EXPERT INTERVIEW

UPDATED
03/11/2022

Lenny Bogdonoff, co-founder and CTO of Milk Video, on the past, present and future of Javascript

TEAM

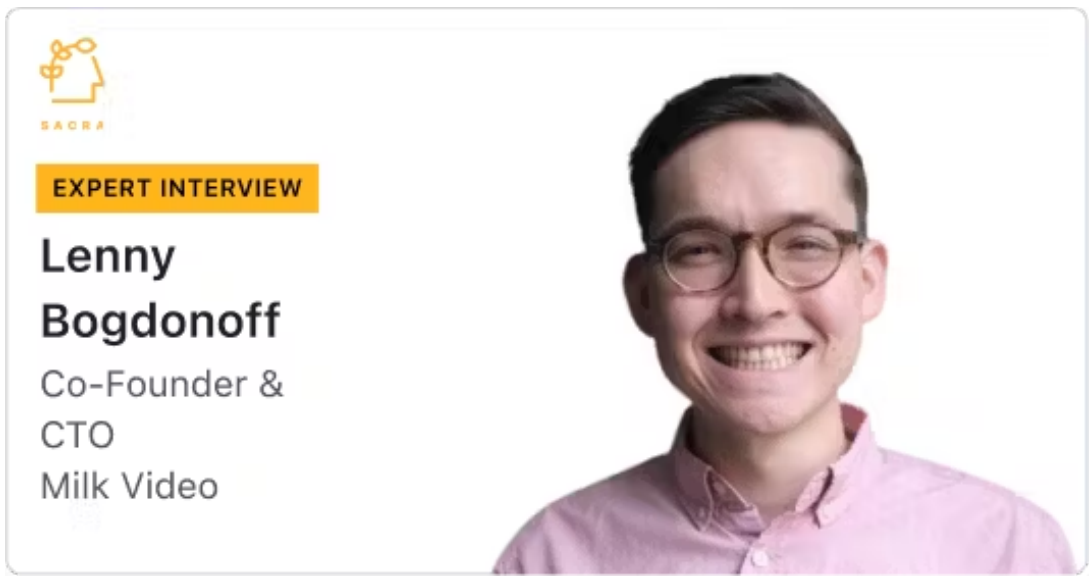
Jan-Erik Asplund
Co-Founder
jan@sacra.com

SACRA

Published on **Mar 11th, 2022**

Lenny Bogdonoff, co-founder and CTO of Milk Video, on the past, present and future of Javascript

By **Jan-Erik Asplund**



Background

Lenny Bogdonoff is the co-founder and CTO of Milk Video. We talked to Lenny to learn more about the historical context of the Jamstack, how it re-configured the state of Javascript development, and how his team at Milk Video thinks about what infrastructure and tools are necessary to build vs. buy in the Jamstack ecosystem.

Interview

What does "Jamstack" mean to you?

Jamstack specifically is an acronym for “JavaScript,” “API” and “markup.” I think it's become popular based on there being a clear delineation between front and back-end development, where back-end development has become commodified to tools like Firebase or spreadsheet, no-code database types of things, and the personal development of front-end engineers largely orients around JavaScript frameworks. I think maybe

ten years ago it was jQuery or Backbone or Ember. Today, the lower-end entry point is probably Vue -- there's less structure, and it's easier to understand how things work. And then the more mature company approach is going to be React-based. Fundamentally, there's a frontend that's plugged into a backend that often is an external service, as opposed to something that's hosted. From our perspective, having a singular interface, like a front-end interface that has codebase similarities across our public-facing static websites, our actual application, and maybe even our blog or other things -- that's the reason why we heavily leverage Jamstack-based front-end interfaces. Because our application is such a complicated web application, it becomes just easy to use the components that we've developed there for our static application also. In some cases, someone might make a static website in Gatsby or something like that. We've decided to use Next because we're able to use the same components we use for our actual application.

