EPISODE 1058

[INTRODUCTION]

[00:00:00] JM: A data workflow scheduler is a tool used for connecting multiple systems together in order to build pipelines for processing data. A data pipeline might include a Hadoop task for ETL, a Spark task for stream processing, and a Tensorflow task to train a machine learning model. The workflow scheduler manages the tasks in that data pipeline and the logical flow between them.

Airflow is a popular data workflow scheduler that was originally created at Airbnb. Since then, the project is been adapted by numerous companies that need workflow orchestration for their data pipelines.

Jeremiah Lowin was a core contributor to Airflow for several years before he identified several features of Airflow that he wanted to change. Prefect is a dataflow scheduler that was born out of Jeremiah's experience working with Airflow. Prefect's features include data sharing between tasks, task parameterization and a different API than Airflow.

Jeremiah joins the show to discuss prefect and how his experience with Airflow led to his current work in dataflow scheduling with Prefect.

[SPONSOR MESSAGE]

[00:01:12] JM: Today's show is sponsored by StrongDM. StrongDM is a system for managing and monitoring access to servers, databases and Kubernetes clusters. You already treat infrastructure as code. StrongDM lets you do the same with access. With StrongDM, easily extend your identity provider to manage infrastructure access.

It's one click to onboard and one click to terminate. Instantly pull people in and out of roles. Admins get full auditability into anything that anyone does. When they connect? What queries they run? What commands are typed? It's full visibility into everything. For SSH, RDP and Kubernetes, that means video replays. For databases, it's a single unified query log across all

database management systems. StrongDM is used by admins at Greenhouse, Hurst, Peloton, Betterment and SoFi Control Access.

Start your free 14-day trial of StrongDM by going to softwareengineeringdaily.com/strongdm. That's softwareengineeringdaily.com/strongdm.

Thank you to StrongDM for sponsoring Software Engineering Daily.

[INTERVIEW]

[00:02:35] JM: Jeremiah Lowin, welcome to the show.

[00:02:38] JL: Thanks so much for having me.

[00:02:40] JM: You've spent several years in the Airflow community. How did you first get involved with Airflow?

[00:02:47] JL: At my previous company before I formed Prefect, I was overseeing a very broad data science and engineering mandate, and I was very literally running out of hours to get all of my work done. I began searching for something, anything that would help me basically automate myself.

By coincidence, Airflow was open sourced right around that time. So if it was in the day it was released, it was the day after I saw and I immediately said, "This will solve my problem. This is going to help me automate the things I need to do."

However, when Airflow was released, it was a Python 2 codebase and I switched basically cold turkey to Python 3 years ago. So that didn't work for me, but I recognized in Airflow something that would really be beneficial to me. I wrote to Max, who is the maintainer, still is, and I asked him, "If I go ahead and modernize this and make it Python 3 compatible, is that cool with you?" He said, "Of course. Why would anybody say no to that?"

My first interaction with the Airflow codebase not just as a user but as the developer was to introduce full Python 3 compatibility. It took a long time to touch pretty much every file, but as a result, I just became intimately familiar with the system and the product, and then of course as a user, continue to deploy it, and that was my introduction.

[00:04:03] JM: There are some acute problems that a data engineer encounters in today's space of tools, and one of them that you just touched on this the automation side of things. Data engineering is this funky practice that has kind of gotten formalized with the name data engineering in maybe the last five years, but we've been doing things like data engineering for a very long time, ETL jobs, building search indexes, building nightly reports. Now it's become more formalized just because there're more types of data engineering to be done. What are the problems in data engineering that Airflow solves?

[00:04:51] JL: That's a really wonderful question, and I think Airflow in some ways predates the formalization of data engineering that you're even referring to. I think Airflow actually solves a rather small subset of problems however it does so in a fairly simplistic way. I mean that in a good way actually, that it allows the paradigm that Airflow exposes to be generalized relatively easily as data engineering itself has become more formalized and more understood.

I think that the principal hallmark of a data engineer is someone who understands the access patterns around data. A data engineer might not be an expert in just, say, Postgres as supposed to MySQL, but they are going to be an expert in understanding how to interact with that system. If they in addition are in expert in Postgres itself as a discrete piece of technology, that's wonderful too, but if I'm looking to hire a data engineer, I'm looking for someone who can speak intelligently about the motion of data throughout a system, probably hydrating, being serialized, etc., who I'm going to trust to make sure that that infrastructure is going to keep it healthy.

What Airflow contributed really for the first time was the ability to define those health checks and that infrastructure flow in code, in Python code most importantly, because Python then and certainly now, of course, is the sort of emergent language, de fact language of data science, and therefore data engineering. I could go in with Airflow and I could say I want to do the following things and use a set of libraries and frameworks that felt normal to me as a data scientist and I could put them into motion by specifying when I wanted them to happen. I think more

importantly, what I wanted to happen when they failed, and that failure condition is to me the sort of central hallmark of a workflow orchestration system. It's actually not what it lets you do, because I'm perfectly capable to write code and just run it. It's actually how do I put it into the world in a way that I believe is secure and I have some insurance and confidence around its execution. Among all of Airflow's features, to me, the hallmark is the introduction of states, success and failure, and retry as code, things that I can respond to in my workflow infrastructure directly, not implicitly.

[00:07:07] JM: The workflow scheduler idea in today's context I think of a workflow scheduler for data jobs doing things like a nightly Hadoop scheduled job or a nightly Spark scheduled job that's copying all of the data from HDFS into a data warehouse as a simple operation. A more complex operation might be take a bunch of data that represents all the transactions that were generated over the previous month and load those into a Spark cluster and then do a multistage machine learning job across that dataset that is now in Spark. You've get a sequence of tasks to do there and you may represent these tasks as a DAG. Can you describe the kinds of DAGS that Airflow can execute, the DAG representation that Airflow formalized?

