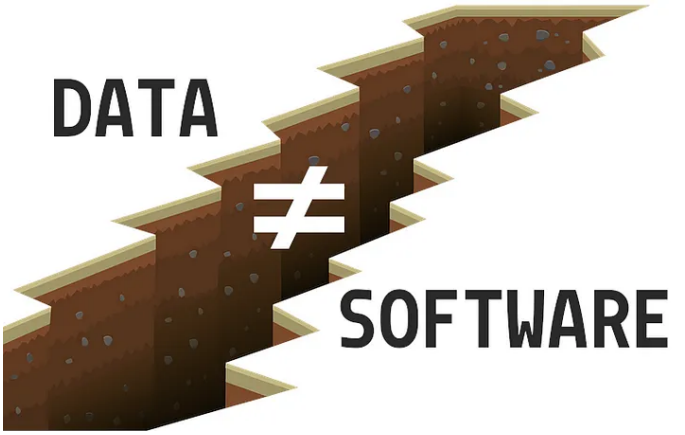
Data Engineering is Not Software Engineering

Pretending like data and software are the same is counterproductive to the success of your data engineers

Niels Cautaerts · Follow
Published in Better Programming · 14 min read · Nov 10, 2022



In recent years, it would appear that data engineering is converging with DevOps. Both have embraced cloud infrastructure, containerization, CI/CD, and GitOps to deliver reliable digital products to their customers. The convergence on a subset of tooling has led many to the opinion that there is no significant distinction between data engineering and software engineering. Consequently, the fact that data engineering is quite “rough around the edges” is simply because data engineers lag behind in the adoption of good software development practices.

This assessment is misguided. Data engineering and software engineering share many common tools and practices, but they also differ substantially in a number of key areas. Ignoring these differences and managing a data engineering team like a software product team is a mistake. Think of data engineering like a tomato: it is a fruit but that doesn’t mean it should be added to a fruit salad. This post aims to highlight some of the unique challenges in data engineering, and why this requires a custom approach.

Data Pipelines Are Not Applications

Software engineering tends to be concerned with building applications. In the context of this post an application is defined very broadly and could be a website, a desktop application, an API, a monolithic mainframe application, a game, a microservice, a library, or anything in between. The characteristics these all share are that they:

- Provide value to users by offering a new interaction modality. A game can be played, a website can be browsed, an API can be utilized in other software.
- Have a number of mostly independent features. A website can grow its number of pages, a game can grow in the number of levels or playable characters, an API can add more endpoints. As a corollary, an application is never truly complete.
- Deal with a relatively small amount of state that they create. State is designed to support the application. The management of this state is often offloaded to an external system. The goal is for large parts of the software to be stateless. A web application can be killed and restarted any time; its state is managed by a database that runs in a separate process.
- Are loosely coupled with other software and services. Good software should function independently in any environment, hence the popularity of microservices and containers.

Data engineers are concerned with building data pipelines. Data pipelines take data from the place where it is produced, transform it, and put it in another place from where it is consumed. Usually the goal is to automate these pipelines on a schedule so that datasets are updated over time with new data. Like applications, data pipelines are usually pieces of software. Contrary to applications, a data pipeline:

- Does not provide any direct value. There are no users of the pipeline. Only the dataset that is produced by the pipeline is valuable to downstream consumers. If the data would get to its destination via some elaborate copy-pasting scheme, the customer would be equally satisfied.
- Has only a single feature of relevance to the customer: produce the dataset that was requested. Therefore, there is a clear point of completion, though a pipeline will require continuous maintenance due to changes in upstream systems user requirements.

- Manages a large amount of states. A pipeline is designed to process existing state from other software it does not control, and convert it to state it does control. Many pipelines build datasets incrementally, adding more data on every run. In this sense, these pipelines could be viewed as very long-running processes that continuously create more and more states.
- Has unavoidable tight coupling. The goal of a data pipeline is precisely to bind to a data source. The pipeline will only ever be as stable and reliable as the source.

