**EPISODE 1127**

[INTRODUCTION]

**[00:00:00] JM:** APIs within a company change all the time. Every service owner has an API to manage, and those APIs have upstream and downstream connections. APIs need to be tested for integration points as well as for their contract, which is the agreement between an API owner and the consumers of that API.

Aidan Cunniffe if is the founder of Optic; a product built for API change management. He joins the show to explain why there is an opportunity for such a product and the market dynamics of the space of API testing and change management.

[INTERVIEW]

**[00:00:37] JM:** Aidan Cunniffee, welcome to the show.

**[00:00:39] AK:** Thanks, Jeff. It's good to be here.

**[00:00:40] JM:** We're talking today about API change management. Can you describe the problems that you see around API change management?

**[00:00:48] AK:** Yeah, absolutely. I think we have to look at it kind of in the broader trend of where technology has moved over the last couple of years. Big companies who used to have monoliths and maybe a couple of individual APIs now find themselves with dozens or sometimes even hundreds of interdependent services. So we've made one thing easier by splitting everything into smaller services, which is development and deployment. But that comes along with a coordination costs, where now a lot of every developer's day is trying to understand the impacts of the changes they're making on the consumers of these APIs.

And as we know, like this leads to real-time breaks in some companies that happen several times a week where someone will change an API. They won't understand that another team is consuming that API. They'll cause a break, which either causes a , which isn't good. Sometimes

it can cause security issue. A lot of security issues have API problems behind them. Or it just blocks another team for several days. And if you have dozens of people working, that's a lot of money.

**[00:01:53] JM:** Are we talking about internally-facing APIs? Or like externally-facing APIs like Twilio, or Stripe?

**[00:01:59] AK:** I think that's an interesting question. And when you look at how the environment has evolved over the last couple years, there's been a lot of focus on external APIs and there're huge teams. Like there're teams that are oriented around making external APIs, really high-quality. Making sure that they're not breaking consumers helping migrate, but then the internal APIs, the 10 internal APIs for every one external API, it's sort of like the Wild West. It doesn't get the same amount of love or attention to make sure that these contracts between the teams are being met.

**[00:02:35] JM:** So APIs, whether externally or internally-facing, they get updated on a regular basis. What is the workflow for updating an API?

**[00:02:43] AK:** So that's sort of the golden question. And I think, today, it's really handled differently in every company. I'd say there're sort of three approaches that we've seen in the Fortune 500. The first is like only relevant to two companies. It's like Google and Facebook. In Google, the classic story is if you make an API change in the mono repo, they have tooling that'll tell you which consumers use that, who you've broken, and the onus is on you, the person who made the change, to update other people's codebases. So that works really well if you're Google and you've invested huge amounts of time into building good tooling. But for all the other companies, it's sort of comes down to two things. A lot of companies have told us that they just never make changes to their APIs. So they add only, which leads to croft and sort of a growth of all these sort of unneeded fields as the requirements change. Or they just sort of bite the bullet and they know that as they change things, they're going to affect other teams. Some people try to build process around getting changes approved, around talking to consumers, around writing consumer-driven contract test. But there's really not a good workflow today that is easy for teams to follow to prevent these breaking changes or even to tell them who is going to be affected by a change that they make.

**[00:04:01] JM:** Is there some kind of continuous integration flow or API testing that we can do in order to make this safer?

**[00:04:09] AK:** There definitely are ways of mitigating the challenges, but they come with such a high human cost. And I think that's really where the opportunity for better tooling and this space comes from. If you write a contract test for your API, you have to test pretty much every field, all the polymorphism, all the response types you can get. It's really hard to mock these things.

We're working with a company a few weeks ago that came to start using Optic mainly because they have this huge infrastructure around simulating their APIs and mocking them. So you can do these sorts of tests, and that becomes a whole project in and of itself. So I think there are ways to do this, but there's high human cost around maintaining those test suites. And overtime, teams just over and over again have not been willing to make those investments to keep their software stable, especially for the internal APIs that don't really get the same kind of love that the external APIs get.