Kind of a little run-on there, but I think probably the most important delineation for Jamstack stuff is that, generally, you only have to deploy static assets. When you think about JavaScript, it's generally not like Node JavaScript. It's a front-end JavaScript package and some HTML entry point, like an index on an HTML file. That gets built once and deployed, and then the client does all of the requesting or accessing of remote services. So super high level that's Jamstack.

You talked about having a singular front-end interface with these codebase similarities between the various marketing sites, blog, etc. Imagine trying to build all this pre-Jamstack, what might that have looked like? And what does that singular front-end interface do for you now?

Let's say circa 2005, though maybe that's super arbitrary. I would say jQuery is in its prime, but we really don't have too many examples of well-organized front-end applications that are largely JavaScript-based. What I mean by “well-organized” is JavaScript applications that inherit the model-view-controller code organization explicitly to manage the client state or application store. JavaScript at that time was really just augmenting static HTML. AJAX -- basically, asynchronous JavaScript request to update the frontend -- was really not that big of a thing. We can think of Gmail as the emergence or

popularization of that. I forget exactly when Gmail came out, so the time range could be a little bit off.
Generally, jQuery JavaScript popularized more and more interactive logic. Then, moving five to seven years forward, we start having popularized front-end applications that are doing much more interactive work. Maybe this has to do with internet speed, and the fact that the internet got so much faster at sending payloads to the client side. We moved away from sending just static HTML updates to actually having some level of an entire JavaScript payload that was sent, and then the frontend would update because it had the templates and whatnot built in.

Backbone was essentially just an extension of jQuery. It was more for file organization, and it would update the frontend primarily based on literally updating the front-end markup using the browser's API. And then the way of updating the browser, or the client, matured. So it became not just using JavaScript APIs for updating the frontend to having templating languages, which were meant to reduce code repeating for updating stuff. And those templates then spurred their own framework -- Ember and Angular emerged in these days after Backbone, and before Ember was SproutCore and then eventually React. I think Backbone became popularized from the New York Times, the Document Viewer -- I forgot what that guy's name is.

These frameworks emerged in response to frontend end users' expectations that applications were going to be faster and update more. And it also had to do with more long-running content on a page, so video actually is a great example. You can't refresh a page if there's a video playing and you need to change things. Or maybe parts of a page would update, like a live social feed, but you don't want other things to update. So with that maturing of the end user's taste and ability to interact with the web and expectations -- and, again, also internet speed improving -- these front-end frameworks became much more mature.

Then there was this interesting line where, if you built your applications in JavaScript, your website wasn't actually indexable by the search engines. So there was a long period of time where people avoided building out JavaScript applications because their websites wouldn't be searchable, and that was your primary way of being discovered. I would say where we're

at in the timeline of arbitrary events is around 2012 or 2013. It's fucking crazy to think that that's how much time passed. If you go a couple of years forward to 2013, 2014, 2015, there was a popularization of these server-side rendered front-end applications. I forgot when Node came out, maybe around the 2010s. The tooling for JavaScript became not just "augment onto static websites," but it became their own frameworks on the client side. And then it became not just these front-end frameworks on the client-side for managing larger codebases, but it expanded to running on the backend. Now you had this unified language, which people started to harp on and try to create frameworks that were "same back-end codebase, same front-end codebase." People got excited about that. I think company executives at bigger companies were like, "Oh yeah, this will make our lives easier." So there was investment in that space, but it didn't really go anywhere.

But you had these frameworks appear, like Meteor. It's supposed to be a single endpoint for managing your backend and frontend and whatnot. As these front-end application frameworks got more mature -- and with Facebook and Google really staking their claim, LinkedIn staking their claim on Ember, all this kind of stuff -- then you had a lot more people who wanted to learn these frameworks. For a while, there was this window where it was like every couple of months there was a new framework that came out.

They were legitimately evolving and improving. The biggest improvement that came was the virtualization of updating the frontend. When that happened, everyone copied that. But up to that point, there was opinionated codebase structure, opinionated ways of delineating actions and updating stuff and whatnot. And there was some cool stuff where Google really invested in minimizing packages and being able to have JavaScript load on the fly. The code that you used on Google+ was also the code that was used on Gmail and on the Google homepage. If you opened the Google+ page, then you didn't have to download the same packages again. That's how Angular was intentionally designed. And React was designed around Instagram and slowly got integrated into the web.

