EPISODE 950

[INTRODUCTION]

[0:00:00.3] JM: Data engineering is difficult. Companies want to be able to maximize the value that they get from their large data sets, but there are so many steps required for deriving that value that most companies feel they are always far behind the ideal state of where they could be.

The cloud makes it cheap to save data. Tools like Spark and Snowflake give us usable APIs for simplifying the large-scale data processing. Workflow engines like Airflow help us visualize a complex execution path of a data pipeline. With all of this tooling, why is it so hard for us to make use of our data?

Nick Schrock is the creator of Dagster, an open source system for building modern data applications. Nick is also the CEO of Elementl, a company that he is building around Dagster. Before creating Dagster, Nick worked at Facebook where he co-created GraphQL. Nick returns to the show to discuss modern data engineering and why it continues to be so difficult for engineers to be productive with their data.

For anyone who is trying to understand the space of data engineering and feels intimidated and just feels completely confused by why data engineering is as difficult and seemingly unproductive as it is, this is a great episode. Nick is quite good at simplifying the strange complexities of the data engineering ecosystem. I hope you enjoy it.

Full disclosure, I am an investor in Nick's company, Elementl.

[SPONSOR MESSAGE]

[0:01:48.9] JM: This podcast is brought to you by PagerDuty. You've probably heard of PagerDuty. Teams trust PagerDuty to help them deliver high-quality digital experiences to their customers. With PagerDuty, teams spend less time reacting to incidents and more time building software.

Over 12,000 businesses rely on PagerDuty to identify issues and opportunities in real-time and bring together the right people to fix problems faster and prevent those problems from happening again. PagerDuty helps your company's digital operations run more smoothly. PagerDuty helps you intelligently pinpoint issues, like outages, as well as capitalize on opportunities, empowering teams to take the right real-time action.

To see how companies like GE, Vodafone, Box and American Eagle rely on PagerDuty to continuously improve their digital operations, visit pagerduty.com. I'm really happy to have PagerDuty as a sponsor. I first heard about them on a podcast, probably more than five years ago. It's quite satisfying to have them on Software Engineering Daily as a sponsor. I've been hearing about their product for many years and I hope you check it out at pagerduty.com.

[INTERVIEW]

[0:03:15.9] JM: Nick Schrock, welcome back to Software Engineering Daily.

[0:03:18.2] NS: Thanks, Jeff. Great to be here.

[0:03:20.1] JM: There is a term that gets frequently used these days, called data platform. I hear it being used so many times in so many different contexts. What is a data platform?

[0:03:33.6] NS: Well, that's interesting, Jeff, because I think it's a term in popular use, because it's something that everyone wish exists, but doesn't exist yet. What do people actually want? How I think about this is thinking about all the different data that enterprises have to deal with. It is classified into a few different ones. One is data that is inputted by users and manipulated by a traditional application. The other data is data that is ingested from the outside, either from outside entities, or even sensors.

Then the reason why this is so difficult is that most of the data in an enterprise that has to be managed is moved around and processed and copied, meaning that you're copying data from the application into something. I think the term du jour is a data lake or a data warehouse these days. Or it's ingested and copied from an external entity, or taken from sensors. Then those are

processed by a wide variety of tools, different computational runtimes, different storage formats and there's no either vertically integrated solution that manages all that data that gets computed within an organization, and the vertical integration doesn't exist, nor does some horizontal integration exist that ties all the systems together. Again, I think to summarize, I think the data platform in the way people want it to be does not exist yet, but everyone instinctively recognizes the need.

[0:05:17.2] JM: When a company gets started, it's just a database. Many of the listeners are probably when they think about applications that they've built, they just think, “Okay, I've got a website and I've got a database.” I remember as I was starting to do this podcast, I started to learn about the fact that bigger companies have more databases, more data platforms, more data processing systems.

I want to paint a picture for people who are less familiar for how companies get to a place where they need multiple data systems and multiple data processing techniques. What is the journey that a company takes from just having a single node database to needing all these different systems?

[0:06:04.3] NS: There's many dimensions to this. First of all, I think that most companies today are not those single database companies. Meaning, that either through the constraints of their domain or through self-inflicted technical decisions do not have a single database. This transitions into one of my favorite topics, which is microservices.

The moment that you decide to adopt a microservices architecture, you typically now have more than one database. Meaning that you cannot do an analytics query against those two databases without introducing a third system. The moment that you introduce a microservice, which is very popular these days, you require more data management. At minimum, you need to copy that thing just to a place, likely a data warehouse where you can join those two datasets in a coherent way. Or you use a tool, like Presto, which can connect to those two databases and then provide an analytics engine over that.

Regardless if it’s Presto or Snowflake or something else, you need some system that can stitch those things together. That's what I'll call the self-inflicted technical reason why people need to

do it. Let's imagine you have a single database, right? The other vector that this happens is you maybe start out in your newly analytics queries against your production database. The moment that you bring down your own users based on your analyst showing up at 9 in the morning and slamming your production database, you decide to change your architecture. Then you typically introduce read replicas, that some point that doesn't scale. Then you end up replicating that data to a data warehouse, or data lake, where typically using a column store in order to cheat performance.

The other way this happens is what I'll say is your domain requires it. Meaning that yes, you have your application, but you are also integrating your application data with data from some external source, either you're ingesting it from a SaaS service, you're scraping data from somewhere else, you're ingesting data from a third-party vendor of another sort, or you're ingesting sensor data that is outside the scope of your app.

The moment that you have more than one data source and one additional data source beyond your application database, you generally need another system to at minimum, store and likely process that data. In today's world, I think most systems grow beyond a single database fronted by a single application, actually increasingly early in their lifecycle.

[0:08:55.8] JM: I'll give you one other thing. Elasticsearch is super common, right? You need to have this search-centered database that is essentially, often a copy of your core data, but you have to have some system for copying over your core database into the Elasticsearch database, which is not exactly the same thing as the analytics tasks you're talking about, but it's often done within the same purview, right? This is still the data platform that we're talking about and this furthers the confusion.

[0:09:30.2] NS: Yeah. That's a great point, Jeff. I need to add that to my FAQ sheet when I'm explaining those to people.