These fundamental differences lead to unique challenges for data engineering that are often poorly understood by business, IT management, and even software engineers. Let's go through them.

A Pipeline Is Either Completed or Worthless

Many organizations these days manage their software teams under some flavor of Agile. The core philosophy of these frameworks is to maximize the speed at which the software delivers value to the customer by building and release software in short iterations. This should deliver a minimum viable product (MVP) as fast as possible and ensure fast feedback loops, thereby ensuring the team is always working on the features with the highest priority.

These ideas do not map onto data engineering.

A data pipeline can not be developed in small iterations of increasing customer value. There is no MVP equivalent in a data pipeline. It produces the desired dataset for the customer or it doesn't.

Consequently, data pipeline development does not fit neatly into agile frameworks. A complex data pipeline corresponds to a single user story, but usually requires multiple sprints to be completed. Non-technical management rarely takes this subtle point into consideration and tries to shoehorn data engineers into scrum teams anyway. The result is that user stories are replaced with tasks, e.g. "build API connector" and "build ingestion logic", which inevitably turns the scrum board into a device of micromanagement.

When management does not understand the fundamentals of what they manage, they tend to make poor and unworkable decisions. Frustrated by the slow progress on a pipeline, a manager once demanded from a data engineer on our team that he build up the dataset iteratively, column by column so that the customer "at least had some data to already start working with". Data engineers with practical experience in complex pipelines and data scientists who've ever received a useless dateset will recognize the ridiculousness of this situation. For the readers without this background, here are three reasons why this doesn't work:

1. A partial dataset does not have proportional utility

If a dataset contains 9 out of 10 columns, is it 90% useful? It depends on which column is omitted. If a data scientist aims to build a predictive model based on the data, but the column that is missing is the labels or values they want to predict, then the dataset is 0% useful. If the column is some random metadata uncorrelated with the labels, it may be 100% useful. Most commonly, a column represents a field that may or may not be correlated with the labels; finding out if it is correlated is exactly what the data scientist wants to find out through experimentation. That's why a data scientist wants an as complete as possible dataset to begin experimenting with, explore, and gradually optimize their model. Providing them with a partial dataset ensures that experimentation and optimization will need to be redone once additional fields become available.

2. Time to develop a pipeline is not correlated with dataset size

Even if a customer would be happy with half a dataset, producing this would not take half the time it takes for the full dataset. A data pipeline does not consist of independent jobs that each produce a column. Multiple columns may be related if they originate from the same source, so including one or multiple in the final dataset is the same amount of work. However, the logic to merge these columns with columns in another table could be as simple as a single join or may require a convoluted series of window functions. Additionally, a lot of boilerplate may need to be written in a data pipeline before any data comes out at the other end, for example, a client to access an API or a parser to process nonstructured data. Once this is in place, expanding the logic to process additional fields is probably trivial. Hence, the number of columns in the final dataset is a terrible metric for the pipeline complexity, on par with using number of lines of code to measure productivity.

Dataset size could also refer to the number of rows/records. A well-designed pipeline should be able to handle any number of records, so development time should be entirely decoupled. There may however be "jumps" in development time depending on specific requirements like:

- how often does the dataset need to be updated, i.e. do we need batch or streaming?
- what are the expected data volumes and velocity?
- does the data fit into RAM?

These considerations should be known a priori and will influence the entire design of the pipeline.

3. Time and economic cost to build a dataset correlates with its size

The larger a dataset is, both in terms of rows and columns, the longer it takes to build and update it. Editing a single record in a huge database is trivial and fast, but this is an uncommon scenario for analytical datasets. Modifying an analytical dataset typically involves adding/changing entire columns (which changes all records) or updating thousands or millions of rows. There are two ways to deal with change to data, and neither is cheap.

From a developer's perspective, the easiest way to update a dataset is to overwrite everything by updating and re-running the pipeline. However, this is the most expensive in terms of computing costs and time to refresh the dataset. Designing a pipeline such that it correctly overwrites the state from previous runs (idempotency) is also not always trivial and requires proper up-front planning.