At a certain point, when there was a certain threshold of framework maturity, the number of people who started to use these frameworks grew dramatically. From there, you had new ways of deploying just the frontend, and you also had new

tools coming out that catered just to those people. There was Firebase and also a database as a service company called Parse that Facebook bought and eventually open-sourced; Facebook bought it because they used it for Facebook Games or something like that. Essentially, you had these systems that were APIs as a service for storing keys and values -- basically, database entries and whatnot. And then, again, all the onus of needing a backend drifted away.

I've followed the track of the JavaScript side, but in that same timeframe, say from 2005 on, people were deploying PHP applications, premade BBS framework forums, WordPress, and Drupal. And at some level people were augmenting these WordPress or Drupal interfaces to be their backend for these front-end applications. So, whereas originally these frameworks were end-to-end -- both the backend and the frontend, it was just rendering static pages -- people started augmenting these static pages with JavaScript. And then, at a certain point, these static pages were not even being rendered in these PHP back-end services or Java-type things. They were just rendering a JavaScript payload with the initialization data, and the actual website was just an index.html file with a JavaScript package. So there was a transition where we went from end-to-end -- backend and frontend, served always from the backend -- to backend augmenting the JavaScript, to backend literally just serving payloads to the JavaScript, and finally to backend not even being necessary anymore.

Then there were also a couple of other things that happened in between. The popularization of Gatsby came out of things like Jekyll, where you would literally write all of your websites in markup language. I haven't talked about markup at all, but it’s a format of just like, “This is a title. This is your body text” -- whatever it is -- and then you run that through some kind of Ruby script or Go script, or maybe there's static builders for Java and stuff like that. I'm sure there are. Those would then generate these static webpages, and then you just deploy a big set of HTML files.

All of these things start to converge on your being able to make JavaScript applications that are indexable by the web, since the web is becoming more mature and able to render JavaScript pages. There are standards for how you would render templates or markup the data of an application. There’s a whole ecosystem of back-end external API tools that are free and good, and people can use them to get started without

having to do any personal development. And then the front-end application ecosystems also have become really mature. They continue to develop, but now generally they're just iterating around developer aesthetics, as opposed to introducing something sizably important.
That's where we are today. Now, when a mature end user expects a responsive UI, and when you're trying to do things on the web that are not just rendering a static HTML page, it makes sense to use JavaScript and move a lot of this business logic into the client side. Hosting that as a single JavaScript package and separating that from the API in the backend is Jamstack per se.

There's a lot of frameworks and languages that have come and gone that were at one point considered potentially the next big thing. Firebase, for instance, had a similar vision to what Next.js seems to be doing now, and it's popular, but it hasn't become a core infrastructure for the web. Do you see Next.js differently? And how do you think about whether it is durable or whether it might be replaced by something else?

Good for the Firebase guys -- get that big tech company money. I think there were a couple emerging currents over the last ten years, which were cloud computing, AI and ML, the increase in the number of developers -- like coding becoming a thing everyone should be able to do -- then the emergence of mobile, and the no-code movement.

I would say Firebase fit into a really interesting window where you had programming becoming really popular among consumers and not in a boring "IT"/computer science sense, aligning with mobile technologies really manifesting and Facebook getting really popular. People started to look at technology as a cool thing to do, and consumer tech itself became accessible.