**[00:05:05] JM:** What is needed around API testing? If I want to build really good API testing, what does that look like?

**[00:05:10] AK:** There're a lot of different philosophies around it. There's sort of the consumer-driven contract testing model where you try to basically write test from the point of view of the person consuming your API. But that requires you to know who's using your API and how. And a lot of teams don't have that kind of observability. The best approach that I've heard is what Azure is doing. And we actually adapted a lot of their principles that we heard them share at I believe it was API World last year into how Optic does open API testing.

Essentially, what they've realized is running test suites against all those APIs in something the size of Azure is not just really hard to do, but it also is really expensive, because you're turning on and off physical servers. VPNs are being created and destroyed. So what they realize they could do is instead of trying to guess all these usage patterns and test those, they just started writing API contracts with open API. They deployed their APIs normally and they've monitored

those APIs to see how they actually behave in the real world. And when they saw all issues with that, they reported it and they were able to catch these issues based on real traffic. And I think that sort of flipping the script and using real traffic as your API test is a really interesting insight, because by nature, it will always map to the current use cases that the consumers have.

**[00:06:41] JM:** Could you explain the term contract testing and what that has to do with APIs?

**[00:06:46] AK:** Yeah. So when you call an API, you send it some data, post data, and you get something back from the API. The shape of those responses, like which fields are there, what the data types of those fields are, all of that is part of the contract of the API. It's sort of the pre-defined expectations that both the consumer and the producer have agreed upon so that the API is able to behave in a consistent way over and over again. And API change management is really about making sure that when the producer is going to change the contract in some way that the consumers are not adversely affected.

**[00:07:30] JM:** Is there anything wrong with just testing API on live traffic?

**[00:07:34] AK:** So, there's certainly cost to doing it. You have to actually look at every single request coming through or be careful about how your sampling so you don't introduce bias. But, generally, I think when you look at how people are thinking about testing today, like we really think chaos engineering and understanding kind of the baseline what we expect their APIs to do and then being able to diff the actual behavior against our expectation is much more attached to the real world than manually writing test and not necessarily covering all of the cases. So I think it has a lot of advantages. The downside is really just that you need to be able to monitor this traffic securely and you need to be able to monitor it at scale. Because unlike test, which you just run one-time, traffic monitoring scales with your application. So you have all of these other issues where you have to now scale your monitoring as well.

**[00:08:31] JM:** What is needed around API documentation?

**[00:08:33] AK:** Yeah. This is basically what we started the open source project around. There're been standards for many, many years around documenting APIs in a machine-readable and human-readable way. The most popular and sort of industry-standard way to do that is using

open API. But the challenge that a lot of developers and a lot of companies have found themselves in is that – Well, you know this. Developers don't like writing documentation. And if you build a format with sort of an expectation that a human is going to be on the other end updating the documentation and basically being that change log tool themselves, what you start to see is doc drift, where it's Friday at 4 PM and someone changed the API, but they don't update the open API spec. And now you have a deviation where your specification no longer matches the actual behavior for your application. And everything that's sort of been built right now I think relies too heavily on the human elements of doing the versioning manually.

**[00:09:39] JM:** We'll talk about Optic in little more detail in a moment, but we did a show recently about Postman. What problems does the company Postman solve around API management?

**[00:09:49] AK:** Yeah. I don't work for postman, and I'm not up on sort of their latest messaging. But I know Postman started as a REST client. And I think it's the most popular developer tool in the world now. I mean, I think almost everyone has postman downloaded and has used at some time to debug an API. What's interesting is the collection format is basically a way that you can send request to your API, and that's helpful when you're building it and debugging it and testing it. But it's not really tightly integrated with the contract for the API. So just saying here's an example request that will run and execute doesn't tell you all the information about how the API is actually supposed to behave. That comes from a tight specification that says this field can be one of these three options, whereas if you just send an example request, that's only ever going to be one of the options. So you don't see sort of the full range of affordances with a Postman in collection.