[00:08:19] JL: From a software engineering perspective, Airflow probably take any series of tasks that can be arranged into a DAG. But I think to answer your question directly, where is Airflow best used? The answer is very much the first set of examples that you had. Airflow was originally designed and is most popularly deployed to run nightly, maybe hourly if we push it, jobs that are taking place in a third-party or a remote system. The principal use case for Airflow is to make sure that – As I said earlier, to make sure that things happen at the prescribed time, and should they fail, we wait for that signal to comeback. Should they fail, we want to take some cleanup step. In your example, maybe want to tear down the cluster. If in fact we don't reach the end of the application, that should be the tear down itself. That is where Airflow is happiest.

The second set of things that you've described is where you run into trouble if you're using Airflow, and the reason for that is just Airflow didn't anticipate the type of use case as you get into machine learning and data science as it has become understood to us today where it's a very fast-paced, scale-out set of applications. The requirements are quite different.

If I'm running a machine learning training algorithm, I actually don't care as much. It's not as mission- critical to me if it fails. I'll just run it again, which is very different than ETL failing midstride. That is a mission-critical problem. I need to make sure that when I pick up, if I wrote it a an idempotent job, that's great. I can pick up right away. Everything is going to can be fine. But if I didn't, for any reason, I need to be very, very careful how I resume that interruption. Data science is a very qualitatively different set of requirements and Airflow didn't really anticipate those in its original design.

[00:10:03] JM: Okay. We've done a few shows on Airflow in the past. We actually have another one coming up with the Astronomer people, the Airflow company, or one of the companies that's been built around Airflow. I want to spend most of our time talking about your critiques of Airflow and your work on Prefect, which is your own scheduler that you built with some of the lessons you learned from the Airflow community. Tell me a little bit about what shortcomings of Airflow you identified over the course of your time spent as a committer to the project.

[00:10:44] JL: Sure. We've written a blog post about this, and I think we laid out eight or nine major technical considerations, which essentially boiled down not necessarily to things Airflow did wrong, because I'm not going to so far as to tell you that Airflow is bad or anything like that, but these are assumptions that are baked into Airflow that make it in many ways incompatible with what's emerged as a modern data science stack.

If your primary concern, if you want to do data engineering in the sense that we just talked about, every night at midnight, you want to kick off a Spark job. By all means, use Airflow. Something of that nature is well-equipped to the sort of semantic that Airflow exposes. But if you want to do something more interesting than that or more complicated than that, you're going to start fighting with the tool to get it to respect that.

There're a few places that that really rose. One of them is just passing data around through the system. Today, when I say Airflow doesn't support dataflow, people say, "Yes. It does. It has something called XCom." But it only has an XCom, because I wrote XCom just a few years ago to solve this problem. Airflow has no natural understanding of moving data between tasks. For a data scientist, that's a nonstarter. If I can't move data between tasks that are operating on as

transform, why am I wasting my time building a data pipeline on this platform? Why don't I go use a system that natively supports data transparency and data pipelining?

Even XComs are sort of a hack around Airflow's principal vocabulary and that when I set up an XCom relationship between two tasks in Airflow, the Airflow orchestration engine actually has no idea that I've created that dependency. Going back to what you said a moment ago about a DAG being this central object in Airflow, I've just sort of worked around that understanding. I've now presented a strict dependency where one task depends on data generated by another task, and my workflow orchestration system actually has no way to discover that dependency. It might run those tasks out of order.

Now the fix is somewhat trivial. I just manually create the dependency and inform the engine, but here I am now working around the engine double defining a dependency to do something that is a data scientist I actually take for granted. The only user who's going to be of the mindset who wants to take this on is someone who, A, for some reason is doing data science work, and B, for some reason doesn't have access to a quality data science toolkit.

Very quickly we run into this problem where the abstractions that Airflow is providing do not actually match the type of work that people want to deploy on it. Data passing is one idea. Parameterization is another great example. Typically, once I've defined a workflow, especially if I'm doing something with machine learning where a strict parameterization of the training algorithm or even the data I'm passing to the algorithm itself takes on this form where I define the logic, the workflow logic once. Then as a second step, I want to just past new inputs to it or new parameters to it, but rerun that same core logic. Airflow doesn't have a native way to express this. It has a lose concept of global variables.

But if you want to tie those to your workflows, of which might have tens, hundreds, or thousands, you're not taking on the job of managing those inputs and managing their orchestration yourself. That's honestly what you wanted the orchestration engine to take care of for you.

The list of things goes on and on. As you can see, there are many use cases for which none of this matters. You want to move ETL – Excuse me. You want to have an ETL job that moves

data every night from one system to another? Great! You don't really need parameterization. You don't really need this stuff. Just do it very simply in your third-party system. But what if you want to do that in three different environments? Step, staging and product? You want to write that DAG three different times? Because Airflow is going to require you to do that, or do you want to parameterize it? Now all of a sudden even this simple, simple use case of ETL has taken a step into what I would consider a more modern data context where parameterization is key.

We haven't even touched on things like scale and speed, but as you can imagine, just given the way Airflow is architected, those are not primary considerations, whereas in the data science world, I'm very used to pushing a button, spinning up 10,000 tasks, watching the next queue and [inaudible 00:14:50] cluster's been down in less done a second. Airflow takes 10 seconds to look at a task. We're just order of magnitude off as far as what really I can bring to the table.

[00:15:01] JM: Okay. Let's dissect some of those critiques, the first one, the data sharing or dataflow. As you mentioned the blog post that you wrote talks about XCom, which I didn't know. I guess that's a system you wrote? You wrote XCom?

[00:15:18] JL: I did. Yup.

[00:15:19] JM: XCom, from your writing about it, is a system for handing metadata about your overall workflow from one task of the DAG to another. Is that right?

[00:15:34] JL: That is how it was originally designed. They handed metadata around. The specific use case I'll actually share with you was one task would write data to let's just say a location and S3, and then I needed the next task to load that data and operate on. But here's the interesting problem. How do I tell the next task where the data lives? I need a way to move data between the tasks. At the time, before XComs, there was no facility in Airflow for doing that other than using yet another third-party database or something like that to, again, manually configure it, manually set the key. Make sure it's shared. You just have to jump through so many hoops do something so simple. The original conception of this was I just wanted a way to pass that string URI for an S3 blob to another task and developed XComs to do that.