[0:09:35.6] JM: The commonality between all these different things is that data is going from one place to another. You have, maybe it starts with the core database, that data moves, it gets copied into an Elasticsearch, or it gets copied into a Spark cluster, or it gets copied into this or that. From there, it might get copied into somewhere else.

This is the framing around which you see the emergent data platform, the directed acyclic graph of data moving from one place to another. Can you shed some more light on that, that perspective that you take?

[0:10:11.7] NS: Yeah. It was this pattern I saw as I was discovering this domain. If you jump from – What I worked on before was traditional applications. I like to say I was present at the creation of the full hipster stack. I personally was one of the creators of GraphQL. I wrote the first version of that, the first prototype. Then I sat next to – I didn't have anything directly to do with it, but I sat next to the people who built React. Those systems are very principled and have fairly great developer aesthetics and whatnot. They jumped in this data engineering world and it's just very different.

The thing that I noted is that – and a lot of this comes from Chris Burgh's data ops way of thinking is that this is much more like a manufacturing process, where data gets moved, transformed and you don't have control over your ingest. In a traditional application, if the user enters inappropriate data, you simply make their form read and then you force them to re-enter the data, so it's correct. You do not have that option in data ingest systems.

Whatever the other systems give you, you have to deal with, which means this injects a ton of complexity that is not obvious into your software systems. As a result, there's a lot of software complexity in these pipelines, or data production systems, or what you like to call data applications. Programming feels different. Then I noted that most of the data in the enterprise was actually derivative data produced by these processes.

By most data, often the raw ingest dominates in terms of raw storage. I mean, in terms of data complexity and data surface area. If your metric was instead the number of unique columns and data sets you needed to track, it is dominated by data that is produced from other data. The commonality between all of that stuff is every data asset that is produced within an org is typically produced by a single computation running at a regular interval over a long period of time.

There's this one-to-one correlation between a unit of computation of the data asset that produces. Because that, I thought that effectively the DAG is the DAG of functions. I use the term ‘graphs’, because I think there are circular dependencies within ML systems in particular. That is the core abstraction that can unify all of these disparate systems, because whether it's a Spark job, or a data warehouse job, in the end they are just functions that consumed something and produce something. It's actually, there's a missing core primitive in my view.

[0:13:12.6] JM: Who are the different constituencies that are operating over this system?

[0:13:19.3] NS: You asked what the major constituencies in these systems are and this leads to another thing that I noted about these data applications, is that there's a wide range of constituencies. There's data engineers, data scientists, there are analysts, there's the infrastructure engineers that support those people and there's also this missing constituency that people don't talk about today, but who are directly involved in that and that's the actual application developers, who in the end are one of the key sources of data that the organization has control over.

What's fascinating about this is that each of those constituencies is accustomed to using very different tooling to actually build their part of the process. In the end, the end-widget that they're producing does the same thing. Like a data scientist writes a notebook, which is some processing. You can view that as just a function, a coarse-grain function. The data engineer is using, say Spark written in Scala. Again, they're just writing a job, which consumes a file and produces a file or data warehouse table, etc. You can so on and so forth.

I think what everyone notes in this system is that it feels like the different constituencies in these data applications are far too siloed. They exist in totally different worlds. As the data flow from one constituency to another, a ton of context is lost and they don't have a fundamental base of infrastructure on which they can operate in a truly unified way. Those are the major constituencies that I see in this domain.

[0:15:08.4] JM: You and I first talked about what you were doing with Dagster about, I think it was 18 months or two years ago, something like that.

[0:15:17.7] NS: Probably a year and a half. Yeah.

[0:15:18.7] JM: A year and a half, something like that. At first, I didn't really understand what the perspective you were taking was. Part of the reason for that is because you were coming from a very different area than the data engineering conversations I had been having, because most of the data engineering people that I had been talking to had either been working at research for a long time, like the Spark people, or they had been working on Hadoop infrastructure for a really long time. Because you were coming from a different background, the front-end, middleware infrastructure layer, it felt somewhat foreign.

Since then, the way that you explain what you're working on has converged with the more traditional way that these data engineering people talk and the problems that they're confronting. You explore this a little bit in this talk that you gave at Data Council, which was awesome.

One of the things you noted was there are things that people don't talk about in the data engineering world. Whenever people are not talking about something that everybody is experiencing, that is the most important thing in the room. What are these problems that you have observed with the world of data engineering? What are these acute problems that you think people are perhaps under-focusing on?

[0:16:44.9] NS: I think when you come from an adjacent domain, you have that beginner's mindset and ask why is it like this? Not be satisfied with the answer of like, “Well, this is what we've always done. This is what we do.” Well, let's talk about that. This also comes from the experience of Facebook of having that mentality of honoring best practices, but not being captive or hostage to them.

With that being said, there were a few obvious things as I started to talk to people and observe the way they work, is that the programming that they did felt, for lack of a better term, at a lower level of abstraction and with less structure than I felt was appropriate. Meaning, the way they were approaching the software didn't match the complexity and subtleties of the domain. This is most directly expressed in that testing is not a norm in these systems, which is crazy. It's totally

crazy, given how mission-critical and how difficult these things are to build. I believe that's mostly a software abstraction problem.

The other thing is this mantra that you hear over and over and over again, which is I spend 90% of my time data cleaning and 10% of my time doing my job. If you just hear that a couple times then you start thinking, “Well, that's probably the most important problem to work on.” Yeah, so it was just the type of thing where you enter an adjacent domain with a certain amount of experience and a mentality that you developed over there and then you go in and start asking the dumb questions that are staring you in the face.

[0:18:46.1] JM: Why does framing data engineering as a directed acyclic graph help alleviate data cleaning?

[0:18:55.3] NS: It's because the term data cleaning is inaccurate. The way I like to explain that, or think about it is that people say like, “I spend 80% of my time data cleaning and 20% of my time doing my job, or 10% of my time doing my job.” There's one of two possibilities there. One is that they have a misconception of what their job is and maybe the 80% of the time do that is their actual job.