There was a period in the mid to late 2000s when people really thought the mobile web HTML-wise was going to become really popular, and HTML5 was the go-to term. That really fell short. Mobile wasn't really there. iOS development just continued exploding. Five years later, the seeds were planted for some of these front-end technologies and they just kept maturing. Not as quickly as people expected, since there were all these limitations -- like, the device APIs and access to

JavaScript just weren't there, there were limits on memory, the client side couldn't store enough data and didn't have enough compute, Chrome wasn't really mature yet. But when those things started to come to light, I think that's when the JavaScript frameworks started to make more sense because the capacity of the browsers and whatnot was there. It took a while for the JavaScript APIs to mature and then actually become popularized. For such a long time, people were making websites while considering that there might be one or two percent of the users from IE7 or something. You had to think about these things. The convergence of browser compatibility, the maturing of browser APIs -- that kind of stuff I think is a really big deal that I didn't talk about at all.
All of that coming together created these new things that didn't exist before, one after another. The exciting thing about Meteor was that Websockets were not really popular for a while and were hard to implement, and they just made it easy. You also see huge funding rounds, which don't make that much sense, like Gatsby. Or a couple of these CMS services, like Prismic or Sanity. Because they're providing a resource that at some level is replacing a backend, they can charge stupid amounts of money for it. They probably have ginormous margins outside of marketing.

So, why is Next popular? Next is popular because React got popular. React got popular, but people couldn't index their websites, so Next solved the problem of React, being able to be rendered to create static websites. At the same time, Gatsby did that at some level, but performance-wise, Next is just way faster. Gatsby is super slow, especially when your websites get too big. I think Gatsby maybe popularized Jamstack as a term, the “m” being markup. Gatsby really took advantage of the patterns that were implemented by Jekyll and some of these other markup rendering things. But they used React, so people working on front-end applications could also have good content management processes and still use version control with GitHub for managing content.

That's another thing I didn't bring up. Version control through Git became popular over the last twenty years, and especially in the last ten years. Mercurial never really became as popularized. People were comfortable FTPing into their servers and just replacing codebases and whatnot for such a long time.

I mentioned cloud computing. For the end user, that means you can deploy your codebase in a pretty standard way. Tools like Heroku made that stupid easy. Netlify is an example of the early deploying of Jamstack tools. Vercel emerged and provided basically what is a commodity but made good UI/UX around it, understanding what the end user needs to do and making that very easy.

When you ask, “What's the next popular thing?” Back-end services like Firebase or Parse became really big, but I think their big picture was more around realtime web. If I understand it correctly, it was just very easy to make backends, not very high performing throughput backends. Firebase is still quite big, even though Google bought them. My interpretation is that Google bought Firebase because they saw a whole category of developers who would eventually become heavy cloud computing customers, but they would have to start somewhere, and the place they would start is key-value store databases, like Mongo-type interfaces.

All that aside, I think Next is popular today because React is popular. I think it's actually going to be popular for quite a while, as long as they keep innovating on build speed. For developers, the important things are build speed and ease of development environment, like not having huge version conflicts. If a lot of applications are written in React, it makes sense that you can share the components and the code between your static Next website as well as your more complicated JavaScript website. I'm not a genie, so I can't tell you what's going to be popular in the future. But I think within the realm of the things that are important, I think that's definitely there.

You mentioned Vercel being a commodity product but with good UX and good UI. You guys use Vercel, so I’m interested in hearing what's so useful about it and what's good about the UX? Would you consider switching to Netlify or somewhere else?

We actually almost switched to Netlify because Vercel's so stupid expensive. We pay more for Vercel than we do for some other really core infrastructure. It's pretty stupid because all they do is build your JavaScript and then host it. But it's a lot of little things that start compounding and become just easier not to change, basically. So Netlify is significantly cheaper, but the

onboarding, the development UX, just getting started -- there's enough friction where it doesn't make sense to do. They haven’t developed or optimized around that kind of interface. The way that we use Vercel, it's really just a commodity for hosting. We could do it ourselves, but there are little things that become easy. For example, they integrate with GitHub; they build your package; they deploy it; they let you rollback; they let you have previews of different packages, different bundles that are pull requests and whatnot; they manage your environment variables; they manage access between teams. That kind of stuff starts to add up. It's interesting, because five years ago it was very unusual to be able to build a pull request and have a preview branched together. Infrastructure-wise, that was hard to do. Now you take it for granted. We don't even use that feature that much, but just the fact that it's there is nice. Being able to not have to set up Jenkins or some kind of build runner and just have that work is quite nice, relative to how things worked five years ago or probably how things still work at Google or something.