Alternatively, the logic for updating a dataset could be encoded in a separate pipeline that takes the old dataset as input. This can be more economical in terms of computing cost and speed but incurs additional development time and mental overhead. Pipelines that apply deltas are not idempotent, so it's important to keep track of the current state as well as when specific operations were performed. Even in this second scenario, the old pipeline should be updated to include the desired change in new updates.

Anyway, the problem is diced, datasets have an inherent inertia. Attempting to make changes requires more time, effort, and/or money the larger the dataset is.

Conclusion: deploying a partial pipeline to production is wasteful

Deploying a partially completed pipeline to production is not useful to the customer, wastes compute, and complicates life for the engineers building the pipeline as they will have to deal with old state. Cargo-culting DevOps and Agile principles over to data pipeline development, in which incremental change and frequent deployment are encouraged, simply ignores the inertia of data.

Engineers would like to "do it right the first time" and minimize the number of deployments to production. A pipeline that is frequently deployed signifies either that the customer doesn't know what they want, or that the data source is very unstable and the pipeline needs constant patching. Contrary to stateless applications, where updates tend to be as easy as killing a couple of containers and spinning up new ones, updating a dataset is not the same as redeploying pipeline code. Packaging pipeline code into a container and running it on Kubernetes won't bridge that gap.

Feedback loops in pipeline development are glacial

In order to quickly create new features or fix bugs in software, developers need fast feedback that tells them whatever they have written is correct and moves the software in the right direction.

In software development, this is typically achieved using a suite of unit tests, which the developer runs locally to check whether each component of the software (still) works as intended. Unit tests should be fast, not interact with any external systems nor depend on any state. They should test functions, methods and classes independently and in isolation. In this way, a software developer gets rapid feedback during development and can be pretty confident their code works as intended when they open a pull request. If there is a need to test interaction with other systems, a CI pipeline may also run slower integration tests.

Here's a data engineering secret: data pipelines are rarely unit tested (gasp!). Data pipelines are usually tested by simply deploying them — usually first to a development environment. This requires a build and deploy step, after which the engineer must monitor the pipeline for a while to see if it works as intended. If the pipeline is not idempotent, a redeploy may first require manual intervention to reset the state left behind by the previous deployment. This feedback cycle is very slow compared to running unit tests.

So why not just write unit tests?

1. Data pipelines fail on aspects that can not be unit tested

Self-contained logic that can be unit tested is usually limited in data pipelines. Most of the code is *glue and duct tape*. Therefore nearly all failures occur at faulty interfaces between systems or unexpected data entering the pipeline.

Interfaces between systems can not be tested with unit tests, because these tests can not run in isolation. The external systems could be mocked, but this would only prove that the pipeline works with a system that behaves like the data engineer *thinks* the external system works. In reality, the data engineer rarely knows all the relevant details. Let's take a real world example: to prevent DDOS attacks, a public API may have an undisclosed limit on the number of requests that can be sent from the same IP within a certain time interval. A mock of the API would likely not account for this, but the fact that it exists in the real system may break the pipeline in production. Additionally, external systems are rarely stable. In fact, data pipelines are usually built *because* people want to move data from dodgy systems into more reliable ones. A mock would not reveal whether the real system changed in a way that breaks the pipeline.

Data producers are notoriously bad at handing over quality data consistently. A data pipeline must always make some assumptions about the data that will be fed into it. Unexpected content or structure in the data will break the pipeline, or at the very least produce an incorrect result. A common strategy to defend the pipeline against sloppy data sources is by verifying schemas on read. But this does not guard against erroneous content of the data and subtle "data bugs". For example, does a time series deal correctly with day lights savings time? Are there any strings in a column that do not adhere to an expected pattern? Do values in a numerical column, representing real world measurements, make physical sense? None of this is related to pipeline logic that can be tested with unit tests.