[SPONSOR MESSAGE]

[00:16:33] JM: There are two ways to add analytics to your application, you can build them yourself with basic charts and dashboards using free open source charting libraries, or you can use a comprehensive analytics platform from a partner that you trust. If you've tried to build it yourself, you know that free actually is not so free. There are hidden costs like time, and maintenance, and technical debt, and those hidden costs can really add up.

Check out Logi Analytics. Logi Analytics is developer grade embedded analytics solutions and they make it easy to create branded dashboards and report the scale within your own application. You can stop wasting time piecing together analytics and allow yourself to focus on your core application. You can go to logianalytics.com/sedaily and you can get a demo to see what is possible with Logi today. Go to logianalytics.com/sedaily. That L-O-G-I-analytics.com/sedaily.

[INTERVIEW CONTINUED]

[00:17:46] JM: When I think about what I want out of a workflow manager, at least in the way that Airflow was defined, is I think of something lightweight. I think of something where if I want to do data sharing between different parts of my workflow, that workflow should basically be in Spark. Spark was built to do this in-memory resilient distributed dataset thing where I run multiple operations across my RDD. So I should be just defining a data engineering task in Spark and then Airflow can perhaps kick-off that data engineering task. But it's not really within the scope of Airflow to be handing data from one part of the task to another. Are you talking about like kind of designing a totally – Or I guess I'll just let you respond to that.

[00:18:48] JL: Yeah. No. This is a really wonderful point. Generally speaking, I agree with you, or at least I should I used to agree with you very strongly, which is why XComs were originally designed not to pass data, because if you're doing data processing in Airflow, I think you're in like a very different category of superuser, because very few people are doing data processing in Airflow. They are using Airflow to instrument of third-party system. Do that data processing in a system like Spark which as you said is definitely a good place to do it. It's a system specifically designed for that. That's why it's very interesting when you look at the history of XComs. They

were originally designed not to really assist in the data processing, but as in my example, just to pass the data location.

For example, one task might – I'm making this up of course. We're a few years removed from actually doing this, but one task might make sure the data was there. Perhaps validate that the data was there, that it had arrived. The next task might kick off a Spark job to actually do the processing. But somewhere I have to either repeat my logic to do the data discovery or the easier thing, I'm just going to pass that URI.

My opinion a long time ago was exactly the same as the one you just laid out. Have the orchestrator do the orchestration and just pass around the information needed for the jobs in question. Do the jobs in the best place for them.

What almost immediately happened, and this is a great lesson about designing software. What almost immediately happened as once the XCom utility became available, and that must have been four years ago, people started using it to do data processing inside of Airflow. In a very interesting way, this thing that was supposed to facilitate what you just described actually led to exactly the opposite. Why? Probably because it became much easier, and because while it's very easy for us to come up with examples where a different execution engine is obviously the best choice, you mentioned Spark. We might think of something like a compiled machine learning framework. Maybe once upon a time it would be Theano. Today it might be Tensorflow and PyTorch. Once people had the facility to just work with data and then pass it to the next task, if it's simple enough, why not do it?

All of a sudden, data processing comes into Airflow and they're using XComs cons to move it around, and we started getting this weird class of bugs where people would be passing a gigabyte Pandas data frame from task-to-task using an XCom to serialize it. Because XComs were originally designed just to handle these tiny little strings, it never occurred to me, for example, to flush them out of the database.

So if you pass that gigabyte data frame through 10 different tasks, that means every time your DAG ran, you were putting 10 gigabytes of data permanently in the database that Airflow depends on in another database. But also just because of how Airflow is built, anyone who has

access to your system could actually go look at that data. Your data lineage is very – You're in a bad spot.

Anyway, all of this, the point I'm making is I actually used to firmly believe what you just said. Data execution should go live elsewhere. As a matter of fact, Prefect was originally designed not to bring data execution into the orchestration manager, but to make the orchestration manager more compatible with the execution layer.

My favorite execution engine is Dask. I think it's incredible. As a data scientist, I can't really find a better tool for just getting my computations from a single core for testing and then out into the world. But Dask doesn't have that high-level nice state-driven API that Airflow does for describing what I want to happen when things fail. The very first prototype of Prefect was just smashing the two together. It was adding things like failure states, conditions, retries on top of Dask's really excellent work scheduling engine, and Prefect just sat there and govern the execution of that work, but Dask was doing the heavy lifting.

I continue to think that that is actually what you want to do. However, in order to properly orchestrate work, we have to decide what are the bite-size chunks that we want to govern? If my entire application fits in the Spark job and I never want, just for example, my workflow manager to be able to interrupt it or retry it or give me transparency into it, then that's great. My unit of work is the entire Spark application. It might take a minute. It might take 10 hours. It doesn't matter.

As soon as we have a workflow orchestration engine that is capable of expressing more dynamic units of work, smaller units of work, it actually becomes desirable to make those tasks more granular. Because they're more granular, even again if we're still going to farm them out to a Spark cluster or whatever the case may be, because the more granular we need to have better data passing mechanisms in the workflow engine because it's going to connect all these things together in the way that's most familiar to the execution layer, and that of course is going to be premised on moving data around.

We've gotten to a place where if you look through the best practices for something like Airflow, you'll see a recommendation to make big tasks. One of the reasons for that is that Airflow

historically doesn't support moving data between tasks, so you had no choice. You task had to be as big as it could be to keep the data inside the task.

With Prefect, we take the opposite approach. You should have small tasks. Your workflow orchestration engine should be reporting to you at the unit of work level. If your unit of work is a giant Spark application that you kick out and just send them to the world and you don't need the workflow system to really know what's going on, great! That is fantastic. That works well.

In the other hand your unit of work is small string transformations, or loading variables, or just pinging a database to make sure it's alive, or rerun a Slack bot on Prefect, or work there is very small, that's great too. I think the important thing about the workflow orchestration engine is to respect the user's ability to actually do the work where they want without impeding their ability to do that or forcing them to use an execution engine that's inappropriate.

