it's happening as you are typing in the terminal. The workflow we've saved is going to the browser, getting the exact name of the package, and then pasting it into your terminal. Like that all now happens as you are typing, which is super cool. You type and you can see. It's like debounce search over the registry. It's super cool.

This applies to pip. Pip installing something for Python. Like, I use Django, and I need to run some – I think it's like pythonmanage.py. Like that's a really specific workflow in Python. But we offer like special completions for the Django users. And these things apply to package managers like Brew, CLI tools like Cargo for Rust, and like each one for each language. And then finally, there's Git, probably Git and CD are probably the most use CLI tools. So with Git, you have a really specific workflow, you add a file, you check out a branch, you add a file, you commit the change, you push the change. We just make those experiences and that sort of small workflow as fast as it possibly can be and still discoverable and still easy.

[00:31:14] JM: What about the workflows for crypto developers or Web3 developers, whatever you would want to call them?

[00:31:25] MS: There are a bunch of CLI tools that are specific to Web3. And I believe that we support a few of them. I'm not exactly sure the name.

[00:31:36] BF: Yeah, we can look them up. It’s the same idea. NPM is used by the JavaScript community. And there are tools that are used by the crypto Web3 community in the terminal. And we support those tools. And we support descriptions for the tools. And it hasn't been a focus for us as a team. But the beauty of what we've built is all of this stuff is open source. And so what we found is there's a JavaScript developer out there who uses Git, who uses CD, who uses NPM, who uses Brew. They like use all these tools, and they go, “Wow! This experience for these five or six things is just phenomenal. I want this experience for my CLI tool, for my developer workflow.” And it's all open. And it's ridiculously easy to build like a completion spec. It takes two minutes from – Like, literally, you go to your computer now, fig.io/docs, you will like run through the instructions, you can have a completion spec, like mini completions for a new CLI tool up and running in literally two minutes. It's crazy quick.

And what we found is a lot of developers love using Fig. And then we're missing just one little thing. But it's so easy to contribute that they just go and contribute themselves. And then it's open, and they push it to the public repo with Fig/autocomplete. And then the whole world gets to take advantage of this one tiny little workflow that's useful to the one person. It turns out it's useful to hundreds or thousands of people.

[00:32:52] JM: So when you pull back and look at the progress you've made, since we last spoke, I think it was like a year ago or so. When you think about how your vision for the product has changed, what have been the most notable evolutions or developments?

[00:33:13] MS: I think a key thing has been a shift from GUI-first to CLI-first. And so what this means is, originally, installing Fig meant pulling down a MacOS app. But we want to go cross-platform and we want Fig to be everywhere. An easiest way for Fig to be everywhere is just whatever package manager you use, pull down a CLI tool, and then boom! Your setup.

So this has kind of changed how we think about the product. And so it means moving some logic to the cloud, some logic to a daemon that's constantly running on your device that can do this sort of syncing of configuration files. But that's been kind of a key change in how we think about the tool.

[00:33:55] BF: Just to clarify. From the engineering perspective, we are now CLI-first and like very – Previously, it was a lot of clicking around. Yeah, we want to go across platform. But the way we thought we do that was a desktop app. Now it's a CLI. Still, from the user perspective, though, when you download Fig, we have simplified it. As I said, you just type that Fig command and the whole world opens up in this sort of nice GUI. So it is actually more heavy GUI than ever, but it's keyboard-driven. It's super-fast and keyboard-driven, but you get the output and display of a GUI. So it's as fast as the CLI. It's as fast as the terminal. But you get this rich display of a GUI.

And yeah, that hasn't changed too much. It's just more of the engineering implementation. The reason why it's taken us so much time is we have to integrate at the operating system level. Like how do we get the cursor position in your tersminal emulator when we don't own the tumble emulator. It's like it's a really hard challenge. And then the terminal emulator level, when you switch tabs, how do we know you've switch tabs? Like we need that information. When you open up a new pane in iTerm, if you use tmux, we need that information.

At the shell level, if you use bash, or Z shell, or fish, they're different shells, they have different hooks available, we need to do those integrations. And then as I said, we work with hundreds of CLI tools. Each of those CLI tools has really specific workloads that we want to absolutely nail. And so, yeah, honestly, the vision has stayed the same way. We want to be this app ecosystem

on top of the terminal. The engineering infrastructure has taken a ton of time to get really right. But now it's fine. Like it feels super stable. We have these integrations. It's very, very scalable. Now, honestly, we're just executing on the vision that we've always had. So the vision really hasn't changed too much. It's like we want to make you and your team more productive in the terminal. And we've built the infrastructure that makes it ridiculously easy to do.