**[00:10:48] JM:** Tell me about other API tooling companies that you see as influential in the modern stack before we get to Optic.

**[00:10:55] AK:** Yeah. I think, obviously, [inaudible 00:10:57] is where Swagger, which has become open API came from. Their lineage was really as an API testing tool. And I think what's interesting is that a lot of the tooling around APIs has definitely come out of the testing space. WheelSoft is another one. They have a rival spec for APIs called RAML, and they have a lot of new API tooling they've been working on to basically be a full lifecycle management tool. Then

the other I'd say is Kong. They recently acquired Insomnia. They now have their own spec editor built-in there and some tooling that's similar to Postman and have made a lot of investments in sort of getting that to be an end-to-end experience for users who can use Insomnia and then deploy in Kong and get some validation that their APIs are actually working as expected.

**[00:11:51] JM:** Okay. We can get to Optic now. You work on a company called Optic. What is Optic?

**[00:11:57] AK:** So, Optic is sort of like Git, but it's for APIs and their contracts. So, essentially, the workflow is you start your API locally when you're developing it with Optic. So instead of like running node server, you run API start. And whenever you're sending traffic to that API, when you're developing it, Optic is constantly diffing that traffic against your spec. So if it sees a new field that wasn't there before, it tells you, "Hey, we saw this change in behavior. Would you like to mark this as bug or would you like to approve it and add this to your specification?"

And if you hit approve, it'll actually go update the API spec for you. And the advantage to this is that it's basically built-in to the developer's workflow and doesn't require them to actually write any documentation. It's simply notifies them when it sees changes and then helps them make that change. So it's a much more developer-friendly workflow that sort of doesn't leave room for mistakes or omissions.

**[00:13:02] JM:** What problems does it solve to have Optic in your workflow?

**[00:13:06] AK:** So a lot of the time, these problems like API testing get a lot easier when you have a clear idea of how your APIs actually work and confidence that these specifications for how your API behaves actually match reality actually are what the API is doing. So putting Optic in the workflow of the developer and also in your live environments, which is something we're piloting with some users now, really just gives you two things. It gives you a really drop-dead simple way for developers to provide documentation to their consumers internally and externally that's accurate and always up-to-date, and it gives you confidence that all of these APIs are working the way they're supposed to and alerts whenever behavior starts to deviate. So if it sees that you've change the API in a way that's going to break another team, you find out from Optic. You don't find out from that other team when they're angry at you.

**[00:13:59] JM:** What was your inspiration for starting Optic?

**[00:14:01] AK:** So, it's interesting. I have been working on dev tools for a while. And one of the things that I was really fascinated by is this idea of having a developer tool that could interoperate with your code. So, instead of being something that is sort of outside the code, like what if it works with existing legacy that's already there? So I built this tool that was sort of like clippy. This is what we went through YC with. There's this tool that was kind of like clippy that could read your codebase and it would light up when it's all coded, recognized and say, "Hey, this is an API endpoint. Here's what we think it does." And then if that was Python, you could then be a node and it would say, "Hey, here's a form. It looks like you want this to connect to that API," and it would do the coding for you.

And what was really interesting as we had that and maybe 10 other interesting use cases around how this sort of two-way code generator could be used. And after we did our launch Hacker News, it became really clear that the main problem people were interested in using the tool for was thinking there are APIs and there are consumers across all of their codebases.

And when we saw that, I was I was actually kind of confused by it. I didn't understand like this wasn't a solved problem, because I had seen open API and I knew people were already generating clients and doing all of these things. But when I dug in and I started talking to users and understanding kind of what's working and what isn't, it became really clear that the API world is still sort of back before source control when we were mailing code back and forth and copying our lines manually.

Ultimately, I think to really make this world that we've built scalable and without error, we need a tool like Optic that versions not just code, but also the behavior of that code. So we dug in on that. We spent the better part of a year trying like five different variants. And ultimately the version we settled on is this thing that's basically like Git, but for API traffic.