[00:24:38] JM: I find this a compelling pitch, but just to pause a little bit longer on the notion of data sharing in an Airflow workflow, you mentioned that one thing that people do if they're trying to have a multistage data pipeline is write a file to S3, and S3 is your intermediate handoff point. Couldn't these workflows also use a data warehouse? Like a data warehouse would just be faster, right? I would imagine that one of the big issues here is that the dataset can be gigantic in these multistage systems. Loading it into memory multiple times is some work.

[00:25:23] JL: You're absolutely right. You're absolutely right. I don't want to suggest that my example of S3 is necessarily sort of canonical or typical. People use a variety of stateful mechanisms to move data around or create representations of that data. But the important thing I think about the XCom mechanism is not necessary that you would use it to move the data around. In fact, I would argue that if you're moving more than – I don't know. Let's just say one kilobyte of data in an XCom, you are misusing it. You are not using it for its intended purpose, which is to move small bits of metadata around the system and make them available to tasks.

If you are moving massive amounts of data around your system, then course you do not want to bring them into your workflow orchestration and then move them to the next task. That would just be using a tool for a purpose it wasn't designed for. Can you? You can as long as it meets the resource requirements of whatever engine you're using or whatever workflow platform

you've chosen. But I don't think I'd ever go so far as to say use the workflow orchestrator to do the work. I would say use it to govern the work and make sure that the work gets done.

I think this is actually a place where a lot of tools in our space kind of fall down and they don't know what problem they solve. Just to give you an example, very early in the life of Prefect, we decided to define all these things that we're talking about. We called it positive engineering and negative engineering. Positive engineering is what you want to do. Negative engineering is the work it takes to make sure it happens, and it was one of these things where in all honesty when we came up with it, it sounded a bit fluffy, right? It's a little bit ambiguous, but immediately we have this recognition from people we were talking to, even non-technical people, even sort of managers of stuff would look at this and be like, "I recognize these negative engineering frictions. I recognize where my team is doing work that is not directly correlated to their objective in my engineering team," and that could be tracking down logs, trace backs, finding out that errors took place, that could be retrying code, that could be deploying code, it could parameterizing code. That could be fighting with the system to do the thing that you just mentioned where they brought too much data in and they just took down the node, because RAM couldn't handle it. Whatever the case may be, these are all examples of places where they wanted to do something and probably had a tool that was really good at doing it; moving data around, processing data, analyzing it, training a machine learning model, whatever the case may be.

But the place they were spending their time and energy and great frustration was on the surface area around that usually related to the failure case. When we picked up Prefect, that's where we focused. Again, that's why I appreciate the question you asked earlier about sort of philosophically where should the work get done. When we designed a tool like Prefect, we want to take the position not only do we not care where the work gets done. Wherever the work gets done, we want to make sure that you can happily do it there.

Prefect's job, or I would argue, any good workflow manager's job, is to report to you and give you visibility into the status of those operations at the level of granularity that you've chosen to instrument your system. If you want to do that giant Spark job that runs for 10 hours and a system like Prefect doesn't get to find out about it, it's just one task. Great! We will tell you when it starts. We will tell you when it ends and we'll tell you if there was any problem.

If you want to chop that up into 20 small tasks so that each 15 minutes you get an update or whatever it is, that's great too. The work can still be happening out in the Spark cluster, and Prefect is just going to be there to check in. This philosophical divide is super important. We don't want to take the opinion that you must do work away from the system.

Look, in all honesty, at the end of the day, it's Python. People will make it do whatever they want. I didn't write XComs, because I had a special knowledge of how Airflow worked. I wrote XComs because it's Python code, and I said, "This would be useful for Airflow to do."

You can always make any piece of software do whatever you want. The real challenge I think is to make sure that the software represents how people actually use it in the real world and solves the frustrations and the challenges that they have, which by and large when we talk about this orchestration space, it's actually not doing the work, because we do have systems for doing the work. It's actually governing the work and discovering when something went wrong.

[00:29:34] JM: Airflow at this point has been around for about five years in the open and it's one of these tools where you do hear people talk about changes that could be made to it. Everybody has their complaints about Airflow, but there is such a momentum to the project, because it's so – And it's so widely used.

From my point of view, won the first war of the workflow orchestrators. There's also Luigi. I think Luigi is still used, but that was the Spotify kind of equivalent or similar workflow orchestration system. But Airflow has become the more popular one and it's often hard to unseat these things when they have momentum. But sometimes it's not so important to unseat it, like maybe there's not actually so much that has gone into it that you can't reinvent yourself with your own project. You've started Prefect, and your goal has been to make and architect and engineer and fully develop an ecosystem that is an alternative to Airflow. Now you're several years into doing that. Tell me about the process of developing Prefect. What about it has been difficult?

[00:31:03] JL: It's a wonderful question, and you're right. We did originally envisioned Prefect as an alternative to Airflow, not a replacement for Airflow, and it's been very interesting

especially more recently as the project picks up steam to see people describe it as the successor to Airflow. Prefect solves a use case that Airflow cannot solve.

We actually encountered in a very direct way. In the very early days of Prefect, it's more than a year ago when we first open sourced it, we tried to put up examples to motivate why a system like Prefect was helpful and a system like Airflow was not, and we sort of – We put up, "Here's how you do this in Airflow, and here's how you do this in Prefect," and we got these complaints from the community that we weren't being fair. It was a very weird situation, because my argument in some sense was, "Of course we're not being fair. We're showing things that Airflow is bad at doing and we're that Prefect is good at doing them."

What was happening though is that people were viewing Prefect as a complete replacement for Airflow, not a way to do these advanced things, these sort of more modern data science-y things. Therefore, yes, in that light, we were cherry picking things that Airflow is bad at. Actually, that why not Airflow blog post that we mentioned earlier, that was the result of sitting down for about two months and thinking how do we honestly motivate the parts of Airflow that we think are incompatible with the modern data stack and why we feel that way and why we've gone out to solve them.

Again, as I said earlier, Prefect started out by cramming together the pieces of Airflow I really liked stateful architecture. Sorry. By stateful here, I don't mean like writing state to a database. I mean, literally, it deals with states, failure, success, retries, skip with my favorite millisecond latency, smart data moving scheduling engine Dask. We're not tied to Dask. You can really use any execution engine you want with Prefect. I just love Dask. So we recommend using it. We think you get the best result that way.