“Would we try something else?” Yeah. There are tools like render.com. There's a number of emerging tools that are trying to simplify infrastructure management. is one that has good publicity around what they do. But they're all pretty much providing the same thing, which is, “Let us take your Git repo. We'll follow the instructions on how to build it. We can detect if it's a React app or Next app or something. Then we'll handle the routing so that it's available.” What's cool about Vercel is they're trying to differentiate themselves by building Next and then building APIs that integrate only with Vercel. That's smart on their side. They're basically creating defensibility for something that is just a commodity. They provide analytics and stuff like that, but my guess is those APIs are technically accessible anywhere else, but because they're building it, they can do these things that other people can't do as quickly.

I'd love to hear more about this idea of building APIs that only work with Vercel and the extent to which those might be something that you can't do on Netlify. Can you talk about what those APIs do, and how it might be advantageous to be on Vercel to use them?

The APIs I can think of on Vercel's side that we use are performance monitoring. They're serving the file, so they can track the logs of the file being served and the corresponding

time for files served. At some level they're building out their own API on how those logs are being created based on the actual library of Next itself, tracking the first connection, the last connection, the time for a file to be delivered, bandwidth, that kind of stuff. They simplify it into an equivalent to what Google does with speed testing. They have their own version of that, which is smart. I say it’s a private API because I don't know anyone else who integrates with that, but technically there's no reason why anyone else couldn't. It would be very odd for Next to lock that down unnecessarily. Given time, it would make sense that everyone does that. I think Vercel charges money for the analysis, which is interesting -- no one else can do that if they are proprietary in a sense.
The amount of investment that Vercel does is also notable. What they're doing to not be replaced outside of the private APIs is they are paying attention to what is popular. In the last year maybe or two years, a really popular change in the front-end ecosystem is stripping out webpack, which was the traditional compiler tool and was just slow. It could be 30 seconds between saving and seeing an update, or even minutes building a JavaScript application because the way that it was written was not so fast. A popular Go-based builder tool called ESbuild started getting popular and then replaced webpack in a number of places, from the regular React ecosystem to even things like Ruby on Rails and asset building. Next recognized that asset building speed is important, so they also migrated over to a faster asset builder for compiling Next using a Rust-based framework. I would say Next.js competes most closely with Gatsby. Gatsby is a big company, relatively speaking to anyone else -- think Prismic, Contentful or Sanity, those types of tools. So Next was following along with the esbuild trend, and they did that well.

Then they built out other APIs. In Vercel, you can test your dev environment using multiplayer editing tools, and they made it so that you can comment in the webpage. These are things that Next is providing interfaces for, so Vercel can create the equivalent of a browser extension but in a Vercel-hosted environment. They are basically apps on top of your Next app, almost like adware in a negative sense in the preview environments. But they’re smart features that they can talk about and really differentiate themselves with in this otherwise very simple browser JavaScript ecosystem. Because, technically, a JavaScript application is literally just an index on

an HTML file and a JavaScript file that has an entry point that then does other stuff -- it's so simple.
They understand how a designer and a developer work together, how developers work together, how code review happens, what are the priorities in rolling out features. I don't think they have a feature flag system, but I can imagine that's not a bad thing for them to launch hypothetically. They can do this in a very simple way. And it makes sense because developer needs are so prominent and clear.

And there's one more thing that makes Vercel defensible. Technically, Vercel's hosting, and value, really only comes from the moment that you deploy your code to GitHub, which then is built by Vercel. That value really only comes from that post-deployment phase. Normally, all the hosting providers are missing out on all the development activity that happens when the code is still just on the developer's machine, unless they create a really fancy command-line service or interface -- but even then, there are limits outside of testing. So this is super important: Vercel's building of Next -- in providing Next as the platform that's hosted as you're doing development -- lets Vercel have this end-to-end value chain, where their development environment is just as valuable as their deployment environment and everything in between. It's a cool thing that they were able to accomplish, and it's something that Heroku can't do, AWS can't do, Google Cloud isn't doing, even though those are really mature back-end tools. It's not defensible per se, but they're expanding their presence in developers' life cycle in a way that none of these other hosting tools do. It's counterintuitive but smart on their part.