The other way to frame that is that the term data cleaning is accurate. It's a misnomer, that means that I spend 80% of my time doing stuff that is not my job. One of which is data cleaning, but it's actually a far more comprehensive set of issues. One is that people frequently have to roll their own infrastructure. Very typically, you hire data scientist, you have some CSV files that live somewhere, you give them a Jupiter notebook and they're expected to re-derive the meaning of that data from scratch, roll their own infrastructure, figure out how to productionize it and maybe it was limited help from an infrastructure engineer.

They don’t have a software mentality that allows them to control the complexity of the software that need built. All of that I believe gets roped into the term ‘data cleaning’. How Dagster we hope has impact on that is that it's a way to structure your applications, your data applications in a way that brings the best of traditional software engineering and then adapts it to this target domain and presents it in a way that aligns with the way that people think.

I think it's hard to directly tie just purely the concept of a DAG, because lots of systems have DAGs, right? Airflow has a DAG concept and Luigi does and Argo. All these workflow engines do. It's the other abstractions that we're building into the system. Having the notion of having a standard around a unit of computation that's accessible via the API that I think actually is going to have impact in this sector.

[0:21:16.6] JM: Let's talk through some of the terminology of Dagster. When I'm getting started with Dagster, there are some pieces of vocabulary that I need to know. I need to know – there's a term called a pipeline, there's a term called a solid. Describe some of the basic vocabulary primitives of Dagster.

[0:21:40.2] NS: Yeah. We wanted to introduce – I think the most – a pipeline is a DAG of these solids, a directed acyclic graph. I think the most interesting thing to dig into is this notion of solid, because this is the neologism in the system, meaning the new term that doesn't exist in other systems. Other systems use the term task, or operator, in order to describe these things. We felt solid was something different.

One is that for example, Airflow, adjacent system that we have an interesting relationship with, which we can get into later. Airflow is a notion of this operator concept, which doesn't have any notion, it doesn't self-describe what data comes into it or what data comes out to it. It's purely a unit of execution. Then once a DAG is constructed in Airflow, it by design has no parameters. They're essentially functions which take no inputs and have no outputs and are therefore, structurally untestable.

Solids have a bunch of unique properties that we feel makes it a different abstraction that is a higher level abstraction. Meaning, it self-describes. At its core, it's a function. It's a unit of computation. When you write a solid, it's a decorated Python function, where the user can define what actually happens in that function. It can shell out to instigate a Spark job. It can operate on pandas internally. They can do whatever it wants. What it has is it defines inputs, it defines outputs. Therefore, it is a parameterizable thing that you can execute with different parameters and therefore test it.

It's designed to have this notion of context that flows through it, that allows you a layer of indirection from which to isolate your business logic from your surrounding environment. It has a self-describing config system. In the end, the way the system works is that we actually compile that graph of solids, combining it with config is an input to generate a derived DAG that looks much more like an Airflow DAG; one without parameters. It's a machine generator artifact that is then directly executable. We feel it's a higher level of abstraction that is distinct from its progenitors. Therefore, we felt that introducing a new term was appropriate.

[0:24:30.5] JM: The adoption for this tool is very tricky. I want to understand the state in which you imagine a company adopting Dagster and the process by which they adopt it.

[0:24:50.0] NS: Yeah. I think there's a few different use cases here. One is a company, let's say that you are just getting a project off the ground and you have greenfield and you're starting from the blank page. Therefore, you're in the position to think about what infrastructure you want to use from first principles. That's one cohort of user that we're very interested in right now, because they have a lot of flexibility and they aren't beholden to existing systems and whatnot.

This system also is designed to be incrementally adoptable. That definitely stems from the experience of both GraphQL and React and also the internal projects that we worked on at Facebook. The system is designed very explicitly to integrate with a ton of other systems, both existing computational runtimes, like a data warehouse and Spark, etc., etc., etc. Also existing workflow engines.

Because as I said before, because the solid we believe is a fundamentally higher level abstraction than say an Airflow operator, or task graph, that we can actually compile – you can actually use Dagster, the software abstraction within the context of Airflow, by effectively calling out a function that takes a set of solids as an argument and some config and then it compiles an Airflow DAG.

A user that wants to use Dagster within the context of an existing airflow cluster can begin to experiment with it without having to change any of the existing infrastructure. Because of the peer software abstraction and it's what I like to call horizontal opinionated platform, rather than a

vertically integrated one, it's enormously flexible in terms of how you can integrate it into your system. Those are the two modalities that I think about is greenfield adoption.

I would introduce a third, which is a greenfield team that has the opportunity to build new infrastructure in the context of an existing company. That's like, you spin up a new a new team with fresh blood and they have carte blanche to operate independently. Then there is people who are maintaining existing infrastructure. There's a spectrum there.

[0:27:24.1] JM: Yeah, that third team seems really relevant, because I think about two canonical shows that stand out when I think about data platform, are one that I did with Uber and one that I did with Lyft. Just because these companies have so much data coming in and there's so much stuff they could build on top of it. They really wanted to build this rich – I mean, I think these are companies that actually do have data platforms. This fulfills the requirements of the data platform. It does not fulfill may be the ideal UX requirements. Maybe that's what you're doing.

I can totally imagine some new team saying, "Look, we're going to build Lyft Eats, right? We need data from the Lyft driver platform, because we need to figure out who are going to be the best drivers for the Lyft Eats platform, because we want to start doing food delivery." You can totally imagine a new data engineering and data science team standing up within Lyft and saying, "We want to reimagine the kinds of data operations that we're going to need out of this data platform and we're going to start from scratch with Dagster." Do you think that's the third category that you're talking about?

[0:28:35.6] NS: The greenfield embedded within –

[0:28:37.7] JM: Greenfield embedded within a large company as a data platform.

[0:28:41.3] NS: Yeah. That's exactly what I'm talking about. Those companies, they have what I'll call data platforms sort of. Obviously, they have a ton of incredibly talented people working on this and they perform extraordinary feats of infrastructure engineering. If you talk to people who work at Lyft, or especially Uber, or any of these companies, the internal data platform is

incredibly fractured. There are lots of good tools within every vertical. I find the tools, the span every vertical to be relatively limited, even in those mature orgs.