We slammed these things together, but not without trying to make the changes in Airflow first. In a couple of weeks, I think, as record this in what? Early April, I think in a week or two it's going to be four years since I proposed what is now Prefect's API as what at the time I was saying should be Airflow 2, and that gives you some sense since Airflow 2 is still not out, about how long a road it's been to get a new version of Airflow out the door.

But back then I said, “If Airflow wants to really stick around, ironically, with a modern data science stack, it's going to need to adapt the following change. It’s going to need API. It’s going to need first-class way of moving data around. It's going to need to be able to run two workflows at the same time.” This is like one of those little tiny technical details that really matters for some of our users on the Prefect side. Airflow can't schedule the same DAG two times at the same time, because time is a primary key in the database.

If you want to run two different parameterized versions of your workflow, and forget for a moment that Airflow doesn’t support parameters, but if you wanted to do that, you’re just out of luck. A design decision was made years ago that Airflow jobs would never run at the same time for the same workflow. Today, that has severe implications for some of our data science user who like to spin up many jobs at the same time with a variety of different parameters.

Anyway, I'm getting off-topic. When we first slammed these two things together, one of the biggest challenges was how do we motivate this? How do we argue, “Yes, as a data scientist –” And my background is first and foremost as a data scientist. As data scientist, it’s clear to me that a lot of my use cases and workflow space don't map on Airflow semantics. But how do I motivate this into the world?

I started collecting opinions from people. This is before Prefect was a company. This is when Prefect was just a little tool that I built for myself. I started collecting opinions from people. I just started asking them what kind of problems are they running into. I asked data scientists. I asked data engineers. I asked data managers. I asked C levels. I asked junior people. I just talked to people about their problems. They came up with a list of about – I think it was probably seven or eight words that people kept using over and over when they described their frustrations with their workflow system whether it was Airflow or not. It didn't matter. Those words – And they wouldn't surprise you. Some of them were very basic; speed scale, parameterization that we mentioned earlier, data flow, mapping and dynamism, caching of a very specific type. These were the words that people used to describe the frustrations they felt in their day-to-day job trying to use data and trying to deploy data applications.

We spent an enormous amount of time trying to figure out what is the API that we can expose that will take these words that people are telling us? Their principal issues with data workflow

management, and get them into a nice cohesive Python framework. We spent an enormous amount of time on that. We weren't always right, but I think what we've ended up with is a really, really great set of, we might say, Lego bricks, that each one represents one of these dimensions that people say is critical to building a solid data application and are guaranteed the thing that we work hard on us is to make sure that they snap together. It takes on the Lego amount where they all come from the same kit. You're not going to connect to something that looks like a pirate something that looks like a spaceship, because that would be crazy. We promised that they all work together.

That was in the beginning, going from this just base frustration that I felt and trying really hard not to just solve my problem, but understand how is my problem an instance of a more broad description of a problem? How do I solve that? What is the vocabulary we need to expose in the framework? That's where Prefect you was really born and that's why it actually took life very different than Airflow.

[00:36:47] JM: The mention of Dask is timely, because I did a show about Dask literally last week with Matthew Rocklin, and my understanding of Dask is it's a way of efficiently representing large objects in in Python in performing operations over those large objects. So if you have a really big Pandas data frame, as you mentioned earlier, this will allow you to efficiently manage that big in-memory Pandas data frame, or it doesn't have to be entirely in-memory. You could have part of it beyond disk. But in any case, it lets you manage your large objects efficiently. This is something that was not as prominent when Airflow came out. The data engineering, the data science world looked a little bit different when Airflow was first born.

My understanding is, today, with Dask for example, you could string together a large scikit-learn job and a large XG boost job. You could string these things together. Well, using these large Python objects that you can define in Dask. I can imagine wanting the ability to use Dask together with a rich ecosystem of maybe like connectors and handing things from one system to another. But I'm still, I guess, having trouble understanding why that couldn't be accomplished in Airflow. Maybe could help me understand to what extent is Dask an enabler here and how does the contemporary context of how people are building their Python data science and data engineering workflows relate to this comparison between Airflow and Prefect.

[00:38:51] JL: Very good question. There're a lot of pieces we can unpack from it. But it is another example of a place where I tried to solve this problem in Airflow. Airflow, if you dig in it, has a Dask executor, which does exactly what it sounds like. It ships the work off to Dask. I believe no one's maintained it. I think it's not compatible with the current version of Airflow, but it's still there in the codebase if you want to go take a look at it. Why did I build it? I built it because I took exactly the point of view that you just laid out. I said, "Airflow is my orchestrator. Dask is where I'm doing all my data science. Let's just get them to work together."

But what I ran up against is the fact that Airflow's conception of where execution takes place and what it scheduler is responsible for is very interesting. The Airflow scheduler is running on I think like a 5-second loop or a 10-second loop. Some very, very slow process. Let's just say every 10 – I think it's every five seconds, but it takes two looks at a task. I think it's 10 seconds to run a task. Every five seconds, it looks, it finds a task, and then five seconds later it runs that task by submitting it through the Dask executor to the Dask cluster, and that's great, and it runs in Dask. Comes back, it returns successfully. 10 seconds later the next task will go out.

One of the reasons that this doesn't work with the Airflow model is that the Airflow scheduler is the bottleneck, the speed at which it examines tasks, the extreme amount of work that it does. It reparses the entire DAG to load up a task. It just doesn't fit the model where what I'm used to doing with Dask is pushing a button and watching 10,000 little things spin up in a real-time UI all doing work, all communicating with each other and then turning themselves off a second later when my result is computed.

We have this just strict incompatibility between the way I want to do work in Dask and the way Airflow is allowing me to actually form that work out. The other part of your question about the nature of Dask is a really interesting one as well, because it's a great chance to highlight what I think is a hallmark of an effective framework.

Dask is exactly what you just described. I's a way to do out-of-core computation on large Python datasets, and I'll argue it's the easiest and best. However, the aspect of Dask that I actually find most exciting is not that. When I say Dask, and I always forget I need to sort of clarify that sometimes when I say Dask, I'm actually talking about a subproject of Dask, which is the Dask distributed project.