**[00:16:02] JM:** So how does somebody install and consume Optic?

**[00:16:06] AK:** So, there's CLI that you run. So you have to download the CLI. And then basically every developer who is locally building an API has something that they're sort of running in the command line to start that API. So it might be a Docker up. It might be a node server. It might be Go, whatever their command line tool is they're using to start their server. All that Optic does is it wraps that in its own little command called API Start. So it does exactly what it says it does. It will start your API, but it will route all of that traffic through a local instance of Optic. So none of the data ever leaves the machine, but it's comparing basically every request you send to Optic against the spec and giving you these diffs so you know just like when you change a file and Git tells you there's now a diff that you have to stage, Optic is basically doing the same thing, but it's giving you options to stage changes to these API specifications. So if you do deprecate a field that's not lost to the consumers, you're able to actually show that in your history as soon as you make the change.

**[00:17:14] JM:** What's the feedback that you've gotten around people using Optic?

**[00:17:17] AK:** So the teams that have actually been quickest to adapt it have tried other things before. And I think that's what's been most encouraging to us. They have all tried doing manual workflows around open API, but either because of the scale of their API. Like they had hundreds of endpoints, and it was too hard to sort of catch up. Or because developers just didn't like the workflow of having to manually update docs. People are falling off of it. So there're a lot of teams who are really interested in this idea of having governance around their API changes. They're really interested in making sure that every team understands how all these hundreds of internal APIs work and get notified when things change. These are all really important values to them, but there's never been sort of an on-ramp that has made it easy enough for teams to keep an accurate API spec for every single service they offer. Some that's been sort of where we've oriented ourselves. It's just how do we make this romantic, but we say, "How do we make this as easy as using Git?" We really want this to feel that some of the people do every single day. They see code changes. They stage them. They commit them. We really want Optic to feel the same way, where if you do make API changes, instead of having to go manually copy that code into someone else's machine, there should just be that magic button that updates the spec and then all the things that change because of that downstream, like you might have a CI/CD thing that prevents breaking changes based on the Optic spec. All that stuff can happen in CI/

CD. But for the developer, we want the o just have this really seamless experience where when they do change the API, they know about it and they can make sure their team knows about it.

**[00:18:53] JM:** So, when people are using Optic, how are the engineers interacting with each other? When are they using Optic? How does it actually get used in practice?

**[00:19:03] AK:** So, every engineer who is building an API will have their own sort of check-out of that API. They're adding a new feature. Some features require you to change the contract at the API. Other features are changing things under the hood, and the contract is supposed to say the same. So in both of those cases, the developers running the API locally, they're hitting it with some example traffic from tests, or from Postman, or from a web browser that has a web app that connects to that API. However, they're sort of simulating their traffic while they develop. Optic is tapping that traffic and it understands whether the traffic has changed from what it expected last time. So you just go about your day. You just keep developing. Just like you don't think about committing your code to Git until you're ready to share it with others. You don't have to think about what Optic is doing until you're ready to commit your code. And at that point, Optic can tell you, "Hey, we've observed these new behaviors in your API." For the developer that was trying to change the contract, this is their validation, that they've made the changes properly. They're adding it to the spec. And Optic is verifying that that is indeed how the API works.

For the developer who was changing stuff and didn't mean to change the API contract, it may have introduced a bug by accident. They're also made aware of the fact that they've changed something. They've changed the behavior of the code and of the API without realizing it. And now they can catch that and hopefully revert whatever code changes led to those behavior changes.

**[00:20:29] JM:** Very useful. Okay, let's talk a little bit about how it's actually built. Can you give an overview of how you built Optic? What is in the stack?

**[00:20:36] AK:** We started this out – Like I said, we did a lot of experiments. So, the differing engine is written in Scala. There is a node CLI, and that's basically how we start your service and proxy your traffic. And that's all running locally. The big change we're making now over the

next couple of weeks is we've actually been porting the diff engine over to Rust, because some of the companies using Optic now have the biggest API responses you've ever seen, and we need a more performant runtime to actually work through the amounts of data we're seeing. y the end of the year, it will be sort of Rust with these node CLIs built on top of it.