2. Unit tests are more elaborate than the pipeline logic

The unit tests required to test the limited self-contained transformation logic in a pipeline are more complex than the code itself, because it requires the developer to create *representative* test data, as well as the expected output data. This is a lot of work that does not substantially improve confidence in the correct functioning of the pipeline. Additionally, this shifts the question from "does this function work as intended?" to "does this test data represent my real data adequately?". Unit tests ideally cover a good subset of input parameter combinations. But in functions that transform a dataset, e.g. in the form of a dataframe, the dataset argument itself presents a near infinite dimensional parameter space.

Conclusion: developing a pipeline is slow

The best way to get reliable feedback on a data pipeline is to deploy and run it. This will always be slower than running unit tests locally, meaning it takes much longer to get feedback. The result is that pipeline development, especially in the debugging stage, is annoyingly slow.

Integration tests that are faster than running the entire pipeline could be considered. However, these can usually not be run on a developer's machine, since it doesn't have direct access to the relevant source systems. Hence, these tests can only be run in the same environment as the pipeline, which again requires a deployment. This largely defeats the point of writing tests for getting fast feedback.

"Data contracts" are now all the rage for dealing with reckless data producers. Being confident in the data that enters the pipeline would take a lot of uncertainty out of pipeline development and make them less brittle. However, it seems these contracts are difficult to enforce as producers are not incentivized to adhere to them. Additionally, an organisation will want to use data pulled from external sources, like public APIs; good luck negotiating a data contract with these external parties.

Pipeline Development Can Not Be Parallelized

We've established that data pipelines are a single user story that are slow to develop due to the long feedback cycle. But since a pipeline is composed of multiple tasks, some managers try to distribute them among multiple developers in an attempt to speed up the process. Unfortunately, this does not work. The tasks to process the data in a pipeline are sequential. In order to build the second step, the output of the first step must be stable. Insights gained by building the second step feed back into improvements on the first step. As such, the pipeline as a whole must be considered as a feature on which a developer iterates.

Some managers counter that this simply means the pipeline is not sufficiently planned out from the beginning. There is data at the start, and it is clear what data needs to come out at the end. Is it then not obvious what needs to be built in the middle? Paradoxically, the same managers making this argument will vehemently defend the merits of Agile.

Planning out the entire pipeline doesn't work as long as the data source is not properly characterized. In the absence of contracts and documentation, the data engineer has to stumble around in the dark in order to discover the particularities of the data. This discovery process shapes the architecture of the pipeline. In a sense this is agile; it's just not how business stakeholders want it to work.

Conclusion and Recommendation

Data pipelines are software but not software products. They are the factory that serves to build the car the customer actually requested. They are a means to an end, an automation recipe for creating an easily consumable dataset from unwieldy data sources. They are the duct tape between systems that were not designed to talk to each other. They are ugly, brittle, and expensive solutions to data's "last mile" problem. Their sole purpose is to manage state, which makes their development slow, unpredictable, and frequently the main bottleneck to data analytics projects. No matter what data influencers claim by shilling yet another tool, no matter how many layers of abstraction are stacked on top of each other, data is fundamentally different from software. Not recognizing these differences and enforcing agile processes in a data team because it worked in the software team will only backfire.

What can you do to make a data team successful and productive?

- Recognize that a lite form of Waterfall (usually synonymous with *bad* in software engineering) is inevitably what happens in data pipeline projects. Before any development can begin, a lot of conversations need to be held with customers to clarify requirements on the desired dataset, as well as with data producers to transfer knowledge on undocumented APIs and connect to the source system. Don't invest a lot of time in building before a proposed solution is agreed upon by both the customer and the data producers. Recognize that change will be costly once the pipeline is shipped off to production.
- Allow the data engineers some time to experiment with the data sources. Recognize that all time estimates regarding when the dataset will be available will be wrong.
- Don't split a pipeline among multiple developers. Instead, allow two or multiple developers to collaborate simultaneously on a pipeline. Pair/extreme/group programming with trunk-based development ensures maximum productivity, as it avoids git branching hell, pull requests, and code reviews. Having four eyes inspecting the code at all times helps in spotting issues early. This is especially valuable in pipelines where feedback loops are slow.