Putting that point aside, yeah, I think that's exactly the type of use case that is totally amenable to that greenfield within an existing org project. It works, because Dagster is totally agnostic to computational runtime, like I said. We consider things like Spark, or a Jupiter notebook, or data warehouse job as the body of a solid, which from the standpoint of the system is totally black box.

Therefore, you can reuse existing code and existing systems that are built on those technologies. You also have the opportunity to introduce a new layer that's interesting, where you don't have the requirement of integrating with a bunch of existing infrastructure on a different dimension. It simplifies the problem in a way. Then those teams typically are empowered to make their own technology choices, given the culture of those engineering organizations.

[SPONSOR MESSAGE]

[0:30:56.3] JM: Monday.com is a team management platform that brings all of your work, external tools and communications into one place, making cross-team collaboration easy. You can try monday.com and get a 14-day trial by going to monday.com/sedaily. If you decide to become a customer, you will get 10% off by using coupon code SEDAILY.

What I love most about monday.com is how fast it is. Many project management tools are hard to use, because they take so long to respond. When you're engaging with project management and communication software, you need it to be fast, you need it to be responsive and you need the UI to be intuitive. Monday.com has a modern interface that's beautiful to look at. There are lots of ways to use Monday, but it doesn't feel overly opinionated. It's flexible, can adapt to whatever application you need, dashboards, communication, Kanban boards, issue tracking.

If you're ready to change the way that you work online, give monday.com a try by going to monday.com/sedaily and get a free 14-day trial. You will also get 10% off if you use the discount code SEDAILY. Monday.com received a webby award for productivity app of the year and that's

because many teams have used monday.com to become productive; companies like WeWork and Phillips and wix.com.

Try out monday.com today by going to monday.com/sedaily. Thank you to monday.com for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[0:32:45.6] JM: As I understand the developer experience for using Dagster is I go into my code and I find these operations that are doing what is commonly called data cleaning today, but might any data transformation. You decorate it with metadata that says this is a solid.

After decorating it as a solid, the developer now has the experience of being able to see that solid in a visual interface that you've created called dagit. There is essentially a UI experience, almost like an IDE for defining your data workflows. Can you just talk a little bit more about the developer experience when I am using Dagster and I'm – I want to create some data application on top of my data infrastructure, what am I doing?

[0:33:48.9] NS: Yeah. I think you got it right for phase one, right? Which is taking existing computations, doing very few changes to them, annotating them with some metadata that allows you to express what those computations are actually doing. We have a type system, we have a metadata system. As you noted, those who can be visually rendered within dagit and then also monitored.

I think where the system starts to deliver more value is where you actually start to mold your computations to be in-line with the Dagster API, so you make your existing code a little more so called, Dagster native. Once you do that, it unlocks a lot of potential. One of the things is that we designed the system to be very pluggable. Meaning, that when you have a workflow system you typically have to track, or we track what runs have been running, so there's runs database somewhere. Also, we also store intermediates that flow between the solids in order to enable incremental computation and being able to re-execute tasks without re-executing an entire DAG and other features as well.

Because we designed it to be very pluggable, that means we can run it in a mode where you don't depend on any externalized infrastructure to run the pipeline and that makes it so it's far more testable. You can run it in an isolated environment, either on your laptop, or in a CICD pipeline or in your production environment without changing your business logic. This is critical for having layers of testing.

If you talk, for example, to someone who's running an Airflow cluster, the notion of running unit tests is very, very difficult. The notion of running Airflow in a local way where you can execute, say subsets of the DAG, you cannot parameterize it. How could you run it on a test dataset, instead of the production dataset? It's actually quite difficult. People could do it, because software is infinitely flexible. Most existing workflow engines, testability is not a pit of success, right? You have to fight the system in order to achieve that objective.

Whereas, we really try to make it so that if you use the system in the way that we try to guide the user, you get testability and a local development experience all for free. Our IDE environment, I'm actually about to press publish on a Medium article today that announces our latest version of Dagster 0.60 that we've codenamed Impossible Princess. That will also – that turns dagit – you can use it both as local development environment, as well as a production monitoring tool.

It's essentially the same code. The reason that enables that is pluggable API-driven system. You get testability, because our core infrastructure is pluggable. If you buy into what we call using resources and using this context abstraction we have to isolate your business logic environment, or the environment that you need rather than the environment that we provide, because a lot of the environmental concerns when people write these data pipelines are specific to their domain or context, then you can start thinking about isolating your specific infrastructure from your business logic code.

We try to enable these abstractions in order to make it, so that you can really move more of your testing farther up the value chain. Meaning that more of your testing can occur at unit testing time, more of your testing can occur at integration time. Then a smaller subset of that testing has to happen in your production environment. I think that's a big value here.

In general, it also structures your code better in a way that's more self-describing. A lot of our users, it's difficult for them to articulate, but they've been able to reuse code that other builders, I'll use more generically within the system, have written. Because you are guided to build these solids, which are coarse-grained use of computation, which are self-describing, which are more straightforward to reuse in other contexts. I think that's another advantage of the system. Yeah, I'll start out with those two.

[0:38:49.4] JM: Let's talk about Airflow. Because Airflow is a very popular tool that is used to define DAGs commonly used for data engineering. I know you've talked to Maxime a fair amount about data engineering; Maxime the creator of Airflow. What problems did Airflow solve when it came out? What was the white space that remained unsolved after Airflow came out?

[0:39:19.5] NS: The core problem that Airflow solved was being able to express a dependency graph between tasks. Prior to Airflow or similar workflow engines, the people were just running unrelated cron jobs and not expressing their dependency, just dependencies between those systems.

Yeah, you can imagine how this goes wrong. You have one task that runs at midnight every day. Then you know that another test depends on it and the best you can do is run that every day at 4:00 a.m. If the task takes longer than four hours, everything gets screwed up. If that previous task fails and you don't restart it in time –

[0:40:08.6] JM: By the way, where are these things tracked? Like an Excel spreadsheet or something?

[0:40:12.4] NS: I mean, God knows. Maybe if you're lucky. That was the previous state of a lot of these systems. I think the other thing that happened is that systems that did – Yeah, I'll bucket Luigi in with Airflow as well. I think that they were trying to solve similar problems. I think you we can debate why Airflow became dominant versus Luigi, but either system could have one, but they were both attempting to do the same thing. I mean, it allowed you to express dependencies. I think it allowed users in a fairly low barrier of entry to express their DAGs in terms of code.