What are the benefits of the end-to-end value chain in something like Vercel?

When you're doing JavaScript web development today, you have to build your package to be ready to be hosted in something like Vercel. You could store it on S3, a file hosting thing. But the step of building your package can take a couple of minutes based on how big the application is, so you don't develop your application -- like coding it and making changes -- and then do a full build every single time.

Instead, on your development machine you have a developer server where, whenever you make a change, the updates happen in a sub-second timeframe so that you can quickly

iterate and see the changes that are happening quickly. Generally, that development environment and production environment look a little bit different, so you are running two different commands: a build command for production and a local development command for the local development. To access the codebase locally when you do that local development environment, you spin up a local server, which you access on your machine. It's technically only accessible on your machine, though there are things that you can do to make your local environment accessible to a remote machine and whatnot.
Vercel and Next have really invested in that as an augmented service. It’s not a service per se, but Next lets you share your local development environment with someone remotely, which is something that normally you would have to use an external service to do. The value of what Next is doing is they're making the development environment part of the Vercel ecosystem, because you need a development environment regardless. If you're a JavaScript engineer working on an application, you need to host the application locally in some way. Next is just really trying to consume and eat all of the touchpoints that involve that. Then when you do the deployment, that again is a separate value ecosystem. If you weren't using Next, you wouldn't use the Next builder; you would use something like React, Vue or maybe Svelte or something like that. But the big thing is that they're building the tooling in the places where there are things that are inevitable.

Going back to your question about, “There's Next, and there was Firebase, and what's popular.” A couple of other things that are sexy that are coming out are Svelte, which is like the anti-React world. Basically, React solved the problem for people who had been doing web development up to then. And people who were coming directly into web development without familiarity with React were basically looking at React, super confused, like “Why are these things the way they are?”

They were missing the context of what problem these things solved, so introducing these abstractions was making things more complicated than people needed. Whereas in contrast to React, the jQuery world, say, was just like, “You see what you get, and you don't have to look too far beyond that.” Things that were important for people who migrated to React were like, how do you create a fast-performing JavaScript application? That’s what people were trying to do -- the virtual

DOM updating data quickly -- and that kind of stuff was hard. So React made that easy. Now people are like, “Hey, we're not creating data-intensive applications.” There are people who just want simple build packages or simple small applications to minimize the actual JavaScript payload that they're deploying. There are aesthetics that people want to optimize for or trade-offs that people are looking for, which creates the potential for new frameworks to emerge. One example is Astro, which is a framework that's interesting. Again, it allows you to minimize the amount of JavaScript you send to the client side. Svelte is another one, just like an end-to-end JavaScript. There are different aesthetics or reasons that people in different environments may want to try different things out. But it's almost like a reaction to the last generation or the current modes of doing things results in the next thing. It’s evolution within the ecosystem in a sense.

One criticism we've heard is that Jamstack and Vercel are great for getting projects up and running, but there may be limitations when you start trying to build fully featured apps. Do you see a point where you would have to move off Vercel onto something like AWS simply because of the demands of what you're building?

No. I think that criticism comes less from people who are hosting their JavaScript applications on Vercel, and more from people who are leveraging free tiers of products. They want to keep leveraging the free tiers, but they're hitting the limits based on what they're doing. That primarily comes from the back-end abstraction tools that provide database APIs and whatnot as a service. So someone who's using Firebase to build an application can get started right away. They just have to register, they get a URL endpoint, and now they can make a key-value store database. They can create log in, off.