[00:35:44] JM: And we talked a little bit about difficult engineering problems. But I'd like to get a better sense for what the canonical problems of building a novel terminal extension have been. What have you guys struggled with more broadly?

[00:36:04] MS: So I think, in general, the main difficulty is none of this is greenfield development. Everything we build has to deal with the baggage of 50 years' worth of changes in programming languages, changes in – What type of computers people were using. There's just a lot of cruft that gets built up. And I think that that's honestly kind of the main challenge, is like if you were sitting down and building something from scratch, like it is easy to do lots of new stuff. But when backwards compatibility is a core constraint, you need to be more thoughtful in how you develop things.

[00:36:38] BF: People download fig, and in the early days, it would just break. Fig wouldn't work, or it would slightly change one of their settings, which was really important to them. And yeah, we've just been working on integrating with everything, which is important to us. Backwards compatibility is really important. And now that we're finally getting to the point where most people download Fig, and there are zero problems whatsoever, now we can like really start scaling it.

So yeah, integrations is just – We want them to work everywhere. That's the thing. It's like you shouldn't have to choose, "Should I use this? Or should I use Fig?" You can use whatever the hell you want. Fig is going to be there. We've integrated it.

[00:37:18] JM: do you guys have a system for detecting bugs and pushing them to you guys? Like crash tool management or something?

[00:37:30] MS: So, testing has become really, really important, because there just are so many variables at play when you're building a desktop app. Like you're not insulated from the user's

local machine. And especially when you're a terminal extension, where people do all sorts of strange things in their dotfiles, you have to plan for everything. So we have a pretty comprehensive system of dotfile tests that basically set up development environments with various different dotfile plugins, like Oh My ZSH, or Starship, or pure, and various permutations that we've seen in the wild that have historically broken Fig's integration. And then anytime we're making changes to our shell integration, we run a test suite across all of these different environments and make sure that the events that we expect to get are still working. So that's improved the stability a lot. And then in terms of crash reporting, we use Sentry, but we really, really make sure that we don't track any kind of user data. These are just reports about like if something like crashes and panics, it will send us back the results.

[00:37:30] JM: I guess to wrap up, I'd like to get a sense for a little bit more about how engineering has gone and like some of the, I guess, like cloud services you've used to build the new syncing systems. And if there's any other engineering internals you could talk about? I guess we explored that in some detail in the last episode, but maybe we could revisit the engineering and go a little bit deeper.

[00:39:06] MS: Yeah. Originally, Fig was primarily a desktop application. We didn't have a huge cloud component. And so our stack was pretty simple. It was just a node server, I think, on Heroku, hooked up to Postgres database.

As we've started to expand out what we do on the backend, we've evolved the system slightly. We're still using node and TypeScript on the backend. Still using Postgres. But now it's become a more core part of our stack. So on top of Postgres, we use Prisma as an ORM. And I think we've moved from Heroku to render for most of our hosting just for convenience sake.

[00:39:45] BF: GitHub actions, CI/CD.

[00:39:46] MS: Yeah, we use GitHub actions for CI. I'm trying to think of anything else that would be interesting. Like, I really like where our stack is at. Originally, we had a bunch of hacky prototype code. And I think that that's kind of been a big change over the last year as we've matured the product and stabilized a lot of the early engineering decisions. And so now everything is a combination of Rust for low-level system stuff. And then TypeScript React for kind of the application layer. And we are hiring engineers both for Rust roles and for full stack

backend roles. So if you're interested, you should check outthink.io/jobs and see if you might be a good fit, because we would love anyone who's interested in terminals, in low-level systems programming, and just making the developer experience seamless. So just come check us out.

[00:40:34] BF: One thing Matt also didn't mention is we, as I said earlier, integrated the operating system level, terminal emulator, shell level. So we have to write in the language that those tools expose. So to integrate with some terminal emulators, we can do it in Rust, or we can use like accessibility API's in Swift. And with other terminal emulators, we have to use Python. And with some other ones, we have to use some other accessibility API's. With the shells, we have to integrate in bash, in fish, in Z shell, in those languages. So we are using – As I said, we want to work everywhere. And if we need to integrate with Tmux, we're writing in the Tmux conf, like whatever configuration file you need to make sure we can get the integration done.

[00:41:15] JM: Cool. Well, guys, thank you so much for coming on the show. It's been a real pleasure talking to you.

[00:41:19] MS: Likewise. Thanks so much, Jeff.

[00:41:21] BF: Thanks so much, Jeff.

[END]