[0:40:54.1] JM: Data dependency, saying –

[0:40:55.1] NS: No, no. You can't do data dependencies in Airflow. It's purely – all you can define is the order of execution.

[0:41:02.4] JM: Well, but that's what I mean. Sorry, maybe I used the terminology wrong.

[0:41:05.6] NS: I shouldn't have interrupted. I apologize.

[0:41:06.7] JM: No, no. I'd interrupted you. You say for example, I need this data cleaned and refreshed before I operate over it with the PageRank job.

[0:41:19.7] NS: Exactly.

[0:41:21.1] JM: I need this to finish. If a server goes down two or three times and you need to restart the job two or three times, it's really important that the PageRank operation, the actual transformation that you're going to be building a system around takes into account the extra time that the previous operation is going to take. That is one of the core problems that Airflow solved is that dependency.

[0:41:46.2] NS: That's right.

[0:41:47.5] JM: Okay. Also, my sense is it gave people a place to look, to actually get a high-level bird's-eye view for what the heck is going on here.

[0:41:56.1] NS: Yes.

[0:41:57.6] JM: What didn't it solve?

[0:41:59.8] NS: What I think has happened is generally the complexity of these data applications has outstripped the ability for Airflow to meaningfully model them. If you look at a big company, if you go into say a Stripe, or Thumbtack, or Lyft, or whomever, the DAGs are often hundreds or thousands of nodes. At which point, it's complicated enough that execution

order is not sufficient in order to describe what's happening and have it be understandable by a human.

There's also really practical concerns, like the UI is no longer render. There is in unit of composition within Airflow, so you can subdivide your DAG into sub-DAGs. It's literally called sub-DAGs. Because of the way it's implemented in Airflow, if you talk to most Airflow shops, they actually disallow features – talk to a sophisticated Airflow shop, they typically blacklist or disallow sub-DAG sensors and these other features, because they're not reliable enough.

Super high-level, what I would describe is that Airflow is about how something executes. It doesn't describe what it's doing, beyond simply execution dependencies and a label on a task. There's no notion of data flow. There's no notion of self-describing configuration. You therefore can't readily test it without building a bunch of custom stuff. Typically, most Airflow shops build a relatively sophisticated amount of custom software abstractions above it to model that.

I think Airflow solved the dependency problem. Did it in a way that was code-driven. Therefore, it was open-source. It allowed you therefore to integrate with a wide variety of tools, rather than having to bet on a single vertically integrated ETL system, like Informatica or similar. It solved those problems. I believe that in order to correctly model the more complicated data applications that exist today, you just need more complexity.

Airflow is also most specialized for data engineering workflows, rather than data science workflows. Data science workflows, you typically – you really need to parameterize things, because you need to say, run experiments, right? You need to launch 50 versions of a job in order to do say, hyper parameter optimization, or to figure out what's the best way to do your model more generally.

If you model your DAGs in Airflow, without having a higher level of abstraction on top of it, that's actually impossible to do, because you can't take an existing task in Airflow and launch them ad hoc. You cannot parameterize them differently. You can always build these custom systems that work around the abstraction, but doesn't guide you to do so. I think it left a lot of stuff on the table there in terms of things that you need to express.

[0:45:29.5] JM: You used the term ‘self-describing configuration’. Can you explain what that is?

[0:45:33.7] NS: Yeah. Meaning that typically, these data pipelines are in the end extraordinarily complicated functions. They generally take in – if you go to a typical shop that has data engineers and data scientists, they often have these massive piles of undocumented JSON, or YAML parameterize or configure these computations.

[0:46:00.9] JM: Oh, good lord. People actually edit these files.

[0:46:05.6] NS: Oh, yeah, yeah. Of course. They're often write once and then use. Our view is that if you build more sophisticated software that allows you to verify the configuration is correct before you execute, that gives you type-aheads, gives you built-in descriptions. One, the existing configuration you have is much easier to manage, but then you're not afraid to make these computations more flexible, therefore, more reusable in more contexts and more testable.

In my view, a lot of the reasons why a lot of these systems end up not having reusable chunks of code that aren't testable is because the configuration is very complex. Well, that's one of the reasons. We believe that that's a fundamental capability. Really concretely, right?

What this ends up looking like is that if you open up our dagit tool and you are looking at pipeline and then you want to execute it and figure out what's going on, you have this execution console where you start typing into it and then things magically pop up, and it's a type-ahead and there's imbedded descriptions and all this. It's this magical experience, rather than just writing some undocumented JSON.

Then God forbid, you hand over that undocumented JSON to another developer to try to reuse, it's very difficult. Where by having this common configuration system, it's much easier for another builder to reuse a computation written by someone else.

[0:47:34.4] JM: You're describing a much richer development environment than people write their data applications in today. This is centered around dagit, which is this interactive development environment for your data that does take into account what you're annotating in

your code. Can you just focus on dagit and explain what this visual IDE-like experience is for I guess, a data engineer? That's the main person who would be using dagit, right?

[0:48:12.8] NS: Data engineer. We believe it's a generalized tool that will be used by data engineers, data scientists who –

[0:48:22.7] JM: Or even engineers, the person who's setting up the Elasticsearch cluster.

[0:48:25.4] NS: Yeah. Anyone who's participating and building one of these data applications, well use it as an authoring tool. Then again, after I press publish today on this Medium article, we hope that more and more people will use it as an operational tool as well. Therefore, you have this common substrate where both ops people and data application builders will be using a common tool set and a common API to understand what's going on these systems.

[0:49:00.1] JM: Describe the parallels between the problems of the data platform experienced today, as they are largely experienced in well, by any developer. Describe the parallels between that and the problems that we had with web development 10 years ago.

[0:49:17.0] NS: Ah, this is one of my favorite subjects. As I was exploring the state engineering data application space, again as I mentioned previously an interview, kept on hearing this 80/20 thing where people felt that they were effectively wasting the vast majority 80%, 90% of their time. I was searching for in my mind a historical analogy to think about what was another time and place where people felt they were wasting all of their time on incidental complexity, rather than the essential complexity of the task that they were trying to perform.