**[00:21:15] JM:** Why do you use Scala?

**[00:21:17] AK:** At the time it was just because we were experimenting, and it's something I knew really well from previous work I had done. But I am a big believer in sort you have your one to learn and your one to throw away. Now that the project has been scaling in bigger companies had been adapting it, we need to sort of – This like the last piece of the system that hasn't really gotten a full rewrite. So it will now.

**[00:21:38] JM:** What's the hardest engineering problem you've had to solve in building Optic so far?

**[00:21:42] AK:** I think just – It's not a huge amount of code, but it's taken a lot of time to figure out how to presents changes to a user. So I think the early versions we had if there was a new field like deeply nested, shipping information, address, ZIP Code, it would be like shipping information on address as ZIP Code changed. And like it's really hard for engineers, especially ones that are unfamiliar with the API to like follow long trails of logic like that when it's in text form. So we've probably iterated on the UI around this many, many times to really make it as visual as possible. So when you're within Optic, we show you the response the server gave you, kind of like what Postman would show you, but we highlight it and say, "This is the part that doesn't match." That's been really hard.

And then also, these scalability issues, we're still solving them right now. But for these very large APIs with hundreds of distinct fields, the performance has been sort of a sucker. So we've been working hard to improve that over the last couple weeks. And that's probably our biggest project for the rest of the year, is sort of get this in shape for these bigger companies with giant APIs.

**[00:22:53] JM:** Are there any other interesting design challenges, because this is kind of a unique application? Product design challenges in terms of how you display data to the user?

**[00:23:04] AK:** Yeah. Well, one of the challenges we had early on was when you run one of these sessions. So let's say you have a bunch of requests in Postman and you them in a certain order, our initial instinct when we first released the tool at the beginning of last year was just show things in the order that they happen. So for every issue that we – Every diff we detect, let's just show that in the order it came in. And that seemed like the most logical way of doing it. And we figured there might be something meaningful about the order you did things. But it required massive context changes. Like when you were looking at Git request for your shopping cart and then you approve something and then all the sudden you're looking at a user's profile request and then you hit approve there. And then you're back in the shopping cart again. It was a lot of context changes. So we had to experiment a lot with kind of the right way in the right order to present the data to you.

And then there's also this challenge of telling you what was expected and then telling you what was wrong. Because in Git, you just see what it was before and what it is now, those red lines and those green lines. It's a little harder to tell you what we expected and what we actually saw and figure out the right way to present that to the user.

**[00:24:14] JM:** So as the business has grown, how has it changed as you've gotten new customers? Have they fed into you with new feature requests or changes to the system that you have had to implement?

**[00:24:25] AK:** Yeah. I'd say there are two ways it's affected us. The first is it's added bigger requirements. So our big push over the last two months before we started focusing on performance was accuracy. A lot of teams are trusting this to be the thing that's validating their APIs are working properly. So, a small bug in our system might seem low priority in another kind of startup. But for us, like because we are providing a guaranty and a confidence as our main value prop, the bigger and sort of more important these customers are. The higher the impetus on us is to make sure we're always right. So that's led to a big investment we've made in testing.

Again, another similar kind of thing where there's more constraints from customers is that these giant API bodies have not performed well inside of our diff engine. So the scale of their APIs are starting to affect the architecture decisions we make. And then the other kind is definitely workflows. I feel like every time we release something new, it's great that there's something new, but then it also comes with a question. So we have pilots right now with Optic running in production environments. And as soon as we started communicating that this was running in real environments and doing the diffing live, that's great, because it can supplement a lot of your testing. But then people get the idea, “Okay. Well, since you're looking at how my customers are actually using the API, can you tell me which parts of this API can deprecate? Or which fields are never used anymore?” So we’re definitely tackling a big problem. And sort of the more we release, the more ask there are for sort of the imagined capabilities of either the new data or the new workflow that we released.