The Dask engine at its core is a way of figuring out what tasks need to run and where and has these great algorithms for taking, as you described, the Pandas data frame. Chopping it up into little pieces and doing essentially an asynchronous analysis of it out, out of core and distributed. The Dask distributed subproject is a way of doing that transparently over multiple CPUs and just farming the work out to an enormous cluster. It actually doesn't care about the size of the data or the way you're doing it. In fact, you could map over a string. You could take A, B, C, map over it, spin up three different tasks in three different computers. It's the most trivial thing in the world, but the nice thing about using Dask distributed is all of that work, the scheduling over a distributed set of possibly geographically disparate computers, the collection of work, the making sure that the work is act and brought back to the client. All of that distributed computing stuff, which is really hard. That's what the Dask distributed project takes care of.

Thee piece of Dask that's actually the most interesting to us and Prefect is not Dask's ability to move Pandas data frames around, although that's great. If people are using up by all means, go right ahead. The piece of it that we find so cool is that by building on top of Dask, we have instant ability to take advantage of the distributed subproject and get this HPC scale out that's practically invisible to our users. That piece of it, not just because of how slowly the Airflow scheduler is ticking over as a bottleneck, but simply the inability to interface with an external cluster. I think Celery is the only sort of nonlocal process system that Airflow is really supporting right now. I know they're moving to this interesting – It's the first time Airflow is going to sort of learn in on a single infrastructure provider and go for this KNative approach, but Celery is the only way that Airflow allows you to really get off the box that the Airflow scheduler is running on. Even there, you're still subject to this bottleneck, this once every 10-second bottleneck.

Again, as a data scientist, I'm just like, "That doesn't work for me. That's not how I do work." Does that help to explain why Dask, if you think about it as this transparent pure Python way to get HPC scale out through the distributed subproject? All of a sudden, it's like you give your workflow orchestrator superpowers as long as you don't make this decision early on in the orchestration layer to bottleneck yourself.

We've worked with folks like Matt, Matt Rocklin, who's just phenomenal. We've been working with him for a long time. We recently brought Jim Crist who is one of the top maintainers of the

Dask ecosystem over to Prefect where he's been a continuous Dask maintainer, but also obviously help us supercharge Prefect's integrations with the Dask world. This is just one of those cases where if you can transparently take advantage of an HPC set up like that, you just gain these superpowers as a data scientist. You may not care. If you're doing ETL or you're kicking off a single thing in a Spark cluster, you may not care about this. That's completely fine. You don't have to. But if you are one of our data science users, and we have quite a few, this is actually what makes all the difference. Because otherwise they're going to have to maintain their own Dask infrastructure in addition to a workflow orchestration, that's going to be very painful.

[SPONSOR MESSAGE]

[00:44:33] JM: When I'm building a new product, G2i is the company that I call on to help me find a developer who can build the first version of my product. G2i is a hiring platform run by engineers that matches you with React, React Native, GraphQL and mobile engineers who you can trust. Whether you are a new company building your first product, like me, or an established company that wants additional engineering help, G2i has the talent that you need to accomplish your goals.

Go to softwareengineeringdaily.com/g2i to learn more about what G2i has to offer. We've also done several shows with the people who run G2i, Gabe Greenberg, and the rest of his team. These are engineers who know about the React ecosystem, about the mobile ecosystem, about GraphQL, React Native. They know their stuff and they run a great organization.

In my personal experience, G2i has linked me up with experienced engineers that can fit my budget, and the G2i staff are friendly and easy to work with. They know how product development works. They can help you find the perfect engineer for your stack, and you can go to softwareengineeringdaily.com/g2i to learn more about G2i.

Thank you to G2i for being a great supporter of Software Engineering Daily both as listeners and also as people who have contributed code that have helped me out in my projects. So if you want to get some additional help for your engineering projects, go to softwareengineeringdaily.com/g2i.

[INTERVIEW CONTINUED]

[00:46:22] JM: I'm having a little bit of trouble understanding what that looks like in practice. If I am a data scientist and I – Like Prefect is solving this problem for me that involves Dask distributed. Let's say I'm running on AWS infrastructure. Can you give me an example that will make it more concrete for what problem you're describing here?

[00:46:46] JL: Sure. Whether we use Dask or not, for example, we don't even need to use Dask. So let's stick with AWS. We have a lot of people use Fargate to do this. We have an executor for Fargate. One of the core problems that I have as a data scientists is that the nature of my workflows tends to be very wide, very fast, very wide and often not discoverable until runtime. This is a MapReduce problem. Of course, we've talked about Spark a lot. For those of us who live in a Spark world, this is very familiar. If you can live in a Spark world and your systems are appropriate for the semantics that Spark exposes, you probably should. That's great. Again, Prefect is not trying to replace that. Prefect is trying to make it easier for people who use a variety of tools to glue them together and use them in interesting ways.

But if you need parallelism in your workflow, if your workflow involves 10,000 tasks, which is by an order of magnitude not the most people are pumping through our system, you need a way to transparently scale out and you need to have a workflow orchestration manager that can keep track of all that work. So we have to have this delicate partnership. We need to be working with an execution layer that can handle that volume of work. We need to have an orchestration manager that can keep track of that volume of work, and Dask, through something like Prefect, makes that marriage possible.

Airflow is too slow in the orchestration layer to allow that type of scale out. I just don't think it's possible, to be honest. I'm a little more removed from the Airflow codebase at this time, but it sure wasn't possible when I was involved. I could go full in on an execution layer like to Dask or like Spark where I can get that trivial scale out, but sometimes this, and this is where it gets interesting. Sometimes I don't know the scope of a DAG until I start running it. There's a runtime discoverability on it.

Even if I did, I still want my workflows orchestration layer to respect what's happening in the execution layer. I want them to work together. I want it to be aware of this type of scale out. We could get – And I'm happy. We can go into the details of specifically how we do mapping and how we farm it out to the Dask cluster and how we bring it back, but I think the motivation here is if you are doing something that involves spinning up 10,000 tasks, especially ones that come up that are discovered at runtime and you want your execution layer to treat them as first-class objects and do it scale out on as many cores as you provide and you're on your orchestration layer to do what it's good at and give you status updates and tell you which of them succeeded and which of them failed and really let you know what the progress is, then you need to make sure that you have that tight integration.