The analogy was actually staring me in the face. It was exactly what I had been working on for a lot of my career. That's this application development. Rewind back to 2010 and 2009. If you were in screaming distance of a front-end engineer, you would hear them raging about IE6 and IE7 and browser incompatibilities. they would say I spend 10%, 20% of my time building my app and 80, all my time fighting the browser. you'd hear stuff like that.

That didn't just mean that people were wasting their time. It meant that other engineers and other domains didn't want to engage with that. They would hop in to a web browser and make

some toast pop-up or something fly around the screen and then get the hell out of there and run away as fast as they could. Then if you fast-forward to today, the world is transformed. In 2019, you have a very active front-end community that's – to put it lightly, very passionate, especially on Twitter. No one says that they waste all their time fighting the browser. That's not a phrase you hear any more.

People complain about so-called JavaScript fatigue. That's basically like, there's too many tools and they're hard to integrate. I think the ecosystem is working on that. It's more of this general spiritual feeling that one, this is a real engineering discipline that respects itself. People are using abstractions, like Angular, Vue, but React is really dominant that respects the problem. If you sit down to write a web app today and you fully bet on these abstractions using React, using GraphQL, it's magical. It's totally magical. It's a joy to work in. There's this feeling that the ecosystem really is making progress in lockstep and there's real – the world is getting better.

What was interesting is what happened in those interim is yes, the browsers did get better. That's undeniably true. Chrome switching to continuously shipping stuff, for example was a huge leap forward. What I would argue is that the important thing that happened is that the software abstractions got better. Namely, that instead of coding against the Dom, which was good, betrays it in the name. It was the document object model. It was not an application platform. You needed a fundamentally new abstraction to build, or set of abstractions to build front-end applications. It's really a software abstraction problem at its core.

Once you've got the software abstractions mostly right, the entire ecosystem can make progress and it could change the qualitative aspect of this. I really view the state of data engineering, data science and generally the building of these data applications, which by the way for – another new term we – think of it as ETL pipelines, or ML processes, or things of that nature.

[0:53:15.3] JM: Data applications.

[0:53:16.2] NS: Data applications. Right. It feels like front-end is in 2010. There's this general sense that everyone feels like there's fighting this incidental complexity all the time, there's not an ecosystem really where people are building on top of themselves. There's a ton of replicated

infrastructure and work across these ecosystem systematically. I think it's fundamentally a software a distraction problem.

[0:53:45.9] JM: Just to validate that a little bit further, my younger brother who I think you met at the Open Core Summit.

[0:53:52.0] NS: I did.

[0:53:52.9] JM: He is the prototypical hipster developer, right? First of all, he has had no problem being very productive with GraphQL and React. Now he's maturing a little bit, he's getting into data infrastructure, he's very curious, he's data curious.

[0:54:13.7] NS: That's an amazing phrase. Congratulations.

[0:54:16.0] JM: No. Being data curious, it's a tough time to be data curious. If you're a solo developer in a coffee shop like him and many other hipster developers, you don't know where to begin. You would love to get involved with machine learning and data science and big data. We can talk about the problems of access to data sets. That's certainly an issue as well. You cannot really be a solo hacker. Think about indie hacker. I'm obsessed with indie hacker businesses. I think they're really interesting.

These one two person companies that manage to build 2 to 10 million dollar businesses with lots of customers. They are way over-stretched, but they've got a very successful SaaS business and they're just busy with their operational SaaS business.

If they even wanted to get involved with the data platform, they would have no idea where to begin. ETL jobs are just misery. They would not even touch it. It's exactly the same thing that you're saying. There is this huge untapped potential for data applications that is not realized yet. That's what I think it's pretty interesting about what you're doing.

[0:55:25.3] NS: Yeah. I couldn't agree more. One thing I'll point out is if someone looks at Dagster today, they might say, “Okay, I hear what you're saying and that vision sounds great. How on earth does what I see now as the open source artifact enable that?” I think that

feedback is right. This is the kernel of something that we're going to build. Just to harken back to the React analogy, in the early days when Jordan was – Jordan Walke was starting to work on React and then Pete and Tom – Anyway, I wouldn't certain drop names, but the early Facebook developers –

[0:56:06.1] JM: Already did.

[0:56:07.6] NS: The early Facebook developers who were using on it, it looked like a – it was a clever abstraction, this whole notion of a virtual DOM and then the really controversial thing of JSX, which was integrating markup into your business logic, which is a whole different subject. It was very difficult to see in 2010 how that would become the kernel of the thing, which would really unleash this entire ecosystem. I just want to be clear that I think this abstraction has a lot of promise, but we're still in early days. We have a lot of work to realize this vision that we're working on.

[SPONSOR MESSAGE]

[0:56:58.1] JM: Today's show is sponsored by Datadog, a scalable, full-stack monitoring platform. Datadog synthetic API tests help you detect and debug user-facing issues in critical endpoints and applications. Build and deploy self-maintaining browser tests to simulate user journeys from global locations.

If a test fails, get more context by inspecting a waterfall of visualization, or pivoting to related sources of data for troubleshooting. Plus, Datadog's browser tests automatically update to reflect changes in your UI, so you can spend less time fixing tests and more time building features. You can proactively monitor user experiences today with a free 14-day trial of Datadog and you will get a free t-shirt.

Go to softwareengineeringdaily.com/datadog to get that free t-shirt and try out Datadog's monitoring solutions today.

[INTERVIEW CONTINUED]

[0:58:05.5] JM: There was a lot of engineering stuff we didn't exactly cover, that I think people could – I would really encourage people to check out your data council talk. I know you're a little bit unsure about it, but I thought it was awesome. I thought it's really good. Quite entertaining. Just to wrap up – oh, we also – we didn't even talk about Jupiter notebooks, but that's another thing people can check out.

[0:58:25.1] NS: Well, I got time. We can talk about it.

[0:58:26.3] JM: Okay, sure. All right, let's talk about Jupiter notebooks a little bit, then we'll talk a little bit about business. Why are Jupiter notebooks relevant?