**[00:26:04] JM:** Tell me about some other difficult engineering problems that you’ve had to solve.

**[00:26:08] AK:** Yeah. I think the one that we’re kind of most proud of actually is how we represent the API’s specification. So one of the challenges I think that all the other API tool makers have been having for many years is that you have a format like open API that evolves, and there was Swagger, and there was OpenAPI and others, three or four versions of that. And all of these big companies that have adapted it, every time there's an update, they actually don't want to update, because that's a lot of work for them to sort of update their API spec to match the new version of the API spec.

So a lot of people don't want to update. And because of that, tooling vendors have had to support, basically, a half-dozen versions of open API that all say the same thing slightly differently. And that's a huge engineering burden. So most of the use cases you see around generating SDKs and displaying documentation are fairly simple, but that when it comes to doing verification, there’s now probably 50 different libraries you can use to verify that your code is meeting your open API contract and they're all going to verify it slightly differently to a different set of rules from a different time period.

So I think one of the things we thought really early on is if we’re not requiring engineers to manually write a file anymore, we can throw out the human readability requirement. And we've

actually built a spec that looks a lot like CQRS event streams. So every time that someone adds a field, there is not an updated nested JSON structure. There's just a new row that says feel this was added to parent ID, this object. And what sort of naturally has come out of that, we hadn't fully reckoned with this. And when we made that engineering decision to build sort of event-driven spec was that a lot of the questions that you want to answer with your API specifications come down to how something has changed overtime. And what's cool is in our spec, you can sort of adjust the offset and see what has changed this month or what has changed since last release, or since the beginning of time?

And any other sort of specification, there's not really good open API differs that can diff to open API specs to each other, because there are all of these different versions, all these history, all these legacy. So a lot of the decisions we made to sort of build our own internal spec that then exports their open API, at the time it was just making our lives easier. But now, almost all of the important next features people are asking us for have to do with this time axis and how the APIs are changing over time. And I think we haven't fully explored beyond like a change log what those features look like. But it's really clear there's a lot of cool stuff we're going to be able to do because of that architecture that will be more challenging for traditional architectures.

**[00:29:04] JM:** Do you think there's any issue with overburdening developers with new tools? I feel like there're so many new developer tools. You just kind of become overwhelmed with the volume of things. It makes you wish for the days of just having an IDE and a command line. Do you really need all these other tools?

**[00:29:22] AK:** It's a good question. And I think it's one we should always be asking as a community, like what are we making easier? What are we making harder? And I think if a developer tool author only answers the what are we making easier question, then I think that there's definitely something missing that you should inquire further about, because that what are you making harder is important. And I think backing up even more, like we broke up these monoliths into microservices and we traded one kind of complexity for another. So I think there's always this tradeoff happening.

Ultimately, I look for sort of where the most value is for a team. Developers want to be able to just work on new features. They want to be able to build things independently and

asynchronously. And at the same time, they don't want to be broken by their peers. And I think it out for us, that's sort of been our thesis, is how do we make the safety that sort of all businesses are looking for something that is just sort of a drop-in automatic thing that doesn't require a huge behavior change. I mean, literally, you just have to start using our command line, API Start command, instead set of whatever your current start command is. And when you commit your code, if there is an API diff, you have to review it.

So we've tried to make it as easy as possible to sort of get the benefits without the tradeoff, which is, "Hey, now for six hours a week, you have to write open API specs, or review PR's of people changing the open API spec, or write tests to verify the open API spec." So we're really trying to sort of figure out how do you create that new tool and provide that value without the workflow being something that's onerous.