Airflow lacks that. We typically don't see that in other workflow orchestration systems unless they are very, very intimately tied to a specific set of infrastructure. Kubeflow comes to mind where if you're doing machine learning on Kubernetes, it's great. It's a great option. But our goal is to make it transparent. If that's your use case, there's really nothing about a workflow orchestrator that should stop you from doing that. The role of the workflow orchestration engine should not be a bottleneck. It just simply report back to you in a nice way what's happening out in the world.

Again, the classic use case for us is a machine learning training and analytic. Involves thousands of tasks. I want them to run as fast as possible in the fast execution layer I have, but I still want to get an orchestration layer that reports back on them to me. We really need that communication pattern to be solid to make that possible.

[00:50:03] JM: The distinction there in the feedback, so let's select let's say I have – Let's say the Airflow community has fixed whatever Dask driver or executor issue. I think if I understand correctly, I've got some Python script written that uses Dask, and if I'm using the Airflow ecosystem, I have a an executor, and Airflow executor, that knows how to manage that script, the Python script that involves Dask. It's going to spin up my machines and execute that task across them. Then there's going to be some result from that.

In contrast, it's like while that task is going on, Airflow itself is not reporting data back to me about what's going on in those servers I suppose? Then in contrast, in Prefect, you're saying

that there is a tighter integration between the workflow orchestration system and the Dask job that's running and being spun up. Can you drill a little bit deeper into that distinction?

[00:51:15] JL: That's correct. To be honest with you, I'm not even sure that what you're describing is possible in Airflow. I think the best case in Airflow would be that you wrote a job, a single Airflow task, or excuse me, a single Airflow operator to use the right terminology, that when submitted to a Dask cluster, or rather when submitted to Airflow's own executor layer and in turn submitted to a Dask cluster, then spun up thousands of children processes.

Is that a viable way to do it? It not only is it. It's how people have been doing it. It's how they get around this limitation that we're describing. But now I've got these thousands of child processes out there in the world living in my Dask cluster and what happens when one of them fails? It's divorced from the workflow orchestration manager, which means critically it can't be retried.

Just for example. I mean, there are other things. There're a lot of problems as well, but let's just drill in for a second. I cannot retry that one of the 10,000 things that I just spun up. The reason I can't is because retries are a semantic of the workflow orchestration engine not of the execution layer. In this world where I'm co-opting Airflow to do this thing, I guess what I would do is, first of all, I don't have an easy way to find out that that failure took place. I'm actually having trouble motivating how I as a data scientist would solve this problem in Airflow, because what I would need is I would need to launch a new job into that cluster whose sole responsibility was to examine I guess the outputs of those children tasks and report back to me if any of them had failed. Maybe I represent that as in Airflow task so that I can get this failure state back into my orchestration layer. Maybe if that fails, I convince Airflow to rerun the other task, but now I'm starting to introduce something that sounds dangerously like a cycle into my DAG.

This frustration becomes very readily apparent. I am struggling right now to think how would I launch this child processes in an Airflow governed system and also gain introspection into their state? I can't think of it. In Prefect, it's very easy. We have a map operator you call .map that tells the system that there's going to be a bunch of child processes discovered at runtime. If you already know about them, you don't have to do the map. We just treat them as normal tasks.

Now what's going to happen is before any child process runs, it's going to register itself with Prefect and it's going to say, "Hey, here I am. I'm a fully featured Prefect task. You can retry me. You can do all the stuff. I can be rediscovered in the future. I can be resumed." All of a sudden I gain this visibility. I'm going to find out if an error took place. Let's be honest. If I'm doing this type of work, if I'm really doing a large-scale map in a remote system, the likelihood of an error is much higher in the map and before the map, because the map is where the work is taking place. It actually becomes much more important that I have that visibility.

It's sort of funny. I actually don't think this is possible. I shouldn't say possible. Anything is possible when you have access to a language like Python, right? We can find a way to make this happen. But I think it's certainly nontrivial for anyone to get this to work effectively in Airflow. Here's sort of the sad truth if we're completely honest with ourselves. What I'm describing is not hard, and this is sort of a drum that we keep beating at Prefect. We don't need for this to be sophisticated and hard, and big data, and impossible to achieve without the workflow orchestration system. In fact, it's quite the opposite. All these things they we're talking about today, they're kind of easy. They're kind of trivial. We're talking about like retries. We're talking about phoning home when something fails, mapping. I mean, it's not like MapReduce is a new concept, right?

When we really step back and think about what are we doing here, and this is where the sort of fundamental question of what problem does the workflow orchestration layer actually solve. These are easy things. Anyone of them is easy, and that's why Prefect's biggest competition is not actually Airflow. It's actually homegrown systems. The number one solution that people who started using Prefect are coming from is a homegrown system.

That's because anybody who looks at this in any one instance will just shrug. They're like, "Well, this is easy. I'll do it myself," and they'll do it themselves. Then the second problem will come along and they'll say, "Oh! I need to – In addition to retrying, I need to track failures. Oh! In addition to tracking failures, I need to track logs. Oh! In addition to tracking logs, I need to actually register each node so that I can de-serialize the data appropriately where it lived." I think the edge case list will go on and on and on and on and pretty soon you're in maintenance hell, and that's usually when people start looking for something like Prefect.

It's a very important consideration here and it almost feels funny for me to say, because I would love to sit here and tell you, you should use Prefect because you can spin up thousands of advanced machine learning analytic tasks. Yes, you can, and there's a reason to use Prefect. But a more interesting reason to use Prefect is so that it'll save you some time writing retries because it's just a lot easier. That acknowledgment of the triviality of each piece of this problem is something that I think tools that misguided belief that they're there to help with the execution of code mess up over and over and over, and that's what we're seeing in our space right now. We're seeing a lot of use-case-driven workflow orchestrator. Many of them are open sourced by very large companies and they reflect a use case or a way of working that's familiar to a large company, and that's generally speaking good within that company. But when you get out into the real world and you talk to the sort of people that I talk to as an Airflow PMC, thousands of data engineers in businesses ranging from one person to hundreds of thousands. You find that the use cases that people struggle with and the things that would improve their lives, they tend to not be giant scale, advanced, sophisticated problems. They tend to be, "Oh! I've been trying to track down the log for three hours, but the Kubernetes job that I spun off to run this thing in, it's not around anymore and I forgot to configure it to ship the logs to a state somewhere. I'm screwed. I can't find out what this error and I can't trap it and I'm wasting all my time."