[0:58:33.6] NS: Whether you like Jupiter notebooks or not, the reason why they're relevant is they are the way that data scientists do their work. There must be something interesting going on there. That's why they're interesting. I think I went through – I'm like a traditional grumpy – well, I'm not grumpy, but I'm a traditional engineer.

When I saw Jupiter notebooks, I was both amazed and appalled by them. Having this in-line visualization, in-line with code was pretty amazing. It's a crazy programming model. It's very difficult to create reliable computations in Jupiter notebooks, because when you author them, they're very tied to their external environment, they allow for out-of-order execution, they don't constrain you that much.

Typically if you find a notebook and let's say you open a project that you don't know about and there's a folder of notebooks, it's likely to be this black hole of code that's extremely tied to a data scientist environment. It would be very difficult to reuse, or even figure out what's going on, unless they've been incredibly thoughtful and deliberate about doing that.

Jupiter notebooks are relevant though, because this is where data scientists do their work. Then what happens is that they have to then translate that work into a meaningful software artifact that can be productionized, if you want to say, plug that into a machine learning system. Typically, that means, or often that means that you effectively hand the notebook to a data engineer who's then responsible for productionizing it. That's actually a fairly pernicious

relationship between two personas in an organization, because the data scientists don't know how to productionize it. They don't understand what's happening in production.

The data engineers don't actually really understand what the models are doing. Then you have this who's responsible for what. It's how they're – similar dynamic to how there used to be dedicated developers and then dedicated test engineers. You would throw the software over to the test engineers and it causes a high latency feedback loop between the two of them. Most large high-functioning software organizations have ditched that structure, but it's replicated in a slightly different way between data scientists and data engineers. I believe a lot of it is because of notebooks.

Now, notebooks are amazing because they enable – especially in Python, to have code, have a dramatically wider dynamic range. Meaning that you can take someone who isn't necessarily a professional software developer, or even as maybe all they can do is they figured out how to do a lot of damage in Excel. Excel is programming, if you're building a financial model. It's actually probably the most broadly used functional reactive programming platform ever created. Or they have they been able to figure out or learn SQL.

That persona of engineer or builder, I like to call, because they're not all engineers, you can pop them in a Jupiter notebook and teach them enough to do very interesting things. They're incentivized to learn, because they get this immediate feedback. They can graph stuff.

If you compare that to your intro-programming project at a college or something, a Jupiter notebook is way more fun. It's a really interesting problem space. I think there's a project out of Netflix called Papermill, which I think is incredibly promising, which effectively allows a notebook, you can parameterize one of the cells and then turn it into a coarse-grain function. There are other platforms that do this. Databricks has hosted notebooks, which have a similar capability.

Papermill is far more flexible and an open source project. When I saw this, I was ecstatic. I thought that blog post in Papermill was incredibly interesting. I immediately built a prototype of integration that naturally, I called Dagstermill that effectively allows you to wrap Papermill in a solid. Now the notebook becomes this reusable unit of computation that self describes its config,

describes what its inputs and outputs are. Now this is actually a thing which you can use to productionize.

We can go to authoring a notebook, to easily parameterizing it in a pretty simple way, to then running it in production in a very smooth fashion, which I think is incredibly compelling. I've been slightly surprised that Papermill hasn't gotten more traction, given how stunningly effective it's been at Netflix. It's interesting to think about why.

Our goal is to be partners with the Papermill team to really hopefully make Papermill notebooks far easier to productionize by integrating with the Dagster system. That was a very long-winded answer about Jupiter notebooks. Obviously, just to go back to initial point, they obviously are important because they're abused by – they're the de facto tool for data scientists and they feel an obvious need, despite their oddities and flaws.

[1:03:58.4] JM: If somebody wants to get started with Dagster today, who can use it today? Because I know the project is much more mature than it was a year ago. It's still not at your total apex goal. Who should use it today?

[1:04:17.4] NS: I think if you're an engineering-oriented data scientist, or one of those data engineers that has an instinct that there's got to be a better way possible and I care about developer ergonomics, I think those are – you are a great candidate for – this has been open source from beginning. My first line of code was on public github and it's been as the team has grown and as we've worked on it, nearly all the code is open source. Actually, I mean, a 100% of the Dagster code is open source.

We're published on PyPi, so you can just go to your console and type pip install dagster and pip install dagit and you're off to the races. I still think we're relatively early days, which means that I think you need to be a relatively advanced engineer that is willing to roll with the punches with a emerging abstraction. Also, we're really excited about our people who like to build their own tooling to integrate with existing – with these new abstractions, really partner with us to move the system forward. Those are the type of people I think who will be adopters.

If you want to completely out of the box, vertically integrated thing that is hyper mature and you want to abstract away code in a way, well like drag and drop tools and stuff and completely author stuff without opening up VS code or VIM or whatever, it's probably not for you right now.

[1:06:02.6] JM: We'll do another show 1.0. You're building a business around this, Elementl. You worked at Facebook for several years. How does building a startup compare to working at Facebook?

[1:06:15.5] NS: Yeah. I've been thinking about this a lot actually. Working within an organization is very different than working outside the organization, an organization you're partnering with. At Facebook, I was very accustomed to having 100% visibility into how everyone in the company was using systems that I had participated in building.

I remember very late in the company, one of my closest collaborators, Ola Okelola, he was still had a folder which he read every e-mail, which had the summary at minimum of every single piece of code written in our product codebase. Our team was very attuned to what everyone was doing and we could respond proactively to that.

Now the relationship is far different when you're dealing with external folks who are just hopping on to a Slack and submitting bug reports, or asking questions. Naturally, we always want to be like, "Hey, this is great. We're super happy you're using the system." What we really want is full context, like show us all the ways using the system, so we can really understand what's going on.

Adjusting to a lower information environment has been a challenge for us. I think we've had to switch to a mode with more active – call it customer success. I don't know what the non-corporate open source version of that sounds like, called developer success. Really being proactive and reaching out to those folks and really being like, "No. We are here for you. We want you to be a needy user. We want full context." I don't think a lot of people are accustomed to working that way in an open source context.