**[00:30:55] JM:** Got. Well, what are the biggest problems that you see in front of you to solve?

**[00:31:00] AK:** I think, for us, now that we've built the open source version and gotten decent amount of adaption around it, working on sort of what this looks like in a larger company environment. We've been fortunate to pile it with some pretty large companies, but we're still in sort of a subset of those companies teams. So figuring out what this kind of looks like at scale in a company? How is object the tool that's making sure teams are breaking each other that's notifying one team when another team is proposing certain changes that will affect them, and really scaling this up to being summoned as part of the decision-making process. There're not as many technical challenges in doing that. That's much more sort of a human challenge, a workflow challenge. And again, to your earlier point that I think was a good question is like how do you add these capabilities without it becoming sort of just one more workflow to do? So I think we're going to be iterating a lot on that part of the product and trying to scale the success we've seen with object being the tool that just makes maintaining and writing API specs easy into flipping a tool that gives us a lot of safety across our entire company that we're not breaking each other, we're collaborating well and everyone is sort of aware of how these things work. And it's not just living in one engineer's brain how one system is supposed to work with another.

**[00:32:22] JM:** From the business point of view, how do you see expansion into other adjacencies?

**[00:32:27] AK:** Yeah. This is something that is sort of at the core of our thesis. I think that when you look at sort of the primitives in developer tools, kind of everything steps up from version control. Those primitives are used in CI systems. They're used in build systems. They're used in infrastructure world. And I think we've started to sort of source control as sort of the be-all and-all where we do all of our versioning. And I think ultimately in this hyper-connected world where most of your dependencies now live outside your company, not inside your company. It's really important to not just version the code, but to also version the behavior of the code. So why is it that we're versioning entire node modules and not individual functions inside of them? Why is it that we're versioning entire APIs and not specific endpoints?

I think when you break down this problem and you start basically creating snapshots of how everything in your system is supposed to work, those snapshots will change, of course. Things will evolve over time. But I think it's just as important. Imagine if in GitHub on the left you had the code that changed on the right. There was a little summary telling you what the behavior changes are. The contract is different. There's this new side effect. We ultimately sort of see ourselves, the name Optic, all about vision, is we want to give people sort of observability into how all these pieces in this complex world fit together and hopefully not do it in a way that requires you to manually write specs and write documentation. We think those things should be side effects of the system monitoring how things are working today, not things that you have to provide for and then update later.

**[00:34:15] JM:** Well, are there any other business opportunities that you see coming down the line aside from those?

**[00:34:19] AK:** Yeah. I mean, I think applying these principles generally is sort of the biggest opportunity we're looking at, is how do you make the version control primitive based off of the behavior of a system, not based off of the code? And I think once you have this map, like one of the cool side effects of companies that have adapted object across a whole organization is now they know how every single API works. And there're other classes of tools that want to know how your APIs work. No-code tools, and the Airtables, Zapier's of the world. So why aren't you able to login with someone like Optic, grant access to how your APIs work. Now you can use all the data inside of those tools. And then those change, optic and web hook to them and say,

“Hey, this API, this internal API you were using for this marketing task is now different. You might need to ask the next user to login to update the formula.”

Ti me, there's like there's this whole next level of sort of connecting all of these low-code, no code and alternative programming environments together once optic is sort of this layer that's keeping an up-to-date understanding of how all these systems behave. And that’s sort of how we get round trip back to the original vision of Optic when I first started this two years ago.

**[00:35:33] JM:** That's really exciting, the whole idea of being able to stitch together no-code platforms, because this is one of the emergent frictions in the no-code world. It feels like we’re moving towards an age where instead of having like Linux versus Windows, you have like 100 different Windows style walled garden environments of no-code tools where you'd love to get them interacting with each other pleasurably, but it's sort of tricky too, I think.

**[00:36:01] AK:** Yeah. My lineage is I actually founded Dropsource, which is one of the bigger low-code tools a couple years ago for mobile apps. And we had this point of friction where we were deciding whether we would build our own backend as a service walled garden, or if we would import Swagger back then to let you drag-and-drop UI elements that would connect to those services. Yeah, this is a problem that I think is across that whole space. And when you look at Airtable and Retool’s websites, they have like their own page for defining your API sort of like an API spec where you’d say, “Hey, [inaudible 00:36:36] one request works. Here’s what it will send back.” And yeah, and all of that data that connects these tools together is really behavior data. It's understanding how a system is supposed to behave. And right now, yeah, I think when humans have to maintain behavior data themselves, they just don't do it, because it's a lot of work. And we don't ask people to manually merge, Git commits anymore. We made tooling for it. And I think, ultimately, because of how interconnected things are, you're going to need tooling to maintain accurate understandings of how all these different systems work so they can connect together.