That's really where people feel pain. A big part of our philosophy at Prefect is if you could guarantee this stuff worked, you don't need Prefect. You don't need Airflow. You don't need any of these things. You're a competent programmer. You can do this, right? You only need these governance systems when one of two things is true. The scope of what you're doing is big enough that you just can't keep track of it and you want something to tell you what's going on, or you're worried that something will go wrong.

I think you asked earlier what are some of the challenges we found in building Prefect. I think one of the biggest challenges is how do we not get in someone's way when things are going well but still deliver a meaningful and good experience when things go wrong? The only way that we can do that in all honesty, in all sort of intellectual honesty, is to acknowledge that most of the problems we solve are very trivial until an up to the moment when they go wrong and then, man, you wish you had them. It's like buying fire insurance for your house. It's a very easy transaction. It's easy to frankly forget about if it were mandated. Most the time, you never need it. In fact, it's very hard to demonstrate that you should buy fire insurance for your house,

because you're not going to – Your house just light on fire every day. It's very hard to motivate this purchase except for the fear of something going wrong, and that's where we see Prefect and that's where we think workflow orchestration managers need to live. They're insurance products. They're only useful when things go wrong and they should strive to solve these complexities in the failure case. When things go well, when you know your execution layer, whether it's Dask or Spark, whatever it is, that's great. That's your expertise. That's the positive engineering.

[00:58:39] JM: All right. Well, we're up against time, but I just want to ask one more question. We've done a few shows about these different workflow schedulers. We've done a show on – A couple of shows on Airflow. We've done a show on Dagster. We recently did a show on Cadence, which I think is in kind of a different category, because it's more long-lived workflows rather than focused on the data workflows. Tell me about your prediction for how this market unfolds. Do you think it shifts towards a winner-take-all or do you think it shifts in a place where there are domain-specific scheduling systems?

[00:59:25] JL: That is if fabulous question and an easy one to be wrong on. I do not think that domain- specific systems can emerge at this time because I think that the environment we're in the still to nascent even year-by-year. The use cases that data science, even the infrastructure that we might talk about. If we just sat down in December of each of the last five years, forget 10 years, and talked about the dominant pieces of software and infrastructure in the data world, that list would be changing over and over and over. Some things would be constant. Many things would be changing and we'd see a lot of movement in and out of that list. I think we're some ways off from seeing use-case-specific schedulers simply because of where they run, how they run and what their use case is. It's still very much in flux.

I think what we're looking for is we're looking for the next Airflow in the sense that it's general enough. It doesn't – So many assumptions that it prevents certain use cases, but it's general enough to allow a great flexibility of data applications. Whether or not that's a winner-take-all is like another layer of complexity for some time until – I don't know, from my point of view, until a white flag was raised, Airflow is not in a winner-take-all situation. The majority of Airflow's life, it went side-by-side with Luigi, and there was a brief attempt by a tool called Pinball, which was

made by Pinterest, [inaudible 01:00:48]. Airflow is just a fascinating case study in market dynamics.

I don't think Airflow is one because it's the best tool. I don't even think it's the best general purpose tool. I think it's one in the absence of a viable competitor. What we're watching him play out is the strength of a defensible first mover position with a network effect played over it through the open source community.

If you try to attack Airflow 's position purely on doing more or new use cases, it's not enough. It's really not. Airflow is in a very powerful place if all you do is consider the number of people using it. That's why recently, you mentioned you're going to talk to the Astronomer folks. They put out an article, a blog post on why you should use Airflow, and it's called Why Airflow, of course, and it's only argument, is you should use Airflow because other people use it.

The truth is I don't think that's why any of us got into writing software. We don't write software because we like doing what other people do. We like writing software because we like beautiful abstractions that make our lives better, and Airflow is reaching a point as data science and now data engineering advance where I don't think it is achieving that. When you talk to people about Airflow, you don't often hear how much they love it. You year how much they tolerate it, and that's the opportunity to introduce a new set of abstractions that people genuinely love to use and will adapt.

I feel somewhat strongly that, look, there'll always be a place for a use case specific thing. If you're in a hyper specific niche that has extreme requirements, yes, by all means they'll be something for you. But to unseat Airflow, it's going to take a really powerful, useful and applicable vocabulary that is general enough to be used by such a wide audience. I think that's what we're going to see play out in the next few years.

[01:02:32] JM: Okay. Jeremiah, thanks for coming on the show. It's been great talking.

[01:02:36] JL: Absolutely! Thanks for having me, Jeff.

[END OF INTERVIEW]

[01:02:45] JM: As a company grows, the software infrastructure becomes a large complex distributed system. Without standardized applications or security policies, it can become difficult to oversee all the vulnerabilities that might exist across all of your physical machines, virtual machines, containers and cloud services. ExtraHop is a cloud-native security company that detects threats across your hybrid infrastructure. ExtraHop has vulnerability detection running up and down your networking stock from L2 to L7 and it helps you spot, investigate and respond to anomalous behavior using more than 100 machine learning models.

At extrahop.com/cloud, you can learn about how ExtraHop delivers cloud-native network detection and response. ExtraHop will help you find misconfigurations and blind spots in your infrastructure and stay in compliance. Understand your identity and access management payloads to look for credential harvesting and brute force attacks and automate the security settings of your cloud provider integrations. Visit extrahop.com/cloud to find out how ExtraHop can help you secure your enterprise.

Thank you to ExtraHop for being a sponsor of Software Engineering Daily, if you want to check out ExtraHop and support the show, go to extrahop.com/cloud.

[END]