I don't know what the limits are. But if they get past a hundred users, or they want to store multiple data tables -- something that's trivial if you spin up a Laravel instance or a Ruby on Rails application to handle your own application -- then it becomes a problem, because it gets expensive super quick. Something that was cheaper to not have to hire a back-end developer to manage becomes a huge issue. Because now, you're in a situation paying thousands of dollars for something that would be a free MySQL database or something on DigitalOcean.

This is a little tangent here, but you can scale with Vercel, and it's fine. But we're getting to the point where every time we add a new developer, it is a bit expensive. The fact that we have to pay for a new developer seat, which is $40 or $50, doesn't make any sense. So at a certain point, we'll transition. It'll make more sense to just pay to set up Jenkins or one of these build runners and then host our own front-end payloads and AWS. These are such standardized practices that it's an easy thing to do. But I think the criticism to my point is that it's more around the back-end services for Jamstack tools as opposed to the front-end side.

There is a desire for back-end abstraction tools at different parts of the stack. At the most abstracted side, you have these databases for super abstracted no-code tools that just need a backend. A little bit less abstraction is these API services that provide Jamstack interfaces for saving data, accessing it, log in, payment, and infrastructure -- that kind of stuff. The even less abstracted parts are these new things that are just coming out, like a database called PlanetScale. It’s super interesting -- it’s trying to provide highly scalable databases that never hit the limits of Postgres or MySQL. Basically, Amazon's ecosystem as a service, but very specifically around databases like infinite table, infinite column, infinite row that always perform well. Those can be really fucking expensive. But people are willing to move there because they don't have to pay a DevOps person. It becomes a financial trade-off at some level. They don't have to worry about performance as they scale.

We're talking about Jamstack specifically, but the ecosystem for any development environment is the same. Like search -- some people will pay to set up their own Elasticsearch or Solr environment. Other people will pay that French company Algolia to manage their search. Pricewise, it's just totally unreasonable, so freaking expensive to use these things. But at a certain level, it makes sense because you only have so many development resources. Hiring is hard, so it’s good if you don’t have to think about it.

That’s also the reason we use Heroku. Heroku is really expensive, but we don't have to manage our backend, and we can roll back our database. We can scale up services and not have to think about it based on spot demand. There's no thought that goes into that. Paying a couple of thousand bucks

a month for what we're getting is probably not a smart idea, but that's a trade-off that at this time we're willing to make, because hiring somebody just to think about that will cost more.
I think the Jamstack ecosystem is ideal for teams in the one-to-five-person size or environment who have no back-end development and no engineering expertise, but who can build front-end things because the feedback loop is fast, and they can learn that. People who are working on more production-level, planet-scale types of things, maybe they're setting themselves up accordingly. Same with Algolia and those types of tools. They may be bigger teams but still don't have a search expert or a DevOps infrastructure expert.

Are there APIs that you insist on using when you build on the Jamstack, or are you more open-ended?

I think we're pretty open-ended. We don't really use that many external infrastructure APIs. The thing we do use is transcription. I love AssemblyAI, just to toot their horn. I don't want to manage keeping a machine learning model up to date, knowing what the best practices are around rendering certain pronouns or numbers or whatnot, or keeping a dictionary of company names and handling that in different languages and accents. If I can pay something to do that, then, boom, I'm going to do that. And if it's cheap and good and dependable and fast, that’s even more reason why.

Video coding -- Mux is a great example. I think as a product offering, they want to have broader appeal for, say, user-generated content companies. Their money comes from enterprise video and enterprise organizations, so our use case maybe doesn't fit their business model, but they want to expand in that direction. As a result of wanting to expand, their product offers opportunities that work within our space.

There are things that are easier to not have to set up, but then, if you avoid setting them up, at some point you'll have to set them up because the cost will get too expensive. And there are other things where you can avoid ever setting them up. You can always pay somebody forever, and that's going to be okay. Because you’re in the sweet spot in where you fit into the way that the external service makes their money, you can get a lot of value and not hit their limits or whatever.

AWS is basically the only thing we have to use: GCP, AWS, Heroku, Vercel. But for external API services, I can't think of anything in particular.

Disclaimers