**[00:37:08] JM:** While I have you, any other reflections on the no-code space?

**[00:37:12] AK:** Yeah. I mean, I think, ultimately, the reason why I started building tooling that works with legacy code is because I think the low-code space generally has like an interesting

curve to it. Like you can start off either trying to work with like startups that are creating applications and then you're taking bets on whether or not those things work or not. And if they do work, they're probably turn off at some point because they'll need a feature you don't support. So it's like a temporary path. And then if you work with big companies, you're only going to get their greenfield projects, because their real app that's their primary business is never going to get ported on to a low code tool. So you'll get the experimental web UI for a new banking product. And then at some point that matures to the point where they turn off the platform.

So I think like you want to build a developer tool that's free to start and then can become more expensive for the user as they scale it to more and more users. That's what makes infrastructure as a service a business model where they can give away so much upfront, because they're there betting on the fact that many fish will scale. And I think low-code has this constant issue where it never does enough for connects to enough integrations to work forever. So it's hard to like see what that middle term and long-term revenue goal can actually be, unless nothing. I don't know a solution to that problem other than like having low-code tools that do one very specific part of a workflow and can just own that one part of how you work, but not have to sort of absorb all of your requirements.

**[00:38:50] JM:** Indeed. Yeah. Well, that's a really interesting reflection. Do you got any thoughts on the future of developer tools, or low-code tools, or the intersection therein?

**[00:39:01] AK:** Yeah. I think it's hard for me not to see things through the lens of our thesis, but I really believe that low-code tools are going to end up being things like projectional editors. So, maybe I write a react component, and I am a developer, and I add a lot of different affordances to it. And maybe I do everything but the render method. Like I have all the updaters, I have all the internal state management. And then to me, what'd be really interesting, if a designer could go into a visual tool and use those affordances that I wrote as their primitives when they're building out that one little piece of the UI, hit save and then like now I have this beautiful component that I didn't have to write. The designer gets to do the part they're good at. I get to the part that I'm good at. And if I update the code in a way that's incompatible with what the designer did, the designer gets notified and they have to fix it.

And like to me, that's like not super ambitious. It’s not trying to own an entire space, etc. It’s not trying to own the entire app’s lifecycle. It’s just saying like what's this one area where we can let a nonprogrammer contribute to our larger application. And I'm hopeful that tools like Optic that create these bridges between sort of code, the behavior of that code and whatever tool wants to sort of consume that information, I'm hopeful that that sort of creates is world where low-code just becomes projectional editors. And there can be like a really, really good one that you subscribe to and pay real money for that you scale with your business, because if it only has to do one part of the job really well and there is good escape patches, you could build a business around that and have people paying for years, for decades. So I think finding a way to sort of let people contribute in certain parts is like the biggest opportunity.

Another example of that is you know my dad in his role, he defines requirements for a lot of programmers. So he gives them giant Excel spreadsheets with formulas in it, and then they try to turn those formulas into code and they get it right 6 times out of 10 on the first shot. And then like by number five or six, they get it right. But sometimes they don't and it costs huge amounts of money when it goes wrong in the real-world. So why can’t someone build sort of a formula business logic editor that people like my dad can use that generate code that then the developer when they're running their validation or building their transaction object can rely on sort of opaquely just know that that is managed by someone who knows what they're doing and then keep on running code normally. I think those sorts of seams are really, to me, where low-code is going to end up going.

**[00:41:41] JM:** That seems like a good place to stop. Aidan, thank you for giving your reflections on the future as well as your perspective on the status quo of Optic. It's all very exciting.

**[00:41:49] AK:** Awesome. Thanks so much, Jeff.

[END]