# Durable, scalable EL(T) with MIT OSS Airbyte and Temporal

This case study is based on an interview with Abhi Vaidyanatha and Jared Rhizor who were key players in Temporal adoption within Airbyte

### Context

Data warehouses were originally designed around ETL (extract transform load). With ETL, data is extracted from a source and then transformed to meet the schema of the destination before being loaded (schema on write) but it also adds immense overhead of transforming data before it can be stored. For most of computing history, disk space and cycles were expensive, which made ETL a great choice.

Storage and compute costs have dropped dramatically in the last 20 years, which in turn means the primary argument for ETL is far less relevant. This led to the rise in popularity of an alternative approach known as ELT (extract load transform). Transformation is still required but it becomes the responsibility of the data lake end user (schema on read). During the ETL era, most relevant solutions aimed to address the entire problem even though transforming is a much different problem than extracting and loading. With ELT, the equation changed and extracting and loading are far more isolated from transformations. This opened up the potential for systems which only concern themselves with extracting and loading, and left transformation to a dedicated solution.

## Problem

Airbyte is an MIT open source EL(T) solution which makes it possible to migrate data from a source to practically any destination type. They support thousands of connectors making it painless to move data between completely different distributed data sources (ie: Salesforce and Snowflake). Doing this at scale is no easy task, as it requires managing thousands of concurrent extraction and load processes. If any of those processes failed, it could result in inconsistent or incomplete data.

## How was this problem solved before?

Airbyte solved this problem by building a custom orchestration solution with the initial scope of scheduling and running jobs.

> Our original solution was developed completely in house with the scope of running jobs on a regular interval. **It was very hacky** and did not come with a great set of guarantees like Temporal.

The in-house job system worked well enough initially but as the product and team continued to scale some serious problems emerged. Since the original scope of the in-house system had been narrow, teams adding new functionality did not feel they had a framework to contribute holistic improvements. This resulted in **each team implementing their own narrow domain specific solution** within the in-house system. Before long each and every team was reimplementing basic functionality like retries, timeouts, CRON, etc...

> One of the main problems we were facing was the feature set. Using a single feature across multiple areas of the system often required implementing it multiple times. This was acceptable for the MVP, but once we started adding in retries, timeouts, CRON it started to become a real challenge.. **These things are just easier to do with a framework that takes care of all that functionality**.

> With Temporal we started chaining things together in sequence. Before the migration, single jobs were running out of time. Now our jobs have several steps each with retryable behavior. **Temporal seemed like a much better fit than rebuilding Temporal in-house.**

## Change is needed

It was obvious to the Airbyte team that they were rapidly outgrowing their in-house solution. A search started for the next generation solution and every modern workflow engine was on the table. Eventually the list was whittled down and Temporal was evaluated against a few top contenders:

- Daemon lib
- Workflow2
- Airflow

> We have a lot of synchronous calls where we need to execute some process and get the result. For a case like that, Airflow isn't really appropriate.

After analysis, it was clear that the alternative solutions each solved only a sliver of the problem. For example Airflow was relatively well suited for handling asynchronous monitoring jobs but fell apart for synchronous use cases. Workflow2 and Daemon lib arguably address all of Airbytes needs when used in conjunction, but required stitching together separate tools. Eventually it became clear that Temporal would be able to deliver all of the value Airbyte needed (and more) out of the box while also keeping the dependencies at a minimum.

### Migration

Migration was a serious consideration for the Airbyte team when evaluating their next generation solution. The original job system had become fragmented enough that it was unthinkable to make a “big bang” switch to alternative solution like Temporal.

> It involved a bit of a refactor. We played with the interfaces for different workflows to make sure that they would plug in well. It mostly came down to changing a lot of edge cases of how we pass around configs, and what exactly the inputs are for something to be valid as a Workflow or an Activity.

Instead of migrating the entire system at once, the team opted for **an incremental approach**. Temporal would run alongside the original system in a mixed mode, until the time where everything could be ported. At this point a majority of jobs are being done through Temporal but there are still a few miscellaneous pieces of work handled by the original solution.

> It came down to moving conflict persistence in some forms out of one model and into Temporal and slowly dialing our extraction back and using Temporal for more of the execution than we were before. It was gradual and there's still some stuff that we haven't done quite yet but will do in the future. And it should be a lot easier since we can start to delegate the actual scheduling to Temporal.

## Outcomes

Since switching to Temporal, the Airbyte team has seen huge improvements in their ability to identify critical issues and debug them.

> We haven't run into any scheduling related bugs. That is something that we encountered before. **Temporal exposed scenarios where we weren't addressing certain edge cases.** So I'd say that the stability of that has improved quite a bit. Engineering usability and stability has definitely improved.

> Temporal makes it much easier to **extend our functionality**.

> We got Temporal up and running and it kind of just... **stayed out of the way**. It seamlessly integrated into the Airbyte architecture and **always delivers** on the core functionality it provides. It made us much more confident about developing and operating the product.

## How do you feel about your decision post migration?

> Positive... There's always a worry when you pick something up that there will be a bunch of failures that come through that are impossible to understand. And it's just inherent in the tool that you're using and you basically have to become an expert on that tool. We haven't run into that with Temporal at all. We're in a situation where we have many different people trying Airbyte out, each on their own systems and they're operating under wildly different conditions usage scenarios. **The absence of pain has been really positive.**

> Things are really well documented and easy to work through, some things are a little bit harder. But overall, it's been a very positive experience. For the basics, it was **very easy to adopt and pick up.**

> It was **surprisingly easy** to get Temporal up and running for synchronous processes.

> Since migrating to Temporal, it's just been great to see the **reduction in community reported scheduling problems**.