In terms of the peer open source development, that has been the biggest challenge. Because with the two examples which I always go back to, which is React and GraphQL, GraphQL is a

very mature system, conceptually by the time we open sourced it. We had used it for a few years, we had really done a rethink and a redesign. Then we were able to just put out a document. Because it was relatively – intellectually coherent, people were able to build on it with very little interaction, which looking back on it is actually remarkable.

We're much more in the mode of developing this out in the open and we have to change system a little bit as we go along the way. We're very committed to backwards compatibility. If you start using the system, we're not going to break you all the time. We're evolving more out in the open. I think that's been a big adjustment.

[1:09:05.6] JM: Anything else you want to add, Nick?

[1:09:08.8] NS: If you're interested in talking about it, we can – maybe the relationship between business model and open source.

[1:09:14.8] JM: Totally.

[1:09:15.9] NS: Is interesting. I know you're interested in that subject.

[1:09:17.2] JM: Well, of course I am. Give me your thoughts.

[1:09:20.9] NS: I think there's –

[1:09:22.1] JM: By the way, I guess to put this in context, are you talking about this just fundamentally the fact that your code is open source and you're trying to build a business around it, or are you talking about this in the context of other companies that have done this licensing?

[1:09:38.7] NS: Licensing is the particular issue, which is top of mind right now, because Confluent, Cockroach, Elastic, Mongo have all – Redis and probably other ones as well, have all been beginning to – they are trying to layer their software between open source and proprietary stack by having a – what I'll call a pseudo-proprietary license over at least a portion of their software effectively as defense mechanism against public cloud providers, especially AWS.

I've been thinking about this a lot about – because one of the advantages – there's a disadvantage to developing an open source project outside the context of a company, because of the issues I was talking about before. I think maturing it is more difficult if you're not totally embedded within a big company. However, what you can do is sit back and systematically think about how to thoughtfully construct the company to have a sustainable business built over it. That's been a very interesting exercise to do.

Now the model that I actually look to, and this might sound counterintuitive, but I think one of the most successful open core companies that often isn't classified as an open core company is github. The core technology that github is structured around is git.

What they are able to do is they're able to leverage the adoption of git to build a hosted product that's proprietary, that everyone accepts is this makes sense as a proprietary product. It has network effects, an identity layer. Then in addition to that, because they make git easier to use, there is this reflexive relationship between the two entities. Meaning that as github got more popular and git more popular, git became the inevitable technology and therefore, drove the adoption of github.

This is despite the fact that for a long time, github's product development was very slow in my opinion. Git has an extraordinarily hostile user interface, in terms of its command-line interface. It is so unnecessarily intimidating, especially for people who aren't classical computer scientists. Explain to someone who's just figuring out git like, "Oh, it's just all hashes." It's obvious. No, that's not obvious at all. That's the way I think about this.

The nice thing about that is that I think that there are far less cases where you're scared of another organization driving the underlying technology. I always think about this as maximizing the number of win-win relationships in the ecosystem.

What our goal will eventually to be have Elementl.cloud or whatever we call it, be a product where your developers, your builders have been using Dagster as a productivity tool to make themselves more productive and efficient, but we can leverage the adoption of it to build a product that is valuable to enterprises, where if you're at organization with bet on Dagster, it is

just natural for you to adopt Elementl.cloud, so that there's a place where you can log into and understand the health and the status and do that data management platform. It makes sense to have that be hosted and have workflows between different tools managed by product.

That's the shape of the business that I want to form. What's nice about that is that – I totally respect all the companies that are doing it and I understand why they're doing the proprietary licensing stuff. You don't have to deal with those shenanigans, so to speak.

[1:13:41.4] JM: Oh, I think so too.

[1:13:42.2] NS: Right? With github and git, there's a clear separation of church and state, so to speak. Obviously and partially, it's because github is not responsible for git, right? That's the one part of the analogy that doesn't work as well. There's no question that git is open source and will always be open source. There's no question that github is a proprietary product. Everyone's fine with that. It's very clear to communicate what's going on.

What we're going to be able to do is communicate very clearly that Dagster in its current form and whatever form, the moment we – everything in that github repo will be – I mean, I'm never going to say never, but the plan is that that will be open source forever and always. Then that we will explicitly build a closed source product that leverages the adoption of that, that will be a hosted proprietary product that will deliver value to enterprises on dimensions that developers don't want to deal with. That is of the medium-term vision for the company.

[1:14:44.0] JM: Wonderful. Well, I'm definitely a big fan of what you're doing. I encourage people to either check it out, try it for yourself or apply to work at Elementl perhaps. I know you're hiring.

[1:14:55.5] NS: We are hiring.

[1:14:56.6] JM: Nick, thanks for coming on the show.

[1:14:57.7] NS: Thank you so much. Thank you for the opportunity.

[END OF INTERVIEW]

[1:15:09.0] JM: You probably do not enjoy searching for a job. Engineers don't like sacrificing their time to do phone screens and we don't like doing whiteboard problems and working on tedious take-home projects. Everyone knows the software hiring process is not perfect, but what's the alternative? Triplebyte is the alternative. Triplebyte is a platform for finding a great software job faster.

Triplebyte works with 400-plus tech companies, including Dropbox, Adobe, Coursera and Cruise Automation. Triplebyte improves the hiring process by saving you time and fast-tracking you to final interviews. At triplebyte.com/sedaily, you can start your process by taking a quiz. After the quiz, you get interviewed by Triplebyte if you pass that quiz. If you pass that interview, you make it straight to multiple on-site interviews. If you take a job, you get an additional $1,000 signing bonus from Triplebyte, because you used the link triplebyte.com/sedaily.

That $1,000 is nice, but you might be making much more, since those multiple on-site interviews would put you in a great position to potentially get multiple offers. Then you could figure out what your salary actually should be.

Triplebyte does not look at candidates' backgrounds, like resumes and where they've worked and where they went to school. Triplebyte only cares about whether someone can code. I'm a huge fan of that aspect of their model. This means that they work with lots of people from non-traditional and unusual backgrounds.

To get started, just go to triplebyte.com/sedaily and take a quiz to get started. There's very little risk and you might find yourself in a great position getting multiple on-site interviews from just one quiz and a Triplebyte interview. Go to triplebyte.com/sedaily to try it out. Thank you to Triplebyte.

[END]